A Geometric Approach to Support Vector Classification

by

Ben Goodrich, BSc BCompSc (Hons)

Thesis

Submitted by Ben Goodrich

for fulfillment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisors: Dr. David Albrecht

Dr. Peter Tischer

Clayton School of Information Technology

Monash University

December, 2012

Errata

p15, para 1	replace "$(x_1y_1 + x_2y_2)$" with "$(x_1y_1 + x_2y_2)^2$"
p16, para 3	replace "$C = 1/\lambda n$" with "$C = 1/(2\lambda n)$"
p18, para 3	replace "Recall that the supporting planes" with "Recall from Section 2.2.1 that the supporting planes"
p19, fig 2.7	add to caption: "The top line shows the possible values of Lagrange multipliers for the negative class. The bottom line shows the possible values of Lagrange multipliers for the positive class"
p20, final para	replace "Because of the way SVMs incorporate large margins, they are already in some respects a geometrically intuitive classifier" with "SVMs are already in some respects a geometrically intuitive classifier due to the way in which they incorporate large margins"
p46, final para	replace "there are" with "there is"
p47, para 4	replace "separate $\lceil 1/\mu \rceil$ points from the set" with "separate $\lceil 1/\mu \rceil$ points from the rest of the set"
p47, fig 3.5	in caption, replace "separates points $\mathbf{x}_a, \mathbf{x}_b\mathbf{x}_c \in P$ which become" with "separates points $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c \in P$ from the rest of P. Accordingly, $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c$ become"
p52, algo 6	replace "visible from $\mathbf{v}$" with "visible from $\mathbf{h}$"
p57, para 2	replace "any plane" with "any hyperplane"
p58, final para	replace first sentence with "It is also informative to observe how the number of facets for an RCH with μ in the range $(\lfloor 1/k \rfloor, \lceil 1/k \rceil)$ remains constant (Figure 3.12)"
p61, algo 7	add "$s \leftarrow 0$" directly after "$\alpha \leftarrow 0$"
p78-86	replace "accuracy" with "error", so that "higher accuracy" becomes "lower error", and "lower accuracy" or "degraded accuracy" becomes "higher error"
p79, final para	after "The lack of a ρ parameter" add footnote: "Refer to Section 2.5.1 for the definition of ρ, and Section 4.3.1 for an explanation of how ρ impacts the margin"

p118, fig 5.9	add to caption: "Points on these graphs show how the two-class algorithm performed relative to the one-class algorithm. For example, a value of $(x, y) = (200, 1.2)$ indicates that, for $\mu = 1/200$, the two-class approach took 20% longer, or required 20% more iterations, than the one-class approach. The dark line along $y = 1$ provides a basis for comparison since a ratio of 1 indicates equality between the two approaches."
p120, first eq	move "$=\lambda'_{top}/\lambda'_{bot}$" to (5.27)
p148, para 1	immediately after (6.2), add "Here ξ_i's are slack variables, computed by solving the C-SVM optimization task"
p153, para 2	immediately after equation beginning "$(R^{low})^2 = ...$", add "Here β_i's are the Lagrange multipliers associated with the MEB dual optimization task. We denote these as β_i to avoid confusion with the SVM Lagrange multipliers, which remain α_i."
p158, para 2	replace "fixed stopping conditions" with "relative KKT stopping conditions (which we described in detail in Section 5.8)"
p160, para 2	replace paragraph with "For the following empirical trials we compare our method to Cristianini's method. We do not compare to standard radius-margin parameter selection since standard radius-margin parameter selection is outperformed so significantly by Cristianini's method. Comparing our technique to Cristianini's is also a much fairer comparison since both methods use previous solutions to seed new solutions. We perform comparisons using both tight ($\epsilon = 10^{-3}$) and loose ($\epsilon = 10^{-1}$) stopping conditions for Cristianini's method in order to verify that our improvements can not be achieved by simply having looser, fixed, stopping conditions."
p161, para 3	replace "fixed stopping conditions of" with "relative KKT stopping conditions with"
p173, para 1	replace "framework for SVMs provides a means of" with "framework for SVMs provides a conceptual tool for"
p181, para 1	replace "largest class. This is the accuracy" with "smaller class. This is the error rate"
p181, table A.1	divide last column by 100, subtract result from one and remove percentage signs
p182-185	in captions for tables B.1-B.5, replace "Accuracy for" with "Error rate for"
p197-205	in captions for figures C.10-C.18 replace "Accuracies for" with "Error rates for"

Copyright Notices

For Anne.

Contents

List of Tables . . . . . **ix**

List of Figures . . . . . **xi**

List of Algorithms . . . . . **xiv**

Nomenclature . . . . . **xv**

Abstract . . . . . **xvii**

Acknowledgments . . . . . **xix**

1 Introduction **1**

1.1 Thesis Overview and Contributions 3

1.2 Relationship with Existing Work 7

2 Background **9**

2.1 Introduction 9

2.2 Hard Margin SVMs 10

2.2.1 The Hard Margin Dual 11

2.2.2 The Importance of Large Margins 13

2.3 Using Kernels for Non-Linear Classification 14

2.4 Soft Margin SVMs 15

2.4.1 L_1-loss SVMs 17

2.4.2 L_2-loss SVMs 19

2.5 Geometric Interpretations of SVMs 21

2.5.1 μ-SVMs: Reparameterizing the L_1-loss Primal 21

2.5.2 A Geometric Interpretation of μ-SVMs 23

2.5.3 Reparameterizing the L_2-loss Primal 25

2.5.4 A Geometric Interpretation of L_2-loss SVMs 26

2.6 Weighted SVMs 26

2.6.1 Class Weighting 27

2.6.2 Individual Point Weighting 28

2.6.3 Alternative Loss Functions 29

2.7 Minimal Enclosing Balls (MEBs) 29

2.7.1 The MEB Dual . . . 30
2.7.2 Soft MEBs . . . 30
2.8 Perceptrons . . . 31
2.8.1 Rosenblatt's Perceptron . . . 32
2.8.2 AdaTron Algorithm . . . 32
2.8.3 Perceptrons as a Quadratic Programming Task . . . 33
2.8.4 Perceptrons with Soft Margins . . . 34
2.8.5 Perceptrons with a Bias Term . . . 34
2.8.6 Distinguishing Perceptrons from SVMs . . . 35

3 Reduced Convex Hulls . . . 39
3.1 Introduction . . . 39
3.2 Background . . . 39
3.2.1 Convex Hulls . . . 40
3.2.2 Algorithms for Computing Convex Hulls . . . 41
3.3 Reduced Convex Hulls . . . 45
3.4 Finding Vertices and Support Points . . . 46
3.5 Algorithms for Computing RCHs . . . 48
3.5.1 Representing an RCH . . . 48
3.5.2 Hyperplanes Passing Through d Points in $\mathbb{R}^d$ . . . 49
3.5.3 Points in the Plane . . . 50
3.5.4 Points in Arbitrary Dimensional Space . . . 51
3.6 Related Geometric Concepts . . . 53
3.6.1 k-sets . . . 53
3.6.2 k-set Polytopes . . . 54
3.6.3 Centroid Polytopes . . . 54
3.7 Properties of RCHs . . . 54
3.7.1 How the Reduction Works . . . 55
3.7.2 Number of Vertices . . . 55
3.7.3 Number of Facets . . . 56
3.7.4 Symmetry . . . 57
3.8 Computational Complexity of RCH Algorithms . . . 59
3.8.1 Points in the Plane . . . 59
3.8.2 Points in Arbitrary Dimensional Space . . . 60
3.9 Weighted Reduced Convex Hulls . . . 60
3.9.1 Definition . . . 60
3.9.2 Relationship with Point Duplication . . . 61
3.9.3 Finding Vertices and Support Points . . . 61
3.9.4 Adapting RCH Algorithms to Support Weights . . . 62
3.10 Conclusions . . . 63

4 Understanding SVMs from a Geometric Perspective 65
4.1 Introduction 65
4.2 The Geometric Framework 66
4.2.1 Why Use Reduced Convex Hulls for Learning? 66
4.2.2 Significance of the Centroids 67
4.2.3 Impact of the Kernel 68
4.2.4 Relationship with k-Means Classifiers 70
4.2.5 Relationship with k-Nearest Neighbor Classifiers 70
4.3 The Threshold 72
4.3.1 The KKT Threshold 73
4.3.2 The Geometric Threshold 75
4.3.3 The Probabilistic Threshold 76
4.3.4 Empirical Trials 77
4.3.5 Bad Thresholds 79
4.4 Incorporating Weights into the Geometric Framework 81
4.4.1 A Geometric Interpretation of WSVMs 81
4.4.2 Minimizing the Overall Error 83
4.4.3 Minimizing the Average Proportion of Errors 84
4.4.4 Minimizing the Total Cost of Error 85
4.4.5 An Empirical Demonstration 85
4.4.6 Choosing the Threshold for WSVMs 87
4.5 Perceptrons under the Geometric Framework 88
4.5.1 A Geometric Interpretation of Perceptrons 89
4.5.2 Perceptrons with Weighted Training Data 91
4.6 Conclusions 91

5 Geometric Training Algorithms 93
5.1 Introduction 93
5.2 Nearest Point Algorithms 95
5.2.1 The Schlesinger-Kozinec Algorithm 95
5.2.2 Gilbert's Algorithm 100
5.2.3 The Mitchell-Dem'yanov-Malozemov Algorithm 103
5.2.4 Terminating the Algorithm 105
5.2.5 Choosing the Threshold 106
5.3 Sequential Minimal Optimization 106
5.3.1 Updating a Pair of Lagrange Multipliers 107
5.3.2 Choosing the Lagrange Multipliers to Update 108
5.3.3 Relationship with Nearest Point Algorithms 108
5.4 Improving Nearest Point Algorithms 108
5.5 A Weighted Schlesinger-Kozinec (WSK) Algorithm 109
5.5.1 The WSK Algorithm 110
5.5.2 Comparing WSK to Point Duplication 111
5.5.3 How Weighting Affects Training Time 114

5.6 An Efficient WSK Implementation . . . 116
5.6.1 One-class vs Two-class . . . 116
5.6.2 Maintaining a Cache . . . 118
5.6.3 Computing the Update Step . . . 120
5.6.4 Reaching the Outside of the Hull . . . 122
5.7 Comparing Nearest Point Implementations . . . 124
5.7.1 Comparison as Parameters Change . . . 124
5.7.2 Comparison as Training Set Sizes Change . . . 125
5.7.3 Discussion . . . 126
5.8 Impact of the Stopping Conditions . . . 128
5.8.1 Types of Stopping Conditions . . . 128
5.8.2 Comparing Stopping Conditions . . . 130
5.8.3 Discussion . . . 134
5.9 Training Weighted Perceptrons using the WSK Algorithm . . . 134
5.9.1 The Perceptron Dual Optimization Task . . . 135
5.9.2 The One-Class WSK Algorithm . . . 135
5.9.3 The Threshold . . . 135
5.9.4 Accuracy and Efficiency . . . 136
5.10 Discussion . . . 142
5.11 Conclusions . . . 143

6 Parameter Selection using Geometric Information . . . 145
6.1 Introduction . . . 145
6.2 Existing Error Estimates . . . 146
6.2.1 Hold-out Sets . . . 146
6.2.2 Cross-Validation and Leave-One-Out . . . 146
6.2.3 Support Vector Bound . . . 147
6.2.4 Generalized Approximate Cross-Validation . . . 148
6.2.5 Xi-Alpha Error Estimate . . . 149
6.2.6 Radius-Margin Ratio . . . 149
6.3 Accelerating Radius-Margin Parameter Selection . . . 151
6.3.1 The Geometric L_2-loss SVM . . . 152
6.3.2 Bounding the Margin During Training . . . 152
6.3.3 Bounding the Radius of the MEB During Training . . . 153
6.3.4 Bounding the Radius-Margin Ratio During Training . . . 154
6.3.5 Efficient Computation of Bounds During Training . . . 154
6.3.6 Efficiently Comparing SVMs in Terms of Radius-Margin Ratio . . . 155
6.3.7 Efficiently Minimizing the Radius-Margin Ratio . . . 157
6.3.8 Empirical Trials . . . 157
6.3.9 Discussion . . . 165
6.4 Parameter Selection using μ-SVMs . . . 165
6.4.1 C-SVM Parameter Search . . . 166
6.4.2 μ-SVM Parameter Search . . . 167

6.4.3 Error Estimates for μ-SVMs . . . 169
6.4.4 Empirical Trials . . . 170
6.4.5 Discussion . . . 172
6.5 Conclusions . . . 172

7 Conclusions . . . 175
7.1 Reduced Convex Hulls . . . 175
7.2 Understanding SVMs from a Geometric Perspective . . . 175
7.3 Geometric Training Algorithms . . . 176
7.4 Parameter Selection using Geometric Information . . . 177
7.5 Discussion . . . 177
7.6 Future Work . . . 179
7.7 Closing Remarks . . . 180

Appendix A Datasets used in Empirical Tests . . . 183

Appendix B Extensive Results for Thresholds . . . 185

Appendix C Extensive Results for Nearest Point Algorithms . . . 189
C.1 Training Times . . . 189
C.2 Error Rates . . . 199

Vita . . . 209

References . . . 211

Index . . . 221

List of Tables

2.1 Some commonly used SVM kernels 15
2.2 Alternative Perceptron Loss Functions. Note that a 'B' under type refers to bias-free (i.e. no bias term b is used) 36
2.3 Alternative SVM Loss Functions. All machines have a bias term. 36

4.1 Error for Gaussian SVMs ($\gamma = 0.01$) combined with three possible thresholds 78
4.2 Error for Gaussian SVMs ($\gamma = 0.1$) combined with three possible thresholds 79
4.3 Error for linear SVMs combined with three possible thresholds 80
4.4 Error for Gaussian SVMs, with $\gamma = 0.01$ and $\mu = 1/(0.9\kappa)$, combined with three different weighting strategies. The geometric threshold is used. 87
4.5 Error for Gaussian SVMs, with $\gamma = 0.01$ and $\mu = 1/(0.9\kappa)$, combined with three different weighting strategies. The KKT threshold is used. 88

5.1 Various configurations of the general WRCH SVM 109
5.2 Results of the three algorithms on the `heart` dataset 113
5.3 Results of the three algorithms on the `german` dataset 113
5.4 Results of the three algorithms on the `synthetic` dataset 114
5.5 Test error associated with each stopping condition for the Gaussian kernel with $\gamma = 0.1$ and large μ 131
5.6 Number of iterations associated with each stopping condition for the Gaussian kernel with $\gamma = 0.1$ and large μ 131
5.7 Test error associated with each stopping condition for the polynomial kernel with $q = 4$ and small μ 132
5.8 Number of iterations associated with each stopping condition for the polynomial kernel with $q = 4$ and small μ 132
5.9 Test error for Gaussian perceptrons ($\gamma = 0.01$) combined with four possible thresholds 137
5.10 Test error for polynomial perceptrons ($q = 3$) combined with four possible thresholds 138
5.11 Test error for SVMs compared to perceptrons using the geometric threshold 139

6.1 Average number of SMO iterations required for each parameter selection technique. The Gaussian kernel is used. 159

6.2 Average number of MEB iterations required for each parameter selection technique. The Gaussian kernel is used. . . . 161
6.3 Average test errors for each parameter selection technique. The Gaussian kernel is used. . . . 161
6.4 Average number of SMO iterations required for each parameter selection technique. The polynomial kernel is used. . . . 162
6.5 Average number of MEB iterations required for each parameter selection technique. The polynomial kernel is used. . . . 162
6.6 Average test errors for each parameter selection technique. The polynomial kernel is used. . . . 163
6.7 Test error achieved by combining two different error bounds 163
6.8 GACV parameter selection with μ and C-SVMs 170
6.9 XiAlpha parameter selection with μ and C-SVMs 171
6.10 Minimum test error across entire search 171
6.11 Number of SVMs trained 171

A.1 Summary of datasets used in empirical trials 183

B.1 Error rate for Gaussian SVMs ($\gamma = 0.01$) combined with three possible thresholds 185
B.2 Error rate for Gaussian SVMs ($\gamma = 0.1$) combined with three possible thresholds 186
B.3 Error rate for linear SVMs combined with three possible thresholds 186
B.4 Error rate for polynomial SVMs ($q = 3$) combined with three possible thresholds 187
B.5 Error rate for polynomial SVMs ($q = 5$) combined with three possible thresholds 187

List of Figures

1.1 An SVM separating the RCHs of the two classes 2

2.1 SVMs are a linear classifier with maximal margin Δ 9
2.2 Hyperplane, offset and margin of an SVM 11
2.3 Support vectors and supporting planes of a hard margin SVM 12
2.4 'Skinny' and 'fat' hyperplanes (adapted from Bennett and Campbell [8]) . . 13
2.5 The effect of the mapping in Equation 2.7 14
2.6 Slack variables 16
2.7 Support vectors and supporting planes of a C-SVM 19
2.8 SVMs as a convex hull problem 21
2.9 Reduced convex hulls in the plane 24
2.10 KKT conditions associated with the hard and soft MEB tasks 31

3.1 Convex hulls in two and three dimensions 40
3.2 Identifying whether a point is *below* a facet 41
3.3 Convex hulls have a simplified representation in two dimensions 44
3.4 Updating a convex hull using the Beneath-Beyond Theorem 45
3.5 Finding support points by 'pushing' a plane into a set of points 47
3.6 The hyperplane can not clearly separate three points in this case 48
3.7 2-sets of a set of points in $\mathbb{R}^2$ 53
3.8 Reduced convex hulls for $\mu = 1, \frac{1}{10}, \frac{1}{25}, \frac{1}{50}, \frac{1}{100}$. 55
3.9 Number of facets in the RCH of 25 uniformly distributed random points . . 57
3.10 A plane separating k points on one side and $n-k$ points on the other side 57
3.11 Transforming an RCH between two μ values 58
3.12 RCHs with $\mu \in (\lfloor 1/k \rfloor, \lceil 1/k \rceil)$ 59
3.13 The impact of the parameters on a WRCH 63

4.1 The geometric interpretation of an SVM 66
4.2 The threshold of an SVM 67
4.3 The significance of the centroids in SVM classification 67
4.4 A pair of inseparable RCHs 68
4.5 Gaussian C-SVMs with $C = 1$ while γ is varied 69
4.6 Polynomial C-SVMs with $C = 1$ while q is varied 69
4.7 A k-means classifier 70

4.8 An SVM is equivalent to a k-means classifier under certain conditions . . . 71
4.9 A 1-nearest neighbor classifier compared to a Gaussian SVM with very large γ 71
4.10 A sub-optimal threshold . . . 73
4.11 Choosing the threshold of an SVM using the KKT conditions . . . 74
4.12 A range of thresholds satisfying the KKT conditions . . . 75
4.13 The threshold can take any value such that $b < b'_{max}$ . . . 75
4.14 The geometric threshold compared to the KKT threshold . . . 76
4.15 Threshold as C decreases . . . 80
4.16 The `unbal` dataset . . . 82
4.17 Impact of point weighting on a WRCH . . . 83
4.18 Class reduction when an unweighted SVM is applied to unbalanced data . . 84
4.19 Reducing two RCHs relative to class size . . . 85
4.20 Impact of highly weighted points on a WSVM . . . 86
4.21 The threshold for unbalanced classes with all weights set to equal one . . . 88
4.22 The perceptron dual is equivalent to a minimal norm problem over an RCH 90

5.1 Computing an SVM using a nearest point algorithm . . . 95
5.2 Finding the nearest points using an iterative update step . . . 96
5.3 Updating the approximate nearest points using the S-K algorithm . . . 97
5.4 The point in a convex hull with minimal norm . . . 100
5.5 The MDM update step . . . 104
5.6 A polynomial weighted SVM trained on a `synthetic` dataset . . . 114
5.7 The relationship between weighting and training time. The Gaussian kernel is used with $\gamma = 0.01$ . . . 115
5.8 The relationship between margin and training time. The Gaussian kernel is used with $\gamma = 0.01$ . . . 115
5.9 Comparing the one and two-class nearest point approaches . . . 118
5.10 Choosing the most efficient update step . . . 121
5.11 The KKT conditions associated with the RCH nearest point problem . . . 123
5.12 Comparing the standard S-K algorithm to accelerated WSK on the `image` dataset. The Gaussian kernel with $\gamma = 0.01$ is used. . . . 124
5.13 Selected results for the four algorithms . . . 126
5.14 The results of the four algorithms on the `forest` dataset . . . 127
5.15 Test error as the number of iterations increases . . . 133
5.16 Training times for L_1-loss perceptrons compared to L_1-loss SVMs . . . 140
5.17 The relationship between test error and number of training iterations . . . 141
5.18 The hyperplane found by a bias-free perceptron is constrained to pass through the origin. . . . 142

6.1 The support vector bound on the `banana` dataset. The Gaussian kernel is used. . . . 148
6.2 The GACV error estimate compared to the support vector bound on the `banana` dataset. The Gaussian kernel is used. . . . 148

6.3 The Xi-Alpha bound compared to the support vector bound 149
6.4 The radius R of the MEB enclosing the training data and the margin Δ of an SVM 150
6.5 The L_1-loss SVM can arbitrarily increase the margin by varying μ. Here support vectors are circled. 151
6.6 Lower and upper bounds on the margin of a partially trained SVM 153
6.7 Lower and upper bounds on the MEB of the training data 154
6.8 Progress of Algorithm 12 for two SVMs and two MEBs on the `german` dataset 157
6.9 Adaptive method compared to standard radius-margin parameter selection 160
6.10 Several estimates of the leave-one-out error 164
6.11 Several estimates of the leave-one-out error 164
6.12 Test error and valid parameters for the Gaussian kernel. Regions where parameter values are invalid are shaded. 167
6.13 Values of $\log_2(C)$ and their equivalent values of μ on the `splice` dataset. The polynomial kernel with $q = 2$ is used 168
6.14 Searching through $1/\mu$ provides a much smoother transition in the hulls that searching through μ. 169
6.15 Test error for the Gaussian kernel. Shaded regions not searched. 169
6.16 A C-SVM with $C = 1$ trained on two toy datasets which are identical under scaling 172

7.1 Maximizing the margin between two classes with alternative geometric representations 180

C.1 Training times for the `banana` dataset 190
C.2 Training times for the `b.cancer` dataset 191
C.3 Training times for the `diabetes` dataset 192
C.4 Training times for the `german` dataset 193
C.5 Training times for the `heart` dataset 194
C.6 Training times for the `image` dataset 195
C.7 Training times for the `splice` dataset 196
C.8 Training times for the `thyroid` dataset 197
C.9 Training times for the `titanic` dataset 198
C.10 Error rates for the `banana` dataset 199
C.11 Error rates for the `b.cancer` dataset 200
C.12 Error rates for the `diabetes` dataset 201
C.13 Error rates for the `german` dataset 202
C.14 Error rates for the `heart` dataset 203
C.15 Error rates for the `image` dataset 204
C.16 Error rates for the `splice` dataset 205
C.17 Error rates for the `thyroid` dataset 206
C.18 Error rates for the `titanic` dataset 207

List of Algorithms

1 Rosenblatt's Perceptron [97] 32
2 The AdaTron Algorithm [3] 33
3 Quickhull algorithm 43
4 The Beneath-Beyond Algorithm 45
5 Quickhull algorithm for reduced convex hulls 51
6 General dimension reduced convex hull algorithm 52
7 Finding a Weighted RCH vertex 61
8 The Schlesinger-Kozinec Algorithm 98
9 Gilbert's Algorithm 101
10 The Weighted Schlesinger-Kozinec Algorithm 112
11 The One-Class WSK Algorithm 135
12 Comparing two SVMs (and two associated MEBs) and choosing the one which minimizes the radius-margin ratio 156
13 Searching across a grid of SVM parameters for those which minimize the radius-margin ratio 158

Nomenclature

α_i Lagrange multiplier associated with the SVM dual, page 11

Δ Margin of an SVM, page 9

δ_{ij} The Kronecker delta, $\delta_{ij} = 1$ if $i = j$, 0 otherwise, page 20

γ Width of the Gaussian RBF kernel, page 15

κ The sum of weight in the smaller training class, $\kappa = \min\left(\sum_{i \in I_{pos}} s_i, \sum_{i \in I_{neg}} s_i\right)$, page 86

λ Size of the update step for the S-K algorithm, page 96

$\phi(\mathbf{x}_i)$ An explicit feature map associated with a kernel, $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$, page 14

$\mathbf{w}$ Normal vector associated with the decision surface of an SVM, page 10

b Offset of a hyperplane from the origin, page 10

$f(\mathbf{x})$ The decision function of an SVM, $f(\mathbf{x}) = \sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} - b$, page 10

f_k The decision function applied to a training point, with threshold omitted, $f_k = \sum_i \alpha_i \mathbf{x}_i \cdot \mathbf{x}_k$, page 107

I_{neg} A set containing the indices of points belonging to the positive class, $I_{neg} = \{i \mid y_i = -1\}$, page 23

I_{pos} A set containing the indices of points belonging to the positive class, $I_{pos} = \{i \mid y_i = 1\}$, page 23

$K(\mathbf{x}_i, \mathbf{x}_j)$ Kernel product of $\mathbf{x}_i$ and $\mathbf{x}_j$, page 15

n Number of training samples, page 13

n_f Number of facets in a convex hull, page 41

n_r Number of facets in an RCH, page 56

n_v Number of vertices in an RCH, page 55

P_{neg} A set containing all points from the negative class, $P_{neg} = \{\mathbf{x}_i \in P \mid i \in I_{neg}\}$, page 24

P_{pos} A set containing all points from the positive class, $P_{pos} = \{\mathbf{x}_i \in P \mid i \in I_{pos}\}$, page 24

q Degree of the monomial or polynomial kernel, page 15

s_i The weight associated with training point $\mathbf{x}_i$, page 27

GACV Generalized Approximate Cross-Validation, page 148

MEB Minimal Enclosing Ball, page 29

NPA Nearest Point Algorithm, page 93

RCH Reduced Convex Hull, page 45

S-K Schlesinger-Kozinec, page 95

SMO Sequential Minimal Optimization, page 106

SVM Support Vector Machine, page 11

WRCH Weighted Reduced Convex Hull, page 60

WSK Weighted Schlesinger-Kozinec, page 110

WSVM Weighted Support Vector Machine, page 27

A Geometric Approach to Support Vector Classification

Ben Goodrich, BSc BCompSc (Hons)
[redacted]
Monash University, 2012

Supervisors: Dr. David Albrecht
Dr. Peter Tischer

Abstract

The Support Vector Machine (SVM) is a supervised classification technique which has been applied to a broad range of tasks. There has also been a significant amount of research focused on making SVMs more accurate or efficient. Past work has been approached from many different perspectives. For example, while some authors interpret an SVM as a method for statistical regularization, others interpret it as a large margin classifier. Others still simply consider it as a type of black box which results from a Quadratic Programming (QP) task. More recently, SVMs have been interpreted as the result of a nearest point algorithm operating over the Reduced Convex Hulls (RCHs) of two classes. This has come to be known as the geometric interpretation of SVMs. Previous work has described the relationship between SVMs and computational geometry. However, it has focused largely on using this information in order to present novel SVM training algorithms.

We use the geometric interpretation of SVMs in order to unify a broad range of existing research, and also to inform several new techniques. The main contributions of this work can be divided into two parts. First, we examine existing work from a geometric perspective. This proves invaluable for understanding how and why SVMs work, the impact their parameters have, their relationship to other classifiers, and the circumstances under which they can exhibit poor performance. Second, we use the geometric framework to inform new methods for solving SVM-related tasks such as training, point weighting, and parameter selection. One of the ways in which we achieve this is by generalizing some of the geometric concepts underlying SVMs. For example, one of the concepts we introduce is that of Weighted Reduced Convex Hulls (WRCHs), a generalized form of RCH. By closely examining the properties of RCHs and WRCHs, we are able to propose several algorithms for their construction. In turn, this allows us to incorporate new types of SVMs and their training algorithms under the geometric framework.

A Geometric Approach to Support Vector Classification

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Ben Goodrich
December 12, 2012

Acknowledgments

I would like to thank my supervisors, Dr David Albrecht and Dr Peter Tischer for being so generous with their time and ideas. Writing this thesis has been one of the most challenging, interesting, and rewarding things I've done in my life thus far, and it wouldn't have happened without your help.

Thanks also to other Monash academics who have provided feedback during my candidature, A/Prof. David Dowe during confirmation, and Prof. Geoff Webb during pre-submission.

I'd like to express gratitude for the financial support I received while undertaking my research. I was lucky enough to receive an Australian Postgraduate Award from the Australian Government, and also to obtain tutoring work from Monash University. It isn't often that people can say they were not only able to do something that they loved, but were financially supported in doing so, so I feel extremely fortunate.

Ben Goodrich

Monash University
December 2012

Chapter 1

Introduction

Support Vector Machines (SVMs) implement the machine learning task of supervised classification. This means that they are capable of classifying abstract objects as belonging to a particular class. The process of supervised classification is performed by using a training set to build a model. The training set consists of a number of objects for which the class is already known, making it analogous to a supervisor guiding the classifier on its task, allowing it to learn by example.

There is a range of practical applications which can be framed as classification tasks, and many of these application domains have been addressed using SVMs. For example, SVMs have been applied to perform handwritten digit recognition [15], medical diagnosis [93], spam classification [32], and face recognition [50], to name just a few. Guyon [51] provides a more extensive list of the practical applications of SVMs. Burges [19] has noted that, in many of these applications, SVMs tend to have an error rate which "either matches or is significantly better than that of competing methods".

One of the reasons SVMs are capable of performing so well over a range of applications is that there are a number of ways in which they can be adapted to a particular domain. A large amount of research has been focused on adapting or modifying SVMs to suit particular problems, such as automatically selecting parameter values [63, 33], or making the training process more efficient [91, 66].

Much of the research which is focused on SVMs differs greatly in terms of the framework used to understand how an SVM works and how it should be adapted to a particular task. For example, one of the most widespread interpretations of an SVM is that it is the result of a Quadratic Programming (QP) task [27]:

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i, \\ \text{subject to} \quad & \begin{cases} y_i(\mathbf{w}\cdot\mathbf{x}_i - b) & \geq 1-\xi_i \\ \xi_i & \geq 0. \end{cases} \end{aligned} \tag{1.1}$$

The intricacies of this QP task, including the terms that appear in it, are described later in this thesis, but this framework has been used to generalize SVMs by, for example, changing C, or by squaring or otherwise modifying the ξ_i terms.

SVMs also have an equivalent statistical interpretation as a regularization method [85]:

$$\min \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(\mathbf{x}_i)) + \lambda ||f||_{\mathcal{H}}^2. \tag{1.2}$$

This interpretation provides yet more ways to modify the SVM, for example by using a kernel to alter the feature space $\mathcal{H}$. The loss function L can also be modified, and this has been used to introduce probabilistic variants of the SVM. We refer to the approach of Equations (1.1) and (1.2) as *algebraic* approaches due to their algebraic representation.

Equations (1.1) (1.2) can be tweaked easily to adjust certain properties of SVMs. However, it is difficult to use them to understand exactly how or why an SVM works, and what impact, if any, modifications will have on the final machine. This has led to the more recent introduction of a *geometric* interpretation of SVMs [28, 7]. The geometric interpretation of an SVM is that it is equivalent to the perpendicular bisector of the shortest line between the Reduced Convex Hulls (RCHs) of the two classes. This interpretation allows an SVM to be clearly and intuitively understood in geometric terms. For example, Figure 1.1 shows an SVM separating the RCHs of the two classes.

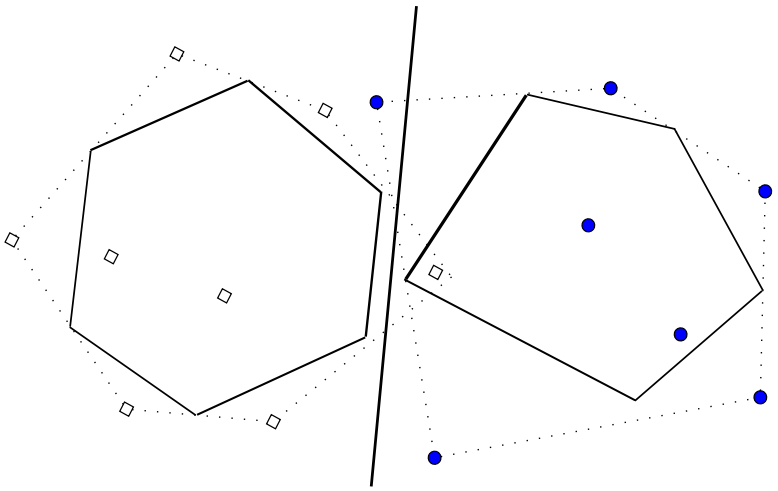


Figure 1.1: An SVM separating the RCHs of the two classes. The dotted outlines are convex hulls. The solid outlines are RCHs.

Our thesis is that the geometric interpretation of SVMs provides a means with which to understand, improve on, and generalize almost all aspects of support vector classification. In order to demonstrate this we first build on the geometric interpretation of SVMs by examining RCHs as a concept in their own right. In doing so we are able to construct several algorithms for computing RCHs independently and in their entirety. This allows us to compute and visualize RCHs, and discover several of their properties.

A better understanding of the geometric concepts underlying SVMs enables us to unify a broad range of existing research under the geometric framework. This in turn leads to a better understanding of how and why SVMs work, the impact their parameters have, their relationship to other classifiers, and the circumstances under which they can exhibit pathological behavior, resulting in extremely poor performance. For example, we show that there is a close relationship (or sometimes even equivalence) between SVMs and other

classifiers such as the k-nearest neighbor and k-means classifiers. We also suggest that geometric insights provide a useful means of illustrating the differences between similar (but distinct) classifiers such as SVMs and perceptrons.

We assert that the main difference between perceptrons and SVMs is that, while an SVM can be represented as a nearest point problem over two RCHs, a perceptron can not. However, perceptrons do still have a geometric interpretation as a one-class minimal norm task over a single RCH in a modified kernel feature space. These geometric interpretations of SVMs and perceptrons encompass both L_1 and L_2-loss variants, allowing for the characteristic differences between all types of SVMs and perceptrons to be understood in an intuitive manner. We hope that this geometric comparison of SVMs and perceptrons will lead to an improved understanding of the two machines, and prevent the terms perceptron and SVM from being used interchangeably in the literature as they have sometimes been in the past.

By generalizing the geometric concepts underlying SVMs we are able to introduce the concept of Weighted Reduced Convex Hulls (WRCHs). We use WRCHs to explain how and why Weighted SVMs (WSVMs) work. WSVMs, previously described in various forms by several authors Veropoulos et al. [118], Zadrozny et al. [129], are important because they allow SVMs to be trained on datasets where the cost of misclassification can vary from point to point or from class to class. This situation arises in many real world applications. For example, in spam classification, the cost of misclassifying spam as legitimate email is much lower than the cost of classifying legitimate email as spam. It follows that being able to understand how WSVMs work when they are applied to datasets such as these is important in order to have faith in the predictions which are made by a WSVM. We suggest that the concept of WRCHs allows for a much improved understanding of how a WSVM makes predictions.

In addition to providing an improved understanding of WSVMs, we show that WRCHs allow nearest point algorithms to be applied to solve the WSVM optimization task without necessarily adding additional computational complexity (depending on the weighting scheme that is applied). We demonstrate this by proposing a nearest point algorithm which operates over WRCHs. The resulting algorithm is novel in that by exploiting the geometric framework it is able to train L_1 and L_2-loss SVMs with individual or class-specific weights. Using previous insights into the differences between SVMs and perceptrons we are also able to show that the algorithm can be adapted to train L_1 and L_2-loss weighted perceptrons.

1.1 Thesis Overview and Contributions

This thesis contains a large amount of notation, including mathematical symbols and acronyms, most of which is introduced in the Background chapter and then used throughout the thesis. To aid in understanding this notation, a list of nomenclature is given on Page xvi. Each entry includes a brief description of the notation, along with an associated page number on which a more detailed explanation occurs (generally the first use of the

notation in the thesis). Because of the relatively broad scope of the thesis, an index is also provided on Page 221.

Background

Chapter 2 reviews the literature on SVMs and their geometric interpretation. In addition, this chapter reviews perceptrons and Minimal Enclosing Balls (MEBs), and explains how these concepts are closely related to SVMs. Information in this chapter provides a foundation for subsequent chapters, and is required in order to understand much of the later material in this thesis.

Reduced Convex Hulls

Chapter 3 examines reduced convex hulls independently of SVMs. By examining the properties of reduced convex hulls, we are able to generalize several convex hull algorithms so that they can be used to compute reduced convex hulls. We use these algorithms to visualize RCHs and discover several new properties of RCHs.

This chapter introduces the concept of Weighted Reduced Convex Hulls (WRCHs), a generalized RCH which allows for individual point weights to be specified. The weights in a WRCH specify how important it is that a point lies inside or on the hull. We describe several properties of WRCHs and examine how their vertices may be found. Using these properties we are able to adapt RCH algorithms to handle weighted data, and to compute WRCHs in their entirety. The definition of WRCHs is such that they have a close relationship with Weighted SVMs (WSVMs). This relationship is further explored in subsequent chapters, where WRCHs are applied in order to both understand and train WSVMs.

Understanding SVMs from a Geometric Perspective

In Chapter 4 we incorporate existing research under the geometric framework. We examine the significance of using RCHs for classification and their relationship with existing classifiers such as the k-nearest neighbor classifier, the k-means classifier, and the perceptron classifier. Additionally, we explain the differences between SVMs and perceptrons from a geometric perspective.

In this chapter we closely examine the threshold of an SVM. The threshold is the offset from the origin of the hyperplane which separates the RCHs of the two classes. It has previously been noted that the threshold used by most software packages is not the most geometrically intuitive threshold [28]. We build on this work to show that accuracy can actually be increased by choosing a geometrically informed threshold as opposed to the threshold suggested by the Karush-Kuhn-Tucker (KKT) conditions associated with the SVM QP task. Further, we show that, in some cases, the KKT conditions lead to an unsuitable choice of threshold.

Another contribution of this chapter is that we incorporate WSVMs under the geometric framework. By employing the concept of WRCHs, we describe from a geometric

perspective how and why WSVMs work. We use the geometric approach to WSVMs to describe three possible strategies for optimizing WSVM behavior: minimizing the overall error, minimizing the average proportion of errors, and minimizing the total cost of error. We examine the geometric implications of each strategy, and include an empirical demonstration. This provides an understanding of the circumstances under which point weighting can be beneficial (and when it can not).

Geometric Training Algorithms

Chapter 5 applies the theoretical results from the previous chapters to the practical task of training SVMs and WSVMs. We achieve this by generalizing the Schlesinger-Kozinec nearest point algorithm [100, 72] to operate over WRCHs, introducing a Weighted Schlesinger-Kozinec (WSK) algorithm. Because of the relationship between WRCHs and WSVMs, we are able to use this algorithm in order to train WSVMs. Training WSVMs using nearest point algorithms over WRCHs yields the same results as duplicating highly weighted points prior to training over standard RCHs. However, training using WRCHs is inherently faster than point duplication. Additionally, WRCHs are more versatile in that they allow non-integral weights to be specified.

We also investigate how the WSK algorithm can be optimized. Much of the previous work on optimizing nearest point algorithms has been directed towards algorithms operating on convex hulls. We build on this work by optimizing nearest point algorithms for WRCHs. We find that the WSK algorithm can be optimized in two ways. First, while computing the update step we exploit the property that the support vectors are only a subset of the training data. Second, we decrease the number of candidate support vectors by pushing the approximate nearest points towards the facets of the WRCHs as quickly as possible. As well as being an optimization in itself, this second step also boosts the effectiveness of the first step. Because the WSK algorithm can train standard SVMs when all weights are equal to one, we have been able to show that these optimizations also improve on many recently proposed unweighted nearest point algorithms.

Although the WSK algorithm is more efficient than many previous nearest point algorithms which operate over RCHs, and is competitive with SMO for *some* parameter values, it is generally not as efficient as SMO when margins become small and training times increase. However, these results are important because they qualify previous research which suggested that nearest point algorithms were superior to SMO. We suggest that previous results did not compare the two algorithms using equivalent parameter values and provide extensive new results which yield a fairer comparison.

An important contribution of this chapter is that we note that many recent nearest point algorithms which operate over RCHs [40, 109, 82] implement a stopping condition which is not equivalent to the stopping condition used by SMO. Further, this stopping condition is not ideal in that it does not take into account the distance between the nearest points. Consequently, more iterations are performed than required when this distance between the nearest points is large, and fewer iterations than required when the distance is small. We instead recommend a relative nearest point stopping condition (as employed

by Keerthi et al. [65] in a convex hull nearest point algorithm). We show using empirical trials that this stopping condition, while being no more difficult to compute, results in a much more consistent number of iterations being taken.

Finally, we describe how the WSK algorithm can be adapted to train hard or soft margin perceptrons, optionally with weighted training data. Perceptrons are a family of large margin classifiers similar to SVMs that can also be combined with kernels for non-linear classification. This is achieved by formulating the problem as a one-class nearest point (i.e. minimal norm) problem in conjunction with a modified kernel. Despite the simplicity of the perceptron nearest point approach, because of the more complex feature space required, the WSK algorithm can generally not compute perceptrons as efficiently as it can compute SVMs.

Model Selection using Geometric Information

Chapter 6 uses the geometric insights from previous chapters in order to help select the parameters for an SVM. There are several ways in which geometric information can aid in parameter selection. For example, the radius-margin technique selects the parameters which minimize the radius of the Minimal Enclosing Ball (MEB) of the training data compared to the margin of the SVM. The radius-margin technique has been shown to provide an upper bound on the leave-one-out error of an SVM [21].

By considering the geometric properties of the SVM and MEB optimization problems, we show that upper and lower bounds on the radius-margin ratio of an SVM can be efficiently computed at any stage during training. We use these bounds to accelerate radius-margin parameter selection by terminating training routines as early as possible, while still obtaining a guarantee that the parameters minimize the radius-margin ratio. Once an SVM has been partially trained on any set of parameters, we also show that these bounds can be used to evaluate and possibly reject neighboring parameter values with little or no additional training required. Empirical results show that, when selecting two parameter values, this process can reduce the number of training iterations required by a factor of 10 or more, while suffering no loss of precision in minimizing the radius-margin ratio.

We also address the task of performing parameter selection using μ-SVMs. We find that, although parameter selection is most commonly performed in conjunction with C-SVMs, there are several advantages to using the more geometrically intuitive μ-SVM formulation. For example, when μ is searched, the geometric interpretation can be used to set intuitive start points, end points and step sizes which have a clear impact on the reduced convex hulls of the two classes. This provides a more restricted search space than C-SVMs, for which C can take any value greater than zero.

Conclusions

We conclude that the geometric perspective provides an important means of understanding SVMs and their behavior. The geometric framework leads to a clear and precise definition of what an SVM is, and helps determine how SVMs differ from related large margin

techniques such as perceptrons. It also clearly illustrates a relationship between SVMs and other existing classifiers such as the k-means and k-nearest neighbor classifiers.

We feel that in some cases authors have drawn erroneous conclusions regarding SVMs because they have not used a common basis for comparison between different types of SVMs, or between SVMs and other classifiers. For example, the regularization parameter, kernel and kernel parameters, loss function, threshold, and stopping conditions used during training all have an impact on both the training efficiency and test accuracy of a machine. A common basis for comparison of these factors is easier to achieve if SVMs are understood from a geometric perspective.

In the final chapter we present several avenues for future work to focus on. One of these areas is to research whether RCHs and WRCHs can be of any benefit in domains outside of machine learning. For example, convex hulls have previously been used in statistics to order multivariate data [5], detect outliers [99, 53], and estimate probability density contours [36]. It would be interesting to discover whether substituting RCHs or WRCHs for convex hulls could generalize or otherwise increase the effectiveness of any of these techniques.

Another path for future research is to create new and unique geometric class representations which can be used in place of RCHs and WRCHs. RCHs and WRCHs exist in SVMs largely because computing the nearest points in two RCHs has a convenient QP representation. However, this does not mean other geometric class representations could not be used in their place. Indeed, alternative class representations could be used to construct fundamentally new types of classifiers which still retain the theoretical justification of large margins.

A final topic we recommend for future research is weighted SVM classification. In this thesis we have focused on understanding how and why weighted SVMs work, but there is still room for further research into specific applications where weighted SVMs can be applied. Such research could also aim to discover new weighting schemes which can increase the effectiveness of SVMs when applied in particular application domains.

1.2 Relationship with Existing Work

This thesis builds on previous work by several authors. The task of finding the nearest points in two convex hulls has been used to perform classification since long prior to the introduction of SVMs [100]. However, it was later pointed out that existing nearest point algorithms could be combined with kernels in order to train SVMs. This idea was used by Keerthi et al. [65] and Kowalczyk [71], who both proposed early nearest point algorithms for training hard margin and L_2-loss soft margin SVMs. Franc's PhD dissertation [39] is an extensive work studying the application of nearest point algorithms to L_2-loss and hard margin SVM optimization tasks.

Although there has been a geometric interpretation of hard margin and L_2-loss SVMs in terms of convex hulls for some time, the geometric interpretation of L_1-loss soft margin SVMs is more recent. The geometric interpretation of SVMs (in terms of RCHs) was

originally given by Crisp and Burges [28] and Bennett and Campbell [8]. This interpretation has been exploited by Tao et al. [108] and Mavroforakis and Theodoridis [82] in order to train L_1-loss SVMs by applying nearest point algorithms to reduced convex hulls. Mavroforakis's PhD dissertation [81] describes extensively the application of nearest point algorithms to train L_1-loss SVMs.

We distinguish our work from previous works by studying reduced convex hulls as a concept in their own right, rather than as a means to train SVMs. We first examine the theoretical properties of RCHs, and construct several algorithms capable of computing them independently, and in their entirety. This allows us to make contributions towards understanding how and why SVMs work. In addition, it enables us to describe circumstances under which SVMs are not likely to perform well, and to explain why this is the case. We have published selected results on this topic in Goodrich et al. [47].

Despite the fact that we do not focus *solely* on training SVMs, Chapter 5 does address the task of applying nearest point algorithms to training SVMs, and in doing so intersects with work by Mavroforakis [81] and Franc [39]. We do this in order to address questions which were left open in previous work, and also to revisit questions for which there is room for further clarification. In doing so, we are able to make several contributions to this area (as summarized in the previous section of this introduction). Some of these contributions have been published in Goodrich et al. [49].

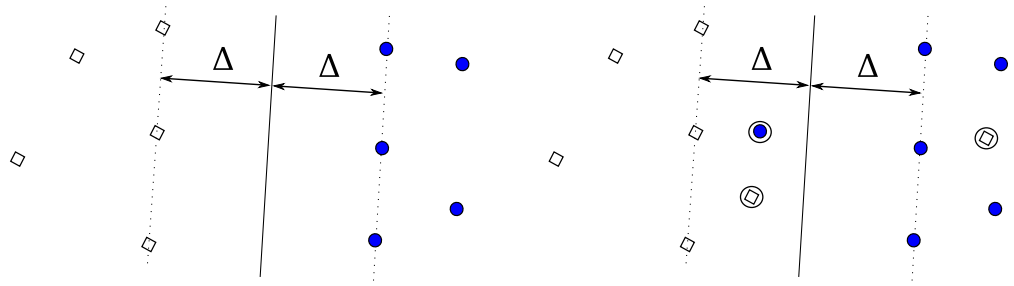
Parameter selection is a topic which has been addressed by several authors [59, 122, 33, 46]. We make our work distinct by exploiting the geometric interpretation of SVMs in our approach to parameter selection. We have published selected results arising from this approach in Goodrich et al. [48].

Chapter 2

Background

2.1 Introduction

This chapter provides an overview of the literature relating to SVM classification. There are a number of different types of SVMs, but the unifying idea is that of large margin classification. When the decision surface of a classifier is linear, and training data is linearly separable, a large margin can be created by maximizing the perpendicular distance from the hyperplane to the closest points in each class. Linear classifiers employing large margins on separable training data are referred to as *hard margin* SVMs (Figure 2.1a). These types of SVMs are introduced first in Section 2.2. Subsequently, Section 2.3 describes how kernels can be used to create non-linear SVMs.



(a) A hard margin SVM

(b) A soft margin SVM. Points which have been circled are on the incorrect side of the margin (note that some may still be on the correct side of the hyperplane).

Figure 2.1: SVMs are a linear classifier with maximal margin Δ

The main shortcoming of hard margin SVMs is that they are only defined when training data is linearly separable. When training data is not linearly separable, another approach must be taken, generally involving a trade-off between maximizing the margin and minimizing the number of points which lie on the incorrect side of the margin (Figure 2.1b). SVMs implementing this strategy are referred to as *soft margin* SVMs. The best way to perform this trade-off is not clear, and it depends on the type of penalty applied to points which lie in between the margin. In Section 2.4 we review a range of different penalties

that have been previously applied to SVMs, and show how they each relate to various loss functions.

Section 2.5 revisits hard and soft margin SVMs from a geometric perspective. By considering the geometric properties of the SVM problem, we describe how SVMs can be considered as the perpendicular bisector of the shortest line between two convex representations of the classes [28, 7]. The geometric interpretation of SVMs has significant implications both in terms of understanding and training SVMs. We address this in later chapters.

The final sections of this chapter covers several concepts which are closely related to SVMs. This includes Weighted SVMs (WSVMs), perceptrons and the Minimal Enclosing Ball (MEB) problem. As well as reviewing each of these concepts, we also describe how they relate to SVMs. This provides a foundation for subsequent chapters, where these concepts arise frequently.

2.2 Hard Margin SVMs

The Support Vector Machine is a binary (two-class) method of supervised classification which implements a maximum margin linear classifier [19, 85]. Although linearly separable training data is not alway achievable, the goal of obtaining a maximum margin is simplest to describe for linearly separable data. For this reason we first describe the case of linearly separable data, as addressed by Boser et al. [13]. Training data $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ and associated class labels $y_1, y_2, \ldots, y_n$ are linearly separable if there exists a hyperplane defined by normal $\mathbf{w}$ and offset b which satisfies:

$$y_i(\mathbf{w} \cdot \mathbf{x} - b) > 0, \quad \forall i.$$

In this case of linearly separable data, a maximum margin hyperplane is found by solving the quadratic optimization problem [117]:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1. \end{aligned} \tag{2.1}$$

The constraint of $y_i(\mathbf{w} \cdot \mathbf{x}_i - b) = 1$ for the closest points in each class is added to ensure a unique solution. With this constraint in place, the margin is given by $2/||w||$ and is hence maximized by Equation (2.1). A visual depiction of these variables is given in Figure 2.2.

Once $\mathbf{w}, b$ satisfying Equation (2.1) have been found, the decision function of the SVM is given by:

$$f(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} - b),$$

with the output corresponding to the predicted class of a new data point $\mathbf{x}$.

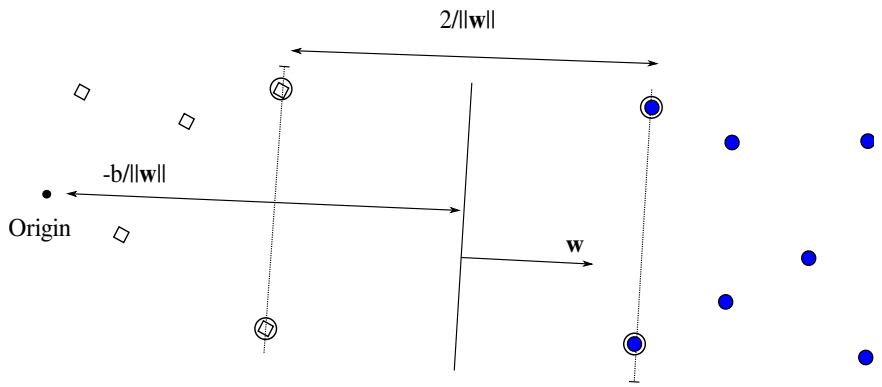


Figure 2.2: Hyperplane, offset and margin of an SVM. Support vectors are shaded.

2.2.1 The Hard Margin Dual

As well as the original *primal* SVM formulation given in Equation (2.1), an equivalent *dual* form can also be derived. The dual [38, 17] enables the use of a number of efficient optimization methods [91, 58] which generally can not be applied directly to the primal. It also reveals some additional geometric insights into the SVM optimization task. The dual is derived by introducing Lagrange multipliers $\alpha_i, \ldots, \alpha_n \geq 0$ for each inequality constraint in (2.1). The introduction of Lagrange multipliers creates a Lagrangian:

$$\mathcal{L} = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - 1)$$

We want to minimize the Lagrangian with respect to the primal variables whilst maximizing with respect to the Lagrange multipliers. Setting partial derivatives with respect to $\mathbf{w}, b$ to zero yields the equalities:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w}^\intercal - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^\intercal = 0 \qquad \Rightarrow \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i, \tag{2.2}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \qquad \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0. \tag{2.3}$$

We also have the complementary slackness condition:

$$\alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - 1) = 0, \tag{2.4}$$

which must be satisfied at optimality. Complementary slackness refers to the fact that if α_i is non-zero, $\mathbf{w} \cdot \mathbf{x}_i - b$ must equal zero. Conversely, if α_i is zero, there is no constraint on $\mathbf{w} \cdot \mathbf{x}_i - b$ [17]. Together Equations (2.2)-(2.4), along with the constraint $\alpha_i \geq 0$, and the condition $y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1$ from the primal, are referred to as the Karush-Kuhn-Tucker (KKT) conditions.

Substituting Equations (2.2)-(2.3) into the Lagrangian yields the dual objective function, which we want to *maximize* with respect to $\alpha_1, \alpha_2, \ldots, \alpha_n$. This maximization must occur under the constraint in Equation (2.3). Together this yields the dual optimization

task:

$$\begin{aligned} \max_{\alpha_1,\ldots,\alpha_n} \quad & -\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i, \\ \text{subject to} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned} \tag{2.5}$$

Primal variables may be recovered from the dual solution by exploiting the KKT conditions from Equations (2.2) and (2.4).

The variable b is often referred to as the *threshold* [19] or *offset* [8] of the hyperplane because it determines the offset of the hyperplane from the origin. If the threshold is zero, the hyperplane will pass through the origin. Because the distance from the hyperplane to the nearest points in both classes is equal, the threshold does not need to be optimized directly, as can be seen in (2.5). Instead, it can be computed using the KKT conditions once the optimal orientation has been found.

The dual and associated KKT conditions highlight several interesting properties of SVMs. Once Equation (2.5) has been optimized, there are generally a large number of α_i's which remain zero. Optimization problems with this property are referred to as *sparse* [17]. The points with an associated non-zero α_i are referred to as *support vectors* and solely determine the position of the hyperplane. Any other training points may be removed either before or after training to no effect [19].

Close examination of the KKT condition in Equation (2.4) reveals that support vectors for the hard margin SVM lie on two hyperplanes which run parallel to the separating hyperplane. These two hyperplanes, sometimes referred to as the *supporting planes* of the two classes [7], satisfy the equality $\mathbf{w} \cdot \mathbf{x}_i - b = \pm 1$. All support vectors from the positive class lie on one of the supporting planes, while all support vectors from the negative class lie on the other supporting plane. The KKT conditions further imply that there may be no points for which $y_i(\mathbf{w} \cdot \mathbf{x}_i - b) < 1$, i.e. points lying on the wrong side of the supporting plane. Further, any point which does not lie on a supporting plane can not be a support vector and can therefore be discarded without impacting the solution.

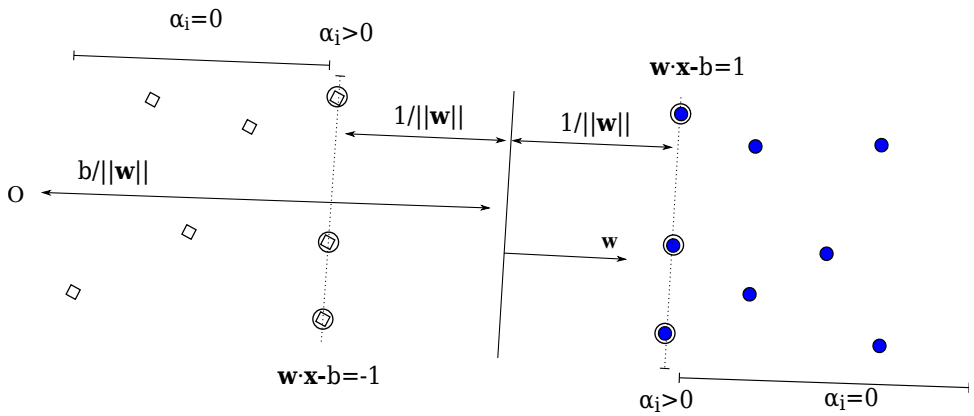


Figure 2.3: Support vectors and supporting planes of a hard margin SVM. Supporting planes are the dotted lines, with support vectors circled.

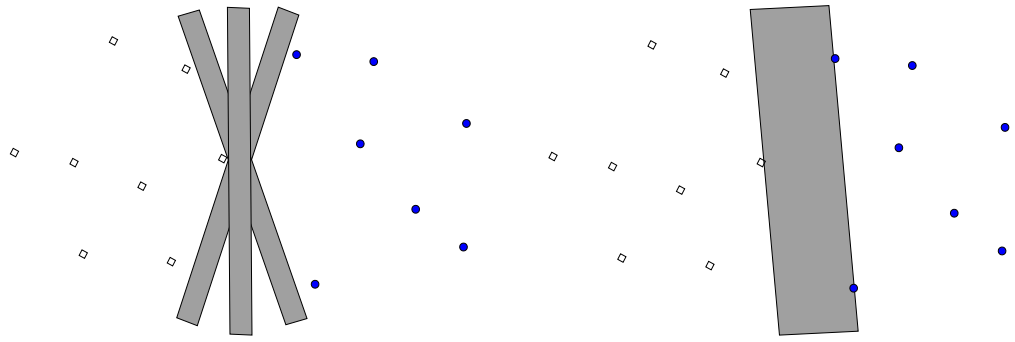
2.2.2 The Importance of Large Margins

Vapnik [117] provides an expression (Equation 2.6) which gives an upper bound on the leave-one-out error of an SVM. The leave-one-out error of a classifier is an approximation of the test error which we will discuss in further detail in Chapter 6. In Equation (2.6), the radius of the Minimal Enclosing Ball (MEB) of all training data is denoted R, the margin of the SVM is Δ and the total number of training samples is n.

$$E_{\text{LOO}} < E\left[\frac{R^2}{\Delta^2 n}\right] \tag{2.6}$$

Notice in Equation (2.6) how, as the width of the margin increases, the upper bound on the leave-one-out error must shrink (provided the radius of the MEB remains fixed). Accordingly, this bound can be used to argue that a larger margin means a greater likelihood of making correct classifications.

The bound in Equation 2.6, often referred to as the radius-margin bound, is derived from the Vapnik-Chervonenkis (VC) dimension of the classifier [117]. It can be most intuitively understood using the 'skinny' (small margin) vs 'fat' (large margin) hyperplane argument of Bennett and Campbell [8]. A 'skinny' hyperplane (Figure 2.4a) can take a vast number of orientations while still separating a dataset, whereas a 'fat' hyperplane (Figure 2.4b) has a greatly reduced number of orientations. If the 'fat' hyperplane has a *maximum* margin, it can only take one possible orientation whilst still separating the data. It is in this regard that a hyperplane with a small margin has a greater capacity to fit the data, and is hence considered more complex [8].



(a) Small margin hyperplanes have greater capacity to overfit the data

(b) Optimal margin hyperplanes have less capacity to overfit the data

Figure 2.4: 'Skinny' and 'fat' hyperplanes (adapted from Bennett and Campbell [8])

One of the greatest benefits the radius-margin bound is that it applies regardless of the dimensionality of the data. Dimensionality often poses a problem in classification because, as the dimensionality increases, so too does the possible number of models which can be formed. For a simple example of this, consider a set of data where each attribute has two possible values. In a one-dimensional input space, there are two possible ways to separate the data. However in a d dimensional space, this increases to 2^d, i.e. each additional feature squares the possible number of models. This effect is further magnified if attributes are not simple binary values.

A justification for large margins can also be understood within a compression framework via the Minimum Message Length (MML) principle [125, 126, 124]. From an MML perspective, when choosing between an infinite number of hyperplanes which describe a set of data equally well, the best choice is the one which can be stated in the smallest number of bits. Tan and Dowe [107] note that, of the infinite separating hyperplanes which could be constructed, the maximum margin hyperplane can be stated with the least precision. This means that the larger the margin of a hyperplane, the greater the compression that can be applied while still having it correctly classify the training set.

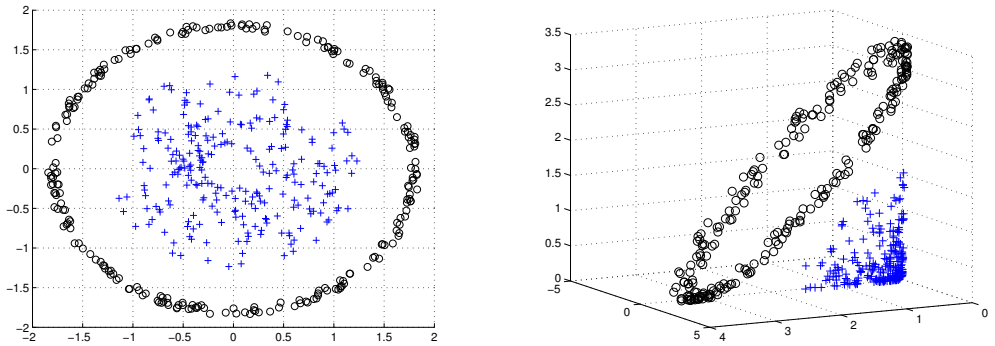
2.3 Using Kernels for Non-Linear Classification

Although the hard margin SVMs described in the previous section work well when applied to linearly separable datasets, there remains a problem of trying to learn more complex surfaces such as clusters or curves in input space (Figure 2.5a). A linear discriminant is incapable of learning such surfaces with a high degree of accuracy.

The problem of learning non-linear decision surfaces with SVMs may be addressed by introducing a technique known as the 'kernel trick' [86, 19]. The kernel trick is equivalent to mapping data from input space to a (generally higher or even infinite dimensional) feature space (denoted $\mathcal{H}$) using a function $\phi : \mathbb{R}^d \rightarrow \mathcal{H}$. A simple example of such a mapping is the function [102, 117]:

$$\phi(\mathbf{x}) = \left(x_1^2, x_1 x_2 \sqrt{2}, x_2^2\right), \tag{2.7}$$

This mapping forms a feature space consisting of second degree monomials. The transformation from input to feature space is shown in Figure 2.5. Note that the data is mapped to a manifold in the feature space. The result is that data becomes linearly separable in feature space and may be more accurately learned by an SVM. The resulting decision surface is non-linear with respect to input space.



(a) Data in input space

(b) The same data in a feature space of second degree monomials

Figure 2.5: The effect of the mapping in Equation 2.7

The beauty of the kernel trick is that dot products in feature space may be computed without explicitly performing (or even knowing) the mapping itself. Consider the function:

$$\begin{aligned} K(\mathbf{x}, \mathbf{y}) &= (\mathbf{x} \cdot \mathbf{y})^2 \\ &= (x_1 y_1 + x_2 y_2)^2 \\ &= x_1^2 y_1^2 + 2 x_1 y_1 x_2 y_2 + x_2^2 y_2^2 \\ &= (x_1^2, \sqrt{2} x_1 x_2, x_2^2) \cdot (y_1^2, \sqrt{2} y_1 y_2, y_2^2). \end{aligned}$$

This function implements the kernel product associated with the second degree monomial mapping in Equation (2.7) [102, 117]. Since training data in the SVM optimization task appears solely in terms of dot products, a kernel product may be substituted for a dot product in order to train a non-linear classifier. Several kernels which can be used in this manner are shown in Table 2.1.

Table 2.1: Some commonly used SVM kernels [86]. Parameters satisfy $q \in \mathbb{N}$, $\gamma \in \mathbb{R}^+$

Linear	$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})$
Polynomial	$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^q$
Monomial	$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^q$
Gaussian RBF	$K(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \|\|\mathbf{x} - \mathbf{y}\|\|^2)$

Despite their elegance, there are also a number of issues which arise with the use of a kernel. The introduction of a kernel means the introduction of associated parameters. For example, polynomial and monomial kernels require a degree to be chosen, and a width must be set for Gaussian radial basis function kernels. The introduction of a kernel also means it becomes infeasible to directly compute the separating hyperplane:

$$\mathbf{w} = \sum_i \alpha_i \phi(\mathbf{x}_i).$$

This means that the decision function must be stated in terms of the support vectors:

$$f(\mathbf{x}) = \text{sgn}(\sum_{i=1}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}) - b). \tag{2.8}$$

It follows that for non-linear SVMs, the number of support vectors has a direct impact on the speed with which predictions can be made [18].

2.4 Soft Margin SVMs

The main problem with the hard margin SVM formulation is that it requires linearly separable training data. This requirement is rarely met with real world datasets, where noise and inherent overlap in the classes mean that a hard margin solution often can not be found.

Cortes and Vapnik [27] first addressed the problem of computing large margin classifiers for non-separable data by introducing a trade-off between the width of the margin and the

number of errors in the training data. The result is a modified optimization task, referred to as the *soft margin* SVM:

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i^k, \\ \text{subject to} \quad & \begin{cases} y_i(\mathbf{w}\cdot\mathbf{x}_i - b) & \geq 1-\xi_i \\ \xi_i & \geq 0. \end{cases} \end{aligned} \tag{2.9}$$

Here k is an integer greater than or equal to one. The variables ξ_i are 'slack variables', equal to the error of a point, measured as the shortest distance to the correct side of the margin (Figure 2.6). Note that a point does not necessarily have to be incorrectly classified to have an associated non-zero slack variable. For example, point $\mathbf{x}_2$ in Figure 2.6 is correctly classified, but because it does not satisfy $\mathbf{w}\cdot\mathbf{x}_2 - b \geq 1$, it is considered to lie on the incorrect side of the margin and must have an associated slack variable.

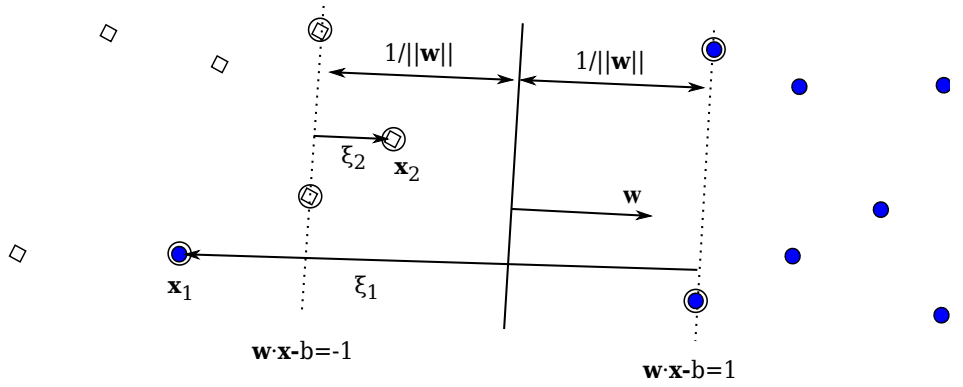


Figure 2.6: Slack variables

Equation (2.9) also has a statistical interpretation as a regularization method which minimizes an empirical loss while penalizing functions with a larger norm [74, 85].

$$\min \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(\mathbf{x}_i)) + \lambda||h||_{\mathcal{H}}^2. \tag{2.10}$$

Here $f(\mathbf{x}) = h(\mathbf{x}) - b$ is the classification function, L is a loss function and λ is a regularization parameter which determines the penalty on solutions with a large norm. The norm $||h||_{\mathcal{H}}^2$ is computed in feature space $\mathcal{H}$.

The optimization task (2.10) is known to have a solution of the form $h(\mathbf{x}) = \sum_i y_i\alpha_i K(\mathbf{x}_i, \mathbf{x})$, and therefore $||h||_{\mathcal{H}}^2 = \sum_{i,j}\alpha_i\alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$ [69, 122]. This means that, using the change of variables $C = 1/(2\lambda n)$, the optimization task in (2.10) can become equivalent to the soft margin SVM in (2.9), depending on the choice of loss function L.

There are three loss functions commonly used in conjunction with Equation (2.10):

- L_1-loss, or hinge loss: $L\left(y_i, f(\mathbf{x}_i)\right) = \max(1 - y_i f(\mathbf{x}_i), 0)$
- L_2-loss, or squared hinge loss: $L(y_i, f(\mathbf{x}_i)) = [\max(1 - y_i f(\mathbf{x}_i), 0)]^2$
- Logistic loss: $L(y_i, f(\mathbf{x})) = \log(1 + e^{-y_i f(\mathbf{x})})$

When an L_1-loss or L_2-loss function is used, Equation (2.10) becomes equivalent to the soft margin SVM (Equation 2.9) using $k = 1$ or $k = 2$, respectively. When a logistic loss function is used, the resulting machine is known as a Kernel Logistic Regression (KLR) machine [67]. We do not elaborate on the KLR machine since it falls under the domain of probabilistic methods. However, we describe the individual properties of the SVMs resulting from the L_1 and L_2 hinge loss functions in the sections below.

Hinge loss is a desirable loss function to use because it ensures only points lying close to the decision boundary have an impact on the solution. This preserves the *sparsity* of the machine, meaning that only some of the input points will become support vectors. Recall that sparsity is a desirable property for an SVM to have, since it means many training points can be discarded either prior to, during, or after training while having no impact on the final solution. Sparsity can be exploited to great effect in order to optimize training algorithms for SVMs [91], as we will discuss in later chapters.

2.4.1 L_1-loss SVMs

One of the first, and perhaps most widely, SVMs arising from the soft margin formulation of Cortes and Vapnik [27] was the L_1-loss C-SVM, given in primal form as:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & \begin{cases} y_i(\mathbf{w}\cdot\mathbf{x}_i - b) \geq 1 - \xi_i. \\ \xi_i \geq 0 \end{cases} \end{aligned} \tag{2.11}$$

In the L_1-loss formulation, equivalent to Equation (2.10) in conjunction with the hinge loss function, the sum of slack variables is used to penalize misclassified points.

Because of the widespread use of the L_1-loss function in popular SVM software packages such as `SVMlight` [58] and `LIBSVM` [20], L_1-loss C-SVMs are often referred to simply as C-SVMs. For this reason, when we refer to a C-SVM without qualifying the type of loss function used, we are always referring to an L_1-loss C-SVM.

The L_1-loss SVM Dual

Similar to the hard margin case, the dual of a soft margin SVM is derived by introducing Lagrange multipliers $\alpha_i, \ldots, \alpha_n, \beta_1, \ldots, \beta_n \geq 0$ for each inequality constraint in (2.1). Notice that we require the additional β_i terms in the L_1-loss soft margin case due to the additional constraints forcing all $\xi_i \geq 0$. This creates a Lagrangian:

$$\mathcal{L} = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i(\mathbf{w}\cdot\mathbf{x}_i - b) - 1 + \xi_i) - \sum_{i=1}^{n}\beta_i\xi_i$$

As before, we want to minimize the Lagrangian with respect to the primal variables whilst maximizing with respect to the Lagrange multipliers. Setting partial derivatives with

respect to $\mathbf{w}, b$ to zero yields:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w}^\intercal - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^\intercal = 0 \qquad \Rightarrow \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i, \tag{2.12}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \qquad \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0, \tag{2.13}$$

$$\frac{\partial \mathcal{L}}{\partial \xi_i} = C - \alpha_i - \beta_i = 0 \qquad \Rightarrow C - \alpha_i = \beta_i \tag{2.14}$$

We also have the conditions of complementary slackness:

$$\alpha_i(y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - 1 + \xi_i) = 0, \qquad \beta_i \xi_i = 0, \tag{2.15}$$

which must be satisfied at optimality.

Substituting Equations (2.12)-(2.14) into the Lagrangian yields the dual objective function, which we want to maximize with respect to $\alpha_1, \alpha_2, \ldots, \alpha_n$. Conveniently, β_i terms disappear, yielding the dual optimization task:

$$\begin{aligned} \max_{\alpha_1,\ldots,\alpha_n} \quad & -\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i, \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i y_i = 0 \\ 0 \leq \alpha_i \leq C \end{cases} \end{aligned} \tag{2.16}$$

Importantly, the L_1-loss soft margin SVM is extremely similar to the hard margin dual given previously in Equation 2.5. On comparison, the only difference is that the original hard margin constraint of $\alpha_i \geq 0$ has been modified to become $0 \leq \alpha_i \leq C$.

Because of the change in KKT conditions between the hard and soft margin SVMs, the significance of the support vectors and α_i values also changes. Recall from Section 2.2.1 that the supporting planes of a C-SVM are the hyperplanes satisfying $\mathbf{w} \cdot \mathbf{x} - b = \pm 1$. We say that a point $\mathbf{x}_i$ is on the correct side of the supporting hyperplane of its class if it satisfies $y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1$. Otherwise it is on the incorrect side of the supporting plane. Note that if a point lies on the correct side of the supporting plane, it must be correctly classified by the decision surface. However, if a point lies on the incorrect side of its supporting plane, it may or may not be correctly classified by the decision surface.

Combining Equations (2.14) and (2.15) reveals three distinct cases regarding the α_i value associated with a training point:

- If $\alpha_i = 0$, the associated training point $\mathbf{x}_i$ is correctly classified by the hyperplane, and lies on the correct side of the supporting hyperplane of its class. Since it has no impact on the decision surface it may be removed before or after training to no effect [117].

- If $0 < \alpha_i < C$, the associated training point $\mathbf{x}_i$ is correctly classified by the decision surface, and lies *on* the supporting plane of its class.

- If $\alpha_i = C$, the associated training point $\mathbf{x}_i$ lies on the incorrect side of the supporting hyperplane of its class. Accordingly, it may or may not be correctly classified by the decision surface.

The change in possible α_i values means that there are now two distinct types of support vectors. Support vectors for which $0 \leq \alpha_i < C$ must satisfy $\mathbf{w} \cdot \mathbf{x}_i - b = \pm 1$ and hence always lie on the supporting plane of a class. Other support vectors, which satisfy $\alpha_i = C$, must satisfy $\mathbf{w} \cdot \mathbf{x}_i - b < 1$ and therefore lie on the incorrect side of their supporting plane (although they may or may not lie on the incorrect side of the hyperplane itself). We refer to these support vectors as *bounded* since their corresponding α's are capped at their upper bound. Figure 2.7 depicts possible values for α_i, with support vectors circled.

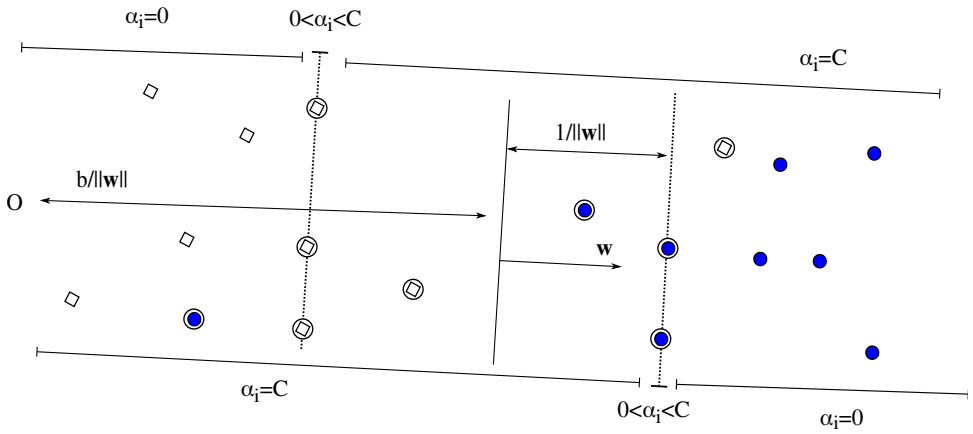


Figure 2.7: Support vectors and supporting planes of a C-SVM. Supporting planes are the dotted lines, with support vectors circled. The top line shows the possible values of Lagrange multipliers for the negative class. The bottom line shows the possible values of Lagrange multipliers for the positive class.

2.4.2 L_2-loss SVMs

An alternative to the L_1-loss SVM described in the previous section is the L_2-loss SVM. L_2-loss SVMs penalize the sum of *squared* slack variables in the objective function, as opposed to the sum of errors [27]. This is stated as the QP task:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i^2 \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1 - \xi_i. \end{aligned} \tag{2.17}$$

The parameter C is a regularization parameter that controls the trade-off between a large margin and a small sum of squared margin errors.

The L_2-loss SVM Dual

The dual optimization task for the L_2-loss SVM is given by:

$$\begin{aligned} \min_{\alpha} \quad & -\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i - \frac{1}{4C}\sum_{i=1}^{n} \alpha_i^2 \\ \text{subject to} \quad & \alpha_i \geq 0. \end{aligned} \tag{2.18}$$

This optimization task has associated KKT conditions:

$$\begin{aligned} \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i, \qquad \xi_i = \frac{\alpha_i}{2C} \\ \alpha_i(y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - 1 + \xi_i) = 0 \end{aligned} \tag{2.19}$$

Notice that there are some differences between the L_2-loss and L_1-loss SVM KKT conditions which have some important implications. The L_2-loss KKT conditions imply that there are two distinct cases for an α_i value associated with a training point:

- If $\alpha_i = 0$, the associated training point $\mathbf{x}_i$ is correctly classified and lies on the correct side of the supporting hyperplane, i.e. $y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1$
- If $\alpha_i > 0$, the associated training point $\mathbf{x}_i$ is a support vector and lies either on the supporting hyperplane, or on the incorrect side of the supporting hyperplane.

Importantly, an L_2-loss SVM no longer involves the concept of bounded or unbounded support vectors. Instead, each support vector has an associated Lagrange multiplier which is proportional to its associated slack variable so that $\alpha_i = 2C\xi_i$. This is consistent with the sum of squares approach, where larger errors have a greater influence on the final solution, and smaller errors have a lesser influence.

The L_2-loss SVM dual also has a different objective function compared with the L_1-loss SVM. A common trick to use in conjunction with an L_2-loss SVM is to write the objective function:

$$-\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \left(\mathbf{x}_i \cdot \mathbf{x}_j + \frac{\delta_{ij}}{2C}\right) + \sum_{i=1}^{n} \alpha_i$$

Here δ_{ij} is the Kronecker delta, defined as:

$$\delta_{ij} = \begin{cases} 1 & \text{if i=j} \\ 0 & \text{otherwise} \end{cases}$$

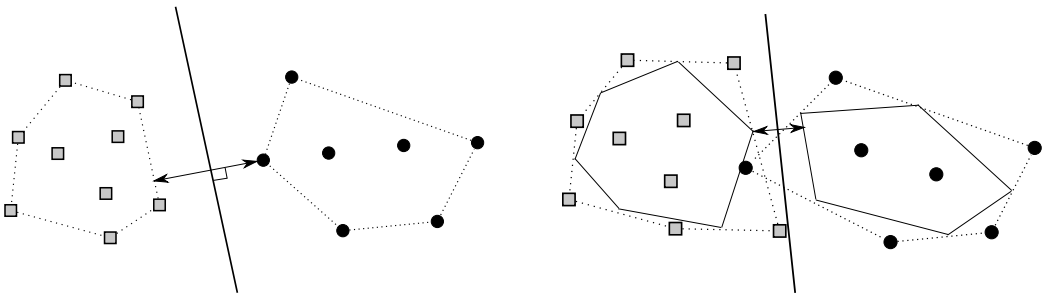
Once this transformation has been made, the L_2-loss SVM becomes equivalent to a hard margin SVM with the modified kernel $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j + \delta_{ij}/2C$. This means that L_2-loss SVMs may be solved using most hard margin techniques. We will further discuss the use of the Kronecker delta and the connection between hard and soft margin SVMs in the following sections.

2.5 Geometric Interpretations of SVMs

SVMs are already in some respects a geometrically intuitive classifier due to the way in which they incorporate large margins. However, the dual form of the various SVMs we have explored are highly analytical problems which are difficult to understand in geometric terms. In this section we describe changes that can be made to the L_1 and L_2-loss SVM primals which result in more geometrically intuitive, but equivalent, dual tasks. The geometric forms of these duals become nearest point problems, where each class is represented by a convex set and the optimization task reduces to finding the nearest points in the two sets [7, 28]. As well as allowing a much greater understanding of how SVMs work, this also allows SVMs to be computed using relatively simple nearest point algorithms.

2.5.1 μ-SVMs: Reparameterizing the L_1-loss Primal

Some years after the introduction of the C-SVM it was discovered that SVMs have an intuitive geometric interpretation [7, 28]. In the hard margin case (Figure 2.8a), it was shown that the hyperplane is the perpendicular bisector of the shortest line between the convex hulls of the two classes [7, 28]. In the soft margin case (Figure 2.8b), the convex hulls are reduced to avoid overlap before the closest points are found [7]. The mechanism behind the reduction of the hulls is understood by considering a reparameterization of the C-SVM known as the μ-SVM.



(a) Hard margin SVM **(b)** Soft margin SVM

Figure 2.8: SVMs as a convex hull problem

The μ-SVM optimization task weighs the slack variables against the width of the margin in a slightly different way, stated as [7, 28]:

$$\begin{aligned} \min_{\mathbf{w},b,\rho,\xi} \quad & ||\mathbf{w}||^2 - 2\rho + \mu \sum_{i=1}^{n} \xi_i, \\ \text{subject to} \quad & \begin{cases} y_i(\mathbf{w} \cdot \mathbf{x}_i - b) & \geq \rho - \xi_i \\ \xi_i & \geq 0 \\ \rho & > 0. \end{cases} \end{aligned} \tag{2.20}$$

The value of $\rho/||w||$ is equal to the width of the margin, so the aim is still to maximize the margin penalized by slack variables. The user-specified constant $0 < \mu \leq 1$ (which

replaces the constant C in the C-SVM) specifies whether more weight is given to a large margin or a small number of slack variables.

The μ-SVM Dual

The Lagrangian dual [38, 17] of the μ-SVM optimization problem, as well as being easier to solve, reveals additional geometric insights. The dual problem is an equivalent formulation derived by introducing Lagrange multipliers $\alpha_i, \ldots, \alpha_n, \beta_i, \ldots, \beta_n, \gamma \geq 0$ for each inequality constraint in (2.20). This defines a Lagrangian:

$$\mathcal{L} = ||\mathbf{w}||^2 - 2\rho + \mu \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - \rho + \xi_i) - \sum_{i=1}^{n} \beta_i \xi_i - \gamma\rho.$$

We want to minimize the Lagrangian with respect to the primal variables whilst maximizing with respect to the Lagrange multipliers. Setting partial derivatives with respect to $\mathbf{w}, b, \rho, \xi_i$ to zero yields the equalities:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 2\mathbf{w}^\intercal - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^\intercal = 0 \qquad \Rightarrow \mathbf{w} = \frac{1}{2} \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i, \tag{2.21}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \qquad \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0, \tag{2.22}$$

$$\frac{\partial \mathcal{L}}{\partial \rho} = -2 + \sum_{i=1}^{n} \alpha_i - \gamma = 0 \qquad \Rightarrow \sum_{i=1}^{n} \alpha_i = 2 + \gamma, \tag{2.23}$$

$$\frac{\partial \mathcal{L}}{\partial \xi_i} = \mu - \alpha_i - \beta_i = 0 \qquad \Rightarrow \beta_i = \mu - \alpha_i. \tag{2.24}$$

We also have the complementary slackness conditions:

$$\alpha_i(y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - \rho + \xi_i) = 0, \qquad \beta\xi_i = 0, \qquad \gamma\rho = 0, \tag{2.25}$$

which must be satisfied at optimality.

Using Equations (2.21)-(2.24) in conjunction with the Lagrangian yields the dual optimization task:

$$\begin{aligned} \max_{\alpha_i, \ldots, \alpha_n} \quad & -\frac{1}{4} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j, \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i y_i = 0 \\ \sum_{i=1}^{n} \alpha_i = 2 \\ 0 \leq \alpha_i \leq \mu \end{cases} \end{aligned} \tag{2.26}$$

From Equations (2.24) and (2.25), we can see that there still exists both bounded and unbounded support vectors in a μ-SVM. The only difference in the support vectors between a μ-SVM and a C-SVM is that bounded support vectors are now fixed at $\alpha_i = \mu$ as opposed to $\alpha_i = C$, and unbounded support vectors must now satisfy $0 < \alpha_i < \mu$ instead of $0 < \alpha_i < C$.

The μ-SVM is a reparameterization of the original C-SVM [7, 28] (described in Section 2.4.1), with

$$\mathbf{w} = \mathbf{w}'\rho \qquad b = b'\rho \qquad \xi_i = \xi_i'\rho \qquad \mu = 2C\rho.$$
$$\rho = \frac{1}{\sum_i \alpha_i'} \qquad \alpha_i = 2\alpha_i'\rho.$$

This can be verified by substituting these variables into the KKT conditions of the μ-SVM KKT conditions, yielding the C-SVM KKT conditions (to within a constant factor). Because the KKT conditions are both necessary and sufficient for optimality of this problem [17, 38], the two solutions must be equivalent (under scaling). This implies that a μ-SVM trained with regularization parameter μ is equivalent to a C-SVM trained with $C = \mu/(2\rho)$.

2.5.2 A Geometric Interpretation of μ-SVMs

The significance of the μ-SVM as compared to other parameterizations is that it has an intuitive geometric interpretation which is more extensive than the parallel supporting planes interpretation of C-SVMs. In order to understand this geometric interpretation let us first review the formal definitions of convex hulls and reduced convex hulls.

The convex hull of a set of n d-dimensional points $P = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \in \mathbb{R}^d$ is defined as the smallest convex set enclosing P, written [94]:

$$\mathrm{CH}(P) = \left\{ \sum_i^n \alpha_i x_i \;\middle|\; \sum_i^n \alpha_i = 1, \quad 0 \leq \alpha_i \leq 1 \right\}.$$

The border of a convex hull is a convex polytope, as depicted previously in Figure 2.8a.

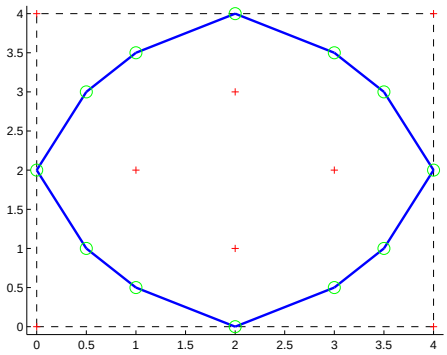
The reduced convex hull is a generalized form of convex hull which allows for a reduction parameter μ to specify the extent to which the convex hull should he shrunk. The reduced convex hull of a set of points $P = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ is given by [7, 28]:

$$\mathrm{RCH}(P, \mu) = \left\{ \sum_i \alpha_i x_i \;\middle|\; \sum_i \alpha_i = 1, \quad 0 \leq \alpha_i \leq \mu \right\}.$$

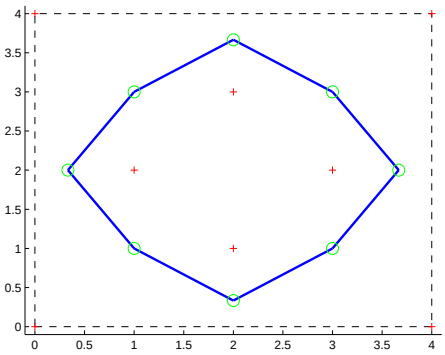
This is the same as the definition of a convex hull, except a constant $0 < \mu \leq 1$ has been introduced, limiting the impact any one point can have on the hull. As μ decreases, the reduced convex hull shrinks non-uniformly towards the centroid (Figure 2.9). At $\mu = 1/n$, the reduced convex hull is the centroid, and for further reductions the hull is empty.

The geometric interpretation of the μ-SVM is that the hyperplane formed is equivalent to the perpendicular bisector of the shortest line between the reduced convex hulls of the two classes. To show this, let us first define sets containing the indices of points in the positive and negative classes:

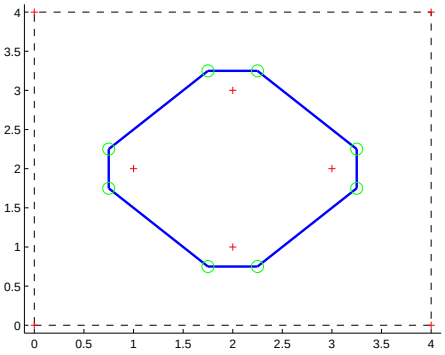
$$I_{pos} = \{i \mid y_i = 1\} \qquad I_{neg} = \{i \mid y_i = -1\}$$



(a) $\mu = 1/2$



(b) $\mu = 1/3$



(c) $\mu = 1/4$

Figure 2.9: Reduced convex hulls in the plane with varying μ. Plus marks represent points from P. Circles represent the vertices of the reduced convex hull of P. The convex hull is shown as a dashed line.

Next, let us define the sets:

$$P_{pos} = \{\mathbf{x}_i \mid i \in I_{pos}\}, \qquad P_{neg} = \{\mathbf{x}_i \mid i \in I_{neg}\},$$

containing only points from the positive and negative classes, respectively. Assuming an equal amount of reduction μ for both hulls, two points $\mathbf{p}_{pos}$ and $\mathbf{p}_{neg}$ from the reduced convex hulls of the two classes can be represented as:

$$\mathbf{p}_{pos} = \sum_{i \in I_{pos}} \alpha_i \mathbf{x}_i, \qquad \mathbf{p}_{neg} = \sum_{i \in I_{neg}} \alpha_i \mathbf{x}_i, \tag{2.27}$$

$$\text{subject to} \qquad 0 < \alpha_i \leq \mu, \quad \sum_{i \in I_{pos}} \alpha_i = 1, \quad \sum_{i \in I_{neg}} \alpha_i = 1. \tag{2.28}$$

Finding the shortest line between the two convex hulls equates to finding the closest two points in the hull [28]. This can be done by minimizing $||\mathbf{p}_{pos} - \mathbf{p}_{neg}||^2$ which, using

(2.27) and (2.28) is the same as minimizing:

$$\begin{aligned} F &= || \sum_{i \in I_{pos}} \alpha_i \mathbf{x}_i - \sum_{i \in I_{neg}} \alpha_i \mathbf{x}_i ||^2 \\ &= || \sum_{i=1}^{n} y_i \alpha_i \mathbf{x}_i ||^2 \\ &= \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j. \end{aligned}$$

Using (2.28) we also have the constraints:

$$0 \leq \alpha_i \leq \mu, \quad \sum_i \alpha_i = 2, \quad \sum_i \alpha_i y_i = 0.$$

Notice that this problem is equivalent to the dual μ-SVM optimization problem (Equation 2.26), where the objective function has been multiplied by a constant and the sign has been inverted in order to change the problem to one of minimization [28].

2.5.3 Reparameterizing the L_2-loss Primal

A ρ term can also be introduced to the L_2-loss primal in order to provide a more geometrically intuitive, but equivalent, dual problem. Consider the primal:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 - 2\rho + C \sum_{i=1}^{n} \xi_i^2 \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq \rho - \xi_i. \end{aligned} \tag{2.29}$$

This primal produces the dual:

$$\begin{aligned} \max_{\alpha} \quad & -\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j - \frac{1}{2C} \sum_{i=1}^{n} \alpha_i^2 \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i = 2, \\ 0 \leq \alpha_i \leq 1. \end{cases} \end{aligned} \tag{2.30}$$

Additional KKT conditions are given by:

$$\begin{aligned} \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i, \qquad \xi_i = \frac{\alpha_i}{2C} \\ \alpha_i(y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - \rho + \xi_i) = 0 \end{aligned} \tag{2.31}$$

This is a reparameterization of the L_2-loss SVM from Section 2.4.2, where:

$$\begin{aligned} \mathbf{w} &= \mathbf{w}'\rho & b &= b'\rho & C &= C' \\ \alpha_i &= 2\alpha_i'\rho & \rho &= \frac{1}{\sum_i \alpha_i'} \end{aligned}$$

This can be verified by substituting these equalities into the KKT conditions in (2.31) to yield the KKT conditions from the original form of the L_2-loss SVM. Notice that the regularization parameter C has not been rescaled in this case, it has precisely the same impact on both machines.

2.5.4 A Geometric Interpretation of L_2-loss SVMs

The conceptual benefits of the L_2-loss SVM become apparent when it is considered in conjunction with a kernel. Recall that a kernel computes the dot product of two points after mapping them to a higher (possibly infinite) dimensional feature space where the classes are more likely to become separable. Conveniently, this mapping does not need to be explicitly computed provided points are expressed solely in terms of dot products. If we replace the dot products in the L_2-loss SVM dual with kernel products $K(\mathbf{x}, \mathbf{y})$ we obtain:

$$\begin{aligned} \max_{\alpha} \quad & -\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \left[K(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij} \frac{1}{2C} \right] \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i = 2, \\ 0 \leq \alpha_i \leq 1. \end{cases} \end{aligned} \tag{2.32}$$

Recall that the Kronecker delta, δ_{ij}, is equal to one if $i = j$, and zero otherwise.

The implication of Equation (2.32) is that the L_2-loss SVM problem is equivalent to the standard hard margin SVM problem using the modified kernel [29]:

$$k(\mathbf{x}, \mathbf{y}) = K(\mathbf{x}, \mathbf{y}) + \frac{\delta_{ij}}{2C}. \tag{2.33}$$

The ability to express an L_2-loss SVM in terms of a hard margin SVM with a modified kernel means that, unlike the L_1-loss SVM, the L_2-loss SVM can always be interpreted as a hard margin problem.

The relationship between L_2-loss SVMs and hard margin SVMs means that L_2-loss SVMs are often used in order to apply hard margin methods to inseparable datasets. For example, Kowalczyk [71] and Keerthi et al. [65] exploit the conceptual simplicity of the hard margin SVM task in order to propose intuitive geometric training algorithms. They then take advantage of the kernel in Equation (2.33) in order to allow the same algorithms to train L_2-loss SVMs.

2.6 Weighted SVMs

In many classification tasks it is beneficial to be able to specify a trade-off between the accuracy in the positive class (sensitivity) and the accuracy in the negative class (specificity). Being able to tune this trade-off provides the ability to sacrifice accuracy in one class in order to achieve a greater accuracy in another. For example, in spam classification, it is much more important that legitimate email is correctly classified than it is that spam is

correctly classified. In medical applications, a false positive result will undoubtedly have a different outcome than a false negative, and the balance needs to be chosen carefully.

The issue of the sensitivity-specificity trade-off is compounded by the fact that many of the applications which require this trade-off are also grossly imbalanced in terms of the number of points that appear in each class. For example, some people may receive significantly more spam than legitimate email. Similarly, a positive diagnosis with an exotic disease will be rare compared to a negative diagnosis. These imbalances tend to cause classifiers which aim to minimize overall classification error to favor the larger class. The bias occurs because, provided one class is large enough, often the best *overall* accuracy can be achieved by simply classifying everything as belonging to the larger class.

A case related the sensitivity-specificity trade-off occurs when misclassification costs can differ not only between classes, but between points. Zadrozny et al. [129] uses the example of donor solicitation, where the aim is for a charity to classify people as donors or non-donors. If a person is incorrectly classified as a donor, a small loss is incurred, e.g. the cost of a stamp. However, if a person is incorrectly classified as a non-donor, that person's donation will never be received. Donation sizes could range from several times the cost of a stamp to several *thousand* times the cost of a stamp.

In the context of SVMs, all of these related cases – the sensitivity-specificity trade-off, imbalanced class sizes and, and cost-proportionate tasks – have been addressed by via the introduction of *weighted* training data [129, 118]. In this instance, a training set consists not just of points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ and their associated labels $y_1, y_2, \ldots, y_n \in \{-1, 1\}$, but also a set of associated weights $s_1, s_2, \ldots, s_n$ satisfying $s_i \geq 0$ for all i. Each weight s_i specifies the importance of a particular training point. Note that weighted classification is not unique to SVMs, indeed many classifiers allow weighted training data to be adopted in a natural manner [129].

Weighted SVMs are a generalized form of SVMs where additional parameters are introduced. SVMs can be weighted either by class or by individual point. In a class-weighted SVM, each class is given its own specific weight [118]. In a point-weighted SVM, each individual point is given a separate weight [129]. We discuss the implications of these weighting schemes in the following sections.

2.6.1 Class Weighting

One of the earliest introductions of a WSVM was by Veropoulos et al. [118], who introduced weights in order to balance the trade-off between the sensitivity and specificity of an SVM. Because Veropoulos et al. were mainly interested in compensating for imbalanced classes, weights were assigned to each class, rather than to each individual point. This was expressed in primal form as the modified optimization problem:

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & ||\mathbf{w}||^2 + C^+ \sum_{i \in I_{pos}} \xi_i + C^- \sum_{i \in I_{neg}} \xi_i, \\ \text{subject to} \quad & \begin{cases} y_i(\mathbf{w} \cdot \mathbf{x}_i - b) & \geq 1 - \xi_i \\ \xi_i & \geq 0. \end{cases} \end{aligned} \tag{2.34}$$

Recall that I_{pos} and I_{neg} are sets containing the indices of points belonging to the positive and negative classes, respectively. Notice that this primal uses two regularization parameters (C^+, C^-), both of which must be set by the user prior to training. Providing two regularization parameters allows the misclassification penalty of each class to be set individually.

Veropoulos et al. [118] suggest that their method provides an effective means of balancing the sensitivity and specificity of an SVM. Importantly, it also results in a very simple optimization problem, no more difficult to solve than that of the standard SVM. This can be seen by deriving the dual form of Equation (2.34), written [118]:

$$\begin{aligned} \max_{\alpha} \quad & -\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \mathbf{x}_j + \sum_i \alpha_i. \\ \text{subject to} \quad & \begin{cases} 0 \leq \alpha_i \leq C_i \\ \sum_{i=1}^{n} \alpha_i y_i = 0 \end{cases} \end{aligned} \tag{2.35}$$

Here C_i is given by:

$$C_i = \begin{cases} C^+ & \text{if } i \in I_{pos} \\ C^- & \text{if } i \in I_{neg} \end{cases}.$$

Notice that the only difference between Equation (2.35) and the standard SVM is the change in upper bound on α_i values.

2.6.2 Individual Point Weighting

The natural generalization of Veropoulos et al.'s [118] work is to weight every individual point separately, yielding the primal:

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & ||\mathbf{w}||^2 + C\sum_{i \in I_{pos}} s_i \xi_i, \\ \text{subject to} \quad & \begin{cases} y_i(\mathbf{w}\cdot\mathbf{x}_i - b) & \geq 1 - \xi_i \\ \xi_i & \geq 0. \end{cases} \end{aligned} \tag{2.36}$$

Here each point has a unique weighting s_i which determines its importance. As was the case for the class-weighted version of this optimization problem (Equation 2.34), the only change in the dual is in the upper bounds on α_i values [118]. Solving the weighted dual generally requires such minimal changes to optimization routines that some popular SVM solver packages such as `LIBSVM` [20] feature the ability to solve weighted problems such as (Equations 2.34 and 2.36).

Zadrozny et al. [129] showed using empirical trials that individual point weighting provide an effective means of solving problems where misclassification costs can vary greatly. The example used by Zadrozny et al. was charity collection, where the cost of soliciting donations must be weighed against the return. In this case large donors were rare, but

provided such a large return that it was beneficial to assign them a greater weighting compared to other training points. The introduction of individual cost-proportionate weights on this problem was beneficial, providing a significant monetary return. By contrast, standard SVMs failed to provide any monetary return at all, opting to classify everyone as belonging to the significantly larger non-donor class.

2.6.3 Alternative Loss Functions

Although the methods we refer to for weighted SVM classification generally use the L_1-loss function, it is no more difficult to substitute the L_2-loss function. For example, Veropoulos et al. [118] has shown that adapting the L_1-loss WSVM to penalize squared errors results in the dual:

$$\begin{aligned} \max_{\alpha} \quad & -\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \left[K(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij} \frac{1}{s_i 2C} \right] \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i = 2, \\ 0 \leq \alpha_i \leq 1. \end{cases} \end{aligned} \tag{2.37}$$

Conveniently, the interpretation of a convex hull problem in a kernel feature space remains unchanged. The only modification from the unweighted L_2-loss SVM is that, instead of adding a constant $1/C$ to the diagonals of the kernel matrix, $1/(s_i C)$ is added. To implement class-based penalties, $1/C_+$ would be added to the diagonal elements of the kernel matrix associated with points from the positive class, while $1/C_-$ would be added to other diagonal elements [118].

2.7 Minimal Enclosing Balls (MEBs)

The Minimal Enclosing Ball (MEB) of n points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ is the minimum volume sphere enclosing all points [116]. Recall from Section 2.2.2 that MEBs can be used to provide an upper bound on the leave-one-out error of an SVM [117, 103]. They have also been used to perform clustering [6] as well as one-class domain description [110, 104]. Because the volume of a sphere depends solely on its radius, an MEB can be found by representing the ball by its center $\mathbf{c}$ and radius R and then solving [116]:

$$\begin{aligned} \min_{R,\mathbf{c}} \quad & R^2 \\ \text{subject to} \quad & ||\mathbf{x}_i - \mathbf{c}||^2 \leq R^2. \end{aligned} \tag{2.38}$$

Equation (2.38) is a QP task, the aim of which is to minimize the radius of a ball while constraining it to enclose all points.

2.7.1 The MEB Dual

The MEB dual [17, 103, 116] is derived by introducing Lagrange multipliers $\alpha_i, \ldots, \alpha_n \geq 0$ for each inequality constraint in (2.38), defining a Lagrangian:

$$\mathcal{L} = R^2 - \sum_{i=1}^{n} \alpha_i(-\mathbf{x}_i^2 + 2\mathbf{c} \cdot \mathbf{x}_i - \mathbf{c}^2 + R^2).$$

The objective function in (2.38) can only be at a minimum when the partial derivatives of the Lagrangian with respect to R and $\mathbf{c}$ are equal to zero, yielding the equalities:

$$\frac{\partial \mathcal{L}}{\partial R} = 2R - 2R\sum_{i=1}^{n} \alpha_i = 0 \qquad \Rightarrow \sum_{i=1}^{n} \alpha_i = 1, \tag{2.39}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}} = \sum_{i=1}^{n} \alpha_i(2\mathbf{x}_i^\intercal - 2\mathbf{c}^\intercal) = 0 \qquad \Rightarrow \mathbf{c} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i. \tag{2.40}$$

Further conditions for complementary slackness are:

$$\alpha_i(||\mathbf{x}_i - \mathbf{c}||^2 - R^2) = 0. \tag{2.41}$$

Substituting the equalities in (2.39) and (2.40) into the Lagrangian results in the dual optimization problem [116]:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i \mathbf{x}_i^2 - \sum_{i,j=1}^{n} \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i & = 1, \\ \alpha_i & \geq 0 \end{cases} \end{aligned} \tag{2.42}$$

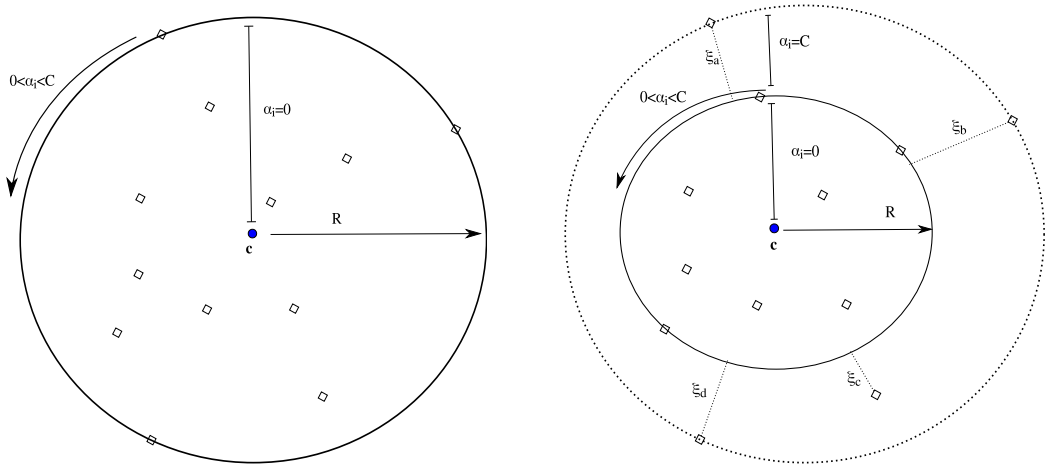
The complementary slackness condition in (2.41) also reveals some interesting geometric significance in the values of the Lagrange multipliers. For any α_i, either α_i must equal zero, or the corresponding $\mathbf{x}_i$ must lie on the surface of the MEB. This means that only points lying on the surface of the MEB will have a non-zero α_i, indicating that the solution to Equation (2.42) will in most cases be sparse [128].

2.7.2 Soft MEBs

When MEBs are used for domain description, it is beneficial to be able to reduce an MEB in order to exclude noisy or outlying points. Tax and Duin [110] suggest the modified primal:

$$\begin{aligned} \min_{R,\mathbf{c}} \quad & R^2 + C\sum_{i=1}^{n} \xi_i \\ \text{subject to} \quad & \begin{cases} ||\mathbf{x}_i - \mathbf{c}||^2 \leq R^2 + \xi_i \\ \xi_i \geq 0 \end{cases} \end{aligned} \tag{2.43}$$

We refer to Equation (2.43) as the *soft* MEB primal. The move from standard to soft MEB is very similar to the move from hard margin to soft margin SVM.



(a) KKT conditions associated with the hard margin MEB

(b) KKT conditions associated with the soft margin MEB

Figure 2.10: KKT conditions associated with the hard and soft MEB tasks

Although there may be many ways to reduce an MEB, one of the benefits of the soft MEB is that it results in a convenient dual:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i \mathbf{x}_i^2 - \sum_{i,j=1}^{n} \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i = 1, \\ 0 \leq \alpha_i \leq C. \end{cases} \end{aligned} \tag{2.44}$$

Notice that the only difference between the hard and soft MEB duals is the introduction of the upper bound C on the α_i's in the soft margin dual. This change is analogous to the introduction of same bound when moving from the hard margin to soft margin SVM.

The benefit of soft MEBs is that they allow the impact of outlying points to be reduced. By contrast, hard MEBs are forced to enclose all points, even outlying points, so a single outlying point can result in a very large MEB.

2.8 Perceptrons

Perceptron algorithms were first introduced by Rosenblatt [97] as a computational model of a biological neural network. Several decades later, after the introduction of learning theory and the idea of large margins, it was pointed out that perceptrons implement a type of large margin classifier [43, 62]. Perceptron algorithms have also been implemented in kernel feature spaces [43], with the resulting algorithms *similar* (although not equivalent) to SVMs. In this section we review several perceptron algorithms. We also describe the relationship between SVMs and perceptrons, and describe how we distinguish between the two types of machines.

2.8.1 Rosenblatt's Perceptron

One of the first algorithms for training a large margin linear classifier is Rosenblatt's *perceptron* algorithm [97, 98]. This algorithm pre-dates learning theory and work on large margins by several decades. The perceptron was proposed by Rosenblatt, a psychologist, as a computational model of a biological neural network. The perceptron algorithm is described in Algorithm 1.

Algorithm 1 Rosenblatt's Perceptron [97]

function PERCEPTRON($P, \mathbf{y}$)
  initialize $\alpha_i \leftarrow 0$ for all $i = 1, \ldots, n$
  let $f(\mathbf{x}) = \sum_i \alpha_i \mathbf{x}_i \cdot \mathbf{x}$
  repeat
    choose k such that $y_i f(\mathbf{x}_k) <= 0$
    update $\alpha_i \leftarrow \alpha_i + y_i$
  until $y_i f(\mathbf{x}_i) > 0$ for all $i = 1, 2, \ldots, n$
  $\mathbf{w} \leftarrow \sum_i \alpha_i \mathbf{x}_i$
  return w ▷ return the hyperplane normal
end function

The perceptron algorithm (Algorithm 1) does not include a bias term, meaning that hyperplane found by the algorithm is constrained to pass through the origin. If this is not desired, Rosenblatt's perceptron can optionally be modified so that it contains a bias term. This is done by mapping the training data prior to training so that [52]:

$$\mathbf{z}_i \leftarrow [1 \ \mathbf{x}_i].$$

This results in a decision function:

$$f(\mathbf{x}) = \sum_i \alpha_i \mathbf{z}_i \mathbf{x} = \sum_i \alpha_i \mathbf{x}_i + \sum_i \alpha_i.$$

Note that this is equivalent to the standard SVM decision function $f(\mathbf{x}) = \sum_i \alpha'_i y_i \mathbf{x}_i - b$, provided we use $\alpha'_i = |\alpha_i|$ and $b = -\sum_i y_i \alpha'_i$.

2.8.2 AdaTron Algorithm

Anlauf and Biehl [3] introduced the AdaTron algorithm as a means of finding perceptrons of optimal stability. A perceptron of optimal stability Δ satisfies $y_i f(\mathbf{x}_i) \geq \Delta$ [73]. This means that the AdaTron algorithm does not simply terminate once all training points are correctly classified (like Rosenblatt's perceptron), but continues training to a much higher degree of precision (Algorithm 2). The algorithm also has the beneficial property of converging towards the maximum optimal stability, i.e. the largest possible Δ [3]. An alternative stopping condition to the one shown in Algorithm 2 is to run until changes to $\min_i\{y_i f(\mathbf{x}_i)\}$ become very small.

The parameter $\eta > 0$ is the learning rate and affects the rate of convergence. Points can be updated either sequentially or in parallel [3]. In a parallel implementation, all δ_k's

Algorithm 2 The AdaTron Algorithm [3]

function ADATRON($P, \mathbf{y}, \eta, \Delta$)
  initialize $\alpha_i \leftarrow 1$ for all $i = 1, \ldots, n$
  let $f(\mathbf{x}) = \sum_i \alpha_i \mathbf{x}_i \cdot \mathbf{x}$
  repeat
    choose point $\mathbf{x}_k$ to update ▷ can iterate through points
    $\delta_k \leftarrow \max(-\alpha_k, \eta(1 - y_i f(\mathbf{x}_i)))$ ▷ sequentially or in parallel (see below)
    $\alpha_k \leftarrow \alpha_k + \delta_k$
  until $y_i f(\mathbf{x}_i) \geq \Delta,\ \forall k$ ▷ stop once stability Δ is reached
  $\mathbf{w} \leftarrow \sum_i \alpha_i \mathbf{x}_i$
  return w ▷ return hyperplane normal
end function

are computed in parallel before α_i's are updated. In a sequential implementation, once a single δ_k is computed, the corresponding α_i is updated before any more steps are taken. Anlauf and Biehl [3] suggest that the sequential algorithm is generally preferred since it is guaranteed to converge for all $0 < \eta < 2$, whereas the parallel algorithm sometimes requires small values of η to ensure convergence.

In addition to increased precision and stability over Rosenblatt's algorithm, the AdaTron algorithm also provides a mechanism with which non-zero α_i values can return to zero if necessary [3]. This results in increased sparsity during training, as well as faster convergence if a large optimal stability Δ is required.

2.8.3 Perceptrons as a Quadratic Programming Task

It has been noted by several authors [3, 87] that finding a perceptron with maximum optimal stability can be stated as a QP task:

$$\begin{aligned} &\min \frac{1}{2}||\mathbf{w}||^2 \\ \text{subject to} \quad & y_i \mathbf{w} \cdot \mathbf{x}_i \geq 1 \qquad \forall i \end{aligned} \tag{2.45}$$

Note that the optimal stability term Δ from the AdaTron is also maximized and hence no longer appears in the QP task. Although this QP task pre-dates SVMs, it resembles a hard margin SVM with the bias term omitted. The omission of b results in the dual:

$$\begin{aligned} \max_{\alpha_1,\ldots,\alpha_n} \quad & -\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i, \\ \text{subject to} \quad & \alpha_i \geq 0. \end{aligned} \tag{2.46}$$

Notice that this dual can not be trained using standard hard margin optimization techniques since the sum of α_i's is no longer constraint to unity. However, it can be trained using modified SMO techniques [119]. Because the constraint on the sum of $\alpha_i's$ has been omitted from this task, SMO can operate on a single α_i per update step. In fact, such

update steps are equivalent to the AdaTron update step [62], meaning the AdaTron algorithm can be improved by using a heuristic to choose the points to update at each step [54].

2.8.4 Perceptrons with Soft Margins

Perceptrons can be trained with a soft margin by introducing slack variables ξ_i to the primal. Let us first examine the use of an L_1-loss penalty, which forms the soft margin perceptron primal:

$$\begin{aligned} &\min \frac{1}{2}||\mathbf{w}||^2 + C\sum_i \xi_i \\ &\text{subject to} \qquad y_i \mathbf{w}\cdot\mathbf{x}_i \geq 1-\xi_i \qquad \forall i \end{aligned} \tag{2.47}$$

The dual of Equation (2.47) is written:

$$\begin{aligned} &\max_{\alpha_1,\ldots,\alpha_n} \qquad -\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i\cdot\mathbf{x}_j + \sum_{i=1}^{n}\alpha_i, \\ &\text{subject to} \qquad 0 \leq \alpha_i \leq C. \end{aligned} \tag{2.48}$$

It is informative to note that, like SVMs, the introduction of an L_1-loss penalty term simply changes the constraint on α_i's in the dual.

If instead of an L_1-loss function we applied an L_2-loss function to (2.47), the resulting dual would be:

$$\begin{aligned} &\max_{\alpha_1,\ldots,\alpha_n} \qquad -\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i\cdot\mathbf{x}_j - \frac{1}{2}\sum_{i,j=1}^{n}\frac{\alpha_i^2}{C} + \sum_{i=1}^{n}\alpha_i, \\ &\text{subject to} \qquad 0 \leq \alpha_i \end{aligned} \tag{2.49}$$

Notice how, as in the case of SVMs, this dual is equivalent to a hard margin perceptron with the modified kernel $k(\mathbf{x}_i,\mathbf{x}_j) = \delta_{ij}/C$, where δ_{ij} is the Kronecker delta (refer to Section 2.5.4).

2.8.5 Perceptrons with a Bias Term

One of the downsides to the hard margin perceptron above is that it omits a bias term. This forces the hyperplane to pass through the origin, which may not be a desired property. For this reason, distinct methods have been developed for providing perceptrons with a bias term. Perceptrons with bias use the mapping:

$$\mathbf{x}_i \leftarrow [1\ \mathbf{x}_i]. \tag{2.50}$$

Substituting this mapping into the L_1-loss perceptron dual in Equation (2.48) results in the optimization task:

$$\begin{aligned} \max_{\alpha_1,\ldots,\alpha_n} \quad & -\frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j - \frac{1}{2}\sum_{i,j} y_i y_j \alpha_i \alpha_j + \sum_{i=1}^{n} \alpha_i, \\ \text{subject to} \quad & \alpha_i \geq 0. \end{aligned} \tag{2.51}$$

For a perceptron with bias, the classification function is given by:

$$f(\mathbf{x}) = \sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} - b \tag{2.52}$$

where, by combining Equations (2.50) and (2.52), the bias can be given by:

$$b = -\sum_i y_i \alpha_i$$

The primal associated with (2.51) is given by:

$$\begin{aligned} \min \frac{1}{2}||\mathbf{w}||^2 + \frac{1}{2}b^2 + C\sum_i \xi_i \\ \text{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1 - \xi_i \quad \forall i \end{aligned} \tag{2.53}$$

The existence of the term b in the objective function is difficult to justify, since all it does is favor a hyperplane closer to the origin. However, it is the addition of this term that results in the perceptron having no constraint forcing the sum of Lagrange multipliers in each class to be equal [65]. This suggests that the bias term may be added to the objective function more for numerical convenience than for theoretical justification. However, Mangasarian and Musicant [79] have suggested that the addition of this term does not significantly degrade accuracy compared to an SVM, where the bias term does not appear in the objective function.

2.8.6 Distinguishing Perceptrons from SVMs

It may seem difficult to identify differences between perceptrons and SVMs, particularly when accounting for the various number of different loss functions and the impact they have on the dual. It is tempting to compare the hard margin SVM and the hard margin perceptron and say that a perceptron is an SVM without a bias term. However, this distinction becomes muddled once perceptrons with bias are considered.

We assert that there is a more rigorous way to identify whether a machine is a perceptron. Notice that the hard margin perceptron in Equation (2.46), the perceptron with bias in Equation (2.51), and the L_1 and L_2-loss soft margin perceptrons in Equations (2.48)

and (2.49) all have the general dual form:

$$\begin{aligned} \max_{\alpha_1,\ldots,\alpha_n} \quad & -\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j k(\mathbf{x}_i,\mathbf{x}_j)+\sum_{i=1}^{n}\alpha_i, \\ \text{subject to} \quad & 0\leq\alpha_i\leq C \end{aligned} \tag{2.54}$$

The type of perceptron trained by (2.54) depends on the choice of the modified kernel $k(\mathbf{x}_i,\mathbf{x}_j)$ and the upper bound C on the Lagrange multipliers. Possible values are shown in Table 2.2. Notice from this table that a bias term can be combined with soft margins.

Table 2.2: Alternative Perceptron Loss Functions. Note that a 'B' under type refers to bias-free (i.e. no bias term b is used)

Type	**Constraint**	**Kernel** $k(\mathbf{x}_i,\mathbf{x}_j)$	**Threshold**
Hard Margin	$0\leq\alpha_i$	$y_iy_jK(\mathbf{x}_i,\mathbf{x}_j)+y_iy_j$	$-\sum_i\alpha_iy_i$
Hard Margin B	$0\leq\alpha_i$	$y_iy_jK(\mathbf{x}_i,\mathbf{x}_j)$	0
L_1-loss	$0\leq\alpha_i\leq C$	$y_iy_jK(\mathbf{x}_i,\mathbf{x}_j)+y_iy_j$	$-\sum_i\alpha_iy_i$
L_1-loss B	$0\leq\alpha_i\leq C$	$y_iy_jK(\mathbf{x}_i,\mathbf{x}_j)$	0
L_2-loss	$0\leq\alpha_i$	$y_iy_jK(\mathbf{x}_i,\mathbf{x}_j)+y_iy_j+\delta_{ij}/D$	$-\sum_i\alpha_iy_i$
L_2-loss B	$0\leq\alpha_i$	$y_iy_jK(\mathbf{x}_i,\mathbf{x}_j)+\delta_{ij}/D$	0

There is also a similar general case for SVMs, as has previously been described by Chu et al. [23]. The hard margin SVM, and both L_1 and L_2-loss soft margin SVMs are able to be trained from the dual [23]:

$$\begin{aligned} \max_{\alpha_1,\ldots,\alpha_n} \quad & -\frac{1}{2}\sum_{i,j=1}^{n}y_iy_j\alpha_i\alpha_j k(\mathbf{x}_i,\mathbf{x}_j)+\sum_{i=1}^{n}\alpha_i, \\ \text{subject to} \quad & \begin{cases}\sum_{i=1}^{n}\alpha_iy_i=0\\ 0\leq\alpha_i\leq C\end{cases} \end{aligned} \tag{2.55}$$

The kernels which can be used in conjunction with this optimization task are given in Table 2.3.

Table 2.3: Alternative SVM Loss Functions. All machines have a bias term.

Type	**Constraint**	**Kernel** $k(\mathbf{x}_i,\mathbf{x}_j)$
Hard Margin	$0\leq\alpha_i$	$K(\mathbf{x}_i,\mathbf{x}_j)$
L_1-loss	$0\leq\alpha_i\leq C$	$K(\mathbf{x}_i,\mathbf{x}_j)$
L_2-loss	$0\leq\alpha_i$	$K(\mathbf{x}_i,\mathbf{x}_j)+\delta_{ij}/D$

A comparison of the general SVM and perceptron duals in Equations (2.54) and (2.55) reveals that the main difference between the two machines is that an SVM constrains the sum of Lagrange multipliers associated with the positive class to equal the sum of Lagrange multipliers associated with the negative class. This constraint is enforced by the equality:

$$\sum_i\alpha_iy_i=0. \tag{2.56}$$

This constraint does not exist in the perceptron dual, and so the sum of Lagrange multipliers in each class may differ.

Recall from Section 2.5 that it is the constraint in (2.56) that allows SVMs to be interpreted as a nearest point problem over two RCHs. The lack of this constraint in the perceptron dual means that perceptrons can not be interpreted as the perpendicular bisector of the shortest line between the RCHs of the two classes. However, perceptrons do have an alternative geometric interpretation which we will discuss further in Chapter 4.

In this thesis we define perceptrons as a *family* of classifiers. Perceptrons include all classifiers derived from the general perceptron QP task in Equation (2.54). This includes hard and soft margin perceptrons both with and without bias. The distinguishing feature of the perceptron family is that the sum of weights in each class is *not* constrained to be equal. The lack of this constraint means that, unlike SVMs, they *cannot* be solved as a nearest point problem over two RCHs.

We also define SVMs as a family of classifiers. SVMs include all classifiers defined from the general SVM QP task in Equation (2.55). This includes hard and soft margin SVMs, as well as their equivalent geometric reparameterizations. The distinguishing feature of SVMs is that they may all be solved as a nearest point problem over the reduced convex hulls of the two classes. Unlike perceptrons, the sum of Lagrange multipliers in each class of an SVM is constrained to equality.

Although we assert that there is a clear way to distinguish between SVMs and perceptrons, we should emphasize that our conventions are not necessarily adhered to in the literature. Because of the similarity between SVMs and perceptrons, there has been some overlap in the use of the terms 'SVM' and 'perceptron'. For example, Mangasarian and Musicant [79] call the L_2-loss perceptron a Lagrangian Support Vector Machine (LSVM). Tsang et al. [114] solves the same L_2-loss perceptron task approximately, and calls the resulting machine a Core Vector Machine (CVM). On the other hand, Kowalczyk [71] describes an algorithm for solving the L_2-loss SVM dual. However, due to the algorithm's similarity with classical perceptron algorithms, it is often referred to as a perceptron algorithm [71, 68].

Chapter 3

Reduced Convex Hulls

3.1 Introduction

In the previous chapter we reviewed the geometric interpretation of SVMs, as given by Crisp and Burges [28] and Bennett and Bredensteiner [7]. The geometric interpretation arises via the introduction of the reduced convex hull. A reduced convex hull is a type of convex hull where the influence any single point can have on the hull is bounded from above by a constant. Although reduced convex hulls are well defined as a stand-alone concept, most research has focused on using them solely for the purpose of training [40, 109] or understanding [28, 7] SVMs. By contrast, convex hulls and their applications have been the study of a significant amount of research [94, 89, 5].

In this chapter we investigate further some of the theoretical properties of reduced convex hulls, allowing us to construct several algorithms for computing reduced convex hulls in their entirety. This allows us to visualize RCHs and explore several of their properties in detail.

We also generalize the concept of RCHs to introduce the *Weighted* RCH (WRCH). A WRCH allows the impact each point has on the hull to be specified individually. We describe several properties of WRCHs, including how their vertices may be found. This allows us to extend our RCH algorithms to compute WRCHs. The introduction of WRCHs is important because they have a close relationship to weighted SVMs. In later chapters we will elaborate on this relationship, and show how WRCHs can be used in conjunction with nearest point algorithms in order to train weighted SVMs.

3.2 Background

This section describes background information on convex hulls, their properties, and how they may be computed. Because convex hulls are so closely related to reduced convex hulls, this information provides a useful foundation for later sections, in which we examine some of the theoretical properties of reduced convex hulls.

3.2.1 Convex Hulls

The convex hull of a set of n d-dimensional points $P = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \in \mathbb{R}^d$ is defined as

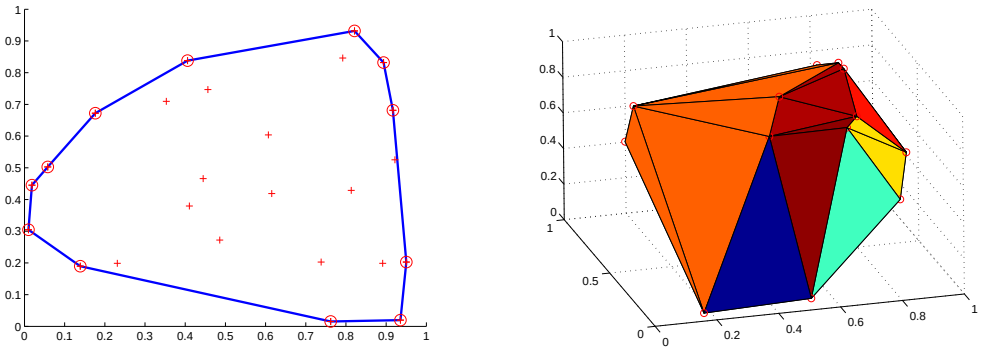
$$\mathrm{CH}(P) = \left\{ \sum_i^n \alpha_i x_i \;\middle|\; \sum_i^n \alpha_i = 1, \quad 0 \leq \alpha_i \leq 1 \right\}.$$

The convex hull of P is the smallest convex set enclosing P or, equivalently, the set of all convex combinations of points in P. $\mathrm{CH}(P)$ forms a solid region, the border of which is a convex polytope.

Convex hulls are simplest to describe and construct under the assumption that the points in P are in *general position*. General position means that, for points in d-dimensional space, there exists no $d+1$ points lying in a $(d-1)$-dimensional plane. To simplify some of the reasoning in this chapter, we will assume that the points we are dealing with are in general position, unless stated otherwise.

When points are in general position, the border of $\mathrm{CH}(P)$ is a simplicial polytope [94]. This means that for $P \in \mathbb{R}^d$, all facets of the border are $(d-1)$-simplices and more complex facets do not need to be considered. For example in two dimensions, facets are 1-simplices (lines) (Figure 3.1a). In three dimensions, facets are 2-simplices (triangles) (Figure 3.1b).

For a simplicial polytope in $\mathbb{R}^d$, each $(d-1)$-simplicial facet is itself a simplicial polytope with a number of $(d-2)$-simplicial *subfacets* or *ridges*. Each ridge is shared by two adjoining facets. This relationship continues down to 1-simplices and finally 0-simplices, which are generally referred to as *edges* and *vertices* respectively, regardless of the dimensionality of the polytope.



(a) Convex hull in two dimensions

(b) Convex hull in three dimensions

Figure 3.1: Convex hulls in two and three dimensions. Extreme points (vertices) are circled

The vertices (or extreme points) of a convex hull (shown circled in Figure 3.1) are the points from P which are on the outside of the hull. A point $\mathbf{x}_k \in P$ forms a vertex of the convex hull if there exists an $\hat{\mathbf{n}}$ such that:

$$\mathbf{x}_k \cdot \hat{\mathbf{n}} > \mathbf{x}_i \cdot \hat{\mathbf{n}}, \qquad \forall i \neq k.$$

This means that for a point to be a vertex it must have a scalar projection onto some direction $\hat{\mathbf{n}}$ which is greater than that for any other point in P. Note that since $\text{CH}(P)$ is a convex polytope, all points from P which are inside the hull can be formed as convex combinations of the vertices so they do not contribute to the hull.

We often refer to the *normal* of a facet. The normal of a facet is a vector which is perpendicular to the facet. In a d-dimensional input space, facet normals will have d components. Note that any facet has two possible normals, one pointing away from the hull, and one pointing towards the center of the hull. For consistency, whenever we refer to the normal of a facet, we are referring to the normal that points *away* from the center of the hull.

We also rely on the notion of a point being above or below a facet. A point $\mathbf{p}$ is *below* a facet if the facet is not 'visible' from $\mathbf{p}$. More formally, the point $\mathbf{p}$ is below a facet $\mathbf{f}$ if:

$$\hat{\mathbf{n}} \cdot \mathbf{p} < \hat{\mathbf{n}} \cdot \mathbf{v},$$

where $\hat{\mathbf{n}}$ is the normal of the facet, and $\mathbf{v}$ is any vertex of $\mathbf{f}$. Because all vertices of a facet must have an equal scalar projection onto the facet's normal, it does not matter which vertex of $\mathbf{f}$ is used for $\mathbf{v}$. This concept is illustrated in Figure 3.2.

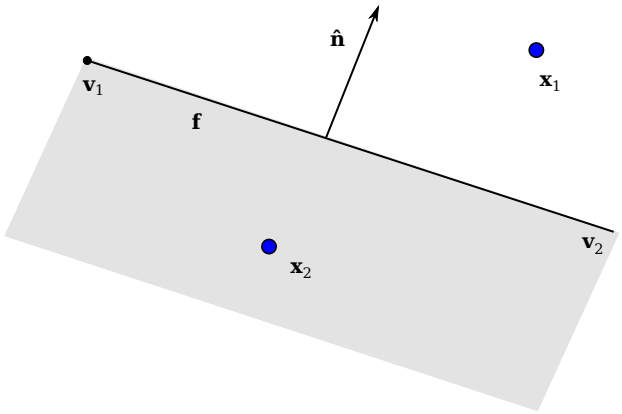


Figure 3.2: Here any point in the shaded region is *below* the facet $\mathbf{f}$ (with normal $\hat{\mathbf{n}}$ and vertices $\mathbf{v}_1, \mathbf{v}_2$). For example, the point $\mathbf{x}_2$ is below the facet $\mathbf{f}$, whereas $\mathbf{x}_1$ is above it

3.2.2 Algorithms for Computing Convex Hulls

The purpose of an algorithm which computes a convex hull is to calculate the facets bordering the hull. For a set of points $P \in \mathbb{R}^d$, each facet consists of d vertices from P. This means the output of a program computing a convex hull is a list of d-vectors, each defining a single facet. Each element of a d-vector specifies a vertex $\mathbf{x}_i \in P$ in terms of its index i.

An ideal representation for a list of fixed length vectors is a matrix. Accordingly, a convex hull algorithm can output a hull as an $n_f \times d$ matrix, where n_f is the number of facets. Each row of the matrix is a d-vector which defines a single facet in terms of its d vertices. For example, for $d = 3$ (points in three dimensions) a convex hull representation

might look like the following matrix:

$$F = \begin{bmatrix} 1 & 2 & 7 \\ 2 & 7 & 9 \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.1}$$

This matrix indicates that the points $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_7$ form a facet, as do the points $\mathbf{x}_7, \mathbf{x}_5, \mathbf{x}_3$. There are additional rows for each of the facets in the convex hull.

Optionally, an algorithm may also compute an adjacency list specifying the facets in a hull which are adjoining. In d dimensions, each facet will have d adjoining neighbors, so an adjacency list can be represented using a matrix with the same dimensions as the one defining the facets. For example, for the facets given in (3.1), an adjacency list might look like:

$$A = \begin{bmatrix} 2 & 3 & 7 \\ 1 & 4 & 12 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

This matrix indicates that the first facet (i.e. the facet corresponding to the first row of F) adjoins the second, third and seventh facets (rows) of F.

Because the convex hull of a set of points in $\mathbb{R}^d$ consists of a number of $(d-1)$-simplicial polytopes, each facet will adjoin d other facets along a ridge. It is not strictly necessary for an algorithm to return an adjacency list since one may be calculated given a list of facets. However, it is generally faster to calculate during the construction of a hull rather than after. Some algorithms, such as the Beneath-Beyond algorithm which we describe below, can also speed up the construction process itself by maintaining an adjacency list.

The Quickhull Algorithm

The Quickhull algorithm [35, 94] is a method of computing convex hulls in the plane. The algorithm (Algorithm 3) takes a divide and conquer approach similar to that taken by its namesake, the Quicksort algorithm. The algorithm takes advantage of the observation that, given a facet of the hull, the point with the largest scalar projection onto the normal of that facet must be a vertex in the convex hull. This allows the facets of a convex hull to be recursively split until all the facets have been constructed. Furthermore, points which fall below a facet can be discarded in recursive calls, meaning the algorithm can take a divide and conquer approach to finding the convex hull [94].

Initially, the Quickhull algorithm takes as input the set of points P. It then constructs two initial vertices from the hull, $\mathbf{l}$ and $\mathbf{r}$. Any two unique initial vertices will work, however convenient initial vertices are those which are most extreme along the x-axis, i.e. the points satisfying:

$$\mathbf{l} = \arg\max_{\mathbf{l} \in P} \{-\hat{\mathbf{n}} \cdot \mathbf{l}\} \qquad \mathbf{r} = \arg\max_{\mathbf{r} \in P} \{\hat{\mathbf{n}} \cdot \mathbf{r}\}$$

Algorithm 3 Quickhull algorithm [94, p114].

function QH(P)
  $\mathbf{n} \leftarrow (1, 0)$ ▷ initial vertices must be extreme in any direction
  $\mathbf{l} \leftarrow \arg\max_{\mathbf{x} \in P} -\mathbf{n} \cdot \mathbf{x}$ ▷ first initial vertex
  $\mathbf{r} \leftarrow \arg\max_{\mathbf{x} \in P} \mathbf{n} \cdot \mathbf{x}$ ▷ second initial vertex
  return qh_helper($P, \mathbf{l}, \mathbf{r}$) ∪ qh_helper($P, \mathbf{r}, \mathbf{l}$)
end function

function QH_HELPER($P, \mathbf{l}, \mathbf{r}$)
  if $P = \{\mathbf{l}, \mathbf{r}\}$ **then**
    return $\{\mathbf{l}, \mathbf{r}\}$ ▷ this facet can not be split, return final vertices
  else
    $\mathbf{n} \leftarrow \text{normal}(\mathbf{r} - \mathbf{l})$ ▷ normal of old facet
    $\mathbf{h} \leftarrow \arg\max_{\mathbf{x} \in P} \mathbf{n} \cdot \mathbf{x}$ ▷ a new vertex
    $\mathbf{n}_r \leftarrow \text{normal}(\mathbf{r} - \mathbf{h})$ ▷ normal of first new facet
    $\mathbf{n}_l \leftarrow \text{normal}(\mathbf{h} - \mathbf{l})$ ▷ normal of second new facet
    $L \leftarrow$ the set of all points $\mathbf{x} \in P$ satisfying $\mathbf{n}_l \cdot \mathbf{x} \geq \mathbf{n}_l \cdot \mathbf{h}$
    $R \leftarrow$ the set of all points $\mathbf{x} \in P$ satisfying $\mathbf{n}_r \cdot \mathbf{x} \geq \mathbf{n}_r \cdot \mathbf{h}$
    return qh_helper($L, \mathbf{l}, \mathbf{h}$) ∪ qh_helper($R, \mathbf{h}, \mathbf{r}$)
  end if
end function

where $\hat{\mathbf{n}} = (1,\ 0)$. These initial vertices can be used to form two facets (with opposing normals), which are then passed to the recursive portion of the Quickhull algorithm.

Rather than returning a complete list of each facet and their vertices, the Quickhull algorithm exploits the fact that, in two dimensions, each facet has only two adjoining facets. Furthermore, adjoining facets share precisely one vertex. This means that a two-dimensional convex hull has a concise representation as an ordered list of vertices which is the same length as the number of facets. For example, if we were to use the matrix representation described in the previous section, the simple two-dimensional hull in Figure 3.3 would be represented by the Quickhull algorithm as a the vector:

$$F = \begin{bmatrix} 1 \\ 5 \\ 2 \\ 4 \end{bmatrix}$$

Notice how the last vertex in F adjoins the first, so any vertex may come first in the output, provided the neighboring vertices remain the same.

Unfortunately, the Quickhull approach is only possible in two dimensions where it is guaranteed that only a single facet will be visible to successive extreme points which are added to the hull. This property does not hold in higher dimensions where multiple facets may be visible to a new extreme point.

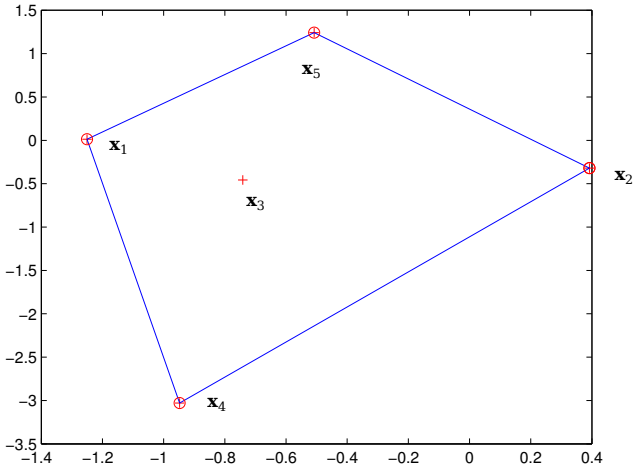


Figure 3.3: Convex hulls have a simplified representation in two dimensions

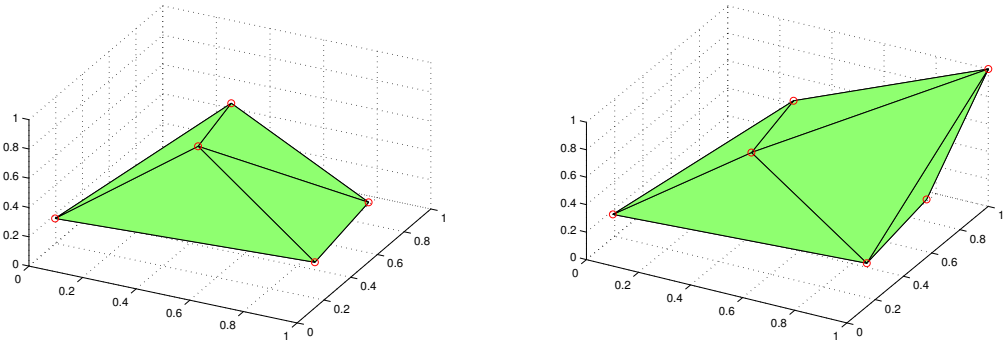
The Beneath-Beyond Algorithm

The Beneath-Beyond algorithm [61, 94] addresses the issue of convex hulls in higher dimensions. It works by iteratively applying Theorem 1 in order to build up a convex hull. Theorem 1 states that, given an existing convex hull and a new point, the convex hull can be updated to include the new point as follows. If the point is within the existing convex hull, no changes are required. Otherwise, any existing facets which are visible from the new point should be paved over with a 'cone' of new facets (Figure 3.4) [94].

A valid starting point for the Beneath-Beyond algorithm is to select any d points from the set of points P and create two facets (with opposing normals) passing through these points. Then, points can be iteratively added to the hull, one at a time, using Theorem 1, until all points have been added. As the algorithm progresses the initial points will likely be 'paved over' by new facets and vertices. The complete algorithm is described in detail in Algorithm 4.

Theorem 1 (Barber et al. [4])**.** *Let H be a convex hull in $\mathbb{R}^d$, and let $\mathbf{p}$ be a point in $\mathbb{R}^d - H$. Then $\mathbf{f}$ is a facet of the convex hull of $\mathbf{p} \cup H$ if and only if*

(a) $\mathbf{f}$ is a facet of H and $\mathbf{p}$ is below $\mathbf{f}$; or
(b) $\mathbf{f}$ is not a facet of H and its vertices are $\mathbf{p}$ and the vertices of a sub-facet of H with one incident facet below $\mathbf{p}$ and the other incident facet above $\mathbf{p}$.



(a) Existing convex hull

(b) Once a new vertex is added, visible facets are paved over

Figure 3.4: Updating a convex hull using the Beneath-Beyond Theorem

Algorithm 4 The Beneath-Beyond Algorithm

```
function BENEATH_BEYOND(P)
    F ← two facets with opposing normals passing through any d points in P
    for all x ∈ P do
        if x lies outside the hull then
            find all facets X which are visible from x
            for all boundary edges E in X do
                add a facet to F which joins x with the vertices in E
            end for
        end if
    end for
    return F                                   ▷ return the facets of the hull
end function
```

3.3 Reduced Convex Hulls

Recall from the previous chapter that the reduced convex hull of a set of points $P = \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ is defined as [28, 7]:

$$\mathrm{RCH}(P, \mu) = \left\{ \sum_i^n \alpha_i x_i \;\middle|\; \sum_i^n \alpha_i = 1, \quad 0 \leq \alpha_i \leq \mu \right\}, \tag{3.2}$$

where $0 \leq \mu \leq 1$. Notice that the difference between a reduced and a standard convex hull is the introduction of the constant μ, which limits the maximum influence any individual point can exert.

Theorem 2. *The μ-reduced convex hull of P is a convex set.*

Proof. A set is convex if a line joining any two points from the set is also enclosed in the set. Take two points $\mathbf{a}, \mathbf{b} \in \mathrm{RCH}(P, \mu)$ from the reduced convex hull of P. These points

can be expressed as

$$\mathbf{a} = \sum_i \alpha_i \mathbf{x}_i \qquad 0 \leq \alpha_i \leq \mu \qquad \sum_i \alpha_i = 1$$
$$\mathbf{b} = \sum_i \beta_i \mathbf{x}_i \qquad 0 \leq \beta_i \leq \mu \qquad \sum_i \beta_i = 1.$$

The line segment joining $\mathbf{a}$ and $\mathbf{b}$ is given by

$$\begin{aligned}\lambda\mathbf{a} + (1-\lambda)\mathbf{b} &= \lambda \sum_i \alpha_i \mathbf{x}_i + (1-\lambda) \sum_i \beta_i \mathbf{x}_i \\ &= \sum_i (\lambda\alpha_i + (1-\lambda)\beta_i)\mathbf{x}_i\end{aligned}$$

where $\lambda \in [0,1]$. Note that $\sum_i (\lambda\alpha_i + (1-\lambda)\beta_i) = 1$, $0 \leq \lambda\alpha_i \leq \lambda\mu$ and $0 \leq (1-\lambda)\beta_i \leq (1-\lambda)\mu$. Using this, and letting $\gamma_i = (\lambda\alpha_i + (1-\lambda)\beta_i)$, we have

$$\lambda\mathbf{a} + (1-\lambda)\mathbf{b} = \sum_i \gamma_i \mathbf{x}_i \qquad 0 \leq \gamma_i \leq \mu \qquad \sum_i \gamma_i = 1,$$

This is also a point in $\mathrm{RCH}(P, \mu)$, hence the reduced convex hull is a convex set. □

3.4 Finding Vertices and Support Points

Because reduced convex hulls are convex sets (Theorem 2), their border forms a convex polytope. However, unlike convex hulls, the vertices of $\mathrm{RCH}(P, \mu)$ are no longer guaranteed to be points from P but can instead be convex combinations of points from P. When $\mu \leq 1/k$, vertices *must* be formed as a convex combination of at least k points, since any fewer points would fail to satisfy the constraint that the sum of all α_i values must equal one.

Being able to compute reduced convex hulls depends critically on being able to find their vertices. The task of finding vertices is achieved by noting that, owing to the fact that reduced convex hulls are convex sets, their vertices must still be extreme points. This means that, for some direction $\hat{\mathbf{n}}$, a vertex $\mathbf{v} \in \mathrm{RCH}(P, \mu)$ must satisfy:

$$\mathbf{v} \cdot \hat{\mathbf{n}} > \mathbf{r} \cdot \hat{\mathbf{n}} \qquad \forall \mathbf{r} \in \mathrm{RCH}(P, \mu), \quad \mathbf{r} \neq \mathbf{v}$$

At a glance finding vertices satisfying this equation seems difficult, given that there is an infinite number of possible convex combinations for points in P. However, the process is simplified by noting that any vertex can be written:

$$\mathbf{v} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i, \qquad 0 \leq \alpha_i \leq \mu, \quad \sum_{i=1}^{n} \alpha_i = 1.$$

Since the scalar projection of $\mathbf{v}$ onto direction $\hat{\mathbf{n}}$ is given by:

$$\mathbf{v} \cdot \hat{\mathbf{n}} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i \cdot \hat{\mathbf{n}},$$

it follows that a vertex which is extreme in direction $\hat{\mathbf{n}}$ must be formed as a convex combination of the points from P which have the largest possible scalar projection onto $\hat{\mathbf{n}}$. The process of finding a vertex in a given direction is described in Theorem 3.

Theorem 3. *(Bern and Eppstein [9], Mavroforakis and Theodoridis [82]). For direction* $\hat{\mathbf{n}}$*, a vertex* $\mathbf{v} \in \mathrm{RCH}(P, \mu)$ *which maximizes* $\mathbf{v} \cdot \hat{\mathbf{n}}$ *satisfies*

$$\mathbf{v} = \sum_{i=1}^{m-1} \mu \mathbf{z}_i + (1 - (m-1)\mu)\mathbf{z}_m, \quad m = \lceil 1/\mu \rceil, \tag{3.3}$$

where $P = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$ *and points are ordered in terms of scalar projection onto* $\hat{\mathbf{n}}$ *such that* $\mathbf{z}_i \cdot \hat{\mathbf{n}} > \mathbf{z}_j \cdot \hat{\mathbf{n}} \Rightarrow i < j$*.*

We refer to the points which appear with a non-zero α_i in Equation (3.3) as the *support points* of a vertex.

The process of finding a vertex of a reduced convex hull can be visualized in terms of a plane being 'pushed' into a set of points (Figure 3.5). If the plane can separate $\lceil 1/\mu \rceil$ points from the rest of the set, there is a convex combination of these points (given by Theorem 3) which forms a vertex of the μ-reduced convex hull. For the case of $\mu = 1/k$ (where k is an integer), all k points will have $\alpha_i = \mu$.

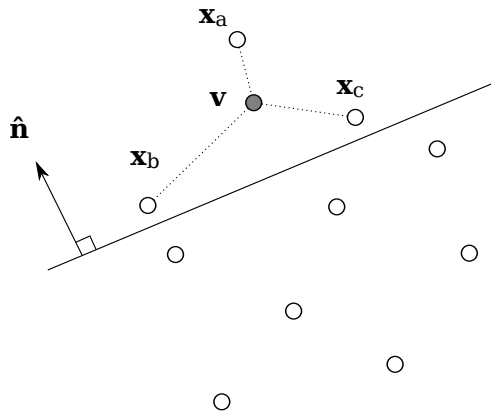


Figure 3.5: Finding support points by 'pushing' a plane into a set of points. For $\mathrm{RCH}(P, 1/3)$, the plane with normal $\hat{\mathbf{n}}$ separates points $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c \in P$ from the rest of P. Accordingly, $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c$ become the support points for the vertex $\mathbf{v} \in \mathrm{RCH}(P, 1/3)$.

Theorem 3 is simplest to apply when no two points have an equal scalar projection onto $\hat{\mathbf{n}}$. However it is not always guaranteed that this will be the case. When points are in general position, ties like this may be broken arbitrarily and a vertex will still be obtained. Arbitrary tie breaking is possible because, for d-dimensional points in general position, at most d points will lie on the separating hyperplane.

Finding vertices becomes more complex when points are not in general position, since there can exist cases where breaking a tie arbitrarily can result in a non-vertex being found. For example, Figure 3.6 shows a set of points not in general position. For this case, the three possible ways to break the tie can result in two valid vertices and one non-vertex. Because all possible results maximize the scalar projection onto $\hat{\mathbf{n}}$, $\mathbf{v}$ will always lie on the outside of the hull. However, one possible value for $\mathbf{v}$ (shown as $\mathbf{v}'$ in Figure 3.6) is not a vertex since it falls on a line joining the two vertices. If used in the construction of an RCH, this non-vertex will not result in an incorrect RCH being formed, since it does lie on the outside of the RCH. However, a larger number of facets than is strictly necessary will be constructed.

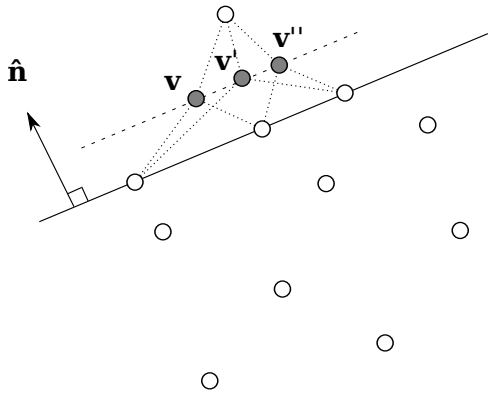


Figure 3.6: The hyperplane can not clearly separate three points in this case. There are three possible values for $\mathbf{v} \in \text{RCH}(P, 1/3)$ given by Theorem 3. One of these ($\mathbf{v}'$) is not a true vertex.

3.5 Algorithms for Computing RCHs

Given that reduced convex hulls are convex polytopes, they can be constructed in a similar manner to convex hulls. In this section we generalize the Quickhull and Beneath-Beyond algorithms to create two RCH algorithms: an algorithm for points in arbitrary dimensional space, and a faster algorithm for the special case of points in the plane. The main feature of our new algorithms is that they allow the computation of reduced convex hulls with any value of μ, whereas the original algorithms were limited to convex hulls only.

3.5.1 Representing an RCH

In Section 3.2.2 we described a commonly used representation for convex hulls in $\mathbb{R}^d$. However, it is not feasible to use this representation when vertices are no longer individual points, but convex combinations of points, as is the case in RCHs. This means that an alternative representation is required. Although the algorithms we describe in this section are general in that they can be used in conjunction with any RCH representation, we suggest the following one because it is efficient to work with and stores the complete set of information that defines an RCH. That is, it stores the facets, the vertices making up

those facets, the support points making up those vertices and, optionally, an adjacency matrix which specifies the facets which are adjacent.

The most basic component of an RCH is its vertices. Unlike a convex hull, these are no longer points from the original set, but are convex combinations of points from that set. We represent the vertices of an RCH using an $n_v \times m$ matrix, where $m = \lceil 1/\mu \rceil$ and n_v is the number of vertices in the RCH. For example, if $\mu = 1/2$, the vertices of an RCH can be represented as:

$$V = \begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 2 & 3 \\ \vdots & \vdots \end{bmatrix}$$

This matrix implies that $\mathbf{x}_1, \mathbf{x}_2 \in P$ are support points for one vertex, and $\mathbf{x}_5, \mathbf{x}_6$ are support points for another. If $1/\mu$ is not an integer, then the final column specifies the support point which is given a lesser weighting than other points. This means that V is capable of representing all of the vertices of an RCH.

The facets of an RCH can be represented as an $n_f \times d$ matrix, where n_f is the number of facets, and d is the dimensionality of the input space. This matrix references rows from the vertex matrix V, described above. For example, if one facet is made up of the vertices from the first three rows of V, the facet matrix would appear as:

$$F = \begin{bmatrix} 1 & 2 & 3 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Although the dimensions of this matrix are identical to the one used to represent the facets of a convex hull, the two matrices store different information. This is because the facets of a convex hull are made up of vertices from the original input set P, whereas the facets of an RCH are made up of vertices from V.

3.5.2 Hyperplanes Passing Through d Points in $\mathbb{R}^d$

An operation which is required in both RCH algorithms is to find the normal of a plane passing through d points in $\mathbb{R}^d$. This operation is simplest in two dimensions, in which the normal $\hat{\mathbf{n}} = (n_1, n_2)$ of the line passing through points $\mathbf{x}_1 = (x_{11}, x_{12})$ and $\mathbf{x}_2 = (x_{21}, x_{22})$ can be given by:

$$\begin{aligned} \hat{\mathbf{n}} \cdot (\mathbf{x}_2 - \mathbf{x}_1) &= 0, \\ n_1(x_{21} - x_{11}) + n_2(x_{22} - x_{12}) &= 0. \end{aligned}$$

A solution for this is:

$$\mathbf{n} = (x_{12} - x_{22}, x_{21} - x_{11}),$$

which can then be normalized if necessary.

In higher dimensions, this operation is less trivial. For the general case of d points in $\mathbb{R}^d$, we want to find the normal $\hat{\mathbf{n}}$ satisfying:

$$\mathbf{x}_1 \cdot \hat{\mathbf{n}} = \mathbf{x}_2 \cdot \hat{\mathbf{n}} = \cdots = \mathbf{x}_d \cdot \hat{\mathbf{n}} = c,$$

where c is any positive constant. This problem can be solved as a system of linear equations:

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & \cdots & x_{dd} \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_d \end{bmatrix} = \begin{bmatrix} c \\ c \\ \vdots \\ c \end{bmatrix}.$$

3.5.3 Points in the Plane

Reduced convex hulls are simplest to construct for two-dimensional point sets. For this case our algorithm takes as input a set of points P and a reduction coefficient μ. The output is an ordered list of vertices of the RCH, V, which serves as both a facet list and an adjacency list. Any two consecutive vertices in V form a facet. This means that the algorithm only needs to output V, and not the separate facet list F which we described previously.

The first step taken by the algorithm is to find two initial vertices of the reduced convex hull $\mathbf{l}$ and $\mathbf{r}$. This is achieved by finding two vertices which are extreme along the x-axis (one in the positive direction, one in the negative direction). More explicitly, we let $\hat{\mathbf{n}} = (1,\ 0)$ and then compute:

$$\mathbf{l} = \underset{\mathbf{l} \in \mathrm{RCH}(P,\mu)}{\arg\max} \{-\hat{\mathbf{n}} \cdot \mathbf{l}\} \qquad \mathbf{r} = \underset{\mathbf{r} \in \mathrm{RCH}(P,\mu)}{\arg\max} \{\hat{\mathbf{n}} \cdot \mathbf{r}\}$$

Recall that solving for $\mathbf{l}$ and $\mathbf{r}$, i.e. finding vertices which are extreme in a given direction, is a straightforward process that we described in Theorem 3.

The two initial vertices form a line segment which is treated as two initial facets with opposing normals. Facets are then recursively split into two by adding new vertices, found using the normal vector of a facet in conjunction with Theorem 3. This process is described in detail in Algorithm 5. Notice that the recursive update step taken by the RCH algorithm is a generalization of the recursive step taken by the Quickhull algorithm for standard convex hulls.

The subsets L and R are formed in order to facilitate the removal of points which can no longer contribute to future iterations of the algorithm (since they are not 'extreme' enough with respect to the current facet being considered). Because the vertices of a reduced convex hull can share multiple support points, L and R are generally not disjoint partitions as they are in the original Quickhull algorithm.

When $\mu = 1$, the algorithm becomes a standard convex hull algorithm. This means that S (the set of support points for a new vertex $\mathbf{h}$) will contain a single point only, which

Algorithm 5 Quickhull algorithm for reduced convex hulls

function QRH(P, μ)
  $\mathbf{n} \leftarrow (1, 0)$
  $\mathbf{l} \leftarrow \underset{\mathbf{x} \in \mathrm{RCH}(P,\mu)}{\arg\max} -\mathbf{n} \cdot \mathbf{x}$ ▷ these two steps find the initial vertices in the RCH,
  $\mathbf{r} \leftarrow \underset{\mathbf{x} \in \mathrm{RCH}(P,\mu)}{\arg\max} \mathbf{n} \cdot \mathbf{x}$ ▷ and should be performed using Theorem 3
  return qrh_helper($P, \mathbf{l}, \mathbf{r}$) ∪ qrh_helper($P, \mathbf{r}, \mathbf{l}$)
end function

function QRH_HELPER($P, \mathbf{l}, \mathbf{r}$)
  $\mathbf{n} \leftarrow \mathrm{normal}(\mathbf{r} - \mathbf{l})$ ▷ normal of old facet
  $\mathbf{h} \leftarrow \underset{\mathbf{x} \in \mathrm{RCH}(P,\mu)}{\arg\max} \mathbf{n} \cdot \mathbf{x}$ ▷ new vertex, use Theorem 3 to compute
  if $\mathbf{h} = \mathbf{l}$ or $\mathbf{h} = \mathbf{r}$ **then**
    return $\{\mathbf{l}, \mathbf{r}\}$ ▷ return final facet, since it can not be split
  end if
  $\mathbf{n}_l \leftarrow \mathrm{normal}(\mathbf{h} - \mathbf{l})$ ▷ normal of first new facet
  $\mathbf{n}_r \leftarrow \mathrm{normal}(\mathbf{r} - \mathbf{h})$ ▷ normal of second new facet
  $S \leftarrow$ the support points of $\mathbf{h}$
  $L \leftarrow$ the set of all points $\mathbf{x} \in P$ satisfying $\mathbf{n}_l \cdot \mathbf{x} \geq \mathbf{n}_l \cdot \mathbf{s}$, for any $\mathbf{s} \in S$
  $R \leftarrow$ the set of all points $\mathbf{x} \in P$ satisfying $\mathbf{n}_r \cdot \mathbf{x} \geq \mathbf{n}_r \cdot \mathbf{s}$, for any $\mathbf{s} \in S$
  return qrh_helper($L, \mathbf{l}, \mathbf{h}$) ∪ qrh_helper($R, \mathbf{h}, \mathbf{r}$)
end function

must be a vertex of the convex hull. For this case, the splits performed by the algorithm will be identical to those performed by the standard Quickhull algorithm. This means that, for the special case of $\mu = 1$, the two-dimensional RCH algorithm becomes identical to the Quickhull algorithm.

3.5.4 Points in Arbitrary Dimensional Space

When working with input points in three or more dimensions, a more complex approach (compared to that of the Quickhull algorithm) is required in order to deal with the property that multiple facets may be visible to each successive vertex which is added to the hull. Our approach is derived from the Beneath-Beyond algorithm [61, 94], and shares similarities with Barber et al.'s algorithm [4] for the case of $\mu = 1$.

We avoid the randomized incremental approach used in several Beneath-Beyond implementations in order to avoid intermediate vertices in the reduced convex hull. We define intermediate vertices as vertices which are replaced in successive iterations of the algorithm. Although such operations do not pose a problem in the computation of convex hulls, they become difficult in reduced convex hulls. This is because the introduction of a new support point could require not only the addition of and removal of several facets, but the recalculation of a large number of vertices as the new support point is added and any old support points are removed. For this reason we want to ensure that any time a vertex is introduced to the reduced convex hull it is guaranteed to be a final and permanent vertex.

Input to our reduced convex hull algorithm (Algorithm 6) is a set of points P and a reduction coefficient μ. Output consists of:

- A list of vertices V. For d-dimensional input, each vertex is a d-dimensional point.
- A list of facets F. For d-dimensional input, each facet has d vertices. Each vertex is represented as an index of a vertex in V.
- An adjacency list A. Each entry in F has a corresponding entry in A which specifies the indices of neighboring facets.

These outputs could be represented in a number of ways, and the algorithm itself does not specify explicitly which representation to use. However, we recommend matrix representations which we previously described in Section 3.5.1, for their simplicity and completeness.

The starting point for the algorithm (for a d-dimensional point set) is to find any d unique vertices of the reduced convex hull (using Theorem 3 in conjunction with d arbitrary directions). These vertices are used to form two initial facets with opposing normals. For each facet, the algorithm finds a new vertex with the largest possible scalar projection onto the facet's normal. The new vertex is added to the hull by 'paving over' any visible facets and removing them as described in the Beneath-Beyond theorem. This process is repeated over all facets until no new vertices can be found.

Algorithm 6 General dimension reduced convex hull algorithm

```
function BENEATH_RCH(P, μ)
    V ← any d unique vertices from RCH(P, μ)
    F ← two facets with opposing normals passing through the points in V
    while non-final facets exist in F do
        choose a non-final facet f from F
        n̂ ← normal vector of f
        h ← arg max_{x∈RCH(P,μ)} n̂ · x        ▷ new RCH vertex, found using Theorem 3
        if h is not in V then                  ▷ if vertex is new, add it to the hull
            find all facets X which are visible from h
            for all boundary edges E in X do
                add a (non-final) facet to F which joins h with the vertices in E.
            end for
            remove the facets in X from F
        else                                   ▷ otherwise, facet is final
            mark f as a final facet and do not revisit
        end if
    end while
    return (V, F)                              ▷ return vertices and facets
end function
```

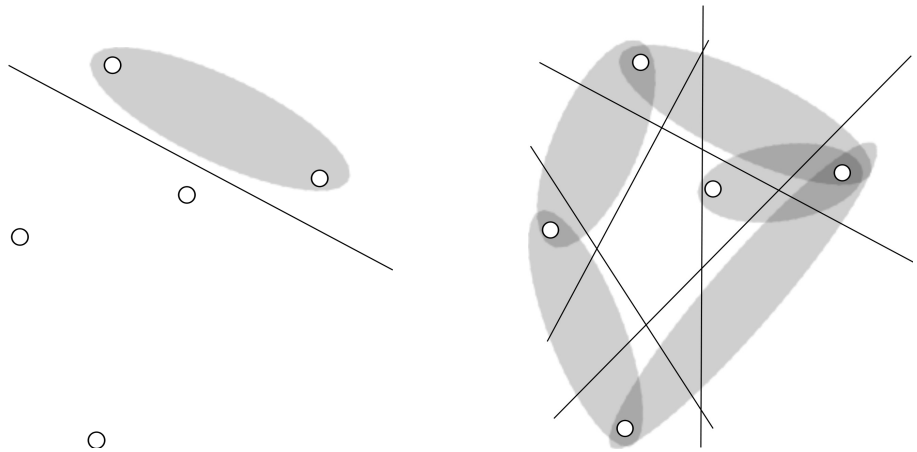
The connection to Barber et al.'s algorithm is further elucidated when the case of $\mu = 1$ is considered. For this case the two algorithms become equivalent, with the exception of the partitioned outside sets which are omitted in our algorithm. Such outside sets are not possible to maintain in the same manner as Barber et al.'s implementation since distinct vertices in a reduced convex hull can share support points.

3.6 Related Geometric Concepts

There are several existing concepts in computational geometry which are closely related to, and in some cases equivalent to, RCHs. For example, Bern and Eppstein [9] note that RCHs are a subset of centroid polytopes [10]. There is also an existing geometric concept of a k-set which is related to RCHs in a way that we believe has not yet been pointed out. We review the definition of centroid polytopes and k-sets below, and explore their relationship to RCHs.

3.6.1 k-sets

A k-set of P is a subset of k elements from P which can be separated from the remaining points using a hyperplane [25, 121, 1]. An example of a 2-set is shown in Figure 3.7a. Here a plane can clearly separate two points from the remaining points in P, creating a 2-set. Figure 3.7b shows that there are a total of five 2-sets which can be formed from this particular set of points.



(a) A plane separates a 2-set of P

(b) P has a total of five 2-sets (shown circled)

Figure 3.7: 2-sets of a set of points in $\mathbb{R}^2$

Recall from Section 3.4 that the procedure for finding a vertex in an RCH can be visualized as a plane being 'pushed' into a set of points until it separates $\lceil 1/\mu \rceil$ points. This means that each vertex of an RCH is formed from a k-set of P, where $k = \lceil 1/\mu \rceil$. It follows that the number of k-sets in P is equal to the number of vertices in $\text{RCH}(P, 1/k)$.

There are two common problems associated with k-sets: computing the number of k-sets in a set of points, and bounding the maximum number of k-sets that can exist for any set of size n. It is interesting to note that Algorithms 5 and 6 can be used to compute the number of k-sets in a set of points by simply totalling the number of vertices in a RCH with $\mu = 1/k$.

3.6.2 k-set Polytopes

There are two equivalent definitions of a k-set polytope. Andrzejak and Fukuda [2] define the k-set polytope of P as the convex hull of the set:

$$X(P;k) = \left\{ \sum_{\mathbf{t}\in T} \mathbf{t} \;\middle|\; T \in P,\ |T| = k \right\}.$$

Notice that $X(P;k)$ is the set of all points which are equal to the sum of any k points from P. An equivalent (scaled) definition is provided by Oraiby and Schmitt [88], who defines a k-set polytope as the convex hull of $(1/k)X(P;k)$.

The relationship between RCHs and k-set polytopes is best understood using the definition given by Oraiby and Schmitt [88], in which case the k-set polytope of P is the convex hull of all centroids of the k-sets of P, making it equivalent to an RCH with $\mu = 1/k$. The only difference between an RCH and a k-set polytope, is that the RCH parameter μ can take any of a continuous range of values, whereas a k-set polytope is limited to integer values of k.

3.6.3 Centroid Polytopes

Bern et al. [10] define a centroid polytope $C(P;\mathbf{l},\mathbf{h})$ as:

$$C(P;\mathbf{l},\mathbf{h}) = \left\{ \sum_{i=1}^{n} \alpha_i \mathbf{x}_i \;\middle|\; \sum_{i=1}^{n} \alpha_i = 1, \quad l_i \le \alpha_i \le h_i, \quad \sum_{i=1}^{n} l_i \le 1 \le \sum_{i=1}^{n} h_i \right\}.$$

A centroid polytope is the set of all weighted centroids of P, where all points have a unique upper and lower bound determining their minimum and maximum influence on the polytope.

The centroid polytope is an even further generalized form of reduced convex hull. When $l_i = 0$ and $h_i = \mu$ for all i, the centroid polytope becomes equivalent to the μ-reduced convex hull. Rather than having a shared upper bound on all α_i values, each point in a centroid polytope has its own unique upper bound. In addition, each point also has an associated lower bound. This means that while it is possible to limit the influence certain points can have on the polytope, other points can be forced to exert a greater influence.

3.7 Properties of RCHs

With the aid of the algorithms introduced in the previous section we are now able to explore some of the more interesting properties of reduced convex hulls, such as how many vertices and facets are present and how this number will change as the parameter μ changes. There is also an isomorphism between an RCH with $\mu = 1/k$ and an RCH with $\mu = 1/(n-k)$, which we will describe in more detail in this section.

3.7.1 How the Reduction Works

Reduced convex hulls provide a desirable method of reducing a convex hull because they take into account the density of points. In regions where there are a lot of points (high density regions), the border of the RCH will contract more strongly towards the centroid than in regions where there are relatively few points (low density regions). This means that, in the context of SVMs, reduced convex hulls are used to lessen the impact of outlying points and increase the margin between two classes. Figure 3.8 shows an RCH with varying μ reducing over both low and high density regions.

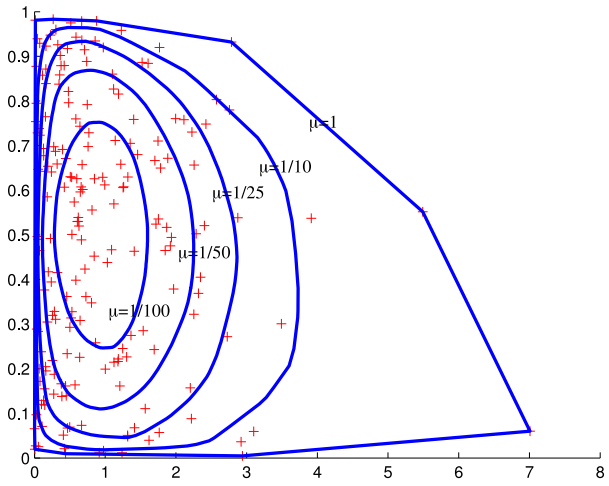


Figure 3.8: Reduced convex hulls for $\mu = 1, \frac{1}{10}, \frac{1}{25}, \frac{1}{50}, \frac{1}{100}$.

3.7.2 Number of Vertices

The number of vertices in a reduced convex hull depends on the number of input points and their dimensionality, as well as the parameter value μ. Letting $m = \lceil 1/\mu \rceil$, the simplest (and most trivial) bound on the number of vertices in an RCH of n points is given by:

$$n_v(n, d, m) \leq \begin{cases} \binom{n}{m} & \text{if } 1/\mu \text{ is an integer,} \\ \binom{n}{m} m & \text{otherwise.} \end{cases} \tag{3.4}$$

The additional vertices that can be present when $1/\mu$ is not an integer occur because, in addition to there being $\lceil 1/\mu \rceil$ support points per vertex, one of those support points must be given a smaller α_i than the others.

Although Equation (3.4) is a simple and valid bound, it tends to be insufficiently loose, since the only time a hyperplane can separate *all* combinations of k points is when $n = d + 1$. Tighter bounds on the number of vertices in an RCH can be achieved using results from k-set theory. For example, Clarkson [24] notes that an upper bound on the number of $\leq k$-sets that can be formed from a set of n points in d dimensions can be given by $O(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil})$. Note that the number of $\leq k$-sets is the sum of the number of all i-sets, where $i \leq k$.

Clarkson's bound can be turned into a bound on the number of vertices in an RCH:

$$n_v(n,d,m) \leq \begin{cases} O(n^{\lfloor d/2 \rfloor} m^{\lceil d/2 \rceil}) & \text{if } 1/\mu \text{ is an integer,} \\ O(n^{\lfloor d/2 \rfloor} m^{\lceil d/2 \rceil + 1}) & \text{otherwise.} \end{cases} \tag{3.5}$$

Here $m = \min(\lceil 1/\mu \rceil, n - \lceil 1/\mu \rceil)$, meaning similar to Equation (3.4), the bound only increases until $m = n/2$. The bound is equal for the cases of $\mu = 1/k$ and $\mu = 1/(n-k)$. This suggests an interesting isomorphism between RCHs with these values of μ, which we will explore in the following sections.

Note that although Clarkson's bound is for $\leq k$-sets, it is currently also the tightest known upper bound on the number of k-sets for points in an arbitrary dimensional space. Tighter bounds apply only to specific cases of small d. For example Dey [31] provides an upper bound of $O(nk^{1/3})$ on the number of k-sets for points in the plane. This bound can be used to construct a slightly tighter bound on the number of vertices of an RCH in the plane:

$$n_v(n,m) \leq \begin{cases} O(nm^{1/3}) & \text{if } 1/\mu \text{ is an integer,} \\ O(nm^{4/3}) & \text{otherwise.} \end{cases} \tag{3.6}$$

Here we again use $m = \min(\lceil 1/\mu \rceil, n - \lceil 1/\mu \rceil)$.

3.7.3 Number of Facets

The maximum number of facets in an RCH is closely related to the maximum number of facets in a standard convex hull. Preparata and Shamos [94] provide an upper bound (attributed to Klee [70]) on the number of facets in the convex hull of n points in d dimensions as:

$$n_f(n,d) \leq \begin{cases} \frac{2n}{d}\binom{n-\frac{d}{2}-1}{\frac{d}{2}-1}, & \text{for } d \text{ even} \\ 2\binom{n-\lfloor \frac{d}{2} \rfloor - 1}{\lfloor \frac{d}{2} \rfloor}, & \text{for } d \text{ odd} \end{cases}$$

This bound corresponds to the case where all of the n points form vertices in the convex hull.

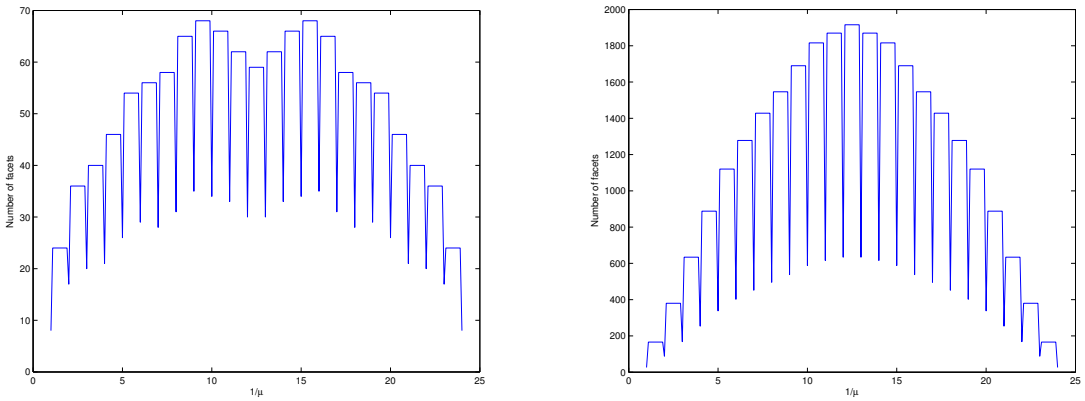
In the case of a reduced convex hull, the maximum number of facets is greater, and depends on the value of μ. Combining Klee's upper bound with either Equation (3.4) or (3.5) provides an upper bound on the number of facets in the RCH of n points in d dimensions with $m = \lceil 1/\mu \rceil$:

$$n_r(n,d,m) \leq n_f(n_v(n,d,m),d) \tag{3.7}$$

The upper bound on the number of vertices in an RCH in the plane (given in Equation 3.6) may also be used to provide a tighter facet bound specifically for the two-dimensional case.

3.7.4 Symmetry

In all bounds on both the number of vertices and facets in an RCH there is an interesting symmetry which occurs between the cases of $\mu = 1/k$ and $\mu = 1/(n-k)$. These cases have an identical upper bound on the number of vertices in the hull and, consequently, an identical upper bound on the number of facets. Figure 3.9 shows how this symmetry is also reflected in the *actual* number of vertices and facets in an RCH.



(a) Points in $\mathbb{R}^2$ **(b)** Points in $\mathbb{R}^3$

Figure 3.9: Number of facets in the RCH of 25 uniformly distributed random points

The reason for the symmetry in the cases of $\mu = 1/k$ and $\mu = 1/(n-k)$ can be found by considering some the vertices belonging to the RCHs for these μ values, as depicted in Figure 3.10. Notice how any hyperplane that separates k points must also separates $n-k$ points on the opposing side of the plane. From Theorem 3, we know that if a hyperplane can separate k points in a set P from the remaining points in P, these points must form a vertex of RCH$(P, \frac{1}{k})$. It follows that RCH$(P, \frac{1}{k})$ must have the exact same number of vertices as RCH$(P, \frac{1}{n-k})$.

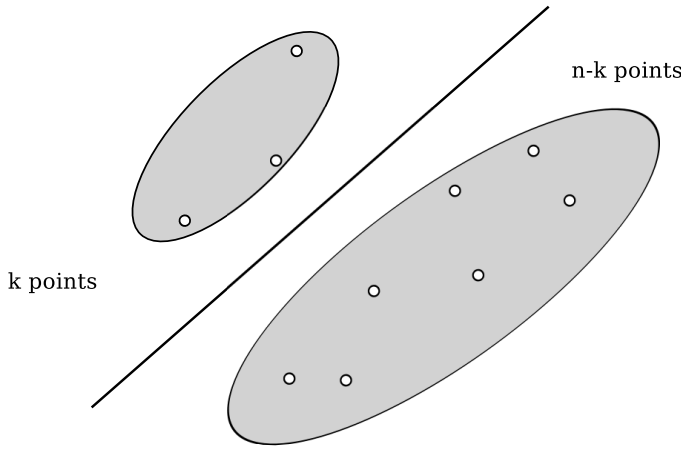


Figure 3.10: A plane separating k points on one side and $n-k$ points on the other side

The property depicted in Figure 3.10 means that any vertex $\mathbf{v}_a \in \text{RCH}(P, \frac{1}{k})$ can be transformed into a vertex $\mathbf{w}_a \in \text{RCH}(P, \frac{1}{n-k})$, which is extreme in the opposite direction to $\mathbf{v}_a$. If vertex $\mathbf{v}_a$ is given by:

$$\mathbf{v}_a = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i,$$

then, using Theorem 3, the transformed vertex $\mathbf{w}_a$ can be given by:

$$\mathbf{w}_a = \frac{k}{n-k} \sum_{i=1}^{n} (\frac{1}{k} - \alpha_i)\mathbf{x}_i.$$

Letting $\mathbf{c} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i$ equal the centroid of P, $\mathbf{w}_a$ can also be written:

$$\begin{aligned} \mathbf{w}_a &= \frac{k}{n-k} \sum_{i=1}^{n} \frac{1}{k}\mathbf{x}_i - \sum_{i=1}^{n} \alpha_i \mathbf{x}_i, \\ &= \frac{n}{n-k}\mathbf{c} - \frac{k}{n-k}\mathbf{v}_a. \end{aligned} \tag{3.8}$$

Equation 3.8 implies that the entire $\text{RCH}(P, \frac{1}{k})$ is a reflected and scaled $\text{RCH}(P, \frac{1}{n-k})$. Figure 3.11 shows an example of this transformation for six points in $\mathbb{R}^2$, with $k = 2$. The larger hull corresponds to $\mu = 1/k = 1/2$, while the smaller hull corresponds to $\mu = 1/(n-k) = 1/4$. Notice how the smaller hull is simply a reflected and scaled copy of the larger hull. Just as $\mathbf{v}_b$ connects to $\mathbf{v}_a$ and $\mathbf{v}_c$ via an edge, so too does $\mathbf{w}_b$ connect to $\mathbf{w}_a$ and $\mathbf{w}_c$ via an edge. Each vertex and facet in the larger hull can be paired with its transformed vertex or facet in the smaller hull, and vice versa.

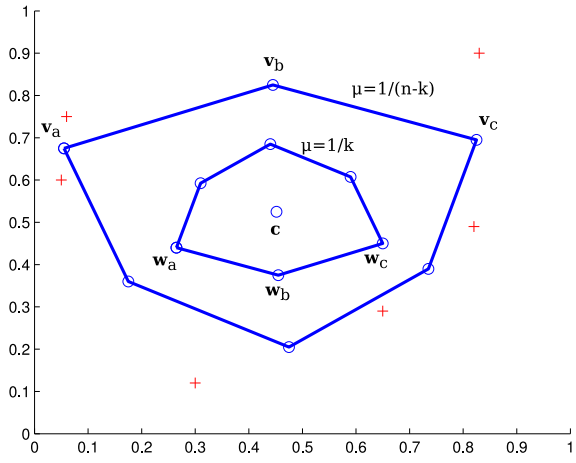


Figure 3.11: Transforming an RCH with $\mu = 1/k$ into an RCH with $\mu = 1/(n-k)$. Here $k = 2$ is used.

It is also informative to observe how the number of facets for an RCH with μ in the range $(\lfloor 1/k \rfloor, \lceil 1/k \rceil)$ remains constant (Figure 3.12). For RCHs with μ values falling in this range, Theorem 3 suggests that there must be an identical number of vertices, each

with identical support points. This is due to the fact that, for μ values in this range, there must be one support point with a smaller α_i than all other support points. This means that, as μ is increased in this range, the weight assigned to the support point with a smaller α_i value will be decreased, while the α_i's of other support points will be increased. The effect of this shift in α_i's is shown in Figure 3.12.

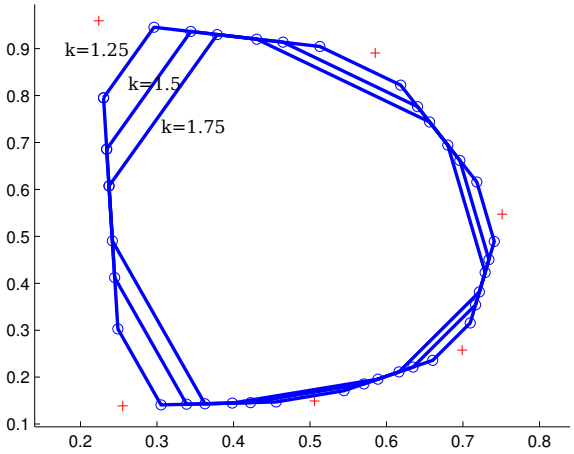


Figure 3.12: RCHs with $\mu \in (\lfloor 1/k \rfloor, \lceil 1/k \rceil)$. Notice how the number of facets and vertices is equal.

This relationship between RCHs with $\mu \in (\lfloor 1/k \rfloor, \lceil 1/k \rceil)$ means that an RCH can have its μ value changed within this range without having to recompute the entire RCH. Instead, vertices can simply be updated, with the connections between vertices (the facets) remaining the same.

3.8 Computational Complexity of RCH Algorithms

Using the properties discussed in the previous section we can now describe the computational complexities of the algorithms introduced in Section 3.2.2.

3.8.1 Points in the Plane

The Quickhull-based algorithm for computing RCHs in the plane is much simpler and more efficient than the algorithm for points in arbitrary dimensional space due to the recursive approach, which is only possible in two-dimensional space.

Each recursive step of the algorithm computes one additional vertex in the RCH. Let there be a maximum of $n_v(n, m)$ vertices in the RCH of n points with $\mu = 1/m$. The dominant cost of each step is to compute the scalar projections required to find the support points for the next vertex. If the subsets formed by the algorithm at each step were disjoint, we could assume the subsets approximately halved with each step to yield a worst case complexity of $O(n_v(n, m) \times \log(n))$. However, since the subsets may overlap significantly, we use a worst case of n scalar projections per step, making the total worst

case complexity of the algorithm $n_v(n,m) \times n$. Taking $n_v(n,m)$ as given by Equation (3.6) results in a worst case complexity of $O(n^2 m^{4/3})$.

3.8.2 Points in Arbitrary Dimensional Space

Constructing RCHs in higher dimensional spaces is difficult, both conceptually and computationally, for a number of reasons. Not only is a recursive approach no longer viable, but the number of vertices and facets increases rapidly with the dimensionality.

The main computational cost in computing RCHs in an arbitrary dimensional space is that of finding the scalar projection of all points onto the normal vectors of all facets, an $O(n_r(n,d,m) \times nd)$ operation. There are additional costs such as: finding $n_v(n,d,m)$ vertices, a cost of $O(n_v(n,d,m) \times nd)$; calculating the normal vectors of new facets each time a vertex is added; and iteratively adding $n_v(n,d,m)$ vertices to the hull. However, all of these costs are eclipsed by the original cost of $O(n_r(n_v(n,d,m),d,m) \times nd)$.

Writing the complexity as $O(n_r(n,d,m) \times nd)$ tends to understate the true difficulty in computing RCHs in high dimensions, so it helps to expand this. Using the simplified $n_f(n,d) = n^{\lfloor d/2 \rfloor}/\lfloor d/2 \rfloor!$ (which Barber et al. [4] attribute to Klee [70]), in conjunction with the relationship between n_f and n_r given in Equation (3.7), the worst case complexity expands to:

$$O\left(\frac{nd\left(n^{\lfloor d/2 \rfloor} m^{\lceil d/2 \rceil + 1}\right)^{\lfloor d/2 \rfloor}}{\lfloor d/2 \rfloor!}\right). \tag{3.9}$$

This scales very poorly with d, and can quickly make the algorithm infeasible as d grows. Even if the cost of computing facets could be reduced to constant time (hence reducing the complexity above by a factor of nd), the sheer number of facets as d grows would still pose a problem.

3.9 Weighted Reduced Convex Hulls

In this section we introduce the concept of a Weighted Reduced Convex Hull (WRCH). A WRCH is an RCH where each individual point is assigned a unique weight specifying its influence on the hull. We define WRCHs such that, in later chapters, we are able to exploit WRCHs in order to provide an intuitive geometric interpretation of Weighted SVMs (WSVMs). As well as allowing us to better understand how and why WSVMs work, this also allows us to construct intuitive geometric algorithms for training WSVMs.

3.9.1 Definition

The WRCH of a set of points is an RCH where, as well as having an overall reduction coefficient μ, each point also has an individual weight s_i specifying its influence on the hull. The WRCH of a set of points P is expressed formally as:

$$\mathrm{WRCH}(P,\mu,\mathbf{s}) = \left\{ \sum_i^n \alpha_i x_i \;\middle|\; \sum_i^n \alpha_i = 1, \quad 0 \le \alpha_i \le s_i \mu \right\}. \tag{3.10}$$

Here the upper bound on the α_i value associated with a point is given by $s_i\mu$, that is, the point's individual weight multiplied by the overall reduction coefficient μ.

3.9.2 Relationship with Point Duplication

Assigning a point a weight of $s_i = k$, where $k \in \mathbb{Z}$, is equivalent to duplicating it so that it exists k times, with each of the k instances given a weight of 1. To verify this equivalence, note that:

$$\left\{ \sum_{i=1}^{k} \alpha_i \mathbf{z} \;\middle|\; 0 \leq \alpha_i \leq \mu \right\} \equiv \left\{ \beta \mathbf{z} \;\middle|\; 0 \leq \beta \leq k\mu \right\}. \tag{3.11}$$

The relationship described by Equation (3.11) means that there are also several other equivalences in WRCHs. Namely, μ is no longer strictly necessary since changing μ by a factor of k is identical to changing all s_i by a factor of k. However we retain μ as a parameter because separating individual point weights and overall hull reduction makes for a more intuitive parameterization. Generalizing the relationship between duplicate points and weights, note also that a single point $\mathbf{x}_i$ with associated weight s_i has the same impact on the hull as multiple duplicates of $\mathbf{x}_i$ with associated weights summing to s_i.

3.9.3 Finding Vertices and Support Points

The process of finding a vertex of a WRCH is not as straightforward as finding a vertex of an RCH. In a WRCH there are no longer a fixed number of support points associated with each vertex. Instead, the number of support points depends on the weight associated with each support point.

Algorithm 7 describes the process for finding a WRCH vertex which is extreme in direction $\hat{\mathbf{n}}$. Notice how this process is similar to the standard RCH vertex finding process, except instead of capping the α_i value of points at μ, they are now capped at $s_i\mu$. Because this means it is not possible to know in advance how many support points are required to ensure $\sum_i \alpha_i = 1$, support points are iteratively added until this constraint is satisfied. The final support point added is the only one which can have $\alpha_i < s_i\mu$.

Algorithm 7 Finding a Weighted RCH vertex

Require: $\hat{\mathbf{n}} \in \mathbb{R}^d$, $P = \mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, $\mathbf{s} \in \mathbb{R}^n$
  function WRCH_VERTEX(P, $\mathbf{s}$, μ, $\hat{\mathbf{n}}$)
    $\alpha \leftarrow \mathbf{0}$
    $s \leftarrow 0$
    repeat
      select i such that $\alpha_i = 0$ and $\hat{\mathbf{n}} \cdot \mathbf{x}_i \geq \hat{\mathbf{n}} \cdot \mathbf{x}_j$, $\forall j \neq i$
      $\alpha_i \leftarrow \min(s_i\mu, 1 - s)$
      $s \leftarrow s + \alpha_i$
    until $s = 1$
    $\mathbf{v} \leftarrow \sum_i \alpha_i \mathbf{x}_i$
    return v
  end function

The point found by the algorithm must be extreme in direction $\hat{\mathbf{n}}$ because the scalar projection $\hat{\mathbf{n}} \cdot \mathbf{v}$ is given by

$$\hat{\mathbf{n}} \cdot \mathbf{v} = \sum_i \alpha_i \hat{\mathbf{n}} \cdot \mathbf{x}_i,$$

which is maximized by assigning the largest possible α_i to the points with the largest scalar projection onto $\hat{\mathbf{n}}$, and zero to all other α_i's, as done by Algorithm 7. As expected, the algorithm becomes equivalent to the RCH vertex finding technique (Theorem 3) when all weights are equal.

Notice in Algorithm 7 that vertices in a WRCH are still formed as a convex combination of points from P. This means that a weight s_i does not in any way scale a point $\mathbf{x}_i$ itself, but rather modifies the *maximum* influence a point can have on the WRCH. If $\mathbf{x}_i$ is not a support point of the WRCH, changing its weight has no impact on the hull.

3.9.4 Adapting RCH Algorithms to Support Weights

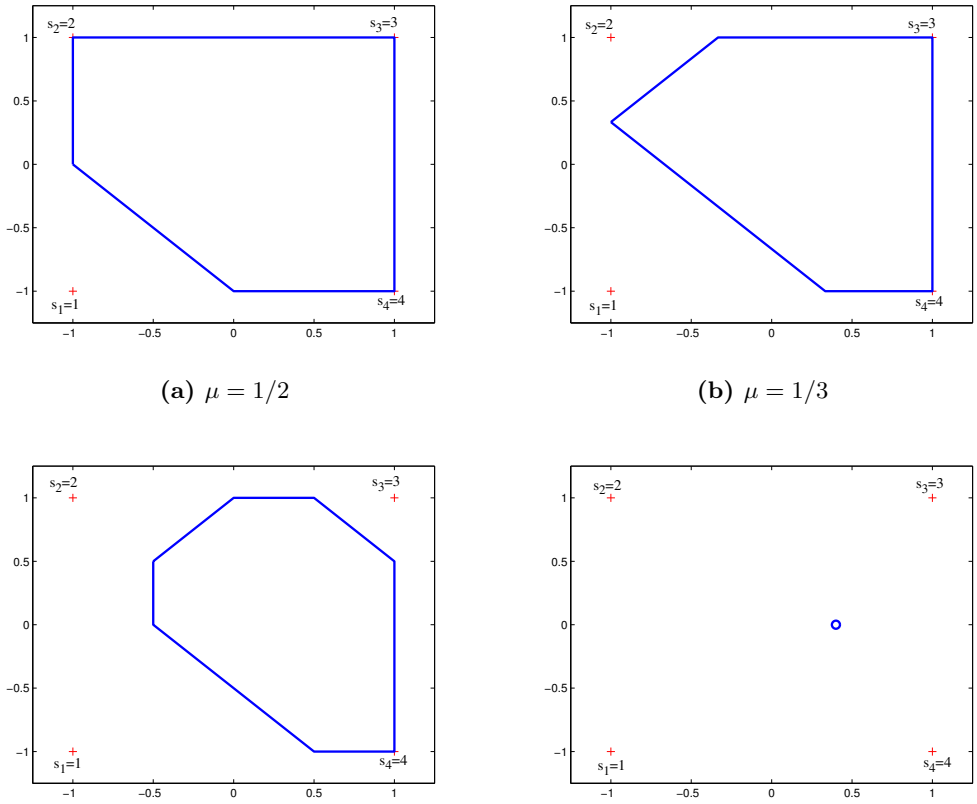
The only change that is required in order to adapt RCH algorithms to find WRCHs is to alter the way in which vertices are found. This means that, instead of finding vertices using Theorem 3, which does not take into account individual point weights, they should be found using Algorithm 7. A vector of individual training weights s_i can then be supplied to the algorithm in order to train WRCHs.

Allowing input points to be assigned an individual weight changes the way the reduction in a WRCH works. An example of the impact of weights is shown in Figure 3.13. In this example, there are four points, with weights given by $s_1 = 1$, $s_2 = 2$, $s_3 = 3$, $s_4 = 4$. Notice how the hull will not recede from a point $\mathbf{x}_i$ until the overall reduction coefficient μ becomes smaller than $1/s_i$. Notice also how, rather than reducing to the centroid, the WRCH reduces to the weighted centroid:

$$\mathbf{c}_{\text{weighted}} = \frac{1}{\sum_i s_i} \sum_i s_i \mathbf{x}_i$$

This reduction to the weighted centroid occurs at $\mu = 1/\sum_i s_i$, which may be greater or less than $1/n$, depending on the weighting scheme that is chosen.

The weighting scheme chosen in Figure 3.13 is equivalent to duplicating points $\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4$ so that they exist 2, 3, and 4 times in the training set, respectively. Notice from this figure how assigning a point a large weight will never result in a scaling of the point itself, nor will it ever cause the RCH to grow past that point. Rather, the weight specifies a threshold at which the hull should be allowed to recede over that point. The larger this threshold is set for a particular point, the smaller the overall reduction parameter must be set in order for the hull to recede over this point.



(a) $\mu = 1/2$ **(b)** $\mu = 1/3$

(c) $\mu = 1/4$ **(d)** $\mu = 1/10$

Figure 3.13: The impact of the parameters on a WRCH

3.10 Conclusions

To summarize, in this chapter we have examined RCHs as a concept in their own right, separate from their origins in SVMs. We have proposed two algorithms for computing RCHs. The first algorithm, a generalization of the Quickhull algorithm, computes RCHs in the plane. The second algorithm computes RCHs in an arbitrary dimensional space.

Using these algorithms, we have been able to explore some of the properties of RCHs. For example, we have described how the number of vertices and facets changes as the reduction parameter μ changes. Generally, the number of vertices and facets tends to reach a maximum when $\mu \approx 2/n$, where n is the number of points.

We have also described an interesting relationship between RCHs with $\mu = 1/k$ and $\mu = 1/(n-k)$, where k is any integer between 1 and n. For these two cases, the number of facets and vertices is identical. Further, the RCH with $\mu = 1/(n-k)$ is a copy of the RCH with $\mu = 1/k$ that has been scaled and rotated by 180 degrees about the centroid. The value assigned to μ can alternate between these two values without having to recompute the entire hull. Instead, the vertices of the original hull can simply be transformed to produce the new hull. There is a similar relationship between RCHs with μ values in the range $(\lfloor 1/k \rfloor, \lceil 1/k \rceil)$, where RCHs can be transformed in this range without having to be completely recomputed.

The algorithm for computing RCHs in the plane has worst case time complexity $O(n^2m^{4/3})$, where n is the number of points, and $m = \min(1/\mu, n - 1/\mu)$. This makes the algorithm less efficient than some of the fastest convex hull algorithms, which have worst case time complexity of $O(n\log(n))$. This difference becomes particularly apparent as μ approaches $2/n$,

We have also proposed an algorithm for computing RCHs in an arbitrary dimensional space. This algorithm has worst case time complexity $O(n^{\lfloor d/2\rfloor+1}m^{\lceil d/2\rceil+1}/\lfloor d/2\rfloor!)$. This means that the algorithm scales extremely poorly as d increases, quickly making the algorithm infeasible. However, this is only slightly worse than most convex hull algorithms, which generally share this property of becoming infeasible as d grows [4].

Another contribution of this chapter is the introduction of WRCHs. WRCHs are RCHs where each individual input point is assigned a unique weight. Assigning a point a weight of two is equivalent to duplicating it. We have described how the vertices of a WRCH may be found, and used this information in order to adapt our RCH algorithms to be able to compute WRCHs. WRCHs allow a greater level of control over the way in which a hull is reduced. For example, certain input points can be assigned a greater or lesser importance, so that the hull recedes from them either slower or faster as μ is decreased.

The concept of WRCHs is further explored in subsequent chapters. For example, in Chapter 4 we describe the relationship between WRCHs and Weighted SVMs (WSVMs). By examining this relationship we are able to better understand how WSVMs work and why they are important. We further extend on this work in Chapter 5 by exploring how WSVMs can be trained using geometrically intuitive algorithms which exploit the concept of WRCHs.

In future work, we suggest that RCHs and WRCHs have the potential to provide an alternative to convex hulls for applications in which convex hulls act poorly due to noise or outlying points. For example, convex hulls have been used in statistics to order multivariate data [5], detect outliers [99, 53], and estimate probability density contours [36]. It would be interesting to explore whether the use of RCHs or WRCHs in these applications could be of any benefit.

There are several ways in which RCHs could be extended in future work. One of the most beneficial improvements that could be made to our RCH algorithms is in reducing the computation complexity as d grows. We suggest that one way in which this might be accomplished is in setting a tolerance parameter ϵ. Rather than computing every single facet, an ϵ-optimal RCH algorithm could stop computing new facets once an update vertex has distance $\leq \epsilon$ to an existing facet.

Another way in which RCHs could be extended is to use alternative measures of centrality in computing vertices. Recall that a vertex in a μ-RCH is formed as the centroid of $1/\mu$ support points (in the simple case where $1/\mu$ is an integer). This measure of centrality is inherently non-robust, and the large number of support points which contribute to the position of each vertex make an RCH inherently less concise to represent than a convex hull. Instead, alternative measures of centrality could be proposed and used to reduce convex hulls in new ways.

Chapter 4

Understanding SVMs from a Geometric Perspective

4.1 Introduction

In this chapter we use the geometric interpretation of SVMs, in conjunction with the properties of Reduced Convex Hulls (RCHs) and Weighted Reduced Convex Hulls (WRCHs) described in the previous chapter, in order to understand better how and why SVMs work. The geometric interpretation makes such analysis possible because it aids in building intuitive mental models of the task being performed by SVMs.

We begin by building a geometric framework which extends the original geometric interpretation of SVMs introduced by Bennett and Bredensteiner [7] and Crisp and Burges [28]. We discuss why convex hulls and RCHs have desirable properties for learning. We also examine the significance of the centroids in SVM classification, and show how SVMs are related to other types of classifiers such as the k-means and k-nearest neighbor classifiers.

Another way in which we build on the geometric framework is by extending it to cover the case of weighted classification. We use the concept of WRCHs, which we introduced in the previous chapter, in order to provide an intuitive geometric interpretation of Weighted SVMs. In later chapters we will exploit this result in order to train WSVMs using geometrically intuitive algorithms.

We build further on the geometric framework by incorporating a geometric interpretation of perceptrons, both in weighted or unweighted forms, with varying loss functions. This allows us to provide a simple geometric definition of a perceptron which encompasses all variants. Comparing this geometric definition of perceptrons with that of SVMs provides a simple way to illustrate the differences between the two machines in geometric terms.

In Section 4.3 we discuss how the threshold of an SVM is computed. The threshold computed by the most commonly used SVM software packages is the one that is given by the KKT conditions (which we described in Section 2.4). However, it has previously been suggested that this is not necessarily the best threshold to use [92]. We show that under certain conditions this threshold can clearly lead to an increase in error rate. We describe some of the alternative thresholds that have been proposed, and provide theoretical and

empirical comparisons of these thresholds. Our main finding from this section is that, in general, there seems to be little reason to prefer the KKT threshold over a geometric or probabilistic threshold.

4.2 The Geometric Framework

The geometric framework for SVMs centers on the concept of RCHs introduced by Bennett and Bredensteiner [7] and Crisp and Burges [28], and elaborated on in Chapter 3. Recall that an SVM is equivalent to the perpendicular bisector of the shortest line between the RCHs of the two classes (Figure 4.1). In this section we build on the geometric framework for SVMs by using the geometric concepts explored in Chapter 3 to understand better how SVMs work, and their relationship to other types of classifiers.

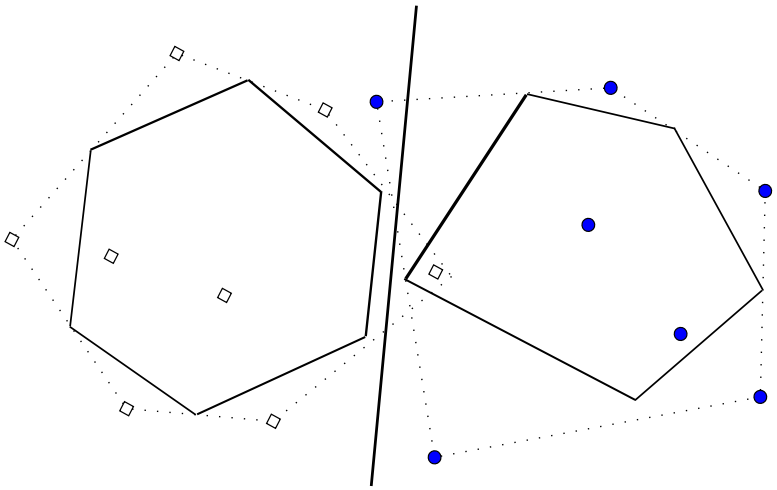


Figure 4.1: The geometric interpretation of an SVM

The threshold is the offset of the decision surface of an SVM from the origin (Figure 4.2). In previous sections we have referred to the offset as b. Changes to b, although they can have a significant impact on classification error, do not alter the orientation (i.e. the normal vector) of the decision surface. It is important to note that there is a discrepancy between the most geometrically intuitive threshold and the threshold given by the KKT conditions of the SVM optimization task [28]. The most geometrically intuitive threshold places the hyperplane half way between the nearest points in the two RCHs, whereas the KKT conditions do not adhere to this placement. We explore this discrepancy in detail in Section 4.3.

4.2.1 Why Use Reduced Convex Hulls for Learning?

For convex hull learning (i.e. RCHs with $\mu = 1$), there is an implicit rule: if points $\mathbf{x}_a, \mathbf{x}_b \in P$ are in the same class, any point on the line segment joining those two points should also be in that class. RCHs generalize this assumption by ensuring that if the centroid of any k points from P and the centroid of any other k points from P are in the same class, then any point on the line segment between the two centroids is also in that class.

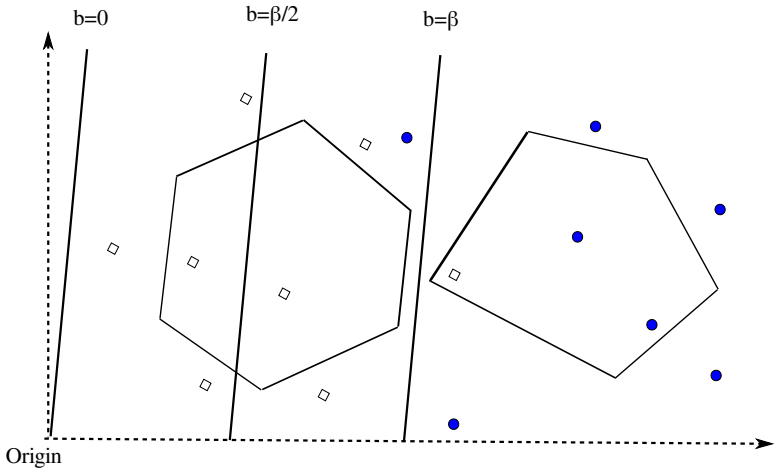


Figure 4.2: The threshold of an SVM. Here $b = \beta$ is the threshold placing the plane halfway between the nearest points of the two classes. $b = \beta/2$ brings the plane closer to the origin, while a hyperplane with $b = 0$ will always intersect the origin.

4.2.2 Significance of the Centroids

The centroids of the classes play an important role in support vector classification. For instance, assuming the hyperplane is placed halfway between the nearest points in the RCH of the two classes (which will always be the case if the geometric threshold is chosen, but may not be the case of other thresholds are chosen, as we will discuss in Section 4.3), the hyperplane must always intersect the line between the centroids of the two classes (Figure 4.3).

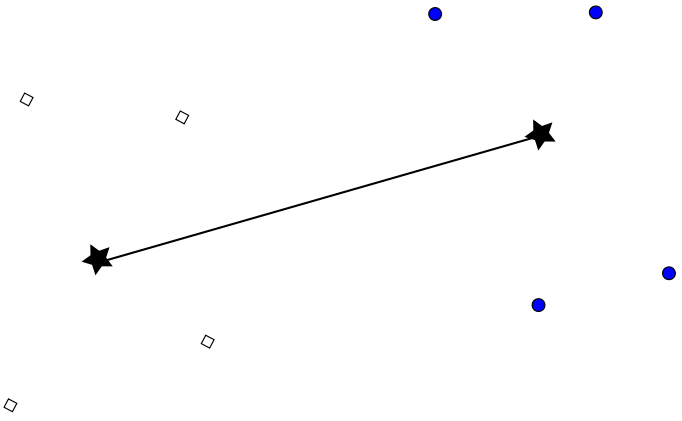


Figure 4.3: Here a solid line joins the centroids of the two classes (represented by stars). Regardless of the way the RCHs of the two classes are reduced, a hyperplane separating them must always intersect this line segment.

The centroids of the classes also play a role in determining whether a dataset can be learned at all by an SVM. For example, suppose the centroid of one class (with k points) lies inside the RCH of the other class, with $\mu = 1/k$ (Figure 4.4). In this case there is no value of μ which can separate the two classes. The result is that, unless different kernel parameters or a weighting scheme is applied, an SVM can not be trained on the data.

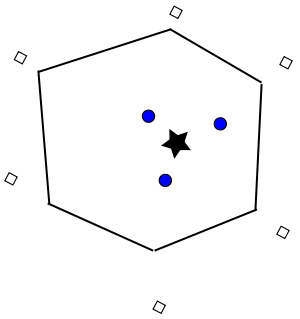


Figure 4.4: When the centroid of one class (represented here by a star) lies inside the RCH of the other class, an SVM can not be trained on the data unless different kernel parameters or weighting schemes are applied

The way in which centroids arise in many aspects of support vector classification reveals an important point regarding their robustness. For any training set, we could add a single point to each class which moves the centroids of the classes to any desired position. Because an SVM is constrained to intersect the line between the means of the two training classes, moving the centroids in such a manner can place the separating hyperplane in almost any desired position. Biggio et al. [11] refer to this as a 'poisoning attack', and describe in more detail how this type of 'malicious' data can be introduced.

4.2.3 Impact of the Kernel

We previously introduced several of the most commonly used kernels and their parameters in Chapter 2. The choice of kernel and kernel parameters has the ability to alter completely the type of decision surface formed by an SVM. In this section we explore further the impact of some of these parameters.

Gaussian Radial Basis Function (RBF) Kernels

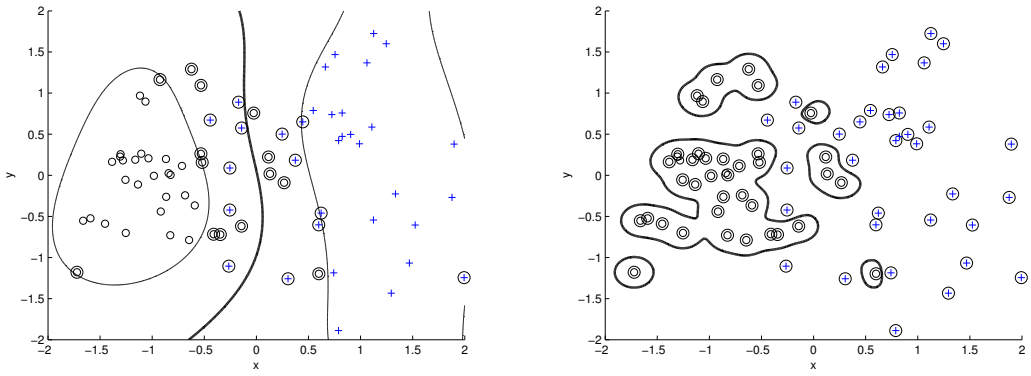
The Gaussian RBF kernel with width γ is written:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2\right). \tag{4.1}$$

Equation (4.1) results in an SVM decision surface of the form:

$$f(\mathbf{x}) = \sum_i \alpha_i y_i \exp\left(-\gamma||\mathbf{x}_i - \mathbf{x}||^2\right) - b.$$

This decision surface consists of a combination of basis functions formed around the support vectors of the machine. The width of these basis functions is specified by the parameter γ, so that smaller values of γ result in basis functions with a larger width. It may seem counter-intuitive that larger values of γ correspond to smaller width basis functions, however we use this convention in order to ensure consistency with common SVM software such as `SVMlight` [58] and `LIBSVM` [20]. Other authors will occasionally use the parameter $\sigma^2 = 1/2\gamma$ to be more consistent with the Gaussian probability density function [64].



(a) $\gamma = 0.5$ **(b)** $\gamma = 50$

Figure 4.5: Gaussian C-SVMs with $C = 1$ while γ is varied

For very large width basis functions, the decision surface of a Gaussian SVM will appear almost linear in attribute space (Figure 4.5a). As the width of the basis functions shrink, the capacity of the machine to fit the data will increase. Consequently, smaller width basis functions are more likely to result in a decision surface which, due to its highly nonlinear nature, is overfit to the data.

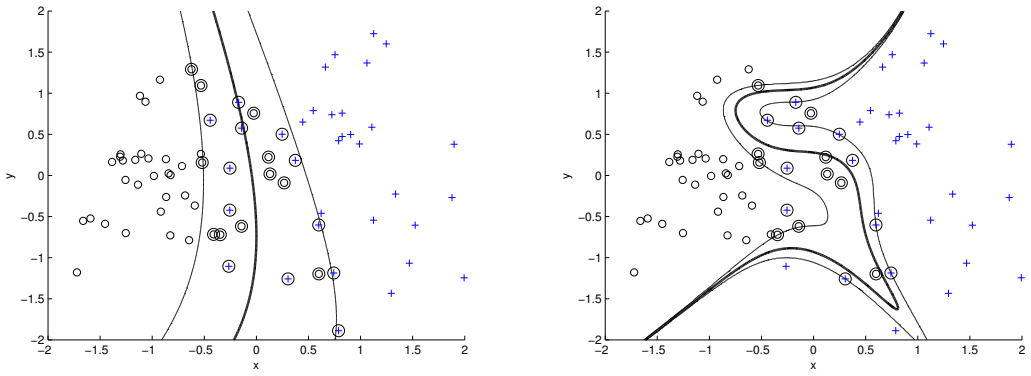
Polynomial Kernels

The polynomial kernel with degree q is written:

$$K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i \cdot \mathbf{x}_j)^q \,. \tag{4.2}$$

A polynomial SVM has a decision surface given by:

$$f(\mathbf{x}) = \sum_i \alpha_i y_i \, (1 + \mathbf{x}_i \cdot \mathbf{x})^q - b. \tag{4.3}$$



(a) $q = 2$ **(b)** $q = 6$

Figure 4.6: Polynomial C-SVMs with $C = 1$ while q is varied

Equation (4.3) is a degree q multivariate polynomial, which is easiest to visualize in lower dimensions. Figure (4.6a) shows a degree 2 polynomial SVM for two-dimensional data. Notice how the change in degree q results in higher degree polynomials which have a greater capacity to fit the data (Figure 4.6b). However, if the dimensionality of the input points d and also the polynomial degree q are both large, the number of terms in Equation (4.3) can become extremely large, making the decision surface complex enough that is has a good chance of overfitting the data.

4.2.4 Relationship with k-Means Classifiers

A k-means classifier is a simple classifier which predicts the class of a point based on the class mean it is closest to. For example, Figure 4.7 shows a k-means classifier making a prediction. For the case of supervised classification, which we refer to here, the k-means classifier is sometimes also referred to as an Euclidean distance classifier [111].

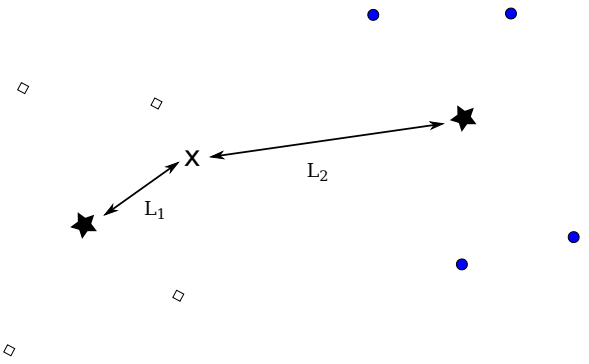


Figure 4.7: A k-means classifier predicts the class of the point marked X based on the nearest class centroid. In this case, $L_1 < L_2$, so the point is classified as belonging to the left-most class

The geometric interpretation makes it apparent that there is a relationship between the k-means classifier and SVMs. To see this, consider the case of a μ-SVM trained on a set of data of size n with an equal number of points in each class. In this case, choosing $\mu = 2/n$ means that the two classes will be reduced to their centroids, and a corresponding SVM can be given by setting all $\alpha_i = \mu$.

The decision surface associated with the SVM described above (combined with a geometric threshold) is shown in Figure 4.8. Notice how any point on this line must have an equal distance to the centroids of each class, meaning it is equivalent to the k-means decision boundary.

4.2.5 Relationship with k-Nearest Neighbor Classifiers

The k-nearest neighbor classifier is another simple type of supervised classifier which predicts the class of a point based on the class of its nearest k neighboring points. For example, in the case of $k = 1$, a point is always classified as belonging to the same class as its closest neighboring point.

Compare the plot of the decision surface of a 1-nearest neighbor classifier in Figure 4.9a to that of a Gaussian SVM (Figure 4.9b), trained with width $\gamma = 50$, regularization

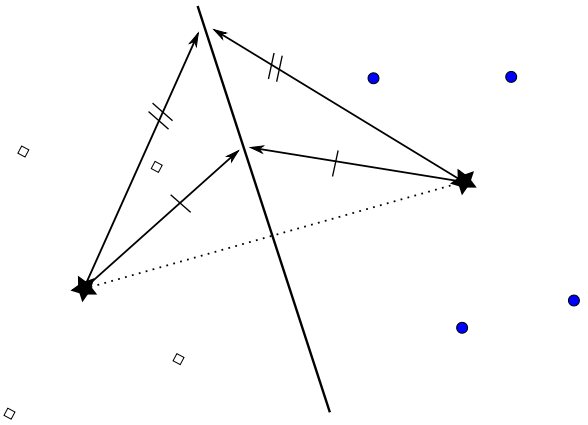
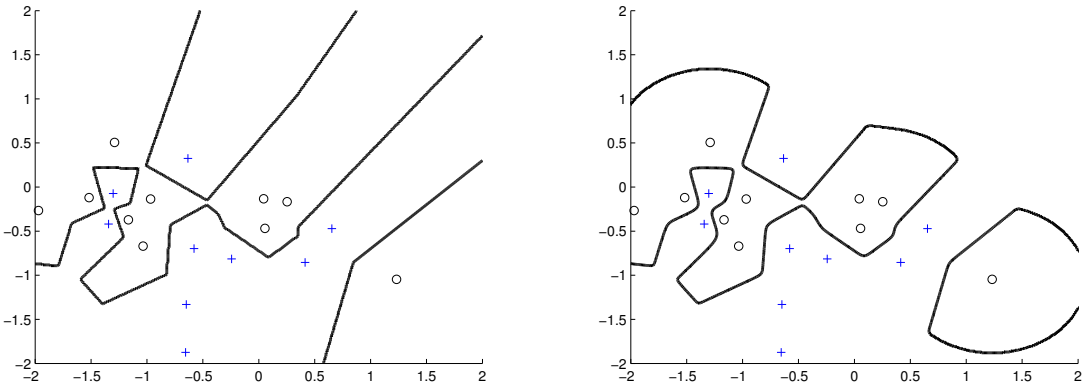


Figure 4.8: An SVM is equivalent to a k-means classifier under certain conditions

parameter $\mu = 1$, and with the threshold forced to $b = 0$. Notice how the decision surfaces are almost identical, with the exception of regions where there are no training points, where the SVM simply classifies everything as belonging to the positive class.



(a) 1-nearest neighbor classifier **(b)** SVM

Figure 4.9: A 1-nearest neighbor classifier compared to a Gaussian SVM with very large γ

The similarity between the decision surfaces of the Gaussian SVM and the 1-nearest neighbor classifier stems from a conceptual similarity, which can be examined by substituting the Gaussian kernel into the SVM decision function to form:

$$f(\mathbf{x}) = \sum_i \alpha_i y_i \exp\left(-\gamma ||\mathbf{x}_i - \mathbf{x}||^2\right) - b. \tag{4.4}$$

Notice how, when γ is large, the greatest influence on this decision function is likely to come from the support vector which is closest to $\mathbf{x}$.

Another reason why Gaussian SVMs with large γ share such a strong resemblance to a nearest neighbor classifier is that, for these large values of γ, it is also likely that every point will become a support vector. To understand why this is the case, recall that the KKT conditions of the μ-SVM dual (described in Section 2.5.1), combined with the

Gaussian decision function in (4.4) imply that we must have:

$$y_i \left[\sum_k \alpha_k y_k \exp\left(-\gamma ||\mathbf{x}_k - \mathbf{x}_i||^2\right) - b \right] = \rho \qquad \text{if } 0 < \alpha_i < \mu \tag{4.5}$$

$$y_i \left[\sum_k \alpha_k y_k \exp\left(-\gamma ||\mathbf{x}_k - \mathbf{x}_i||^2\right) - b \right] \geq \rho \qquad \text{if } \alpha_i = 0 \tag{4.6}$$

$$y_i \left[\sum_k \alpha_k y_k \exp\left(-\gamma ||\mathbf{x}_k - \mathbf{x}_i||^2\right) - b \right] < \rho \qquad \text{if } \alpha_i = \mu \tag{4.7}$$

Considering the extreme case as $\gamma \to \infty$, we must have:

$$\exp(-\gamma ||\mathbf{x}_k - \mathbf{x}_i||^2) \to \begin{cases} 0 & \text{if } k \neq i \\ 1 & \text{if } k = i \end{cases}$$

Given this property, and assuming there are no duplicate points in the training set, conditions (4.6) and (4.7) can not be satisfied and we must instead satisfy (4.5) for all points using the solution:

$$\alpha_i = \begin{cases} \frac{1}{n_{pos}} & \text{if } y_i = 1 \\ \frac{1}{n_{neg}} & \text{otherwise} \end{cases}$$
$$b = \frac{1}{2}\left(\frac{1}{n_{pos}} - \frac{1}{n_{neg}}\right)$$
$$\rho = \frac{1}{2}\left(\frac{1}{n_{pos}} + \frac{1}{n_{neg}}\right)$$

For this case the nearest points are the centroids of the two classes (in feature space). However, these centroids are not chosen because the hulls have been forced to reduce by using a small reduction parameter μ (as they were in the previous section). Instead, these centroids have arisen because the dimensionality of the feature space in this case is so high that *every point* must be a vertex in the facets of the two hulls containing the nearest points.

4.3 The Threshold

As described in previous sections, the threshold of an SVM, b, is the offset of the hyperplane from the origin. One of the ways to compute the threshold associated with an SVM is to derive it from the KKT conditions of the dual once α_i values have been optimized. Although this threshold, which we refer to as the KKT threshold, makes for a consistent optimization task, it is sub-optimal in several circumstances, which we elaborate on in this section. We also review an alternative geometric placement of the threshold [28], and explore its effectiveness. By performing empirical trials, we compare the geometric and KKT thresholds, and explore the strengths and weaknesses of both approaches.

We then examine a particular case where using an L_1-loss C-SVM in conjunction with the KKT threshold can result in a clearly sub-optimal threshold. We describe the circumstances leading to this threshold placement and how these conditions can be avoided. An example of this sub-optimal threshold placement is shown in Figure 4.10. Here the threshold places the hyperplane completely outside the range of the data, so that everything is classified as belonging to the larger class. Despite incorrectly classifying an entire class, this hyperplane and threshold satisfy all of the KKT conditions of the C-SVM optimization task.

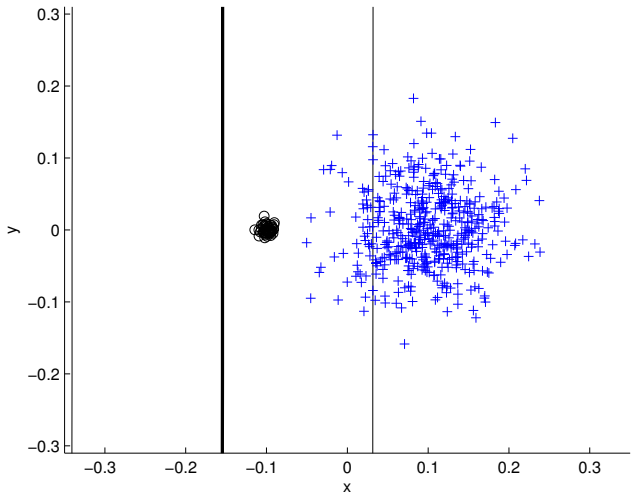


Figure 4.10: A linear C-SVM trained with $C = 1$. The dark line shows the decision surface of the SVM, while the thin lines show the supporting planes. There are 500 points in the positive class and 50 points in the negative class

4.3.1 The KKT Threshold

The KKT threshold is the 'standard' threshold of an SVM, as computed by most widely used SVM packages such as `SVMlight` [58] and `LIBSVM` [20]. This threshold can be computed for the L_1-loss μ-SVM using the KKT condition:

$$\alpha_i(y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - \rho + \xi_i) = 0.$$

Since α_i is zero when $||\mathbf{w} \cdot \mathbf{x} - b|| > \rho$, and ξ_i is zero unless $||\mathbf{w} \cdot \mathbf{x} - b|| < \rho$, b can be retrieved from this KKT condition by using two points $\mathbf{x}_{pos}, \mathbf{x}_{neg}$ (one from each class) for which $\alpha_{pos}, \alpha_{neg} > 0$ and $\xi_{pos}, \xi_{neg} = 0$ (i.e. a point from each class lying *on* a supporting plane). Such points can be identified in a μ-SVM because they satisfy $0 < \alpha_i < \mu$. Points $\mathbf{x}_{pos}$ and $\mathbf{x}_{neg}$ can be used to derive two equalities:

$$\mathbf{w} \cdot \mathbf{x}_{pos} - b = \rho \qquad\qquad \mathbf{w} \cdot \mathbf{x}_{neg} - b = -\rho,$$

These equalities are enough to solve the two unknown quantities ρ and b using:

$$b_{kkt} = \frac{1}{2}\left(\mathbf{w}\cdot\mathbf{x}_{pos} + \mathbf{w}\cdot\mathbf{x}_{neg}\right) \qquad \rho = \frac{1}{2}\left(\mathbf{w}\cdot\mathbf{x}_{pos} - \mathbf{w}\cdot\mathbf{x}_{neg}\right) \tag{4.8}$$

A geometric representation of the margin and the terms ρ and b is shown in Figure 4.11.

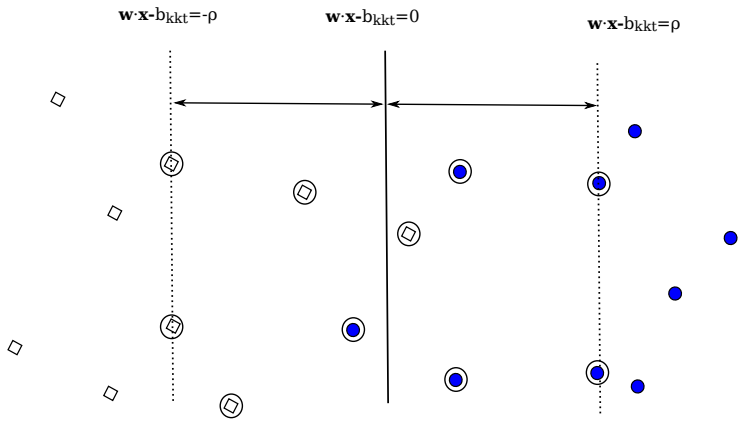


Figure 4.11: Choosing the threshold of an SVM using the KKT conditions

Under most circumstances, the threshold is simple to compute using Equation (4.8). However, it is possible for *no* $\mathbf{x}_{pos}$ and/or $\mathbf{x}_{neg}$ to exist. One such example is shown in Figure 4.12. This example shows a μ-SVM where all α_i's are capped at their maximum value of $\mu = 1/2$. For this case, b can take a range of possible values $b_{min} < b_{kkt} < b_{max}$, any of which satisfy the KKT conditions. When this occurs, b_{min} and b_{max} are given by:

$$b_{min} = \frac{1}{2}\left[\overbrace{\min_{i\in I_{neg},\alpha_i>0}\{\mathbf{w}\cdot\mathbf{x}_i\}}^{b_1} + \overbrace{\min_{i\in I_{pos},\alpha_i=0}\{\mathbf{w}\cdot\mathbf{x}_i\}}^{b_3}\right]$$

$$b_{max} = \frac{1}{2}\left[\overbrace{\max_{i\in I_{neg},\alpha_i=0}\{\mathbf{w}\cdot\mathbf{x}_i\}}^{b_2} + \overbrace{\max_{i\in I_{pos},\alpha_i>0}\{\mathbf{w}\cdot\mathbf{x}_i\}}^{b_4}\right]$$

The parts of these equations labeled b_1, b_2, b_3, b_4 are depicted in Figure 4.12 for a simple toy dataset. In this figure any threshold placing the hyperplane in the range indicated by the arrows will satisfy the KKT conditions. Notice that, although the threshold can take a range of possible values, the normal of the hyperplane does not change.

All thresholds within the possible range of values for b shown in Figure 4.12 appear reasonable at a glance. However, much worse cases can be encountered when there is an imbalance in class sizes. For example, consider the case where one class is reduced to its centroid but the other is not. When this occurs, ρ can grow arbitrarily large and the threshold can be shifted to such an extent that everything lies on one side of the hyperplane. An example of this is shown in Figure 4.13, where the only constraint on the

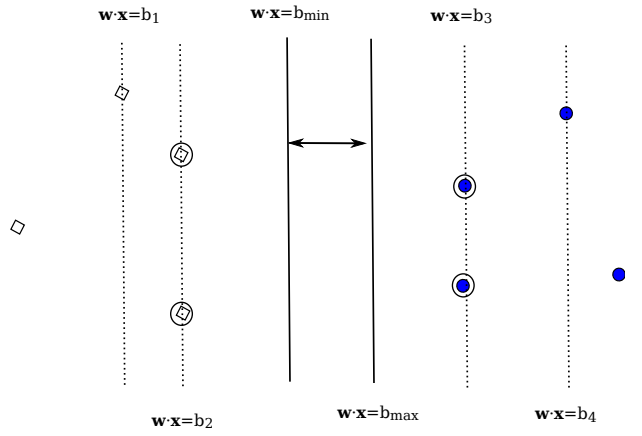


Figure 4.12: A range of thresholds satisfying the KKT conditions

threshold is given by $b < b'_{max}$, where:

$$b'_{max} = \frac{1}{2}\left[\overbrace{\min_{i \in I_{neg}, \alpha_i > 0} \{\mathbf{w} \cdot \mathbf{x}_i\}}^{b'_1} + \overbrace{\max_{i \in I_{pos}, \alpha_i > 0} \{\mathbf{w} \cdot \mathbf{x}_i\}}^{b'_2}\right]$$

Although a bad threshold in this case can be avoided by choosing the b corresponding to the smallest possible value of ρ, this is not always possible when the value of ρ is fixed, such as in the C-SVM optimization problem. This implication is discussed when the C-SVM threshold is addressed in Section 4.3.5.

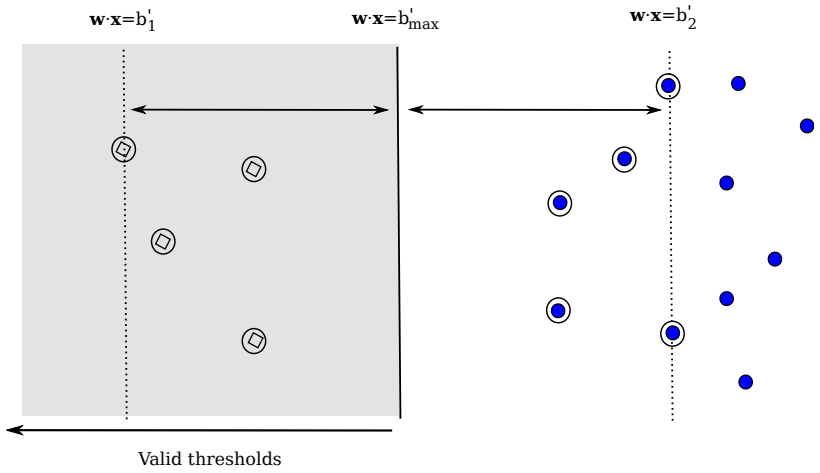


Figure 4.13: The threshold can take any value such that $b < b'_{max}$

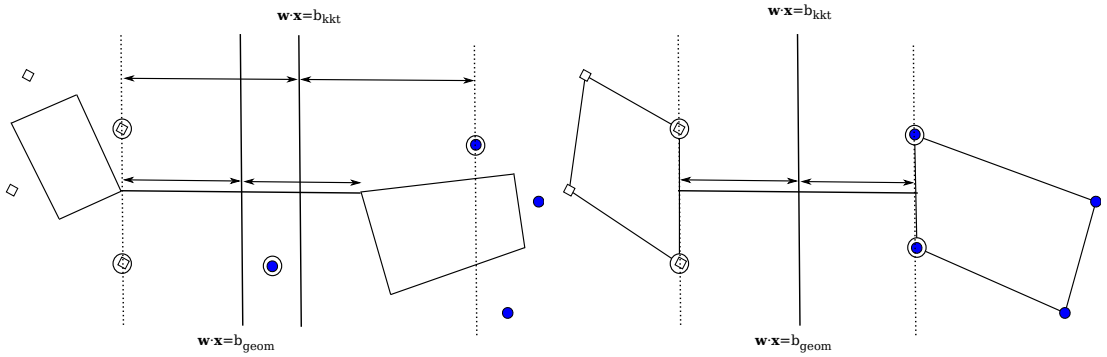
4.3.2 The Geometric Threshold

When considering the geometric interpretation of SVMs in terms of RCHs, the most intuitive choice of threshold places the hyperplane half way between the nearest points in the RCHs of the two classes. However, this choice is rarely the same as the threshold given by the KKT conditions [28].

The threshold which places the hyperplane halfway between the nearest points in the RCHs of the two classes is referred to as the geometric threshold. This threshold is given by [28]:

$$b_{geom} = \frac{1}{2}\sum_i \alpha_i \mathbf{w} \cdot \mathbf{x}_i. \tag{4.9}$$

One of the benefits of the geometric threshold is that it is less susceptible to being biased towards a class with a larger deviation. An example of a situation where this occurs is shown in Figure 4.14a. Although the KKT threshold is consistent with the SVM QP task, it is not necessarily going to be optimal in this case. The second benefit of a geometric threshold is that there is only one possible value for the threshold. In contrast to this, there are sometimes a range of potential thresholds which satisfy the KKT conditions, as we described in the previous section.



(a) Soft Margin μ-SVM with $\mu = 1/2$ **(b)** Hard Margin SVM

Figure 4.14: The geometric threshold compared to the KKT threshold

When an SVM is trained with $\mu = 1$, it becomes equivalent to a hard margin SVM. The hard margin SVM threshold is an interesting case because it results in a geometric and a KKT threshold that coincide (Figure 4.14b). The equivalence between the two thresholds arises because, for the hard margin SVM, all support vectors must lie on the supporting planes of the SVM. Because the nearest points in each class are formed from a convex combination of support vectors, they must also lie on the supporting planes of the SVM. It follows that placing the hyperplane half way between the nearest points in this case is equivalent to placing it half way between the supporting planes.

4.3.3 The Probabilistic Threshold

Platt [91] has previously made the observation that the KKT threshold can sometimes be less than optimal. Platt discovered this while proposing probabilistic outputs for SVMs, which take the form:

$$p(\mathbf{x}) = \frac{1}{1 + \exp(Af(\mathbf{x}) + B)}$$

Rather than simply outputting a positive or negative class prediction, $p(\mathbf{x})$ estimates the probability of $\mathbf{x}$ belonging to the positive class. A value of $p(\mathbf{x}) < 0.5$ suggests $\mathbf{x}$ belongs to the negative class, whereas a value of $p(\mathbf{x}) \geq 0.5$ suggests $\mathbf{x}$ belongs to the positive class. A and B are computed using the training data in conjunction with a maximum likelihood estimate. This method extends the standard functionality of an SVM, which does not associate certainties with its predictions.

As well as allowing SVMs to make approximate probabilistic predictions, Platt noted that the threshold trained using maximum likelihood was different to the threshold computed by the SVM, and sometimes superior in terms of the resulting test error [91].

4.3.4 Empirical Trials

In this section we compare using empirical trials the three thresholds described previously: the KKT threshold, the geometric threshold and the probabilistic threshold. Results are shown in their entirety in Appendix B. Because of the large number of combinations of kernels and parameters, there are a large number of tables in this appendix. A smaller subset of these results are described in this section.

The tables in this section show the mean test error for each threshold, calculated over 20 runs, with standard errors also shown. SVMs are trained using `SVMlight`. Because `SVMlight` uses the C-SVM parameterization, we use $C = 0.1$ to correspond to 'small μ', and $C = 100$ to correspond to 'large μ'. This variation is important because results could theoretically change depending on the regularization parameter which is chosen.

The KKT threshold is used as a basis for comparison in empirical trials. If an alternate threshold achieves an error which is significantly less ($p < 0.05$ using a paired differences t-test) than the error achieved by the KKT threshold, that result is shown in bold. Similarly, if a threshold results in a significantly higher error than the KKT threshold, that result is underlined.

Table 4.1 shows the mean test error for Gaussian SVMs ($\gamma = 0.01$) trained on each dataset using a particular threshold. Table 4.2 repeats these experiments for Gaussian SVMs with $\gamma = 0.1$ (smaller basis functions). It is informative to note that, in these tables, both the geometric and probabilistic thresholds generally resulted in an error which either equalled or bettered that of the KKT threshold. Platt [92] has previously suggested that the probabilistic threshold can provide an improvement over the standard KKT threshold, and our results are consistent with this observation. However, to the best of our knowledge it has not previously been noted that the geometric threshold can also provide an improvement over the KKT threshold. Interestingly, the KKT threshold, despite being one of the most widely used thresholds, generally offers the highest error rate compared with the other available thresholds.

For the Gaussian kernel, the geometric and probabilistic thresholds tended to provide a more consistent decrease in test error over the KKT threshold when μ values were small. This may be because the position of the KKT threshold is determined solely based on the unbounded support vectors of the machine. Recall from Section 2.4.1 that these are generally a small portion of the support vectors, and that they may all lie a long way from

Table 4.1: Error for Gaussian SVMs ($\gamma = 0.01$) combined with three possible thresholds

		KKT	Geometric	Probabilistic
banana	small μ	0.419 ± 0.012	0.428 ± 0.011	0.425 ± 0.013
	large μ	0.394 ± 0.012	0.394 ± 0.013	0.390 ± 0.014
b.cancer	small μ	0.274 ± 0.011	0.277 ± 0.008	0.271 ± 0.007
	large μ	0.275 ± 0.011	0.295 ± 0.010	0.281 ± 0.010
diabetes	small μ	0.269 ± 0.004	$\mathbf{0.241 \pm 0.005}$	$\mathbf{0.239 \pm 0.005}$
	large μ	0.260 ± 0.004	$\mathbf{0.247 \pm 0.004}$	$\mathbf{0.244 \pm 0.005}$
german	small μ	0.249 ± 0.005	$\mathbf{0.235 \pm 0.005}$	$\mathbf{0.235 \pm 0.005}$
	large μ	0.252 ± 0.005	0.250 ± 0.006	0.248 ± 0.006
heart	small μ	0.182 ± 0.009	$\mathbf{0.158 \pm 0.008}$	$\mathbf{0.158 \pm 0.007}$
	large μ	0.195 ± 0.008	0.187 ± 0.009	$\mathbf{0.185 \pm 0.009}$
image	small μ	0.265 ± 0.002	$\mathbf{0.208 \pm 0.004}$	$\mathbf{0.198 \pm 0.003}$
	large μ	0.160 ± 0.024	$\mathbf{0.131 \pm 0.018}$	$\mathbf{0.126 \pm 0.016}$
splice	small μ	0.160 ± 0.002	$\mathbf{0.158 \pm 0.002}$	$\mathbf{0.158 \pm 0.002}$
	large μ	0.136 ± 0.006	$\mathbf{0.134 \pm 0.006}$	$\mathbf{0.135 \pm 0.005}$
thyroid	small μ	0.223 ± 0.011	$\mathbf{0.133 \pm 0.009}$	$\mathbf{0.101 \pm 0.009}$
	large μ	0.145 ± 0.020	$\mathbf{0.095 \pm 0.012}$	$\mathbf{0.083 \pm 0.009}$
titanic	small μ	0.238 ± 0.007	0.226 ± 0.001	0.226 ± 0.001
	large μ	0.233 ± 0.005	0.227 ± 0.001	0.226 ± 0.001

the decision surface itself, particularly for small values of μ. By contrast, the geometric and probabilistic thresholds take all of the support vectors into account when determining the placement of the threshold.

Despite the decreaed error rate for the geometric threshold in conjunction with Gaussian kernels, the geometric threshold showed little improvement over the KKT threshold when applied to linear SVMs (Table 4.3). In this case the geometric threshold actually resulted in a significantly increased error rate in several cases. However, it is important to notice that in the cases where the geometric threshold increased the test error, this increase in error was quite small (less than 0.01 in absolute terms). By contrast, when the KKT threshold showed an increase in test error compared to the geometric threshold, the increase tended to be much larger than 0.01.

These experiments suggest that the geometric threshold is likely to provide a small decrease to test error when used in conjunction with Gaussian SVMs. Although results suggest that the geometric threshold will only rarely decrease the test error of linear SVMs, they also suggest that the geometric threshold tends to provide a *safer* threshold. By safe we mean that even when it is not the best threshold, its use is unlikely to increase the error rate by a large amount. By contrast, the KKT threshold was occasionally prone to very large increases in test error compared to the geometric threshold. We support this observation in the following section by examining some of the special cases in which the KKT threshold can fail to provide a reasonable error rate.

Table 4.2: Error for Gaussian SVMs ($\gamma = 0.1$) combined with three possible thresholds

		KKT	Geometric	Probabilistic
banana	small μ	0.372 ± 0.009	0.365 ± 0.014	0.363 ± 0.014
	large μ	0.248 ± 0.029	0.246 ± 0.029	0.245 ± 0.028
b.cancer	small μ	0.265 ± 0.010	0.264 ± 0.006	0.260 ± 0.008
	large μ	0.297 ± 0.012	0.304 ± 0.013	0.299 ± 0.012
diabetes	small μ	0.265 ± 0.006	$\mathbf{0.246 \pm 0.005}$	$\mathbf{0.243 \pm 0.004}$
	large μ	0.280 ± 0.006	0.273 ± 0.007	$\mathbf{0.271 \pm 0.007}$
german	small μ	0.276 ± 0.005	$\mathbf{0.238 \pm 0.005}$	$\mathbf{0.237 \pm 0.004}$
	large μ	0.271 ± 0.005	$\mathbf{0.251 \pm 0.006}$	$\mathbf{0.249 \pm 0.006}$
heart	small μ	0.215 ± 0.012	$\mathbf{0.163 \pm 0.008}$	$\mathbf{0.166 \pm 0.008}$
	large μ	0.218 ± 0.010	$\mathbf{0.192 \pm 0.010}$	$\mathbf{0.193 \pm 0.010}$
image	small μ	0.119 ± 0.002	$\mathbf{0.108 \pm 0.002}$	$\mathbf{0.106 \pm 0.002}$
	large μ	0.075 ± 0.010	$\mathbf{0.069 \pm 0.009}$	$\mathbf{0.069 \pm 0.009}$
splice	small μ	0.454 ± 0.006	$\mathbf{0.387 \pm 0.003}$	$\mathbf{0.373 \pm 0.005}$
	large μ	0.418 ± 0.009	$\mathbf{0.384 \pm 0.003}$	$\mathbf{0.378 \pm 0.004}$
thyroid	small μ	0.101 ± 0.007	$\mathbf{0.081 \pm 0.006}$	$\mathbf{0.057 \pm 0.007}$
	large μ	0.072 ± 0.009	$\mathbf{0.061 \pm 0.007}$	$\mathbf{0.052 \pm 0.006}$
titanic	small μ	0.229 ± 0.001	0.227 ± 0.001	0.229 ± 0.001
	large μ	0.227 ± 0.002	0.225 ± 0.001	0.226 ± 0.002

4.3.5 Bad Thresholds

In the previous section we found evidence to suggest that choosing the KKT threshold over other thresholds can result in a small increase in the error rate for some datasets and kernels. In this section we look at a particular case of extremely high error that can result from using the KKT threshold in conjunction with a C-SVM trained under certain parameters. This result is important because C-SVMs are often used in conjunction with the KKT threshold, including by some of the most frequently used SVM packages such as `LIBSVM` [20] and `SVMlight` [58].

Because there is no ρ variable associated with the C-SVM optimization task[1], the margin is fixed to equal:

$$\Delta = \frac{1}{||\mathbf{w}||}.$$

Accordingly, the threshold can be computed given only one point for which $0 < \alpha_i < C$. Given such a point, the threshold is calculated as:

$$b = \mathbf{w} \cdot \mathbf{x}_i - y_i.$$

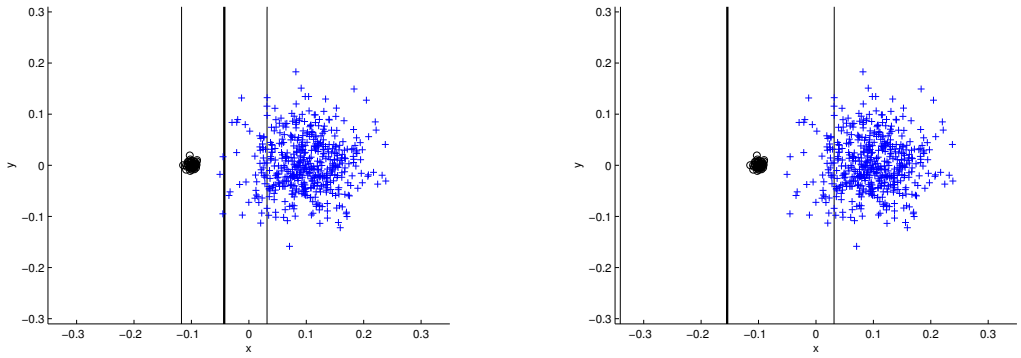
The lack of a ρ parameter in the C-SVM formulation means that there is a greater potential for rather unintuitive thresholds to be chosen when C becomes small, particularly if class sizes and/or distributions are imbalanced. For example, Figure 4.15a shows the threshold for a C-SVM with $C = 2.5$. Here one of the classes has been reduced to

[1] Refer to Section 2.5.1 for the definition of ρ, and Section 4.3.1 for an explanation of how ρ impacts the margin

Table 4.3: Error for linear SVMs combined with three possible thresholds

		KKT	Geometric	Probabilistic
banana	small μ	0.485 ± 0.010	0.481 ± 0.011	0.484 ± 0.011
	large μ	0.497 ± 0.012	$\mathbf{0.485 \pm 0.012}$	0.486 ± 0.011
b.cancer	small μ	0.292 ± 0.010	0.286 ± 0.007	0.290 ± 0.009
	large μ	0.294 ± 0.009	0.294 ± 0.008	0.293 ± 0.009
diabetes	small μ	0.234 ± 0.004	0.238 ± 0.005	0.233 ± 0.004
	large μ	0.234 ± 0.004	0.239 ± 0.005	0.233 ± 0.004
german	small μ	0.236 ± 0.005	0.243 ± 0.005	0.234 ± 0.005
	large μ	0.241 ± 0.010	0.248 ± 0.009	0.236 ± 0.006
heart	small μ	0.162 ± 0.007	0.166 ± 0.007	0.164 ± 0.007
	large μ	0.169 ± 0.006	0.173 ± 0.006	0.169 ± 0.006
image	small μ	0.161 ± 0.002	0.166 ± 0.003	0.161 ± 0.003
	large μ	0.157 ± 0.002	0.162 ± 0.003	0.156 ± 0.003
splice	small μ	0.164 ± 0.001	0.164 ± 0.001	0.164 ± 0.001
	large μ	0.164 ± 0.002	0.165 ± 0.002	0.164 ± 0.001
thyroid	small μ	0.154 ± 0.009	$\mathbf{0.105 \pm 0.009}$	$\mathbf{0.115 \pm 0.009}$
	large μ	0.130 ± 0.010	$\mathbf{0.110 \pm 0.008}$	$\mathbf{0.113 \pm 0.008}$
titanic	small μ	0.225 ± 0.001	0.225 ± 0.001	0.225 ± 0.001
	large μ	0.225 ± 0.001	0.225 ± 0.001	0.225 ± 0.001

its centroid, meaning this class contains no points satisfying $0 < \alpha_i < C$. If C becomes smaller, as in Figure 4.15b, the margin is forced to become larger, which forces the threshold towards the class which has been reduced to its centroid. Eventually, the threshold will be skewed to such an extent that all points will be classified as belonging to the same class, as in Figure 4.15b.



(a) $C = 2.5$ **(b)** $C = 1$

Figure 4.15: Threshold as C decreases

Although the threshold shown in Figure 4.15b is a valid KKT threshold for the μ-SVM, it would generally not be chosen since it would require an arbitrarily large ρ to be chosen. However, in a C-SVM, ρ does not exist and so small choices of C can force the KKT threshold to be placed outside the range of both of the classes.

It is interesting to note that the KKT threshold, despite its shortcomings, is still widely used in common software packages. One explanation for why this is the case may lie in the fact that SVMs are often coupled with extensive parameter selection. Rather than simply

training a single SVM on a classification task, there are generally a large number of SVMs trained with differing parameter values. The SVMs which are unlikely to perform well are then filtered out using error estimation techniques. It may be that this process, as well as having the ability to filter out parameter values which provide a poor error rate, also inadvertently filters out poor thresholds. We will further explore the role of parameter selection in SVM classification in Chapter 6.

4.4 Incorporating Weights into the Geometric Framework

Weighted SVMs allow the user to set the importance of each point in terms of the contribution it should make to the final decision surface. We described the WSVM optimization problem in Section 2.6, and reviewed how WSVMs have been applied to problems which involve cost-proportionate weights or unequal misclassification costs. We then introduced the related concept of Weighted RCHs in Section 3.9. In this section we use Weighted RCHs in order to understand how WSVMs work and to describe several cases when the introduction of weights can be beneficial.

In particular, we describe the geometric implications of three existing strategies for applying Weighted SVMs to classification problems. These three strategies are: a) minimizing the overall error, b) minimizing the average proportion of errors [118], and c) minimizing the total error cost [129]. When class sizes are exactly balanced (i.e. 50 percent of training and testing data belongs to each class and the cost of misclassification for each class is identical), these strategies are all equivalent. However, when the number of points in each class, or the misclassification costs of each class become unbalanced, each strategy results in a unique outcome, which we describe in more detail in the following sections.

In order to demonstrate the strengths of each weighting strategy, we perform some empirical demonstrations using as unbalanced toy dataset, which we refer to as `unbal`. This dataset is generated from two overlapping Gaussian distributions (Figure 4.16), with 200 points in the positive class, and 1000 points in the negative class. The positive class is assumed to have an error cost which is 10 times greater than that of the negative class.

4.4.1 A Geometric Interpretation of WSVMs

By exploiting the concept of WRCHs, which we introduced in Section 3.9, we can show that there is also a geometric interpretation of WSVMs. This geometric interpretation of WSVMs in terms of WRCHs follows intuitively from the relationship between RCHs and SVMs. To see this, consider a geometric reparameterization of the WSVM from Section

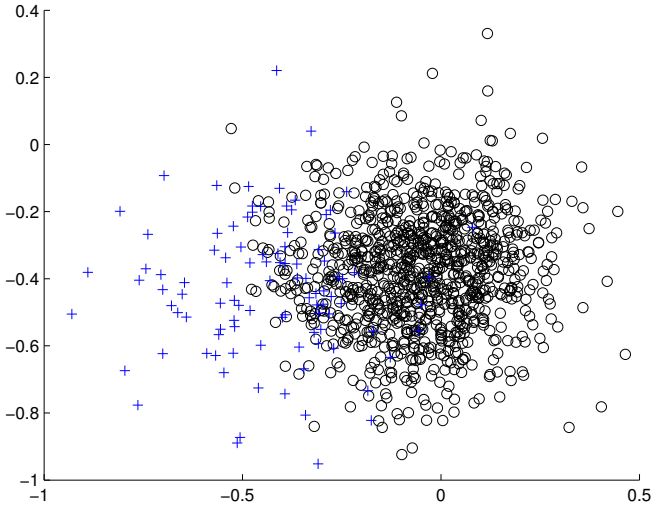


Figure 4.16: The `unbal` dataset

2.6:

$$\begin{aligned} \min_{\mathbf{w},b,\rho,\xi} \quad & ||\mathbf{w}||^2 - 2\rho + \mu \sum_{i=1}^{n} s_i \xi_i, \\ \text{subject to} \quad & \begin{cases} y_i(\mathbf{w}\cdot\mathbf{x}_i - b) & \geq \rho - \xi_i \\ \xi_i & \geq 0 \\ \rho & > 0. \end{cases} \end{aligned} \tag{4.10}$$

This is essentially an L_1-loss μ-SVM from Section 2.5.1, where an additional weight s_i has been introduced for each point. This optimization task has the dual form:

$$\begin{aligned} \max_{\alpha_i,\ldots,\alpha_n} \quad & -\frac{1}{4}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j, \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i y_i = 0 \\ \sum_{i=1}^{n} \alpha_i = 2 \\ 0 \leq \alpha_i \leq s_i \mu \end{cases} \end{aligned} \tag{4.11}$$

Now compare this dual to the squared distance between the WRCHs of two classes. From the definition of a WRCH in Section 3.9.1, this distance is written:

$$\begin{aligned} D^2 &= || \sum_{i\in I_{pos}} y_i \alpha_i \mathbf{x}_i + \sum_{i\in I_{neg}} y_i \alpha_i \mathbf{x}_i||^2 \\ &= || \sum_{i} y_i \alpha_i \mathbf{x}_i||^2 \\ &= \sum_{i,j} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j, \end{aligned}$$

under the constraints $0 \leq \alpha_i \leq \mu s_i$, $\sum_{i \in I_{pos}} \alpha_i = 1$, $\sum_{i \in I_{neg}} \alpha_i = 1$. Notice that these constraints are equivalent to those used in the weighted μ-SVM dual above, and minimizing D^2 is equivalent to optimizing the objective function.

This relationship provides an intuitive geometric interpretation of WSVMs. A WSVM is equivalent to the perpendicular bisector of the shortest line between the WRCHs of the two classes. The weights s_i applied to the WRCHs are the same as the weights s_i that are applied to the WSVM.

The relationship between WSVMs and WRCHs enables us to reason about the behavior of WSVMs using the properties of WRCHs. For example, Figure 4.17 demonstrates the role of the overall reduction parameter combined with the individual point weights. Notice how, as the reduction parameter shrinks, the point with the larger weight is enclosed by the hull much longer than points with smaller weights. Even when a point with a larger weight is not enclosed by the hull, it still exerts a much greater influence on the hull than other points, making the hull reduce towards a weighted centroid as opposed to a standard centroid.

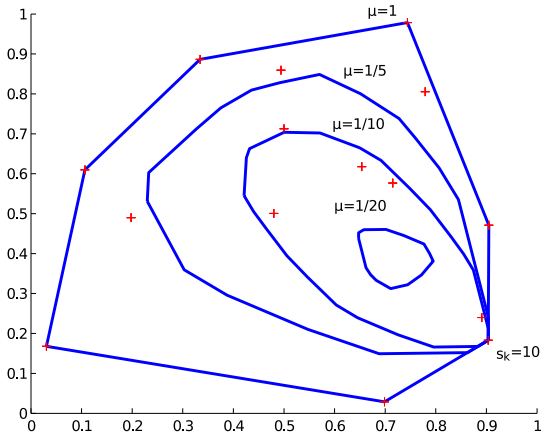


Figure 4.17: Point $\mathbf{x}_k$ is given a weight of $s_k = 10$ while all other points are given a weight of $s_i = 1$. Notice how $\mathbf{x}_k$ remains enclosed by the hull until μ becomes less than $1/10$.

We will exploit the geometric interpretation of WSVMs in the following sections in order to understand better some of the WSVM weighting schemes that have been previously employed. We achieve this by examining the impact the various weighting schemes have on the WRCHs of the two classes, as well as the resulting impact on the decision surface itself.

4.4.2 Minimizing the Overall Error

Minimizing the overall error (using the sum of slack variables as a proxy) is the default functionality of an SVM and so point weighting is not required. A classifier which minimizes the overall error tends to correctly classify a greater proportion of points from the larger class, while a greater proportion of points from the smaller class will be misclassified. This is not necessarily incorrect behavior. In effect, it means that a greater proportion of

the test data will be classified as belonging to the larger class, mimicking the imbalance in the training data. This is likely the desired behavior if the test data is expected to contain the same imbalance as the training data.

Figure 4.18 shows the manner in which the RCHs of two unbalanced classes are reduced when an unweighted SVM is trained on the `unbal` dataset. Recall that, on this dataset, the positive class contains 5 times as many points than the negative class. Because both classes share the same reduction coefficient μ, the larger class ends up being reduced by a much smaller proportion than the smaller class. This means that the final hyperplane separating these two classes will favor the larger class in its predictions.

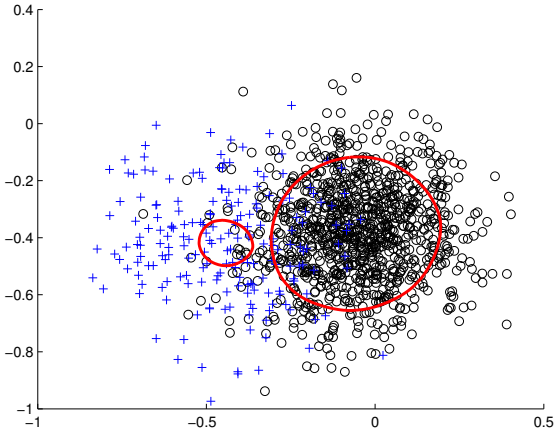


Figure 4.18: Reduction in two RCHs when an unweighted μ-SVM with $\mu = 1/150$ is trained on the `unbal` dataset. Reduced convex hulls are outlined in red.

4.4.3 Minimizing the Average Proportion of Errors

It may occur that imbalances in the training data are not expected to reoccur in the test data. Under these circumstances, a more sensible option is to minimize the average proportion of errors across both classes, rather than simply the raw number of errors. This means that if the proportion of errors is p^+ for the positive class and p^- for the negative class, we would aim to minimize $(p^+ + p^-)/2$. Some authors refer to this as the 'balanced loss' [112, 127]. When classes in the test data are expected to have an equal number of points (whereas classes in the training data do not), this strategy minimizes the overall expected error on test data.

This strategy is implemented in conjunction with SVMs by training a Weighted SVM in which each class is given a weight proportional to its size. For example, if the positive class is has k times as many training points as the negative class, weights should be set so that:

$$s_i = \begin{cases} 1 & \text{if } i \in I_{pos} \\ \text{k} & \text{if } i \in I_{neg} \end{cases} \tag{4.12}$$

The weighting described by Equation (4.12) is equivalent to reducing each hull by an amount proportional to its size. An example of an SVM trained using these weights is shown in Figure 4.19. Notice in this figure how each hull is approximately the same size, reflecting the similar variances of the Gaussian distributions representing the two classes. It is important to note that this weighting scheme is not necessarily an improvement on the unweighted method. Rather, the correct scheme to use should be dictated by the characteristics of the data.

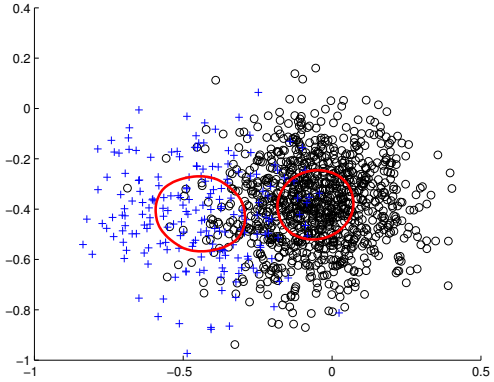


Figure 4.19: The positive class is reduced using $\mu = 1/100$, while the negative class is reduced using $\mu = 1/500$. This corresponds to a 50% reduction in both classes.

4.4.4 Minimizing the Total Cost of Error

Understanding WSVM point weighting schemes from a geometric perspective is made possible by exploiting the geometric interpretation of WSVMs in terms of WRCHs. When a point in a WRCH is assigned a large weight, its influence persists longer than a point with a lesser weighting. This means that highly weighted points will remain enclosed by the hull until the reduction parameter becomes small. By contrast, points with a lesser weight are much less likely to be enclosed by the hull, particularly as the reduction parameter becomes smaller.

Within the context of cost-proportionate weights, points with large weights are those which are costly to misclassify. Taking Zadrozny et al.'s [129] example of donor solicitation, training points with large weights would be large donors, which result in significant monetary losses if misclassified. In this case, training points associated with these large donors are forced to fall within the WRCH of their class (Figure 4.20). In turn, the WSVM hyperplane is forced to classify correctly training points with large weights. Unless a very small overall reduction parameter μ is used, only small donors or non-donors are likely to be allowed to fall outside the WRCH of their class.

4.4.5 An Empirical Demonstration

The strengths and weaknesses of each of the weighting schemes we have described can be illustrated using an empirical comparison. Table 4.4 shows results on the `unbal` dataset

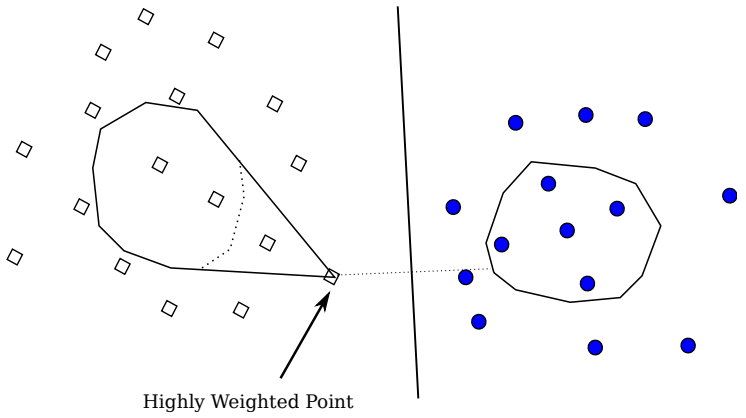


Figure 4.20: Highly weighted points in a WSVM are forced to be correctly classified by the hyperplane. The solid outlines border the WRCHs of the two classes, with one point given a large weight. The dotted line borders an RCH, where points can not be individually weighted.

for each of these weighting schemes. In this table, 'Overall' refers to the overall error minimization method, which assigns each training point an equal weight. 'Avg Prop' refers to class proportionate weights which minimize the average proportion of errors. 'Cost' refers to cost proportionate weighting.

Error rates and costs in this table are computed as an average over 100 runs, with standard errors also shown. Each run is performed using 1200 training points and a separate test set of 1200 points. Both training and test sets contain 1000 points in the positive class and 200 points in the negative class, all generated synthetically. Bolded error rates are significantly lower than other error rates in that row ($p < 0.05$ using a paired differences t-test across the 100 runs).

For the regularization parameter in these experiments we use $\mu = 1/(0.9\kappa)$, where κ is the sum of weights in the smaller (by weight) class:

$$\kappa = \min\left(\sum_{i \in I_{pos}} s_i, \sum_{i \in I_{neg}} s_i\right)$$

We define μ in terms of κ to allow a single regularization parameter to be applied to many different weighting schemes. Without this use of κ, a change to the weighting scheme is likely to cause a previously valid regularization parameter to become invalid due to either overlapping hulls, or one of the WRCHs being reduced to such an extent that it becomes empty.

Our definition of κ means that a reduction of $\mu = 1/(0.9\kappa)$ causes around 90% of training points in each class (depending on weighting and class balance) to be used as support points for each vertex. We use this value of μ largely for convenience, ensuring the hulls are separable across all datasets and weighting schemes that we apply.

Notice how each strategy in Table 4.4 optimizes a different error measurement. For example, using equal weights (the 'Overall' method) results in the best overall error rate by a significant margin. This is achieved by favoring the negative class, since it has a larger

Table 4.4: Error for Gaussian SVMs, with $\gamma = 0.01$ and $\mu = 1/(0.9\kappa)$, combined with three different weighting strategies. The geometric threshold is used.

	Overall	**Avg Prop**	**Cost**
Overall Err	$\mathbf{0.090 \pm 0.001}$	0.141 ± 0.001	0.213 ± 0.002
Err Cost	710.6 ± 8.0	454.6 ± 6.1	$\mathbf{419.4 \pm 4.4}$
Pos Class Err	0.335 ± 0.004	0.159 ± 0.003	$\mathbf{0.091 \pm 0.002}$
Neg Class Err	$\mathbf{0.042 \pm 0.001}$	0.137 ± 0.001	0.237 ± 0.002
Average Class Err	0.188 ± 0.002	$\mathbf{0.148 \pm 0.002}$	0.164 ± 0.001

number of points than the positive class. However, by doing so it achieves the highest error rate of all methods on the positive class.

Using weights proportional to class sizes (the 'Avg Prop' method), the error in the positive class is higher than that achieved by the cost-proportionate weighting method. Similarly, the error rate in the negative class is higher than that achieved when using equal weights. However, the average proportion of correctly classified points is maximized. This is achieved by ensuring a comparable error rate in *both* the positive and negative classes, rather than favoring a single class.

When cost-proportionate weights are used, the positive class (due to its large misclassification cost) is assigned a much greater importance than the negative class. For this reason, using cost-proportionate weights results in the lowest error rate for the positive class, but the highest error rate for the negative class. Accordingly, the cost-proportionate weighting method achieves a lower overall error cost than any other method.

4.4.6 Choosing the Threshold for WSVMs

For the results in Table 4.4 we used the geometric threshold. However, there is still a choice of threshold available for use in conjunction with each of these weighting schemes.

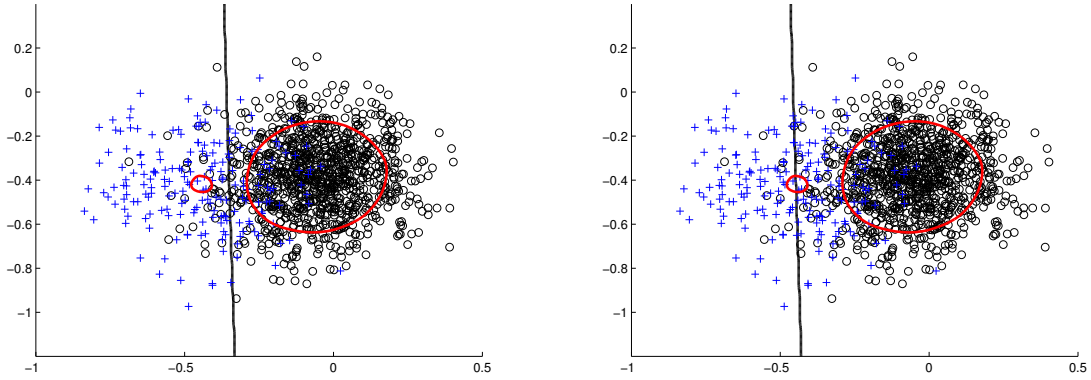
Table 4.5 repeats the experiments from the previous section using the KKT threshold in place of the geometric threshold. The three methods continue to optimize the intended quantity when using the KKT threshold. However, notice how almost every method has its error rate increased by using the KKT threshold. For example, the optimal error cost using both thresholds occurs when cost-proportionate weights are used. However, using the geometric threshold decreases the error cost by over 30 units. Similarly, the average overall error using the geometric threshold was slightly lower than that using KKT threshold. Not only was this an improvement in itself, but the geometric threshold also provides a much more desirable class error rate in conjunction with this decreased error.

We believe that the issue causing the KKT threshold to perform poorly in conjunction with WSVMs is that it may ignore class boundaries given by the WRCHs of the two classes. For example, Figure 4.21 compares the KKT and geometric thresholds when the overall error is minimized (i.e. all weights are equal to one). Notice how, although the orientation of the hyperplanes is the same, the geometric threshold places the hyperplane directly between the two RCHs, whereas the KKT threshold cuts into one of the RCHs. This demonstrates that the KKT threshold, unlike the geometric threshold, does not cut

Table 4.5: Error for Gaussian SVMs, with $\gamma = 0.01$ and $\mu = 1/(0.9\kappa)$, combined with three different weighting strategies. The KKT threshold is used.

	Overall	**Avg Prop**	**Cost**
Overall Err	$\mathbf{0.097 \pm 0.001}$	0.129 ± 0.001	0.300 ± 0.002
Err Cost	1067.1 ± 8.6	473.8 ± 6.8	$\mathbf{452.4 \pm 3.4}$
Pos Class Err	0.528 ± 0.004	0.177 ± 0.004	$\mathbf{0.051 \pm 0.002}$
Neg Class Err	$\mathbf{0.011 \pm 0.000}$	0.120 ± 0.002	0.350 ± 0.003
Average Class Err	0.270 ± 0.002	$\mathbf{0.148 \pm 0.002}$	0.201 ± 0.001

the shortest line between the WRCHs into two equal parts. The result in this case is a hyperplane which tends to favor the negative class, as consistent with the results in Table 4.5.



(a) Geometric threshold **(b)** KKT threshold

Figure 4.21: The threshold for unbalanced classes with all weights set to equal one

4.5 Perceptrons under the Geometric Framework

In Section 2.8 we described the general form of the perceptron dual optimization task, from which all hard and soft margin perceptrons can be derived:

$$\begin{aligned} \max_{\alpha_1,\ldots,\alpha_n} \quad & -\frac{1}{2}\sum_{i,j=1}^{n} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^{n} \alpha_i, \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \end{aligned} \tag{4.13}$$

We asserted that the characteristic trait possessed by all perceptrons is that their dual does *not* constrain the sum of Lagrange multipliers in each class to be equal. This means that the perceptron decision surface is not perpendicular to the shortest line between two convex or reduced convex hulls.

Since perceptrons can not always be geometrically interpreted in the same manner as SVMs, it raises the question of whether they have an alternate geometric interpretation. It has previously been pointed out that hard margin and L_2-loss perceptrons have a

geometric interpretation in terms of a minimal norm problem over a single convex hull [65, 39]. In this section we extend this interpretation to the case of general L_1-loss and weighted perceptrons.

4.5.1 A Geometric Interpretation of Perceptrons

Because perceptrons do not share the same geometric interpretation as SVMs, it is difficult to understand exactly how they work. However, perceptrons do have a geometric interpretation of their own. To see this, introduce a ρ term to the perceptron primal to yield a geometric reparameterization of the perceptron primal [113]:

$$\begin{aligned} \min_{\mathbf{w},b,\rho,\xi} \quad & \frac{1}{2}||\mathbf{w}||^2 + \frac{1}{2}b^2 - \rho + \mu \sum_{i=1}^{n} \xi_i, \\ \text{subject to} \quad & \begin{cases} y_i(\mathbf{w} \cdot \mathbf{x}_i - b) & \geq \rho - \xi_i \\ \xi_i & \geq 0 \\ \rho & > 0. \end{cases} \end{aligned}$$

This primal has dual form:

$$\begin{aligned} \max_{\alpha} \quad & -\frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i & = 1 \\ 0 \leq \alpha_i \leq \mu, \ i & = 1, \ldots, n. \end{cases} \end{aligned} \tag{4.14}$$

Notice how adding the ρ term to the primal forces the sum of α_i's to be constrained to equal one, which makes it easier to form a geometric interpretation.

This reparameterized perceptron has a general form:

$$\begin{aligned} \max_{\alpha} \quad & -\frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i & = 1 \\ 0 \leq \alpha_i \leq \mu, \ i & = 1, \ldots, n. \end{cases} \end{aligned} \tag{4.15}$$

where $k(\mathbf{x}_i, \mathbf{x}_j) = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) + y_i y_j$ makes the machine equivalent to (4.14). Because this is simply a reparameterized version of the perceptron described in Section 2.8, the additional kernels we described in Table 2.2 can also be applied to this machine in order to make it an L_2-loss or hard margin perceptron.

Tsang et al. [113] has noted that the optimization task in (4.15) is equivalent to the MEB dual when $K(\mathbf{x}_i, \mathbf{x}_j) = c$ is constant (i.e. the Gaussian kernel). However, in a more general sense (for any kernel), this task is also equivalent to a minimal norm task over an

RCH. To see this, consider a point $\mathbf{a}$ in the RCH of all training points:

$$\begin{aligned} &\mathbf{a} \in \text{RCH}(\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}, \mu) \\ &\mathbf{a} = \sum_i \alpha_i \mathbf{x}_i, \quad \sum_i \alpha_i = 1, \ 0 \leq \alpha_i \leq \mu. \end{aligned} \tag{4.16}$$

The squared norm of $\mathbf{a}$ can be written:

$$||\mathbf{a}||^2 = \sum_{ij} \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j.$$

Notice that this is the same quantity that is minimized (in a kernel feature space) by the perceptron dual, under the constraints from (4.16). This means that the perceptron dual is equivalent to finding the point in an RCH closest to zero, i.e. the point in an RCH with minimal norm (Figure 4.22).

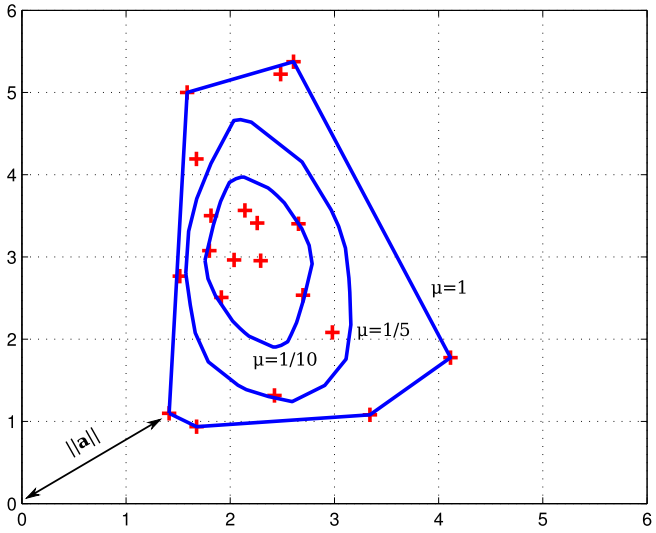


Figure 4.22: The perceptron dual is equivalent to a minimal norm problem over an RCH

We should emphasize that we are not claiming that it is impossible to represent an SVM as a one-class nearest point task. Indeed, using a type of set known as a *Minkowski* set it is possible to solve an SVM using a one-class approach [65]. We will further explore this technique in Chapter 5. However, it is *not* possible to solve the perceptron optimization task using the two-class nearest point approach taken in SVMs. This is due to the fact that the sum of Lagrange multipliers in each class is not constrained to equality. This means that the two-class nearest point approach is unique to SVMs. The one-class approach, while it is naturally suited to the perceptron optimization task in (4.15), can be taken when training either SVMs or perceptrons.

4.5.2 Perceptrons with Weighted Training Data

Although it has not, to the best of our knowledge, been considered, perceptrons can also handle individual training point weighting in much the same way that SVMs can. For example, let us modify the L_1-loss perceptron primal to form the optimization task to add weight terms s_i:

$$\min_{\mathbf{w},b,\rho,\xi} \quad \frac{1}{2}||\mathbf{w}||^2 + \frac{1}{2}b^2 - \rho + \mu\sum_{i=1}^{n} s_i\xi_i,$$

$$\text{subject to} \quad \begin{cases} y_i(\mathbf{w}\cdot\mathbf{x}_i - b) & \geq \rho - \xi_i \\ \xi_i & \geq 0 \\ \rho & > 0. \end{cases}$$

This optimization task has the dual form:

$$\max_{\alpha} \quad -\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{x}_i\cdot\mathbf{x}_j - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j$$

$$\text{subject to} \quad \begin{cases} \sum_{i=1}^{n}\alpha_i & = 1 \\ 0 \leq \alpha_i \leq s_i\mu, \ i & = 1,\ldots,n. \end{cases}$$

Conveniently, introducing weights to the L_1-loss perceptron primal only changes the upper bounds on the α_i values in the dual. This means that a weighted perceptron retains the geometric interpretation depicted in Figure 4.22, where WRCHs have been substituted for RCHs. In other words, a perceptron may be trained by calculating the point in a WRCH which is nearest to zero.

4.6 Conclusions

In this chapter we have used the geometric interpretation in order to build on the understanding of SVMs. For example, we have described how the choice of kernel and kernel parameters impacts the final decision surface. We have also examined the relationships between SVMs and other classifiers, showing that SVMs are often similar, or even equivalent to, other classifiers such as the k-nearest neighbor and k-means classifiers.

One of the most important issues we have uncovered in this chapter is that the most widely used threshold, the KKT threshold, in many cases yields a higher error rate than the geometric threshold. For example, in Section 4.3, we showed that when the KKT threshold was combined with some parameters and kernels, it degraded the overall error rate of the machine. Further, in Section 4.4.6, we showed that the KKT threshold can have a similar negative impact on Weighted SVMs. Furthermore, the KKT threshold, when used in conjunction with the C-SVM with certain (usually small) values of C, was capable of complete failure, classifying everything as belonging to a single class. By contrast, the geometric threshold did not suffer from this issue. Accordingly, we recommend that the geometric threshold be used in most cases, since there are few drawbacks to its use.

By exploiting the concept of WRCHs introduced in the previous chapter we have provided an intuitive geometric interpretation of WSVM classifiers. The geometric interpretation of a WSVM is that it is the perpendicular bisector of the shortest line between the WRCHs of two classes. This means that a WSVM has the same nearest point interpretation as an SVM, except that a WRCH is used to represent the classes instead of a standard RCH. In the next chapter we will exploit this geometric interpretation of WSVMs in order to apply nearest point algorithms to train WSVMs.

It has previously been shown that an L_2-loss perceptron is equivalent to a minimal norm problem over a convex hull [65, 39]. We have generalized this result to show that any type of perceptron, either L_1 or L_2-loss, with optional weights, may be trained as a minimal norm problem over a single WRCH. This result helps to clarify the relationship between SVMs and perceptrons. We assert that the definition of an SVM is that it can be computed as a nearest point problem over two WRCHs. This definition includes both L_1 and L_2-loss SVMs, and also their weighted variants. By contrast, the definition of a perceptron is a machine that can be computed as a minimal norm problem over a single WRCH. This definition includes both L_1 and L_2-loss perceptrons with optional weights.

Chapter 5

Geometric Training Algorithms

5.1 Introduction

In this chapter we investigate SVM training algorithms which exploit the geometric interpretation. SVM training algorithms which take this approach are referred to as nearest point algorithms, since they compute the nearest points in two convex sets. There has been a significant amount of previous study into using nearest point algorithms for training SVMs. Most of this work concentrates on training L_2-loss and hard margin SVMs over convex hulls [39, 80, 65], although some authors have also proposed L_1-loss SVM training algorithms which operate over Reduced Convex Hulls (RCHs) [81, 109].

We distinguish our research from previous work by studying the application of nearest point algorithms mainly to Weighted Reduced Convex Hulls (WRCHs) in order to train Weighted SVMs (WSVMs). The benefit of this approach is that the resulting algorithms, as well as being able to train SVMs, can also be used to train either L_1 or L_2-loss WSVMs. In studying the application of nearest point algorithms to WRCHs, we are also able to obtain several general results regarding the efficiency of nearest point algorithms and how they can be optimized.

We begin in Section 5.2 by reviewing several Nearest Point Algorithms (NPAs) which have previously been applied to SVMs. We focus first on nearest point algorithms which have been applied to convex hulls in order to compute hard margin SVMs. We then describe how some of these algorithms have been adapted to operate over RCHs in order to compute L_1-loss soft margin SVMs. Subsequently, Section 5.3, reviews Sequential Minimal Optimization (SMO), an SVM training algorithm which is used in many popular software packages due to its efficiency. We compare SMO to the nearest point approach and describe the similarities and differences of the two approaches.

In Section 5.4 we make some general observations on geometric training algorithms and suggest ways in which the current state-of-the-art can be better understood and improved. This leads to Section 5.5, in which we address the task of applying nearest point algorithms to train WSVMs. To the best of our knowledge, nearest point algorithms have not previously been used to train WSVMs. We combine the concept of WRCHs, a variant of RCHs which we introduced in Section 3.9, with an existing nearest point algorithm. The result is a Weighted Schlesinger-Kozinec (WSK) nearest point algorithm capable of

operating over WRCHs. This algorithm can train WSVMs with either an L_1 or L_2 loss function, with precise weights, without inflating the training set size.

We further refine WSK algorithm in Section 5.6, by describing how efficient caching and updating steps can accelerate the algorithm. In particular, we improve convergence rates by pushing the approximations of the positions of the two nearest points towards the surface of the WRCHs. This step is informed by the geometric interpretation of WSVMs, from which it is known that the nearest points must lie on the surfaces of the two WRCHs. This acceleration is conceptually similar to the update step taken in Kowalczyk's [71] convex hull nearest point algorithm.

Section 5.5.2 explores the efficiency of the S-K algorithm in general and compares it to our accelerated WSK implementation. We show that the steps taken in Section 5.6 are in most cases able to accelerate convergence of the algorithm by a significant amount. In turn, these improvements make the algorithm more competitive with SMO. In particular, the WSK algorithm tends to be competitive with SMO when margins are large (and there are a large number of support vectors). When the parameter values cause the margin to become small, SMO will generally complete faster than the WSK algorithm. These results are interesting because other authors have suggested that S-K algorithms operating over RCHs tend to train SVMs faster than SMO [81]. However, by careful consideration of the impact the parameter values have on training time, we suggest that previous differing results were caused by using different parameter values when comparing the two algorithms.

An important feature of nearest point algorithms is that they generally employ different stopping conditions to SMO. In Section 5.8 we perform empirical trials which compare the various stopping conditions used in both nearest point algorithms and SMO. We find that the stopping conditions favored by previous nearest point algorithms operating over RCHs [40, 109, 82] are not ideal since they do not account for the width of the margin when bounding the error in the estimated position of the nearest points. This means they will perform more iterations than required when the margin is large, and fewer iterations than required when the margin is small. We instead favor the relative stopping conditions used previously in convex hull nearest point algorithms [65]. We adapt these stopping conditions to be applicable to the WRCH case, and show that they provide a better trade-off between training iterations performed and test accuracy achieved.

Finally, we describe how nearest point algorithms operating over a single WRCH can also be used to train weighted perceptrons with either an L_1 or an L_2-loss function. This generalizes work by Franc [39], who showed that L_2-loss unweighted perceptrons could be trained using nearest point algorithms operating on a single convex hull. Despite the simplicity of the perceptron nearest point task, empirical trials suggest that training perceptrons over WRCHs is less efficient than training SVMs over WRCHs because of the more complex feature space required by perceptrons.

5.2 Nearest Point Algorithms

Nearest point algorithms compute the nearest points in two disjoint convex hulls (Figure 5.1). As described in Section 2.5, a hard margin SVM is equivalent to the perpendicular bisector of the line between the nearest points in the convex hulls of the two training classes. Because of this relationship, nearest point algorithms can be used to train SVMs. However, despite being capable of training SVMs, nearest point algorithms were not a consequence of SVMs and learning theory. Rather, the first applications of nearest point algorithms to learning appeared several decades before the introduction of SVMs [72, 100]. It is likely that the ability for nearest point algorithms to generalize well helped influence the development of SVMs and statistical learning theory.

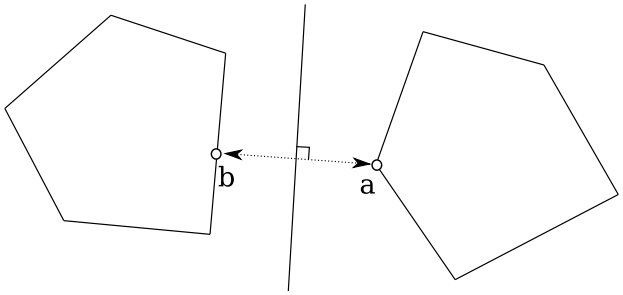


Figure 5.1: A nearest point algorithm minimizes the distance between two points in the convex hulls of each classes (**a** and **b**). The perpendicular bisector of the line joining **a** and **b** is the decision surface of an SVM.

In this section we describe several of the most commonly used nearest point algorithms. We describe these algorithms in terms of an iterative *update step*. This update step is designed to move two approximations of the nearest points closer together, and continues until a set of *stopping conditions* is reached. Generally, the stopping conditions flag termination of the algorithm once the approximate nearest point in each class is within some tolerance ϵ of the true nearest point in that class. Because all of the algorithms described here can share common stopping conditions, we introduce the stopping conditions last, in Section 5.2.4.

The algorithms we describe in this section were some of the first nearest point algorithms to be introduced. Because of this, they share many similarities with later algorithms that have been introduced. For example, Keerthi et al. [65] and Kowalczyk [71] have both described accelerated variants and/or hybrids of the following algorithms. We will explore these techniques for improving nearest point algorithms in more detail in Sections 5.4-5.6, and in doing so apply similar improvements to the case of nearest point algorithms operating over WRCHs.

5.2.1 The Schlesinger-Kozinec Algorithm

One of the simplest and most widely used nearest point algorithms is the Schlesinger-Kozinec (S-K) Algorithm. The S-K algorithm consists of a rule by Kozinec [72] for updating two approximate nearest points, combined with relaxed stopping conditions by

Schlesinger et al. [100]. The S-K algorithm has been used to train hard margin and L_2-loss SVMs by Franc and Hlavàč [40]. Subsequently, Tao et al. [109] showed that the S-K algorithm could also be used to train L_1-loss SVMs by iterating over RCHs instead of convex hulls.

The rationale of the S-K algorithm is to take two approximate nearest points, one from each of the two convex hulls, and update them iteratively, bringing them closer together with each iteration (Figure 5.2). Two initial approximations of the nearest points $\mathbf{p}_{pos}, \mathbf{p}_{neg}$ can be chosen as arbitrary points in the convex hulls of the two classes. One of the easiest ways to achieve this is to simply initialize $\mathbf{p}_{pos} = \mathbf{x}_j$, for any j such that $y_j = 1$, and initialize $\mathbf{p}_{neg} = \mathbf{x}_k$, for any k such that $y_k = -1$.

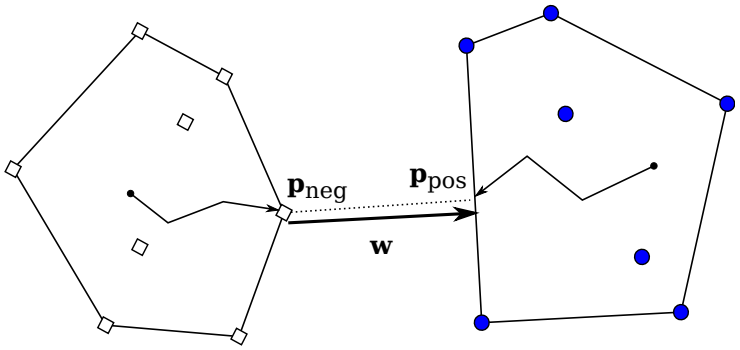


Figure 5.2: Finding the nearest points using an iterative update step

The approximate nearest points, $\mathbf{p}_{pos}$ and $\mathbf{p}_{neg}$ provide a direction (Figure 5.2):

$$\mathbf{w} = \mathbf{p}_{pos} - \mathbf{p}_{neg}$$

The vector $\mathbf{w}$ can be considered the normal of a hyperplane which approximately separates the two convex hulls. As the two approximate nearest points become more and more accurate, the hyperplane will eventually become perpendicular to the shortest line between the two classes, and hence will define an SVM.

The vector $\mathbf{w}$ is used to update the estimated locations of the nearest points as follows. In order to update the positive class, a vertex $\mathbf{v}_{pos}$ from the positive class is found which is extreme in the direction $-\mathbf{w}$, i.e. for which:

$$\mathbf{v}_{pos} = \max_{i \in I_{pos}} \{-\mathbf{w} \cdot \mathbf{x}_i\}.$$

Given such a vertex, $\mathbf{p}_{pos}$ can be updated to move it closer to $\mathbf{p}_{neg}$ by setting [72]:

$$\mathbf{p}_{pos}^{new} = (1 - \lambda)\mathbf{p}_{pos} + \lambda \mathbf{v}_{pos}. \tag{5.1}$$

Here λ is chosen to minimize the distance between the approximate nearest points. This distance is at a minimum when the line segment joining $\mathbf{p}_{pos}^{new}$ and $\mathbf{p}_{neg}$ forms a right angle with the line segment joining $\mathbf{p}_{pos}$ and $\mathbf{v}_{pos}$ (Figure 5.3). To ensure the updated approximation of the nearest point does not fall outside of the hull, λ is clamped between

$[0, 1]$. This means the optimal value of λ is given by:

$$\begin{aligned} 0 &= (\mathbf{p}_{neg} - (1-\lambda)\mathbf{p}_{pos} - \lambda\mathbf{v}_{pos}) \cdot (\mathbf{p}_{pos} - \mathbf{v}_{pos}) \\ &= (\mathbf{p}_{neg} - \mathbf{p}_{pos})(\mathbf{p}_{pos} - \mathbf{v}_{pos}) + \lambda(\mathbf{p}_{pos} - \mathbf{v}_{pos})^2 \\ \Rightarrow \lambda &= \text{clamp}\left(\frac{(\mathbf{p}_{neg} - \mathbf{p}_{pos}) \cdot (\mathbf{p}_{pos} - \mathbf{v}_{pos})}{(\mathbf{p}_{pos} - \mathbf{v}_{pos})^2}, 0, 1\right) \end{aligned} \tag{5.2}$$

In Equation (5.2) we use the clamp function, defined as:

$$\text{clamp}(c, c_{min}, c_{max}) = \min\left(\max\left(c, c_{min}\right), c_{max}\right).$$

The clamp function forces a variable to fall within a particular range, returning:

$$\text{clamp}(c, c_{min}, c_{max}) = \begin{cases} c_{min} & \text{if } c \leq c_{min} \\ c & \text{if } c_{min} < c < c_{max} \\ c_{max} & \text{if } c \geq c_{max} \end{cases}$$

This use of the clamp function ensures that the two approximations of the nearest points can never be allowed to leave the convex hulls of the two classes.

Once the optimal λ has been found, the new value of $\mathbf{p}_{pos}$ can be computed, and then used to update $\mathbf{w}$. This new value of $\mathbf{w}$ provides a new direction in which an update vertex can be found, allowing the update step to be repeated.

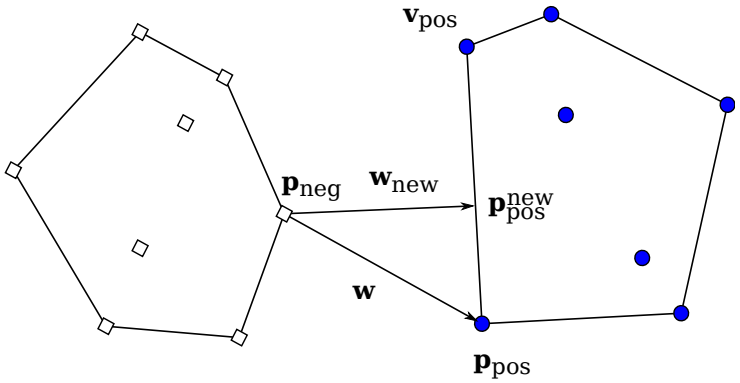


Figure 5.3: Updating the approximate nearest points using the S-K algorithm

The negative class is updated in a similar way to the positive class, except $\mathbf{v}_{neg}$ is instead chosen such that $\mathbf{v}_{neg} = \arg\max_{i \in I_{neg}}\{\mathbf{x}_i \cdot \mathbf{w}\}$. The update rule then becomes:

$$\mathbf{p}^{new}_{neg} = (1-\lambda)\mathbf{p}_{neg} + \lambda\mathbf{v}_{neg}. \tag{5.3}$$

With λ now computed using:

$$\lambda = \text{clamp}\left(\frac{(\mathbf{p}_{pos} - \mathbf{p}_{neg}) \cdot (\mathbf{p}_{neg} - \mathbf{v}_{neg})}{(\mathbf{p}_{neg} - \mathbf{v}_{neg})^2}, 0, 1\right) \tag{5.4}$$

Algorithm 8 The Schlesinger-Kozinec Algorithm

function SK($P, \mathbf{y}$)
  initialize $\mathbf{p}_{pos}$ to any point from the positive class
  initialize $\mathbf{p}_{neg}$ to any point from the negative class
  loop
    $\mathbf{w} \leftarrow \mathbf{p}_{pos} - \mathbf{p}_{neg}$
    $\mathbf{v}_{pos} \leftarrow \arg\max_{i \in I_{pos}} -\mathbf{w} \cdot \mathbf{x}_i$ ▷ find update vertices
    $\mathbf{v}_{neg} \leftarrow \arg\max_{i \in I_{neg}} \mathbf{w} \cdot \mathbf{x}_i$
    if $\mathbf{w}(\mathbf{v}_{pos} - \mathbf{p}_{neg}) < \mathbf{w}(\mathbf{p}_{pos} - \mathbf{v}_{neg})$ **then**
      if $1 - \mathbf{w} \cdot (\mathbf{v}_{pos} - \mathbf{p}_{neg})/||\mathbf{w}||^2 < \epsilon$ **then**
        break ▷ stopping conditions reached
      end if
      $\lambda \leftarrow \text{clamp}\left(\frac{(\mathbf{p}_{neg} - \mathbf{p}_{pos})(\mathbf{p}_{pos} - \mathbf{v}_{pos})}{(\mathbf{p}_{pos} - \mathbf{v}_{pos})^2}, 0, 1\right)$ ▷ positive class update
      $\mathbf{p}_{pos}^{new} \leftarrow (1 - \lambda)\mathbf{p}_{pos} + \lambda \mathbf{v}_{pos}$
    else
      if $1 - \mathbf{w} \cdot (\mathbf{p}_{pos} - \mathbf{v}_{neg})/||\mathbf{w}||^2 < \epsilon$ **then**
        break ▷ stopping conditions reached
      end if
      $\lambda \leftarrow \text{clamp}\left(\frac{(\mathbf{p}_{pos} - \mathbf{p}_{neg})(\mathbf{p}_{neg} - \mathbf{v}_{neg})}{(\mathbf{p}_{neg} - \mathbf{v}_{neg})^2}, 0, 1\right)$ ▷ negative class update
      $\mathbf{p}_{neg}^{new} \leftarrow (1 - \lambda)\mathbf{p}_{neg} + \lambda \mathbf{v}_{neg}$
    end if
  end loop
  $b \leftarrow \frac{1}{2}(\mathbf{w} \cdot \mathbf{p}_{pos} + \mathbf{w} \cdot \mathbf{p}_{neg})$
  return $(\mathbf{w}, b)$ ▷ return hyperplane normal and offset
end function

Using Kernels

The S-K algorithm can be adapted to operate in a kernel feature space by representing the nearest point estimates $\mathbf{p}_{pos}$ and $\mathbf{p}_{neg}$, as well as the update point $\mathbf{v}_{pos}$, as convex combinations of points from P. This yields:

$$\mathbf{p}_{pos} = \sum_{i \in I_{pos}} \alpha_i \mathbf{x}_i \qquad \mathbf{p}_{neg} = \sum_{i \in I_{neg}} \alpha_i \mathbf{x}_i \qquad \mathbf{v}_{pos} = \sum_{i \in I_{pos}} \beta_i \mathbf{x}_i \tag{5.5}$$

In order to preclude the need for explicit feature maps, we can use Equations (5.2) and (5.5) to calculate λ (for the positive class) as:

$$\lambda = \text{clamp}\left(\frac{\sum_{i=1}^{n} \sum_{j \in I_{pos}} y_i \alpha_i (\alpha_j - \beta_j) K(\mathbf{x}_i, \mathbf{x}_j)}{\sum_{i \in I_{pos}} \sum_{j \in I_{pos}} (\alpha_i - \beta_i)(\alpha_j - \beta_j) K(\mathbf{x}_i, \mathbf{x}_j)}, 0, 1 \right). \tag{5.6}$$

A similar equation can be derived using Equation (5.4) for the negative class. Although Equation (5.6) appears quite computationally expensive to compute, Franc and Hlavàč [40] describe how to use caching to accelerate the computation of λ. We further elaborate on the efficient implementation of kernel S-K algorithms in Section 5.6.

Computing the Nearest Points in two RCHs

The S-K algorithm was originally applied to the convex hull nearest point task, meaning it could originally train only hard margin SVMs (or soft margin L_2-loss SVMs, which have a hard margin formulation in a kernel feature space, as we discussed in Section 2.5.4). However, the standard S-K algorithm operating over convex hulls may not be used in order to train L_1-loss SVMs. If the L_1-loss function is desired, the S-K algorithm must be modified to operate over reduced convex hulls instead of convex hulls.

The Schlesinger-Kozinec algorithm has been adapted to train L_1-loss SVMs by operating over RCHs [82, 109]. This is possible because the update step depends only on the current approximations of the nearest points and a single update vertex. It follows that the update step from Equation (5.1) can be rewritten:

$$\mathbf{p}_{pos}^{new} = (1 - \lambda)\mathbf{p}_{pos} + \lambda \mathbf{v}_{pos}, \tag{5.7}$$

where $\mathbf{v}_{pos}$ is a vertex in the RCH of the positive class which is extreme in the direction $-\mathbf{w}$. This vertex can be given by:

$$\mathbf{v}_{pos} = \underset{\mathbf{v} \in \text{RCH}(P,\mu)}{\arg\max} \; -\mathbf{v} \cdot \mathbf{w}.$$

Recall that Theorem 3 (in Section 3.4) provides a simple means for computing such a vertex, making it possible to apply the S-K algorithm to RCHs.

Drawbacks of the S-K Algorithm

The main drawback of the S-K algorithm is best illustrated by considering the kernel S-K implementation. In this implementation, the update step in Equation (5.1) is very unlikely to return an α_i to zero once it has been given a positive value. This means that, although convergence can be fast at first, it is likely to become slow in later iterations [65, 83]. An excess of non-zero α_i values has the effect of getting the approximate nearest points 'stuck' inside the hulls, rather than allowing them to quickly reach the border, where the solution will always lie.

5.2.2 Gilbert's Algorithm

Gilbert's algorithm [44], developed prior to the S-K algorithm, is analogous to a one-set version of the S-K algorithm. Instead of operating on two convex hulls like the S-K algorithm, Gilbert's algorithm finds the point in a single convex hull which is closest to zero (i.e. which has minimal norm) (Figure 5.4). Both Gilbert's algorithm and the S-K algorithm use the same nearest point update procedure. However, in the S-K algorithm the current nearest point in the class being updated is brought closer to the nearest point in the other class, whereas in Gilbert's algorithm it is brought closer to zero.

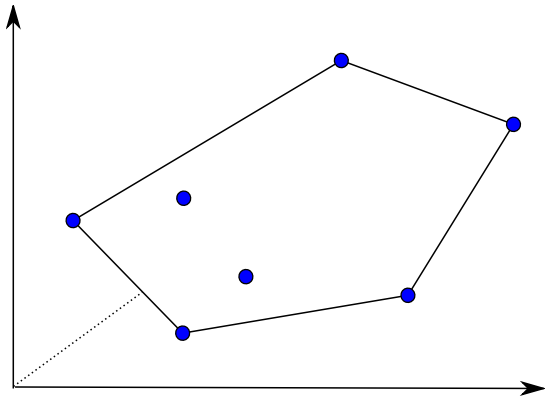


Figure 5.4: The point in a convex hull with minimal norm

Gilbert's algorithm begins by initializing an approximate nearest point $\mathbf{p} \in P$. Any point from P may be used as an initial nearest point. An update point $\mathbf{v} \in P$ is chosen by finding:

$$\mathbf{v} = \min_{i=1}^{n} \{\mathbf{p} \cdot \mathbf{x}_i\}.$$

The update point is used to drag $\mathbf{p}$ closer to the origin using the same update step used by the S-K algorithm. Namely, by letting:

$$\mathbf{p}_{new} = (1 - \lambda)\mathbf{p} + \lambda\mathbf{v}. \tag{5.8}$$

λ is chosen to minimize $||\mathbf{p}||$, which occurs when the vector $\mathbf{p}_{new}$ forms a right angle with the vector $(\mathbf{p} - \mathbf{v})$. This occurs when [44, 65]:

$$\begin{aligned} 0 &= ((1-\lambda)\mathbf{p} + \lambda\mathbf{v}) \cdot (\mathbf{p} - \mathbf{v}) \\ &= \mathbf{p}(\mathbf{p} - \mathbf{v}) - \lambda(\mathbf{p} - \mathbf{v})^2 \\ \Rightarrow \lambda &= \text{clamp}\left(\frac{\mathbf{p} \cdot (\mathbf{p} - \mathbf{v})}{(\mathbf{p} - \mathbf{v})^2}, 0, 1\right) \end{aligned}$$

Algorithm 9 Gilbert's Algorithm

```
function GILBERT(P)
    Initialize p to equal any point from P
    loop
        find point v ∈ P which minimizes p · v
        if 1 − p · v/||p||² < ε then
            break                                  ▷ stopping conditions reached
        else
            λ ← clamp((p·(p−v))/(p−v)², 0, 1)
            p ← (1 − λ)p + λv
        end if
    end loop
end function
return p
```

Using Kernels

Kernels are introduced to Gilbert's algorithm by representing the current nearest point $\mathbf{p}$ and update vertex $\mathbf{v}$ as convex combinations of other points:

$$\mathbf{p} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i \qquad\qquad \mathbf{v} = \sum_{i=1}^{n} \beta_i \mathbf{x}_i.$$

Using this representation, the coefficient λ to use in the update step (Equation 5.8) becomes:

$$\lambda = \text{clamp}\left(\frac{\sum_{i,j=1}^{n} \alpha_i(\alpha_j - \beta_j)K(\mathbf{x}_i, \mathbf{x}_j)}{\sum_{i,j=1}^{n} (\alpha_i - \beta_i)(\alpha_j - \beta_i)K(\mathbf{x}_j, \mathbf{x}_j)}, 0, 1\right).$$

For Two Classes

Although both Gilbert's algorithm and the S-K algorithm may appear to solve separate problems, Gilbert's algorithm can be extended to compute the nearest points in two classes if it is applied to a set which combines the two classes using the Minkowski set difference operator [101, 65]. This has led several authors [65, 83, 80] to apply Gilbert's algorithm to the training of SVMs. However, as Keerthi et al. [65] point out, applying Gilbert's

algorithm to the Minkowski difference of two sets is not necessarily simpler (in terms of computational effort) than operating directly over the original two sets.

Gilbert's one-class nearest point algorithm can be used to solve the two-class nearest point problem by combining the two classes using their Minkowski set difference. The Minkowski set difference M of sets P_{pos} and P_{neg} is defined as [101, 65]:

$$M = P_{pos} \ominus P_{neg} = \{\mathbf{p}_{pos} - \mathbf{p}_{neg} \mid \mathbf{p}_{pos} \in P_{pos},\ \mathbf{p}_{neg} \in P_{neg}\}.$$

Let $\mathbf{m} \in \text{CH}(P_{pos} \ominus P_{neg})$ be a point in the convex hull of the Minkowski set difference $P_{pos} \ominus P_{neg}$. Notice that the point in this set with minimal norm is given by solving:

$$\min ||\mathbf{m}|| = \min || \sum_{i \in I_{pos}} \alpha_i \mathbf{x}_i - \sum_{i \in I_{neg}} \alpha_i \mathbf{x}_i ||.$$

It follows that finding the point in the convex hull of $P_{pos} \ominus P_{neg}$ with minimal norm is equivalent to finding the nearest points in $\text{CH}(P_{pos})$ and $\text{CH}(P_{neg})$.

The additional difficulty with solving the Minkowski form of the nearest point problem is finding the update vertex $\mathbf{v} \in \text{CH}(P_{pos} \ominus P_{neg})$, which no longer exists in the convex hull of a simple set of points. This means the update step must be adapted to find a vertex of $\text{CH}(P_{pos} \ominus P_{neg})$. This may seem difficult given the size of the Minkowski set, however finding an update vertex $\mathbf{v}$ amounts to finding:

$$\begin{aligned} \mathbf{v} &= \underset{\mathbf{m} \in \text{CH}(P_{pos} \ominus P_{neg})}{\arg\min} \mathbf{p} \cdot \mathbf{m} \\ &= \underset{\mathbf{p}_{pos} \in P_{pos}}{\arg\min} \{\mathbf{p} \cdot \mathbf{p}_{pos}\} + \underset{\mathbf{p}_{neg} \in P_{neg}}{\arg\min} \{-\mathbf{p} \cdot \mathbf{p}_{neg}\} \end{aligned}$$

Recall that $\mathbf{p}$ is the current approximation of the point in $\text{CH}(P_{pos} \ominus P_{neg})$ with minimal norm.

Notice how $\mathbf{v}$ combines one vertex from each class. This makes the Minkowski update similar to the S-K update step, except with both classes updated simultaneously rather than sequentially. This is why Keerthi et al. [65] suggest that Gilbert's algorithm applied to a Minkowski set is only *superficially* simpler than the S-K algorithm. The similarities between the two algorithms also mean that Gilbert's algorithm suffers from the same main drawback as the S-K algorithm: slow convergence in later iterations [65, 83]. We further examine the difference in the performance of the one and two-class approaches in later sections.

Computing the Nearest Points in two RCHs

Like the S-K algorithm, Gilbert's algorithm was originally applied to convex hulls, meaning it can only train hard margin or L_2-loss SVMs. However, it can be modified to compute the nearest points in two RCHs in the same way that the S-K algorithm can. Namely, by rewriting the update step as:

$$\mathbf{p}_{new} = (1 - \lambda)\mathbf{p} + \lambda \mathbf{v}.$$

Here $\mathbf{v}$ is a vertex of the RCH which is extreme in direction $-\mathbf{p}$. Recall that this vertex is simple to find using Theorem 3.

5.2.3 The Mitchell-Dem'yanov-Malozemov Algorithm

The Mitchell-Dem'yanov-Malozemov (MDM) algorithm [84, 65, 77] is another nearest point algorithm which uses an iterative update step to move two approximate nearest points gradually as close together as possible. The two approximate nearest points must be represented as a convex combination of input points:

$$\mathbf{p}_{pos} = \sum_{i \in I_{pos}} \alpha_i \mathbf{x}_i \qquad\qquad \mathbf{p}_{neg} = \sum_{i \in I_{neg}} \alpha_i \mathbf{x}_i$$

Although the MDM algorithm can also be applied to the one-class nearest point problem [84] (in the same manner as Gilbert's algorithm), we describe the two-class variant here.

The defining feature of the MDM algorithm is that it updates an approximate nearest point (in the positive class) by setting:

$$\mathbf{p}_{pos}^{new} = \mathbf{p}_{pos} + \lambda \mathbf{z}.$$

In order to keep the convex hull constraints of the optimization task intact, this update step requires that $\mathbf{z}$ be given by:

$$\mathbf{z} = \mathbf{x}_{dst} - \mathbf{x}_{src}.$$

Letting $\mathbf{w} = \mathbf{p}_{pos} - \mathbf{p}_{neg}$, indices src and dst satisfy:

$$src = \underset{i \in I_{pos}, \alpha_i > 0}{\arg\min} \{\mathbf{w} \cdot \mathbf{x}_i\} \qquad\qquad dst = \underset{i \in I_{pos}, \alpha_i < 1}{\arg\max} \{\mathbf{w} \cdot \mathbf{x}_i\}.$$

The size of the update step, λ, for the MDM algorithm is computed by minimizing:

$$\begin{aligned} ||\mathbf{p}_{pos}^{new} - \mathbf{p}_{neg}||^2 &= ||\mathbf{p}_{pos} + \lambda \mathbf{z} - \mathbf{p}_{neg}||^2 \\ &= ||\mathbf{w} + \lambda \mathbf{z}||^2 \\ &= ||\mathbf{w}||^2 + 2\lambda \mathbf{w} \cdot \mathbf{z} + \lambda^2 ||\mathbf{z}||^2 \end{aligned}$$

This quantity is at a minimum when $2\mathbf{w} \cdot \mathbf{z} + 2\lambda ||\mathbf{z}||^2 = 0$, implying that:

$$\begin{aligned} \lambda &= \text{clamp}\left(-\frac{\mathbf{w} \cdot \mathbf{z}}{||\mathbf{z}||^2}, 0, \lambda_{max}\right) \\ \text{where } \lambda_{max} &= \min\left(\alpha_{src}, 1 - \alpha_{dst}\right). \end{aligned} \tag{5.9}$$

Here we clamp the value of λ between $[0, \lambda_{max}]$. This forces $0 \leq \alpha_i \leq 1$ to be satisfied, meaning an update step can not move an approximation of a nearest point outside of the hull.

Figure 5.5 depicts the MDM update step. Note that line segment from $\mathbf{x}_{src}$ to $\mathbf{x}_{dst}$ runs parallel to the line segment from $\mathbf{p}_{pos}$ to $\mathbf{p}_{pos}^{new}$. The planes $P1$ and $P2$ are also parallel, and are perpendicular to the line $\mathbf{p}_{pos} - \mathbf{p}_{neg}$ between the two approximate nearest points

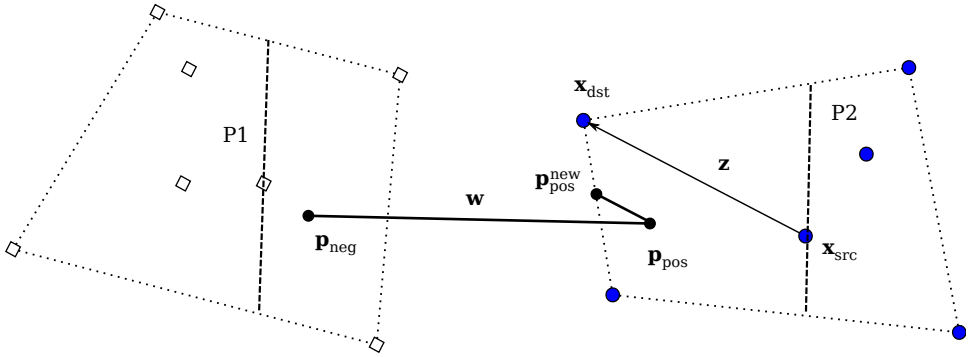


Figure 5.5: The MDM update step

Keerthi et al. [65] describe the MDM update step as not only moving the estimated nearest points closer together, but simultaneously moving the 'slab' formed by planes $P1$ and $P2$ (depicted in Figure 5.5) closer together. By contrast, the S-K algorithm focuses solely on moving the estimated nearest points closer together. Because of this difference, the MDM algorithm generally converges much faster than Gilbert's algorithm in later iterations because it provides a mechanism by which non-zero α_i values can be reduced to zero. By contrast, α_i values in the S-K algorithm are very unlikely to be quickly reduced to zero once they are given a positive weight.

Computing the Nearest Points in two RCHs

The MDM algorithm, unlike the S-K and Gilbert's algorithm, does not depend solely on the vertices of the convex hull in order to decide how to make progress towards the nearest points. Because of this, applying it to RCHs requires more than simply substituting a new vertex finding rule into the algorithm.

There have been two recent approaches to applying MDM to RCHs. One of these approaches, taken by López et al. [78], is to make two modifications to the algorithm. First, the way in which update points $\mathbf{x}_{src}$ and $\mathbf{x}_{dst}$ are modified to take into account the RCH parameter μ. This results in update indices (for the positive class) of:

$$src = \underset{i \in I_{pos}, \alpha_i > 0}{\arg\min} \{\mathbf{w} \cdot \mathbf{x}_i\} \qquad dst = \underset{i \in I_{pos}, \alpha_i < \mu}{\arg\max} \{\mathbf{w} \cdot \mathbf{x}_i\}.$$

This means that the algorithm will only attempt to shift weight to a point if its Lagrange multiplier is not at its upper bound. The second modification to the algorithm is to cap all α_i's at the maximum value of μ in order to satisfy the RCH constraints.

The second approach to applying MDM to RCHs, taken by Tao et al. [109], is to modify the MDM update step so that it becomes:

$$\mathbf{p}_{pos}^{new} = (1 - \lambda)\mathbf{p}_{pos} + \lambda \mathbf{z}_{pos}, \tag{5.10}$$

Rather than $\mathbf{z}_{pos}$ being a vertex of the RCH, it is instead given by:

$$\mathbf{z}_{pos} = \mathbf{v}_{pos} + \gamma(\mathbf{x}_{dst} - \mathbf{x}_{src}),$$

where $\mathbf{v}_{pos}$ is the standard S-K update vertex described in Section 5.2.1, $\gamma = \min(\alpha_{dst}, \alpha_{src})$, and:

$$src = \underset{i \in I_{pos}, \alpha_i > 0}{\arg\max} \ \mathbf{w} \cdot \mathbf{x}_i$$
$$dst = \underset{i \in I_{pos}, \alpha_i < \mu}{\arg\min} \ \mathbf{w} \cdot \mathbf{x}_i$$

Notice that Equation (5.10) bears more resemblance to the S-K algorithm than MDM, so Tao et al.'s algorithm could also be described as an S-K/MDM hybrid or a modified S-K algorithm.

5.2.4 Terminating the Algorithm

Many nearest point algorithms can not simply be repeated until an exact solution is reached, since this is not guaranteed to occur in a finite number of iterations [100, 40, 65]. Instead updates are stopped stopped once the solution becomes correct to within some tolerance parameter ϵ. We suggest terminating once each approximate nearest point lies with a relative factor of ϵ of the true nearest points. This occurs when the current solution satisfies:

$$1 - \frac{\mathbf{w} \cdot (\mathbf{v}_{pos} - \mathbf{p}_{neg})}{||\mathbf{w}||^2} < \epsilon,$$
$$\text{and} \qquad 1 - \frac{\mathbf{w} \cdot (\mathbf{p}_{pos} - \mathbf{v}_{neg})}{||\mathbf{w}||^2} < \epsilon$$

This can also be written as:

$$\min \left(\frac{\mathbf{w}(\mathbf{v}_{pos} - \mathbf{p}_{neg})}{||\mathbf{w}||^2}, \ \frac{\mathbf{w}(\mathbf{p}_{pos} - \mathbf{v}_{neg})}{||\mathbf{w}||^2} \right) \geq 1 - \epsilon \tag{5.11}$$

Recall that $\mathbf{p}_{pos}$ and $\mathbf{p}_{neg}$ are the approximate nearest points in the positive and negative classes, respectively. The hyperplane normal is given by $\mathbf{w} = \mathbf{p}_{pos} - \mathbf{p}_{neg}$, and $\mathbf{v}_{pos}$ and $\mathbf{v}_{neg}$ satisfy:

$$\mathbf{v}_{pos} = \underset{\mathbf{x}_i, i \in I_{pos}}{\arg\max} -\mathbf{w} \cdot \mathbf{x}_i \qquad \mathbf{v}_{neg} = \underset{\mathbf{x}_i, i \in I_{neg}}{\arg\max} \mathbf{w} \cdot \mathbf{x}_i$$

Notice that these values of $\mathbf{v}_{pos}$ and $\mathbf{v}_{neg}$ are specific to the convex hull case, where vertices are guaranteed to be points from P. If the nearest point algorithm is instead being applied to an RCH, these vertices must become:

$$\mathbf{v}_{pos} = \underset{\mathbf{v} \in \text{RCH}(P_{pos}, \mu)}{\arg\max} -\mathbf{w} \cdot \mathbf{v} \qquad \mathbf{v}_{neg} = \underset{\mathbf{v} \in \text{RCH}(P_{neg}, \mu)}{\arg\max} \mathbf{w} \cdot \mathbf{v}$$

These maximization tasks find vertices in the RCH which are extreme in directions $\mathbf{w}, -\mathbf{w}$, so they can be easily solved using Theorem 3.

A relative nearest point stopping condition similar to this one is used by Keerthi et al. [65]. However, other authors have also used an absolute stopping condition where the algorithm terminates once the *absolute* distance between the approximate nearest point and the true nearest point falls below a tolerance [40, 81, 109]. We will more closely examine the impacts of varying the stopping conditions in Section 5.8. However, for consistency in this section, all algorithm listings will refer to the relative stopping conditions in (5.11).

5.2.5 Choosing the Threshold

Once these stopping conditions are met, the threshold of the hyperplane, b, is set so that it places the hyperplane half way between the two nearest points:

$$b = \frac{1}{2}\left(\mathbf{w} \cdot \mathbf{p}_{pos} + \mathbf{w} \cdot \mathbf{p}_{neg}\right).$$

This is the geometric threshold, which we previously described in detail in Chapter 4. Recall that this threshold is not identical to the one given by the KKT conditions. However, it is common to use it in conjunction with nearest point algorithms [40, 109].

The benefit of using the geometric threshold in conjunction with the S-K algorithm is that this threshold is not susceptible to change significantly due to extremely small (but non-zero) α_i's. Recall that these are quite likely to occur, particularly in conjunction with the S-K algorithm, since it is difficult for this algorithm to return an α_i value to zero once it has been given a non-zero value. Using a geometric threshold prevents these α_i's from having an undue influence on the placement of the hyperplane by skewing the threshold.

5.3 Sequential Minimal Optimization

Sequential Minimal Optimization (SMO) is an SVM training algorithm introduced by Platt [91], who noted that SVMs can be efficiently trained by solving a sequence of small analytical update steps involving just two Lagrange multipliers at a time. The update step is a two-part process; first a heuristic is used to select the Lagrange multipliers to update, then the dual objective function is optimized with respect to those two multipliers, while keeping the dual constraints intact.

SMO has been applied to many different types of SVMs. Refer, for example, to Platt [91] for the original C-SVM formulation, to Vogt [119] for a perceptron variant, and to Keerthi et al. [67] for a Kernel Logistic Regression (KLR) variant. In this section we focus specifically on SMO for the L_1-loss μ-SVM for its consistency with the geometric framework, and hence with the other algorithms we discuss in this chapter.

5.3.1 Updating a Pair of Lagrange Multipliers

The SMO update step consists of updating two Lagrange multipliers $\alpha_{dst}, \alpha_{src}$ such that:

$$\alpha_{dst} \leftarrow \alpha_{dst} + \delta \qquad\qquad \alpha_{src} \leftarrow \alpha_{src} - \delta. \tag{5.12}$$

In order to keep the constraints on the dual satisfied, we must ensure that both update points are from the same class, i.e. $y_{dst} = y_{src}$. The value of δ to use is derived by considering the L_1-loss μ-SVM dual objective function with respect to the updated Lagrange multipliers. Recall that the μ-SVM objective function is given by:

$$F = -\frac{1}{4}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j$$

Updating α_{dst} and α_{src} using (5.12), we have:

$$F = -\frac{1}{4}||\sum_i \alpha_i y_i \mathbf{x}_i + \delta y_{dst}\mathbf{x}_{dst} - \delta y_{src}\mathbf{x}_{src}||^2 \tag{5.13}$$

Using $y_{dst} = y_{src}$ and expanding this yields:

$$\begin{aligned} F = &-\frac{1}{4}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j - \frac{1}{2}\sum_i \alpha_i y_i y_{dst} \delta \mathbf{x}_i \cdot \mathbf{x}_{dst} + \frac{1}{2}\sum_i \alpha_i y_i y_{dst} \delta \mathbf{x}_i \cdot \mathbf{x}_{src} \\ &- \frac{1}{4}\delta^2 \mathbf{x}_{dst} \cdot \mathbf{x}_{dst} + \frac{1}{2}\delta^2 \mathbf{x}_{dst} \cdot \mathbf{x}_{src} - \frac{1}{4}\delta^2 \mathbf{x}_{src} \cdot \mathbf{x}_{src} \end{aligned} \tag{5.14}$$

Equation 5.14 is optimized with respect to δ when:

$$\begin{aligned} \frac{\partial F}{\partial \delta} &= \frac{y_{dst}}{2}\left(\sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x}_{src} - \sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x}_{dst}\right) \\ &\quad - \frac{\delta}{2}\left(\mathbf{x}_{dst} \cdot \mathbf{x}_{dst} - 2\mathbf{x}_{dst} \cdot \mathbf{x}_{src} + \mathbf{x}_{src} \cdot \mathbf{x}_{src}\right) \\ &= 0 \end{aligned}$$

This results in δ given by:

$$\begin{aligned} \delta &= \text{clamp}\left(\frac{y_{dst}\left(f_{src} - f_{dst}\right)}{\mathbf{x}_{dst} \cdot \mathbf{x}_{dst} - 2\mathbf{x}_{dst} \cdot \mathbf{x}_{src} + \mathbf{x}_{src} \cdot \mathbf{x}_{src}}, 0, \delta_{max}\right) \\ \text{where } \delta_{max} &= \min\left(\alpha_{src}, 1 - \alpha_{dst}\right) \end{aligned} \tag{5.15}$$

Notice that δ has been clamped between $[0, \delta_{max}]$ to avoid any individual α_i becoming negative or greater than one.

Values in (5.15) are used in conjunction with Equation (5.12) in order to complete the SMO update step. In this equation we have used $f_k = \sum_i y_i \alpha_i \mathbf{x}_i \cdot \mathbf{x}_k$ to refer to the dot product of the decision function normal with the training vector $\mathbf{x}_k$, with threshold omitted. We will continue to use this shortened notation in subsequent sections.

5.3.2 Choosing the Lagrange Multipliers to Update

The Lagrange multipliers to update in each iteration of SMO are chosen using a heuristic. Platt's original heuristic for choosing the points to update was to take the *maximal violating pair*, that is the pair of points in the same class which maximize $||f_{src} - f_{dst}||$. Because this expression appears in Equation 5.15, maximizing it generally results in a large step size δ being taken. Since finding the maximal violating pair requires computing f_i's for all training points, some authors combine SMO with smaller working sets where points are incrementally added to and discarded from the working set as training progresses [91, 66].

The heuristic used by Platt is rather simple. This is intentional since the SMO update step is very fast to compute. It therefore does not make sense to spend an excessive amount of time searching for the best Lagrange multipliers to optimize when several approximate guesses could be made and optimized in the same amount of time. However, there have been more recent contributions which suggest that the heuristic for choosing the points to optimize (and by extension, the points to include/exclude from the working set) can be further improved using second order information [37, 22, 45, 14].

5.3.3 Relationship with Nearest Point Algorithms

López et al. [77] point out that the update step performed by SMO on the μ-SVM problem is identical to that performed by the MDM algorithm. This can be seen by expanding the MDM update coefficient in Equation (5.9):

$$\begin{aligned}
\lambda &= \text{clamp}\left(-\frac{\mathbf{w}\cdot\mathbf{z}}{||\mathbf{z}||^2}, 0, \lambda_{max}\right) \\
&= \text{clamp}\left(-\frac{f_{src} - f_{dst}}{\mathbf{x}_{src}^2 - 2\mathbf{x}_{src}\cdot\mathbf{x}_{dst} + \mathbf{x}_{dst}^2}, 0, \lambda_{max}\right) \\
\text{where } \lambda_{max} &= \min(\alpha_{src}, 1 - \alpha_{dst})
\end{aligned}$$

This is equivalent to the SMO update step in Equation (5.15).

In fact, MDM can be thought of as an SMO implementation with a maximal violating pair heuristic over the entire training set (i.e. with no working set selection). This means that the only difference between MDM and most SMO implementations for the μ-SVM is in the way in which the points to optimize are chosen. Once these points have been chosen, the update step itself is identical across the two approaches.

5.4 Improving Nearest Point Algorithms

We suggest that nearest point algorithms can be generalized by applying them to operate over WRCHs instead of convex or reduced convex hulls. This is the most general type of nearest point implementation, since it can be used to train both L_1-loss (RCH) SVMs and L_2-loss (convex hull) SVMs, with optional point weights. This can be seen in the WRCH SVM dual in Equation (5.16). Notice how this optimization task can be combined with

the various configurations in Table 5.1 in order to train each different type of SVM.

$$\begin{aligned} \max_{\alpha_i,\ldots,\alpha_n} \quad & -\frac{1}{4}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i y_i = 0 \\ \sum_{i=1}^{n} \alpha_i = 2 \\ 0 \leq \alpha_i \leq s_i \mu \end{cases} \end{aligned} \tag{5.16}$$

Table 5.1: Various configurations of the general WRCH SVM

SVM Type	**Parameters**	**Kernel** $k(\mathbf{x}_i, \mathbf{x}_j)$
Hard Margin	$\mu' = 1, s_i = 1$	$k(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j)$
L_1-loss (unweighted)	$s_i = 1, 0 \leq \mu \leq 1$	$k(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j)$
L_1-loss (weighted)	$0 \leq \mu \leq 1$	$k(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j)$
L_2-loss (unweighted)	$\mu = 1$	$k(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij}/(2C)$
L_2-loss (weighted)	$\mu = 1$	$k(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij}/(2s_i C)$

There is also a question of how nearest point algorithms operating over WRCHs can best be optimized. Previously, highly optimized nearest point implementations were generally specific to the L_2-loss SVM formulation [71, 65]. In Section 5.6 we look at how nearest point implementations can be optimized specifically for the general WRCH form of the problem.

There is further room for improvement of nearest point algorithms in terms of understanding the stopping conditions and their impact on the final SVM. We mentioned when introducing nearest point algorithms in previous sections how there are several different types of stopping conditions which have been applied in the past. However, to the best of our knowledge there has been limited comparison of these stopping conditions. In Section 5.8 we go into further detail on these stopping conditions and investigate the impact they have on the final solution.

It is important to carefully consider the impact of the stopping conditions in any SVM training algorithm. Because the stopping conditions decide when an algorithm should terminate, they can provide an illusion that one particular training algorithm is faster than another, when the reality is that the algorithm is simply stopping with a more approximate solution.

5.5 A Weighted Schlesinger-Kozinec (WSK) Algorithm

Although the effect of point weighting for SVMs can be simulated by duplicating training data (refer to Section 3.9.2), such an approach introduces problems because it limits the precision with which weights can be specified, as well as increasing the size of the training set. Increasing training set sizes for SVMs is particularly undesirable because training algorithms tend to scale worse than linearly as the amount of data increases [19].

In this section we approach the problem of applying nearest point algorithms to train weighted SVMs by using the concept of *Weighted* Reduced Convex Hulls (WRCHs).

Weighted reduced convex hulls were generalized from reduced convex hulls and their theoretical properties were described in Section 3.9. In Section 4.4.1, we further described how weighted SVMs can be interpreted as the perpendicular bisector of the shortest line between the WRCHs of the two classes. Using this information, we can now construct a geometric algorithm for training weighted SVMs. The algorithm is a generalization of the Schlesinger-Kozinec algorithm, which has previously been applied to both standard and reduced convex hulls [65, 40, 108]. We refer to the new algorithm as the Weighted S-K (WSK) algorithm. The WSK algorithm is capable of naturally handling weighted training data without the need for inflating the training set size.

5.5.1 The WSK Algorithm

The WSK algorithm is a variant of the S-K algorithm which has been modified to operate over WRCHs. This means it begins by taking two approximate nearest points from the WRCHs of each class, and iteratively updates them, bringing them closer together. Initial points $\mathbf{p}_{pos}$ and $\mathbf{p}_{neg}$ can be chosen as arbitrary points in the WRCHs of the two classes. Such points can be found using Algorithm 7 (introduced in Section 3.4) in conjunction with any direction.

The normal of a hyperplane separating two approximate nearest points, $\mathbf{p}_{pos}$ and $\mathbf{p}_{neg}$, can be given by:

$$\mathbf{w} = \mathbf{p}_{pos} - \mathbf{p}_{neg}$$

Because $\mathbf{p}_{pos}$ and $\mathbf{p}_{neg}$ exist in two WRCHs, this hyperplane approximately separates the WRCHs of the two classes, rather than the convex or reduced convex hulls.

In order to update the approximation of the nearest point in the positive class, the vector $\mathbf{w}$ is used to find a vertex, $\mathbf{v}_{pos}$ of the WRCH of the positive class. This is achieved by solving:

$$\mathbf{v}_{pos} = \underset{\mathbf{v} \in \mathrm{WRCH}(P_{pos}, \mathbf{s}, \mu)}{\arg\max} -\mathbf{w} \cdot \mathbf{v}$$

Recall that P_{pos} denotes the set of all points in P which belong to the positive class, and P_{neg} denotes the set of all points from P which belong to the negative class. We described previously in Section 3.4 how this optimization task can be solved using Algorithm 7.

Once the vertex, $\mathbf{v}_{pos}$, has been found, the approximate nearest points can be moved closer together. This is done by setting:

$$\mathbf{p}_{pos}^{new} = (1 - \lambda)\mathbf{p}_{pos} + \lambda \mathbf{v}_{pos}. \tag{5.17}$$

Here λ is chosen to minimize the distance between the two approximations of the nearest points. The calculation of λ does not need to change from the original λ calculation in

the S-K algorithm applied to standard convex hulls, given by [72, 40]:

$$\begin{aligned} 0 &= (\mathbf{p}_{neg} - (1-\lambda)\mathbf{p}_{pos} - \lambda\mathbf{v}_{pos}) \cdot (\mathbf{p}_{pos} - \mathbf{v}_{pos}) \\ \Rightarrow \lambda &= \text{clamp}\left(\frac{(\mathbf{p}_{neg} - \mathbf{p}_{pos}) \cdot (\mathbf{p}_{pos} - \mathbf{v}_{pos})}{(\mathbf{p}_{pos} - \mathbf{v}_{pos})^2}, 0, 1\right) \end{aligned} \tag{5.18}$$

The vector $\mathbf{w}$ is then updated using the new value of $\mathbf{p}_{pos}$ before the nearest points are updated again.

In order to update the negative class using a vertex $\mathbf{v}_{neg}$ from the WRCH of the negative class, the update step becomes [72, 40]:

$$\mathbf{p}_{neg}^{new} = (1-\lambda)\mathbf{p}_{neg} + \lambda\mathbf{v}_{neg}. \tag{5.19}$$

With λ now computed as [72, 40]:

$$\lambda = \text{clamp}\left(\frac{(\mathbf{p}_{pos} - \mathbf{p}_{neg}) \cdot (\mathbf{p}_{neg} - \mathbf{v}_{neg})}{(\mathbf{p}_{neg} - \mathbf{v}_{neg})^2}, 0, 1\right) \tag{5.20}$$

This equation itself has not changed from the original S-K algorithm, however the update vertex $\mathbf{v}_{neg}$ and the way in which it is computed *has* been changed so that it is now a vertex from a WRCH.

The WSK algorithm can retain the relative ϵ-optimal nearest point stopping conditions described previously in Section 5.2.4. These are written as:

$$1 - \frac{\mathbf{w} \cdot (\mathbf{v}_{pos} - \mathbf{p}_{neg})}{||\mathbf{w}||^2} < \epsilon, \qquad \text{and} \qquad 1 - \frac{\mathbf{w} \cdot (\mathbf{p}_{pos} - \mathbf{v}_{neg})}{||\mathbf{w}||^2} < \epsilon$$

The complete WSK algorithm is described in Algorithm 10. This algorithm uses the vertex finding algorithm which we previously described in Algorithm 7 in order to find vertices of the WRCHs of the two classes. The fundamental difference between the WSK algorithm and the original S-K algorithm is that the WSK algorithm takes advantage of the properties of WRCHs we described in previous sections. This allows it to operate over WRCHs in order to train WSVMs, rather than operating over convex hulls and training standard SVMs like the original S-K algorithm.

5.5.2 Comparing WSK to Point Duplication

To test the effectiveness of our algorithm we use two datasets from the UCI machine learning repository which have associated misclassification costs, the `german` and `heart` datasets. We adopt the recommended misclassification costs of 1 for the negative class, and 5 for the positive class. This means that it is 5 times as costly to misclassify points from the positive class than it is to misclassify points from the negative class.

Initial empirical trials use a Gaussian kernel with $\gamma = 0.01$, i.e. basis functions with a very large width which result in an almost-linear decision surface. This ensures that overfitting is unlikely to occur during initial trials. We explore alternative kernels in subsequent trials.

Algorithm 10 The Weighted Schlesinger-Kozinec Algorithm

function WSK($P, \mathbf{s}, \mathbf{y}$)
  initialize $I_{pos} = \{i \mid y_i = 1\}$, $I_{neg} = \{i \mid y_i = -1\}$.
  initialize $P_{pos} = \{\mathbf{x}_i \in P \mid i \in P_{pos}\}$, $P_{neg} = \{\mathbf{x}_i \in P \mid i \in P_{neg}\}$
  initialize $\mathbf{p}_{pos}$ to any point from WRCH($P_{pos}, \mathbf{s}, \mu$)
  initialize $\mathbf{p}_{neg}$ to any point from WRCH($P_{neg}, \mathbf{s}, \mu$)
  loop
    $\mathbf{w} \leftarrow \mathbf{p}_{pos} - \mathbf{p}_{neg}$
    $\mathbf{v}_{pos} \leftarrow \underset{\mathbf{v} \in \mathrm{WRCH}(P_{pos}, \mathbf{s}, \mu)}{\arg\max} -\mathbf{w} \cdot \mathbf{v}$ ▷ use Algorithm 7 to solve this
    $\mathbf{v}_{neg} \leftarrow \underset{\mathbf{v} \in \mathrm{WRCH}(P_{neg}, \mathbf{s}, \mu)}{\arg\max} \mathbf{w} \cdot \mathbf{v}$
    if $\mathbf{w} \cdot (\mathbf{v}_{pos} - \mathbf{p}_{neg}) < \mathbf{w} \cdot (\mathbf{p}_{pos} - \mathbf{v}_{neg})$ **then**
      if $1 - \mathbf{w} \cdot (\mathbf{v}_{pos} - \mathbf{p}_{neg}) / ||\mathbf{w}||^2 < \epsilon$ **then**
        break ▷ stopping conditions reached
      end if
      $\lambda \leftarrow \mathrm{clamp}\left(\frac{(\mathbf{p}_{neg} - \mathbf{p}_{pos}) \cdot (\mathbf{p}_{pos} - \mathbf{v}_{pos})}{(\mathbf{p}_{pos} - \mathbf{v}_{pos})^2}, 0, 1\right)$
      $\mathbf{p}_{pos}^{new} \leftarrow (1 - \lambda)\mathbf{p}_{pos} + \lambda \mathbf{v}_{pos}$
    else
      if $1 - \mathbf{w} \cdot (\mathbf{p}_{pos} - \mathbf{v}_{neg}) / ||\mathbf{w}||^2 < \epsilon$ **then**
        break ▷ stopping conditions reached
      end if
      $\lambda \leftarrow \mathrm{clamp}\left(\frac{(\mathbf{p}_{pos} - \mathbf{p}_{neg}) \cdot (\mathbf{p}_{neg} - \mathbf{v}_{neg})}{(\mathbf{p}_{neg} - \mathbf{v}_{neg})^2}, 0, 1\right)$
      $\mathbf{p}_{neg}^{new} \leftarrow (1 - \lambda)\mathbf{p}_{neg} + \lambda \mathbf{v}_{neg}$
    end if
  end loop
  $b \leftarrow \frac{1}{2}(\mathbf{w} \cdot \mathbf{p}_{pos} + \mathbf{w} \cdot \mathbf{p}_{neg})$
  return $(\mathbf{w}, b)$ ▷ return hyperplane normal and offset
end function

For the regularization parameter, we use $\mu = 1/(0.9\kappa)$. Recall from Section 4.4.5 that κ is the sum of weights in the smaller (by weight) class. Defining μ in terms of κ like this ensures that the WRCHs remain separable regardless of the weighting scheme that is applied. Mean error rates (and associated standard errors) are calculated over 100 runs, with 90% of data used for training and 10% used for testing. The error cost is the sum of weights for misclassified points.

Table 5.2: Results of the three algorithms on the `heart` dataset

	Time (ms)	Total Err Cost	Pos Class Err	Neg Class Err
WSK	33.7 ± 1.0	$\mathbf{11.2 \pm 0.5}$	$\mathbf{0.112 \pm 0.009}$	0.309 ± 0.013
S-KDUP	124.3 ± 3.8	$\mathbf{11.2 \pm 0.5}$	$\mathbf{0.112 \pm 0.009}$	0.309 ± 0.013
S-K	$\mathbf{6.4 \pm 0.1}$	14.8 ± 0.7	0.223 ± 0.011	$\mathbf{0.118 \pm 0.008}$

Table 5.3: Results of the three algorithms on the `german` dataset

	Time (ms)	Total Err Cost	Pos Class Err	Neg Class Err
WSK	$\mathbf{150.7 \pm 3.0}$	$\mathbf{57.7 \pm 1.2}$	$\mathbf{0.202 \pm 0.008}$	0.390 ± 0.006
S-KDUP	1286.3 ± 32.1	$\mathbf{57.7 \pm 1.2}$	$\mathbf{0.202 \pm 0.008}$	0.390 ± 0.006
S-K	528.4 ± 12.6	73.1 ± 1.5	0.420 ± 0.009	$\mathbf{0.148 \pm 0.0004}$

Tables 5.2 and 5.3 show the results of the WSK algorithm. We compare the algorithm to two alternatives: the standard (unweighted) S-K algorithm where points have been duplicated to simulate weights (denoted S-KDUP), and the standard S-K algorithm with no weighting and no point duplication (denoted S-K). The comparison to point duplication shows that the WSK algorithm provides the same solution as given by point duplication, whereas it does so much faster due to the smaller training set size. The comparison to a standard S-K algorithm with no point duplication shows that the weighted variant allows a trade-off where the accuracy in one class is sacrificed in order to increase the accuracy in the other class.

An interesting observation is that the time required to perform the WSK and S-K algorithms is not the same. This means that training times can be altered significantly by introducing different weighting schemes. This difference in training time arises from changes in the shape of the hulls that are being optimized, rather than any inherent inefficiency in the WSK or S-K approach. This is supported by the fact that the WSK algorithm, although relatively slower than the S-K algorithm on the `heart` dataset, was relatively faster on the `german` dataset. We will further explore the relationship between weighting schemes and training time in the following section.

Because the `german` and `heart` datasets have simple misclassification costs which depend solely on the class, they do not exercise the true capability of the WSK algorithm. For this reason we also ran empirical trials which imposed a more complex cost structure on a `synthetic` dataset. Figure 5.6 shows the result of a polynomial SVM ($q = 4$, $\mu = 1/(0.25\kappa)$ trained on a `synthetic` dataset. Each class has points generated from two Gaussian distributions, so there are in effect two subsets in each class. One of these subsets is given the weight of $s_i = 6$, with the other subset given the weight of $s_i = 1$.

In this figure, highly weighted subsets are emphasized using shaded ellipses. If weights were ignored, an optimal decision surface would be a vertical line alone $\mathbf{x} = 0$. Notice, however, that the SVM forces the decision surface to classify correctly the highly weighted subsets, despite the fact that the two classes have identical variance and an equal number of points.

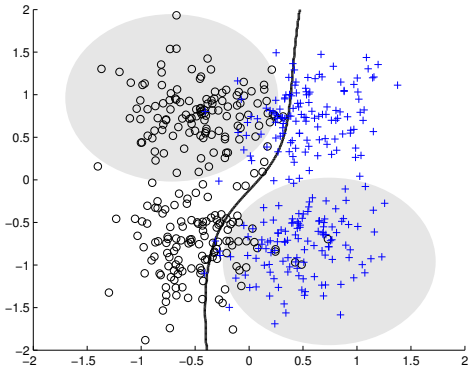


Figure 5.6: A polynomial weighted SVM trained on a `synthetic` dataset with $q = 4$, $\mu = 1/(0.25\kappa)$. The shaded ellipses emphasize the highly weighted subsets of points.

Empirical results on the `synthetic` dataset are shown in Table 5.4. These results are important because they highlight the fact that weights are not only useful for trading accuracy in one class for another, but also for implementing more complex cost structures. For the `synthetic` dataset, the error in *both* classes is worsened by using a weighted classifier. However the total error cost is greatly improved. This means that the points misclassified by the weighted SVM are on average less costly errors than points misclassified by the standard SVM.

Table 5.4: Results of the three algorithms on the `synthetic` dataset

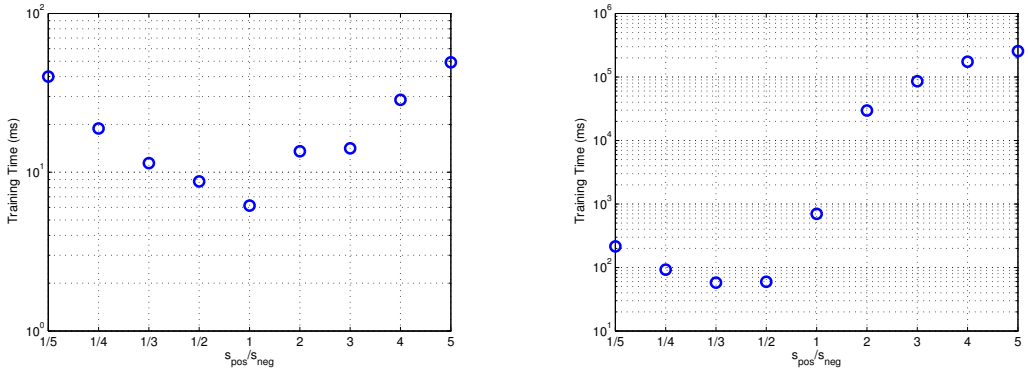
	Time (ms)	Total Err Cost	Pos Class Err	Neg Class Err
WSK	$\mathbf{38.8 \pm 0.7}$	$\mathbf{9.44 \pm 0.37}$	0.171 ± 0.007	0.137 ± 0.007
S-KDUP	394.4 ± 5.1	$\mathbf{9.44 \pm 0.37}$	0.171 ± 0.007	0.137 ± 0.007
S-K	166.2 ± 11.1	16.1 ± 0.84	$\mathbf{0.095 \pm 0.006}$	$\mathbf{0.098 \pm 0.006}$

5.5.3 How Weighting Affects Training Time

Results from the previous section indicated that the weighting of training data could significantly impact training times. However, changing the weights of the points did not necessarily increase training times. Rather, training times could be either increased or decreased depending on the weighting scheme applied and the characteristics of the data. In this section we further investigate how weighting schemes impact training times.

Figure 5.7 shows the time required to train a WSVM using the WSK algorithm, again using $\mu = 1/(0.9\kappa)$, where κ is the sum of weights in the smaller (by weight) class. The x-axis represents the value of s_{pos}/s_{neg}. There are many different sets of weights that

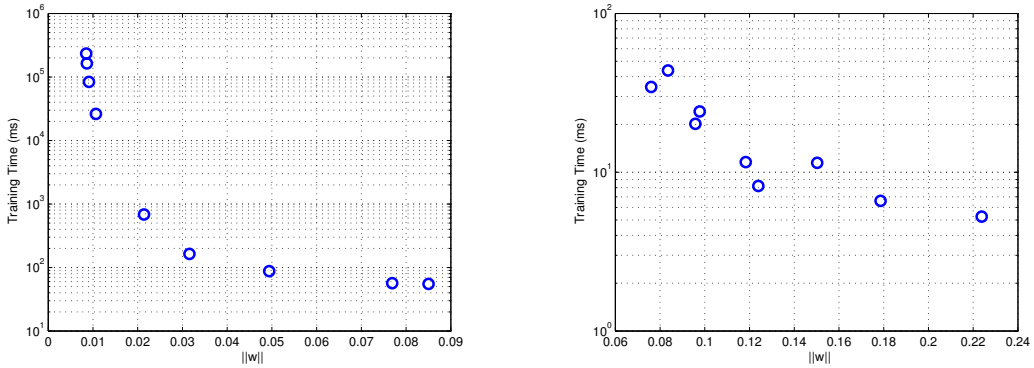
can satisfy this condition. However, because μ is set relative to the total weight, they all have the equivalent effect in this case (provided neither class weight is zero). These results suggest that assigning extreme weights to a class can *possibly* increase training time. However, this is not always the case, as shown in Figure 5.7b where training time is very low when the negative class is given a large weight.



(a) `heart` dataset **(b)** `german` dataset

Figure 5.7: The relationship between weighting and training time. The Gaussian kernel is used with $\gamma = 0.01$

The results in Figure 5.7 are best explained by considering the training times in conjunction with their associated margin, rather than their weights. Figure 5.8 shows the results of training WSVMs with the same weights used in the previous figure, however the x-axis now shows the width of the margin instead of relative class weights. Notice that smaller margins are generally associated with increased training times. This suggests that changing the weighting scheme is only likely to increase training time if the weighting scheme results in a decrease in the width of the margin.



(a) `heart` dataset **(b)** `german` dataset

Figure 5.8: The relationship between margin and training time. The Gaussian kernel is used with $\gamma = 0.01$

5.6 An Efficient WSK Implementation

In Section 5.2.1 we reviewed the existing Schlesinger-Kozinec nearest point algorithm for training SVMs, and in Section 5.5 we introduced the WSK algorithm, an S-K variant which operates over WRCHs in order to train WSVMs. In this section we discuss how the WSK algorithm can be implemented most efficiently. We find that the fastest way to perform the update step depends on the parameters used. We suggest several alternate ways of computing the update step and describe the parameters under which each is likely to be fastest.

Because the WSK algorithm can be used to train unweighted SVMs over RCHs by setting all weights to equal one, the implementation details in this section also improve on some existing RCH nearest point algorithms. Further optimization specifically for RCH nearest point algorithms is a worthy goal, since many existing highly optimized nearest point algorithms take advantage of properties which are unique to convex hulls. In particular, they tend to rely on the property that all vertices in a convex hull are points from the input set [65, 71]. Since vertices in an RCH are convex combinations of points from the input set, some of these algorithms can be difficult to apply directly to RCHs.

5.6.1 One-class vs Two-class

It was mentioned in Section 5.2 that there are two ways of implementing nearest point algorithms which train SVMs. They can operate over two classes, or they can operate over a single class (where that class is the Minkowski set difference of the two classes). At a glance, it may appear preferable to choose the single class option because of the increased simplicity in implementing the algorithm. Although they employ the one-class approach, Keerthi et al. [65] note that its increased simplicity does not necessarily translate into increased efficiency. We elaborate on this and suggest that, in the case of the WSK algorithm, the two-class approach can actually be implemented more efficiently than the one-class approach.

The reason we prefer the two-class approach is that the update step can be made more efficient. This increased efficiency stems from the fact that each update step in the two-class approach depends on only a portion of the kernel matrix. For example, the update step for the one-class S-K algorithm is given by:

$$\lambda' = \frac{\sum_{i,j=1}^{n} \alpha_i(\alpha_j - \beta_j) y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j}{\sum_{i,j=1}^{n} (\alpha_i - \beta_i)(\alpha_j - \beta_j) y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j}. \tag{5.21}$$

For a simplified analysis, we refer to the unclamped λ' in this section, which is related to λ by $\lambda = \text{clamp}\,(\lambda', 0, 1)$.

Because $f_k = \sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x}_k$ values are generally kept cached during training (as we will discuss in the following section), the main cost involved in computing the update step in (5.21) is the denominator. This operation requires at most n^2 kernel products, where

n is the total number of training points. Compare this to the two-class S-K update step for the positive class:

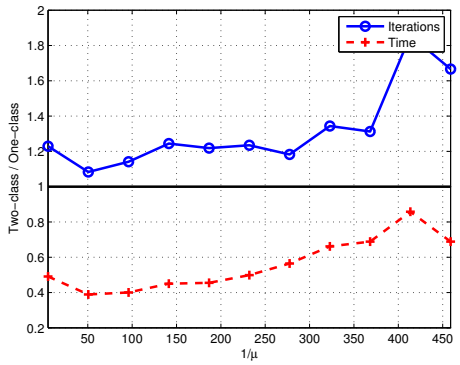
$$\lambda' = \frac{\sum_{i=1}^{n} \sum_{j \in I_{pos}} \alpha_i(\alpha_j - \beta_j) y_i \mathbf{x}_i \cdot \mathbf{x}_j}{\sum_{i \in I_{pos}} \sum_{j \in I_{pos}} (\alpha_i - \beta_i)(\alpha_j - \beta_j) \mathbf{x}_i \cdot \mathbf{x}_j}.$$

The denominator of this fraction requires a maximum of only n_{pos}^2 kernel products, where n_{pos} is the number of points in the positive class. If there are an approximately equal number of points in each class, this means the two-class update step requires only around a quarter the number of kernel products as the one-class update step.

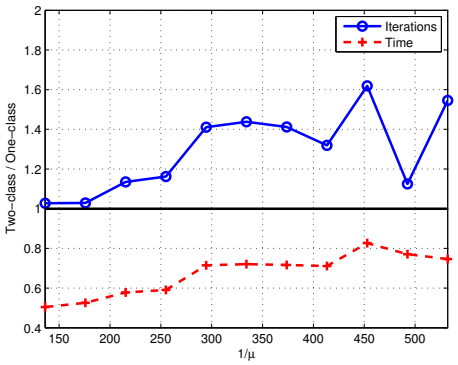
Of course, the increased efficiency of the update step in the two-class algorithm must be measured against any decreased efficacy (i.e. the total improvement made in the course of the update step). There can sometimes be a decrease in the efficacy of the two-class update step resulting from the fact that only a single class is involved in each update step, requiring more iterations in total.

Figure 5.9 compares both the number of iterations and overall training time for the one and two-class approaches. The datasets used to generate Figure 5.9 were chosen to give a relatively broad coverage in terms of number of features and number of training samples. These graphs show the number of iterations and the running time of the two-class approach, *relative* to the number of iterations and running time of the one-class approach. A value of $y = 0.5$ for time indicates that the two-class approach completed twice as fast as the one-class approach. Similarly, a value of $y = 2$ for iterations indicates that the two-class approach required twice as many iterations as the one-class approach in order to train an SVM to the same precision. We use this format because running times increased drastically as μ increased, so details in the graph were obscured when not using relative comparisons. Identical stopping conditions were applied to both algorithms.

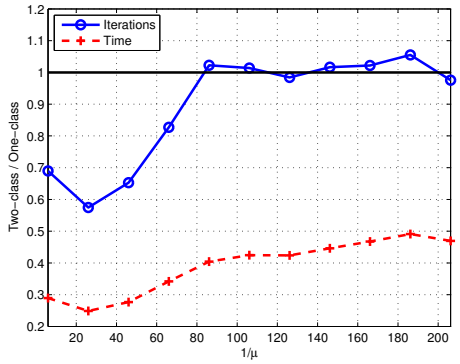
Figure 5.9 shows that the two-class approach tends to require more iterations than the one-class approach when applied to the same datasets. However, it also shows that this increase in the number of iterations does not outweigh the increased efficiency of the two-class update step. This effect was more pronounced when μ values were large. For example, on the `splice` and `image` datasets, the two-class approach was around twice as fast for large values of μ (depicted in the Figure as a small value of $1/\mu$). Even for other parameters on these datasets, the two-class approach still tended to be faster. By comparison, on the `banana` dataset, which was smaller and of lower dimensionality, and trained extremely quickly, there was little difference in the running times of the two algorithms. Overall, we found that the two-class approach tended to be the more efficient approach under most circumstances. For this reason, we focus mainly on the two-class nearest point approach in this chapter.



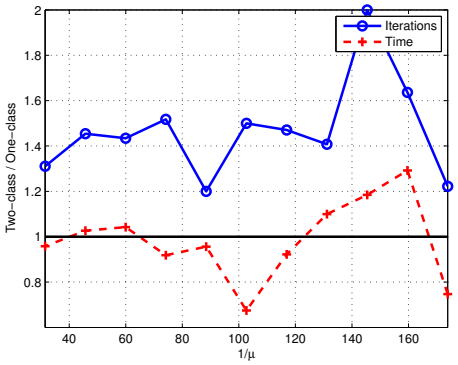
(a) `splice` dataset, $\gamma = 0.01$



(b) `image` dataset, $\gamma = 0.01$



(c) `german` dataset, $\gamma = 0.1$



(d) `banana` dataset, $\gamma = 1$

Figure 5.9: Comparing the one and two-class nearest point approaches. Points on these graphs show how the two-class algorithm performed relative to the one-class algorithm. For example, a value of $(x, y) = (200, 1.2)$ indicates that, for $\mu = 1/200$, the two-class approach took 20% longer, or required 20% more iterations, than the one-class approach. The dark line along $y = 1$ provides a basis for comparison since a ratio of 1 indicates equality between the two approaches for that μ value.

5.6.2 Maintaining a Cache

Caching frequently used values is an effective way of accelerating an SVM solver [91, 58, 66]. Some of the most useful values to cache in this context are the f-values:

$$f_i = \sum_k \alpha_k y_k \mathbf{x}_k \cdot \mathbf{x}_i$$

A cache of f-values is desirable for two reasons. First, notice that $f_i = \mathbf{w} \cdot \mathbf{x}_i$, so f-values can be used to find the $\lceil 1/\mu \rceil$ points which have the largest scalar projection onto $\mathbf{w}$, and hence can be used to find the vertices of the RCH needed to update the approximate nearest points. Second, f-values can be used to accelerate computation of the update step itself (i.e. calculating λ), as we will discuss in the next section.

In our implementation all f-values are cached, and updated after each iteration. This is a larger cache than most SMO implementations, which, depending on the heuristic they use, often only cache f_i values associated with a non-zero α_i, or even a subset of these

[91, 65]. However, we use a larger cache in our WSK implementation since it provides access to all f-values which are needed in order to find the update vertex with each iteration.

The cache is updated after each iteration by considering the WSK update step (for the positive class):

$$\mathbf{p}_{pos}^{new} = (1-\lambda)\mathbf{p}_{pos} + \lambda \mathbf{v}_{pos} \tag{5.22}$$

$$= (1-\lambda) \sum_{i \in I_{pos}} \alpha_i \mathbf{x}_i + \lambda \sum_{i \in I_{pos}} \beta_i \mathbf{x}_i \tag{5.23}$$

During the process of this update step (after λ has been computed, but before α_i's have been updated), the cache can be updated in either of two ways (Equations 5.24 and 5.25).

$$f_i = f_i^{old} + \lambda \sum_k y_i(\beta_k - \alpha_k)\mathbf{x}_k \cdot \mathbf{x}_i \tag{5.24}$$

$$= (1-\lambda) f_i^{old} + \lambda \sum_k y_i \beta_i \mathbf{x}_k \cdot \mathbf{x}_i \tag{5.25}$$

The best way to update the cache depends on the properties of the nearest point problem as well as the parameters. Although it may seem unlikely that a β_k should be exactly equal to an α_k, this is actually a common occurrence, particularly in later iterations. These matches occur because the current nearest point in the class is generally close to the outside of the hull, creating many α values which are capped at $\alpha_k = s_k\mu$. Similarly, the update point must also be on the outside of the class (and probably in a similar region of the hull, particularly as the algorithm progresses).

In general, once Condition (5.26) is satisfied, Equation (5.24) will provide the faster cache update. Otherwise, Equation (5.25) will be faster.

$$\text{card}\left(\{i \mid \beta_i = \alpha_i\}\right) < \text{card}\left(\{i \mid \beta_i \neq 0\}\right) \tag{5.26}$$

The LHS of this inequality is equal to the number of points for which the equality $\beta_i = \alpha_i$ holds. The RHS is equal to the number of points for which the condition $\beta_i \neq 0$ holds. Notice that the RHS is equal to the number of support points in the current update vertex. It follows that if weights are all equal to one, the RHS will be equal to $\lceil 1/\mu \rceil$.

Another choice depending on Condition (5.26) occurs when computing the update step, so we will analyze further the circumstances under which (5.26) is likely to be satisfied in the next section.

5.6.3 Computing the Update Step

Recall that the S-K algorithm update step for the positive class is performed by computing:

$$\lambda' = \frac{\lambda'_{top}}{\lambda'_{bot}} = \frac{\sum_{i=1}^{n} \sum_{j \in I_{pos}} y_i \alpha_i (\alpha_j - \beta_j) \mathbf{x}_i \cdot \mathbf{x}_j}{\sum_{i \in I_{pos}} \sum_{j \in I_{pos}} (\alpha_i - \beta_i)(\alpha_j - \beta_j) \mathbf{x}_i \cdot \mathbf{x}_j}. \tag{5.27}$$

Using cached f-values, the numerator of the fraction on the right hand side of Equation (5.27) can be expressed as:

$$\lambda'_{top} = \sum_{i \in I_{pos}} (\alpha_i - \beta_i) f_i.$$

This is generally the most efficient way of computing λ'_{top}. However more options are available for computing λ'_{bot}, which can be expressed as either of:

$$\lambda'_{bot} = \sum_{i \in I_{pos}} \sum_{j \in I_{pos}} (\alpha_i - \beta_i)(\alpha_j - \beta_j) \mathbf{x}_i \cdot \mathbf{x}_j \tag{5.28}$$

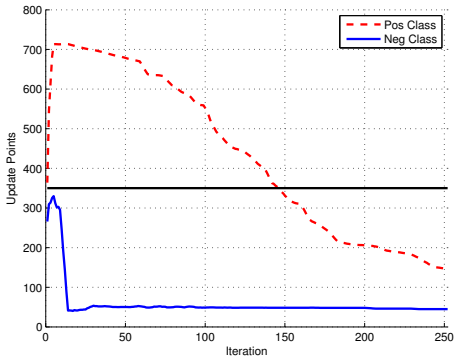
$$= \sum_{i \in I_{pos}} \sum_{j \in I_{pos}} (\beta_i \beta_j \mathbf{x}_i \cdot \mathbf{x}_j) + \sum_{i \in I_{pos}} (\alpha_i - 2\beta_i) f_i \tag{5.29}$$

Equation (5.29) is the calculation used by Franc and Hlavàč [40], whereas Equation (5.28) is another option that is preferable if most points satisfy $\alpha_i = \beta_i$.

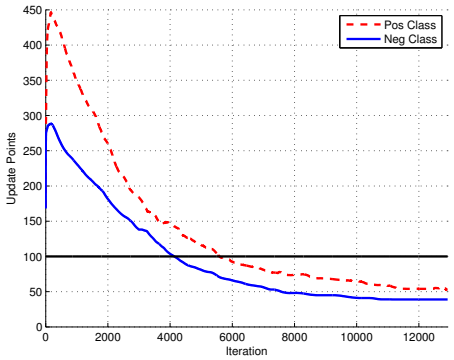
Figure 5.10 depicts the number of training points for which $\alpha_i \neq \beta_i$ after each iteration of the WSK algorithm. This is the number of points that would need to be included in the calculation of Equation (5.28). A smaller number of points makes the calculation more efficient. All weights have been made equal to one, meaning the number of support points in each update vertex is given by $\lceil 1/\mu \rceil$ (shown as a dark horizontal line in the figure). When the number of training points for which $\alpha_i \neq \beta_i$ dips under the horizontal line, update steps in Equations (5.24) and (5.28) become more efficient than those in Equations (5.25) and (5.29).

Notice how, for an L_1-loss SVM with $\mu < 1$, the number of points for which $\alpha_i \neq \beta_i$ is likely to decrease as the algorithm progresses. The main exception to this trend is in Figure 5.10e, where the number of points satisfying this condition tends to stay consistent throughout training. The reason we suggest for this is that this machine uses a Gaussian kernel with very narrow width (i.e. large γ). Recall from Section 4.2.5 that, with this type of kernel, all points are likely to become support vectors, resulting in an overfitted decision surface that resembles a k-nearest neighbor classifier. Under such circumstances almost all points will satisfy $0 < \alpha_i < \mu$, in which case they can not satisfy $\alpha_i = \beta_i$.

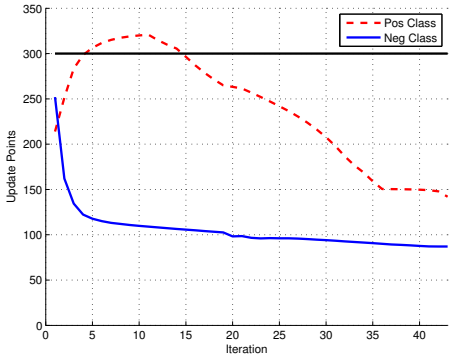
Figure 5.10 demonstrates that an appropriate choice of how to update the cache and the nearest points can reduce the number of kernel products which must be computed. The number of kernel products computed is sometimes used as a measure of efficiency for SVM training algorithms [109, 40]. However, our implementations use an extensive kernel



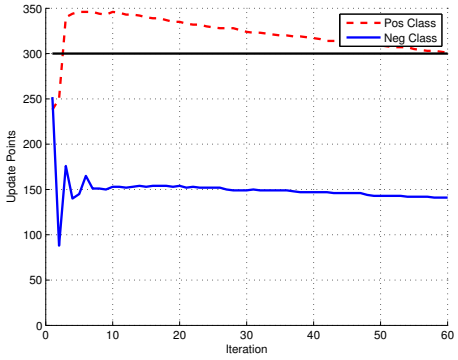
(a) `image` dataset, linear kernel with $\mu = 1/350$



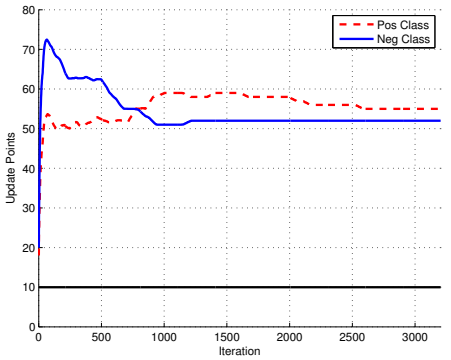
(b) `image` dataset, Gaussian kernel with $\mu = 1/100$,$\gamma = 0.01$



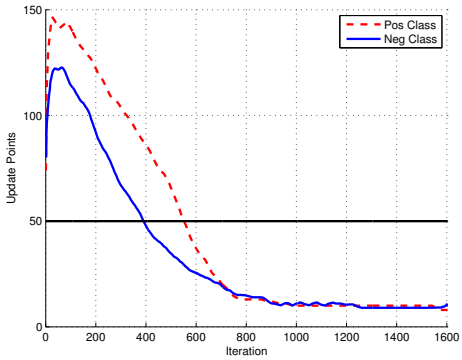
(c) `splice` dataset, linear kernel with $\mu = 1/300$



(d) `splice` dataset, Gaussian kernel with $\mu = 1/300$,$\gamma = 0.01$



(e) `banana` dataset, Gaussian kernel with $\mu = 1/10$,$\gamma = 10$



(f) `banana` dataset, polynomial kernel with $\mu = 1/50$,$d = 4$

Figure 5.10: Choosing the most efficient update step. Here the number of points satisfying $\alpha_i \neq \beta_i$ is shown for each iteration of the WSK algorithm. When this number drops below the dark horizontal line, the update step in (5.24) becomes more efficient than (5.25).

matrix cache combined with fast vector and matrix routines implemented in MATLAB. Due to this, we found that the impact of the reduced number of kernel products was negligible in terms of training times.

5.6.4 Reaching the Outside of the Hull

The S-K algorithm (and by extension the WSK algorithm) has a tendency to become 'stuck' inside the hull and suffer from slow convergence (refer to Section 5.2.1). For the (hard margin) convex hull task, Kowalczyk [71] addresses this problem by combining the S-K update step which moves the current nearest point towards an update vertex (an *increaseStep*) with an update step which moves the nearest point away from one of its support points (a *decreaseStep*). The *decreaseStep* is an important addition because it helps the α_i of unwanted support points reach zero, something the S-K algorithm can not quickly do. Similarly, Keerthi et al. [65] uses an S-K/MDM hybrid which has the capability to reduce points to zero.

Most of the previous methods of helping the S-K algorithm reach the outside of the hulls have been for the convex hull case. For the RCH nearest point problem, Tao et al. [109] has attempted to improve the rate with which the S-K algorithm pushes points to the outside of the RCHs. Recall from Section 5.2.3 that Tao et al. uses the S-K update step:

$$\mathbf{p}_{pos}^{new} = (1 - \lambda)\mathbf{p}_{pos} + \lambda \mathbf{w}_{pos},$$

However, rather than $\mathbf{w}_{pos}$ being a vertex of the RCH, it is instead given by:

$$\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i + \gamma(\mathbf{x}_{dst} - \mathbf{x}_{src}),$$

where $\gamma = \min(\alpha_{dst}, \alpha_{src})$, and:

$$\begin{aligned} src &= \arg\max_{i:\alpha_i > 0} \mathbf{w} \cdot \mathbf{x}_i \\ dst &= \arg\min_{i:\alpha_i < \mu} \mathbf{w} \cdot \mathbf{x}_i \end{aligned} \tag{5.30}$$

This step ensures that the constraints of the nearest point problem retains intact, while attempting to reduce α_i values more quickly to zero when appropriate. We tried this approach, but it did not seem to increase significantly either the speed of convergence or the number of Lagrange multipliers which were set to zero. This observation is supported by the empirical trials in the following section. The limited improvement is most likely due to the fact that points are only zeroed if $\lambda = 1$ is computed for the update step, which is an extremely infrequent occurrence.

Because of the limited ability for existing approaches to quickly reach the outside of the hull, we use an intermediate step between iterations which aims to move directly towards the outside of the hull. This is achieved by, for the positive class, shifting weight from a Lagrange multiplier α_{src} to another Lagrange multiplier α_{dst}. This transfer of weight should bring α_{dst} as close to its upper bound as possible, and α_{src} as close to its lower bound as possible. However, this needs to be achieved *without* moving the current approximations for the two nearest points any farther apart in the process.

Ideal values of *src* and *dst* satisfying the objectives stated above are those used by the MDM algorithm. However, we can not shift enough weight to cap either the source or destination α_i value because this may not decrease the distance between the nearest points. We could evaluate the objective function and only shift the weight if it improved the value of the objective function. However, it is no more difficult to compute the optimal amount of weight to shift by minimizing $||\mathbf{w} - \delta\mathbf{x}_{src} + \delta\mathbf{x}_{dst}||^2$.

Recall from Section 5.3 that this process of shifting weight between two Lagrange multipliers describes a standard SMO iteration on these indices. Furthermore, the indices *src* and *dst* correspond to an SMO heuristic using maximal violating pairs over the entire training set[22]. This means that if this shift alone was repeated it would in fact eventually converge on the solution. However, rather than repeating this step until convergence, we use it as an intermediate step in order to help the approximate nearest points reach the outside of the hull. This means that the algorithm we describe can be considered an MDM-SK or SMO-SK hybrid.

The main benefits of this intermediate SMO step is that it helps push α_i values towards their bounds of $[0, s_i\mu]$. This is beneficial when the KKT conditions associated with the nearest point problem are considered (Figure 5.11). These KKT conditions indicate that points from the positive class with large $\mathbf{w} \cdot \mathbf{x}_i$ are most likely be able to be set to $\alpha_i = 0$. Similarly, points from the positive class with small $\mathbf{w} \cdot \mathbf{x}_i$ are most likely to be able to be set to $\alpha_i = s_i\mu$.

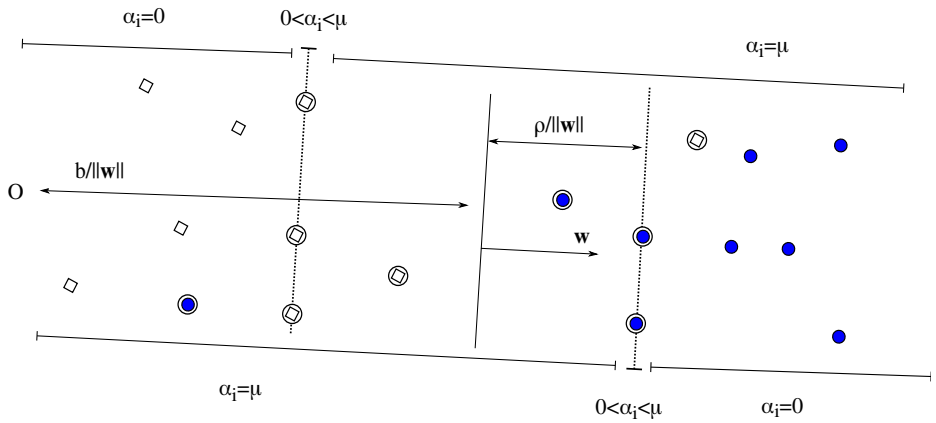


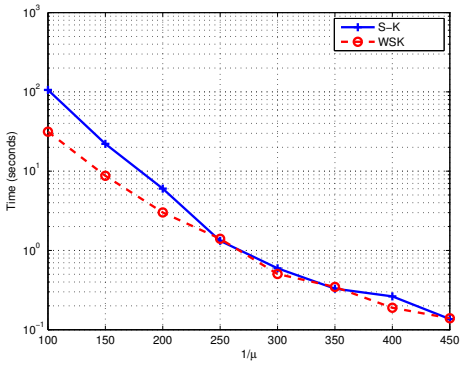
Figure 5.11: The KKT conditions associated with the RCH nearest point problem. Position class represented by filled circles. Negative class represented by diamonds. Support vectors are circled.

Helping to push α_i values towards a bound accelerates the WSK algorithm in several ways. First, it helps prevent the approximate nearest points from becoming 'stuck' inside the hulls, therefore decreasing the total number of iterations required. Second, it increases the number of points for which $\alpha_i = 0$ or $\alpha_i = s_i\mu$, which has the potential to accelerate each iteration of the algorithm which is taken. The acceleration occurs when Equations (5.24) and (5.28) are used to perform the update, which become increasingly fast as more α_i values reach their bound.

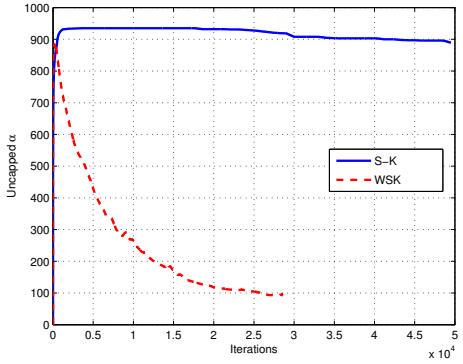
Figure 5.12 shows the number of unbounded support vectors at each iteration of the algorithm. Recall from Section 2.4.1 that we define an unbounded support vector as a

training point which satisfies $0 < \alpha_i < s_i\mu$. Notice how the use of the accelerated algorithm leads to a significant decrease in the number of unbounded support vectors.

The time required to execute the accelerated WSK method compared to the standard S-K method is shown in Figure 5.12a. This time was calculated on the `image` dataset using a Gaussian kernel with $\gamma = 0.01$. Notice how the accelerated algorithm tends to become relatively much faster than the standard algorithm as μ becomes larger. More exhaustive empirical comparisons in the following section will confirm with greater certainty the effectiveness of the accelerated method.



(a) Training time as μ varies



(b) Number of unbound SVs for $\mu = 1/100$

Figure 5.12: Comparing the standard S-K algorithm to accelerated WSK on the `image` dataset. The Gaussian kernel with $\gamma = 0.01$ is used.

5.7 Comparing Nearest Point Implementations

Several authors have drawn different conclusions regarding the efficiency of S-K algorithms in general. For example, Keerthi [63] suggests that the S-K algorithm is rather slow to converge due to the approximate nearest points often becoming 'stuck' inside the hull. However, Mavroforakis [81] suggests that S-K algorithms are generally equivalent to or better than SMO in terms of training time for SVMs. In this section we compare our WSK implementation to the previous RCH nearest point implementations. We compare the algorithms based on training time.

Although it is common practice to also compare SVM training algorithms based on the number of kernel evaluations [83, 109, 40], such comparisons can produce biased results. This is due to the fact that most modern SVM implementations use at least some level of caching in order to accelerate the training process [91, 20, 58]. It follows that the number of kernel evaluations does not necessarily relate directly to training time.

5.7.1 Comparison as Parameters Change

Figure 5.13 shows a comparison of the four algorithms on a limited selection of datasets. For more extensive results refer to Appendix C. In these figures, S-K denotes the standard S-K algorithm, and WSK denotes the Weighted Schlesinger-Kozinec algorithm, which contains additional acceleration steps described in Section 5.5. TAO denotes the algorithm

described by Tao et al. [109], which is best described as a hybrid between the MDM and S-K algorithms. SMO is the SMO algorithm using the maximal violating pair heuristic over the entire training set (equivalent to the MDM algorithm). For consistency, all algorithms use kernel and f-value caching and are implemented in the same MATLAB framework.

The S-K, WSK and TAO algorithms use identical relative nearest point stopping conditions (described in Section 5.2.4) with $\epsilon = 10^{-3}$. However, it was not feasible to use this same stopping condition with the SMO algorithm since it does not operate on vertices of the WRCHs of the classes like the other algorithms do. Computing these vertices with each iteration solely for the purpose of checking the stopping conditions was slower than simply using an alternate stopping condition. For this reason we instead used KKT stopping conditions with $\epsilon = 10^{-3}$. We will describe this stopping in more detail in Section 5.8. We will also examine the impact differing stopping condition have on training time and accuracy.

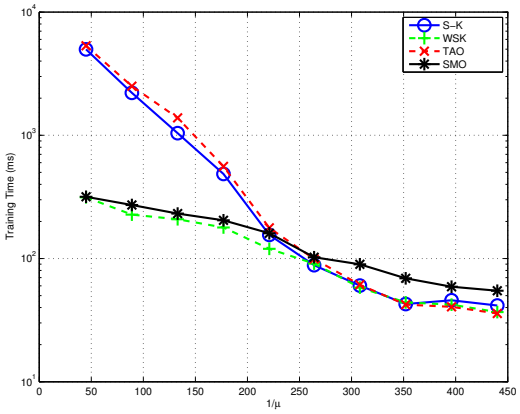
The results on the four datasets shown in Figure 5.13 are characteristic of most datasets in that SMO tended to perform better than the nearest point algorithms as μ became large (forcing the margin to become small). The nearest point algorithms showed greatest efficiency when μ was small (forcing the margin to become large). This is reasonable behavior since these SVMs have a large number of support vectors which can all be updated simultaneously by the nearest point update step, whereas SMO is forced to update support vectors only two at a time.

The nearest point algorithms could also perform well for larger values of μ, *provided* the kernel parameters were set so that a large number of support vectors were found (i.e. Gaussian RBF kernels with a large width). For an example of this, refer to the results for the `splice` dataset with $\gamma = 1$ (Figure C.7 in Appendix C). However, it is important to note that kernel parameters which force this number of support vectors generally results in a decision surface which has been overfitted to the data (as evidenced by the test error for the `splice` dataset in Figure C.16), so these parameters are rarely used in practice.

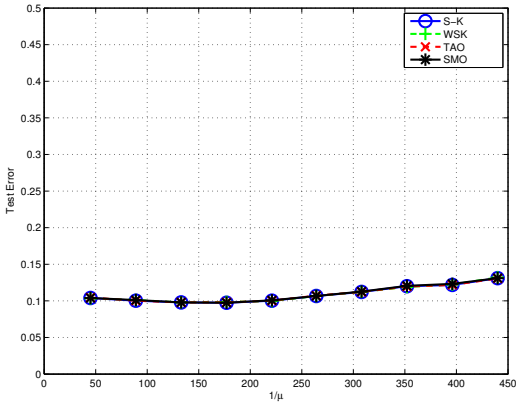
5.7.2 Comparison as Training Set Sizes Change

To test whether any of the algorithms become more or less efficient as the training set size changes we also performed a large scale test using the `forest` dataset. The `forest` dataset was originally gathered by Blackard [12] with the aim of classifying the type of forest cover in an area. We use only the 'Spruce/Fir' and 'Lodgepole pine' classes (the two larger classes) and discard the smaller ones in order to create a binary classification task, as is sometimes done for benchmarking binary classifiers [113, 26].

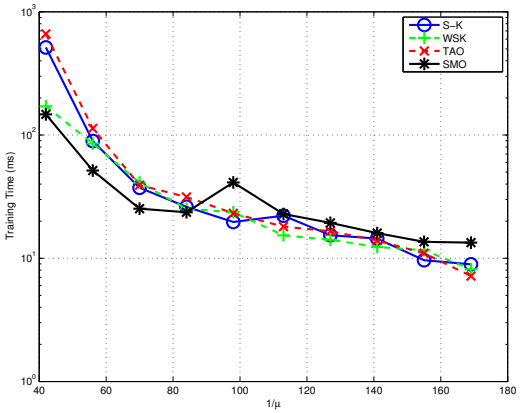
We use a random subset of $30,000$ points from the two classes of the `forest` dataset, create a subset of the required size for training and use the remaining data for testing. The Gaussian kernel is used with $\gamma = 1/10000$, as suggested by Tsang et al. [113]. The reduction in the hulls is specified as a multiple of the sum of weight in the smaller class, κ (refer to Section 4.4.5). The results of these trials are shown in Figure 5.14, with the number of training points on the x-axis. The y-axis shows the training time for figures on the left, and test error for figures on the right.



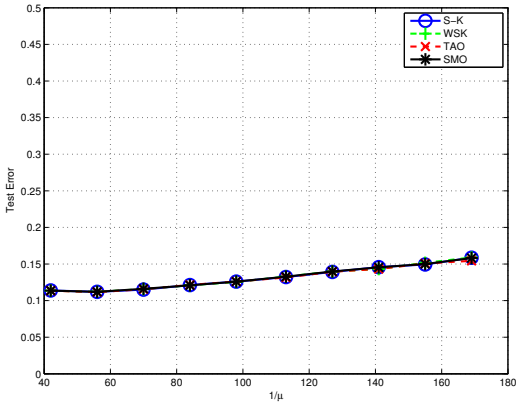
(a) Training time for `splice` dataset, $\gamma = 0.01$



(b) Error on the `splice` dataset, $\gamma = 0.01$



(c) Training time for `banana` dataset, $\gamma = 1$



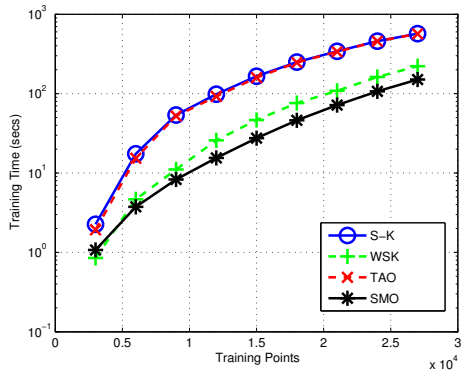
(d) Error on the `banana` dataset, $\gamma = 1$

Figure 5.13: Selected results for the four algorithms

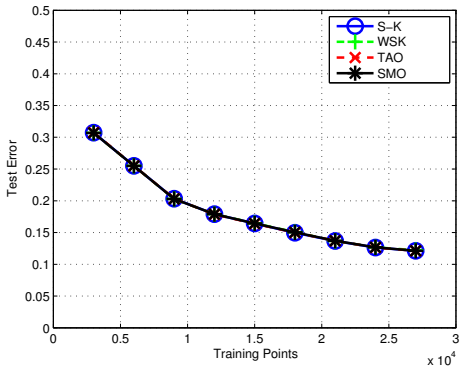
The results on the larger `forest` dataset are consistent with those of other datasets in that the SMO algorithm tended to perform much better than other algorithms when μ was large, whereas the nearest point algorithms became faster for smaller μ. Increasing or decreasing the amount of training data that was used did not have a significant impact on this relationship for the `forest` dataset. Interestingly, the value of μ chosen on this dataset did not have a great impact on the final test accuracy, whereas test error decreased dramatically as the amount of training data increased. However, as observed in the extensive results (Appendix C), it is not common for the value of μ to have such a minimal impact on test error, so μ should not be defaulted to a small value simply to decrease training times.

5.7.3 Discussion

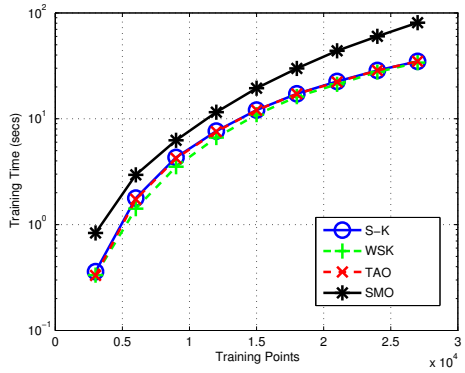
Our results qualify previous results reported by Mavroforakis [81]. Mavroforakis suggested that the S-K algorithm operating over RCHs (in standard form, not our accelerated version) "exhibits equivalent or better speed results" compared to SMO. However, our results suggest that, when the margin of the SVM being trained is small, the S-K algorithm is generally not as fast as SMO, although this gap could be bridged to some extent using



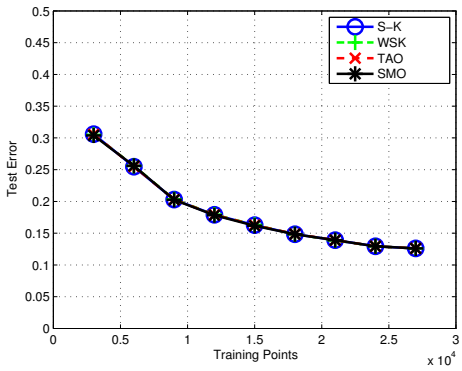
(a) Training time (seconds) for $\mu = 1/(0.2\kappa)$



(b) Error for $\mu = 1/(0.2\kappa)$



(c) Training time (seconds) for $\mu = 1/(0.8\kappa)$



(d) Error for $\mu = (1/0.8\kappa)$

Figure 5.14: The results of the four algorithms on the `forest` dataset

optimizations described in previous sections. Results also suggested that SMO tended to perform better on 'harder' tasks where there is a small margin and long training times.

The results obtained do not precisely match those reported by Mavroforakis [81]. The explanation for our differing results is that we applied the algorithms using a wide range of parameter values. The parameters had an enormous influence on the training time for both algorithms. However, Mavroforakis used only a single value per parameter for each algorithm in their comparisons. Furthermore, the parameter values they used when comparing different algorithms were not equivalent, with large C values for SMO (which are slow to train) compared against small μ values for the nearest point approach (which are fast to train).

The near-identical test accuracies of the SVMs produced by each of the four algorithms is evidence that each algorithm has trained an SVM to the same level of precision. In any case where different SVM training algorithms are compared, test accuracies should generally be the same or it indicates that either the comparison is biased or the stopping conditions are extremely loose.

Something we do not address in this section is the fact that the nearest point and SMO algorithms have differing stopping conditions. Although both stopping conditions used were comparable (as evidenced by the near identical test accuracies shown by the four algorithms), it is not clear what kind of impact these stopping conditions have on the

final SVM. In the following section we analyze these stopping conditions to discover how they impact on the training time and accuracy of an SVM.

It is informative to note the impact the choice of parameters (particularly μ) has on training times. Often training times could increase up to 100-fold simply by increasing μ. However, these large μ values are by no means 'bad' parameters, since they can sometimes achieve the lowest test error (refer to, for example, results on the `banana` and `image` datasets in Appendix C). Faster training times for smaller μ values are due to the way that an SVM approaches a k-means classifier as μ shrinks, which is a rather simple classifier which can be easily computed.

A good understanding of the relationship between μ and training times helps to understand how alternative training algorithms such as Joachims's [60] `SVMperf` can claim to scale linearly as the training set size increases (for linear SVMs). This can be achieved because the parameter used by `SVMperf` is given by:

$$C = \frac{\mu}{2\rho} = \frac{100 \times C_{perf}}{n}.$$

Note that, as n grows, the equivalent C or μ values used by `SVMperf` shrinks. This enables the algorithm to compensate for the increase in training time that would have been caused by the increased training set sizes by further reducing the hulls, which decreases training time.

5.8 Impact of the Stopping Conditions

There are many approaches to training SVMs, and often these approaches entail differing stopping conditions. However, when the stopping conditions differ it is not always clear whether differences in training times are due to a genuinely more efficient approach, or simply stopping conditions which allow for a greater level of approximation. In this section we compare several of the most commonly used stopping conditions for SMO and nearest point algorithms in order to determine how significantly the stopping conditions contribute to both training time and test error.

5.8.1 Types of Stopping Conditions

Many stopping conditions are possible. We experiment with what we consider to be the four most viable stopping conditions. We explain the viability of these stopping conditions below.

Absolute KKT The absolute KKT stopping conditions terminate once the KKT conditions are satisfied to within a tolerance ϵ:

$$\begin{aligned} y_i(\mathbf{w} \cdot \mathbf{x}_i - b) &< \rho \pm \epsilon \qquad \text{for } \alpha_i = C \\ y_i(\mathbf{w} \cdot \mathbf{x}_i - b) &= \rho \pm \epsilon \qquad \text{for } 0 < \alpha_i < C \\ y_i(\mathbf{w} \cdot \mathbf{x}_i - b) &> \rho \pm \epsilon \qquad \text{for } \alpha_i = 0 \end{aligned}$$

These stopping conditions are most commonly used in conjunction with SMO methods designed for C-SVMs [91, 66]. Within this context, these stopping conditions are reasonable since ρ is essentially fixed to equal one regardless of the width of the margin. However, in a μ-SVM, using this stopping condition poses a potential issue since ρ will change depending on the width of the margin. If ρ is large, it could potentially take a very large amount of computational effort to satisfy these stopping conditions. Conversely, if ρ is small (particularly if $\rho \approx \epsilon$), these stopping conditions may terminate before iterations have converged.

Relative KKT An alternative to the absolute KKT stopping conditions are the relative KKT stopping conditions:

$$\begin{aligned} y_i(\mathbf{w} \cdot \mathbf{x}_i - b) &< \rho(1 \pm \epsilon) && \text{for } \alpha_i = \mu \\ y_i(\mathbf{w} \cdot \mathbf{x}_i - b) &= \rho(1 \pm \epsilon) && \text{for } 0 < \alpha_i < \mu \\ y_i(\mathbf{w} \cdot \mathbf{x}_i - b) &> \rho(1 \pm \epsilon) && \text{for } \alpha_i = 0 \end{aligned}$$

These stopping conditions ensure that the tolerance parameter ϵ retains the same meaning regardless of whether ρ is large is small. This is achieved by specifying the stopping tolerance *relative* to ρ. Notice that, for a C-SVM, the absolute and relative KKT stopping conditions are interchangeable since ρ is essentially fixed to equal one. However, for a μ-SVM to be trained to an equivalent precision to a C-SVM, it must use the relative stopping conditions.

Absolute Nearest Point Recent nearest point algorithms [40] have favored using the absolute difference between $||\mathbf{w}||$ and $\mathbf{w} \cdot (\mathbf{v}_{pos} - \mathbf{p}_{neg})$. This is an upper bound on the distance between the current nearest point and the true nearest point.

$$\begin{aligned} & ||\mathbf{w}|| - \frac{\mathbf{w} \cdot (\mathbf{v}_{pos} - \mathbf{p}_{neg})}{||\mathbf{w}||} < \epsilon, \\ \text{and} \qquad & ||\mathbf{w}|| - \frac{\mathbf{w} \cdot (\mathbf{p}_{pos} - \mathbf{v}_{neg})}{||\mathbf{w}||} < \epsilon \end{aligned}$$

Relative Nearest Point We note that the absolute nearest point stopping condition used above suffers from similar drawbacks as the absolute KKT stopping conditions. That is, it becomes tighter for large $||\mathbf{w}||$ and looser for small $||\mathbf{w}||$. We suggest instead using:

$$\begin{aligned} & 1 - \frac{\mathbf{w} \cdot (\mathbf{v}_{pos} - \mathbf{p}_{neg})}{||\mathbf{w}||^2} < \epsilon, \\ \text{and} \qquad & 1 - \frac{\mathbf{w} \cdot (\mathbf{p}_{pos} - \mathbf{v}_{neg})}{||\mathbf{w}||^2} < \epsilon \end{aligned}$$

This stopping condition computes an upper bound on the distance between the current approximate nearest point and the actual nearest point, *relative* to the distance between the two approximate nearest points. This stopping condition was used by Keerthi et al. [65] in a convex hull nearest point algorithm. However, more recent nearest point algorithms

operating over RCHs seem to have instead adopted the absolute nearest point stopping conditions [40, 82, 109].

Some other stopping conditions, although rarely used in practice, are also possible. For example, López et al. [77] note that it is possible to stop when changes to $||\mathbf{w}||$ with each iteration become very small. However, we avoid this stopping condition since the size of changes to $||\mathbf{w}||$ are not strictly decreasing as the number of iterations increases.

5.8.2 Comparing Stopping Conditions

In this section we perform empirical trials which record the average number of iterations taken to satisfy each of the stopping conditions. We also record the test accuracy which is associated with each stopping condition. By keeping the training algorithm consistent while varying the stopping conditions, we are able to determine the impact each of the stopping conditions has on the final solution. Results are calculated as an average over 10 runs, with different train/test splits used on each run. We reuse the datasets previously introduced in this chapter, and described in more detail in Appendix A.

Table 5.5 shows the test error associated with each of the stopping conditions. Machines were trained using a Gaussian kernel with $\gamma = 0.1$. Large μ values were chosen, which were set to be equivalent to $C = 100$. The associated Table 5.6 shows the number of training iterations which were required before each stopping condition was reached. This experiment is repeated in Tables 5.7 and 5.8, using a polynomial kernal with degree 4, and using smaller μ values (set to be equivalent to $C = 1$). This broad mixture of parameter values ensure that results are obtained from SVMs with both small and large margins.

We use the SMO training algorithm in this section, since we found that many of the other nearest point algorithms took a too long to satisfy the KKT conditions. This is due to the way in which nearest point algorithms often leave α_i values which are very close to zero. While these values, if they are small enough, do not prevent the nearest point stopping conditions from being satisfied, they do cause the KKT conditions to be violated. By contrast, SMO is much better at precisely satisfying the KKT conditions and can therefore be used in conjunction with any of the stopping conditions.

Table 5.5: Test error associated with each stopping condition for the Gaussian kernel with $\gamma = 0.1$ and large μ

	KKT Rel			KKT Abs			Nearest Point Rel			Nearest Point Abs		
	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$
banana	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.39 ± 0.01	0.28 ± 0.01	0.14 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.35 ± 0.01	0.16 ± 0.01	0.13 ± 0.00
b.cancer	0.34 ± 0.02	0.34 ± 0.02	0.34 ± 0.02	0.45 ± 0.02	0.36 ± 0.03	0.36 ± 0.02	0.34 ± 0.02	0.34 ± 0.02	0.34 ± 0.02	0.41 ± 0.02	0.37 ± 0.02	0.34 ± 0.02
diabetes	0.30 ± 0.00	0.30 ± 0.00	0.30 ± 0.00	0.26 ± 0.01	0.33 ± 0.01	0.30 ± 0.00	0.30 ± 0.01	0.30 ± 0.00	0.30 ± 0.00	0.27 ± 0.01	0.29 ± 0.01	0.30 ± 0.00
german	0.26 ± 0.01	0.26 ± 0.01	0.26 ± 0.01	0.44 ± 0.03	0.30 ± 0.01	0.26 ± 0.01	0.27 ± 0.01	0.26 ± 0.01	0.26 ± 0.01	0.38 ± 0.01	0.27 ± 0.01	0.26 ± 0.01
heart	0.22 ± 0.01	0.22 ± 0.01	0.22 ± 0.01	0.27 ± 0.02	0.23 ± 0.01	0.22 ± 0.01	0.22 ± 0.01	0.23 ± 0.01	0.22 ± 0.01	0.24 ± 0.02	0.23 ± 0.02	0.23 ± 0.01
image	0.03 ± 0.00	0.03 ± 0.00	0.03 ± 0.00	0.15 ± 0.01	0.06 ± 0.00	0.03 ± 0.00	0.03 ± 0.00	0.03 ± 0.00	0.03 ± 0.00	0.12 ± 0.01	0.03 ± 0.00	0.03 ± 0.00
splice	0.38 ± 0.00	0.38 ± 0.00	0.38 ± 0.00	0.48 ± 0.02	0.46 ± 0.03	0.38 ± 0.00	0.38 ± 0.00	0.38 ± 0.00	0.38 ± 0.00	0.44 ± 0.04	0.38 ± 0.00	0.38 ± 0.00
thyroid	0.04 ± 0.01	0.05 ± 0.01	0.05 ± 0.01	0.09 ± 0.02	0.05 ± 0.00	0.04 ± 0.01	0.04 ± 0.00	0.05 ± 0.01	0.05 ± 0.01	0.06 ± 0.01	0.04 ± 0.01	0.04 ± 0.01
titanic	0.22 ± 0.00	0.22 ± 0.00	0.22 ± 0.00	0.25 ± 0.00	0.40 ± 0.03	0.35 ± 0.04	0.22 ± 0.00	0.22 ± 0.00	0.22 ± 0.00	0.33 ± 0.04	0.26 ± 0.02	0.23 ± 0.01

Table 5.6: Number of iterations associated with each stopping condition for the Gaussian kernel with $\gamma = 0.1$ and large μ

	$\|\|\mathbf{w}\|\|$	KKT Rel			KKT Abs			Nearest Point Rel			Nearest Point Abs		
		$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$
banana	8.837141e-03	3575	6455	10627	6	24	321	1029	2955	4948	8	120	1040
b.cancer	2.440872e-02	1454	2799	4156	12	100	631	911	1950	3296	24	428	1283
diabetes	1.427747e-02	5265	10084	15661	20	79	763	2321	6103	11010	25	482	2811
german	5.787472e-02	1340	2380	3496	25	230	943	808	1705	2757	62	646	1487
heart	1.341588e-01	246	436	633	23	132	292	154	318	512	48	172	341
image	1.802462e-02	4625	10262	18403	24	131	919	1701	5011	10086	34	513	2394
splice	6.280310e-02	1708	2476	3065	39	631	1539	1547	2119	2755	106	1352	1998
thyroid	7.463674e-02	135	394	763	8	40	124	73	194	470	13	62	167
titanic	1.113977e-02	123	280	544	13	27	66	196	438	746	15	69	190

Table 5.7: Test error associated with each stopping condition for the polynomial kernel with $q = 4$ and small μ

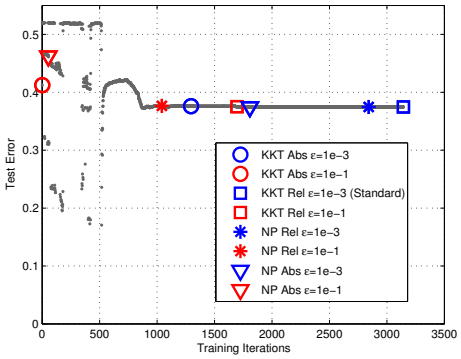
	KKT Rel			KKT Abs			Nearest Point Rel			Nearest Point Abs		
	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$
banana	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00	0.12 ± 0.00
b.cancer	0.38 ± 0.02	0.38 ± 0.02	0.38 ± 0.02	0.38 ± 0.02	0.38 ± 0.02	0.38 ± 0.02	0.38 ± 0.02	0.38 ± 0.02	0.38 ± 0.02	0.38 ± 0.02	0.37 ± 0.02	0.38 ± 0.02
diabetes	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01	0.33 ± 0.01
german	0.28 ± 0.01	0.28 ± 0.01	0.28 ± 0.01	0.28 ± 0.01	0.28 ± 0.01	0.28 ± 0.01	0.27 ± 0.01	0.28 ± 0.01	0.28 ± 0.01	0.28 ± 0.01	0.28 ± 0.01	0.28 ± 0.01
heart	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02	0.25 ± 0.02
image	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00	0.04 ± 0.00
splice	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00
thyroid	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01	0.06 ± 0.01
titanic	0.23 ± 0.01	0.22 ± 0.00	0.22 ± 0.00	0.22 ± 0.00	0.22 ± 0.00	0.22 ± 0.00	0.22 ± 0.00	0.22 ± 0.00	0.22 ± 0.00	0.23 ± 0.01	0.22 ± 0.00	0.22 ± 0.00

Table 5.8: Number of iterations associated with each stopping condition for the polynomial kernel with $q = 4$ and small μ

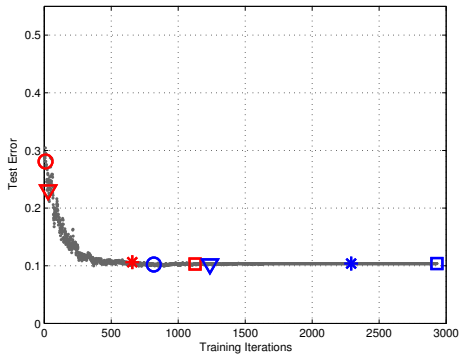
	$\|\|\mathbf{w}\|\|$	KKT Rel			KKT Abs			Nearest Point Rel			Nearest Point Abs		
		$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$
banana	2.816127e-01	1517	3004	5423	1017	2118	4860	646	1410	2317	355	1017	1791
b.cancer	1.325634e+00	3429	5434	7633	4435	6651	8805	2486	4373	6503	2722	4556	6812
diabetes	9.255594e-01	56441	103300	156314	58700	104629	157685	31251	70636	119327	30415	68928	117760
german	1.287650e+01	4310	6907	9546	10110	12778	15406	2709	5077	7727	5394	8024	10688
heart	1.847633e+01	471	753	1034	1180	1466	1752	312	585	858	655	928	1214
image	1.248155e+00	277714	740492	948073	395747	824637	957574	112005	382825	844544	124695	439553	861538
splice	5.500539e+03	3396	6332	9600	29266	32969	36706	1914	4384	7444	13246	16815	20386
thyroid	2.418854e+00	110	222	361	207	344	483	58	128	252	82	179	295
titanic	5.595426e-01	136	174	221	152	198	246	154	194	245	140	182	228

We measure how effective a stopping condition is by comparing its accuracy to that of the tightest stopping condition. For example, on the `banana` dataset for the Gaussian kernel (Table 5.5), a stopping condition is sufficient if the accuracy of the resulting SVM is comparable to that of the relative KKT stopping condition with $\epsilon = 10^{-3}$, which is the tightest stopping condition on that particular dataset (because it results in the greatest number of iterations being performed). We can not simply say that the stopping condition which results in the lowest overall accuracy is the best, because loose stopping conditions could potentially increase accuracy, particularly if the parameters overfit the dataset.

For an example of how looser stopping conditions can improve test error, refer to Figure 5.15a. This figure shows a plot of the test error for the `splice` dataset as the number of training iterations increases. Notice how a stopping condition could luckily choose the portion of the graph where the training accuracy is low. However, rather than trying to minimize the test error via the stopping conditions, we instead want a stopping condition which best approximates the exact solution, i.e. a solution on the more stable portion to the right of the graph. If test accuracy is insufficient with these stopping conditions, it indicates that the kernel parameters, rather than the stopping conditions, should be adjusted to increase the accuracy (Figure 5.15b).



(a) $\gamma = 0.1$. This parameter value is likely to lead to overfitting of the decision surface to the training data.



(b) $\gamma = 0.01$. The larger kernel width provides a much lower test error.

Figure 5.15: The test error (shown as grey dots) as the number of iterations increases. The `splice` dataset is used in conjunction with the Gaussian kernel. The reduction parameter has the value of $\mu = 1/30$

The results in Tables 5.5-5.8 indicate that the absolute stopping conditions are looser than the relative ones when $||\mathbf{w}||$ is small, and tighter when $||\mathbf{w}||$ is large. This means that they are less consistent that the relative stopping conditions. For example, absolute KKT or absolute nearest point stopping conditions with $\epsilon = 10^{-2}$ are loose enough to degrade accuracy on most datasets where $||\mathbf{w}||$ is small (Table 5.5), whereas on the `splice` dataset in Table 5.7 where $||\mathbf{w}||$ is largest, even the loosest absolute stopping condition of $\epsilon = 10^{-1}$ results in more iterations than any of the relative stopping conditions, and achieves no gain in test accuracy for this additional computational effort.

The inconsistency in the results of the absolute nearest point stopping conditions used in most previous RCH nearest point implementations [81, 109] lead us to recommend

the use of relative nearest point stopping conditions instead. The relative nearest point stopping conditions have been shown to be more consistent across differing datasets and parameter values.

Results also indicate that the 'standard' value of $\epsilon = 10^{-3}$ used with the relative KKT stopping conditions almost always results in more training iterations being taken than is necessary, *for the datasets used*. However, there is no guarantee that loosening the stopping conditions would not degrade accuracy on some other datasets. The relatively tight value of $\epsilon = 10^{-3}$ may simply be commonly used in practice because it is a 'safe' value. It is difficult to suggest further improvements on the relative KKT and nearest point stopping conditions without a) having prior knowledge of the data being used, or b) risking a lower test accuracy.

5.8.3 Discussion

Empirical trials run on the stopping conditions suggested that the relative KKT and relative nearest point stopping conditions were both adequate stopping conditions. However, there is the potential to decrease training times greatly if loose stopping conditions are used. Indeed, there were many circumstances were very few training iterations were required in order to achieve an adequate accuracy. Some algorithms, such as Tsang et al.'s Core Vector Machine (CVM) [114], exploit this observation by employing a looser stopping condition. By choosing such stopping conditions however, there is generally a risk that test accuracy can be impacted [76].

We suggest that there are two distinct components to an SVM training algorithm. The first component makes progress towards finding the maximum margin hyperplane, generally by tuning the Lagrange multipliers of the dual. The second component decides when to terminate. In order to make a fair comparison of two algorithms, both of these components must be taken into account. For example, if SMO with an extremely loose stopping condition is compared to a nearest point algorithm with a tight stopping condition, it is not clear whether changes to training time are due to the algorithm making more efficient progress towards finding a maximum margin hyperplane, or simply due to the algorithm terminating earlier.

5.9 Training Weighted Perceptrons using the WSK Algorithm

The algorithms we have previously described in this chapter address the task of finding the nearest points in the RCHs of two classes. By solving the nearest point task it is possible to train both hard margin SVMs, and L_1-loss SVMs. Because L_2-loss SVMs are simply hard margin SVMs in a modified kernel space, L_2-loss SVMs can also easily be trained with these algorithms.

Despite the fact that most nearest point algorithms have previously been applied to these specific types of SVMs, they may be adapted to train perceptrons. Recall that perceptrons are a kernel machine similar to SVMs where the sum of Lagrange multipliers in

each class is not constraint to be equal (refer to Section 2.8). In this section we describe how nearest point algorithms may be used to train hard margin and soft margin perceptrons (with both L_1 and L_2 loss functions, and optional point weights).

5.9.1 The Perceptron Dual Optimization Task

The main soft margin L_1-loss perceptron dual optimization task is given in (5.31). We address this optimization task since, given the appropriate parameter and kernel choices, it may also be used to compute L_2-loss and hard margin perceptrons. This is achieved by varying $k(\cdot,\cdot)$ and μ, as we described in Section 5.4. Refer back to Table 5.1 to see potential values for $k(\cdot,\cdot)$.

$$\begin{aligned} \max_{\alpha} \quad & -\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j k(\mathbf{x}_i,\mathbf{x}_j) \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n}\alpha_i = 1, \\ 0 \leq \alpha_i \leq s_i\mu. \end{cases} \end{aligned} \tag{5.31}$$

5.9.2 The One-Class WSK Algorithm

Recall from Section 2.8 that the perceptron dual in (5.31) is equivalent to finding the point in a WRCH with minimal norm. This means that the WSK algorithm operating over a single class can be used to solve this optimization task. A one-class variant of the WSK algorithm is shown in Algorithm 11.

Algorithm 11 The One-Class WSK Algorithm

function WSK_PERCEPTRON($P, \mathbf{s}, \mathbf{y}$)
  initialize $\mathbf{p}$ to any point from WRCH($P, \mathbf{s}, \mu$)
  loop
    $\mathbf{v} \leftarrow \underset{\mathbf{v}\in\text{WRCH}(P_{pos})}{\arg\max} -\mathbf{p}\cdot\mathbf{v}$ ▷ use Algorithm 7 to solve this
    if $1 - \mathbf{p}\cdot(\mathbf{p}-\mathbf{v})/||\mathbf{p}||^2 < \epsilon$ **then**
      break ▷ stopping conditions reached
    end if
    $\lambda \leftarrow \text{clamp}\left(\frac{(\mathbf{p})\cdot(\mathbf{p}-\mathbf{v})}{(\mathbf{p}-\mathbf{v})^2}, 0, 1\right)$
    $\mathbf{p}^{new} \leftarrow (1-\lambda)\mathbf{p} + \lambda\mathbf{v}$
  end loop
  $b \leftarrow -\sum_i \alpha_i y_i$
  return $(\mathbf{p}, b)$ ▷ return hyperplane normal and offset
end function

5.9.3 The Threshold

Similar to SVMs, there are multiple ways to compute the threshold of a perceptron. Each of these thresholds is likely to yield a difference in accuracy, so in order to compare the performance of SVMs and perceptrons it is important to consider carefully the impact of

the thresholds of the two machines. For perceptrons, the threshold may be derived from the KKT conditions as:

$$b = \sum_i -\alpha_i y_i$$

There is also a geometric threshold, which can be given by:

$$b = \sum_i \alpha_i K(\mathbf{x}_i, \mathbf{x}_j).$$

Note the use of the kernel $K(\mathbf{x}_i, \mathbf{x}_j)$ here, *not* the modified kernel $k(\mathbf{x}_i, \mathbf{x}_j)$ which is used in the objective function of the dual. Alternatively, the perceptron threshold can also be computed in the same way as the threshold given by the KKT conditions of the SVM optimization task:

$$b = \frac{1}{2}\left(\max_{i \in I_{pos}, \alpha_i > 0} \left\{ \sum_{k=1}^{n} \alpha_k K(\mathbf{x}_k, \mathbf{x}_i) \right\} + \min_{i \in I_{neg}, \alpha_i > 0} \left\{ \sum_{k=1}^{n} \alpha_k K(\mathbf{x}_k, \mathbf{x}_i) \right\} \right).$$

Before comparing SVMs and perceptrons, it is important to gauge the impact the different thresholds have on the accuracy of a perceptron. We determine the impact of the perceptron threshold by repeating the threshold experiments form Section 4.3, using perceptrons instead of SVMs. For consistency with previous experiments, 'large' μ values are equivalent to $C = 1$, while 'small' μ values are equivalent to $C = 0.1$. In these experiments we compare perceptrons with the KKT threshold, the geometric threshold, no threshold (the bias-free perceptron), and perceptrons using the SVM-style KKT threshold.

Results are shown in Tables 5.9 and 5.10, with error rates shown as a $mean \pm stderr$ over 20 runs. In these tables, error rates for the various thresholds are compared against the KKT threshold. An error rate is bolded if it is significantly lower (better) than that achieved by the KKT threshold. An error rate is underlined if it is significantly higher (worse) than that achieved by the KKT threshold. A paired differences t-test over the 20 runs is used for significance testing.

Notice how the differences in test error across the thresholds shown in Tables 5.9 and 5.10 are quite large in many cases. These differences are large enough that it would be easy to draw erroneous conclusions by comparing perceptrons and SVMs with different thresholds. For this reason our comparisons of SVMs and perceptrons in the next sections use the geometric threshold for both machines. This threshold tended to provide the best test accuracy when used in conjunction with perceptrons. This is consistent with SVM results from previous sections, in which the geometric threshold also tended to provide the greatest test accuracy.

5.9.4 Accuracy and Efficiency

We conducted some preliminary experiments using nearest point algorithms for perceptrons, and our results were consistent with Mangasarian and Musicant [79], who suggest that the perceptron and SVM optimization tasks both result in a similar test accuracy.

Table 5.9: Test error for Gaussian perceptrons ($\gamma = 0.01$) combined with four possible thresholds

		KKT	Geometric	Bias-Free	SVM-KKT
banana	small μ	0.448 ± 0.011	0.436 ± 0.010	0.448 ± 0.011	$\mathbf{0.417 \pm 0.010}$
	large μ	0.445 ± 0.010	0.436 ± 0.010	0.445 ± 0.010	$\mathbf{0.417 \pm 0.010}$
b.cancer	small μ	0.275 ± 0.011	0.277 ± 0.007	0.275 ± 0.011	0.271 ± 0.010
	large μ	0.277 ± 0.010	0.278 ± 0.007	0.277 ± 0.010	0.273 ± 0.010
diabetes	small μ	0.281 ± 0.005	$\mathbf{0.241 \pm 0.005}$	0.281 ± 0.005	$\mathbf{0.266 \pm 0.004}$
	large μ	0.256 ± 0.007	$\mathbf{0.239 \pm 0.005}$	0.256 ± 0.007	0.252 ± 0.006
german	small μ	0.248 ± 0.005	$\mathbf{0.235 \pm 0.005}$	$\mathbf{0.245 \pm 0.005}$	0.243 ± 0.008
	large μ	0.243 ± 0.005	$\mathbf{0.234 \pm 0.005}$	0.241 ± 0.005	0.240 ± 0.006
heart	small μ	0.184 ± 0.010	$\mathbf{0.159 \pm 0.008}$	0.182 ± 0.009	$\mathbf{0.174 \pm 0.009}$
	large μ	0.170 ± 0.009	$\mathbf{0.159 \pm 0.007}$	0.168 ± 0.008	0.169 ± 0.008
image	small μ	0.264 ± 0.002	$\mathbf{0.207 \pm 0.004}$	$\mathbf{0.264 \pm 0.002}$	$\underline{0.265 \pm 0.002}$
	large μ	0.194 ± 0.016	$\mathbf{0.162 \pm 0.010}$	0.194 ± 0.016	0.198 ± 0.016
splice	small μ	0.158 ± 0.002	$\mathbf{0.156 \pm 0.002}$	$\underline{0.165 \pm 0.002}$	$\underline{0.164 \pm 0.003}$
	large μ	0.139 ± 0.005	$\mathbf{0.138 \pm 0.004}$	$\underline{0.143 \pm 0.005}$	$\underline{0.151 \pm 0.005}$
thyroid	small μ	0.273 ± 0.011	$\mathbf{0.133 \pm 0.009}$	0.273 ± 0.011	$\mathbf{0.223 \pm 0.011}$
	large μ	0.222 ± 0.015	$\mathbf{0.119 \pm 0.009}$	0.222 ± 0.015	$\mathbf{0.197 \pm 0.011}$
titanic	small μ	0.255 ± 0.010	$\mathbf{0.226 \pm 0.001}$	0.255 ± 0.010	$\mathbf{0.239 \pm 0.007}$
	large μ	0.243 ± 0.008	$\mathbf{0.226 \pm 0.000}$	0.243 ± 0.008	$\mathbf{0.232 \pm 0.005}$

Results are shown in Table 5.11. The leftmost two columns of this table compare SVMs and perceptrons using Gaussian kernels, whereas the rightmost two columns compare SVMs and perceptrons using polynomial kernels. A perceptron error rate is bolded if it is significantly lower than the SVM error rate using the same kernel, and underlined if it is significantly higher.

Notice in these tables that the differences in test accuracy between the two machines are extremely small. In fact, the differences between SVMs and perceptrons when both using the geometric threshold tended to be less than the difference between two perceptrons using different thresholds, or two SVMs using different thresholds.

Despite the ability for perceptrons to achieve accuracy which is on par with SVMs, we focused in this chapter mainly on SVMs. This is because we found that the perceptron minimal norm task, despite its conceptual simplicity, could not be solved as efficiently as the SVM nearest point task. This is demonstrated in Figure 5.16, where we trained both perceptrons and SVMs using the accelerated WSK algorithm described in Section 5.6. The algorithms had a near identical implementation, with the only difference being that the perceptron algorithm was a one-class variant of the SVM algorithm. Despite the conceptual simplicity of the perceptron algorithm, it was slower across every dataset we tested on.

The slower progress of the WSK algorithm when applied to perceptrons can be more clearly examined by tracking the test error achieved by the machine after each individual iteration. This is the same method we used to analyze the stopping conditions in previous sections. Figure 5.17 shows the test error of the WSK algorithm applied to both perceptrons (left) and SVMs (right). Notice how the algorithm tends to converge more quickly

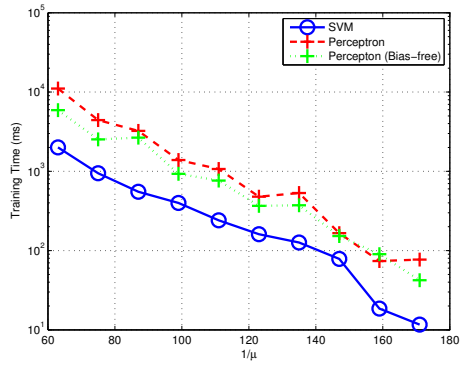
Table 5.10: Test error for polynomial perceptrons ($q = 3$) combined with four possible thresholds

		KKT	**Geometric**	**Bias-Free**	**SVM-KKT**
banana	small μ	0.235 ± 0.006	0.229 ± 0.006	0.238 ± 0.005	0.336 ± 0.022
	large μ	0.232 ± 0.006	$\mathbf{0.227 \pm 0.006}$	0.233 ± 0.005	0.319 ± 0.022
b.cancer	small μ	0.319 ± 0.008	0.351 ± 0.008	0.319 ± 0.008	0.314 ± 0.008
	large μ	0.332 ± 0.010	0.356 ± 0.010	0.332 ± 0.010	0.329 ± 0.010
diabetes	small μ	0.301 ± 0.006	0.304 ± 0.006	0.299 ± 0.006	0.308 ± 0.010
	large μ	0.312 ± 0.006	0.314 ± 0.006	0.312 ± 0.006	0.316 ± 0.008
german	small μ	0.303 ± 0.006	0.306 ± 0.007	0.303 ± 0.006	0.303 ± 0.006
	large μ	0.303 ± 0.006	0.306 ± 0.007	0.303 ± 0.006	0.303 ± 0.006
heart	small μ	0.224 ± 0.008	0.225 ± 0.008	0.224 ± 0.008	0.224 ± 0.008
	large μ	0.224 ± 0.008	0.224 ± 0.008	0.224 ± 0.008	0.224 ± 0.008
image	small μ	0.039 ± 0.002	0.040 ± 0.002	0.039 ± 0.002	0.080 ± 0.013
	large μ	0.041 ± 0.002	0.047 ± 0.005	0.041 ± 0.002	0.099 ± 0.015
splice	small μ	0.129 ± 0.001	0.136 ± 0.003	0.129 ± 0.001	0.130 ± 0.002
	large μ	0.129 ± 0.001	0.136 ± 0.003	0.129 ± 0.001	0.130 ± 0.002
thyroid	small μ	0.074 ± 0.008	0.068 ± 0.007	0.074 ± 0.008	0.074 ± 0.008
	large μ	0.076 ± 0.007	0.076 ± 0.007	0.079 ± 0.008	0.076 ± 0.007
titanic	small μ	0.224 ± 0.002	0.223 ± 0.002	0.225 ± 0.002	0.226 ± 0.003
	large μ	0.224 ± 0.002	0.223 ± 0.002	0.225 ± 0.002	0.225 ± 0.003

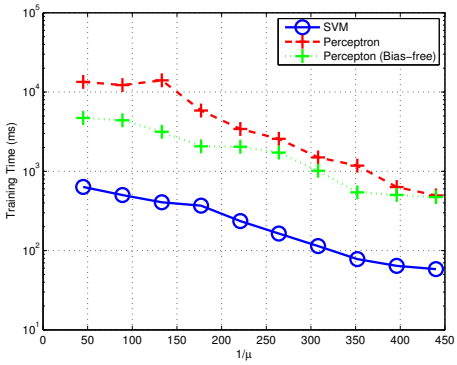
for SVMs than for perceptrons. In addition, each iteration is also faster for SVMs than for perceptrons, since the two-class update step involves fewer computations (as discussed in Section 5.6).

Table 5.11: Test error for SVMs compared to perceptrons using the geometric threshold

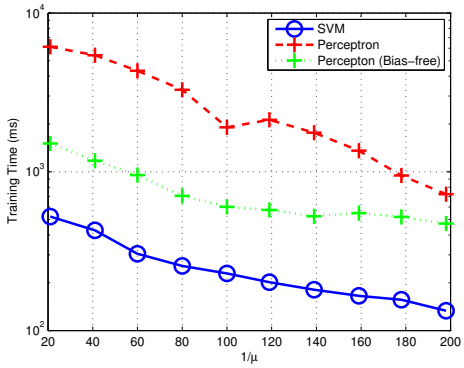
		Gaussian kernel ($\gamma = 0.01$)		**Polynomial kernel** ($q = 3$)	
		SVM	**Perceptron**	**SVM**	**Perceptron**
`banana`	small μ	0.435 ± 0.010	0.436 ± 0.010	0.228 ± 0.006	0.229 ± 0.006
	large μ	0.436 ± 0.010	0.436 ± 0.010	0.226 ± 0.006	0.227 ± 0.006
`b.cancer`	small μ	0.277 ± 0.007	0.277 ± 0.007	0.351 ± 0.007	0.351 ± 0.008
	large μ	0.277 ± 0.007	0.278 ± 0.007	0.356 ± 0.009	0.356 ± 0.010
`diabetes`	small μ	0.241 ± 0.005	0.241 ± 0.005	0.304 ± 0.005	0.304 ± 0.006
	large μ	0.240 ± 0.005	0.239 ± 0.005	0.315 ± 0.006	0.314 ± 0.006
`german`	small μ	0.236 ± 0.005	0.235 ± 0.005	0.300 ± 0.007	0.306 ± 0.007
	large μ	0.234 ± 0.005	0.234 ± 0.005	0.300 ± 0.007	0.306 ± 0.007
`heart`	small μ	0.157 ± 0.008	0.159 ± 0.008	0.229 ± 0.009	0.225 ± 0.008
	large μ	0.159 ± 0.008	0.159 ± 0.007	0.229 ± 0.009	$\mathbf{0.224 \pm 0.008}$
`image`	small μ	0.207 ± 0.004	0.207 ± 0.004	0.037 ± 0.001	0.040 ± 0.002
	large μ	0.162 ± 0.011	0.162 ± 0.010	0.039 ± 0.001	0.047 ± 0.005
`splice`	small μ	0.158 ± 0.002	$\mathbf{0.156 \pm 0.002}$	0.131 ± 0.003	0.136 ± 0.003
	large μ	0.139 ± 0.005	$\mathbf{0.138 \pm 0.004}$	0.131 ± 0.003	0.136 ± 0.003
`thyroid`	small μ	0.133 ± 0.009	0.133 ± 0.009	0.066 ± 0.007	0.068 ± 0.007
	large μ	0.117 ± 0.009	0.119 ± 0.009	0.071 ± 0.006	0.076 ± 0.007
`titanic`	small μ	0.226 ± 0.001	0.226 ± 0.001	0.224 ± 0.002	0.223 ± 0.002
	large μ	0.226 ± 0.000	0.226 ± 0.000	0.224 ± 0.002	0.223 ± 0.002



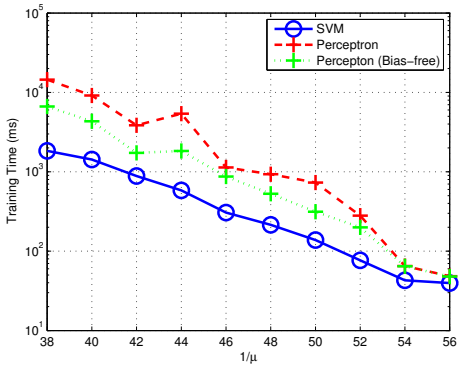
(a) `banana` dataset, Gaussian kernel, $\gamma = 0.1$



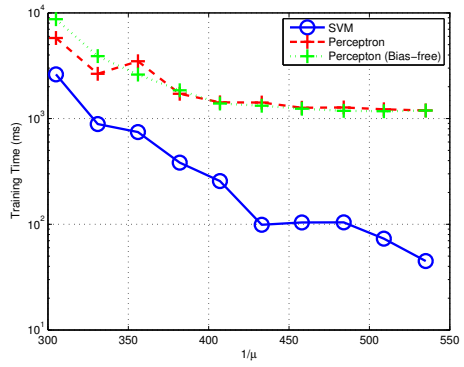
(b) `splice` dataset, Gaussian kernel, $\gamma = 0.01$



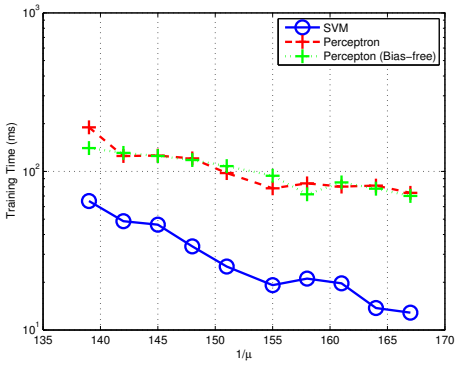
(c) `german` dataset Gaussian kernel, $\gamma = 0.1$



(d) `b.cancer` dataset, Gaussian kernel, $\gamma = 0.01$

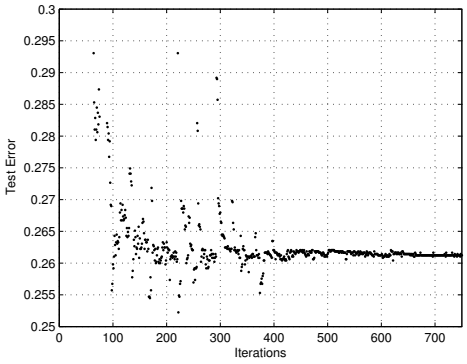


(e) `image` dataset, linear kernel

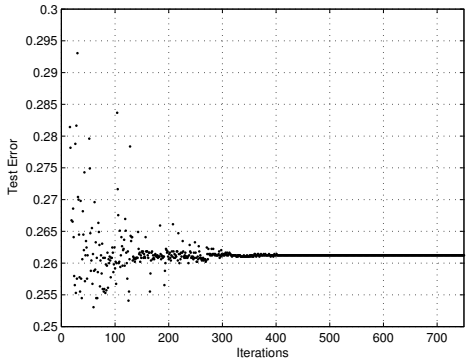


(f) `diabetes` dataset, polynomial kernel, $q = 2$

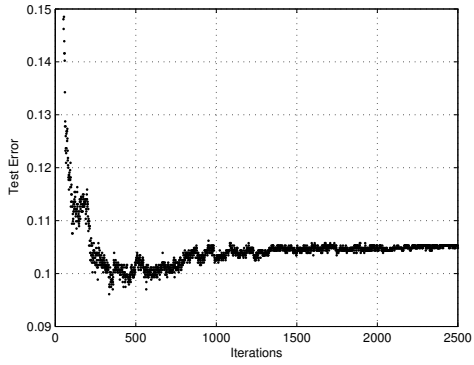
Figure 5.16: Training times for L_1-loss perceptrons compared to L_1-loss SVMs



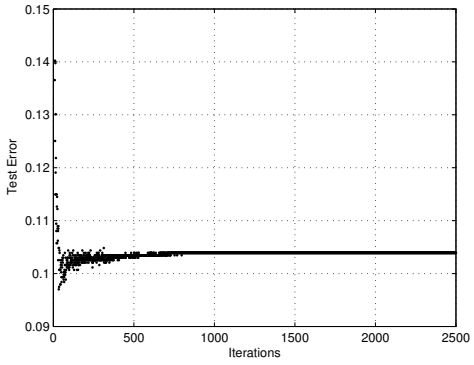
(a) Perceptron on `banana` dataset. $\gamma = 0.1, \mu = 1/200$



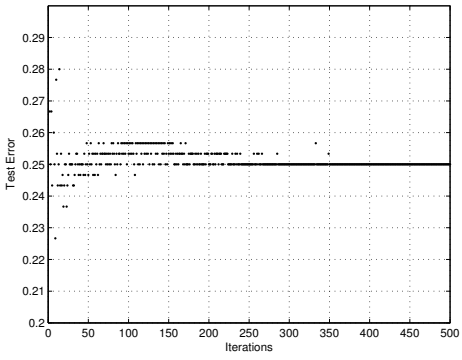
(b) SVM on `banana` dataset. $\gamma = 0.1, \mu = 1/100$



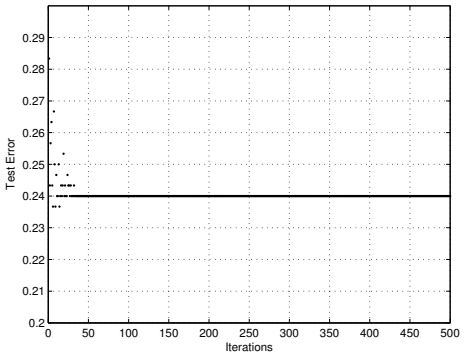
(c) Perceptron on `splice` dataset. $\gamma = 0.01, \mu = 1/60$



(d) SVM on `splice` dataset. $\gamma = 0.01, \mu = 1/30$



(e) Perceptron on `diabetes` dataset. $q = 2, \mu = 1/300$



(f) SVM on `diabetes` dataset. $q = 2, \mu = 1/150$

Figure 5.17: Test error as number of training iterations increases. The Gaussian kernel is used. Notice that the α_i values in the perceptron sum to one, whereas in an SVM they sum to two. This means that the perceptron μ parameter must be given half the value of the SVM μ value in order to have a comparable effect.

5.10 Discussion

For Gaussian and polynomial kernels, the bias-free perceptron was generally an acceptable choice. This can be seen in the accuracies achieved by the bias-free perceptron in Tables 5.9 and 5.10. However, for linear kernels, the bias-free perceptron is not always a viable choice. This is because, for these kernels, a hyperplane forced to pass through the origin simply can not achieve a high classification accuracy. For example, Figure 5.18 depicts a simple dataset where a bias-free perceptron simply can not achieve a high accuracy. For this reason we did not reproduce many of the tables in this section using linear kernels.

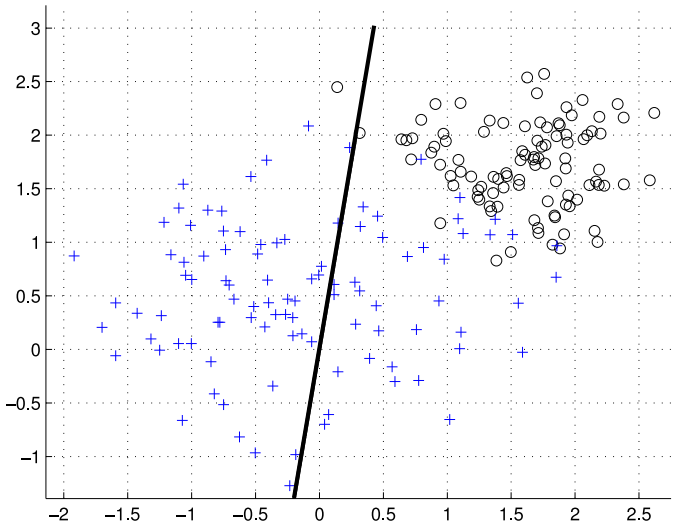


Figure 5.18: The hyperplane found by a bias-free perceptron is constrained to pass through the origin.

It may seem counter-intuitive that the increased simplicity of the one-class minimal norm task should result in a less efficient algorithm than the two-class nearest point task. However, the decrease in efficiency likely arises due to the modified kernel that must be used in conjunction with the minimal norm task. This follows from the enormous influence that the kernel parameters had on training time in Section 5.5.2. It is also supported by Figure 5.17, which shows that the WSK algorithm makes slower progress training perceptrons than it does training SVMs.

Our results on perceptrons are consistent with those of Keerthi et al. [65], who have performed similar comparisons between the SVM and perceptron optimization tasks using the L_2-loss function (as opposed to the L_1-loss function that we use here). Keerthi et al. suggest that it is more efficient to solve the two-class nearest point problem associated with the SVM optimization task that it is to solve the one-class nearest point task associated with perceptrons.

5.11 Conclusions

There are several important conclusions which can be drawn from the work in this chapter. First, we have described how the concept of WRCHs (introduced in Chapter 3) can be used in order to understand how WSVMs work. This has led to the introduction of a weighted variant of the Schlesinger-Kozinec algorithm, which can operate over WRCHs in order to train WSVMs. Empirical trials have demonstrated that the WSK algorithm is equivalent to, but much faster than, point duplication. The WSK algorithm is also more versatile in that it can handle non-integral point weights.

We have also described two improvements to the WSK algorithm which maximize its efficiency. The first improvement was to compute the update step in a way which best exploits the sparsity of the Lagrange multipliers. The second improvement was in ensuring that the nearest points points were pushed towards the facets of the WRCHs as quickly as possible. Beneficially, pushing the points towards the outside of the hull also increased the sparsity of the Lagrange multipliers, increasing the effectiveness of the first improvement. Because the WSK algorithm can train standard SVMs when all weights are equal to one, we have been able to show that these optimizations also improve on current state-of-the-art unweighted nearest point algorithms.

In Section 5.8 we demonstrated that the absolute stopping condition commonly applied to many nearest point algorithms can be inconsistent across differing parameter values and datasets. We have instead recommended a relative stopping condition which we have shown to be more consistent across multiple parameter values and datasets. In this section we also noted that most of the stopping conditions perform more training iterations than are strictly necessary. However, it is difficult to propose more efficient stopping conditions without having prior information on the datasets being trained on or risking a decrease in testing accuracy.

The results in Section 5.8 regarding SVM stopping conditions are interesting in that they suggest an area of further research. Although it is difficult to propose better stopping conditions for the purposes of training an SVM which maximizes test accuracy, there are other circumstances where the aim of the stopping conditions differ. For example, when automatically choosing the parameters of an SVM, the goal is to *compare* two SVMs rather than to simply minimize test error. We explore the possibility of very early termination in model selection in the next chapter.

Chapter 6

Parameter Selection using Geometric Information

6.1 Introduction

This chapter addresses the task of *parameter selection* for SVMs. In previous chapters we have described several parameters inherent in SVMs. The existence of these parameters makes SVMs customizable enough to achieve a state-of-the-art accuracy across a range of application domains [19]. However, it also introduces the task of ensuring the parameters are properly 'tuned' to a particular problem. It is difficult to know in advance which parameters will best suit a particular problem, and an incorrect choice of parameters can have a detrimental effect on the accuracy of the final classifier [33].

The process of choosing the various parameters associated with an SVM automatically is most commonly addressed as a two-part process. First, a method for estimating the generalization accuracy of an SVM must be devised. The generalization accuracy is the expected accuracy of an SVM when classifying data which does not necessarily exist in its training set. Computing such an estimate is more difficult than simply computing the accuracy of an SVM on its training data. The second part of the process is to find the combination of parameters which minimizes the estimate of the generalization accuracy, which in most cases involves the repeated training of SVMs across a range of parameter values.

There are two main contributions in this chapter. First, we use the geometric understanding of SVMs to accelerate the existing method of radius-margin parameter selection. We achieve this by showing that a geometric approach to SVM training allows upper and lower bounds on the radius and margin of an SVM to be computed at any stage *during training*. This allows for parameter values to be compared quickly without necessarily training SVMs to completion. The result is a reduction in the number of training iterations required by a factor of 10 or more.

Our second contribution is to describe how parameter selection can be performed using μ-SVMs. Although parameter selection is most commonly performed using C-SVMs, we suggest that the C-SVM formulation is not necessarily advantageous. We show that existing estimates of the test error designed for C-SVMs may be applied to μ-SVMs. We

also describe how a search for optimal parameters can be performed using μ-SVMs. The advantage of using μ-SVMs is that any regions of parameter space where hulls intersect or are reduced to their centroids may be excluded. This allows for a smooth search over class reduction, from the centroids to the point where the hulls intersect, with clearly defined start and end points. By contrast, searching for the optimal value of C for a C-SVM often requires searching over large ranges such as $\log_2(C) \in \{10, -9, \ldots, 9, 10\}$, to account for the fact that C can take any positive value. For some datasets these values can produce large variations, whereas for others they yield nearly identical SVMs.

6.2 Existing Error Estimates

Estimating the test error (also sometimes referred to as the generalization error [33]) of an SVM is a difficult problem. Test error differs greatly from the error an SVM achieves on the training set, since the training error is extremely vulnerable to overfitting. For example, a Gaussian kernel with a narrow width can often achieve a perfect training error, while test errors will be much greater. In this section we describe several estimates of the test error of an SVM.

In this section we reuse some notation from previous chapters. Namely, we refer to the SVM decision function:

$$f(\mathbf{x}) = \sum_{i=1}^{n} y_i \alpha_i \mathbf{x}_i \cdot \mathbf{x} - b,$$

Refer back to Chapter 2 for a more detailed explanation of all of the terms involved in the SVM decision function.

6.2.1 Hold-out Sets

One of the simplest ways to estimate the error of an SVM is to withhold training data for use as a validation set [34]. This means that an SVM is trained on only a subset of its training data. The SVM is then used to classify the remaining points (referred to as the hold-out set). The error of the SVM on the hold-out set provides an estimate of the generalization error of the SVM.

The main issue associated with the use of a hold-out set is that it takes training data away from the classifier. Because training set size is generally correlated with test accuracy [106], reducing the training set size may cause an overestimate of the test error. Furthermore, if there is only a small amount of training data available, estimates of the test error can become unreliable due to the small number of samples present in the training and validation sets.

6.2.2 Cross-Validation and Leave-One-Out

Cross-validation addresses some of the deficiencies of the hold-out method by performing repeated trials. In k-fold cross-validation, training data is first separated into k disjoint subsets or *folds*. The hold-out procedure is then repeated for each fold. For example, for

$k = 10$, the training data is separated into 10 folds. Initially the first fold is withheld and a classifier is trained on the remaining 9 folds. The accuracy of the classifier on the first fold is then recorded. The second fold is then withheld and the process is repeated. The average error across all of the folds provides an estimate of the generalization error of the classification technique [34].

When k-fold cross-validation is performed with k equal to the number of points, the result is the leave-one-out estimate of the test error. The leave-one-out error is the most unbiased (but also the most computationally expensive) form of cross-validation [95]. The process of computing the leave-one-out error of an SVM may be accelerated by exploiting the fact that non-support vectors in the training data are guaranteed to be correctly classified by the leave-one-out procedure and therefore do not require the training of a new machine [117].

The cross-validation and leave-one-out procedures tend to be some of the more accurate estimates of the test error of a classifier [33, 16]. However, they are also some of the more computationally expensive estimates of the test error. Because a large number of SVMs must be trained in order to give a single estimate of the test error, these procedures may become computationally infeasible on larger datasets.

6.2.3 Support Vector Bound

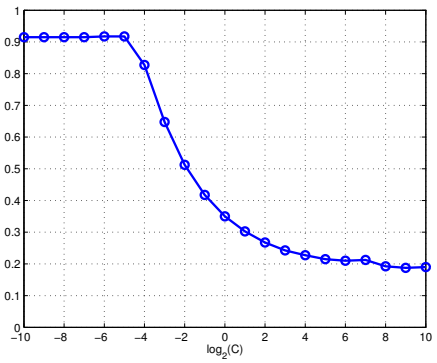
The support vector bound is one of the simplest upper bounds on the leave-one-out error. For an SVM trained on n training points, the support vector bound states that the leave-one-out error is bounded by [117]:

$$E_{\text{LOO}} \leq \frac{n_s}{n}, \tag{6.1}$$

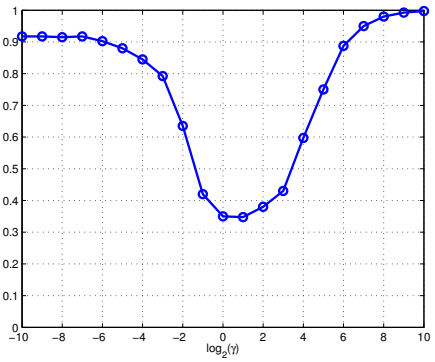
where n_s is the number of support vectors.

The number of support vectors must form an upper bound on the leave-one-out error of an SVM because of the sparseness property, which we described in Section 2.2.1. Recall that if a non-support vector is removed from a training set, either before or after training an SVM, it has no impact on the decision surface. Non-support vectors also must always lie on the correct side of the supporting planes of an SVM. It follows that any non-support vector would be correctly classified using the leave-one-out procedure, providing the upper bound in Equation (6.1).

The main disadvantage of the support vector bound is that it counts all support vectors as being incorrectly classified by the leave-one-out procedure, even though this may not be the case. This means the bound becomes larger as the number of support vectors increases. However, it is important to note that as the regularization parameter (C or μ) increases, the number of support vectors will generally decrease. This means that the support vector bound will generally choose the largest possible value of C or μ (Figure 6.1). Because of this trivial outcome, E_{LOO} is, to the best of our knowledge, not widely used in parameter selection.



(a) $\gamma = 1$, C varied



(b) $C = 1$, γ varied

Figure 6.1: The support vector bound on the `banana` dataset. The Gaussian kernel is used.

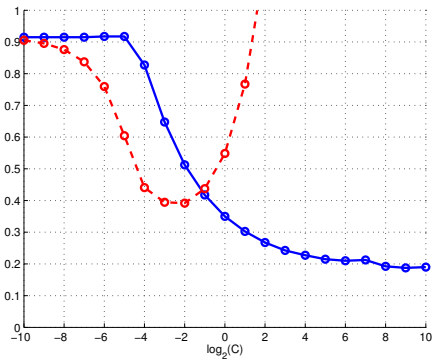
6.2.4 Generalized Approximate Cross-Validation

Generalized Approximate Cross-Validation (GACV) is given for the C-SVM by [122, 123, 75, 46, 33]:

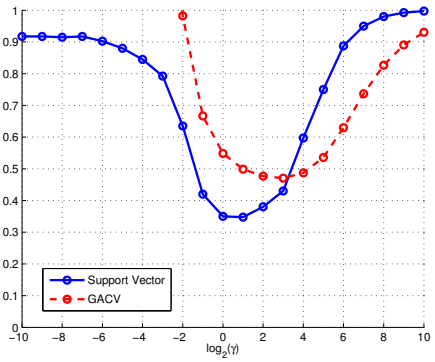
$$E_{\text{GACV}} = \frac{1}{n}\left[\sum_{i=1}^{n}\xi_i + \sum_{y_i f(\mathbf{x}_i)<-1} 2\alpha_i \mathbf{x}_i \cdot \mathbf{x}_i + \sum_{y_i f(\mathbf{x}_i)\in[-1,1]} \alpha_i \mathbf{x}_i \cdot \mathbf{x}_i\right]. \tag{6.2}$$

Here ξ_i's are slack variables, computed by solving the C-SVM optimization task from Section 2.4.1. GACV was proposed by Wahba [122] as a computable proxy for the Generalized Comparative Kullback-Leibler distance, which is itself an upper bound on the generalization error of an SVM.

Equation (6.2) uses the C-SVM parameterization of GACV given by Duan et al. [33]. We use this parameterization since Wahba originally proposed GACV for an SVM with an alternative parameterization. This raises an important point regarding error estimates: if the parameterization of an SVM is changed (e.g. from a C-SVM to a μ-SVM), this parameterization also needs to be reflected in the error estimate.



(a) $\gamma = 1$, C varied



(b) $C = 1$, γ varied

Figure 6.2: The GACV error estimate compared to the support vector bound on the `banana` dataset. The Gaussian kernel is used.

Figure 6.2 compares GACV to the support vector bound. Notice how, unlike the support vector bound, GACV does not always prefer larger C values. This is because increases to C increase the upper bound on α_i values, which can in turn increase (6.2). Notice also how GACV applies an additional penalty to points for which $y_i f(\mathbf{x}_i) < -1$, and sometimes reaches 1 (i.e. estimates a 100% error rate). Because of this, GACV is considered only an approximate upper bound. Rather than tightly bounding the leave-one-out error, it instead aims to best reflect the optimal parameters at its minimum [74].

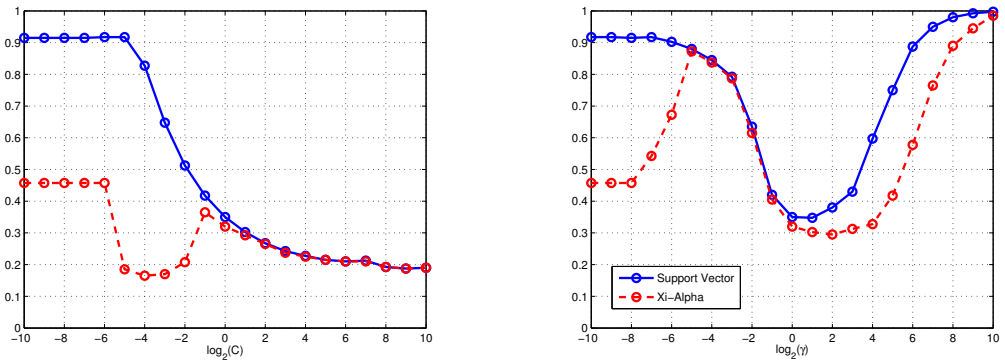
6.2.5 Xi-Alpha Error Estimate

Joachims [59] defines the $\xi\alpha$-estimator of the generalization error of an L_1-loss C-SVM as

$$E_{\xi\alpha} = \frac{m}{n} \qquad \text{with} \quad m = \operatorname{card}\left(\left\{i \mid \left(2\alpha_i R_\Delta^2 + \xi_i\right) \geq 1\right\}\right).$$

Here R_Δ^2 is an upper bound on $\mathbf{x}_i \cdot \mathbf{x}_i - \mathbf{x}_i \cdot \mathbf{x}_j$, ξ_i and α_i values are the solutions to the SVM optimization task, meaning an SVM must be trained for each set of parameters which is to be compared using the $\xi\alpha$-estimator.

The $\xi\alpha$-estimator is closely related to the support vector leave-one-out bound, described in Section 6.2.3. For hard margin SVMs (or C-SVMs with large C), the two bounds are likely to provide quite similar estimates, since almost all support vectors are likely to satisfy $2\alpha_i R_\Delta^2 \geq 1$ when α_i values are large (Figure 6.3). However, the advantage of the $\xi\alpha$-estimator is that it provides a tighter bound for smaller values of C. The increased tightness arises from taking into account the fact that, in a soft margin SVM, points can be support vectors while lying quite far from the decision surface. These points will not necessarily be incorrectly classified if left out from training.



(a) $\gamma = 1$, C varied **(b)** $C = 1$, γ varied

Figure 6.3: The Xi-Alpha bound compared to the support vector bound on the `banana` dataset. The Gaussian kernel is used. Both figures share the same legend.

6.2.6 Radius-Margin Ratio

Radius-margin parameter selection [117, 63, 33] is performed by searching across potential parameter values in order to find a set that minimizes the radius-margin ratio R^2/Δ^2. Here

R is the radius of the MEB enclosing the training data (discussed previously in Section 2.7), and Δ is the margin of the SVM. Computing these values generally requires solving two QP problems: the SVM QP task and the MEB QP task. One justification for this approach is given by the radius-margin bound, an upper bound on the leave-one-out error of an SVM, written [21]:

$$E_{\text{LOO}} \leq \frac{R^2}{\Delta^2 n}. \tag{6.3}$$

This means minimizing the radius-margin ratio minimizes an upper bound on the leave-one-out error of an SVM.

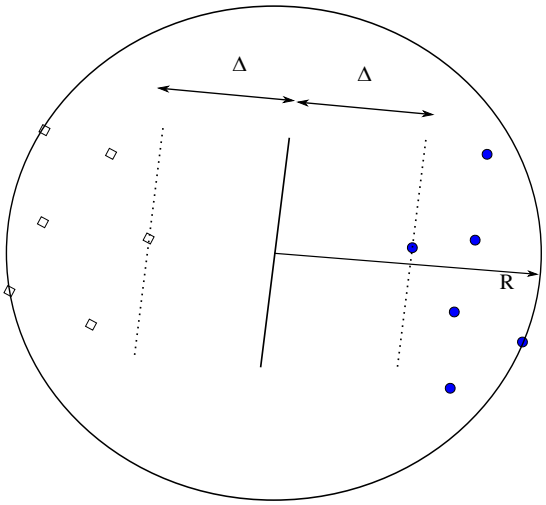
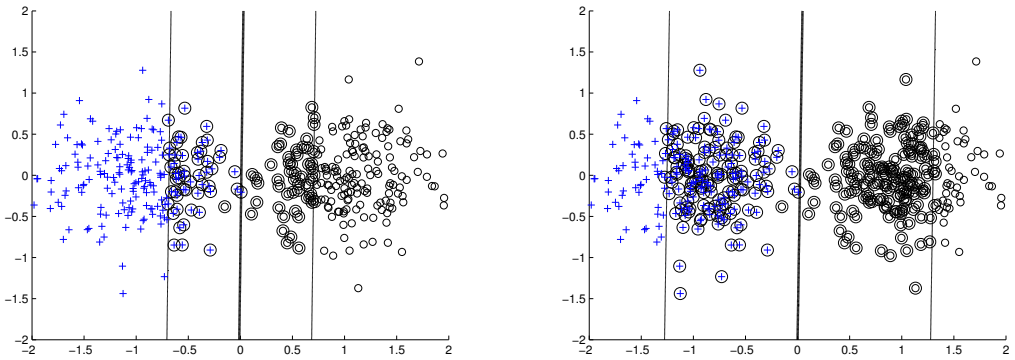


Figure 6.4: The radius R of the MEB enclosing the training data and the margin Δ of an SVM

The radius-margin technique tends to provide reasonable parameter values, *provided* it is applied to an SVM with a hard margin interpretation [33]. For an example of why the radius-margin technique is inappropriate in conjunction with an L_1-loss SVM, consider the `toy` dataset depicted in Figure 6.5. In this case the parameter μ is varied, with each reduction in μ resulting in an increase in the margin. However, this increase in margin width is not offset by any change to the MEB of the data. This means the radius-margin technique will always choose the smallest possible μ value, resulting in at least one of the classes being reduced to its mean. Reducing all of the class information down to a single point is clearly an oversimplification which will underfit most problems.

Although a hard margin SVM is required in order to use the radius-margin technique, recall that the L_2-loss SVM has a hard margin *interpretation* in feature space (refer to Section 2.4.2). This means the radius-margin technique can be used in conjunction with an L_2-loss SVM in order to choose both the regularization parameter and any kernel parameters. The key is that the regularization parameter needs to be applied to the kernel, and that same kernel should be applied in order to compute the radius of the MEB of the training data. This ensures that changes to the regularization parameter impact both the margin of the SVM and the radius of the MEB.



(a) Linear L_1-loss μ-SVM with $\mu = 1/50$ **(b)** Linear L_1-loss μ-SVM with $\mu = 1/150$

Figure 6.5: The L_1-loss SVM can arbitrarily increase the margin by varying μ. Here support vectors are circled.

6.3 Accelerating Radius-Margin Parameter Selection

A previously used method of parameter selection for SVMs which we described in Section 6.2 is the radius-margin technique [117, 33, 63]. Using the radius-margin technique, parameter values are chosen so that they minimize the ratio of the radius of the Minimal Enclosing Ball (MEB) of the training data compared to the margin of the SVM (Figure 6.4). Computing the radius-margin ratio of an SVM requires the solution of two Quadratic Programming (QP) problems, and this is generally repeated across a range of parameter values. The radius-margin approach is justified by the observation that the squared radius-margin ratio forms an upper bound on the leave-one-out error of an SVM [21].

By performing parameter selection in conjunction with the geometric formulation of the L_2-loss SVM optimization problem (which we reviewed in detail in Section 2.5.4), we show in this section that upper and lower bounds on both the margin of an SVM and the radius of an MEB can be calculated at any stage *during training*. These bounds converge on the exact radius and margin as the number of training iterations increases. When used in conjunction with existing training methods, bounds are available at almost no additional computational cost by taking advantage of values that are already computed during training. We focus specifically on the L_2-loss SVM because it can be formulated as a hard margin SVM in a modified kernel feature space. Recall from Section 6.2.6 that a hard margin interpretation is a requirement of the radius-margin bound.

The ability to compute upper and lower bounds on the margin of an SVM and the radius of an MEB allows us to compute upper and lower bounds on the radius-margin ratio of an SVM during training. This allows two SVMs to be compared very efficiently by training them only until the upper and lower bounds of the two SVMs no longer overlap. This allows extremely early termination of SVM training, while still obtaining a guarantee that a particular set of parameters minimize the radius-margin ratio. We are further able to reduce the number of training iterations required by using previous and

partial solutions in order to evaluate and possibly reject neighboring parameter values with little or no additional training.

Empirical trials demonstrate that, when used in conjunction with a simple grid search, our technique can reduce the number of training iterations required by a factor of 10 or more compared to similar methods.

6.3.1 The Geometric L_2-loss SVM

Recall from Section 2.5.4 that the geometric parameterization of the L_2-loss SVM is given by:

$$\begin{aligned} \min_{\alpha} \quad & \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \\ \text{subject to} \quad & \begin{cases} \sum_{i=1}^{n} \alpha_i = 2, \\ 0 \leq \alpha_i \leq 1, \\ i = 1, \ldots, n. \end{cases} \end{aligned}$$

Here $k(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij}/(2C)$. Conveniently, this means that the regularization parameter C, along with any kernel parameters, are encapsulated within the kernel k. It is this property that makes the the L_2-loss SVM equivalent to finding the nearest points in two convex hulls (i.e. a hard margin SVM). This in turn enables the application of the radius-margin bound.

6.3.2 Bounding the Margin During Training

Both an upper and a lower bound on the margin of an SVM can be calculated at any stage during training, provided the constraints on the dual are met. These bounds become increasingly tight as training continues, and will eventually coincide with the margin if an exact solution is reached. The upper bound on the margin can be computed based on the distance between the two current nearest points,

$$(\Delta^{\text{up}})^2 = \frac{1}{4} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j). \tag{6.4}$$

When the constraints of the dual optimization problem are met, the two current nearest points will always belong to the two convex hulls, meaning this quantity must bound the margin from above.

The lower bound is given by the vector joining the two inner-most points in the two classes, projected onto the current hyperplane normal

$$\Delta^{\text{low}} = \frac{1}{4\Delta^{\text{up}}} \left(\min_{k \in I_{pos}} \left\{ \sum_{i=1}^{n} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}_k) \right\} - \max_{k \in I_{neg}} \left\{ \sum_{i=1}^{n} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}_k) \right\} \right).$$

This quantity must bound the margin from below since it is at a maximum when it is equal to the true margin, which can only occur when the KKT conditions are satisfied precisely.

Notice that it may be possible for Δ^{low} to become less than zero, so it is preferable to use $\max(0, \Delta^{\text{low}})$ in practice. A graphical representation of Δ^{up} and Δ^{low} is given in Figure 6.6.

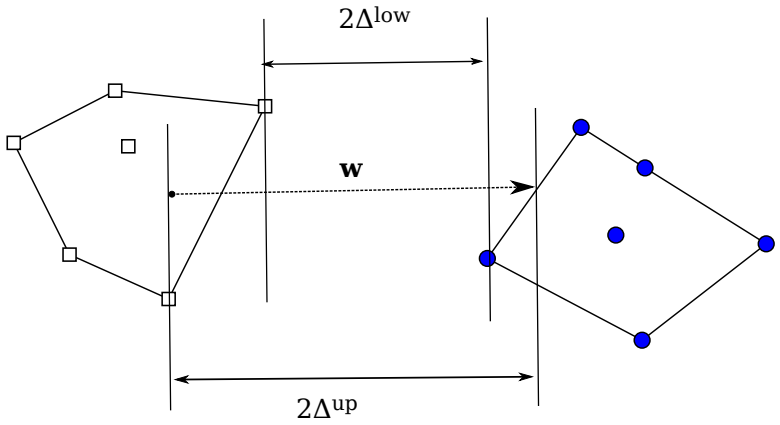


Figure 6.6: Lower and upper bounds on the margin of a partially trained SVM. The vector $\mathbf{w}$ connects the two current nearest points.

6.3.3 Bounding the Radius of the MEB During Training

Upper and lower bounds can also be computed for the radius of an MEB. Provided the dual constraints are met, at any stage during training, the MEB dual objective function must provide a lower bound on the radius of the MEB:

$$(R^{\text{low}})^2 = \sum_{i=1}^{n} \beta_i k(\mathbf{x}_i, \mathbf{x}_i) - \sum_{i,j=1}^{n} \beta_i \beta_j k(\mathbf{x}_i, \mathbf{x}_j).$$

Here β_i's are the Lagrange multipliers associated with the MEB dual optimization task. We denote these as β_i to avoid confusion with the Lagrange multipliers associated with the SVM dual, which remain α_i.

An upper bound on the radius of the MEB may be given by computing the smallest radius enclosing all training data based on the current (non-optimal) center $\mathbf{c}$ of the MEB,

$$\begin{aligned}(R^{\text{up}})^2 &= \max_k \left\{ ||\mathbf{x}_k - \mathbf{c}||^2 \right\} \\ &= \max_k \left\{ k(\mathbf{x}_k, \mathbf{x}_k) - 2\sum_{i=1}^{n} \beta_i k(\mathbf{x}_i, \mathbf{x}_k) + \sum_{i,j=1}^{n} \beta_i \beta_j k(\mathbf{x}_i, \mathbf{x}_j) \right\}\end{aligned}$$

This bound can only equal the true radius if the KKT conditions of the MEB dual are satisfied (i.e. if $\mathbf{c}$ is the true MEB center). Otherwise, it will bound the radius from above, with the tightness depending on how much progress has been achieved towards computing $\mathbf{c}$.

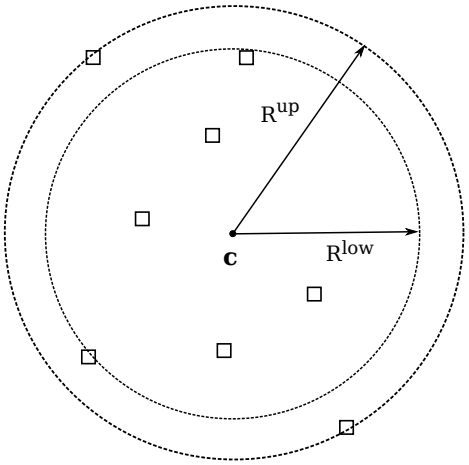


Figure 6.7: Lower and upper bounds on the MEB of the training data

6.3.4 Bounding the Radius-Margin Ratio During Training

Using the previously stated bounds on the radius and margin, upper and lower bounds on the radius-margin ratio can be written

$$T^{\text{up}} = \frac{R^{\text{up}}}{\Delta^{\text{low}}} \qquad\qquad T^{\text{low}} = \frac{R^{\text{low}}}{\Delta^{\text{up}}} \tag{6.5}$$

These bounds can be computed at any stage during training for a given MEB and SVM.

6.3.5 Efficient Computation of Bounds During Training

In order to solve the L_2-loss SVM problem we apply Platt's Sequential Minimal Optimization (SMO) [91] using the maximal violating pairs heuristic over the entire training set. This variant of SMO has previously been used by López et al. [77], who note that it is equivalent to the MDM algorithm. The algorithm begins by initializing one (arbitrarily chosen) Lagrange multiplier in each class to equal one. This ensures the constraints $\sum_i \alpha_i = 2$ and $\sum_i \alpha_i y_i = 0$ are satisfied. In order to make progress, two points are chosen from which to transfer weight between. The source and destination indices are given by:

$$\left.\begin{aligned} src &= \underset{k \in I_{pos}, \alpha_k > 0}{\arg\max} \{\mathbf{w} \cdot \mathbf{x}_k\} \\ dst &= \underset{k \in I_{pos}, \alpha_k < 1}{\arg\max} \{-\mathbf{w} \cdot \mathbf{x}_k\} \end{aligned}\right\} \text{ positive class}$$

$$\left.\begin{aligned} src &= \underset{k \in I_{neg}, \alpha_k > 0}{\arg\max} \{-\mathbf{w} \cdot \mathbf{x}_k\} \\ dst &= \underset{k \in I_{neg}, \alpha_k < 1}{\arg\max} \{\mathbf{w} \cdot \mathbf{x}_k\} \end{aligned}\right\} \text{ negative class}$$

The way in which the *dst* indices are calculated is very convenient for our purposes, since these are the same indices required to compute the lower bound on the margin of an SVM during training. This means that the bound is available at any stage during training at barely any extra computational cost. Also convenient is the fact that the sum of Lagrange multipliers in each class is always equal to one, meaning bounds are

computable after any iteration. Although this may not be the fastest possible approach to training [77], it allows for such simple calculation of margin bounds that we use it in preference to alternate approaches. We discuss the possibility of using alternative training algorithms in Section 6.3.9.

Since Δ^{up} is so closely related to the dual objective function, and because only two Lagrange multipliers are modified per iteration, it is also quite easy to cache and update Δ^{up} after each training iteration. This means that it is not necessary to compute the entire sum in Equation (6.4) at each iteration. Rather, the bound may be computed incrementally with a small update each iteration. This means that both upper and lower bounds are available with a minimal amount of computational effort.

SMO with a maximal violating pairs heuristic can also be adapted in order to solve the MEB QP problem. We begin by initializing one arbitrarily chosen Lagrange multiplier to equal one in order to satisfy the constraint $\sum_i \beta_i = 1$. Weight is then shifted in order to optimize the dual objective function with respect to two Lagrange multipliers at a time, while ensuring that at no stage the dual constraints are violated.

The source and destination indices used in the weight transfer are given by:

$$src = \arg\min_{k,\beta_k>0}\{||\mathbf{x}_k - \mathbf{c}||\}$$
$$dst = \arg\max_{k,\beta_k<1}\{||\mathbf{x}_k - \mathbf{c}||\}$$

Notice that, as in the case of the SVM dual, this is a convenient way to solve the MEB dual since the index *dst* is the same index used to compute the upper bound on the radius of the MEB.

6.3.6 Efficiently Comparing SVMs in Terms of Radius-Margin Ratio

The bounds on the radius-margin ratio presented in Section 6.3.4 enable a simple and fast way of comparing two SVMs. Given two SVMs and two corresponding MEBs after any number of training iterations, we can construct the bounds $T_1^{\text{up}}, T_1^{\text{low}}, T_2^{\text{up}}, T_2^{\text{low}}$ using Equation (6.5). Bounds T_1^{up} and T_1^{low} refer to the first SVM, whereas T_2^{up} and T_2^{low} refer to the second SVM. If at any stage during training:

$$T_1^{\text{up}} < T_2^{\text{low}},$$

it is guaranteed that the first SVM has a smaller radius-margin ratio and there is no need to continue training the SVMs or MEBs any further. Similarly, the condition:

$$T_2^{\text{up}} < T_1^{\text{low}}$$

indicates that the second SVM has a smaller radius-margin ratio and training can be stopped.

During training, the bounds we have described can also provide a heuristic for where to focus most of the training effort. Given two SVMs and two MEBs, the inequality:

$$\frac{T_1^{\text{up}}}{T_1^{\text{low}}} > \frac{T_2^{\text{up}}}{T_2^{\text{low}}} \tag{6.6}$$

suggests that the first SVM and/or MEB is in a more 'untrained' state than the second and that focusing training effort on them is likely to provide the greatest overall improvement in the tightness of all of the bounds.

If Equation (6.6) indicates that the first SVM/MEB should be the focus of training, we can then decide whether to focus on the SVM or the MEB by checking the inequality:

$$\frac{R_1^{\text{up}}}{R_1^{\text{low}}} > \frac{\Delta_1^{\text{up}}}{\Delta_1^{\text{low}}}. \tag{6.7}$$

If this inequality is met, then training effort should be focused on the MEB. Otherwise, effort should be focused on the SVM. The complete comparison process is described in Algorithm 12

Algorithm 12 Comparing two SVMs (and two associated MEBs) and choosing the one which minimizes the radius-margin ratio

```
function COMPARE(svm1, meb1, svm2, meb2)
    while true do
        get T1^low, T1^up, R1^low, R1^up, Δ1^low, Δ1^up from svm1 and meb1
        get T2^low, T2^up, R2^low, R2^up, Δ2^low, Δ2^up from svm2 and meb2
        if T1^up < T2^low then
            return (svm1, meb1)          ▷ svm1 must have lower radius/margin ratio
        else if T2^up < T1^low then
            return (svm2, meb2)          ▷ svm2 must have lower radius/margin ratio
        end if
        if T1^up/T1^low ≥ T2^up/T2^low then      ▷ check where most progress can be made
            if Δ1^up/Δ1^low ≥ R1^up/R1^low then
                svm1 = iterate_svm(svm1)
            else
                meb1 = iterate_meb(meb1)
            end if
        else
            if Δ2^up/Δ2^low ≥ R2^up/R2^low then
                svm2 = iterate_svm(svm2)
            else
                meb2 = iterate_meb(meb2)
            end if
        end if
    end while
end function
```

Figure 6.8 shows Algorithm 12 operating on two SVMs and two MEBs. The two filled regions show the possible range of values for the radius-margin ratio for two SVMs as the algorithm progresses. The solid vertical line marks the number of iterations after

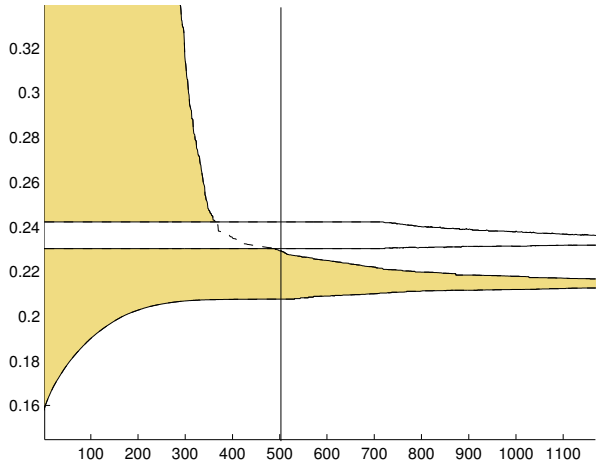


Figure 6.8: Progress of Algorithm 12 for two SVMs and two MEBs on the `german` dataset

which one of the SVMs is guaranteed to have a lower radius-margin ratio. This line is the threshold value at which the algorithm would terminate, however we show future progress in order to demonstrate that it is a waste of iterations. Notice that the most training effort is applied to the SVM with the greatest difference between the upper and lower bounds.

6.3.7 Efficiently Minimizing the Radius-Margin Ratio

Once two SVMs can be compared quickly, the technique can be expanded in order to search across a range of parameter values in order to find those that are optimal. In order to find parameters minimizing the radius-margin ratio, we perform a grid search. This process is described in Algorithm 13. This algorithm exploits the fast comparison algorithm we described in the previous section (Algorithm 12) in order to perform an accelerated grid search. We also ensure we adapt previous solutions when parameters change rather than computing a new machine from scratch. This technique for adapting previous solutions was originally employed by Cristianini et al. [30], who note that using SVMs with nearby parameters to seed new SVM solutions can save significant computational effort.

For an initial guess to supply to Algorithm 13, we use $C = 1$, and the grid point closest to $\gamma = 1/n_d$, where n_d is the dimensionality of the training data. These are commonly used parameter values when a crude estimate of the optimal parameters is desired without performing extensive parameter selection [120, 90]. The algorithm loops over γ in the outside loop and C in the inside loop because we found that this was the more efficient ordering. This likely indicates that adapting a solution after C has been changed requires fewer iterations than adapting a solution after γ has been changed.

6.3.8 Empirical Trials

This section describes empirical trials demonstrating the effectiveness of the accelerated radius-margin parameter selection described in previous sections. In our experiments we

Algorithm 13 Searching across a grid of SVM parameters for those which minimize the radius-margin ratio

```
function MODELSEL_GRID(guesssvm, guessmeb)
    bestsvm = guesssvm;
    bestmeb = guessmeb;
    for γ = 2^-10, 2^-9, ..., 2^10 do
        for C = 2^-10, 2^-9, ..., 2^10 do
            if parameters match initial guess then
                continue
            else
                adapt svm and meb to new parameters γ and C
                [bestsvm bestmeb] = compare(bestsvm, bestmeb, svm, meb)
            end if
        end for
    end for
end function
```

use two kernels; the Gaussian kernel, given by

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\gamma||\mathbf{x} - \mathbf{y}||^2\right),$$

and the polynomial kernel, written

$$K(\mathbf{x}, \mathbf{y}) = \left(\frac{\mathbf{x} \cdot \mathbf{y}}{\tau} + 1\right)^q.$$

In order to avoid having extremely large values across the diagonals of the kernel matrix (which would reduce the impact of the regularization parameter C), we use $\tau = \max_i \left\{\mathbf{x}_i^2\right\}$

By measuring efficiency in terms of the number of SMO training iterations, we compare the efficiency of our method to that of two other methods which use radius margin bounds in parameter selection. The first method we compare with is *standard* radius-margin parameter selection, where an SVM and MEB is trained from scratch for each set of parameters. The second method we compare to is Cristianini's method [30] of using previous nearby SVM and MEB solutions as seeds for new solutions.

Until later sections we use relative KKT stopping conditions (which we described in detail in Section 5.8) with $\epsilon = 10^{-3}$ for all methods unless specified otherwise. To ensure a fair comparison, we also use these stopping conditions in conjunction with our method, rather than potentially allowing Algorithm 12 to continue to an arbitrary precision. In empirical trials we refer to our method as *adaptive* since it adapts stopping conditions based on the bounds it computes during training. Searches are performed over the range $C \in [2^{-10}, 2^{-9}, \ldots, 2^{10}]$, with width $\gamma \in [2^{-10}, 2^{-9}, \ldots, 2^{10}]$ for the Gaussian kernel or degree $q = 2, 3, \ldots, 8$ for the polynomial kernel.

The datasets we use for empirical trials were originally collected from the UCI, DELVE and STATLOG repositories by Rätsch et al. [96]. Trials consist of repeated test runs, with around 60% of data used for training and 40% for testing. Attribute values have been scaled to have zero mean and unit standard deviation. These train-test splits and scalings were originally performed by Rätsch et al., and are now used in other studies, so we use

the same datasets for consistency. Refer to Appendix A for a more extensive overview of these datasets.

Search

Algorithm 13 is basically a grid search, but because it is paired with efficient methods of comparison and adaptation, it performs very well. Figure 6.9 demonstrates the application of the algorithm in order to choose the parameter C and the parameter γ for the Gaussian kernel on several datasets. This figure also shows the number of iterations required for standard radius-margin parameter selection and Cristianini's method.

Notice that, for our adaptive method, iterations tend to be concentrated around the optimal parameter. Most other parameters can be rejected almost immediately. Such fast rejection occurs because previous solutions provide good enough bounds on the radius and margin in order for nearby solutions to be rejected without further training. This avoids the large computational effort required to train Gaussian SVMs with large C values and small γ values. On the datasets tested our method is the only one that does not focus a disproportionate number of training iterations on the section of the search space which combine these extreme values of γ and C.

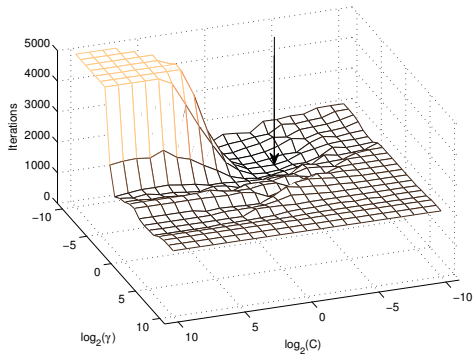
Gaussian Kernel

For the following empirical trials we compare our method to Cristianini's method. We do not compare to standard radius-margin parameter selection since standard radius-margin parameter selection is outperformed so significantly by Cristianini's method. Comparing our technique to Cristianini's is also a much fairer comparison since both methods use previous solutions to seed new solutions. We perform comparisons using both tight ($\epsilon = 10^{-3}$) and loose ($\epsilon = 10^{-1}$) stopping conditions for Cristianini's method in order to verify that our improvements can not be replicated by simply having looser, fixed, stopping conditions.

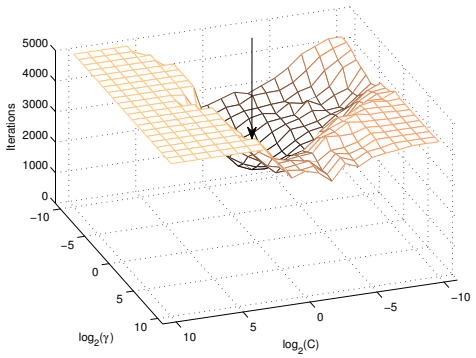
Table 6.1: Average number of SMO iterations required for each parameter selection technique. The Gaussian kernel is used.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	ϵ =adaptive
`banana`	3735.4 ± 116.0	704.3 ± 18.3	$\mathbf{13.7 \pm 0.5}$
`b.cancer`	1200.6 ± 40.4	276.7 ± 5.1	$\mathbf{13.3 \pm 0.1}$
`diabetes`	2024.0 ± 22.1	415.6 ± 4.1	$\mathbf{28.6 \pm 0.4}$
`german`	1616.7 ± 15.4	334.8 ± 4.0	$\mathbf{42.3 \pm 0.7}$
`heart`	226.0 ± 7.5	46.8 ± 1.6	$\mathbf{7.8 \pm 0.2}$
`image`	3172.0 ± 30.0	675.9 ± 6.9	$\mathbf{47.3 \pm 0.8}$
`splice`	941.1 ± 111.5	176.6 ± 39.2	$\mathbf{12.0 \pm 0.2}$
`thyroid`	155.1 ± 3.1	33.0 ± 0.7	$\mathbf{3.2 \pm 0.1}$
`titanic`	935.8 ± 63.5	68.4 ± 6.0	$\mathbf{17.0 \pm 0.6}$

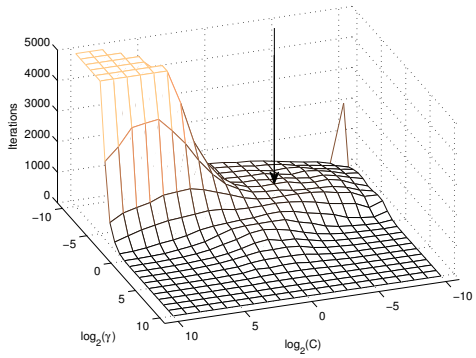
Results for Gaussian kernels are shown in Tables 6.1 and 6.2, where the number of iterations given is the average per grid point over the 21×21 grid of parameter values. The total number of iterations (combined SVM+MEB) for our adaptive method was



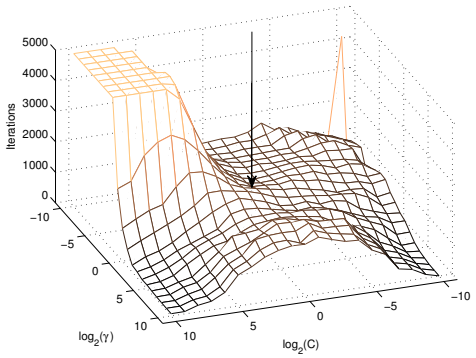
(a) Standard method on `german` dataset. Total $1.0e+6$ iterations.



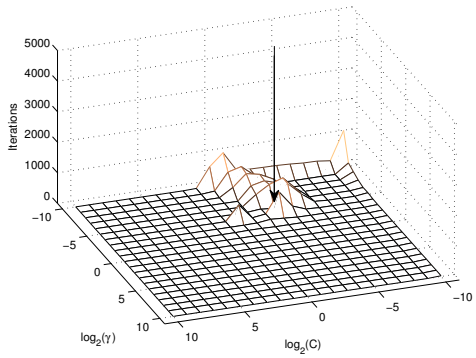
(b) Standard method on `image` dataset. Total $1.7e+6$ iterations.



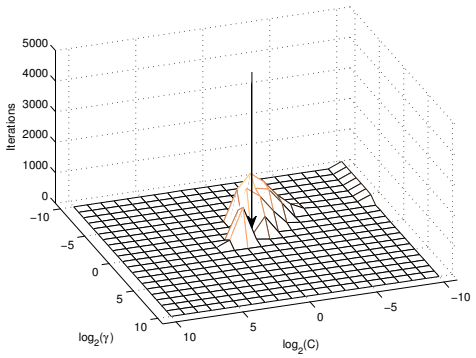
(c) Cristianini's method on `german` dataset. Total $3.3e+5$ iterations.



(d) Cristianini's method on `image` dataset. Total $7.8e+5$ iterations.



(e) Adaptive method on `german` dataset. Total $1.7e+4$ iterations.



(f) Adaptive method on `image` dataset. Total $1.7e+4$ iterations.

Figure 6.9: Adaptive method compared to standard radius-margin parameter selection and Cristianini's method. Note that we have clamped the number of iterations per grid point to a maximum of 5000 in order to avoid obscuring details in other regions of the plot. Final parameters, which differ across datasets but not across methods in these instances, are marked with arrows.

significantly lower across all datasets, by around a factor of 10 compared with stopping conditions of $\epsilon = 10^{-1}$, and by a factor of 100 or more compared with stopping conditions

Table 6.2: Average number of MEB iterations required for each parameter selection technique. The Gaussian kernel is used.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	ϵ =adaptive
banana	56.5 ± 0.5	$\mathbf{1.5 \pm 0.0}$	3.0 ± 0.1
b.cancer	27.5 ± 0.1	$\mathbf{1.2 \pm 0.0}$	4.6 ± 0.1
diabetes	41.0 ± 0.1	$\mathbf{1.2 \pm 0.0}$	7.4 ± 0.1
german	40.4 ± 0.1	$\mathbf{1.1 \pm 0.0}$	10.6 ± 0.2
heart	19.2 ± 0.1	$\mathbf{1.2 \pm 0.0}$	2.5 ± 0.1
image	78.7 ± 0.2	$\mathbf{1.1 \pm 0.0}$	5.2 ± 0.3
splice	38.0 ± 0.0	$\mathbf{1.0 \pm 0.0}$	1.8 ± 0.1
thyroid	28.5 ± 0.2	$\mathbf{1.4 \pm 0.0}$	1.3 ± 0.0
titanic	38.0 ± 0.4	$\mathbf{1.1 \pm 0.0}$	5.1 ± 0.1

of $\epsilon = 10^{-3}$. The number of MEB iterations was generally lowest when a fixed tolerance parameter of $\epsilon = 10^{-1}$ was used, however the dominant cost in all cases was the number of SVM iterations taken.

Table 6.3: Average test errors for each parameter selection technique. The Gaussian kernel is used.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	ϵ =adaptive
banana	0.106 ± 0.002	0.106 ± 0.002	0.106 ± 0.002
b.cancer	0.291 ± 0.016	0.291 ± 0.016	0.291 ± 0.016
diabetes	0.231 ± 0.006	0.232 ± 0.006	0.231 ± 0.006
german	0.252 ± 0.008	0.261 ± 0.009	0.252 ± 0.008
heart	0.174 ± 0.010	0.169 ± 0.010	0.174 ± 0.010
image	0.038 ± 0.002	0.038 ± 0.002	0.038 ± 0.002
splice	0.110 ± 0.003	0.113 ± 0.002	0.110 ± 0.003
thyroid	0.045 ± 0.008	0.045 ± 0.008	0.044 ± 0.008
titanic	0.228 ± 0.001	0.239 ± 0.009	0.228 ± 0.001

Average test errors associated with the parameters chosen by each method are shown in Table 6.3. In order to calculate test error, parameter selection was performed on the training set only, with the error measured on the test set only. This was repeated 10 times with random train-test splits used each time. The error shown in Table 6.3 is the $mean \pm stderr$ proportion of test errors over the 10 runs.

These results highlight the benefits of using adaptive stopping conditions. Not only does our method require fewer training iterations than Cristianini's method in conjunction with even the loosest of stopping conditions, but it also provides the same test error as relative KKT stopping conditions with $\epsilon = 10^{-3}$. Note that, on the `titanic` and `german` datasets, the slight increase in overall error for $\epsilon = 10^{-1}$ was caused by poor parameters being chosen for 1 of the 10 runs. However, since the remaining 9 runs were successful, the overall differences did not end up being statistically significant.

Polynomial Kernels

Results for polynomial kernels are given in Tables 6.4 and 6.5. Enormous reductions in the number of iterations taken are shown. In many cases these reductions were even greater

than for the Gaussian kernel. Notice, however, that test errors (Table 6.6) are much higher than those for the Gaussian kernel in many cases, particularly the `banana`, `thyroid` and `image` datasets. This appears to be a limitation of radius-margin parameter selection, rather than the polynomial kernel itself. On these datasets the radius-margin method tends to select very low degrees and very small values of C, much lower in both cases than the values which provide the optimal test accuracy.

Table 6.4: Average number of SMO iterations required for each parameter selection technique. The polynomial kernel is used.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	ϵ =adaptive
`banana`	30370.2 ± 1712.8	4053.8 ± 197.4	$\mathbf{72.3 \pm 2.6}$
`b.cancer`	6491.2 ± 275.5	1586.2 ± 71.1	$\mathbf{20.8 \pm 1.2}$
`diabetes`	29841.1 ± 1772.2	5831.7 ± 332.6	$\mathbf{37.2 \pm 1.3}$
`german`	13480.8 ± 308.4	2844.3 ± 64.2	$\mathbf{55.5 \pm 1.5}$
`heart`	707.5 ± 44.6	140.3 ± 10.2	$\mathbf{11.6 \pm 0.4}$
`image`	9972.2 ± 605.8	2060.5 ± 102.2	$\mathbf{124.7 \pm 2.1}$
`splice`	3919.0 ± 217.4	755.1 ± 77.1	$\mathbf{46.1 \pm 2.6}$
`thyroid`	544.1 ± 23.1	115.3 ± 4.0	$\mathbf{13.7 \pm 0.6}$
`titanic`	4736.8 ± 500.4	383.8 ± 62.3	$\mathbf{16.8 \pm 0.6}$

Table 6.5: Average number of MEB iterations required for each parameter selection technique. The polynomial kernel is used.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	ϵ =adaptive
`banana`	63.1 ± 0.8	$\mathbf{2.2 \pm 0.0}$	35.3 ± 1.9
`b.cancer`	44.4 ± 0.3	$\mathbf{1.8 \pm 0.0}$	8.4 ± 0.5
`diabetes`	70.9 ± 0.6	$\mathbf{1.9 \pm 0.0}$	12.0 ± 0.2
`german`	88.4 ± 0.2	$\mathbf{1.7 \pm 0.0}$	18.5 ± 0.3
`heart`	35.9 ± 0.4	$\mathbf{1.9 \pm 0.0}$	4.7 ± 0.4
`image`	128.9 ± 2.9	$\mathbf{1.7 \pm 0.1}$	20.8 ± 2.7
`splice`	102.1 ± 0.4	$\mathbf{1.6 \pm 0.1}$	3.6 ± 0.2
`thyroid`	38.4 ± 0.3	$\mathbf{2.4 \pm 0.0}$	7.4 ± 0.4
`titanic`	37.5 ± 0.9	$\mathbf{1.5 \pm 0.1}$	5.4 ± 0.2

One way to improve results for polynomial kernels is to incorporate alternate bounds on the leave-one-out error. Recall from Section 6.2.3 that the support vector bound on the leave-one-out error is given by [117]:

$$E_{\text{LOO}} \leq \frac{n_s}{n},$$

where n_s is the number of support vectors in an SVM and n is the number of training points. It may be possible to compute upper and lower bounds on the number of support vectors during training, so the support vector bound is a candidate for improving the radius-margin method.

Table 6.7 compares test errors obtained by selecting parameters using the radius-margin bound to those obtained using the support vector bound. These results suggest that, for polynomial kernels, the support vector bound often characterizes the test error

Table 6.6: Average test errors for each parameter selection technique. The polynomial kernel is used.

	$\epsilon = 10^{-3}$	$\epsilon = 10^{-1}$	ϵ =adaptive
banana	0.404 ± 0.009	0.437 ± 0.013	0.409 ± 0.009
b.cancer	0.282 ± 0.015	0.297 ± 0.017	0.283 ± 0.015
diabetes	0.243 ± 0.007	0.242 ± 0.006	0.244 ± 0.007
german	0.280 ± 0.008	0.288 ± 0.010	0.280 ± 0.008
heart	0.167 ± 0.011	0.166 ± 0.012	0.167 ± 0.011
image	0.271 ± 0.007	0.274 ± 0.005	0.271 ± 0.007
splice	0.158 ± 0.002	0.158 ± 0.002	0.158 ± 0.002
thyroid	0.212 ± 0.014	0.211 ± 0.013	0.212 ± 0.014
titanic	0.236 ± 0.004	0.256 ± 0.013	0.236 ± 0.004

better than the radius-margin bound. However, it also suggests that the support vector bound is much more likely to reach a lower minimum than the radius-margin bound, even in cases where the radius-margin bound better characterizes the test error. This issue is demonstrated in Figure 6.10, where even though minimizing the radius-margin bound results in the lowest test error, the support vector bound has the lowest overall minimum of the two bounds. Because of this, it is not possible to choose the better bound by simply taking the minimum of the two bounds.

Table 6.7: Test error achieved by combining two different error bounds. The polynomial kernel is used. The near-perfect match of the first and second columns shows that the SV bound usually reaches the lowest minimum when the polynomial kernel is used.

	SV Bound	min(SV,RM)	RM Bound
banana	$\mathbf{0.110 \pm 0.003}$	$\mathbf{0.110 \pm 0.003}$	0.404 ± 0.009
b.cancer	0.353 ± 0.023	0.353 ± 0.023	$\mathbf{0.282 \pm 0.015}$
diabetes	0.305 ± 0.005	0.305 ± 0.005	$\mathbf{0.243 \pm 0.007}$
german	0.298 ± 0.007	0.298 ± 0.007	$\mathbf{0.280 \pm 0.008}$
heart	0.251 ± 0.012	0.251 ± 0.012	$\mathbf{0.167 \pm 0.011}$
image	$\mathbf{0.049 \pm 0.002}$	$\mathbf{0.049 \pm 0.002}$	0.271 ± 0.007
splice	$\mathbf{0.120 \pm 0.002}$	$\mathbf{0.120 \pm 0.002}$	0.158 ± 0.002
thyroid	$\mathbf{0.044 \pm 0.005}$	$\mathbf{0.044 \pm 0.005}$	0.212 ± 0.014
titanic	0.235 ± 0.010	0.236 ± 0.004	0.236 ± 0.004

There were also cases where the radius-margin bound combined with the polynomial kernel simply was not characteristic of the test error. For example, Figure 6.11 shows the radius-margin bound on the banana dataset for $q = 4$. Here choosing the value of C using the radius-margin bound results in almost the worst possible test error. Even when both parameters are chosen simultaneously (as in Table 6.6), the final accuracy was still extremely poor.

Although the test errors for the polynomial kernel are poor on some datasets, they are important because they illustrate the risks of performing radius-margin parameter selection in conjunction with polynomial kernels. For example, the test error achieved on the banana dataset is greater than 40%, whereas Table 6.7 shows that errors of around 11%

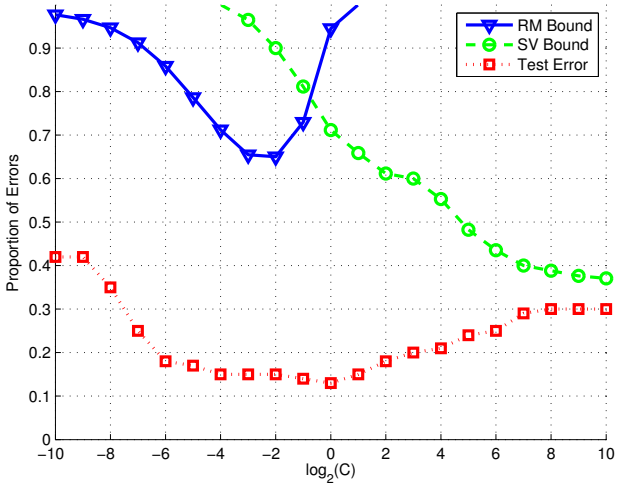


Figure 6.10: Several estimates of the leave one out error compared to the test error for the `heart` dataset in conjunction with the polynomial kernel. The degree is fixed at $q = 2$ while the regularization parameter is changed

are possible. Despite this, the radius-margin ratio has previously been used for successful polynomial kernel parameter selection on a limited number of datasets [117, 116], and indeed it was able to provide an accuracy comparable to the Gaussian kernel in *some* of our test cases. This suggests that the success of the radius-margin technique is likely to depend on the characteristics of the individual dataset being classified, and that care must be taken.

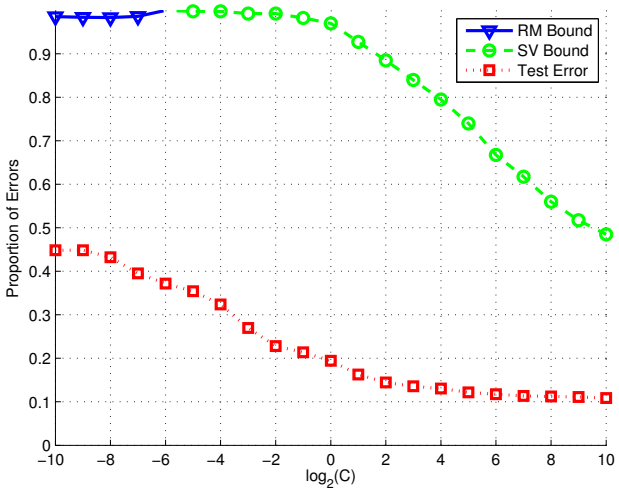


Figure 6.11: Several estimates of the leave one out error compared to the test error for the `banana` dataset in conjunction with the polynomial kernel. The degree is fixed at $q = 4$ while the regularization parameter is changed

6.3.9 Discussion

Despite the significant reductions in training iterations we have shown, we should emphasize that training iterations alone are not the *sole* cost in performing parameter selection. This means that a factor of 10 reduction in training iterations will not necessarily translate into a factor of 10 reduction in the time required to perform parameter selection. Even if no training iterations are performed for a particular combination of parameters, $\mathbf{w} \cdot \mathbf{x}_i$ values for each training point must be computed in order to check the KKT conditions for optimality and initialize the bounds. However, this process can be sped up via caching of dot products, and even if training iterations are not the sole cost of parameter selection, they are still one of the most significant costs.

The savings that can be achieved also clearly depend on the size of the grid over which parameter selection is performed. Results in Section 6.3.8 suggest that most of the training iterations tend to be concentrated around the optimal parameters. This means if a small grid enclosing the optimal parameters is chosen, savings obtained using our method will be reduced. However, it is difficult and risky to choose the limits for a small grid correctly. If a small grid is chosen which does not enclose the optimal parameters, the result will be a very poor set of parameters.

It is important to note that the method of comparison described in Algorithm 12 does not necessarily have to be used in conjunction with a grid search. A natural direction for future research is to improve the search routine. However, one of the benefits of a grid search is that it is can not be drawn away from the optimum and terminate with poor parameters, which Keerthi [63] notes is a possibility using gradient descent search on some datasets. This means a grid search provides confidence that a reasonable set of parameters will be found. A grid search is also made feasible given the fact that bad choices for the parameter values can be so quickly rejected using Algorithm 12.

Another factor impacting the overall time required to perform parameter selection is the training algorithm chosen. Although we use the SMO variant described by López et al. [77] due to it enabling simple calculation of our bounds, it is possible to apply alternative methods. The downside to using other methods is that the bounds are no longer 'free' to calculate, requiring a pass through all training data to find the points which define R^{up} and Δ^{low}. Although this may be too costly to perform every iteration, bounds could still be computed every k iterations. Bounds would also still be useful in rejecting neighboring parameter values where no further training is required. According to Figure 6.9, such parameter values tend to form a large portion of the search space.

6.4 Parameter Selection using μ-SVMs

In the previous section we focused specifically on parameter selection for L_1-loss SVMs. For L_1-loss SVMs the radius-margin technique is inappropriate because it will always choose the greatest possible amount of reduction in the hulls (as we described in Section 6.2.6). Accordingly, alternative methods such as XiAlpha [59] and GACV [74, 122] have

been proposed. However, in previous studies [33, 46], these estimates of the test error have generally been applied to C-SVMs.

We begin this section by reviewing some of the issues that arise when performing parameter selection using C-SVMs. Although the C-SVM formulation is the most common type of SVM with which to perform parameter selection, this may simply be because C-SVMs are so widely used, rather than because they provide any practical benefits compared to other formulations.

The subsequent parts of this section describe how parameter selection can be performed by combining L_1-loss μ-SVMs with several estimates of the test error. Despite the fact that parameter selection is rarely performed using this type of SVM, we suggest that there are many benefits to their use.

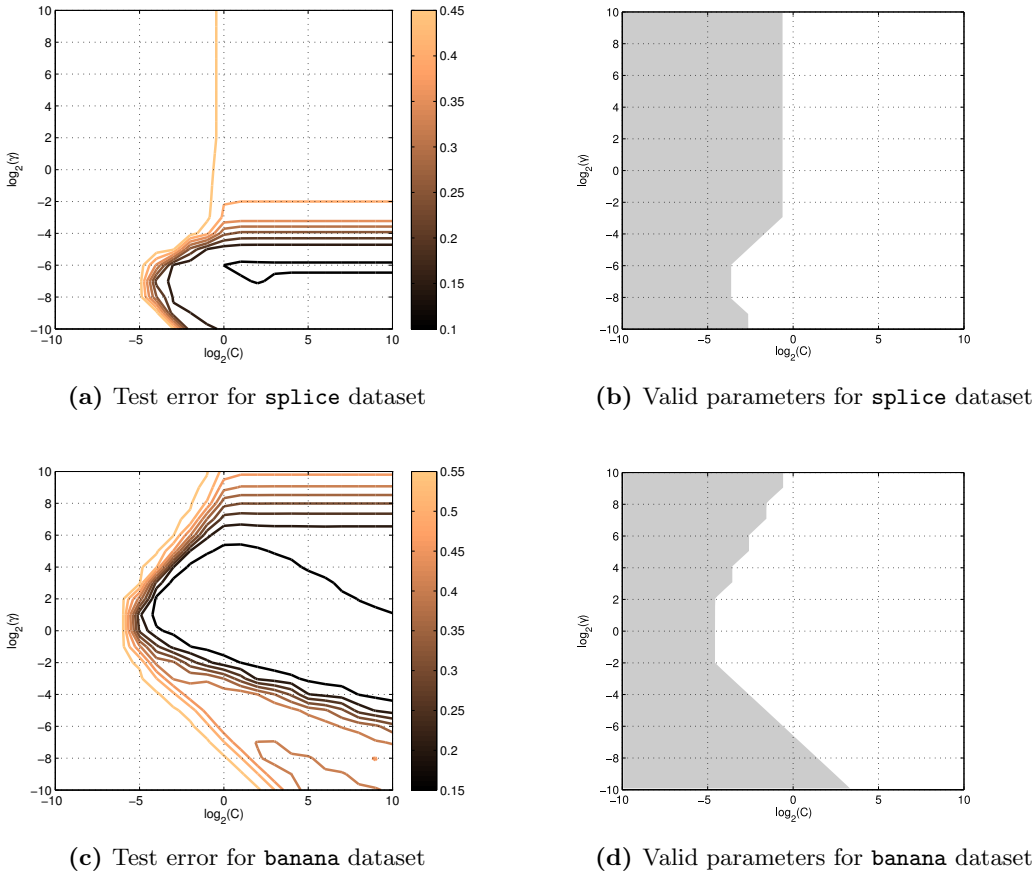
6.4.1 C-SVM Parameter Search

Because C can take any positive value, it is most convenient to search through $\log_2(C)$, meaning a search is performed over a range such as $\log_2(C) \in \{-10, -9, \ldots, 9, 10\}$. The search through log-space allows a large range of C-values to be searched in a reasonable amount of time. However, it is not necessarily an ideal way to select a regularization parameter.

One of the issues that arises in C-SVM parameter search is that many smaller values of C reduce one or both classes to their centroid. Once this occurs, the threshold given by the KKT conditions tends to exhibit unintuitive behavior by classifying all points as belonging to whichever class has the greater number of points (refer to Section 4.3). Not only are such SVMs extremely unlikely to yield a high classification accuracy, but some estimates of the test error may not provide sensible results for these special cases. For example, Joachims [59]'s estimate of the test error requires that an SVM have at least one support vector from each class for which $0 < \alpha_i < C$. This can not be the case when one of the classes is reduced to its centroid, since in such a case all support vectors in that class will reach their upper bound (refer to Section 4.2.4).

This issue is illustrated in Figure 6.12. The left side of this figure shows contour plots of test accuracy for two datasets. A matching pair of figures on the right side of the figure shows the region of parameter space for which neither class is reduced to its centroid. Notice how this region varies based on the dataset and kernel parameters, so it is not possible to know in advance whether a particular combination of parameters will reduce one of the classes to its centroid. Notice also that the SVMs which fall outside of this region generally classify everything as belonging to one class.

Another potential issue with C-SVM parameter selection is that, on some datasets, changing C may result in either a very large or very small change to the reduction in the hulls of the two classes. This is shown in Figure 6.13, where $\log_2(C)$ values are iterated through on the `splice` dataset and plotted against their corresponding $1/\mu$ value. The polynomial kernel with $q = 2$ is used. On this dataset there are then several rapid decreases in $1/\mu$ until $\mu = 1$ is reached, at which point subsequently larger C values all result in a



(a) Test error for `splice` dataset

(b) Valid parameters for `splice` dataset

(c) Test error for `banana` dataset

(d) Valid parameters for `banana` dataset

Figure 6.12: Test error and valid parameters for the Gaussian kernel. Regions where parameter values are invalid are shaded.

value of $\mu = 1$. A smooth transition in the reduction of the hulls is not achieved in this case.

6.4.2 μ-SVM Parameter Search

Using a μ-SVM for parameter search appears difficult due to the fact that there are only a small range of valid μ values. Unlike a C-SVM, where C can be set to any positive value, there are many values of μ which result in either intersecting hulls, or empty hulls, both of which make the μ-SVM optimization task yield no solution. It is also not immediately apparent what sort of step size should be taken when searching μ values, since a reasonable step size for one dataset may be much too large or small for another dataset.

Despite these apparent difficulties, there are reasonable start points, end points and step sizes which can be chosen for any dataset, provided the class sizes are used to guide this choice. We set the starting point of the search to $\mu = 1/(m-2)$, where:

$$m = \min\left(\operatorname{card}\left(I_{pos}\right), \operatorname{card}\left(I_{neg}\right)\right).$$

This starting point is small enough to ensure the hulls are separable for our datasets, while still ensuring neither class is reduced to its centroid.

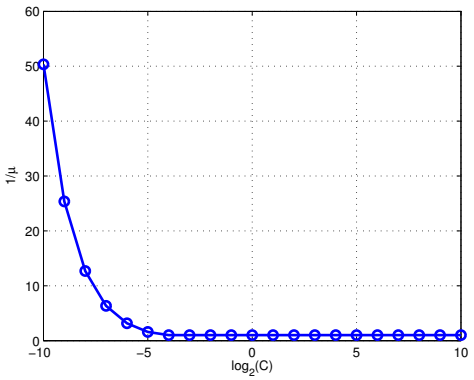


Figure 6.13: Values of $\log_2(C)$ and their equivalent values of μ on the `splice` dataset. The polynomial kernel with $q = 2$ is used

The step size we use is given by:

$$\delta = \frac{-m}{n_{steps}}.$$

However, this step is not simply added to μ. Instead, the new value of μ after each step is given by:

$$\mu \leftarrow \frac{1}{\max\left(1/\mu + \delta, 1\right)}.$$

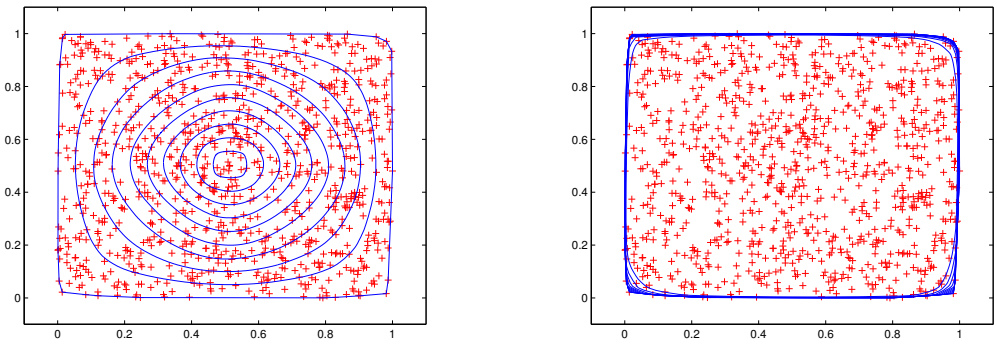
This means that if there are, say, 200 points in the smallest class, μ will step through the values:

$$\left\{\frac{1}{198}, \frac{1}{188}, \frac{1}{178}, \ldots, \frac{1}{8}, 1\right\}$$

We use the value of $n_{steps} = 20$ throughout most experiments because it means there are at most 20 SVMs trained in order to choose μ, similar to the amount of steps taken when choosing $\log_2(C)$ over $-10, -9, \ldots, 9, 10$. The constant n_{steps} can be adapted in order to adjust the coarseness of the search.

Searching through $1/\mu$ values is deliberate, and not equivalent to stepping linearly through μ values in the range $[1/m, 1]$. For an example of why this is the case, consider Figure 6.14. Figure 6.14a shows a search through $1/\mu$ values on a set of uniform random points. Notice the smooth transition in the reduction of the hulls. By contrast, Figure 6.14b shows a search directly through μ values. As seen in this figure, such a search skips over many important values of μ.

The search is stopped once μ reaches one. However, it is likely that, before this occurs, a value of μ will be chosen which is sufficiently large that the RCHs of the two classes intersect. We detect this by checking whether $||\mathbf{w}||$ has grown small (less than 10^{-5}). If $||\mathbf{w}||$ reaches this point, training is terminated and smaller values of μ are not searched. Although such small values of $||\mathbf{w}||$ do not guarantee intersecting hulls, we found that this was a reasonable indicator, as supported by the empirical trials in the following section.

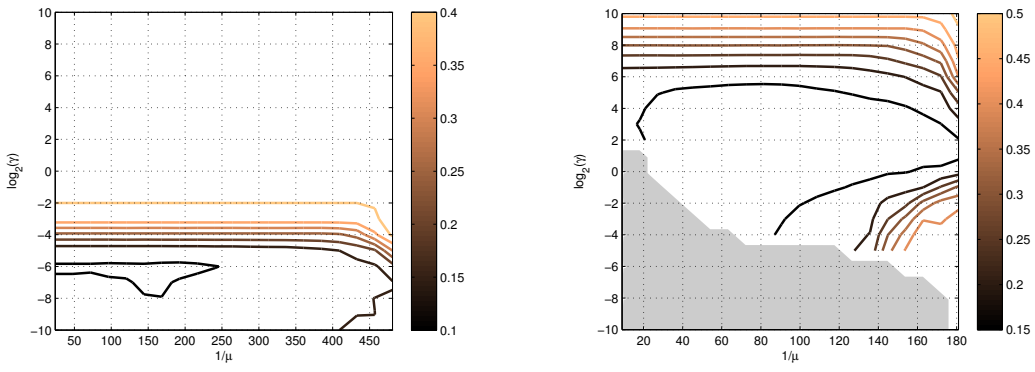


(a) RCHs for $\mu = \frac{1}{900}, \frac{1}{800} \ldots, \frac{1}{100}, 1$ **(b)** RCHs for $\mu = \frac{100}{1000}, \frac{200}{1000} \ldots, \frac{900}{1000}, 1$

Figure 6.14: Searching through $1/\mu$ provides a much smoother transition in the hulls that searching through μ.

Figure 6.15 shows a contour plot of test accuracies for a range of values of μ and γ for the Gaussian kernel. Notice that these plots look similar to a scaled and reflected version of the plots for C-SVMs in Figure 6.12. However, unlike the C-SVM plots, there are no large regions of 'wasted' search space where parameters which reduce the classes past their centroids are searched.

Figure 6.15 also illustrates a further benefit of searching through μ. For the `banana` dataset, the two hulls overlap for most values of μ when γ is small. Searching through μ values, we are able to detect this and omit a large portion of the search space. By contrast, if we were searching through C values, we would train an equal number of SVMs for each kernel parameter. This means potentially training 20 SVMs on a relatively small interval of $\mu = 1/170$ to $\mu = 1/180$.



(a) `splice` dataset **(b)** `banana` dataset

Figure 6.15: Test error for the Gaussian kernel. Shaded regions not searched.

6.4.3 Error Estimates for μ-SVMs

Existing estimates of the test error designed for C-SVMs can be converted to estimate the test error for μ-SVMs by taking advantage of the fact that the μ-SVM is a reparameterized C-SVM. Using the relationship between the dual variables of the μ and C-SVM

optimization tasks which we described in Section 6.2, we obtained both GACV and XiAlpha estimates of the test error for μ-SVMs. The GACV estimate of the test error is shown in Equation (6.8), while the XiAlpha estimate is given in Equation (6.9). In these equations R_Δ^2 is an upper bound on $\mathbf{x}_i \cdot \mathbf{x}_i - \mathbf{x}_i \cdot \mathbf{x}_j$.

$$E^\mu_{\text{GACV}} = \frac{1}{2\rho n}\left[\sum_{i=1}^{n} 2\xi_i + \sum_{y_i f(\mathbf{x}_i)<-\rho} 2\alpha_i \mathbf{x}_i \cdot \mathbf{x}_i + \sum_{y_i f(\mathbf{x}_i)\in[-\rho,\rho]} \alpha_i \mathbf{x}_i \cdot \mathbf{x}_i\right] \tag{6.8}$$

$$E^\mu_{\xi\alpha} = \frac{1}{n}\,\text{card}\left(\left\{ i \;\middle|\; (\alpha_i R_\Delta^2 + \xi_i) \geq \rho \right\}\right) \tag{6.9}$$

6.4.4 Empirical Trials

In order to test whether performing parameter selection with μ-SVMs is viable, we combined a μ-SVM search with two existing estimates of test error: GACV and XiAlpha. The results are shown in Table 6.8 (for GACV) and Table 6.9 (for XiAlpha). Error rates are calculated as a mean over 10 runs, with standard errors also shown. Because differences in accuracy were not statistically significant using a paired differences t-test, we have simply bolded the method which achieved the lowest mean test error on each dataset. These results indicate that estimates of the test error such as XiAlpha and GACV can be viably combined with μ-SVMs in order to perform parameter selection without degrading accuracy.

Table 6.8: GACV parameter selection with μ and C-SVMs

dataset	C-SVM	μ-SVM
`banana`	0.109 ± 0.002	0.109 ± 0.002
`b.cancer`	0.299 ± 0.013	$\mathbf{0.297 \pm 0.014}$
`diabetes`	0.249 ± 0.007	$\mathbf{0.244 \pm 0.007}$
`german`	$\mathbf{0.266 \pm 0.007}$	0.270 ± 0.008
`heart`	$\mathbf{0.176 \pm 0.009}$	0.177 ± 0.010
`image`	$\mathbf{0.051 \pm 0.003}$	0.057 ± 0.003
`splice`	$\mathbf{0.123 \pm 0.002}$	0.124 ± 0.002
`thyroid`	0.055 ± 0.011	$\mathbf{0.049 \pm 0.010}$
`titanic`	0.230 ± 0.005	0.230 ± 0.005

Although these results are positive, they are not conclusive since it has not been established that XiAlpha and GACV are unbiased. For example, Duan et al. [33] show that GACV and XiAlpha can sometimes fail to identify the optimal parameters. This means that a μ-SVM search could actually search through better or worse regions of parameter space, but this could be obscured because estimates of the test error may not correctly evaluate for these solutions. In order to check whether this is the case, we repeated the previous tests without using an estimate of the test error to gauge the performance of the search. Instead, we simply recorded the minimum test error across the parameter search. The results of this test are shown in Table 6.10. There were no statistically significant differences in the minimum test error found using the two search methods, so we have

Table 6.9: XiAlpha parameter selection with μ and C-SVMs

dataset	C-SVM	μ-SVM
banana	0.119 ± 0.005	$\mathbf{0.117 \pm 0.004}$
b.cancer	0.308 ± 0.014	$\mathbf{0.304 \pm 0.014}$
diabetes	$\mathbf{0.248 \pm 0.008}$	0.249 ± 0.008
german	0.283 ± 0.010	$\mathbf{0.274 \pm 0.009}$
heart	0.195 ± 0.015	$\mathbf{0.191 \pm 0.015}$
image	0.030 ± 0.002	$\mathbf{0.029 \pm 0.001}$
splice	0.162 ± 0.005	$\mathbf{0.161 \pm 0.004}$
thyroid	$\mathbf{0.051 \pm 0.009}$	0.055 ± 0.008
titanic	0.227 ± 0.002	0.227 ± 0.001

simply bolded the method with the lower error rate in this table. Results suggest that a μ-SVM search is likely to include the same important areas of the search space which are included in a C-SVM parameter search.

Table 6.10: Minimum test error across entire search

dataset	C-SVM	μ-SVM
banana	$\mathbf{0.103 \pm 0.001}$	0.104 ± 0.002
b.cancer	0.240 ± 0.013	$\mathbf{0.236 \pm 0.012}$
diabetes	0.217 ± 0.006	$\mathbf{0.216 \pm 0.005}$
german	0.212 ± 0.006	$\mathbf{0.211 \pm 0.005}$
heart	0.144 ± 0.010	0.144 ± 0.011
image	$\mathbf{0.023 \pm 0.001}$	0.024 ± 0.001
splice	0.105 ± 0.002	0.105 ± 0.002
thyroid	0.020 ± 0.005	0.020 ± 0.005
titanic	0.216 ± 0.001	0.216 ± 0.001

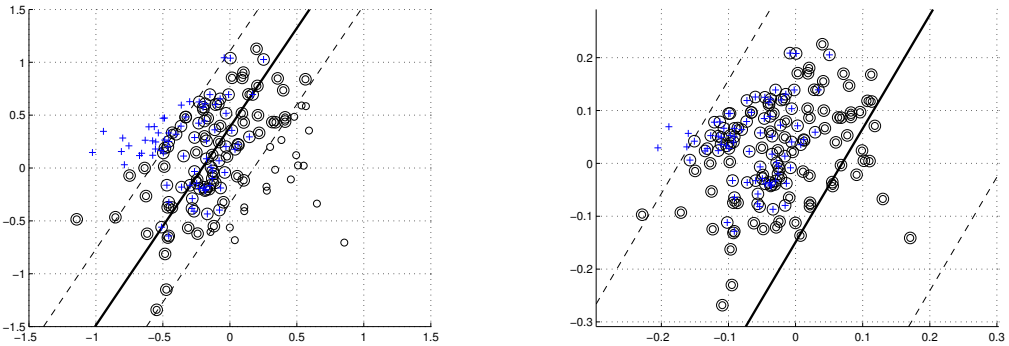
There was also a further benefit associated with μ-SVM parameter search. Because some portions of the search space could be identified as non-viable, a reduced number of parameter combinations could be searched. Portions of the search space become inviable when the classes are reduced to their centroids, or not reduced enough causing their hulls to intersect. The number of valid parameter combinations for each search method is shown in Table 6.11.

Table 6.11: Number of SVMs trained

dataset	C-SVM	μ-SVM
banana	441	293.8 ± 2.1
b.cancer	441	330.5 ± 3.2
diabetes	441	348.7 ± 1.0
german	441	369.2 ± 0.8
heart	441	404.3 ± 0.8
image	441	400.4 ± 0.4
splice	441	420.0 ± 0.0
thyroid	441	381.0 ± 1.2
titanic	441	162.2 ± 9.5

6.4.5 Discussion

In this section we have examined the viability of searching for the optimal SVM parameters using the μ-SVM formulation, as opposed to the more popular C-SVM. Our results suggest that this is a viable approach, and that searching through μ values provides two main benefits. First, combinations of parameter values which cause the classes to be reduced to their centroids can be avoided. This is beneficial since it prevents searching through SVMs which predict everything as belonging to a single class. Second, it allows the search to make a smooth transition in the reduction of the hulls, regardless of the scaling of the data. By this we mean that the value of $\mu = 1/k$ has the same meaning even if all training points are multiplied by a constant. By contrast, the value of $C = 1$ for a C-SVM has a different impact if training points are multiplied by a constant (Figure 6.16).



(a) Original dataset

(b) The same dataset with points scaled so that $\mathbf{x}'_i = \mathbf{x}_i/5$

Figure 6.16: Here the parameter of $C = 1$ is used to train SVMs on two toy datasets which are identical under scaling. Notice the large difference in hyperplane placement and number of support vectors.

6.5 Conclusions

Our results from this chapter have focused primarily on improving existing methods of parameter selection by exploiting the geometric approach. For example, in Section 6.3 we described how radius-margin parameter selection could be accelerated using geometric information. We found that geometric information could be used to provide upper and lower bounds on both the margin of an SVM and the radius of an MEB enclosing the training data. This allowed for SVM training to be terminated extremely early while still selecting parameters which minimized the radius-margin ratio of the SVM.

In Section 6.4, we described how geometric information can guide the parameter search process. We adapted several error estimates, and a simple search method, to apply to the μ-SVM formulation. We found that there was some small advantage to taking such a geometric approach to parameter selection. For example, it provided a clearly defined and geometrically intuitive search space. Accordingly, several parameter values were able to be safely excluded from the search, reducing the number of SVMs which had to be trained.

Something we did not discuss in detail in this chapter is the accuracy of the various test error estimates themselves. This topic has been covered by several other authors [33, 46, 16]. Generally, although estimates of the test error can reduce computational time and provide a reasonable estimate [33], there are as yet no estimates of the test error which are as accurate as the cross-validation or leave-one-out error [16]. This means that, although we would expect techniques such as accelerated radius-margin parameter selection to have immense computational advantages over cross-validation, these computational advantages are likely to come at the cost of some accuracy.

Chapter 7

Conclusions

The geometric framework for SVMs provides a conceptual tool for understanding, generalizing, and improving SVMs. By focusing on the geometric concepts underlying SVMs, we have developed a better understanding of how and why SVMs and WSVMs work. We have applied this knowledge in order to generalize and improve on important SVM-related tasks such as training and parameter selection. In the following sections we summarize in more detail some of the main findings and original contributions from each chapter.

7.1 Reduced Convex Hulls

In Chapter 3 we proposed two algorithms for computing RCHs in their entirety, independently of SVMs. One algorithm is a generalization of the Quickhull algorithm, and operates over points in the plane. The other algorithm operates over points in an arbitrary dimensional space. These algorithms have allowed us to study RCHs in their own right, as a separate concept from SVMs. RCH algorithms provide an alternative to convex hull algorithms where there may be noise or outlying points. The cost of this ability to compensate for noise or outlying points is that an RCH often has more facets than a convex hull due to the smoothing effect that the reduction parameter, μ, has. This increased number of facets can, in turn, make RCHs more costly to compute.

In this chapter we also introduced the concept of Weighted RCHs (WRCHs). WRCHs are a generalized form of RCH where each individual input point can be assigned a unique weight. Allowing individual point weighting provides greater control over the way in which the hull is reduced. For example, the hull can be forced to recede either more slowly or more quickly from certain localized areas. By exploring some of the properties of WRCHs, such as how their vertices may be computed, we have also been able to adapt the previously mentioned RCH algorithms so that they may also compute WRCHs.

7.2 Understanding SVMs from a Geometric Perspective

In Chapter 4 we exploited the geometric interpretation of SVMs in order to understand better how SVMs work. We described the impact of the kernel, the kernel parameters, and the regularization parameter, and showed how changing these parameters can often

make an SVM similar or equivalent to other classifiers such as the k-means and nearest neighbor classifiers.

In this chapter we described the geometric significance of the threshold for the nearest point problem, and explored the difference between the geometric threshold and the threshold suggested by the KKT conditions of the SVM QP optimization task. Empirical results suggested that the KKT threshold often resulted in decreased test accuracy compared to the geometric threshold.

The geometric interpretation also provided a means by which to identify and understand pathological behavior in SVMs. For example, we described how the KKT threshold could result in extremely unintuitive hyperplane placement when it was combined with certain parameter values. In particular, parameters that cause a large amount of reduction in the hulls could result in the hyperplane being placed so that it classified all points as simply belonging to the larger class. This leads us to suggest that, in general, there are few reasons not to prefer a geometric or probabilistic threshold over the KKT threshold.

7.3 Geometric Training Algorithms

Using the concept of WRCHs and the geometric interpretation of WSVMs, we were able to generalize the Schlesinger-Kozinec nearest point algorithm. The resulting algorithm computes the nearest points in two WRCHs, and in doing so is able to train WSVMs. This is, to the best of our knowledge, the first nearest point algorithm which has been proposed for training WSVMs. The algorithm can handle precise (non-integral) weights, without the need for duplicating training points. It can also be used in conjunction with either L_1 or L_2-loss functions. Empirical trials demonstrated that the algorithm is faster than using simple point duplication for weighting.

There are two ways to implement nearest point algorithms which train WSVMs: over two classes, or over a single class consisting of a Minkowski set difference. We suggest that, for the WSK algorithm, the two-class approach often requires more iterations than the one-class approach to train a WSVM to the same precision. However, due to a more efficient update step which involves a smaller number of points, each iteration of the two-class approach is quicker to complete. This means that, overall, we were able to train WSVMs faster using the two-class approach.

We also showed using empirical trials that the stopping conditions commonly used in conjunction with nearest point algorithms tended to stop earlier than necessary when margins were small, and later than necessary when margins were large. We addressed this by recommending an alternative stopping condition which checks the distance from the estimated positions of the nearest points to the true nearest points *relative* to the width of the margin. Empirical trials demonstrated that these stopping conditions led to more consistent behavior across a range of parameter and kernel values.

By using the geometric framework, and by closely examining the differences between perceptrons and SVMs, we were able to adapt the WSK algorithm to train perceptrons, with either L_1 or L_2-loss function and optional weighted training points. Although it is interesting that the geometric similarities between SVMs and perceptrons allow for

very similar training algorithms to be applied to each, we suggest that the perceptron optimization task is generally not as efficient as the SVM optimization tasks when solved using geometric algorithms. The source of this inefficiency is likely to be the feature space induced by the modified kernel which must be used to train perceptrons using a nearest point algorithm.

7.4 Parameter Selection using Geometric Information

In Chapter 6 we studied how the geometric interpretation of SVMs could be used to guide the parameter selection process. We achieved this by performing parameter selection in conjunction with the nearest point optimization task. By contrast, previous methods of parameter selection have most commonly been applied to C-SVMs without considering the geometric interpretation.

One of the main contributions we were able to make in this chapter is to show that the existing method of radius-margin parameter selection could be accelerated to a great extent by applying insights from the geometric framework. This was accomplished by noting that, when the nearest point approach to training was taken, upper and lower bounds on both the margin of an SVM and the Minimal Enclosing Ball (MEB) of the training data could be computed after any number of iterations. Using this observation, we were able to compare upper and lower bounds on the radius-margin ratio across a range of parameter values, without training multiple SVMs to completion. This resulted in a reduction in the number of training iterations required in order to perform parameter selection by a factor of ten or more compared to previous approaches.

We also addressed the task of parameter selection for μ-SVMs. We proposed a basic grid search for μ-SVMs, and showed that using μ-SVMs for parameter selection had several advantages over the more commonly used C-SVM. For example, there were clearly defined start and end points for the search, which were informed by the geometric interpretation of SVMs. By contrast, C-SVMs have no such end points, often requiring an extensive search over $\log_2(C)$, i.e. ranging from small fractional values to extremely large values. We further showed that existing error estimates which have been applied to C-SVMs could easily be adapted to estimate the error of a μ-SVM. Empirical trials suggest that, although C-SVMs are by far the most common SVM to be used for parameter selection, they are not necessarily the best.

7.5 Discussion

The many different types of SVMs and perceptrons and the multitude of ways they can be adapted has led to some confusion in the literature regarding what is and is not an SVM. We hope that our work can aid in clearly identifying how other classifiers relate to SVMs, particularly some of the recently proposed SVM-like classifiers.

One such classifier is the Extreme Learning Machine (ELM) [55, 56, 57]. The ELM uses the decision function:

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{h}$$

It trains this machine using the optimization task:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \frac{1}{2}||\mathbf{w}||^2 + \frac{C}{2}\sum_i \xi_i^2 \\ \text{subject to} \quad & \mathbf{w} \cdot \mathbf{h}(\mathbf{x}) = y_i - \xi_i \end{aligned}$$

This optimization task is equivalent to the L_2-loss perceptron optimization task which we described in Section 2.8.4, and can be solved using one-class nearest point algorithms as described in Section 5.9. However, the ELM optimization task uses an explicit mapping $\mathbf{h}(\mathbf{x})$ on all training data *prior* to training.

Although several mappings for $\mathbf{h}(\mathbf{x})$ are possible, Huang et al. [55] generally use the sigmoidal mapping:

$$\mathbf{h}_i(\mathbf{x}) = \frac{1}{1 + \exp(-(\mathbf{a}_i \cdot \mathbf{x} + b_i))}$$

Here $\mathbf{h}(\mathbf{x})$ has L components (chosen as a parameter), with $\mathbf{a}_i$ and b_i values chosen randomly from any continuous distribution [55].

Something not noted by Huang et al. [55], is that using this mapping does not allow the ELM to reach a solution that the L_2-loss perceptron can not. Indeed, the solution to the ELM optimization task is the same as the solution to the L_2-loss perceptron optimization task, provided the following kernel is used:

$$K(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{L} \frac{1}{1 + \exp(-(\mathbf{a}_i \cdot \mathbf{x} + b_i))} \times \frac{1}{1 + \exp(-(\mathbf{a}_i \cdot \mathbf{y} + b_i))}$$

This means that, rather than being a distinct type of classifier, an ELM could also be described as an L_2-loss perceptron combined with a novel type of kernel. Because this was not considered, it led to ELMs being proposed as a completely new type of classifier, rather than a new type of kernel. In turn, ELMs were compared to Gaussian SVMs and claimed to be superior in many ways [57]. However, due to the drastically different feature spaces induced by the kernels used by ELMs, this is an inherently unfair comparison.

Another SVM-like classifier is the Core Vector Machine (CVM) [114]. The CVM solves the optimization task:

$$\begin{aligned} \min \quad & \frac{1}{2}||\mathbf{w}||^2 + b^2 - 2\rho + C\sum_i \xi_i^2 \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq \rho - \xi_i \qquad \forall i \end{aligned} \tag{7.1}$$

This optimization task is a geometric reparameterization of the L_2-loss perceptron optimization task described in Sections 2.8.4 and 2.8.5, so it is not a new type of classifier in

itself. However, Tsang et al. [115] solve this problem by noting that, when the kernel being used satisfies $K(\mathbf{x}, \mathbf{x}) = \text{constant}$ (e.g. the Gaussian kernel satisfies this property), this optimization task becomes equivalent to the Minimal Enclosing Ball (MEB) optimization task and can hence be solved using MEB solvers.

Because Tsang et al. [115] use an SMO variant to solve the MEB optimization task, this formulation itself is not necessarily an optimization. However, they make large reductions in training time by using vastly different (and often *much* looser [76]) stopping conditions which relate to the number of points contained within the MEB.

When new SVM-like classifies such as CVMs or ELMs are introduced, several different issues often become conflated. This results in simultaneous changes being made to the training routines, the stopping conditions, the kernel being used, and even the threshold. Each of these aspects individually have a large impact on the training efficiency and test accuracy of a machine (as we showed in Chapter 5). However, when a new machine is proposed which modifies several of these factors as once, it is easy to achieve illusory benefits.

In order to determine whether a new SVM-like machine provides real benefits over existing techniques, it is important to keep constant as many components as possible during comparison. For example, to determine whether a new training method provides benefits, it should be compared against existing training methods using an equivalent stopping condition and kernel (including kernel parameters and any regularization parameter). Similarly, in order to determine the benefits of a new type of kernel, the training method and stopping conditions should remain constant. Following this approach aids in determining whether a new method provides substantial benefits over existing techniques and, if so, exactly how these benefits are achieved.

7.6 Future Work

One of the most natural areas for future work is in geometric classifiers which do not use RCHs for class representations. Recall that the use of RCHs arose from the way in which the SVM optimization task penalizes the sum of slack variables. This choice of penalty term was originally made largely for numerical convenience, since trying to minimize the *number* of errors would have resulted in an NP-complete optimization task [27]. It follows that maximizing the margin between two RCHs is not necessarily more theoretically justified than maximizing the margin between other types of geometric class representations. Changing the underlying geometric representation of the classes would almost certainly have an impact on the resulting classifier (Figure 7.1). It would be interesting to investigate whether alternative geometric representations could provide any benefits over RCHs.

An area which we did not cover in detail is that of probabilities in support vector classification. In Chapter 2 we mentioned how it has been shown that replacing the hinge loss function associated with SVMs with the logistic loss function results in the Kernel Logistic Regression (KLR) machine. This is a kernel machine which favors large margins and provides a probabilistic output. Future work may be able to reconcile the geometric

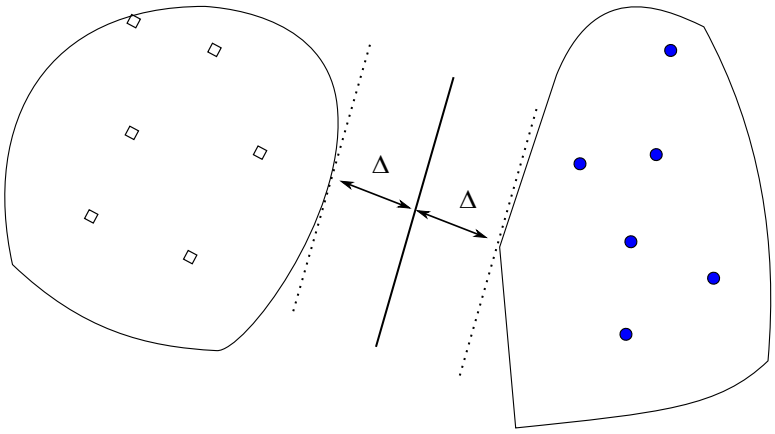


Figure 7.1: Maximizing the margin between two classes with alternative geometric representations

and probabilistic approaches by providing a classifier with both an intuitive geometric interpretation and probabilistic outputs.

In Chapter 4 we provided a geometric interpretation of WSVMs and described several weighting schemes from a geometric perspective. A promising direction for future research is to discover alternative weighting schemes for particular application domains. For example, recent research by Sheng et al. [105] has investigated the use of weighted classifiers in applications where multiple training labels are available, yet labels are uncertain and costly to acquire. It would be interesting to investigate uncertain class labels in conjunction with SVMs and, in turn, gain geometric insight into such classification tasks.

There is still room for further developments in the field of parameter selection for SVMs. In general, none of the error estimates which exist for SVMs can provide an estimate of the error which is as reliable as the estimate given by cross-validation. Furthermore, cross-validation requires such a large amount of computational effort that it can sometimes become infeasible. Bousquet and Schölkopf [16] also hold this view, stating in 2006 that "... there is no satisfactory method for choosing the parameters other than using cross-validation, which can be an obstacle in applications."

It may be that a greater understanding of the geometric framework could aid in the discovery of methods of parameter selection which are as accurate as cross-validation, while not requiring the same amount of computational effort. Franc et al. [41] have taken steps in this direction by describing how the stopping conditions used in computing the leave-one-out error can be relaxed in such a way that there is still a guarantee that the error estimate will be unchanged. This is a result similar to the one we obtained using the radius-margin bound in Chapter 6.

7.7 Closing Remarks

The contributions we have summarized in the previous sections support our claim that the geometric interpretation of SVMs provides invaluable information which can be used to understand, improve on, and generalize several aspects support vector classification. We

hope that this thesis will lead to closer attention being given to the geometric interpretation when research on SVMs is undertaken. Ideally, this thesis will also serve as a foundation for the future work laid out in this chapter.

Appendix A

Datasets used in Empirical Tests

For our empirical trials, we use a selection of datasets from the UCI [42], STATLOG, and Delve machine learning repositories. The datasets we use were originally gathered by Rätsch et al. [96], and have been scaled to unit mean and standard deviation. Table A.1 summarizes the properties of each of the datasets. These datasets were chosen because they provide a large variation in each of the attributes shown in the table, and also because they provide a common benchmark for SVM research [33, 39]. Note that *balance* is the proportion of points belonging to the smaller class. This is the error rate that can be achieved on the dataset by a 'dumb' classifier which simply classifies everything as belonging to the larger class.

Table A.1: Summary of datasets used in empirical trials

	Features	Train Samples	Test Samples	Balance
`banana`	2	400	4900	0.448
`b.cancer`	9	200	77	0.292
`diabetes`	8	468	300	0.349
`german`	20	700	300	0.300
`heart`	13	170	100	0.444
`image`	18	1300	1010	0.429
`splice`	60	1000	2175	0.481
`thyroid`	5	140	75	0.302
`titanic`	3	150	2051	0.323

Appendix B

Extensive Results for Thresholds

Table B.1: Error rate for Gaussian SVMs ($\gamma = 0.01$) combined with three possible thresholds

		KKT	Geometric	Probabilistic
banana	small μ	0.419 ± 0.012	0.428 ± 0.011	0.425 ± 0.013
	large μ	0.418 ± 0.011	0.429 ± 0.012	0.427 ± 0.013
b.cancer	small μ	0.274 ± 0.011	0.277 ± 0.008	0.271 ± 0.007
	large μ	0.276 ± 0.010	0.277 ± 0.007	0.272 ± 0.007
diabetis	small μ	0.269 ± 0.004	$\mathbf{0.241 \pm 0.005}$	$\mathbf{0.239 \pm 0.005}$
	large μ	0.251 ± 0.006	$\mathbf{0.239 \pm 0.005}$	$\mathbf{0.236 \pm 0.005}$
german	small μ	0.249 ± 0.005	$\mathbf{0.235 \pm 0.005}$	$\mathbf{0.235 \pm 0.005}$
	large μ	0.244 ± 0.005	$\mathbf{0.234 \pm 0.005}$	$\mathbf{0.233 \pm 0.005}$
heart	small μ	0.182 ± 0.009	$\mathbf{0.158 \pm 0.008}$	$\mathbf{0.158 \pm 0.007}$
	large μ	0.168 ± 0.008	$\mathbf{0.159 \pm 0.007}$	$\mathbf{0.159 \pm 0.007}$
image	small μ	0.265 ± 0.002	$\mathbf{0.208 \pm 0.004}$	$\mathbf{0.198 \pm 0.003}$
	large μ	0.195 ± 0.016	$\mathbf{0.163 \pm 0.011}$	$\mathbf{0.158 \pm 0.009}$
splice	small μ	0.160 ± 0.002	$\mathbf{0.158 \pm 0.002}$	$\mathbf{0.158 \pm 0.002}$
	large μ	0.140 ± 0.005	$\mathbf{0.139 \pm 0.005}$	0.139 ± 0.005
thyroid	small μ	0.223 ± 0.011	$\mathbf{0.133 \pm 0.009}$	$\mathbf{0.101 \pm 0.009}$
	large μ	0.196 ± 0.011	$\mathbf{0.117 \pm 0.009}$	$\mathbf{0.092 \pm 0.008}$
titanic	small μ	0.238 ± 0.007	0.226 ± 0.001	0.226 ± 0.001
	large μ	0.232 ± 0.005	0.226 ± 0.000	0.226 ± 0.001

Table B.2: Error rate for Gaussian SVMs ($\gamma = 0.1$) combined with three possible thresholds

		KKT	Geometric	Probabilistic
banana	small μ	0.372 ± 0.009	0.365 ± 0.014	0.363 ± 0.014
	large μ	0.349 ± 0.010	0.336 ± 0.013	$\mathbf{0.334 \pm 0.013}$
b.cancer	small μ	0.265 ± 0.010	0.264 ± 0.006	0.260 ± 0.008
	large μ	0.264 ± 0.010	0.271 ± 0.008	0.269 ± 0.009
diabetis	small μ	0.265 ± 0.006	$\mathbf{0.246 \pm 0.005}$	$\mathbf{0.243 \pm 0.004}$
	large μ	0.255 ± 0.006	0.246 ± 0.005	$\mathbf{0.244 \pm 0.004}$
german	small μ	0.276 ± 0.005	$\mathbf{0.238 \pm 0.005}$	$\mathbf{0.237 \pm 0.004}$
	large μ	0.256 ± 0.007	$\mathbf{0.238 \pm 0.005}$	$\mathbf{0.238 \pm 0.004}$
heart	small μ	0.215 ± 0.012	$\mathbf{0.163 \pm 0.008}$	$\mathbf{0.166 \pm 0.008}$
	large μ	0.198 ± 0.010	$\mathbf{0.171 \pm 0.008}$	$\mathbf{0.174 \pm 0.008}$
image	small μ	0.119 ± 0.002	$\mathbf{0.108 \pm 0.002}$	$\mathbf{0.106 \pm 0.002}$
	large μ	0.092 ± 0.006	$\mathbf{0.084 \pm 0.006}$	$\mathbf{0.084 \pm 0.005}$
splice	small μ	0.454 ± 0.006	$\mathbf{0.387 \pm 0.003}$	$\mathbf{0.373 \pm 0.005}$
	large μ	0.421 ± 0.009	$\mathbf{0.384 \pm 0.003}$	$\mathbf{0.378 \pm 0.004}$
thyroid	small μ	0.101 ± 0.007	$\mathbf{0.081 \pm 0.006}$	$\mathbf{0.057 \pm 0.007}$
	large μ	0.087 ± 0.008	$\mathbf{0.065 \pm 0.007}$	$\mathbf{0.055 \pm 0.006}$
titanic	small μ	0.229 ± 0.001	0.227 ± 0.001	0.229 ± 0.001
	large μ	0.229 ± 0.001	$\mathbf{0.228 \pm 0.001}$	0.229 ± 0.001

Table B.3: Error rate for linear SVMs combined with three possible thresholds

		KKT	Geometric	Probabilistic
banana	small μ	0.485 ± 0.010	0.481 ± 0.011	0.484 ± 0.011
	large μ	0.489 ± 0.011	0.480 ± 0.011	0.486 ± 0.011
b.cancer	small μ	0.292 ± 0.010	0.286 ± 0.007	0.290 ± 0.009
	large μ	0.294 ± 0.009	0.293 ± 0.007	0.293 ± 0.009
diabetis	small μ	0.234 ± 0.004	$\underline{0.238 \pm 0.005}$	0.233 ± 0.004
	large μ	0.234 ± 0.004	$\underline{0.239 \pm 0.005}$	0.233 ± 0.004
german	small μ	0.236 ± 0.005	0.243 ± 0.005	0.234 ± 0.005
	large μ	0.236 ± 0.005	$\underline{0.242 \pm 0.005}$	0.235 ± 0.005
heart	small μ	0.162 ± 0.007	0.166 ± 0.007	0.164 ± 0.007
	large μ	0.169 ± 0.006	0.171 ± 0.006	0.169 ± 0.007
image	small μ	0.161 ± 0.002	$\underline{0.166 \pm 0.003}$	0.161 ± 0.003
	large μ	0.158 ± 0.002	$\underline{0.164 \pm 0.003}$	0.158 ± 0.003
splice	small μ	0.164 ± 0.001	0.164 ± 0.001	0.164 ± 0.001
	large μ	0.164 ± 0.001	$\underline{0.165 \pm 0.001}$	0.164 ± 0.002
thyroid	small μ	0.154 ± 0.009	$\mathbf{0.105 \pm 0.009}$	$\mathbf{0.115 \pm 0.009}$
	large μ	0.134 ± 0.010	$\mathbf{0.098 \pm 0.008}$	$\mathbf{0.111 \pm 0.009}$
titanic	small μ	0.225 ± 0.001	0.225 ± 0.001	0.225 ± 0.001
	large μ	0.225 ± 0.001	0.225 ± 0.001	0.225 ± 0.001

Table B.4: Error rate for polynomial SVMs ($q = 3$) combined with three possible thresholds

		KKT	Geometric	Probabilistic
banana	small μ	0.233 ± 0.006	0.228 ± 0.006	$\mathbf{0.229 \pm 0.006}$
	large μ	0.231 ± 0.006	$\mathbf{0.226 \pm 0.006}$	$\mathbf{0.227 \pm 0.006}$
b.cancer	small μ	0.320 ± 0.007	$\underline{0.351 \pm 0.007}$	0.323 ± 0.007
	large μ	0.332 ± 0.009	$\underline{0.356 \pm 0.009}$	0.334 ± 0.009
diabetis	small μ	0.301 ± 0.005	0.304 ± 0.006	0.302 ± 0.006
	large μ	0.312 ± 0.006	0.314 ± 0.006	0.313 ± 0.006
german	small μ	0.300 ± 0.007	0.300 ± 0.007	0.298 ± 0.006
	large μ	0.300 ± 0.007	0.300 ± 0.007	$\mathbf{0.298 \pm 0.006}$
heart	small μ	0.229 ± 0.009	0.229 ± 0.009	0.230 ± 0.010
	large μ	0.229 ± 0.009	0.229 ± 0.009	0.230 ± 0.010
image	small μ	0.037 ± 0.001	0.037 ± 0.001	0.037 ± 0.001
	large μ	0.041 ± 0.001	0.041 ± 0.001	0.040 ± 0.001
splice	small μ	0.127 ± 0.002	0.130 ± 0.003	0.126 ± 0.002
	large μ	0.127 ± 0.002	0.130 ± 0.003	$\mathbf{0.126 \pm 0.002}$
thyroid	small μ	0.067 ± 0.008	0.066 ± 0.007	$\underline{0.079 \pm 0.009}$
	large μ	0.070 ± 0.007	0.071 ± 0.006	$\underline{0.083 \pm 0.008}$
titanic	small μ	0.225 ± 0.002	0.224 ± 0.002	0.223 ± 0.002
	large μ	0.225 ± 0.002	0.224 ± 0.002	0.223 ± 0.002

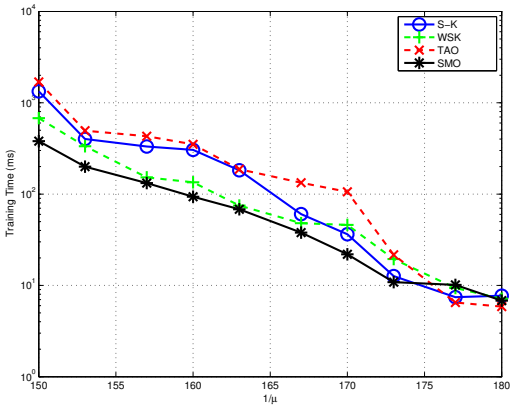
Table B.5: Error rate for polynomial SVMs ($q = 5$) combined with three possible thresholds

		KKT	Geometric	Probabilistic
banana	small μ	0.108 ± 0.001	$\underline{0.111 \pm 0.001}$	$\underline{0.110 \pm 0.001}$
	large μ	0.110 ± 0.003	0.111 ± 0.002	0.109 ± 0.001
b.cancer	small μ	0.359 ± 0.012	$\underline{0.394 \pm 0.013}$	0.360 ± 0.012
	large μ	0.359 ± 0.012	$\underline{0.398 \pm 0.013}$	0.360 ± 0.012
diabetis	small μ	0.333 ± 0.004	0.334 ± 0.004	0.333 ± 0.004
	large μ	0.333 ± 0.004	0.334 ± 0.004	0.332 ± 0.004
german	small μ	0.280 ± 0.008	0.278 ± 0.007	$\mathbf{0.275 \pm 0.007}$
	large μ	0.280 ± 0.008	0.278 ± 0.007	$\mathbf{0.275 \pm 0.007}$
heart	small μ	0.220 ± 0.010	0.220 ± 0.010	0.222 ± 0.011
	large μ	0.220 ± 0.010	0.220 ± 0.010	0.222 ± 0.011
image	small μ	0.050 ± 0.013	0.038 ± 0.001	0.038 ± 0.001
	large μ	0.050 ± 0.012	0.038 ± 0.001	0.038 ± 0.001
splice	small μ	0.162 ± 0.011	$\mathbf{0.132 \pm 0.002}$	$\mathbf{0.129 \pm 0.002}$
	large μ	0.164 ± 0.011	$\mathbf{0.132 \pm 0.002}$	$\mathbf{0.129 \pm 0.002}$
thyroid	small μ	0.077 ± 0.009	0.076 ± 0.009	$\underline{0.167 \pm 0.012}$
	large μ	0.078 ± 0.009	0.077 ± 0.009	$\underline{0.166 \pm 0.012}$
titanic	small μ	0.288 ± 0.035	0.294 ± 0.033	0.246 ± 0.023
	large μ	0.309 ± 0.034	0.308 ± 0.031	$\mathbf{0.254 \pm 0.023}$

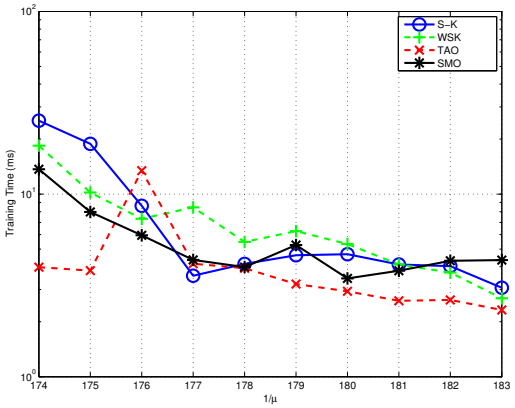
Appendix C

Extensive Results for Nearest Point Algorithms

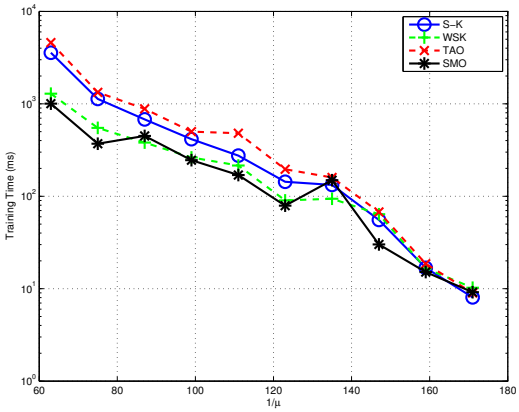
C.1 Training Times



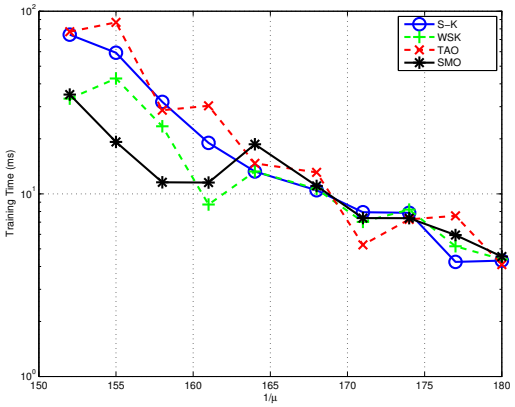
(a) Gaussian kernel with $\gamma = 0.01$



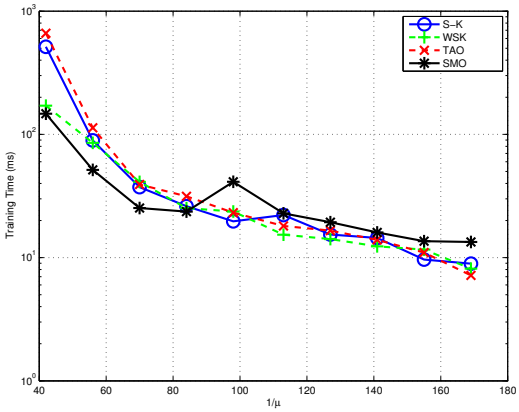
(b) linear kernel



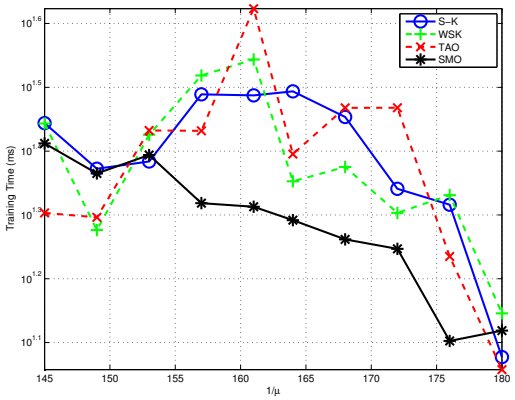
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

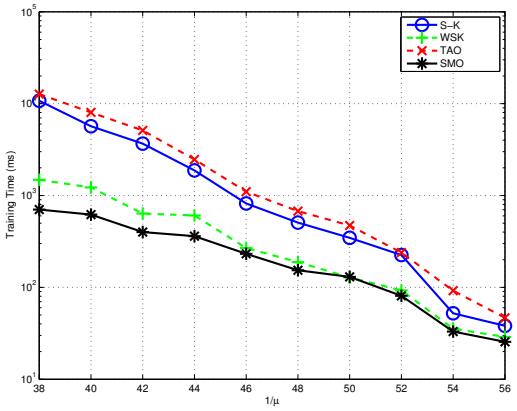


(e) Gaussian kernel with $\gamma = 1$

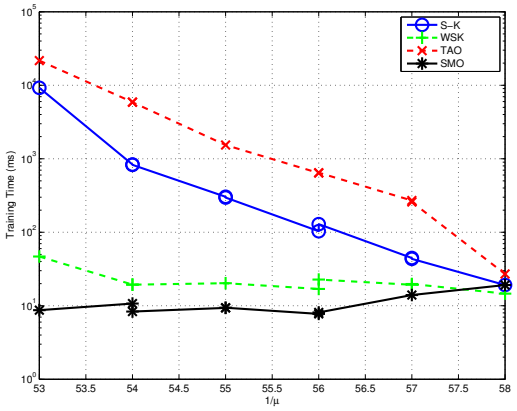


(f) polynomial kernel with $q = 4$

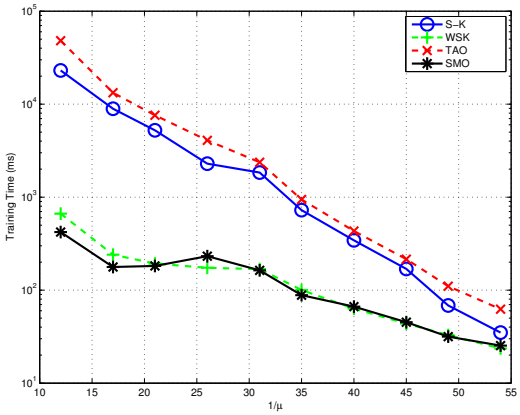
Figure C.1: Training times for the `banana` dataset



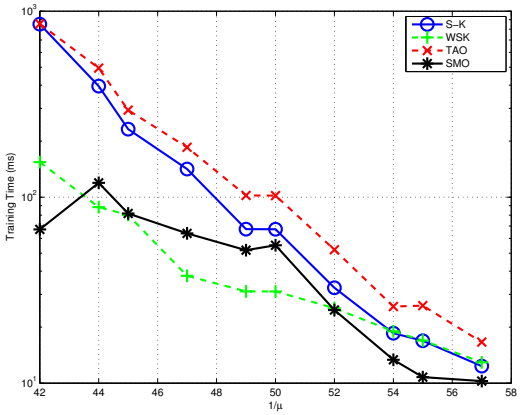
(a) Gaussian kernel with $\gamma = 0.01$



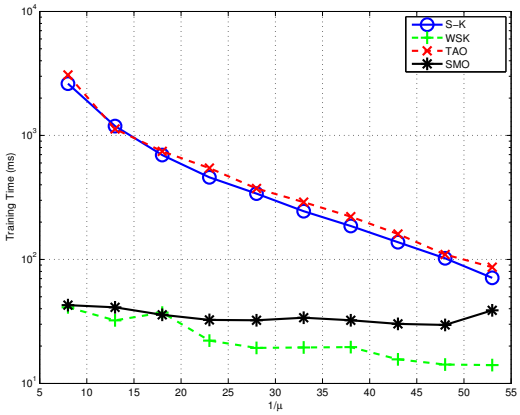
(b) linear kernel



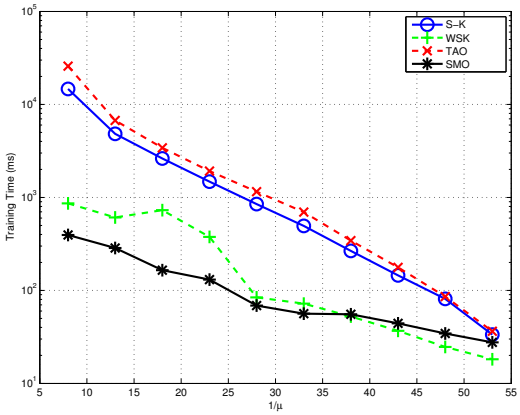
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

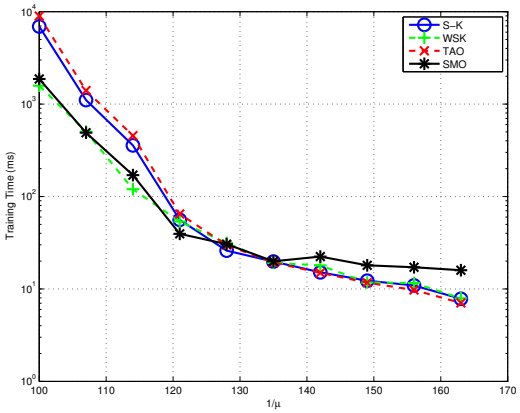


(e) Gaussian kernel with $\gamma = 1$

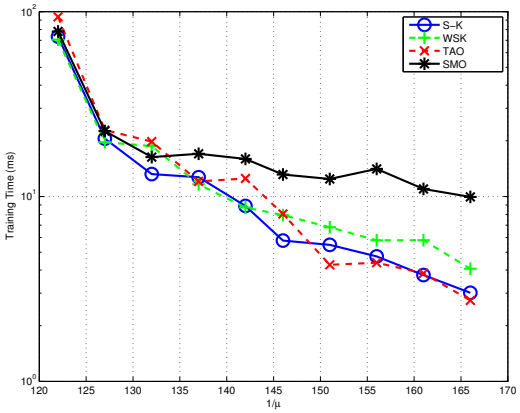


(f) polynomial kernel with $q = 4$

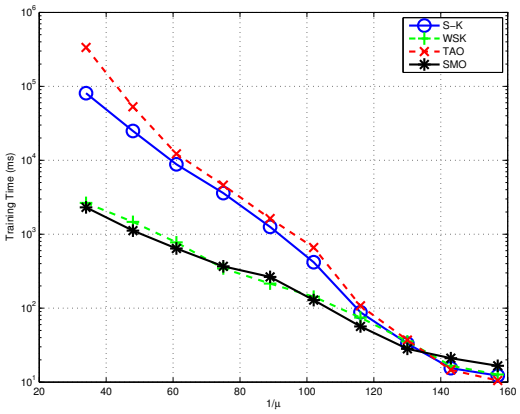
Figure C.2: Training times for the `b.cancer` dataset



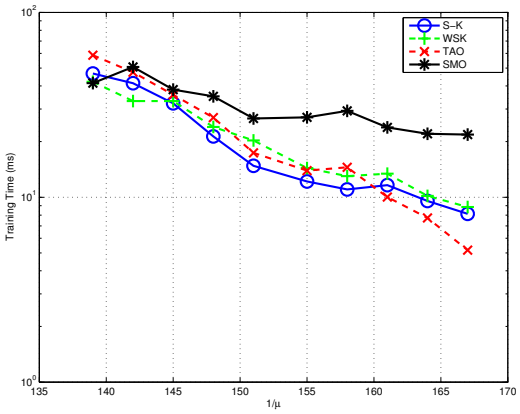
(a) Gaussian kernel with $\gamma = 0.01$



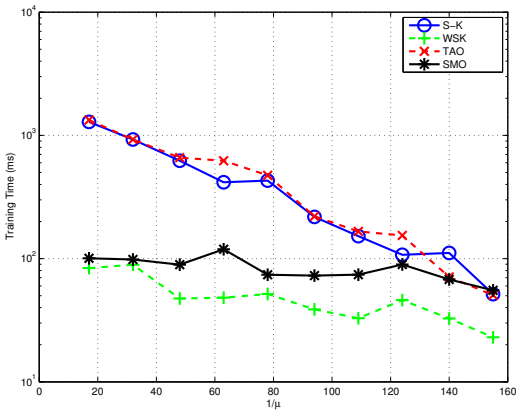
(b) linear kernel



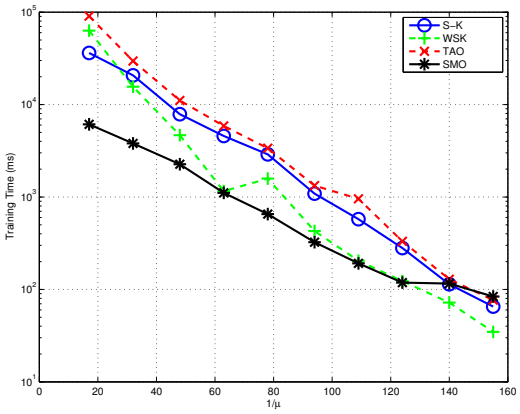
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

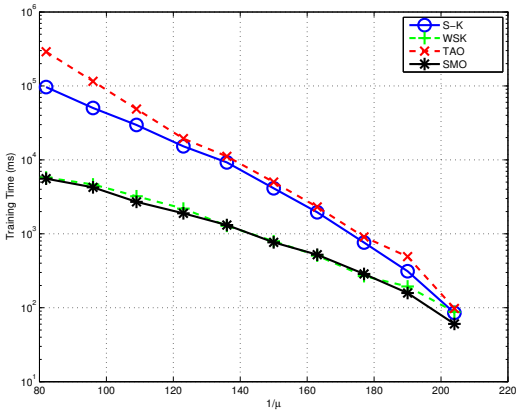


(e) Gaussian kernel with $\gamma = 1$

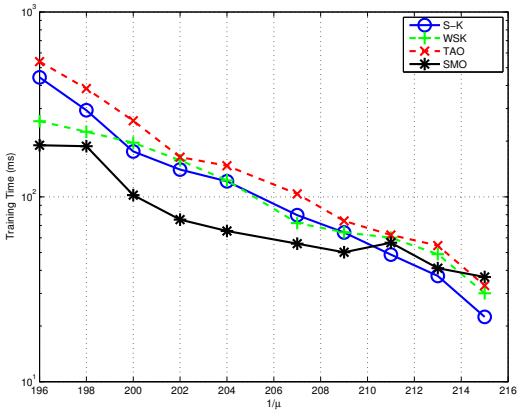


(f) polynomial kernel with $q = 4$

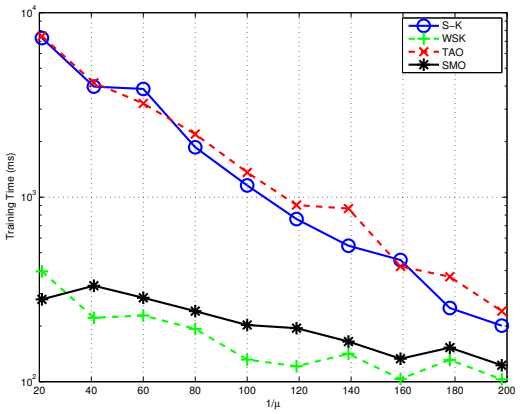
Figure C.3: Training times for the `diabetes` dataset



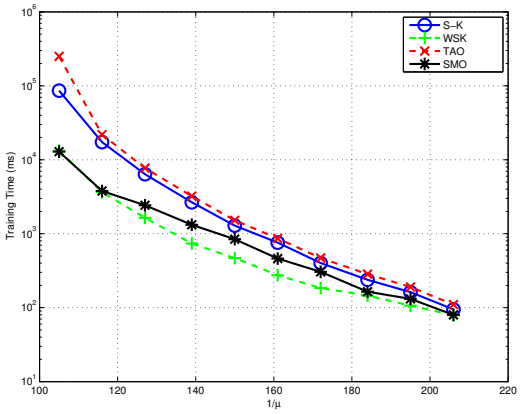
(a) Gaussian kernel with $\gamma = 0.01$



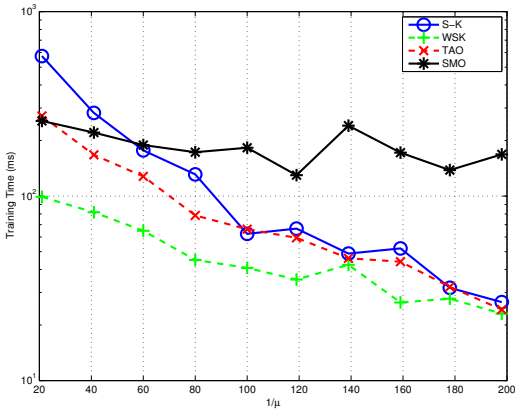
(b) linear kernel



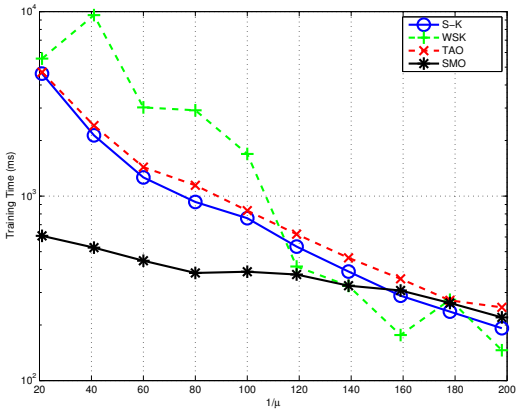
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

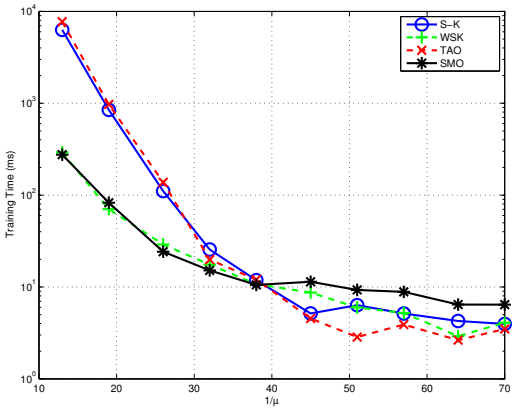


(e) Gaussian kernel with $\gamma = 1$

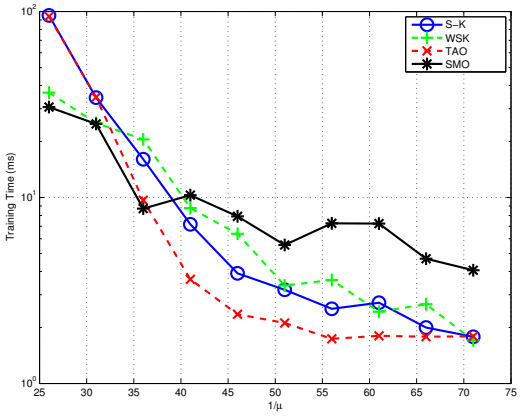


(f) polynomial kernel with $q = 4$

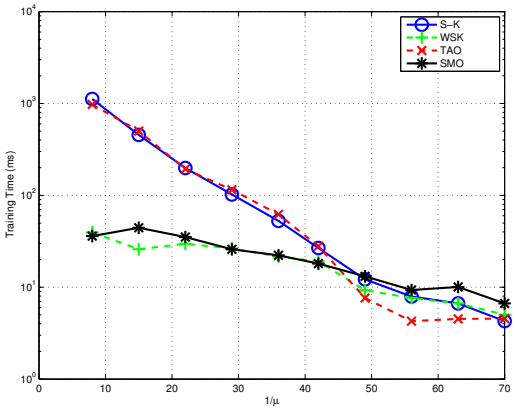
Figure C.4: Training times for the `german` dataset



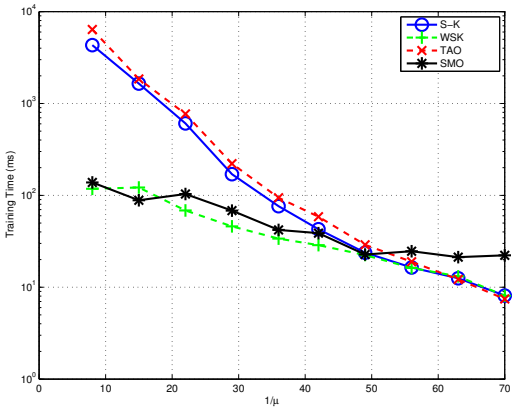
(a) Gaussian kernel with $\gamma = 0.01$



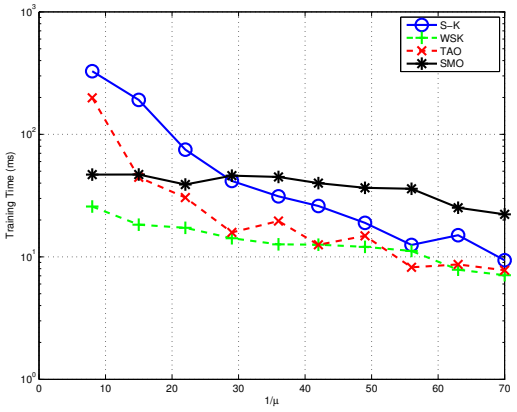
(b) linear kernel



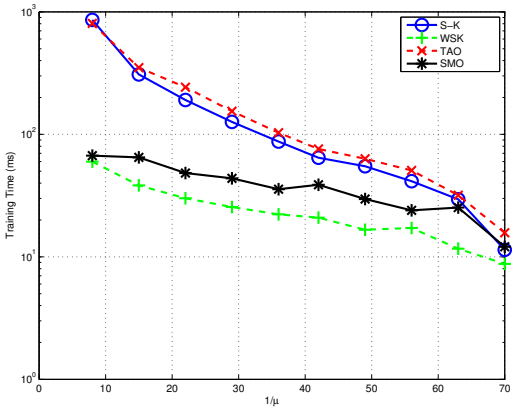
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

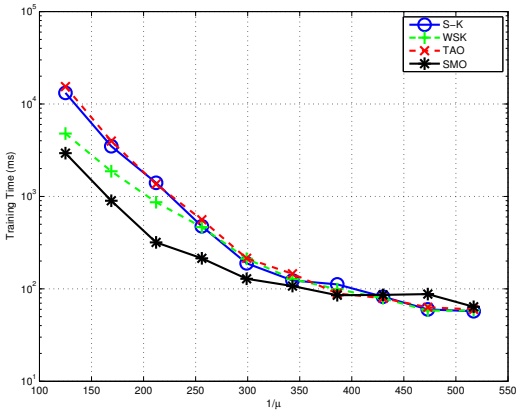


(e) Gaussian kernel with $\gamma = 1$

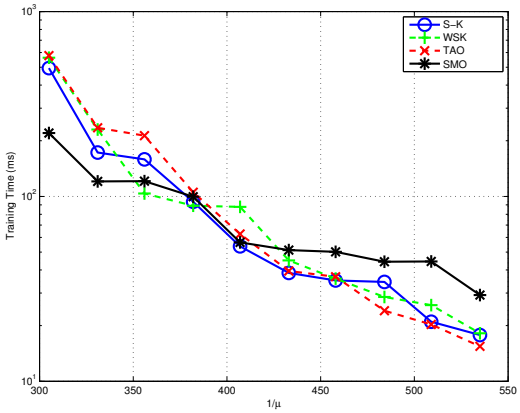


(f) polynomial kernel with $q = 4$

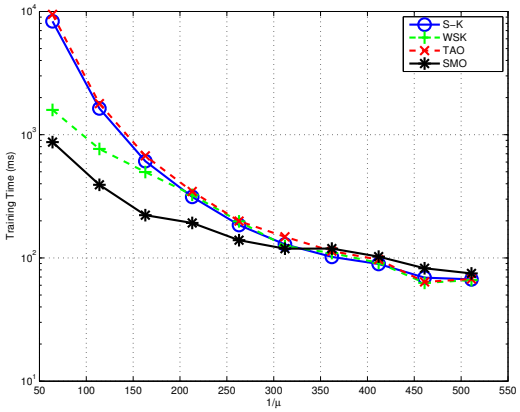
Figure C.5: Training times for the `heart` dataset



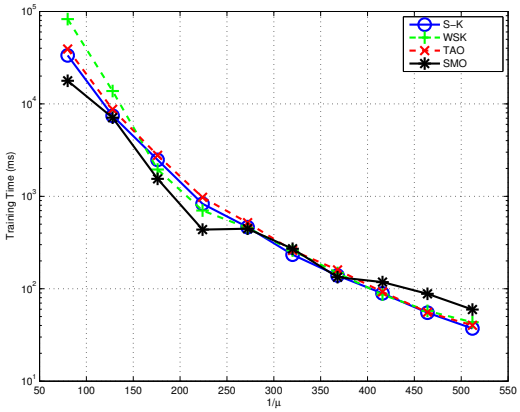
(a) Gaussian kernel with $\gamma = 0.01$



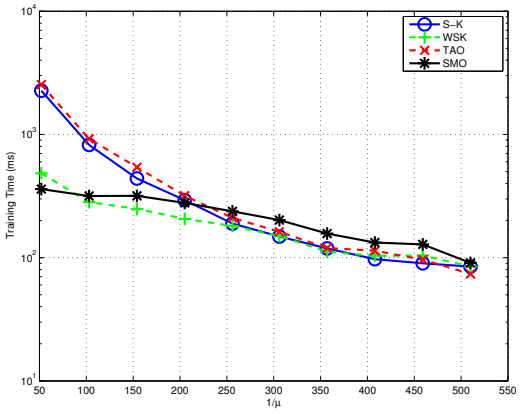
(b) linear kernel



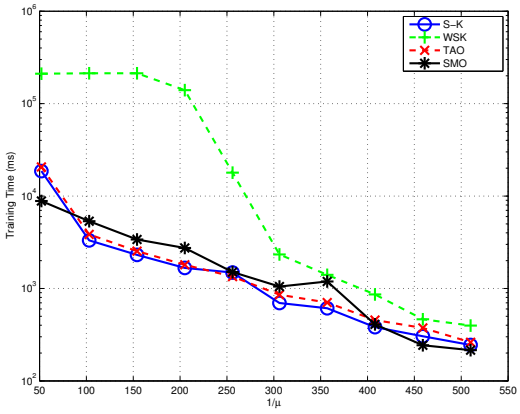
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

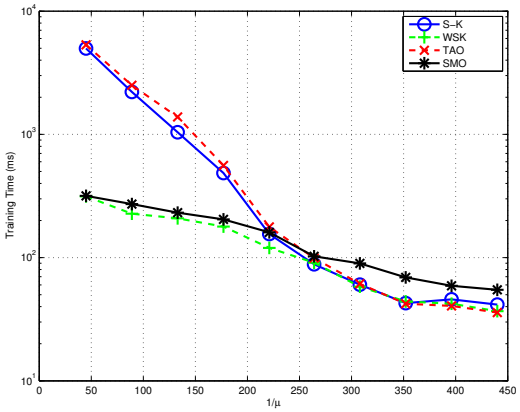


(e) Gaussian kernel with $\gamma = 1$

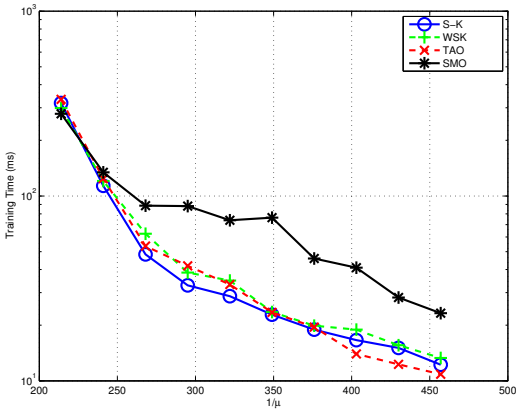


(f) polynomial kernel with $q = 4$

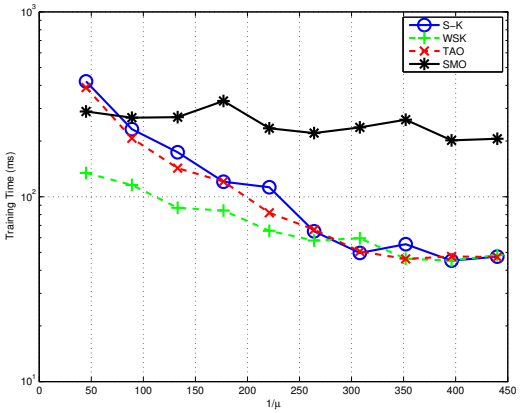
Figure C.6: Training times for the `image` dataset



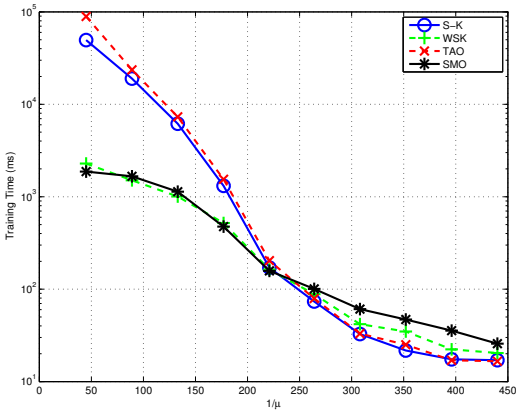
(a) Gaussian kernel with $\gamma = 0.01$



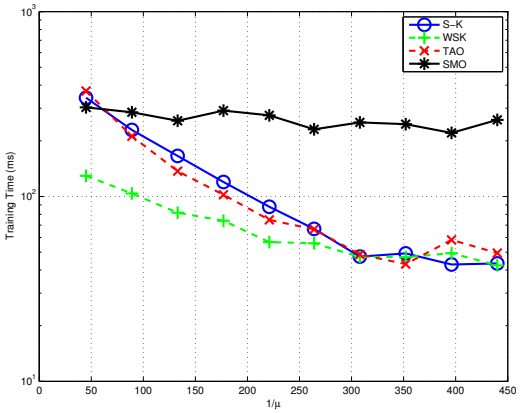
(b) linear kernel



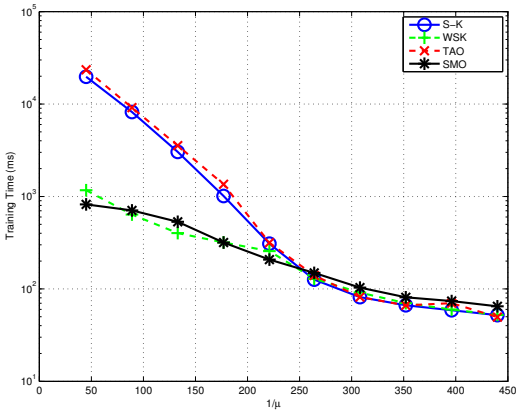
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

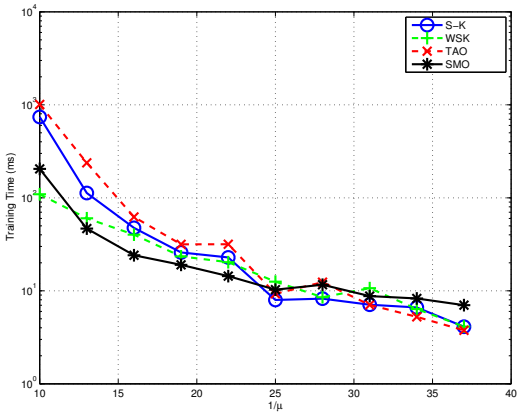


(e) Gaussian kernel with $\gamma = 1$

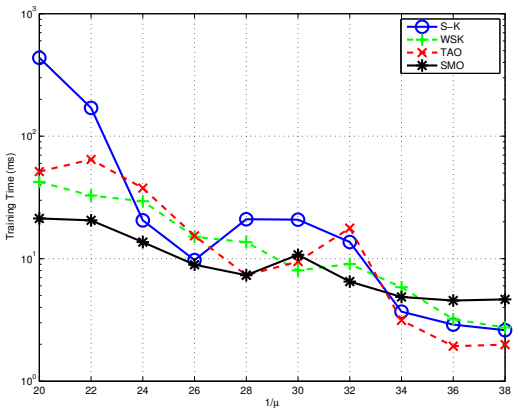


(f) polynomial kernel with $q = 4$

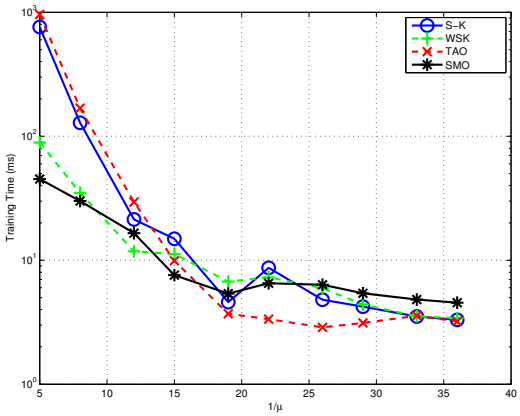
Figure C.7: Training times for the `splice` dataset



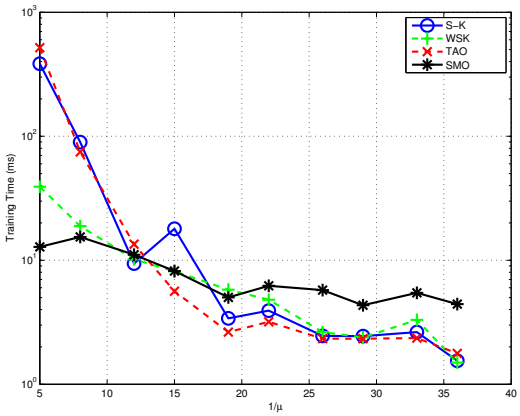
(a) Gaussian kernel with $\gamma = 0.01$



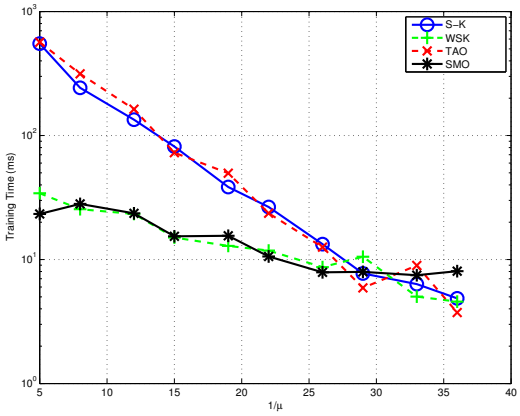
(b) linear kernel



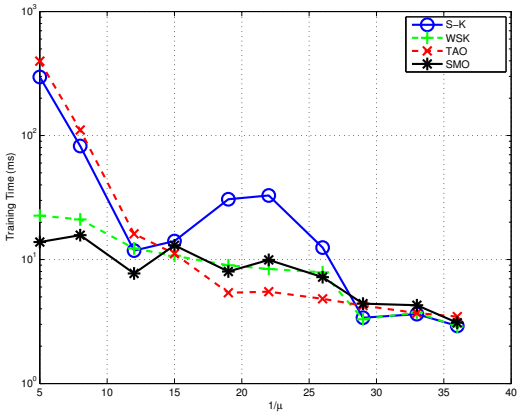
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

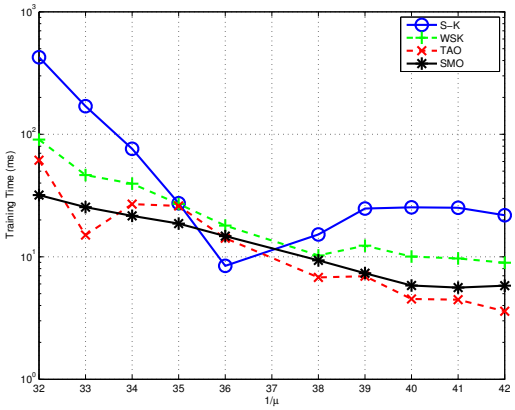


(e) Gaussian kernel with $\gamma = 1$

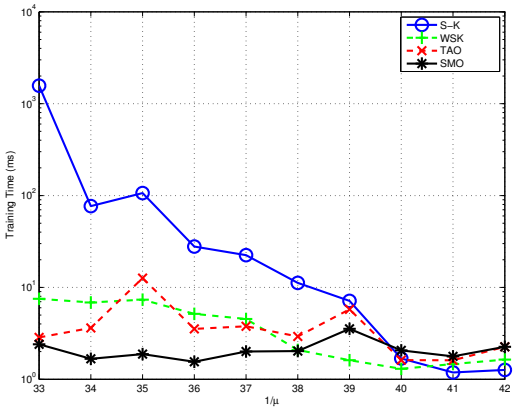


(f) polynomial kernel with $q = 4$

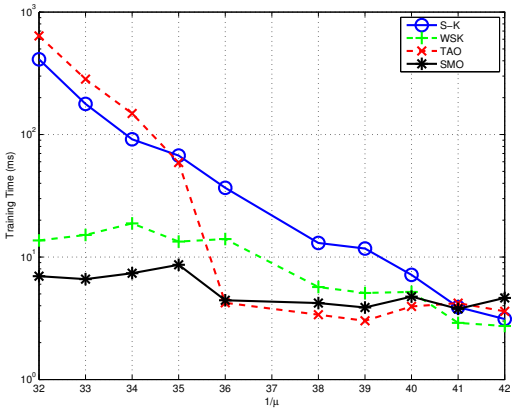
Figure C.8: Training times for the `thyroid` dataset



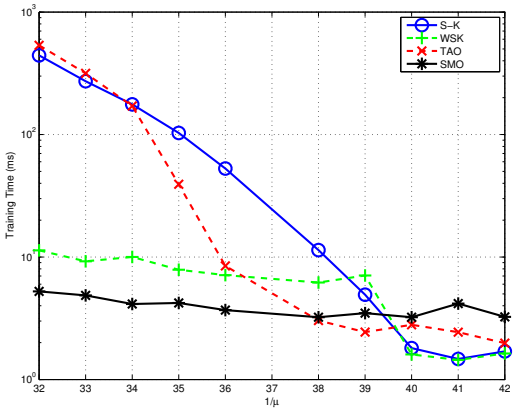
(a) Gaussian kernel with $\gamma = 0.01$



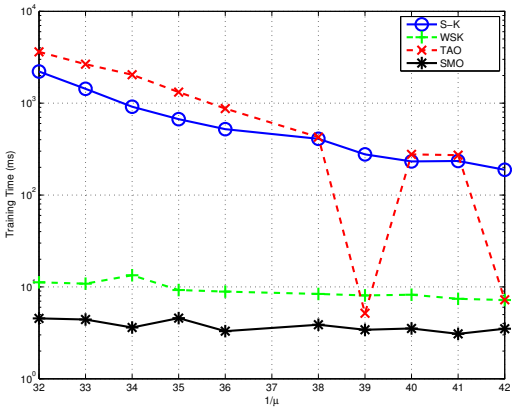
(b) linear kernel



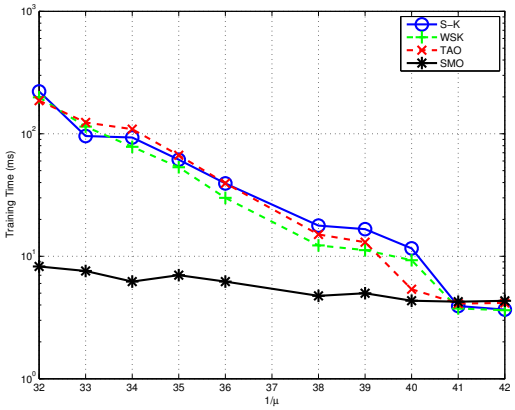
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$



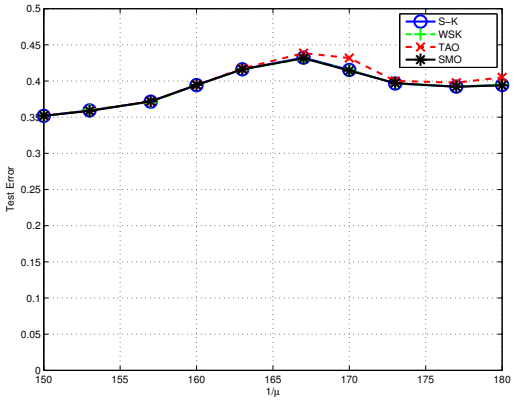
(e) Gaussian kernel with $\gamma = 1$



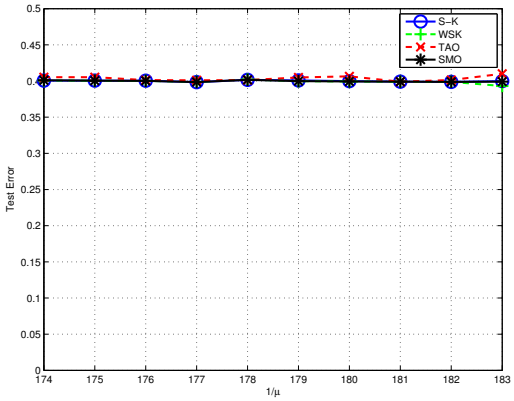
(f) polynomial kernel with $q = 4$

Figure C.9: Training times for the `titanic` dataset

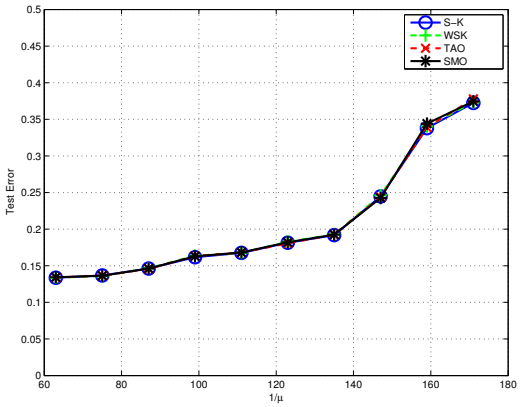
C.2 Error Rates



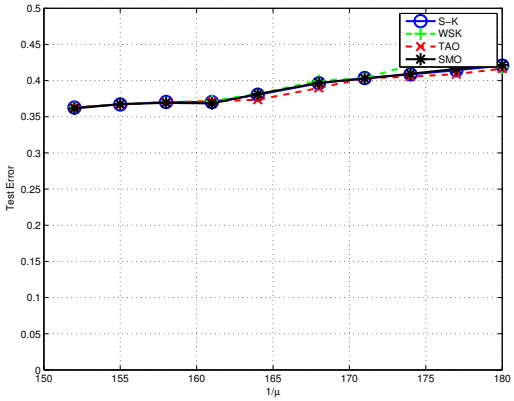
(a) Gaussian kernel with $\gamma = 0.01$



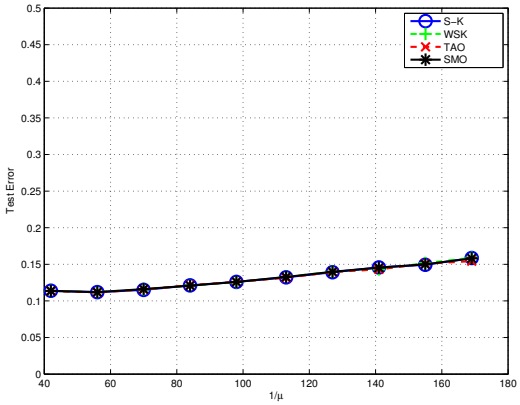
(b) linear kernel



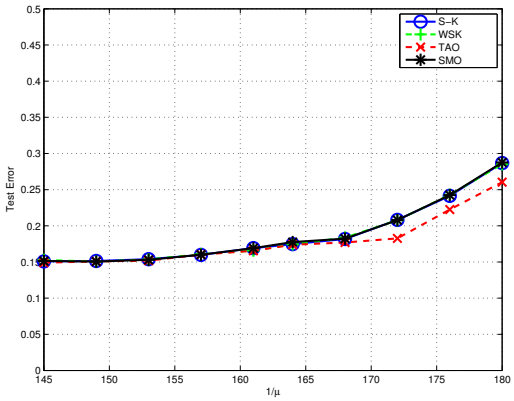
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

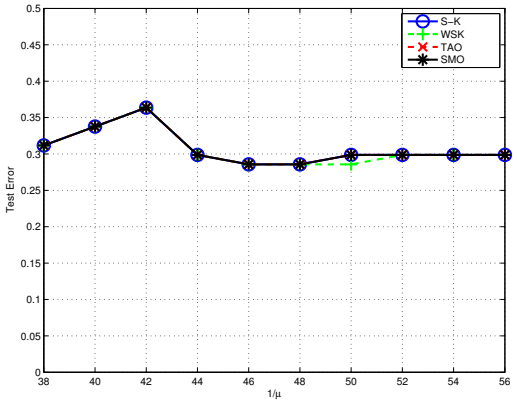


(e) Gaussian kernel with $\gamma = 1$

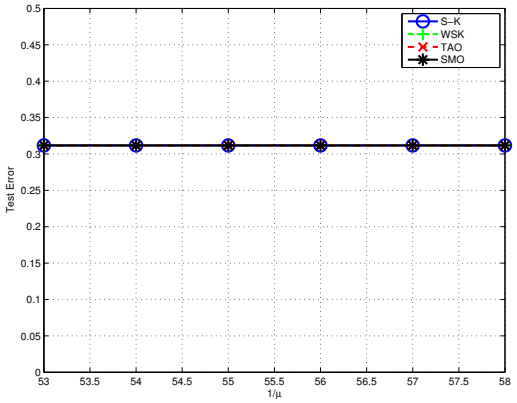


(f) polynomial kernel with $q = 4$

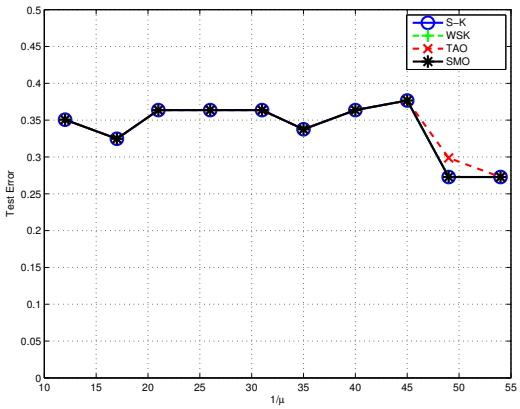
Figure C.10: Error rates for the `banana` dataset



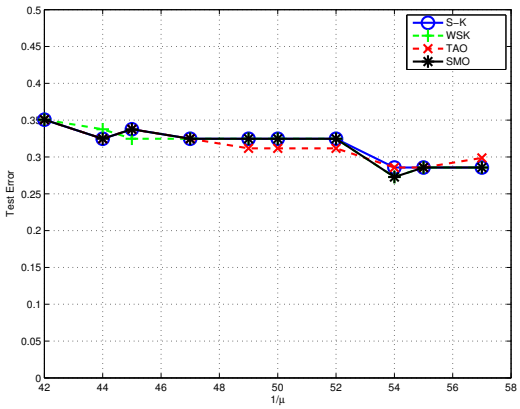
(a) Gaussian kernel with $\gamma = 0.01$



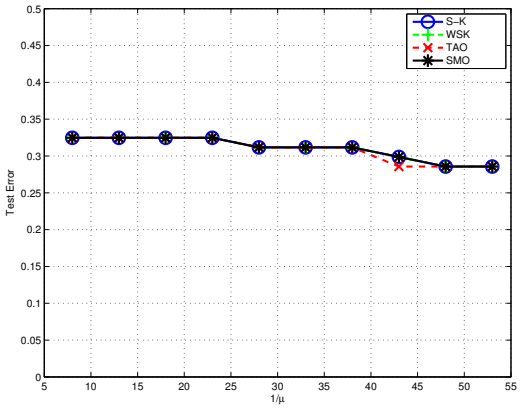
(b) linear kernel



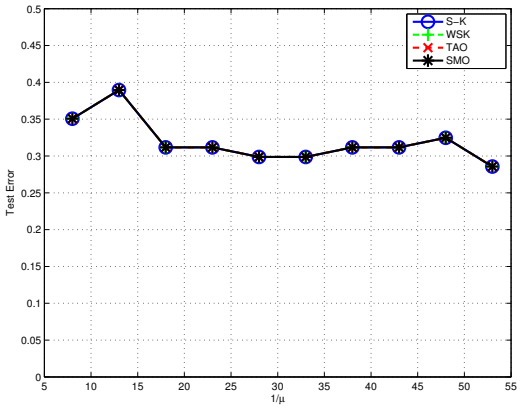
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

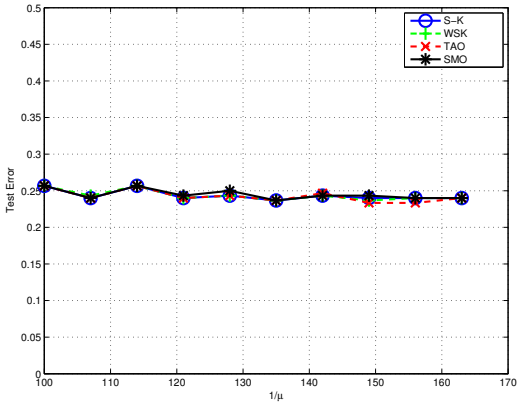


(e) Gaussian kernel with $\gamma = 1$

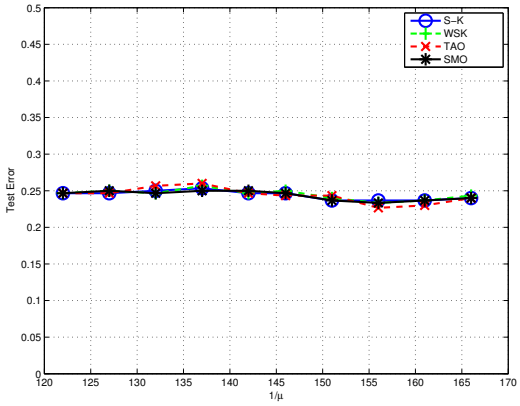


(f) polynomial kernel with $q = 4$

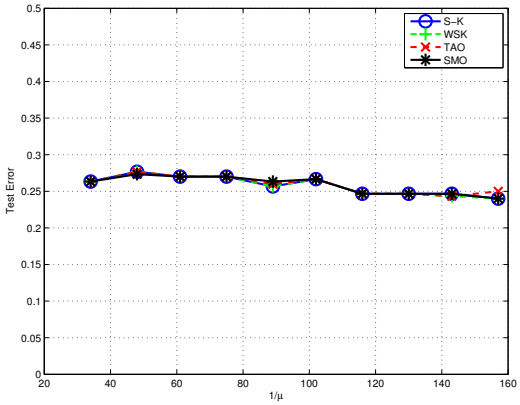
Figure C.11: Error rates for the `b.cancer` dataset



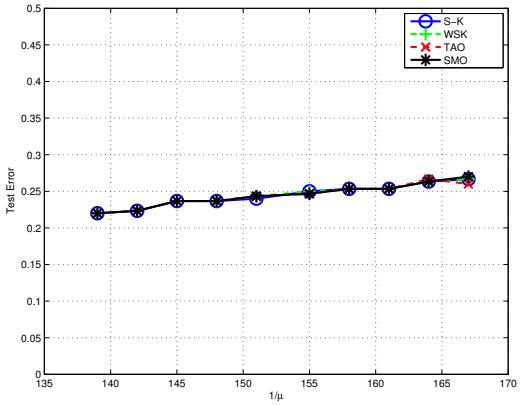
(a) Gaussian kernel with $\gamma = 0.01$



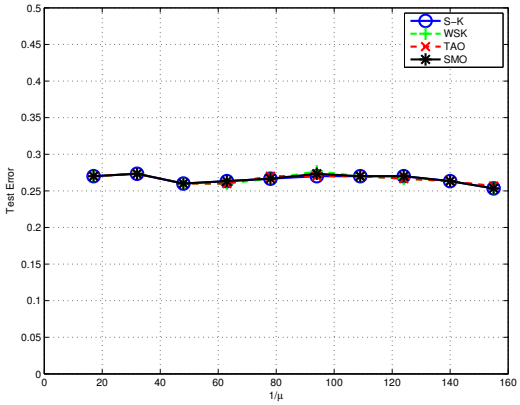
(b) linear kernel



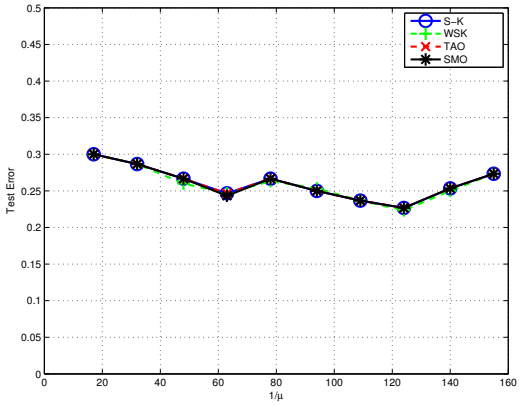
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

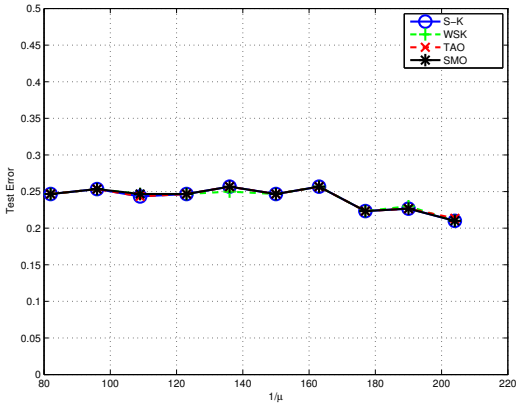


(e) Gaussian kernel with $\gamma = 1$

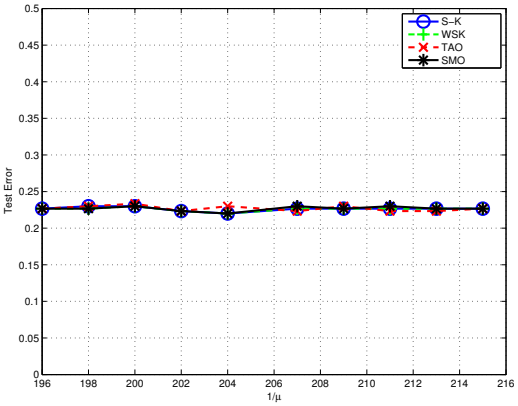


(f) polynomial kernel with $q = 4$

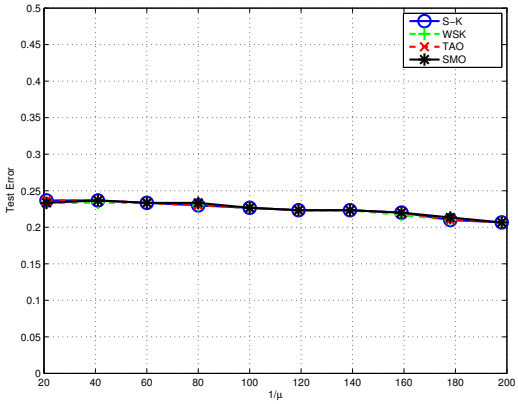
Figure C.12: Error rates for the `diabetes` dataset



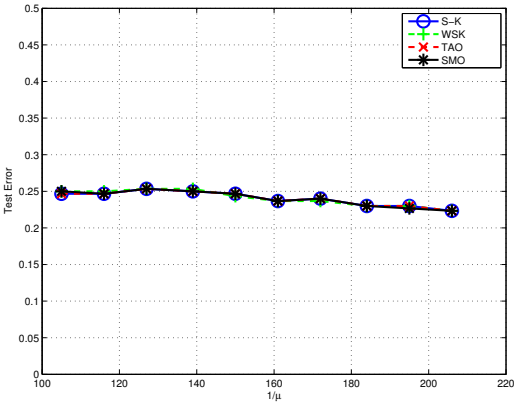
(a) Gaussian kernel with $\gamma = 0.01$



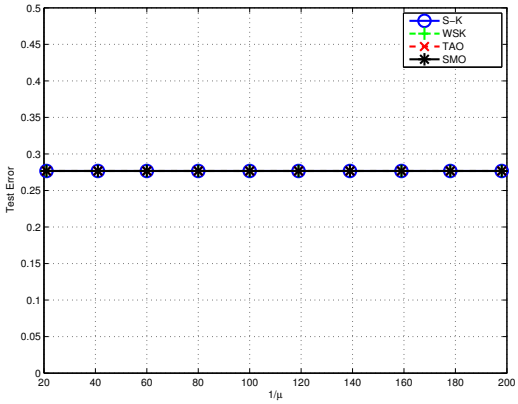
(b) linear kernel



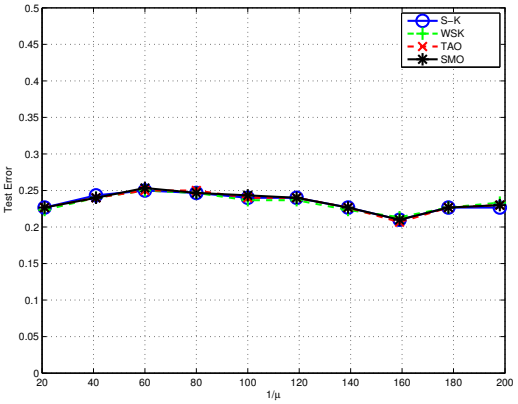
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

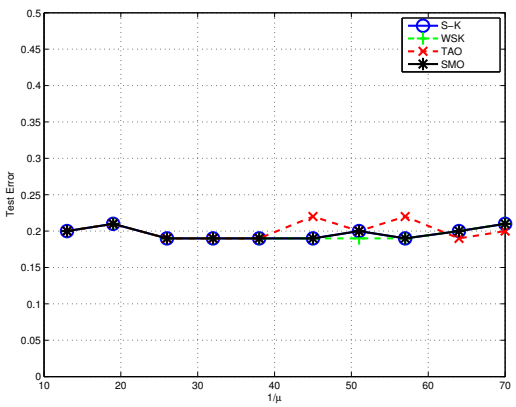


(e) Gaussian kernel with $\gamma = 1$

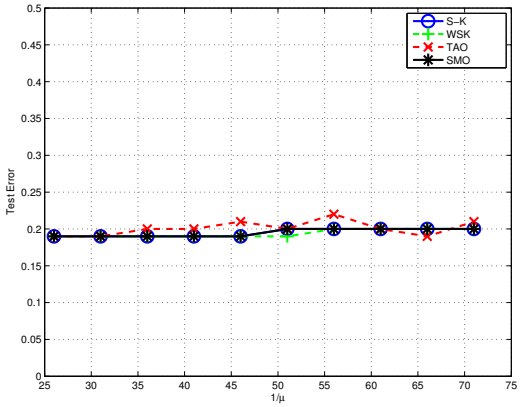


(f) polynomial kernel with $q = 4$

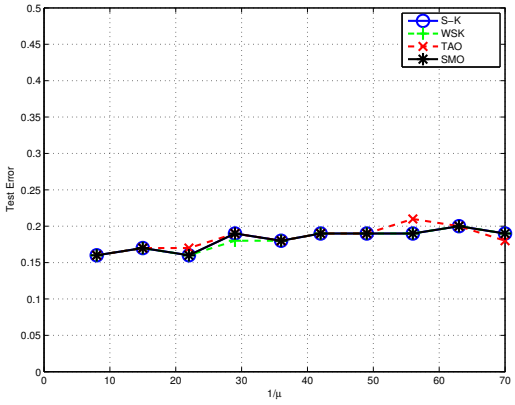
Figure C.13: Error rates for the `german` dataset



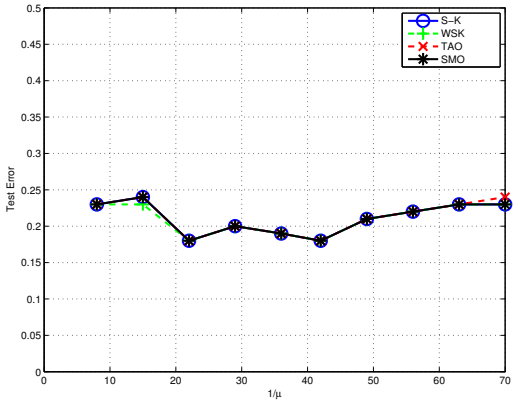
(a) Gaussian kernel with $\gamma = 0.01$



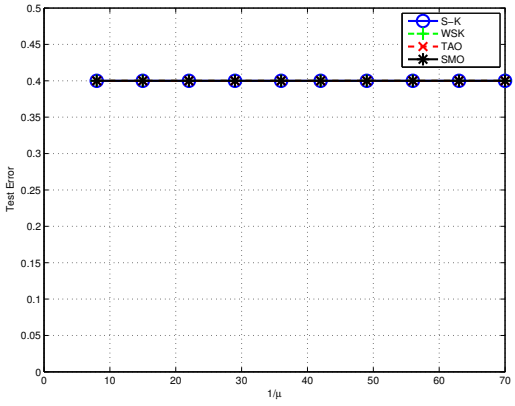
(b) linear kernel



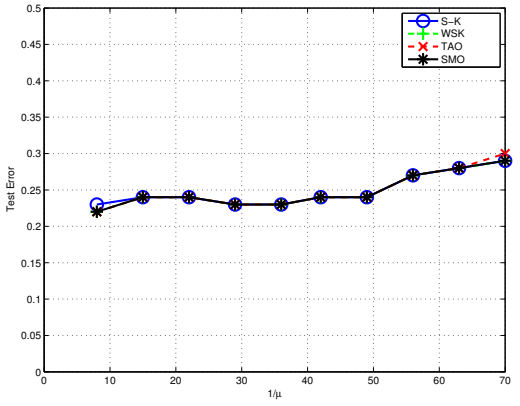
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

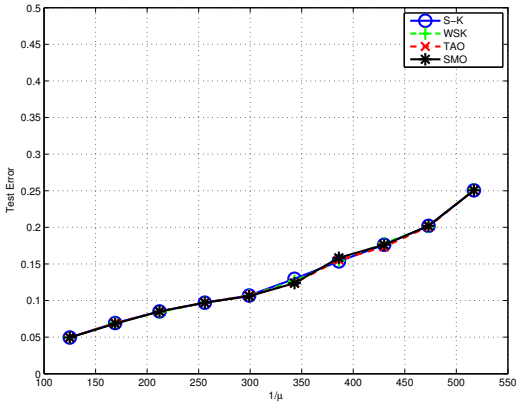


(e) Gaussian kernel with $\gamma = 1$

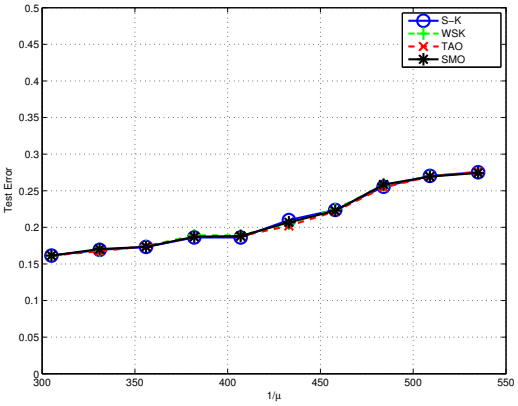


(f) polynomial kernel with $q = 4$

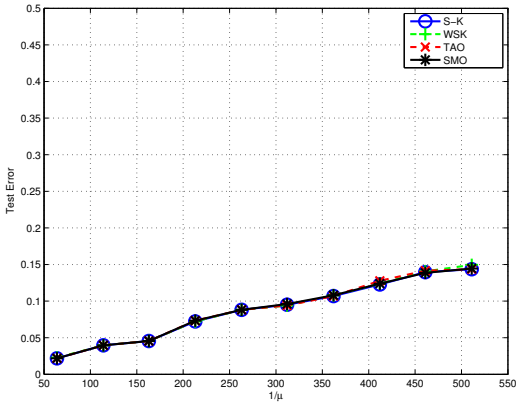
Figure C.14: Error rates for the `heart` dataset



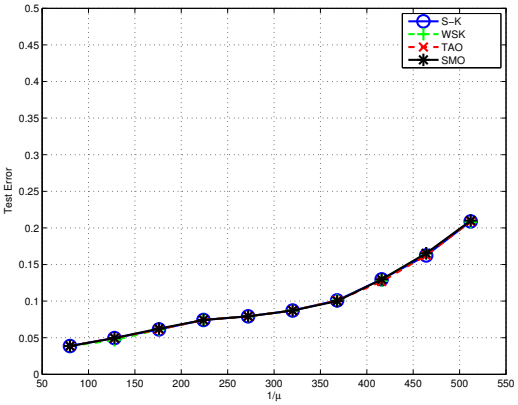
(a) Gaussian kernel with $\gamma = 0.01$



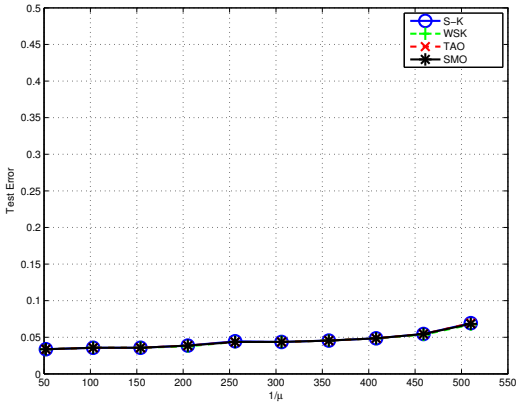
(b) linear kernel



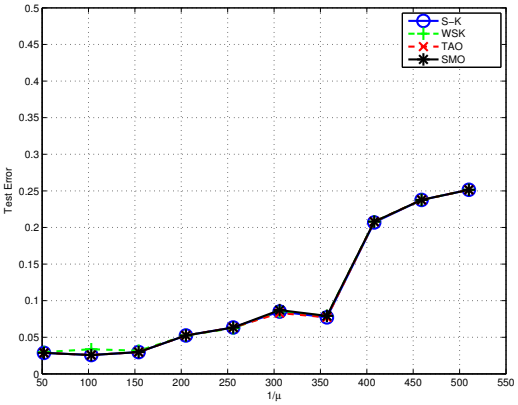
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

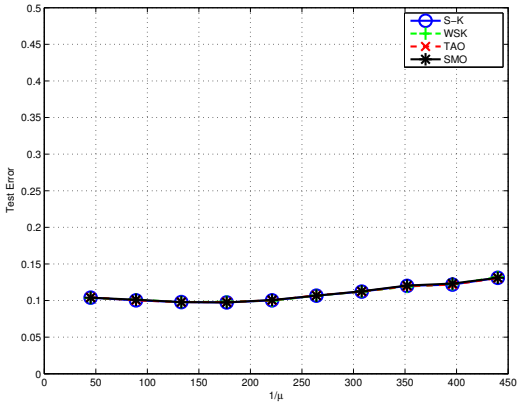


(e) Gaussian kernel with $\gamma = 1$

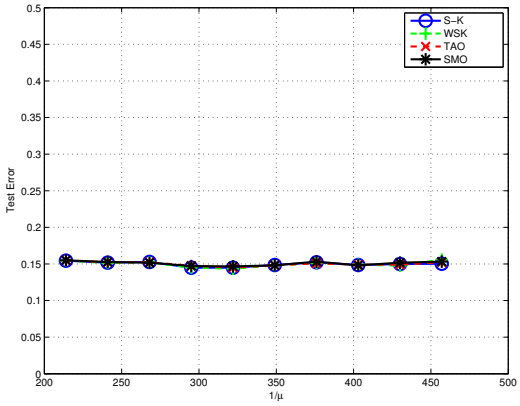


(f) polynomial kernel with $q = 4$

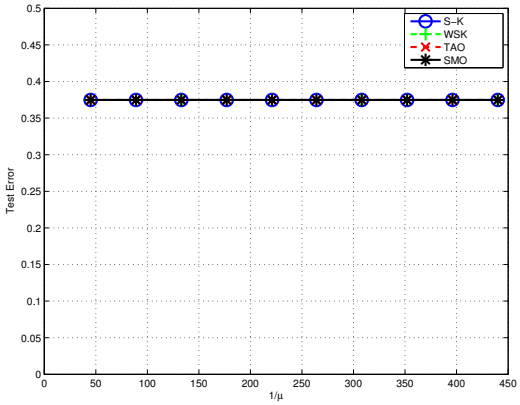
Figure C.15: Error rates for the `image` dataset



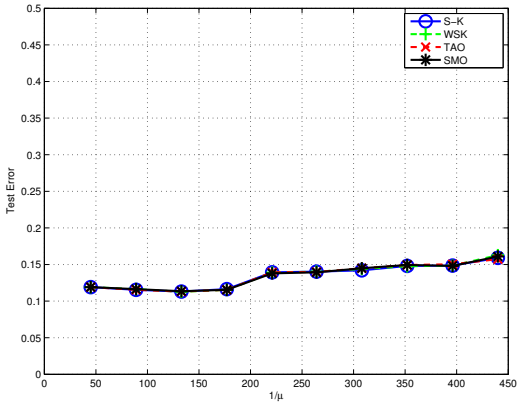
(a) Gaussian kernel with $\gamma = 0.01$



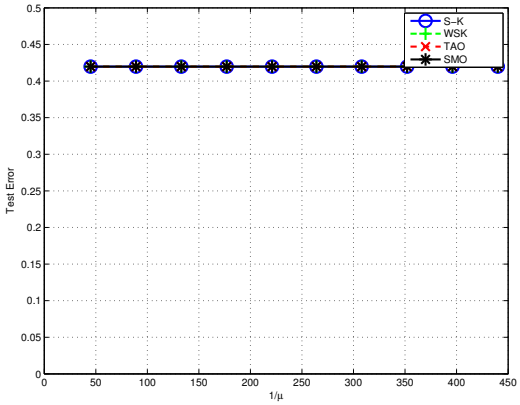
(b) linear kernel



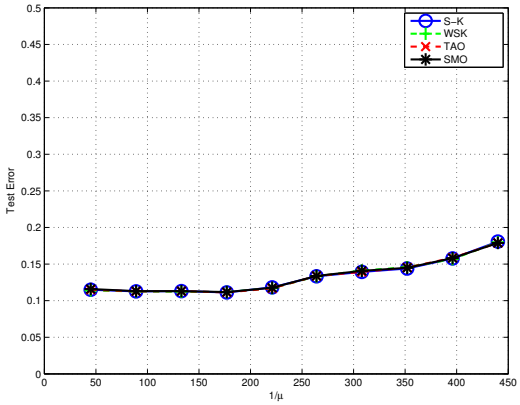
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

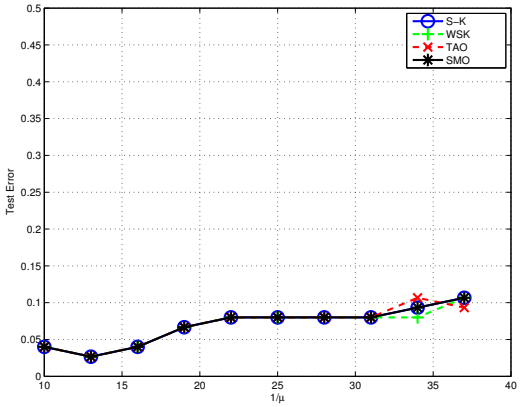


(e) Gaussian kernel with $\gamma = 1$

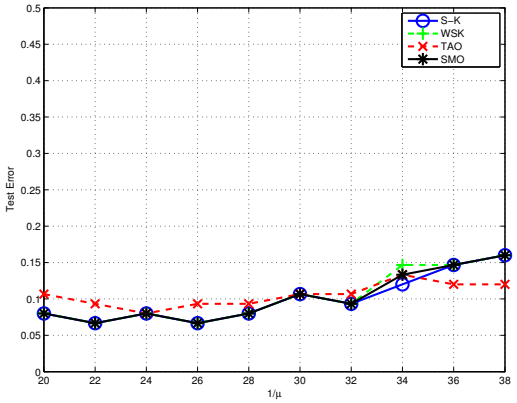


(f) polynomial kernel with $q = 4$

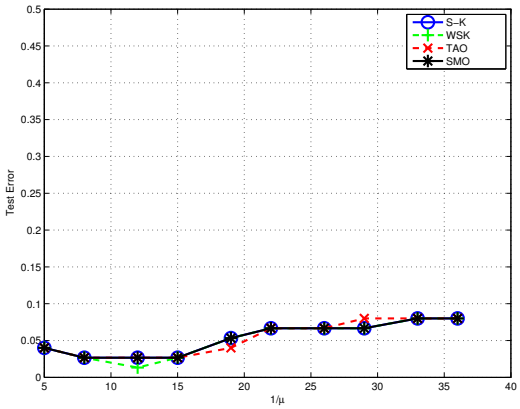
Figure C.16: Error rates for the `splice` dataset



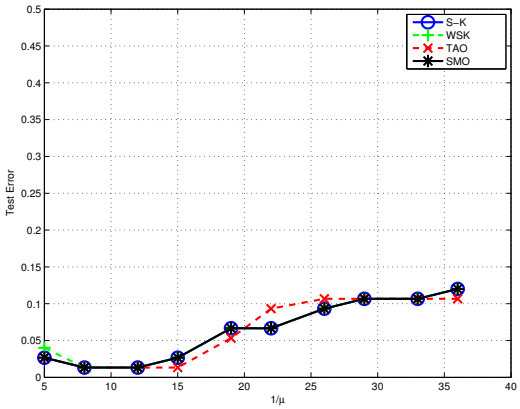
(a) Gaussian kernel with $\gamma = 0.01$



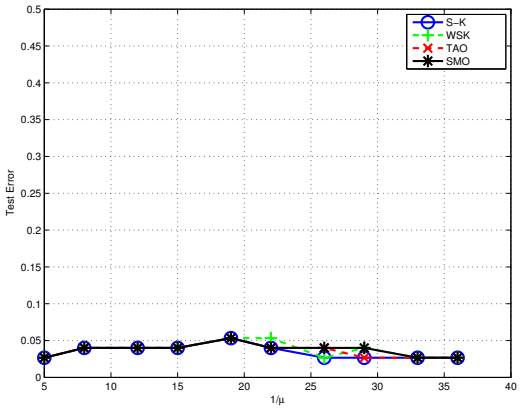
(b) linear kernel



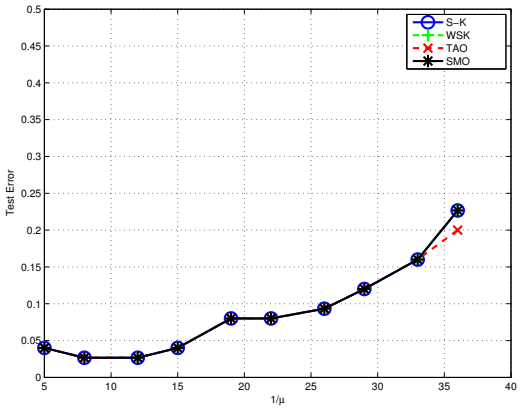
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$

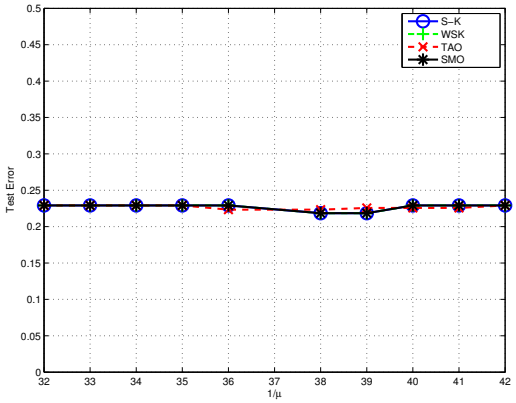


(e) Gaussian kernel with $\gamma = 1$

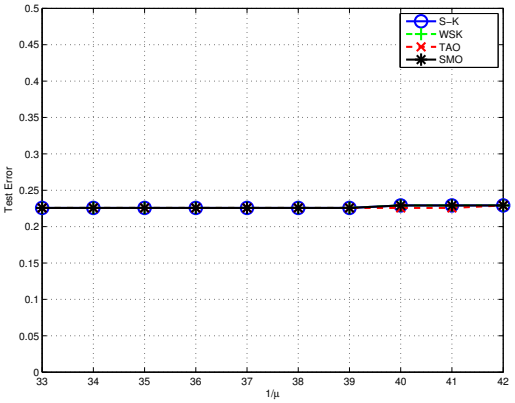


(f) polynomial kernel with $q = 4$

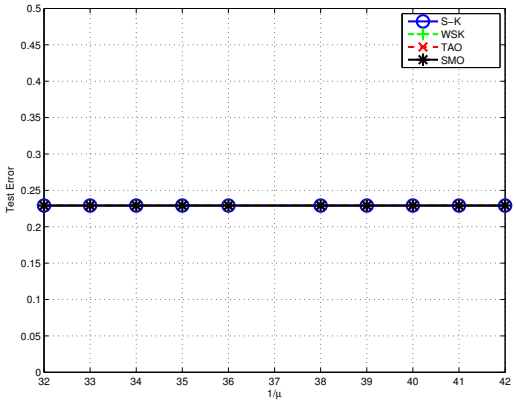
Figure C.17: Error rates for the `thyroid` dataset



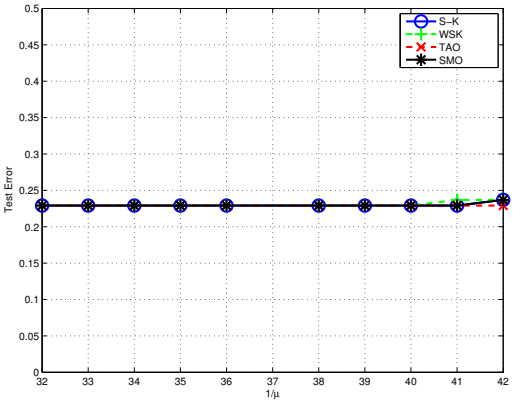
(a) Gaussian kernel with $\gamma = 0.01$



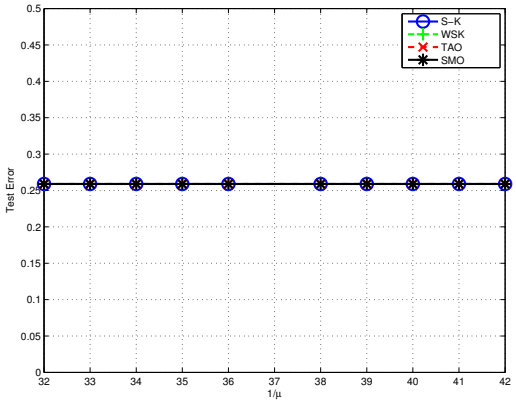
(b) linear kernel



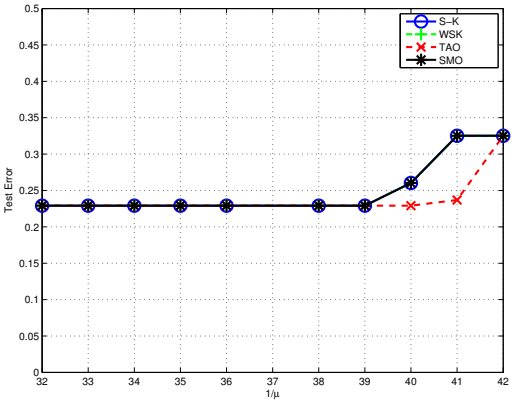
(c) Gaussian kernel with $\gamma = 0.1$



(d) polynomial kernel with $q = 2$



(e) Gaussian kernel with $\gamma = 1$



(f) polynomial kernel with $q = 4$

Figure C.18: Error rates for the `titanic` dataset

Vita

Publications arising from this thesis include:

Goodrich, B, Albrecht, D. and Tischer, P. (2009), Algorithms for the Computation of Reduced Convex Hulls. In *The 22nd Australasian Joint Conference on Artificial Intelligence.* Melbourne, Australia, pp. 230-239.

Goodrich, B, Albrecht, D. and Tischer, P. (2010), Accelerating Radius-Margin Parameter Selection for SVMs using Geometric Bounds, Clayton School of Information Technology, Monash University, 19pp. Technical Report 2010/258.

Goodrich, B, Albrecht, D. and Tischer, P. (2010), Accelerating Radius-Margin Parameter Selection for SVMs using Geometric Bounds. In *The 10th IEEE International Conference on Data Mining.* Sydney, Australia, pp. 827-832.

Goodrich, B, Albrecht, D. and Tischer, P. (2011), A Generalized Schlesinger-Kozinec Algorithm for Training Weighted SVMs. In *The 37th Annual Conference of the IEEE Industrial Electronics Society.* Melbourne, Australia, pp. 2360-2365.

Permanent Address: Clayton School of Information Technology
Monash University
Australia

This thesis was typeset with LaTeX 2$_\varepsilon$[1] by the author.

[1] LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Glenn Maughan and modified by Dean Thompson and David Squire of Monash University.

References

[1] A. Andrzejak. k-sets and j-facets – A tour of discrete geometry. Manuscript, 1997.

[2] A. Andrzejak and K. Fukuda. Optimization over k-set polytopes and efficient k-set enumeration. In *WADS '99: Proceedings of the 6th International Workshop on Algorithms and Data Structures*, pages 1–12, London, UK, 1999. Springer-Verlag.

[3] J. K. Anlauf and M. Biehl. The AdaTron: an adaptive perceptron algorithm. *Europhysics Letters*, 10(7):687–692, 1989.

[4] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[5] V. Barnett. The ordering of multivariate data. *Journal of the Royal Statistical Society*, 139(3):318–355, 1976.

[6] A. Ben-Hur, D. Horn, H. T. Siegelmann, and V. N. Vapnik. Support vector clustering. *Journal of Machine Learning Research*, 2:125–137, 2002.

[7] K. P. Bennett and E. J. Bredensteiner. Duality and geometry in SVM classifiers. In *ICML '00: Proceedings of the 17th International Conference on Machine Learning*, pages 57–64, San Francisco, CA, USA, 2000. Morgan Kaufmann.

[8] K. P. Bennett and C. Campbell. Support vector machines: hype or hallelujah? *SIGKDD Explorations Newsletter*, 2(2):1–13, December 2000.

[9] M. Bern and D. Eppstein. Optimization over zonotopes and training Support Vector Machines. In *Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pages 111–121. Springer Berlin, 2001.

[10] M. Bern, D. Eppstein, L. Guibas, J. Hershberger, S. Suri, and J. Wolter. The centroid of points with approximate weights. In P. Spirakis, editor, *Algorithms – ESA '95*, volume 979 of *Lecture Notes in Computer Science*, pages 460–472. Springer Berlin / Heidelberg, 1995.

[11] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In J. Langford and J. Pineau, editors, *ICML '12: Proceedings of the 29th International Conference on Machine Learning*, pages 1807–1814, New York, NY, USA, July 2012. Omnipress.

[12] J. A. Blackard. *Comparison of Neural Networks and Discriminant Analysis in Predicting Forest Cover Types*. PhD dissertation, Department of Forest Sciences, Colorado State University, Fort Collins, Colorado, 1998.

[13] B. E. Boser, I. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, New York, NY, USA, 1992. ACM.

[14] L. Bottou and C.-J. Lin. Support Vector Machine solvers. In L. Bottou, O. Chapelle, D. DeCost, and J. Weston, editors, *Large-scale kernel machines*, pages 1–27. MIT Press, 2007.

[15] L. Bottou, C. Cortes, J. Denker, H. Drucker, I. Guyon, L. Jackel, Y. LeCun, U. Muller, E. Sackinger, P. Simard, and V. Vapnik. Comparison of classifier methods: a case study in handwritten digit recognition. In *Proceedings of the 12th IAPR International Conference on Pattern Recognition: Conference B: Computer Vision Image Processing*, volume 2, pages 77–82, 1994.

[16] O. Bousquet and B. Schölkopf. Comment: Support Vector Machines with applications. *Statistical Science*, 21(3):337–340, 2006.

[17] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[18] C. J. C. Burges. Simplified support vector decision rules. In *ICML '96: Proceedings of the 13th International Conference on Machine Learning*, pages 71–78, July 1996.

[19] C. J. C. Burges. A tutorial on Support Vector Machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[20] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/ cjlin/libsvm`.

[21] O. Chapelle, V. N. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for Support Vector Machines. *Machine Learning*, 46(1-3):131–159, 2002.

[22] P.-H. Chen, R.-E. Fan, and C.-J. Lin. A study on SMO-type decomposition methods for Support Vector Machines. *IEEE Transactions on Neural Networks*, 17(4):893–908, 2006.

[23] W. Chu, S. S. Keerthi, and C. J. Ong. A general formulation for Support Vector Machines. In *ICONIP '02: Proceedings of the 9th International Conference on Neural Information Processing*, volume 5, pages 2522–2526, 2002.

[24] K. L. Clarkson. Applications of random sampling in computational geometry, II. In *SCG '88: Proceedings of the fourth annual symposium on Computational geometry*, pages 1–11, New York, NY, USA, 1988. ACM.

[25] R. Cole, M. Sharir, and C. K. Yap. On k-hulls and related problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 154–166, New York, NY, USA, 1984. ACM.

[26] R. Collobert, S. Bengio, and Y. Bengio. A parallel mixture of svms for very large scale problems. *Neural Computation*, 14(5):1105–1114, 2002.

[27] C. Cortes and V. N. Vapnik. Support-vector networks. *Machine Learning*, 20(3): 273–297, 1995.

[28] D. J. Crisp and C. J. C. Burges. A geometric interpretation of ν-SVM classifiers. In S. A. Solla, T. K. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 244–251. MIT Press, 2000.

[29] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge University Press, 2000.

[30] N. Cristianini, C. Campbell, and J. Shawe-Taylor. Dynamically adapting kernels in Support Vector Machines. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 204–210. MIT Press, 1999.

[31] T. K. Dey. Improved bounds on planar k-sets and k-levels. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 156–, Washington, DC, USA, 1997. IEEE Computer Society.

[32] H. Drucker, D. Wu, and V. N. Vapnik. Support Vector Machines for spam categorization. *Neural Networks, IEEE Transactions on*, 10(5):1048–1054, 1999.

[33] K. B. Duan, S. S. Keerthi, and A. N. Poo. Evaluation of simple performance measures for tuning SVM hyperparameters. *Neurocomputing*, 51:41–59, April 2003.

[34] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2001.

[35] W. F. Eddy. A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, 3(4):398–403, 1977.

[36] W. F. Eddy and J. A. Hartigan. Uniform convergence of the empirical distribution function over convex sets. *The Annals of Statistics*, 5(2):370–374, 1977.

[37] R.-E. Fan, P.-H. Chen, and C.-J. Lin. Working set selection using second order information for training Support Vector Machines. *Journal of Machine Learning Research*, 6:1889–1918, December 2005.

[38] R. Fletcher. *Practical methods of optimization*. Wiley-Interscience, New York, NY, USA, 2nd edition, 1987.

[39] V. Franc. *Optimization Algorithms for Kernel Methods*. PhD dissertation, Center for Machine Perception, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University, Prague, 2005.

[40] V. Franc and V. Hlavàč. An iterative algorithm learning the maximal margin classifier. *Pattern Recognition*, 36(9):1985–1996, 2003.

[41] V. Franc, P. Laskov, and K.-R. Müllur. Stopping conditions for exact computation of Leave-One-Out error in Support Vector Machines. In *ICML '08: Proceedings of the 25th International Conference on Machine Learning*, pages 328–335, New York, NY, USA, 2008. ACM.

[42] A. Frank and A. Asuncion. UCI machine learning repository, 2010. URL `http://archive.ics.uci.edu/ml`.

[43] T.-T. Frieß, N. Cristianini, and C. Campbell. The kernel-adatron algorithm: a fast and simple learning procedure for support vector machines. In *Machine Learning: Proceedings of the Fifteenth International Conference*. Morgan Kaufmann, 1998.

[44] E. G. Gilbert. An iterative procedure for computing the minimum of a quadratic form on a convex set. *SIAM Journal of Control*, 4:61–79, 1966.

[45] T. Glasmachers and C. Igel. Maximum-gain working set selection for SVMs. *Journal of Machine Learning Research*, 7:1437–1466, December 2006.

[46] C. Gold and P. Sollich. Model selection for Support Vector Machine classification. *Neurocomputing*, 55:221–249, September 2003.

[47] B. Goodrich, D. W. Albrecht, and P. Tischer. Algorithms for the computation of Reduced Convex Hulls. In A. Nicholson and X. Li, editors, *AI 2009: Advances in Artificial Intelligence*, volume 5866 of *Lecture Notes in Computer Science*, pages 230–239. Springer Berlin / Heidelberg, 2009.

[48] B. Goodrich, D. W. Albrecht, and P. Tischer. Accelerating radius-margin parameter selection for SVMs using geometric bounds. In *ICDM '10: Proceedings of the 2010 IEEE International Conference on Data Mining*, pages 827–832. IEEE Computer Society, 2010.

[49] B. Goodrich, D. W. Albrecht, and P. Tischer. Training weighted SVMs using a generalized Schlesinger-Kozinec algorithm. In *IECON '11: Proceedings of the 37th Annual Conference of the IEEE Industrial Electronics Society*, pages 2360–2365. IEEE Computer Society, 2011.

[50] G. Guo, S. Z. Li, and K. Chan. Face recognition by Support Vector Machines. In *Proceedings of the Fourth IEEE International Conference on Automatic Face and Gesture Recognition*, pages 196 –201, 2000.

[51] I. Guyon. SVM application list, 2011. URL `http://www.clopinet.com/isabelle/Projects/SVM/applist.html`.

[52] S. Haykin. *Neural Networks and Learning Machines*. Prentice Hall, 3rd edition, 2009.

[53] V. Hodge and J. Austin. A survery of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[54] T. M. Huan and V. Kecman. Bias term b in SVMs again. In *ESANN '04: Proceedings of the 12th European Symposium on Artificial Neural networks*, pages 441–448, 2004.

[55] G.-B. Huang, X. Ding, and H. Zhou. Optimization method based extreme learning machine for classification. *Neurocomputing*, 74(1-3):155–163, 2010.

[56] G.-B. Huang, D. Wang, and Y. Lan. Extreme learning machines: a survey. *International Journal of Machine Learning and Cybernetics*, 2:107–122, 2011.

[57] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang. Extreme learning machine for regression and multiclass classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 42(2):513–529, 2012.

[58] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods: Support Vector Learning*, pages 169–184. MIT Press, 1999.

[59] T. Joachims. Estimating the generalization performance of an SVM efficiently. In P. Langley, editor, *ICML '00: Proceedings of the 17th International Conference on Machine Learning*, pages 431–438, July 2000.

[60] T. Joachims. Training linear SVMs in linear time. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 217–226, New York, NY, USA, 2006. ACM.

[61] M. Kallay. Convex hull algorithms in higher dimensions, 1981. Unpublished manuscript, Department of Mathematics, University of Oklahoma.

[62] V. Kecman, M. Vogt, and T. M. Huan. On the equality of kernel AdaTron and Sequential Minimal Optimization in classification and regression tasks and alike algorithms for kernel machines. In *ESANN '03: Proceedings of the 11th European Symposium on Artificial Neural Networks*, pages 215–222, 2003.

[63] S. S. Keerthi. Efficient tuning of SVM hyperparameters using radius/margin bound and iterative algorithms. *IEEE Transactions on Neural Networks*, 13(5):1225–1229, 2002.

[64] S. S. Keerthi and C.-J. Lin. Asymptotic behaviors of support vector machines with gaussian kernel. *Neural Computation*, 15(7):1667–1689, 2003.

[65] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. A fast iterative nearest point algorithm for Support Vector Machine classifier design. *IEEE Transactions on Neural Networks*, 11(1):124–136, 2000.

[66] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt's SMO algorithm for SVM classifier design. *Neural Computation*, 13(3):637–649, 2001.

[67] S. S. Keerthi, K. B. Duan, S. K. Shevade, and A. N. Poo. A fast dual algorithm for kernel logistic regression. *Machine Learning*, 61(1-3):151–165, 2005.

[68] R. Khardon and G. Wachman. Noise tolerant variants of the perceptron algorithm. *Journal of Machine Learning Research*, 8:227–248, 2007.

[69] G. Kimeldorf and G. Wahba. Some results on Tchebycheffian spline functions. *Journal of Mathematical Analysis and Applications*, 33(1):82 – 95, 1971.

[70] V. Klee. Convex polytopes and linear programming. In *IBM Scientific Computing Symposium: Combinatorial Problems*, pages 123–158, Armonk, N.Y., 1966. IBM.

[71] A. Kowalczyk. Maximal margin perceptron. In A. J. Smola, P. L. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 75–114. MIT Press, 2000.

[72] B. N. Kozinec. Rekurentnyj algoritm razdelenia vypuklych obolochek dvuch mnozhestv; in Russian (recurrent algorithm separating convex hulls of two sets). In V. N. Vapnik, editor, *Algoritmy obushenia raspoznavania (Learning algorithms in pattern recognition)*, pages 43–50. Sovetskoje radio, Moskva, 1973.

[73] W. Krauth and M. Mézard. Learning algorithms with optimal stability in neural networks. *Journal of Physics A: Mathematical and General*, 20(11):745–752, 1987.

[74] C.-F. Lin and S.-D. Wang. Fuzzy Support Vector Machines. *IEEE Transactions on Neural Networks*, 13(2):464–471, 2002.

[75] Y. Lin, G. Wahba, H. Zhang, and Y. Lee. Statistical properties and adaptive tuning of Support Vector Machines. *Machine Learning*, 48(1-3):115–136, July 2002.

[76] G. Loosli and S. Canu. Comments on the "Core Vector Machines: Fast SVM training on very large data sets". *Journal of Machine Learning Research*, 8:291–301, May 2007.

[77] J. López, A. Barbero, and J. Dorronsoro. On the equivalence of the SMO and MDM algorithms for SVM training. In *ECML PKDD '08: Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I*, pages 288–300. Springer-Verlag, 2008.

[78] J. López, Á. Barbero, and J. Dorronsoro. Simple clipping algorithms for reduced convex hull SVM training. In E. Corchado, A. Abraham, and W. Pedrycz, editors, *Hybrid Artificial Intelligence Systems*, volume 5271 of *Lecture Notes in Computer Science*, pages 369–377. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-87656-4_46.

[79] O. L. Mangasarian and D. R. Musicant. Lagrangian support vector machines. *Journal of Machine Learning Research*, 1:161–177, September 2001.

[80] S. Martin. Training Support Vector Machines using Gilbert's algorithm. In *ICDM '05: Proceedings of the Fifth IEEE International Conference on Data Mining*, page 8pp, 2005.

[81] M. E. Mavroforakis. *Geometric Approach to Statistical Learning Theory Through Support Vector Machines (SVM) With Application to Medical Diagnosis*. PhD dissertation, School of Science, Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece, 2008.

[82] M. E. Mavroforakis and S. Theodoridis. A geometric approach to Support Vector Machine (SVM) classification. *IEEE Transactions on Neural Networks*, 17(3):671–682, 2006.

[83] M. E. Mavroforakis, M. Sdralis, and S. Theodoridis. A novel SVM geometric algorithm based on reduced convex hulls. In *18th International Conference on Pattern Recognition*, volume 2, pages 564–568, 2006.

[84] B. F. Mitchell, V. F. Dem'yanov, and V. N. Malozemov. Finding the point of a polyhedron closest to the origin. *SIAM Journal on Control*, 12(1):19–26, 1974.

[85] J. M. Moguerza and A. Muñoz. Support vector machines with applications. *Statistical Science*, 21(3):322–336, 2006.

[86] K.-R. Müller, S. Mika, G. Rätsch, K. Tsunda, and B. Schölkopf. An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12(2): 181–201, March 2001.

[87] M. Opper. Learning times of neural networks: Exact solution for a perceptron algorithm. *Physical Review A*, 38(7):3824–3826, Oct 1988.

[88] W. Oraiby and D. Schmitt. Divide and conquer method for k-set polygons. In H. Ito, M. Kano, N. Katoh, and Y. Uno, editors, *Computational Geometry and Graph Theory*, pages 166–177. Springer-Verlag, Berlin, Heidelberg, 2008.

[89] J. O'Rourke. *Computation Geometry in C*. Cambridge University Press, 2nd edition, 1998.

[90] R. L. Ortman, K. Venayagamoorthy, and S. M. Potter. Input separability in Living Liquid State Machines. In *ICANNGA '11: Proceedings of the 10th International Conference on Adaptive and Natural Computing Algorithms - Part I*, pages 220–229, Berlin, Heidelberg, 2011. Springer-Verlag.

[91] J. C. Platt. Fast training of Support Vector Machines using Sequential Minimal Optimization. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods: Support Vector Learning*, pages 185–208. MIT Press, 1998.

[92] J. C. Platt. Probabilities for SV Machines. In A. J. Smola, P. L. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 61–71. MIT Press, 2000.

[93] K. Polat and S. Güneş. Breast cancer diagnosis using least square Support Vector Machine. *Digital Signal Processing*, 17:694–701, 2007.

[94] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 2nd edition, 1988.

[95] K. L. Priddy and P. E. Keller. *Artificial Neural Networks: An Introduction*. SPIE Press, 2005.

[96] G. Rätsch, T. Onoda, and K.-R. Müller. Soft margins for AdaBoost. *Machine Learning*, 42(3):287–320, 2001.

[97] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[98] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Spartan Books, Washington, 1962.

[99] P. J. Rousseeuw and A. M. Leroy. *Robust Regression and Outlier Detection*. Wiley, 1987.

[100] M. I. Schlesinger, V. G. Kalmykov, and A. A. Suchorukov. Sravnitelnyj analiz algoritmov. sinteza linejnogo reshajushchego pravila dlja proverki slozhnych gipotez; in Russian (comparative analysis of algorithms synthesising linear decision rule for analysis of complex hypothesis). *Automatika*, 1:3–9, 1981.

[101] P. J. Schneider and D. H. Eberly. *Geometric tools for computer graphics*. Morgan Kaufmann, 2003.

[102] B. Schölkopf. Statistical learning and kernel methods. Technical Report MSR-TR-2000-23, Microsoft Research, 2000.

[103] B. Schölkopf, C. J. C. Burges, and V. N. Vapnik. Extracting support data for a given task. In U. M. Fayyad and R. Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1995.

[104] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13 (7), 2001.

[105] V. S. Sheng, F. Provost, and P. G. Ipeirotis. Get another label? Improving data quality and data mining using multiple, noisy labelers. In *KDD '08: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 614–622, New York, NY, USA, 2008. ACM.

[106] M. Sordo and Q. Zeng. On sample size and classification accuracy: A performance comparison. In J. Oliveira, V. Maojo, F. Martín-Sánchez, and A. Pereira, editors, *Biological and Medical Data Analysis*, volume 3745 of *Lecture Notes in Computer Science*, pages 193–201. Springer Berlin / Heidelberg, 2005.

[107] P. J. Tan and D. L. Dowe. MML inference of oblique decision trees. In G. Webb and X. Yu, editors, *AI 2004: Advances in Artificial Intelligence*, pages 1082–1088. Springer, 2004.

[108] Q. Tao, G.-W. Wu, and J. Wang. A generalized S-K algorithm for learning ν-SVM classifiers. *Pattern Recognition Letters*, 25(10):1165–1171, 2004.

[109] Q. Tao, G.-W. Wu, and J. Wang. A general soft method for learning SVM classifiers with l_1-norm penalty. *Pattern Recognition*, 41(3):939–948, 2008.

[110] D. M. J. Tax and T. P. W. Duin. Support vector domain description. *Pattern Recognition Letters*, 20:1191–1199, 1999.

[111] S. Theodoridis, A. Pikrakis, K. Koutroumbas, and D. Cavouras. *Introduction to Pattern Recognition: A Matlab Approach*. Academic Press, 2010.

[112] I. W. Tsang and J. T. Kwok. Large-scale sparsified manifold regularization. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 1401–1408. MIT Press, Cambridge, MA, 2007.

[113] I. W. Tsang, J. T. Kwok, and P.-M. Cheung. Core vector machines: Fast SVM training on very large data sets. *Journal of Machine Learning Research*, 6:363–392, 2005.

[114] I. W. Tsang, J. T. Kwok, and J. M. Zurada. Generalized Core Vector Machines. *IEEE Transactions on Neural Networks*, 17:1126–1140, 2006.

[115] I. W. Tsang, A. Kocsor, and J. T. Kwok. Simpler Core Vector Machines with enclosing balls. In *ICML '07: Proceedings of the 24th International Conference on Machine Learning*, pages 911–918, New York, NY, USA, 2007. ACM.

[116] V. N. Vapnik. *Statistical Learning Theory*. John Wiley & Sons, 1998.

[117] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, New York, USA, 2000.

[118] K. Veropoulos, C. Campbell, and N. Cristianini. Controlling the sensitivity of Support Vector Machines. In *IJCAI '99: Proceedings of the 16th International Joint Conference on Artificial intelligence*, pages 55–60, Stockholm, Sweden, 1999. Morgan Kaufmann.

[119] M. Vogt. SMO algorithms for Support Vector Machines without bias term, 2002. Institute Report, Institute of Automatic Control, Technische Universität Darmstadt, Germany.

[120] V. Vural, G. Fung, J. G. Dy, and B. Rao. Fast semi-supervised SVM classifiers using a priori metric information. *Optimization Methods Software*, 23(4):521–532, 2008.

[121] U. Wagner. k-sets and $k-$-facets. In J. E. Goodman, J. Patch, and R. Pollack, editors, *Surveys on Discrete and Computational Geometry: Twenty Years Later*, volume 453 of *Contemporary Mathematics*, pages 443–514. American Mathematical Society, 2008.

[122] G. Wahba. Support Vector Machines, Reproducing Kernel Hilbert Spaces and the Randomized GACV. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods: Support Vector Learning*, pages 69–88. MIT Press, 1999.

[123] G. Wahba, X. Lin, F. Gao, D. Xiang, R. Klein, and B. Klein. The bias-variance tradeoff and the randomized GACV. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 620–626. MIT Press, 1999.

[124] C. S. Wallace. *Statistical and Inductive Inference by Minimum Message Length*. Springer, New York, NY, USA, 2005.

[125] C. S. Wallace and D. M. Boulton. An information measure for classification. *Computer Journal*, 11(2):185–194, August 1968.

[126] C. S. Wallace and P. R. Freeman. Estimation and inference by compact coding. *Journal of the Royal Statistical Society. Series B*, 49(3):240–265, 1987.

[127] J. Weston, B. Schölkopf, E. Eskin, C. Leslie, and W. Noble. Dealing with large diagonals in kernel matrices. *Annals of the Institute of Statistical Mathematics*, 55 (2):391–408, 2003.

[128] E. A. Yildirim. Two algorithms for the Minimum Enclosing Ball problem. *SIAM Journal on Optimization*, 19(3):1368–1391, 2008.

[129] B. Zadrozny, J. Langford, and N. Abe. Cost-sensitive learning by cost-proportionate example weigting. In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, pages 435–442, 2003.

Index

AdaTron, 32

Beneath-Beyond algorithm, 44
    for Reduced Convex Hulls, 51

centroid polytope, 54
clamp function, 97
complementary slackness, 11
convex hull, 40
    algorithms, 41
cross-validation, 146

dual, 11

error estimate, 146
extreme point, *see* vertex

Generalized Approximate Cross-Validation, 148
Gilbert's algorithm, 100

hold-out set, 146

k-means classifier, 70
k-nearest neighbor classifier, 70
k-set, 53
k-set polytope, 54
kernel, 14
Kronecker Delta, 20

Lagrange multiplier, 11
leave-one-out error, 147

maximum margin, 10
Minimal Enclosing Ball, 29
Minkowski set difference, 102

normal (vector), 41

offset, 12

parameter selection, 145
    for μ-SVMs, 166, 168
    for C-SVMs, 167
Perceptron, 31
    bias term, 34
    distinguishing from an SVM, 36
    general form, 35
    geometric interpretation, 88
    optimal stability, 33
    Rosenblatt's perceptron, 32
    soft margin, 34
    training, 134
    weighted, 90
primal, 11

Quickhull algorithm, 42, 51
    for Reduced Convex Hulls, 50

radius-margin bound, 150
    accelerated, 152
Reduced Convex Hull, 45
    algorithms, 48
    weighted, 60
ridge, *see* subfacet

Schlesinger-Kozinec algorithm, 95
    efficient implementation, 116
    weighted, 109
Sequential Minimal Optimization, 106
    heuristic, 107
    update step, 106
stopping condition, 105, 128
subfacet, 40
support point, 47
support vector, 12, 18
    bounded, 19
support vector error bound, 147

Support Vector Machine
  C-SVM, 17
  L_1-loss, 17
  L_2-loss, 19
  μ-SVM, 21
  general form, 36
  hard margin, 10
  soft margin, 16
  weighted, 27, 28

threshold, 12, 72

vertex, 40, 47

Xi-Alpha bound, 149