Fall 2018

Multiclass Classification Using Support Vector Machines

Duleep Prasanna W. Rathgamage Don

MULTICLASS CLASSIFICATION USING SUPPORT VECTOR MACHINES

by

DULEEP RATHGAMAGE DON

(Under the Direction of Ionut Iacob)

ABSTRACT

In this thesis we discuss different SVM methods for multiclass classification and introduce the Divide and Conquer Support Vector Machine (DCSVM) algorithm which relies on data sparsity in high dimensional space and performs a smart partitioning of the whole training data set into disjoint subsets that are easily separable. A single prediction performed between two partitions eliminates one or more classes in a single partition, leaving only a reduced number of candidate classes for subsequent steps. The algorithm continues recursively, reducing the number of classes at each step, until a final binary decision is made between the last two classes left in the process. In the best case scenario, our algorithm makes a final decision between k classes in $O(\log_2 k)$ decision steps and in the worst case scenario DCSVM makes a final decision in $k-1$ steps.

INDEX WORDS: Statistical learning, Classification, Curse of dimensionality, Support Vector Machine, Kernel trick

2009 Mathematics Subject Classification: 62H30, 68T10

MULTICLASS CLASSIFICATION USING SUPPORT VECTOR MACHINES

by

DULEEP RATHGAMAGE DON

B.S., The Open University of Sri Lanka, Sri Lanka, 2008

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial

Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

MULTICLASS CLASSIFICATION USING SUPPORT VECTOR MACHINES

by

DULEEP RATHGAMAGE DON

Major Professor:	Ionut Iacob
Committee:	Mehdi Allahyari
	Stephen Carden
	Goran Lesaja

Electronic Version Approved:
December 2018

DEDICATION

I dedicate this thesis to my beloved wife Kasuni Nanayakkara, Ph.D. for nursing me with affections and guiding me to the success in my life.

ACKNOWLEDGMENTS

I am grateful to my graduate adviser Dr. Huwa Wang and my thesis adviser Dr. Ionut Iacob for their valuable academic and research guidance and placing their trust in my abilities for all my achievements at Georgia Southern University. I am also grateful to all my professors at Georgia Southern University with special thanks to Dr. Saeed Nasseh, Dr. Stephen Carden and Dr. Goran Lesaja who made an enthusiasm in my graduate studies. My forever interested, Sri Lankan professors V. P. S. Perera, W. Ramasinghe, and W. B. Daundasekara: they were always a part of my higher education and professional development. I appreciate your support in my past endeavors. A special gratitude goes to Ms. Kathleen Hutcheson: as my boss she made my 7^{th} teaching job very pleasant. At last but by no means least, I would like to be thankful to all the graduate students and staff at the Department of Mathematical Sciences. Thanks for all your encouragement!

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

LIST OF SYMBOLS

$\mathbb{R}$	Real numbers
$\mathbb{R}^n$	n-dimensional Euclidean space
$\mathcal{H}$	Hilbert Space
sign(x)	Sign function of x
dim(X)	Dimension of vector space X
SV	Set of support vectors
$\mathbf{x}$	Vector x
$\mathbf{x}^T$	Transpose of vector x
$\binom{n}{r}$	Set of all r-combinations of n elements
$\|\cdot\|$	Vector norm
$\sum_{i=1}^{n}$	Summation operator over i from 1 to n
$\bigtriangledown$	Vector differential operator
$\frac{\partial}{\partial x}$	Differential operator with respect to x
$\int_{x,x}$	Double integral operator with respect to x
$< \cdot, \cdot >$	Inner product
$svm_{i,j}$	SVM binary classifier
$C_{i,j}(l, i)$	Class predictions likelihood of SVM for label l
$\mathcal{P}_{i,j}(\theta)$	Purity index of SVM for accuracy threshold θ
$\mathcal{B}_{i,j}(\theta)$	Balance index of SVM for accuracy threshold θ

CHAPTER 1

INTRODUCTION

Support Vector Machines (SVMs) belong to the class of supervised learning methods in machine learning and artificial intelligence. They learn from data and recognize patterns which are useful for classification and regression. The SVMs are widely used as a reliable tool for data mining, pattern recognition, and predictive analytics in medicine, engineering, and business.

In the 1960's the statistical learning theory was mostly developed for learning algorithms of nonlinear functions through the seminal work by Vapnik and Chervonenkis [12]. The statistical learning theory deals with the following fundamental problems about learning from data: what conditions help a model learn from examples? and how does the performance measured on selected of examples lead to the bounds on generalization in (2.3)? There is still ongoing research in this area and the conditions for theories to be valid are almost impossible to check for most practical problems [24]. The same researchers proposed the restoration of linear separability methods, with additional ingredients intended to improve generalization, with the name of Support Vectors Machines (SVM).

In 1992, Bernhard Boser, Isabelle Guyon, and Vladimir Vapnik developed a method for nonlinear classification by introducing the kernel trick to maximum-margin SVM [9]. A later extension of SVM was found with soft margin classification contributing to document classification where preference solution was to better separate the bulk of the data while ignoring a few outliers. Further, multiclass SVMs were developed to handle a multiclass classification problem by decomposing the original problem into a series of binary problems so that the standard SVM could be directly applied to each problem. The multiclass classification with SVM is still an ongoing research problem (see, for example, [13, 18, 19, 21] for some recent work). In this study, we especially focus on a novel SVM designed for the classification of multiclass and high dimensional datasets [24].

Due to recent technological advances, a large amount of high-throughput data can be collected simultaneously and at relatively low-cost [25]. Classification of such data raises new challenges, for instance, deterioration in the accuracy of learning algorithms with the dimension of data. The reason is that the data becomes sparse at high dimensions making training of the learning algorithms problematic due to large regularization error or computational complexity. These phenomena referred to as the curse of dimensionality set the motivation for our research [27].

We observed that a binary SVM decision boundary might separate more than two classes of data in multiclass settings (Fig 5.2). The observation can be repeated until all the classes are eliminated. This concept is even more persistent when the data becomes sparse as their dimension increases. Our method utilizes a decision tree approach to investigate this property for a given dataset. The first step is similar to One-Vs-One method where a pool of SVM classifiers are built considering any pair of classes. We grade them according to their impurity, balance, and score (see the definitions 5.3 and 5.4). The best classifier is selected to be the top node of the decision tree (Fig 5.4). Then the selection procedure is repeated on a reduced pool of SVM to figure out the other nodes of the decision tree until all the classes are separated [24].

The performance of DCSVM is tested against classifying some real datasets found in the UCI machine learning repository. We implemented tenfold cross-validation and averaged the results to compare the DCSVM with the classic One-Vs-One method in terms of the prediction accuracy, number of support vectors, number of binary decisions, and prediction time. The DCSVM has several advantages over the One-Vs-One method including but not limited to less predicting time, and less computational complexity while maintaining the same prediction accuracy.

CHAPTER 2

STATISTICAL LEARNING THEORY

Let us consider the basic model of a learning machine.

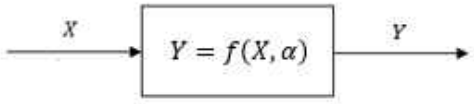


Figure 2.1: Block diagram of the basic model of a learning machine

Suppose we have a learning machine whose task is to learn the mapping $\mathbf{x}_i \mapsto y_i$ for $i = 1, \ldots, n$ where $\mathbf{x}_i \in X$, a m dimensional vector and $y_i \in Y = \{-1, 1\}$, a set of labels. The machine is actually defined by a set of possible mappings $X \mapsto f(X, \alpha)$, where the function is governed by an adjustable parameter α referred to its representational power which is a measure of the complexity of the learning machine. There is an important set of bounds controlling the relationship between the performance of a learning machine and it's representational power [8]. The usual trade-off in this case is that more power results in over-fitting while the less power might end up in under-fitting.

2.1 VAPNIK CHERNONKIS (VC) DIMENSION

In order to characterize the representational power of a learning task assigned to a machine or its hypothesis class, we consider the VC dimension.

Definition 2.1. Given that A is a set and C is a collection of sets. C is said to shatter A if for each a $\subset$ A, there exists c $\in$ C such that,

$$a = c \cap A$$

Definition 2.2. VC dimension of a hypothesis class is the maximum number of data points that can be arranged so that the learning machine can shatter them.

Consider the following hypothesis class. i.e., linear classifier in 2-dimensional space.

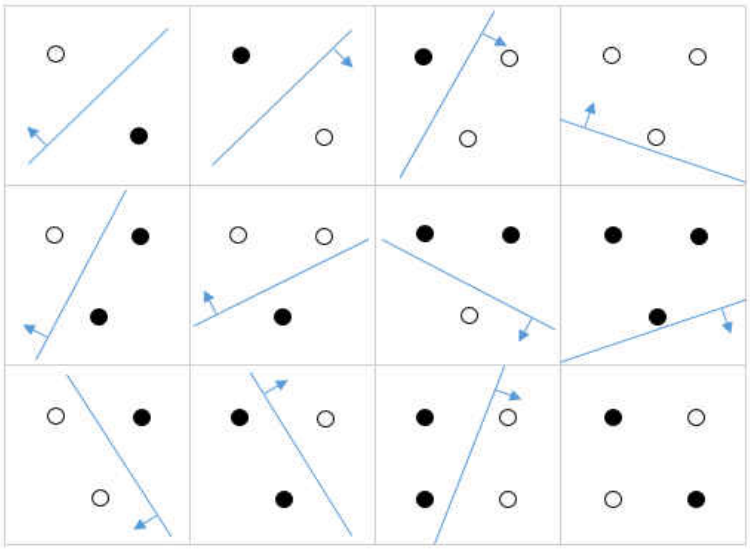


Figure 2.2: Shattering of linear classifier in 2-dimensional space

It is clear that the VC dimension of this particular hypothesis class is three as the given configuration of four data points in the last block of Figure 2.2 cannot be shattered.

Theorem 2.1. *(proof is omitted) Consider some set of m points in* $\mathbb{R}^n$*. Choose any one of the points as origin. Then the m points can be shattered by oriented hyperplanes if and only if the position vectors of the remaining points are linearly independent.*

Corollary 2.2. *The VC dimension of the set of oriented hyperplanes in* $\mathbb{R}^n$ *is* $n + 1$.

Proof. We can always choose $n+1$ points, and then choose one of the points as origin, such that the position vectors of the remaining n points are linearly independent, but can never choose $n + 2$ such points (since no $n + 1$ vectors in $\mathbb{R}^n$ can be linearly independent). □

2.2 STRUCTURAL RISK MINIMIZATION

It is assumed that there exists some unknown cumulative probability distribution $P(X, Y)$ from which the data are drawn i.e., the data assumed to be independently drawn and identically distributed (iid). The expectation of the test error for a trained machine is therefore (using Lebesgue integration):

$$R(f) = \frac{1}{2} \int |Y - f(X, \theta)| \, dP(X, Y) \tag{2.1}$$

Where, $R(f)$ is called the expected risk or true risk. Given that the number of training instances is n, the empirical risk $R_{emp}(f)$ is defined to be the measured mean error on the training set.

$$R_{emp}(f) = \frac{1}{2n} \sum_{i=1}^{n} |Y - f(X, \theta)| \tag{2.2}$$

The quantity $\frac{1}{2}|Y - f(X, \theta)|$ is called the loss. For this case, it can only take the values 0 and 1. Now choose some η such that $0 \leq \eta \leq 1$. Then for losses taking these values, with the probability $1 - \eta$ the following bounds hold [8].

$$R(f) = R_{emp}(f) + \sqrt{\frac{h(\log(2n/h) + 1) - \log(\eta/4)}{n}} \tag{2.3}$$

Where h is VC dimension. The second term on the right hand side is called VC confidence. If we know h, we can easily compute the right hand side of (2.3). Thus we consider several different learning machines, for sufficiently small η. Then choosing the machine that minimizes the right hand side of (2.3), we find the machine which gives the lowest upper bound on the true risk as shown in Figure 2.3. This gives a principle method for choosing a learning machine for a given task, and it is the essential idea of Structural Risk Minimization (SRM) [8].

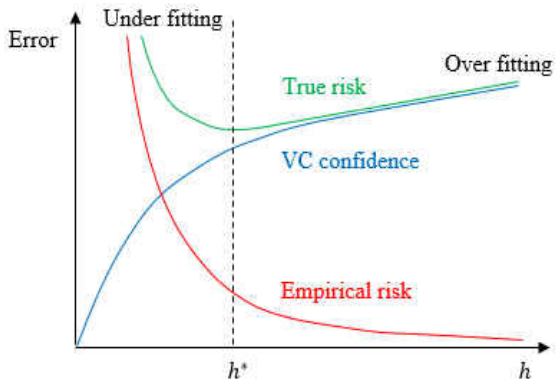


Figure 2.3: Structural risk minimization of a learning machine

Also notice that the VC confidence of (2.3) may be ignored for very large n. In practice, we will not have infinitely many instances in the training dataset. If h is too large with respect to n, then there is a risk of over-fitting the training data. Hence, the SRM plays an important role in implementing supervised learning machines. The SRM is an inductive principle for model selection from finite training datasets. The goal of this principle is to govern the generalization error of the learning machine by limiting the flexibility of the set of decision functions, measured by the VC dimension. Instead of utilizing a decision function f by minimizing the training error on a selected data set, f is selected to minimize an upper bound on the test error given by (2.3) as outlined below.

1. Using a prior knowledge of the domain, choose a class of functions and divide it into a hierarchy of nested subsets in order of increasing complexity.

2. Perform empirical risk minimization on each subset (this is essentially parameter selection).

3. Select the model (h^*) in the series whose sum of empirical risk and VC confidence is minimal.

CHAPTER 3

INTRODUCTION TO SUPPORT VECTOR MACHINE

3.1 HARD MARGIN SUPPORT VECTOR MACHINE

The simplest version of SVM is the learning machine that classify m-dimensional linearly separable data into two classes C_1 and C_2 depending on the labels y_i.

Definition 3.1. A set of m-dimensional binary class data is linearly separable if there exists at least one $m-1$ dimensional vector space called hyperplane with all the data points of C_1 on one side of the hyperplane and all the data points of C_2 on the other side.

The gap between the linearly separable data is known as margin. The hyperplane is made in such a way that the margin is maximized. Suppose that we randomly select some vectors $\mathbf{x}_i$ for the training set. Our goal is to find the optimal hyperplane that classifies all the training data into two classes. Figure 3.1 shows a 2-dimensional feature space, however the following method can be applied to the general case.

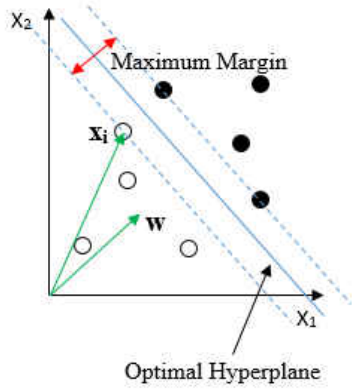


Figure 3.1: Binary classification problem: building up support vector machine

Suppose a given $\mathbf{x}_i$ is on the hyperplane. $\mathbf{w}$ is a normal vector called weight which controls the direction of the hyperplane and b is a scalar called bias that controls the position

of the hyperplane [10]. Then the equation of the hyperplane is $\mathbf{w}^T\mathbf{x}_i + b = 0$ as it linear.

Then the SVM is a learning machine with a decision rule of the form:

$$f(\mathbf{x}_i, \alpha) = \text{sign}(\mathbf{w}^T\mathbf{x}_i + b) \tag{3.1}$$

Suppose for an arbitrary $\mathbf{x}_i$ we have

$$\begin{aligned} f(\mathbf{x}_i, \alpha) &= \mathbf{w}^T\mathbf{x}_i + b > 1, \ if \ \mathbf{x}_i \in C_1 \\ &= \mathbf{w}^T\mathbf{x}_i + b < -1, \ if \ \mathbf{x}_i \in C_2; \end{aligned} \tag{3.2}$$

Now for each $\mathbf{x}_i$ and $\mathbf{w}$ we choose $y_i \in$ Y, set of class labels such that

$$y_i = \begin{cases} 1, \ if \ \mathbf{x}_i \in C_1 \\ -1, \ if \ \mathbf{x}_i \in C_2 \end{cases}$$

Both (3.2) relationships imply that

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 \geq 0; \ \mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R} \tag{3.3}$$

Definition 3.2. Support vectors are the data points of the classes C_1 and C_2 that are at the boundary of the maximum margin as shown in Figure 3.2.

To find the width of the margin, consider the $\mathbf{x}_i$ and $\mathbf{x}_j$ shown in Figure 3.2

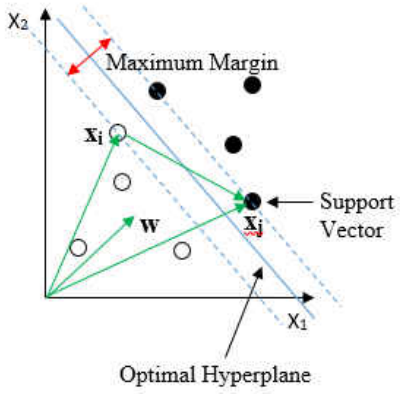


Figure 3.2: Hard margin support vector machine

The width of the margin is $(\mathbf{x}_j - \mathbf{x}_i)\mathbf{w}/\left\|\mathbf{w}\right\|$

By (3.2),

$$\text{The width of the margin} = \frac{2}{\|\mathbf{w}\|} \tag{3.4}$$

Our goal is to maximize the margin, i.e., to minimize $\|\mathbf{w}\|$.

Now the quadratic programing problem is

$$\min_{w \in R^m} \frac{1}{2} \|\mathbf{w}\|^2$$

subject to (3.3)

The solution to the above problem is a global minimum satisfying the Karush−Kuhn−Tucker (KKT) conditions.

Definition 3.3. KKT conditions for the optimization problem with the objective function $f : \mathbb{R}^n \mapsto \mathbb{R}$ and constraint functions $g_j : \mathbb{R}^n \mapsto \mathbb{R}$ and $h_k : \mathbb{R}^n \mapsto \mathbb{R}$ such that *f*, *g*, and *h* are continuously differentiable at a local optimum x^* for constants α_j (j = 1,...,m) and β_k (k = 1,...,l) are given below.

Condition 1 (Stationarity)

For maximizing f(x): $\bigtriangledown f\left(x^*\right) = \sum_{j=1}^{m} \alpha_j \bigtriangledown g_j\left(x^*\right) + \sum_{k=1}^{l} \beta_k \bigtriangledown h_k\left(x^*\right)$

For minimizing f(x): $-\bigtriangledown f\left(x^*\right) = \sum_{j=1}^{m} \alpha_j \bigtriangledown g_j\left(x^*\right) + \sum_{k=1}^{l} \beta_k \bigtriangledown h_k\left(x^*\right)$

Condition 2 (Primal Feasibility)

$g_j\left(x^*\right) \leq 0$ for j = 1,...,m and $h_k\left(x^*\right) = 0$ for k = 1,...,l

Condition 3 (Dual Feasibility)

$\alpha_j \geq 0$ for j = 1,...,m

Condition 4 (Complementary Slackness)

$\alpha_j g_j\left(x^*\right) = 0$ for j = 1,...,m

To apply the KKT conditions, it is customary to write our optimization problem in the form of following auxiliary function (similar to Lagrangian) using the primal feasibility.

$$L_p(\mathbf{w}, b, \alpha_j) = \frac{1}{2} \parallel \mathbf{w} \parallel^2 - \sum_{j=1}^{n} \alpha_j [y_j(w^T \mathbf{x}_j + b) - 1] \tag{3.5}$$

To find the solution, first we minimize (3.5) with respect to $\mathbf{w}$ and b and subject to (3.3). Hence this optimization is known as the primal problem.

Let us write (3.5) in the following way.

$$L_p(\mathbf{w}, b, \alpha_j) = \frac{1}{2} \parallel \mathbf{w} \parallel^2 - \sum_{j=1}^{n} \alpha_j y_j \mathbf{w}^T \mathbf{x}_j - \sum_{j=1}^{n} \alpha_j y_j b + \sum_{j=1}^{n} \alpha_j$$

Considering the stationarity; $\partial L_p / \partial b = 0$ and $\partial L_p / \partial \mathbf{w} = 0$, it follows that:

$$\sum_{j=1}^{n} \alpha_j y_j = 0 \tag{3.6}$$

$$\mathbf{w} = \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j \tag{3.7}$$

To obtain the minimum solution of the primal problem, we substitute (3.6) and (3.7) in (3.5) with new indexing to obtain the dual problem (3.8).

$$L_d(\alpha_i) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \mathbf{x}_j) \tag{3.8}$$

Considering dual feasibility, we can maximize (3.8) with respect to α_i. and subject to (3.6) Assume the solution of the dual problem is found by using Sequential Minimal Optimization [28] which is commonly used in practice. An appropriate quadratic programming package can also be used in this case if the dual problem is set for a minimization. The corresponding maximizers are considered to be the non-zero Lagrange multipliers α^* associated with the support vectors.

If $\mathbf{x}_j$ is a support vector, then from (3.7)

$$\mathbf{w} = \sum_{j \in SV} \alpha_j^* y_j \mathbf{x}_j \tag{3.9}$$

After the dual problem is solved, the bias is calculated by using (3.10) [22]

$$b = \frac{1}{\#SV} \left(\sum_{j \in SV} \left(y_j - \mathbf{w}^T \mathbf{x}_j \right) \right) \tag{3.10}$$

By substituting (3.9) and calculated (3.10) in (3.1), we modify our decision rule on testing data $\mathbf{x}_i$ as in (3.11). This is the essential procedure of training the hard margin SVM. $\mathbf{w}$ and α are called SVM classification parameters. Note that the hard margin SVM is suitable for linearly separable data [10].

$$f(\mathbf{x}_i, \alpha) = \text{sign}\left(\sum_{j \in SV} \alpha_j^* y_j (\mathbf{x}_j \mathbf{x}_i) + b\right) \tag{3.11}$$

3.2 Soft Margin Support Vector Machine

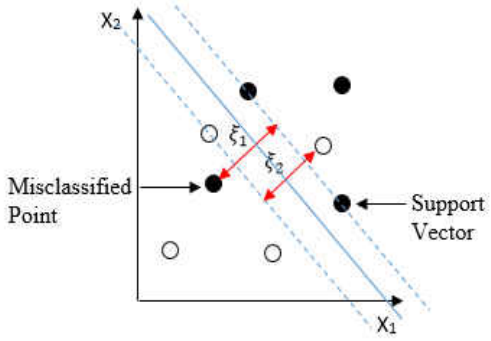


Figure 3.3: Soft margin support vector machine

If the data is nearly linearly separable (e.g., due to noise), the condition for the optimal hyper-plane can be relaxed by including an extra term: slack variable ξ_i as shown in Figure 3.3. This idea will result in an efficient classifier know as soft margin SVM.

From (3.3)

$$y_j(\mathbf{w}^T \mathbf{x}_j + b) \geq 1 - \xi_j; \text{ for } j = 1, 2, \ldots\ldots, n \text{ and } \xi_j \geq 0 \tag{3.12}$$

Then the quadratic programming problem becomes;

$$min_{w \in R^m} \frac{1}{2} \parallel \mathbf{w} \parallel^2 + C \sum_{j=1}^{n} \xi_j \tag{3.13}$$

subject to (3.12)

Here, C is a regularization parameter called the cost of the SVM that controls the trade-off between maximizing the margin and minimizing the training error by imposing a penalty on the objective function for misclassification. Small C tends to emphasize the margin while ignoring the outliers in the training data as Figure 3.4 (A), while large C may tend to over fit the training data as Figure 3.4 (B).

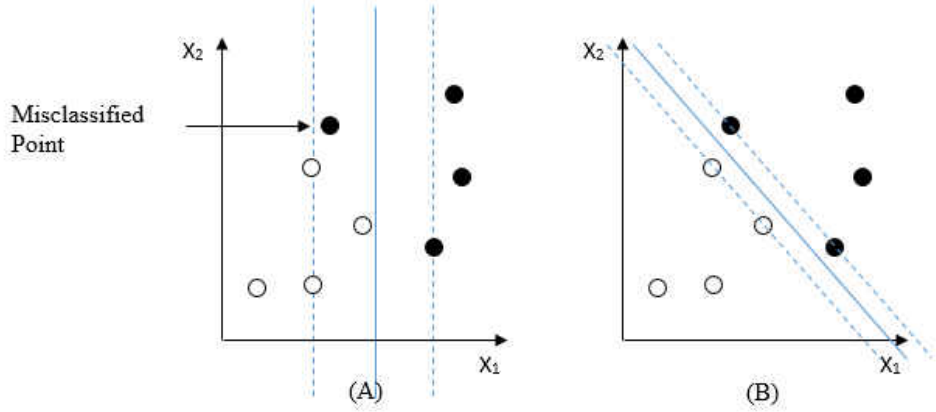


Figure 3.4: Trade-off between maximum margin and minimum training error

As in the hard margin case, using KKT conditions, the primal form of (3.13) can be expressed as,

$$L_p(\mathbf{w}, b, \xi_j, \alpha_j, \lambda_j) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{j=1}^{n} \xi_j - \sum_{j=1}^{n} \alpha_j [y_i(\mathbf{w}^T \mathbf{x}_j + b) - 1 + \xi_j] - \sum_{j=1}^{n} \lambda_j \xi_j \tag{3.14}$$

subject to (3.12)

We write (3.14) in the following way

$$L_p \equiv \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{j=1}^{n} \xi_j - \mathbf{w}^T \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j - b \sum_{j=1}^{n} \alpha_j y_j + \sum_{j=1}^{n} \alpha_j - \sum_{j=1}^{n} \alpha_j \xi_j - \sum_{j=1}^{n} \lambda_j \xi_j$$

considering the stationarity; $\partial L_p/\partial \mathbf{w} = 0$, $\partial L_p/\partial b = 0$, and $\partial L_p/\partial xi_j = 0$, it follows that:

$$\mathbf{w} = \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j \tag{3.15}$$

$$\sum_{j=1}^{n} \alpha_j y_j = 0 \tag{3.16}$$

$$C - \alpha_j - \lambda_j = 0 \tag{3.17}$$

respectively. Substitute (3.15), (3.16), and (3.17) in (3.14), to obtain the dual problem which is similar to (3.8) and subject to (3.16) and (3.17). The only significant difference between the two dual problems is the soft margin dual problem has a Lagrange multiplier bounded above.

It is clear that λ_j is a Lagrange multiplier and hence nonnegative. If $\lambda_j = 0$, then from (3.17) it follows that $\alpha_j = C$. If $\lambda_j > 0$, then from (3.17) it follows that $\alpha_j < C$. Hence the dual problem for (3.14) that needs to be maximized with respect to α_j and subject to $0 \leq \alpha_j \leq C$ is given below.

$$L_d(\alpha_j) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \mathbf{x}_j) \tag{3.18}$$

The solution to (3.18) can be obtained by using the same techniques used to solve (3.8). Then the decision rule of the soft margin SVM is similar to (3.11).

To find the type of support vectors of the soft margin SVM, we consider the complementary slackness for the optimization problem as follows.

$$\alpha_j [y_j (\mathbf{w}^T \mathbf{x}_j + b) - 1 + \xi_j] = 0 \tag{3.19}$$

$$\lambda_j \xi_j = 0 \tag{3.20}$$

It follows that for a support vector $\alpha_j \neq 0$. Then from (3.19) we observe the following.

$$y_j (\mathbf{w}^T \mathbf{x}_j + b) = 1 - \xi_j \tag{3.21}$$

There are two types of support vectors associated with the training of soft margin SVM. If $\lambda_j > 0$, then from (3.20) $\xi_j = 0$ and from (3.21) it follows that this data point is at the boundary of the margin. From (3.19) we have $\alpha_j < C$. If $\lambda_j = 0$, then from (3.20) $\xi_j > 0$ and from (3.21) it follows that this data point is over the margin. From (3.17) we have $\alpha_j = C$. The choice of an appropriate value for C is an essential part of tuning an SVM and is discussed in chapter 3.4.

3.3 KERNEL TRICK

Let us consider the problem with hard margin. If the data is not linearly separable, the optimization of (3.8) can be performed on a higher dimensional vector space. In this case, the data is mapped from input-space to feature-space by a mapping Φ and the optimal separating hyperplane is found. Then the separation is mapped back down to input-space, where it results in a non-linear decision boundary as shown in Figure 3.5.

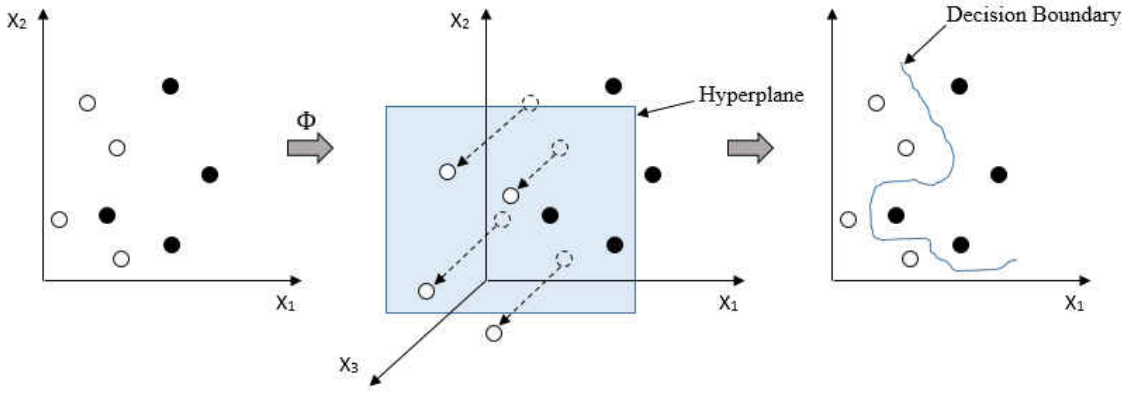


Figure 3.5: Nonlinear classification

Let U and V be vector spaces such that $\dim(V) > \dim(U)$ and Φ be a function given by Φ: $U \to V$. For $\mathbf{x}_i$, $\mathbf{x}_j \in U$ (input space), from (3.8) the corresponding optimization in

V (feature space) is,

$$max \ \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i) \Phi(\mathbf{x}_j) \tag{3.22}$$

with respect to α_i and subject to (3.6). Calculation of $\Phi(\mathbf{x}_i)\Phi(\mathbf{x}_j)$ is very expensive in both computation and memory consumption as the dimension of feature space could be very large or even infinity. Kernel trick allows us to perform the above task without actually transforming the training data into the high dimensional feature space.

Example 3.1. *Let* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *be two dimensional vectors such that* $\boldsymbol{x} \equiv (x_1, x_2)$ *and* $\boldsymbol{y} \equiv (y_1, y_2)$ *and* Φ *be a transformation such that* $\Phi : \mathbb{R}^2 \mapsto \mathbb{R}^3$ *given by* $\Phi(s,t) = (s^2, t^2, \sqrt{2}st)$; $s, t \in \mathbb{R}$. *We can show that the dot product of* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *can be obtained using a certain function* k *know as the kernel function.*

$$\begin{aligned} < \Phi(\boldsymbol{x}), \Phi(\boldsymbol{y}) > &= (x_1^2, x_2^2, \sqrt{2}x_1x_2)(y_1^2, y_2^2, \sqrt{2}y_1y_2)^T \\ &= ((x_1, x_2)(y_1, y_2)^T)^2 \\ &=< \boldsymbol{x}, \boldsymbol{y} >^2 \\ &= k(\boldsymbol{x}, \boldsymbol{y}) \end{aligned}$$

Note that not all kernel functions can perform a sort of calculation in the example above. A kernel function that legitimately behaves as in the example above is called a Mercer kernel k introduced in Mercer's theorem.

Theorem 3.2. *Mercer's Theorem (Proof is omitted) Let X* $\in \mathbb{R}^n$. *For any symmetric continuous function k: X*$\times$*X* $\mapsto \mathbb{R}$ *such that k* $\in$ *L*$_2$*(X) (square integrable in X*$\times$*X) satisfying,*

$$\int_{X,X} f(x') \, k(x, x) \, f(x) \, dx dx' \geq 0 \tag{3.23}$$

for all f $\in$ *L*$_2$*(X)*

There exist functions Φ_i*: X* $\mapsto \mathcal{H}$ *and constants* $\lambda_i \geq 0$ *such that*

$$k(x, x') = \sum_{i=1}^{\infty} \lambda_i \Phi_i(x) \Phi_i(x') \tag{3.24}$$

for all x, x'$\in$ *X*

Definition 3.4. The kernel trick is the relationship in (3.25) where k is a Mercer kernel. It can be used to avoid the mapping needed for linear learning algorithms to learn a non-linear function or decision boundary by implicitly calculating the inner products between the feature vectors.

$$k(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i).\Phi(\mathbf{x}_j) \tag{3.25}$$

It is clear that (3.22) will result in the following decision rule.

$$f(\mathbf{x}_i, \alpha) = \text{sign}\left(\sum_{j \in sv} \alpha_j^* y_j \Phi(\mathbf{x}_j)\Phi(\mathbf{x}_i) + b\right) \tag{3.26}$$

To apply the kernel trick in the nonlinear decision rule, we substitute (3.25) in (3.26)

$$f(\mathbf{x}_i, \alpha) = \text{sign}\left(\sum_{j \in sv} \alpha_j^* y_j \, k(\mathbf{x}_i, \mathbf{x}_j) + b\right) \tag{3.27}$$

There are several different kernel functions used in SVM. Table 3.1 shows some common kernels. Radial Basis Function (RBF) kernel is the most popular kernel function for SVM. Not all RBF kernels are Mercer kernels. The Gaussian kernel is an ideal example for both RBF and Mercer kernels. In Gaussian RBF kernels, $\gamma = \frac{1}{2\sigma}$ where σ is the determinant of the covariance matrix which needs to be tuned properly. If overestimated (σ is very small), the exponential will behave almost linearly and the higher-dimensional projection will start to lose its non-linear power. If underestimated (σ is very large), the function will lack regularization and the decision boundary will be highly sensitive to noise in training data [7].

Table 3.1: Some common kernels

Kernel Type	**Kernel Function and Dimension**	**Remarks**
Linear	$k(x_i, x_j) = x_i^T x_j + c$ For dim(x) = n, dim(k) = n	The simplest kernel function. A SVM built with the linear kernel is generally equivalent to its non-kernel counterpart. c is an arbitrary constant.
Polynomial	$k(x_i, x_j) = (\alpha x_i^T x_j + c)^d$ For dim(x) = n, dim(k) = $\binom{n+d}{d}$	A non-stationary kernel function. It is suitable for problems in which the training is normalized [7]. α is the slope. d is the degree of the polynomial and c is an arbitrary constant.
RBF	$k(x_i, x_j) = exp(-\gamma \left\|x_i - x_j\right\|^2)$ For dim(x) = n, dim(k) = ∞	A versatile and efficient kernel function. It can be computationally expensive for a high dimensional input space. γ is an adjustable parameter.

3.4 IMPLEMENTATION

In this project we focus on implementations of SVM in classification through selection of parameters, training, and measure of performance. The R programing language is extensively used in our experiments which primarily included the package e1071. The other R

packages used are presented in the following subsections.

By construction, the SVM classifier works only on dichotomous data. A practical SVM combines the kernel trick with the soft-margin classifier to enhance its generalization ability making the SVM one of the most sophisticated supervised learning algorithms. The SVM is accurate and robust in nonlinear classification and the emphasis can be extended to multiclass classification in Chapters 4 and 5. The generalization ability of these multiclass SVMs is fully dependent on that of their binary SVM classifiers and determined by selection of correct parameters.

Example 3.3. *The following synthetic dichotomous dataset generated by a bivariate normal distribution ($x \sim N(5, 1.5)$, $y \sim N(4, 1)$, and $\rho = 0.07$) with 100 instances. The dataset is called synthetic1 and used to demonstrate the implementation of SVM in nonlinear classification.*

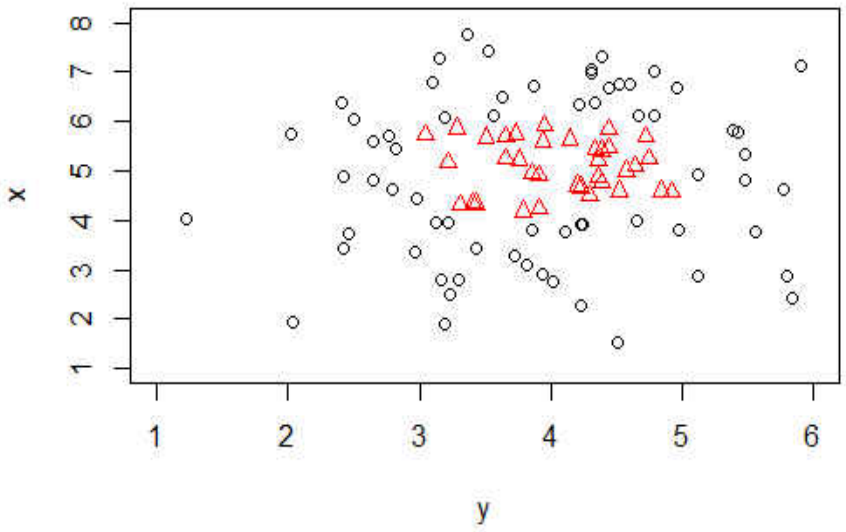


Figure 3.6: Scatter plot of synthetic1

We randomly divide the dataset into two parts: training and testing. 70% of the total

instances are used for training purpose. Recall (Section 2.2) that there is no upper bound for this value as more training instances yield better generalization. Real datasets often require normalization and feature reduction before undergoing classification. Since the SVM classifier is sensitive to distances in Hilbert space, this step is very important when dealing with attributes of different units and scales.

Normalization scales all numeric attributes in the range [0, 1] using the formula given below.

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.28}$$

For the purpose of feature reduction or selecting a subset of relevant features in model building, Principal Component Analysis (PCA) has become a widely used technique. The PCA is a statistical procedure in which an orthogonal transformation is used to convert a set of observations of potentially correlated variables into a set of values of linearly uncorrelated variables called principal components. Depending on the amount of variability captured by each component, the final model ends up with a fewer number of features.

3.4.1 Selection of Parameters

The selection of an appropriate kernel is regarded as a rule in classification experiments. We chose efficient and versatile RBF kernel to construct required SVM classifiers to standardize our experiments. In parameter selection, appropriate values for regularization parameters C and γ of the SVM classifier is found. In general, determination of these parameters are based on so called grid search. The range of value of parameters required for the search algorithm is arbitrary. In our example, the parameter values for C are 1, 10, 100, 1000, 10000, and 100000. The parameter values for γ are 0.00001, 0.0001, 0.001, 0.01, 0.1, and 1. The following output file corresponds to the grid search of the regularization parameters in classifying synthetic1.

```
Parameter tuning of svm:

- sampling method: 10-fold cross validation

- best parameters:
 gamma  cost
  0.01 1e+05

- best performance: 0.01428571

- Detailed performance results:
   gamma  cost      error dispersion
1  1e-05 1e+00 0.28571429 0.21295885
2  1e-04 1e+00 0.28571429 0.21295885
3  1e-03 1e+00 0.28571429 0.21295885
4  1e-02 1e+00 0.28571429 0.21295885
5  1e-01 1e+00 0.28571429 0.21295885
6  1e+00 1e+00 0.04285714 0.06900656
7  1e-05 1e+01 0.28571429 0.21295885
8  1e-04 1e+01 0.28571429 0.21295885
9  1e-03 1e+01 0.28571429 0.21295885
10 1e-02 1e+01 0.28571429 0.21295885
.................................
.................................
34 1e-02 1e+05 0.01428571 0.04517540
35 1e-01 1e+05 0.02857143 0.09035079
36 1e+00 1e+05 0.10000000 0.11761037
c@FancyVerbLinee.10000000 0.11761037
```

The R package $e1071$ has a built-in function for the grid search of SVM learning parameters that minimizes the training error to figure out the optimal values of the parameters. In this

case, the minimum training error is known as the best performance. Fig 3.9 illustrates the performance of the grid search in which the darker regions imply better accuracy.

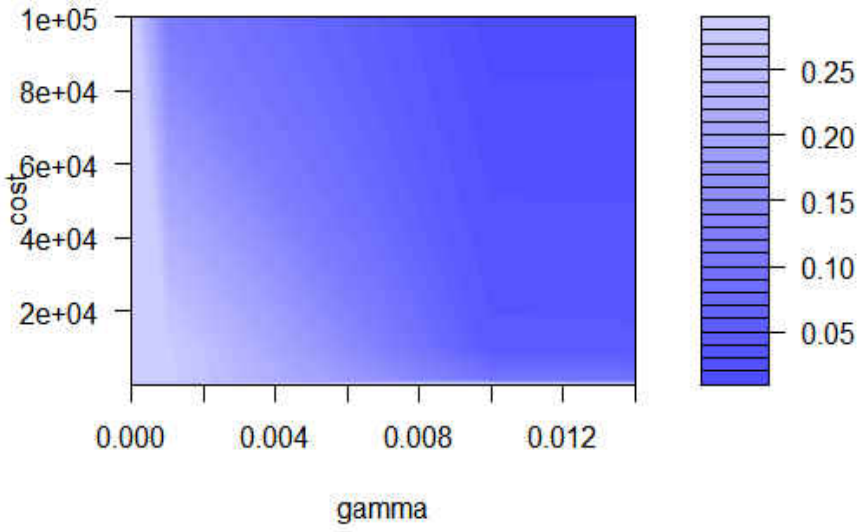


Figure 3.7: Performance of SVM - grid search of learning parameters

3.4.2 TRAINING

The optimal learning parameters directly go into the training model of the SVM classifier. In general, the training procedure of the SVM requires cross validation using training data to determine the bias and support vectors. In our example, 10 fold cross validation is used. In nonlinear classification, training occurs in the high dimensional feature space and the resulting classification is mapped to the original lower dimensional space as in Fig 3.10 where the support vectors (cross sign) and nonlinear decision boundary for the whole dataset can be identified.

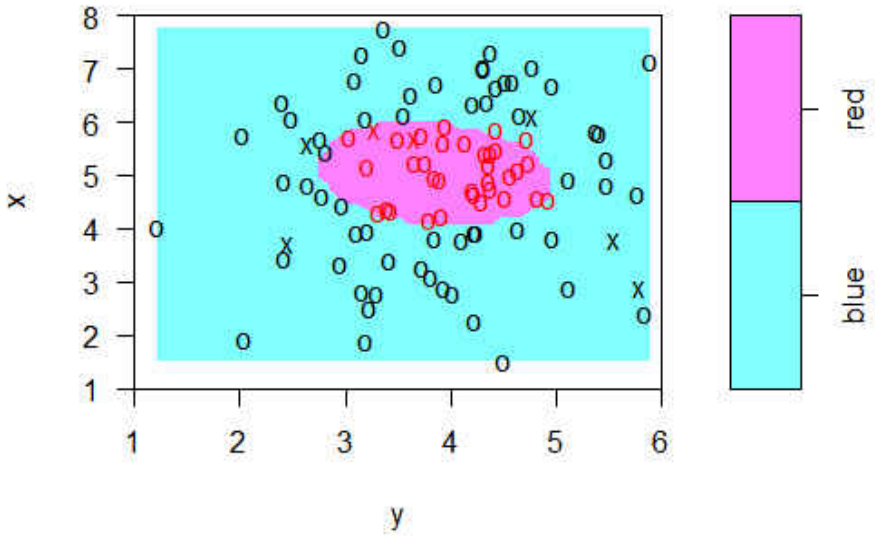


Figure 3.8: SVM classification plot of synthetic1

3.4.3 Evaluation of Performance

Testing data is used to conclude the performance of the SVM and there are several metrics such as accuracy, precision, sensitivity, specificity, and F1 score. In more detailed analysis, average training time, number of support vectors, and Receiver Operating Characteristic (ROC) curve can be considered. The first set of metrics can be obtained through the confusion matrix similar to Figure 3.9. The confusion matrix in itself is not a performance measure, but several useful performance metrics are based on the confusion matrix and the numbers inside it. For our case in Figure 3.9, 'Red' refers to a positive subject and 'Blue' refers to a negative subject.

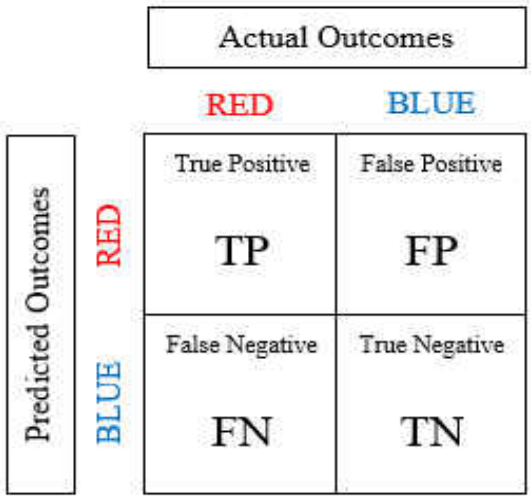


Figure 3.9: The format of confusion matrix for synthetic1

1. **Accuracy**

   The accuracy in classification problems is the number of correct predictions made by the model over all the predictions made. The accuracy is a strong metric if the classes in the dataset are nearly balanced.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.29}$$

2. **Precision**

   The precision a measures of the conditional probability of making a true positive decision given that the decision made is positive.

$$Precision = \frac{TP}{TP + FP} \tag{3.30}$$

3. **Sensitivity**

The sensitivity also referred as recall is a measure of the probability at which the actual positive subject is classified as positive.

$$Sensitivity = \frac{TP}{TP + FN} \tag{3.31}$$

4. **Specificity**

   The specificity is the exact opposite of the sensitivity. It is a measure of the probability at which the actual negative subject is classified as negative.

$$Specificity = \frac{TN}{TN + FP} \tag{3.32}$$

5. **F1 Score**

   The F1 Score is designed to represent both precision and sensitivity, hence the harmonic mean of precision and sensitivity.

$$F1\,Score = \frac{2 \times Precision \times Sensitivity}{Precision + Sensitivity} \tag{3.33}$$

The following output file corresponds to the confusion matrix for classification of synthetic1.

```
Confusion Matrix and Statistics

          Reference
Prediction blue red
      blue   14   3
      red     1  12

               Accuracy : 0.8667
                 95% CI : 0.6928, 0.9624
    No Information Rate : 0.5
```

```
    P-Value [Acc > NIR] : 2.974e-05

                  Kappa : 0.7333
 Mcnemar's Test P-Value : 0.6171

            Sensitivity : 0.9333
            Specificity : 0.8000
         Pos Pred Value : 0.8235
         Neg Pred Value : 0.9231
             Prevalence : 0.5000
         Detection Rate : 0.4667
   Detection Prevalence : 0.5667
      Balanced Accuracy : 0.8667

       'Positive' Class : blue
c@FancyVerbLinee' Class : blue
```

The ROC curve is a fundamental tool for evaluation of binary classification. In a ROC curve the true positive rate (Sensitivity) is plotted against the false positive rate (1 - Specificity) for different cut-off points in classification. Each point on the ROC curve represents an ordered pair (sensitivity, specificity) corresponding to a particular decision threshold. The area under the ROC curve (AUC) is a measure of how well a classification can distinguish between two classes. The AUC measures the quality of the model's predictions irrespective of what classification threshold is chosen.

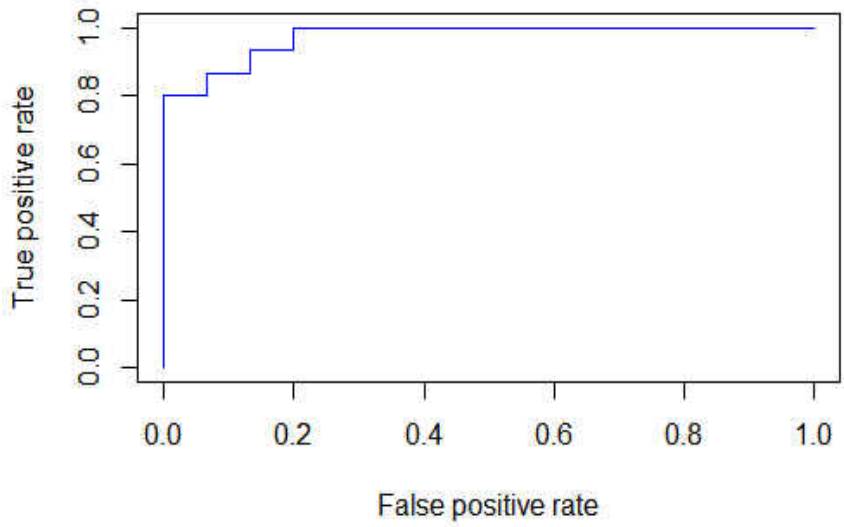


Figure 3.10: Receiver Operating Characteristic (ROC) curve for synthetic1

CHAPTER 4

MULTICLASS SUPPORT VECTOR MACHINES

SVMs are originally designed for binary classification [11]. However, it can be effectively extended to multiclass classification by breaking down the multiclass classification problem into a series of binary classification problems. This technique is known as divide and conquer in computer science and still an active research area.

The three most popular multiclass SVM methods: One Vs Rest, One Vs One and Directed Acyclic Graph Support Vector Machine will be discussed in this section.

4.1 One Versus Rest Support Vector Machine

The one-vs-rest method construct k number of SVM models where k is the number of classes.

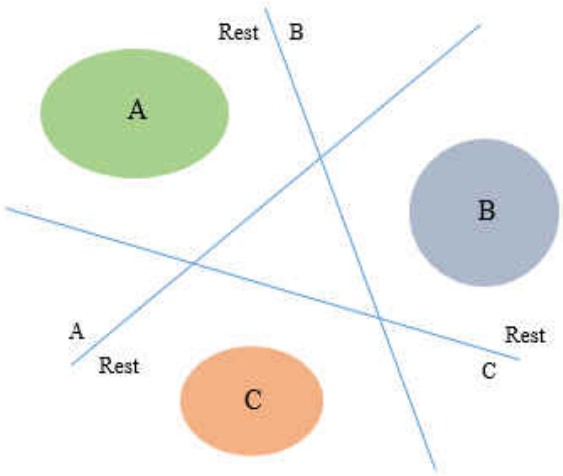


Figure 4.1: One versus rest method: classification between 3 classes

The rth SVM is trained with all of the instances in the rth class with positive labels while the rest of instances with negative labels. Thus, given n training data ($\mathbf{x}_1$, y_1), ...,($\mathbf{x}_n$,

y_n); where, $\mathbf{x}_i \in \mathbb{R}^n$, i = 1,...,$n$ and $y_i \in \{1,\ldots,k\}$ is the class of $\mathbf{x}_i$. The r^{th} SVM solves the following quadratic optimization problem [2]:

$$\min \frac{1}{2}(w^r)^T w^r + C\sum_{i=1}^{n} \xi_i^r \tag{4.1}$$

subject to

$$(\mathbf{w}^r)^T \Phi(\mathbf{x}_i) + b^r \geq 1 - \xi_i^r \text{ ; if } y_i = r,$$

$$(\mathbf{w}^r)^T \Phi(\mathbf{x}_i) + b^r \leq -1 + \xi_i^r \text{ ; if } y_i \neq r.$$

Where, $\xi_i^r \geq 0$, and i = 1, ...,n. Solving (4.1), leads to k decision functions.

We can say that $\mathbf{x}_i$ belongs to the class that has the greatest value of the decision function [2]. A well-known drawback of the one-vs-rest method is the highly imbalanced training set for each binary classifier. Assuming equal number of training instances for all classes, the ratio of positive to negative instances for each binary classifier is $1/(k-1)$. Hence, the original problem loses the symmetry of classification, affecting the results; especially, sparse classes may suffer seriously [12, 16, 18].

4.2 One Vs One Support Vector Machine

The one-vs-one method aims to get rid of the imbalance problem of the one-vs-rest method by training binary classifiers only with the data belong to two original classes designated by each classifier. The one-vs-one method constructs $k(k-1)/2$ given that k is the number of classes [17, 19, 20].

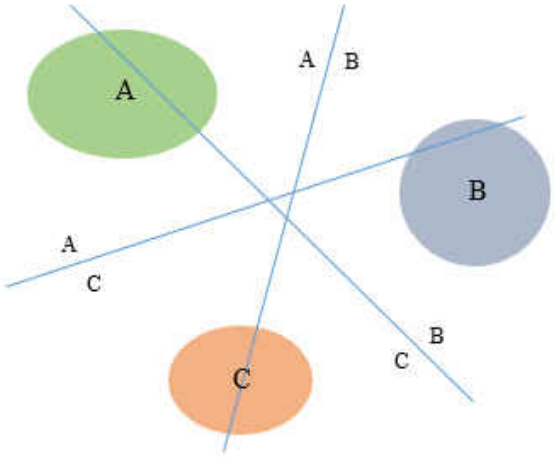


Figure 4.2: One against one method: classification between 3 classes

i-vs-j classifier solves the following quadratic optimization problem.

$$\min \frac{1}{2}(w^{ij})^T w^{ij} + C \sum_{i=1}^{n} \xi_i^{ij} \tag{4.2}$$

subject to

$$(\mathbf{w}^{ij})^T \Phi(\mathbf{x}_t) + \mathrm{b}^{ij} \geq 1 - \xi_i^{ij} \text{ ; if } \mathrm{y}_t = \mathrm{i},$$

$$(\mathbf{w}^{ij})^T \Phi(\mathbf{x}_t) + \mathrm{b}^{ij} \leq -1 + \xi_i^{ij} \text{ ; if } \mathrm{y}_t = \mathrm{j}.$$

Where, $\xi_t^{ij} \geq 0$, and t = 1, . . . ,n

After all classifiers are constructed, we can use the following voting strategy: If the sign of i-vs-j classifier says that $\mathbf{x}_t$ is in the i^{th} class, then the vote for the i^{th} class is increased by 1. Otherwise, the vote for the j^{th} class is increased by 1. Finally, we conclude that $\mathbf{x}_t$ is in the class with the highest vote. This type of voting approach is called the ”Max Wins” strategy. In the case that votes for two classes tie, we simply select the one with the smaller index [2].

4.3 Directed Acyclic Graph Support Vector Machine

The Directed Acyclic Graph Support Vector Machine is constructed by introducing one-vs-one classifiers into the nodes of a decision Directed Acyclic Graph (DAG).

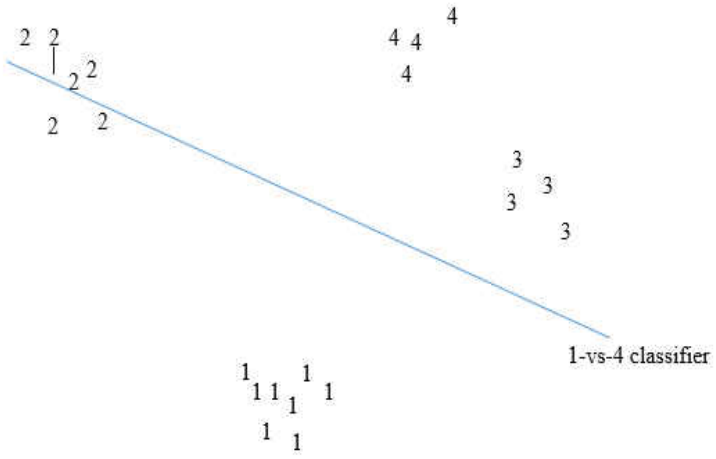


Figure 4.3: Input space of a 4 class problem

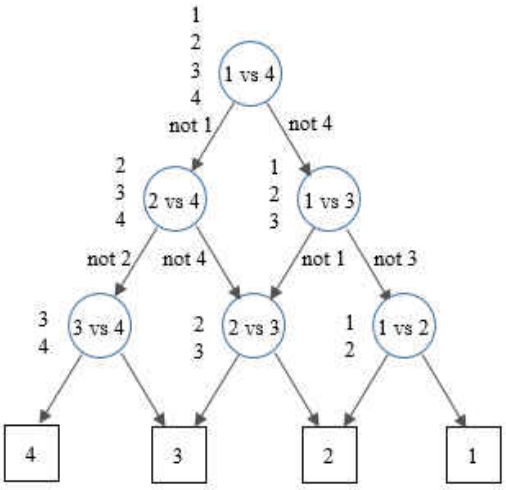


Figure 4.4: Directed acyclic graph support vector machine: 4 class problem

Definition 4.1. A decision DAG on k number of classes over a set of Boolean functions $F = \{f : X \mapsto \{0, 1\}\}$ for a given space X is a function that uses a rooted binary DAG with k leaves labeled with the classes where each of the $k(k\text{-}1)/2$ internal nodes is labeled with an element of F. The nodes are setup in triangular shape with a root node at the top two nodes in the 2^{nd} layer and so on. The final layer has k leaves. The i^{th} node in layer $j < k$ is joined to the i^{th} and $(i + 1)^{st}$ node in the $(j + 1)^{st}$ layer [6].

For an input x $\in X$, of a decision DAG, starting at the root node, each node is exited via the left edge, if the binary function is 0 or the right edge, if the binary function is 1, until x meets a leaf. The corresponding output of the decision DAG is the value associated with the leaf. This particular path taken through the decision DAG is called evaluation path [6]. In the case of the DAGSVM, we use binary values -1 and +1. The DAGSVM has the following properties.

1. Nodes of the DAGSVM are maximum margin classifiers over a kernel induced feature space. Hence maximizing the margin of every node of the decision DAG results in a smaller generalization error bound.

2. It is safe to ignore the losing class at each binary decision, since all of the instances of the losing class are far away from the decision boundary as shown in Figure 4.3.

3. For the DAGSVM, the order of choosing the binary classifiers is arbitrary. Limited experimentations show that changing this order does not significantly affect the accuracy of the algorithm.

4. The DAGSVM is superior to other two multiclass support vector machine algorithms in both training and evaluation time.

CHAPTER 5

FAST MULTICLASS CLASSIFICATION

5.1 CURSE OF DIMENSIONALITY

The curse of dimensionality refers to various issues related to analyzing and organizing data in very high-dimensional spaces. The expression was coined by Richard E. Bellman in a highly acclaimed article considering problems in dynamic optimization [13, 14]. In general, as dimensionality increases, the volume of the space increases rapidly and the data becomes sparser. The sparsity of data is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data needed often grows exponentially with the dimensionality, which prevents common data mining techniques from being efficient; often leading to over-fitting.

Recall that the SRM provides tools to investigate the generalization ability of the SVM under the curse of dimensionality. Since the VC dimension of the SVM classifier grows with the dimensionality of the data points, from (2.3) we can assume that the VC confidence also increases with the dimensionality of the data points. Consider the following example.

Example 5.1. *Figure 5.1 shows how the VC confidence varies with VC dimension/number of training data (h/n). For of 95% confidence level ($\eta = 0.05$) and a training sample of size 10,000, the VC confidence is an increasing function of h/n.*

Hence, it is clear that when the dimensionality increases relative to the number of training instances, the upper bound of the true risk increases. As a result, the SVM suffers from over-fitting.

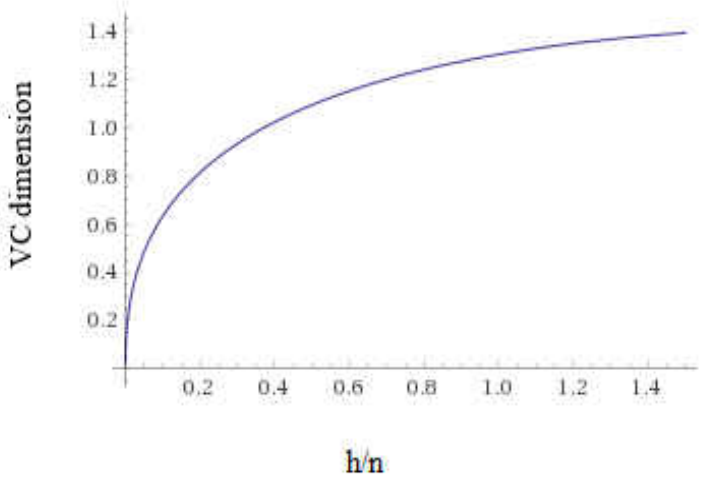


Figure 5.1: VC confidence against VC dimension/number of training data

5.2 Divide and Conquer Support Vector Machine (DCSVM)

In this section we present a SVM-based multiclass classification method called the Divide and Conquer Support Vector Machine (DCSVM) that exploits the curse of dimensionality to efficiently perform classification of high dimensional data.

The DCSVM algorithm idea is based on the following simple observation, best described using the example in Figure 5.2. The figure shows 6 classes (1 – red, 2 – blue, 3 – green, 4 – black, 5 – orange, and 6 – maroon) of two-dimensional points and a linear SVM separation of classes 1 and 2 (the line that separates the points in these classes). It happens that the SVM model for classifying classes 1 and 2 completely separates the points in classes 4 (which takes class 2 side) and 6 (which takes class 1 side). Moreover, the classifier does a relatively good job classifying most points of the class 5 as class 2 (with relatively few points classified as 1) and a poor job on classifying the points of class 3 (as the points in this class are classified about half as 1's and the other half as 2's). With

DCSVM we use the SVM classifier for classes 1 and 2 for a candidate of an unknown class: if the classifier predicts 1, then we next decide between classes 1, 6, 3, and 5; if the classifier predicts class 2, then we next decide between classes 2, 4, 3, and 5. Notice that in either case one or more classes are eliminated and we are left to predict a fewer classes. That is, a multi-class classification problem of a smaller size (less classes). The algorithm then proceeds recursively on the smaller problem. Given that the number of classes is k, in the best case scenario, half of the classes will be eliminated at each step and the algorithm will finish in $\log_2 k$ steps. Notice that, in the above scenario, classes 2 and 4 are completely separated from classes 1 and 6, whereas classes 3 and 5 are not clearly on one side or the other of the separation line. For this reason, classes 3 and 5 are part of the next decision step, regardless of the prediction of the first classifier.

However, there is a significant difference between classes 3 and 5. While class 3 is almost divided in half by the separation line, class 5 can be predicted as "2" with a very small error. In DCSVM we use a threshold value to indicate the maximum classification error accepted in order to consider a class on one side or the other of a separation line. For instance, let us consider that only 2% of the points of class 5 are on the same side as class 1. With the threshold value set to 0.02, DCSVM will separate classes 1, 3, and 6 (when 1 is predicted) from classes 2, 3, 4, and 5 (when 2 is predicted). A higher threshold value will produce a better separation of classes (less overlapping) and less classes to process in the subsequent steps. This comes at the price of possibly sacrificing the accuracy of the final prediction. Clearly, the method presented in the example above is in general suitable for multiclass classification using a binary classifier. Our choice of using SVM is based on the SVM algorithm's remarkable power in producing accurate binary classification.

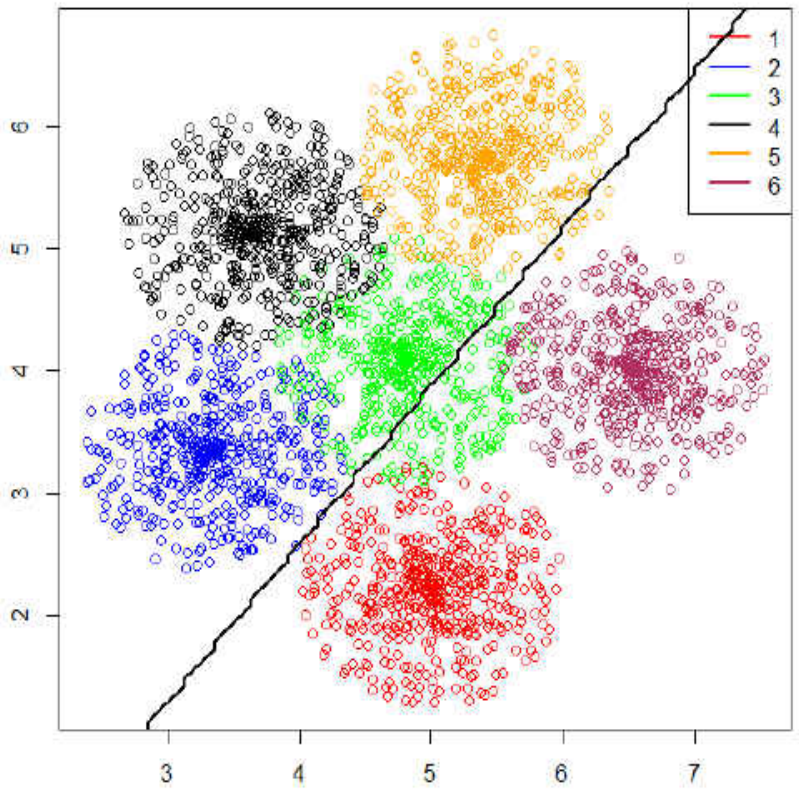


Figure 5.2: Seperation of classess 1 and 2 by a linear SVM classifier

5.2.1 Architecture of DCSVM

Let us first introduce some notations, next we will proceed to the formal description of the algorithm. Given a data set of k classes (labels) D where each data item $\mathbf{x} \in D$ has been assigned a label $l \in \{1, \ldots, k\}$, we want to construct a decision function $dcsvm : D \to \{1, \ldots, k\}$ so that $dcsvm(\mathbf{x}) = l$, where l is the corresponding label of $\mathbf{x} \in D$. By considering a split $D = R \cup T$ of the data set D into two disjoint sets R (the training set) and T (the test set), we use the data in R to construct our decision function $dcsvm()$ and then the data in T to measure its accuracy. Furthermore, we consider $R = R_1 \cup R_2 \cup \cdots \cup R_k$

as an union of disjoint sets R_l, where each $\mathbf{x} \in R_l$ has label l, $l = 1, \ldots, k$. (Similarly, we consider $T = T_1 \cup T_2 \cup \cdots \cup T_k$ as an union of disjoint sets T_l, where each $\mathbf{x} \in T_l$ has label l, $l = 1, \ldots, k$.)

Let $svm_{i,j} : D \rightarrow \{i, j\}$, be a SVM binary classifier created using the training set $R_i \cup R_j$, $i < j$ and $i = 1, \ldots, k-1, j = 2, \ldots, k$. There are $k(k-1)/2$ such one-versus-one binary classifiers. It is important to mention that the $svm()$ decision function considered here is not the ideal one, but the practical one, likely affected by misclassification errors; that is, for some $\mathbf{x} \in R_i \cup T_i$, we may have that $svm_{i,j}(\mathbf{x}) = j$.

Our goal is to create the $dcsvm()$ decision function that uses a minimal number of binary decisions for k-classes classification, while not sacrificing the classification accuracy. Next, we define a few measures we use in the process of identifying the shortest path to a multi-class classification decision.

Definition 5.1. (Class Predictions Likelihoods) The class predictions likelihoods of a SVM binary classifier $svm_{i,j}(\cdot)$ for a label $l \in \{1, \ldots, k\}$, denoted respectively as $\mathcal{C}_{i,j}(l, i)$ and $\mathcal{C}_{i,j}(l, j)$, are:

$$\mathcal{C}_{i,j}(l, i) = \frac{|\{\mathbf{x} \in R_l \mid svm_{i,j}(\mathbf{x}) = i\}|}{|R_l|}$$
$$\mathcal{C}_{i,j}(l, j) = \frac{|\{\mathbf{x} \in R_l \mid svm_{i,j}(\mathbf{x}) = j\}|}{|R_l|} = 1 - C_{i,j}(l, i) \tag{5.1}$$

Each class prediction likelihood represents the expected outcome likelihood for i or j when a binary classifier $svm_{i,j}(\cdot)$ is used for prediction on all data items in R_l. These likelihoods are computed for each binary classifier and each class in the training data set.

All pairs of likelihood predictions, for every binary classifier $svm_{i,j}(\cdot)$ and classes are stored in a table, called All-Prediction table.

Definition 5.2. (*All-Prediction* Table) We arrange all classes predictions likelihoods in rows (corresponding to each binary classifier $svm_{i,j}$) and columns (corresponding to each

class $1, \ldots, k$) to form a table $\mathcal{T}$ where each entry is given by a pair of predictions likelihoods as follows:

$$\mathcal{T}[svm_{i,j}, l] = (\mathcal{C}_{i,j}(l, i), \mathcal{C}_{i,j}(l, j)) \tag{5.2}$$

Fig 5.2 shows the *All-Predictions* table computed for the glass dataset [15]. The dataset contains 6 classes, labeled as 1, 2, 3, 5, 6, and 7. Each row corresponds to a binary classifier $svm_{1,2}, \cdots, svm_{6,7}$ and columns correspond to each class label. Each table cell contains a pair of likelihood predictions (as percentages) for the row classifier and class column. For instance, $\mathcal{C}_{1,2}(1, 1) = 100\%$, $\mathcal{C}_{1,2}(1, 2) = 0\%$ and $\mathcal{C}_{1,6}(2, 1) = 91.8\%$, $\mathcal{C}_{1,6}(2, 6) = 8.2\%$.

	svm	1	2	3	5	6	7
1	1 vs. 2	1=100%; 2=0%	1=0%; 2=100%	1=0%; 2=100%	1=0%; 2=100%	1=0%; 2=100%	1=0%; 2=100%
2	1 vs. 3	1=100%; 3=0%	1=59.02%; 3=40.98%	1=0%; 3=100%	1=58.33%; 3=41.67%	1=14.29%; 3=85.71%	1=9.09%; 3=90.91%
3	1 vs. 5	1=100%; 5=0%	1=86.89%; 5=13.11%	1=100%; 5=0%	1=0%; 5=100%	1=42.86%; 5=57.14%	1=4.55%; 5=95.45%
4	1 vs. 6	1=100%; 6=0%	1=91.8%; 6=8.2%	1=100%; 6=0%	1=50%; 6=50%	1=0%; 6=100%	1=22.73%; 6=77.27%
5	1 vs. 7	1=100%; 7=0%	1=86.89%; 7=13.11%	1=93.33%; 7=6.67%	1=0%; 7=100%	1=14.29%; 7=85.71%	1=0%; 7=100%
6	2 vs. 3	2=100%; 3=0%	2=100%; 3=0%	2=0%; 3=100%	2=91.67%; 3=8.33%	2=71.43%; 3=28.57%	2=90.91%; 3=9.09%
7	2 vs. 5	2=100%; 5=0%	2=100%; 5=0%	2=100%; 5=0%	2=0%; 5=100%	2=28.57%; 5=71.43%	2=0%; 5=100%
8	2 vs. 6	2=100%; 6=0%	2=100%; 6=0%	2=66.67%; 6=33.33%	2=91.67%; 6=8.33%	2=0%; 6=100%	2=22.73%; 6=77.27%
9	2 vs. 7	2=100%; 7=0%	2=100%; 7=0%	2=100%; 7=0%	2=91.67%; 7=8.33%	2=28.57%; 7=71.43%	2=0%; 7=100%
10	3 vs. 5	3=40.74%; 5=59.26%	3=93.44%; 5=6.56%	3=100%; 5=0%	3=0%; 5=100%	3=0%; 5=100%	3=0%; 5=100%
11	3 vs. 6	3=100%; 6=0%	3=100%; 6=0%	3=100%; 6=0%	3=50%; 6=50%	3=0%; 6=100%	3=100%; 6=0%
12	3 vs. 7	3=100%; 7=0%	3=96.72%; 7=3.28%	3=100%; 7=0%	3=33.33%; 7=66.67%	3=28.57%; 7=71.43%	3=0%; 7=100%
13	5 vs. 6	5=100%; 6=0%	5=100%; 6=0%	5=100%; 6=0%	5=100%; 6=0%	5=0%; 6=100%	5=0%; 6=100%
14	5 vs. 7	5=100%; 7=0%	5=100%; 7=0%	5=100%; 7=0%	5=100%; 7=0%	5=0%; 7=100%	5=0%; 7=100%
15	6 vs. 7	6=83.33%; 7=16.67%	6=91.8%; 7=8.2%	6=86.67%; 7=13.33%	6=66.67%; 7=33.33%	6=100%; 7=0%	6=0%; 7=100%

Figure 5.3: Prediction plan of glass for threshold = 0.05

Next, we define two measures for the quality of the classification of each $svm_{i,j}(\cdot)$. The Impurity Index measures how good the binary classifier is for classifying all classes as i or j for a given likelihood threshold θ. In a nutshell, a class l is classified as "definitely" i by $svm_{i,j}(\cdot)$ if $\mathcal{C}_{i,j}(l, i) \geq 1 - \theta$; as "definitely" j if $\mathcal{C}_{i,j}(l, j) \geq 1 - \theta$; otherwise, it is classified as "undecided" i or j. The Impurity Index counts how many "undecided"

decisions a binary classifier produces. The lower the index, the better the separation. The Balance Index measures how "balanced" a separation is in terms of the number of classes predicted as i and j. The larger the index, the better the separation.

Definition 5.3. (SVM Impurity and Balance Indexes) For an likelihood threshold θ of a SVM classifier $svm_{i,j}(\cdot)$, we define:
the Impurity Index, denoted as $\mathcal{P}_{i,j}(\theta)$, as:

$$\mathcal{P}_{i,j}(\theta) = \left(\sum_{l=1}^{k} (\chi_\theta(\mathcal{C}_{i,j}(l,i)) + \chi_\theta(\mathcal{C}_{i,j}(l,j))) \right) - k \tag{5.3}$$

the Balance Index, denoted as $\mathcal{B}_{i,j}(\theta)$, as:

$$\mathcal{B}_{i,j}(\theta) = min \left(k - \sum_{l=1}^{k} \chi_\theta(\mathcal{C}_{i,j}(l,j)), k - \sum_{l=1}^{k} \chi_\theta(\mathcal{C}_{i,j}(l,i)) \right) \tag{5.4}$$

where χ_θ is the step function:

$$\chi_\theta(x) = \begin{cases} 1 & \text{if } x > \theta \\ 0 & \text{if } x \leq \theta \end{cases}$$

For instance, the Impurity Index for row $svm_{1,6}$ and threshold $\theta = 0.05$ in Fig 5.2 is:

$$\mathcal{P}_{1,6}(0.05) = ((1+0) + (1+1) + (1+0) + (1+1) + (0+1) + (1+1)) - 6 = 3$$

and indicates that 3 of the classes (namely 2, 5, and 7) are undecided when the required precision is at least $\theta = 5\%$.

For likelihood threshold $\theta = 0.05$, the Balance Index for row $svm_{1,2}$ in Fig 5.2 is $\mathcal{B}_{1,2} = 1$ and for row $svm_{5,6}$ is $\mathcal{B}_{5,6} = 2$.

The SVM Score, defined below, is a measure of the sensitivity of the binary classifier $svm_{i,j}(\cdot)$ for classifying classes i and j. The higher the Score the better the classifier.

Definition 5.4. (SVM Score) The Score of a SVM classifier $svm_{i,j}(\cdot)$ is its sensitivity, denoted by $\mathcal{S}_{i,j}$, is

$$\mathcal{S}_{i,j} = \frac{\mathcal{C}_{i,j}(i,i) + \mathcal{C}_{i,j}(j,j)}{2} \tag{5.5}$$

For instance, the table in Fig 5.2 shows that

$$\mathcal{S}_{1,2} = \frac{\mathcal{C}_{1,2}(1,1) + \mathcal{C}_{1,2}(2,2)}{2} = \frac{100\% + 100\%}{2} = 100\%.$$

Algorithm 1 DCSVM training

1: **procedure** TRAINDCSVM($R_1, \ldots, R_k, \theta$) ▷ Creates DCSVM classifier
2: **Input**: $R = R_1, \ldots, R_k$: data set, θ: likelihood threshold
3: **Output**: $dcsvm()$
4: $svm_{i,j} \leftarrow$ train SVM with $R_i \cup R_j$, $i = 1, \ldots, k-1$, $j = 2, \ldots, k$, $i < j$
5: $\mathcal{T}[svm_{i,j}, l] = (\mathcal{C}_{i,j}(l,i), \mathcal{C}_{i,j}(l,j))$, for all $svm_{i,j}$, $l = 1, \ldots, k$
6: //Recursively construct a binary decision tree
7: //with each node associated with a $svm_{i,j}$ binary classifier
8: $dcsvm \leftarrow$ empty binary tree
9: $dcsvm.root \leftarrow$ new tree node
10: DCSVM-SUBTREE($dcsvm.root, \mathcal{T}, \theta$)
11: **return** $dcsvm$ ▷ Returns the decision tree
12: **end procedure**

Algorithm 1 and 2 describe DCSVM training and proceeds as follows. In the main procedure, TRAINDCSVM, the SVM binary classifiers for all class pairs are trained (line 4) and the predictions likelihoods are stored in the predictions table (line 5). The decision function $dcsvm$ is created as an empty tree (line 8) and then recursively populated in DCSVM-SUBTREE procedure (line 10). The recursion procedure creates a left and/or a right node at each step (lines 12 and 19, respectively) or may stop with creating a left and/or a right label (lines 9 and 16, respectively). Each new node is associated to the binary $svm_{i,j}$ that is the decider at that node (line 5), or with a class label if an end node (lines 9 and 16).

An important part of the DCSVM-SUBTREE procedure is choosing the "optimal" svm from a current predictions likelihoods table (line 4). For this purpose we use the

Algorithm 2 DCSVM training (continued)

procedure DCSVM-SUBTREE($pnode, \mathcal{T}, \theta$) ▷ Creates subtree routed at $pnode$
Input: $pnode$: current parent node, $\mathcal{T}$: current predictions table, θ: likelihood threshold
Output: recursively constructs sub-tree rooted at $pnode$
    $svm_{i,j} \leftarrow$ optimal svm in $\mathcal{T}$, for given θ
    $pnode[svm] \leftarrow svm_{i,j}$
    $listi \leftarrow$ classes labeled as i or undecided by $svm_{i,j}$
    $listj \leftarrow$ classes labeled as j or undecided by $svm_{i,j}$
    if $length(listi) = 1$ **then** ▷ reached a leaf
        $pnode.leftnode \leftarrow$ tree-node(label in $listi$)
    else
        $\mathcal{T}.left \leftarrow \mathcal{T}$ minus $svm_{m,n}, m \in listj$ or $n \in listj$ rows, and columns of classes not in $listi$
        $pnode.left \leftarrow$ new tree node
        DCSVM-SUBTREE($pnode.left, \mathcal{T}.left, \theta$)
    end if
    if $length(listj) = 1$ **then** ▷ reached a leaf
        $pnode.rightnode \leftarrow$ tree-node(label in $listj$)
    else
        $\mathcal{T}.right \leftarrow \mathcal{T}$ minus $svm_{m,n}, m \in listi$ or $n \in listi$ rows, and columns of classes not in $listj$
        $pnode.right \leftarrow$ new tree node
        DCSVM-SUBTREE($pnode.right, \mathcal{T}.right, \theta$)
    end if
end procedure

SVM Impurity Index, Balance Index, and Score from Definitions 9 and 10, respectively. The order these measures are used may influence the decision tree shape and precision. If Score is used then the Impurity and Balance Indexes are used to break a tie, the resulting tree favors accuracy over the speed of decisions (may yield bushier trees). If Impurity and Balance Indexes are used first, then Score, if a tie, the resulting tree may be more balanced. The decision speed is favored while possibly sacrificing accuracy.

A $dcsvm$ decision tree for the *glass* data set is shown in. Clearly, the algorithm may produce highly unbalanced $dcsvm$ decision trees (when some classes are decided faster than others) or very balanced decision trees (when most of class labels are leaves situated at about same depth). Regardless of outcome, the following result is almost immediate.

Proposition 5.2. *The $dcsvm$ decision tree constructed in Algorithm 2 has depth at most k.*

Proof. The lists of classes labeled i and j (lines 6, 7 in DCSVM-SUBTREE procedure) contain at least one label each: i and j, respectively. Once a class column is removed from $\mathcal{T}$ at some tree node n, it will not appear again in a node or leaf in the subtree rooted at that node n. Hence with each recursion the number of classes decreases by at least one (lines 11, 18) from k to 1, ending the recursion with a left or a right label node in lines 9 or 16, respectively. □

Notice that a scenario where each $dcsvm$ decision tree label has depth k is possible in practice: when no $svm_{i,j}$ binary classifier is a good separator for classes other than i and j (and therefore at each node only classes i and j are separated, while the other are undecided and will appear in both left and right branches). We call this the worst case scenario, for obvious reasons. The opposite case scenario is also possible in practice: each $svm_{i,j}$ separates all classes into two disjoint lists of about same lengths. The $dcsvm$ decision tree is also very balanced in this case, but a lot smaller.

Proposition 5.3. *The $dcsvm$ decision tree constructed in Algorithm 2 when each $svm_{i,j}$ produced balanced, disjoint separation between all classes has depth at most $\lceil \log_2 k \rceil$.*

Proof. Clearly this is a case scenario where at each recursion step a node is created such that half of the classes are assigned to the left subtree and the other half to the right subtree. This produces a balanced binary tree with k leaves, hence of depth at most $\lceil \log_2 k \rceil$. □

Algorithm 3 DCSVM classifier

```
1:  procedure DCSVMCLASSIFY(dcsvm, x)          ▷ Produces DCSVM classification
2:  Input: dcsvm: decision tree; x: data item
3:  Output: Label of data item x
4:      node ← dcsvm.root
5:      while node not a leaf do          ▷ Visits the decision tree nodes towards a leaf
6:          svm_{i,j} ← node[svm]         ▷ Retrieves the svm associated to current node
7:          label ← svm_{i,j}(x)
8:          if label = i then
9:              node ← node.left
10:         else
11:             node ← node.right
12:         end if
13:     end while
14:     return label of node                          ▷ Returns the leaf label
15: end procedure
```

The DCSVM classifier Algorithm 3 relies on the $dcsvm$ decision tree produced by Algorithm 1 to take any data item **x** and predict its label. The algorithm starts at the decision tree root node (line 4) then each node's associated svm predicts the path to follow (lines 6–12) until a leaf node is reached. The label of the leaf node is the DCSVM's prediction (line 14) for the input data item **x**. An example of prediction path in a $dcsvm$ tree is illustrated in Fig 5.4 (b).

Propositions 5.2 and 5.2.1 directly justify the following result.

Theorem 5.4. *The Algorithm 3 performs multi-class classification of any data item* $\boldsymbol{x}$ *in at most* k *binary decisions steps (in the worst case scenario) and at most* $\lceil \log_2 k \rceil$ *binary decision steps (in the best case scenario).*

We illustrate next how the $dcsvm$ decision tree is created and how a prediction is computed using a working example.

5.2.2 A WORKING EXAMPLE

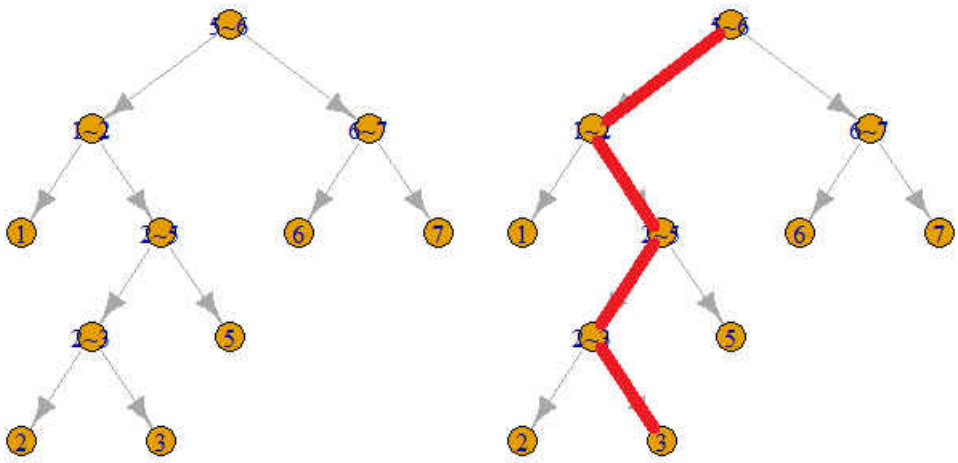
We use the *glass* dataset [15] to illustrate DCSVM at work. This data set contains 6 classes, labeled 1, 2, 3, 5, 6, and 7 (notice there is no label 4). Consequently, $6 * (6-1)/2 = 15$ binary svm classifiers are created and then the *All-Predictions* table $\mathcal{T}$ is computed as shown in Figure 5.3. Let us choose the likelihood threshold $\theta = 0$, for simplicity. That is, a class l is classified by an $svm_{i,j}$ as only i if $svm_{i,j}$ predicts that all data items in R_l have class i; l is classified as only j if $svm_{i,j}$ predicts that all data items in R_l have class j; else, l is undecided and will appear on both sides of the decision tree node associated to $svm_{i,j}$.

We follow next the DCSVM-SUBTREE procedure in Algorithm 2 and construct the $dcsvm$ decision tree. Notice that $S_{i,j} = 100\%$ for all $svm_{i,j}$, so Score does not matter for choosing the optimal $svm_{i,j}$ in line 4. The choice will be solely based on the Impurity and Balance indexes. Table 5.1 shows all values for these measures obtained form the initial *All-Predictions* table.

It is clear that rows 1, 13, and 14 as candidates with minimum Impurity Indexes. Then there is a tie between rows 13 and 14 as the winners among these. Row 13 comes first and hence $svm_{5,6}$ is selected as the root node. Figure 5.4 (a) shows the full decision tree, with $svm_{5,6}$ as the root node. Subsequently, $svm_{5,6}$ labels classes 1, 2, 3, and 5 as "5" (left), and classes 6 and 7 as "6" (right). The algorithm continues recursively with classes $\{1, 2, 3, 5\}$ to the left, and classes $\{6, 7\}$ to the right. The right branch will be completed immediately with one more tree node (for $svm_{6,7}$) and two corresponding leaf nodes (for labels 6 and

Table 5.1: svm optimality measures for *glass* dataset and the initial *All-Predictions* table

	$svm_{i,j}$	$\mathcal{P}_{i,j}(0)$	$\mathcal{B}_{i,j}(0)$	$\mathcal{S}_{i,j}$
1.	$svm_{1,2}$	0	1	100%
2.	$svm_{1,3}$	4	1	100%
3.	$svm_{1,5}$	3	1	100%
4.	$svm_{1,6}$	3	1	100%
5.	$svm_{1,7}$	3	1	100%
6.	$svm_{2,3}$	3	1	100%
7.	$svm_{2,5}$	1	2	100%
8.	$svm_{2,6}$	3	1	100%
9.	$svm_{2,7}$	2	1	100%
10.	$svm_{3,5}$	2	1	100%
11.	$svm_{3,6}$	1	1	100%
12.	$svm_{3,7}$	3	1	100%
13.	$svm_{5,6}$	0	2	100%
14.	$svm_{5,7}$	0	2	100%
15.	$svm_{6,7}$	4	1	100%



(a) (b)

Figure 5.4: DCSVM decision tree with (a) the decision process for *glass* dataset (b) the classification of an unseen instance

7).

For the left branch the algorithm will proceed with a reduced *All-Predictions* table: rows 4, 5, 8, 9, 11, 12, 13, 14, and 15 and columns for classes 6 and 7 are removed. The optimality measures will be subsequently computed for all svm and classes still in competition (1, 2, 3, and 5) in the left branch. The corresponding measures are given in Table 5.2 (for an easier identification, the indexes in the first column are kept the same as the original indexes in the *All-Predictions* table in Figure 5.3).

There is a tie between $svm_{1,2}$ and $svm_{2,5}$, and $svm_{1,2}$ is being used first. A node is consequently created, with a leaf as a left child. The rest of the tree is subsequently created in the same manner.

5.3 Experiments

We implemented DCSVM in R v3.4.3 using e1071 library [21], running on Windows 10, 64-bit Intel Core i7 CPU @3.40GHz, 16GB RAM. For testing we used 14 data sets from

Table 5.2: Optimality measures in the second step of creating the decision tree in Figure 5.3 (b)

	$svm_{i,j}$	$\mathcal{P}_{i,j}(0)$	$\mathcal{B}_{i,j}(0)$	$\mathcal{S}_{i,j}$
1.	$svm_{1,2}$	0	1	100%
2.	$svm_{1,3}$	2	1	100%
3.	$svm_{1,5}$	1	1	100%
6.	$svm_{2,3}$	1	1	100%
7.	$svm_{2,5}$	0	1	100%
10.	$svm_{3,5}$	2	1	100%

UCI repository [15] (listed in Table 5.3). For each experiment, we used cross-validation with 80% data for training and 20% for testing, for each data set. We ran 10 trials and averaged the results. We ran three sets of experiments: (i) multi-class prediction accuracy, (ii) prediction performance in terms of speed (time and number of binary decisions) and resources (number of support vectors), and (iii) DCSVM performance comparisons for different data sets and accuracy threshold parameter values. For the first set of experiments we compared three multi-class predictors: build-in multi-class SVM (LibSVM from the e1071 library), R implementation of One-Vs-One, and R implementation of DCSVM. For a fair comparison, in the second set of experiments we compared R implementations of One-Vs-One and DCSVM. The built-in multi-class SVM would benefit of the inherent speed of native code it relies on. Finally, the third set of experiments focused on the DCSVM's R implementation performance and fine tuning.

5.4 RESULTS

This section summarizes the results of the experiments described in the chapter 5.2.2. To calculate the accuracy of each experiment, the method in (3.29) is used. In graphs, two

different scales are used to illustrate the maximum disparity of information whenever required.

5.4.1 Comparison of Accuracy

The main goal of DCSVM is to improve multi-class prediction performance while not sacrificing the prediction accuracy. The first experimental results compare multi-class prediction accuracy of: (i) built-in SVM multi-class prediction (in the e1071 package), (ii) one-versus-one implementation in R, and (iii) DCSVM implementation in R. The results are displayed in Figure 5.5 and show no significant differences between the three methods.

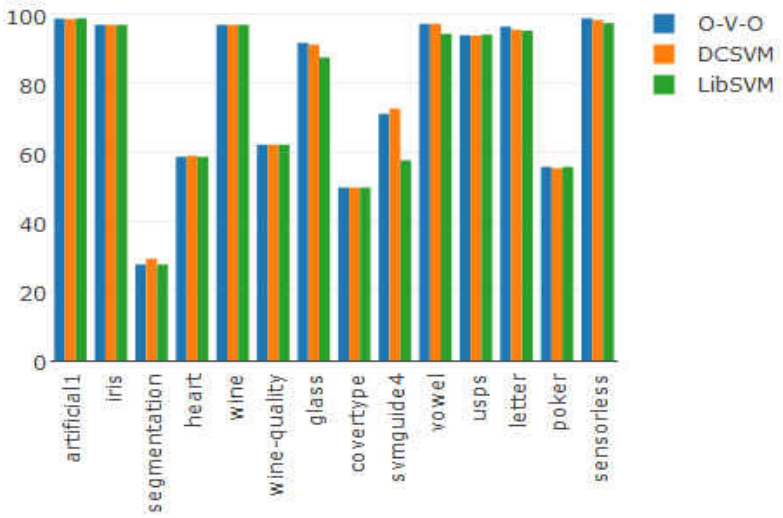


Figure 5.5: Average prediction accuracy (%)

5.4.2 Prediction Performance Comparison

For this purpose we compared the R implementations of One-Vs-One method and DCSVM. We analyzed prediction performance on three aspects: the average number of support vec-

tors, the average number of binary decisions, and average prediction time. The corresponding performance results are presented in Figure 5.6, Figure 5.7 and Figure 5.8 respectively. The number of support vectors used was computed by summing up all support vectors from every binary decider, over all steps of binary decisions until the multi-class prediction was achieved. The number of such support vectors is clearly proportional not only to the number of decision steps (which are illustrated separately), but also to the configuration of data separated by each binary classifier. It is clear that DCSVM significantly outperforms One-Vs-One, clearly being much less computationally intensive (number of support vectors for prediction) and faster (number of binary decisions and prediction times).

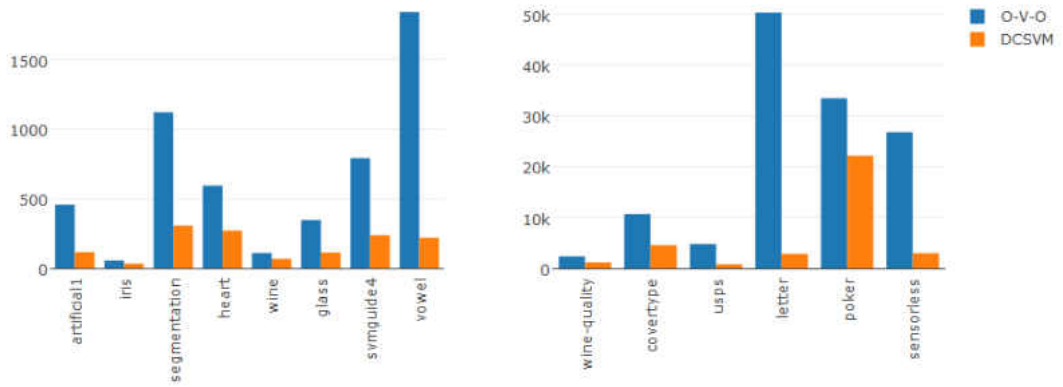


Figure 5.6: Average number of support vectors

5.4.3 DCSVM Performance Fine Tuning

In this set of experiments we analyze in close detail DCSVM's performance in terms of the likelihood threshold parameter. Figure 5.9 shows the trade-off between accuracy (left) and the average prediction steps (right) with various threshold values. Clearly, the likelihood threshold parameter permits a trade-off between accuracy and speed. However, this is largely data dependent. The more separable the data is, the less influence the threshold

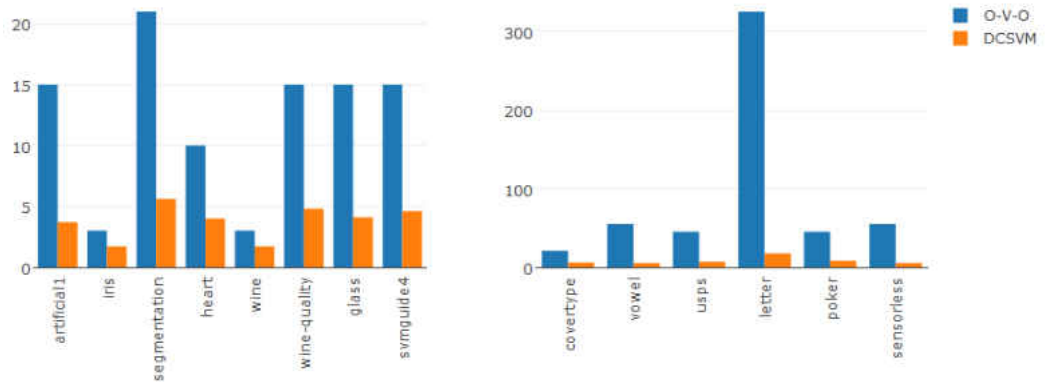


Figure 5.7: Average number of steps

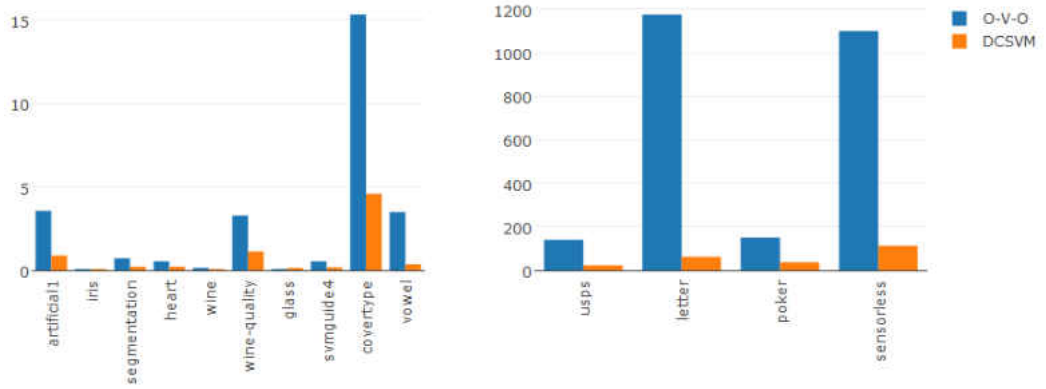


Figure 5.8: Average prediction time (sec)

has on speed. For less separable data (such as the letter data set), fine adjustment of the threshold permits trade-off between prediction accuracy and prediction speed. This is not the case for the vowel dataset, which is highly separable: changes in the threshold influence neither the accuracy of prediction nor the average number of prediction steps.

Table 5.4 shows side-by-side accuracies of multi-class classification using (i) built-in (BI), (ii) one-against-one (OvsO), and (iii) DCSVM (for a few threshold values θ). DCSVM performs very well in terms of accuracy (compared to the other methods) for all data sets,

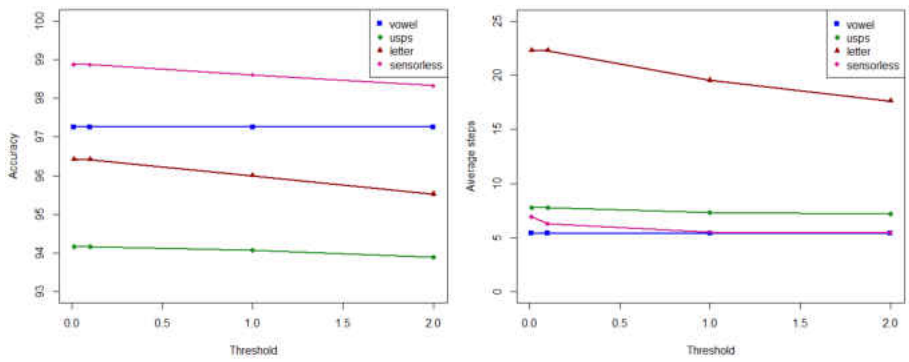


Figure 5.9: Accuracy and the average number of prediction steps for different likelihood thresholds

for threshold values $\theta \in \{2\%, 1\%, 0.1\%, 0.01\%\}$ (the larger the threshold, the better the accuracy, in general). A larger threshold θ may increase the prediction speed (Table 5.5) and reduce the computation effort (Table 5.6). Interesting to notice: Table 5.5 shows that in all cases displayed in the table the number of decision steps is less than $k-1$, where k is the number of classes in the respective data set. DCSVM outperforms (even for very small threshold) one-against-one and its improvement DAGSVM [6], which reaches multi-class prediction after $k-1$ steps.

The *All-Predictions* table in Figure 5.3 collects all information used by DCSVM to construct its multi-class prediction strategy (the dcsvm decision tree in Algorithm 1 and 2). The same information can be used to predict how much separation can be achieved for different threshold values. For instance, for the *glass* dataset *All-Predictions* table in Figure 5.3 and for a threshold value $\theta = 2\%$ there are 58 entries in the table where the percentage of predicting one class or the other is at least $100 - \theta = 98\%$ (out of a total of $15 \times 6 = 80$ entries in the table). The percentage $58/80 = 72.5\%$ is a good indicator of purity for DCSVM with threshold $\theta = 2\%$: the higher the percentage, the more separation is produced at each step and hence a shallow decision tree.

Figure 5.10 shows the class separation percentages for threshold values $0 \leq \theta \leq 5$

and four data sets (*letter*, *vowel*, *usps*, and *sensorless*). Intuitively, as threshold increases so does the separation percentage. The *letter* and *usps* data sets display an almost linear increase of separation with threshold. *sensorless* displays a sharp increase for small threshold values, then it tends to flatten, that is, not much gain for significant increase in threshold (and hence possibly less accuracy). Lastly, *vowel* displays a step-like behavior: not much gain in separation until threshold value reaches approximately $\theta = 2.3\%$, a steep increase until θ approaches 3%, then nothing much happens again. One can use these indicators to decide the trade-off between speed and accuracy of predictions.

Table 5.3: Prediction accuracies for different split thresholds

				DCSVM			
No	Dataset	LibSVM	O-V-O	$\theta = 2$	$\theta = 1$	$\theta = 0.1$	$\theta = 0.01$
1	artificial1	98.85	98.76	98.70	98.70	98.76	98.76
2	iris	96.97	96.97	96.97	96.97	96.97	96.97
3	segmentation	27.71	27.71	29.44	29.44	29.44	29.44
4	heart	58.82	58.82	59.12	59.12	59.12	59.12
5	wine	96.97	96.97	96.97	96.97	96.97	96.97
6	wine-quality	62.33	62.33	62.39	62.39	62.39	62.39
7	glass	87.55	91.70	91.29	91.29	91.29	91.29
8	covertype	49.95	49.95	49.95	49.95	49.95	49.95
9	svmguide4	57.88	71.21	72.73	72.73	72.73	72.73
10	vowel	94.34	97.26	97.26	97.26	97.26	97.26
11	usps	94.17	93.94	93.89	94.08	94.17	94.17
12	letter	95.25	96.43	95.52	96.00	96.41	96.41
13	poker	55.94	55.96	55.56	55.83	55.96	55.96
14	sensorless	97.46	98.87	98.32	98.60	98.86	98.87

Table 5.4: DCSVM: average number of steps per decision for different split thresholds

No	Dataset	$\theta = 2$	$\theta = 1$	$\theta = 0.1$	$\theta = 0.01$
1	artificial1	3.76	3.75	3.67	3.67
2	iris	1.71	1.71	1.71	1.71
3	segmentation	5.63	5.63	5.63	5.63
4	heart	4.00	4.00	4.00	4.00
5	wine	1.69	1.69	1.69	1.69
6	wine-quality	4.85	4.86	4.87	4.87
7	glass	4.09	4.12	4.12	4.12
8	covertype	5.93	5.93	5.93	5.93
9	svmguide4	4.63	4.88	4.88	4.88
10	vowel	5.41	5.41	5.41	5.41
11	usps	7.16	7.29	7.80	7.80
12	letter	17.63	19.56	22.29	22.29
13	poker	8.36	8.40	8.43	8.43
14	sensorless	5.44	5.49	6.28	6.93

Table 5.5: DCSVM: average support vectors per decision for different split thresholds

No	Dataset	$\theta = 2$	$\theta = 1$	$\theta = 0.1$	$\theta = 0.01$
1	artificial1	115.17	117.29	127.22	127.22
2	iris	32.47	32.47	32.47	32.47
3	segmentation	305.49	305.49	305.49	305.49
4	heart	270.18	270.18	270.18	270.18
5	wine	66.41	66.41	66.41	66.41
6	wine-quality	1154.49	1155.55	1157.13	1157.13
7	glass	112.14	114.37	114.37	114.37
8	covertype	4528.47	4528.47	4528.47	4528.47
9	svmguide4	236.58	245.71	245.71	245.71
10	vowel	218.36	218.36	218.36	218.36
11	usps	785.21	798.41	846.84	846.84
12	letter	2822.54	3110.42	3307.19	3307.19
13	poker	22166.49	22735.92	22803.01	22807.62
14	sensorless	2977.89	3057.77	3404.91	3735.30

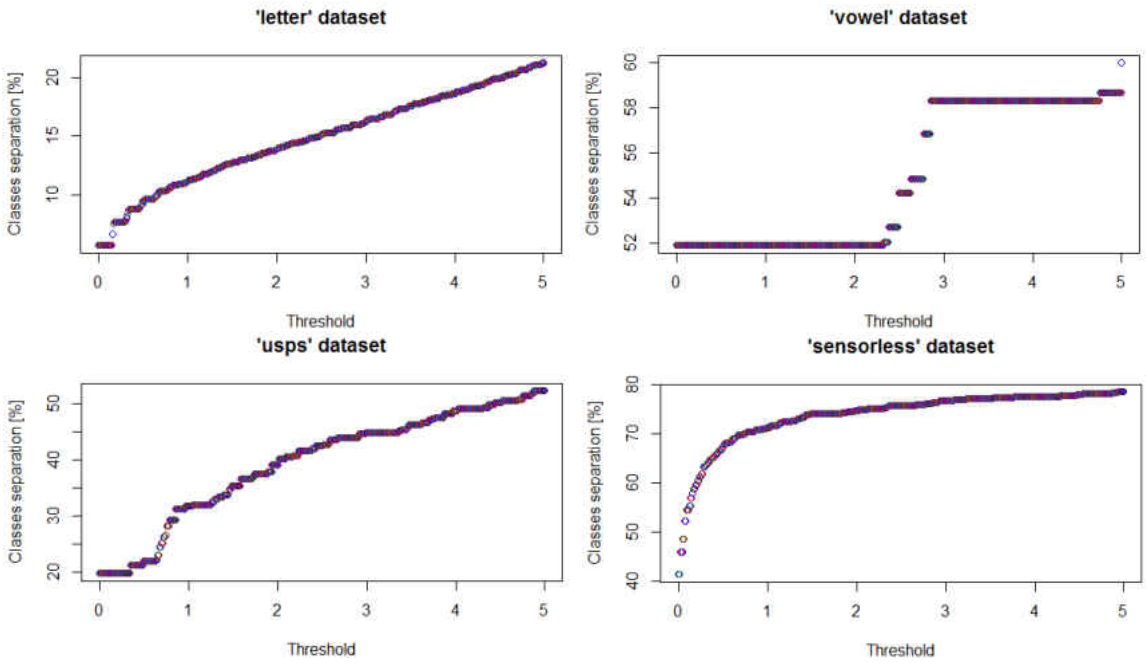


Figure 5.10: Number of separated classes for different thresholds

CHAPTER 6

CONCLUSION

This research presents a Divide and Conquer Support Vector Machine (DCSVM), an algorithm for fast multiclass classification using Support Vector Machines. In a nutshell, DCSVM divides the whole training dataset into two partitions based on a binary separation between two classes. Then, a prediction between the two classes eliminates one or more classes at the time. The algorithm continues recursively until a final binary decision is made between the last two classes left in the process [27].

All the experimental results are supportive of the robustness and efficiency of the DCSVM in classification. Even though the built-in LibSVM would benefit from of the inherent speed of source code it relies on, with respect to the prediction accuracy, the DCSVM slightly outperforms it in classifying some datasets. Our experiments suggest that the novel algorithm is significantly less expensive in computation and faster in classification. The average number of support vectors it makes and the average prediction time it takes is much less than that of the popular One-Vs-One method. Hence, the DCSVM would be desired in big data classification over other learning algorithms [26]. In future research, we plan to test the performance of DCSVM in classifying High Dimension Low Sample Size (HDLSS) datasets.

The SVM based divide and conquer technique we present for multiclass classification can be easily used with any binary classifier. It is rather a consequence of increasing data sparsity with the dimensionality of the space, which can be exploited, in general, in favor of producing fast multiclass classification using binary classifiers. Our experimental results on a few popular datasets show the applicability of the method and guarantee the classification in $k-1$ decision steps while preserving the accuracy. The major shortcoming of the current DCSVM algorithm is that it has a longer training time. We plan to minimize the training time by implementing general purpose programing languages such as Python and C.

REFERENCES

[1] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, *A practical guide to support vector classification*, Tech. report, Department of Computer Science, National Taiwan University (2003).

[2] Chih-W Hsu and Chih-J Lin, *A Comparison of Methods for Multi-class Support Vector Machines*, IEEE Transactions on Neural Networks 13, (2002), 415–425.

[3] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, *An Introduction to Statistical Learning with Applications in R*, Springer (2015).

[4] R-Bloggers, *Machine Learning Using Support Vector Machines*, https://www.r-bloggers.com/machinelearning-using-support-vector-machines/ (2017).

[5] Wikipedia, *Support Vector Machine*, https://en.wikipedia.org/wiki/Support vector machine (2017).

[6] S.A. Solla, T.K. Leen and K.-R. Müller/,(eds.), *Large Margin DAGs for Multiclass Classification*, MIT Press, (2000) 547–553.

[7] César Souza, *Kernek Functions for Machine Learning Applications*, http://crsouza.com/ (2010).

[8] Christopher J.C. Burges, *A Tutorial on Support Vector Machine for Pattern Recognition*, Data Mining and Knowledge Discovery, vol 2, (1998), 121–167.

[9] B.E. Boser, I.M. Guyon and V.N. Vapnik, *A Training Algorithm for Optimal Margin Classifiers*, COLT '92 Proceedings of the fifth annual workshop on Computational learning theory, (1992), 144–152.

[10] M. Mohammed, M.B. Khan and E.B.M. Bashier, *Machine Learning: Algorithms and Applications*, CRC Press, Boca Raton, (2017), 115–126.

[11] Corinna Cortes and Vladimir Vapnik, *Support-Vector Networks*, Machine Learning, (1995), 273–297.

[12] Vladimir Vapnik, *Statistical Learning Theory*, Wiley, (1998).

[13] R. Bellman and Rand Corporation, *Dynamic Programming*, Rand Corporation research study, Princeton University Press, (1957).

[14] Bellman, R.E. *Dynamic Programming*, Dover Books on Computer Science Series, Dover Publications, (2003).

[15] Dheeru, Dua and Karra Taniskidou, Efi, *UCI Machine Learning Repository*, http://archive.ics.uci.edu/ml.

[16] Daniel Silva-Palacios, Cèsar Ferri and María José Ramírez-Quintana, *Probabilistic class hierarchies for multiclass classification*, Journal of Computational Science, vol 26, (2018), 254–263.

[17] Knerr, Stefan and Personnaz, Léon and Dreyfus, Gérard, *Single-Layer Learning Revisited: A Stepwise Procedure for Building and Training a Neural Network*, in Neurocomputing: Algorithms, Architectures and Applications, vol F68 of NATO ASI Series, (1990), 41–50.

[18] Leon Bottou, Corinna Cortes, Denker, J. S., Harris Drucker, I. Guyon, L.D. Jackel, Yann Lecun, U.A. Muller, Eduard Sackinger, Patrice Simard and V. Vapnik, *Comparison of classifier methods: A case study in handwritten digit recognition*, in Proceedings of the International Conference on Pattern Recognition, IEEE, vol II, (1994), 77–82.

[19] Friedman, Jerome H., *Another approach to polychotomous classification*, Department of Statistics, Stanford University, (1996).

[20] Kreßel, Ulrich H.-G., *Advances in Kernel Methods*, Pairwise Classification and Support Vector Machines, MIT Press, (1999), 255–268.

[21] Dimitriadou, Evgenia and Hornik, Kurt and Leisch, Friedrich and Meyer, David and Weingessel, Andrea, *e1071: Misc Functions of the Department of Statistics (e1071), TU Wien. R package version 1.7-0.*, http://cran.r-project.org/package=e1071, (2018).

[22] Te Ming Huang anVojislav Kecman, *Bias Term b in SVMs Again*, ESANN in Proceedings - European Symposium on Artificial Neural Networks, (2004).

[23] Bilge Karacali, Rajeev Ramanath and Wesley E. Snyder, *A comparative analysis of structural risk minimization by support vector machines and nearest neighbor rule*, Pattern Recognition Letters, vol 25, (2004), 63–71.

[24] Roberto Battiti and Mauro Brunato, *Statistical Learning Theory and Support Vector Machines (SVM)* The LION way. Machine Learning plus Intelligent Optimization, Lionsolver Inc., Chapter 10, (2013).

[25] Vijay Pappu, and Panos M. Pardalos, *High Dimensional Data Classification* Industrial and Systems Engineering, University of Florida.

[26] Dingxian Wang, Xiao Liu and Mengdi Wang, *A DT-SVM Strategy for Stock Futures Prediction with Big Data*, 16th International Conference on Computational Science and Engineering, IEEE, 2013.

[27] Duleep R. Don and Ionut E. Iacob, *DCSVM: Fast Multiclass Classification Using Support Vector Machines*, https://arxiv.org/abs/1810.09828, 2018.

[28] John C. Platt, *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*, Microsoft Research, Technical Report MSR-TR-98-14, 1998.

Appendix A

DETAILS OF THE DATASETS

Except *artificial* the rest of the datasets are found at UCI machine learning repository [15]. In some cases, we considered a subset of instances in the original datasets for our experiments. Generation of *artificial* dataset is presented in Appendix section B.2.

Table A.1: Datasets

No	Dataset	Classes	Features	Instances
1.	artificial	6	3	3000
2.	iris	3	5	150
3.	segmentation	7	19	2310
4.	heart	5	14	303
5.	wine	3	13	178
6.	wine-quality	6	12	4898
7.	glass	6	11	214
8.	covertype	7	55	2000
9.	svmguide4	6	21	300
10.	vowel	11	11	528
11.	usps	10	257	7291
12.	letter	26	17	20000
13.	poker	10	11	25010
14.	sensorless	11	49	58509

Appendix B

R CODES OF THE EXPERIMENTS

This section includes all R programing language codes used in this study.

B.1 EXAMPLE 3.3 AND PROCEEDINGS

```
install.packages("e1071")
library(e1071)
#generate random normal distributions
set.seed(101)
x <- rnorm(100, 5, 1.5)
set.seed(110)
y <- rnorm(100, 4, 1)
#genrate and empty vector for categorial attribute
color <- rep(NA, 100)
#assign categorical variables
for (i in 1:100){
if (((x[i] > 4.0) & (x[i] < 6.0)) & ((y[i] > 3.0) & (y[i] < 5.0))){
color[i] = "red"} else {
color[i] = "blue"
   }
}
#make a bivariate normal dataset
data <- data.frame(x, y, color)
#plot the dataset
plot(y, x, col=data$color, xlim=c(1,6), ylim=c(1,8))
#make categorical variable a factor type
data$class <- factor(data$color)
data[,3] <- NULL
#randomly divide data into 70% for training, 30% for testing
set.seed(2018)
train <- sample(nrow(data), 0.7*nrow(data))
#store each fragment of data
data.train <- data[train,]
data.test <- data[-train,]
#tune SVMs (selection of parameters)
```

```
set.seed(2018)
best.para <- tune.svm(class~., data=data.train, kernel="radial",
gamma=2^(-8:6), cost=2^(13:14))
summary(best.para)
#do 10-fold cross validation on the training data to create the SVM
#classifier
svm.data <- svm(class~., data=data.train, kernel='radial', gamma=1,
cost=10, cross=10, probability = TRUE)
#Plot decision boundary
plot(svm.data, data, xlim=c(1,6), ylim=c(1,8))
#plot confusion matrix
library(caret)
svm.prob <- predict(svm.data, type="prob", newdata=data.test,
probability = TRUE)
print(confusionMatrix(svm.prob,data.test$class))
#plot a ROC curve
library(ROCR)
svm.prob.rocr <- prediction(attr(svm.prob,
"probabilities")[,2], data.test$class)
svm.perf <- performance(svm.prob.rocr, "tpr","fpr")
plot(svm.perf, col=4)
```

B.2 PRECESSING DATASETS

```
#*********************************************************************
# Artificial data set 1: six petals, one in the middle; some degree of
# overlapping
#*********************************************************************
if (!exists("DATA_GEN")) {
DATA_GEN = "none"
}
if (DATA_GEN == "artificial1") {
discNorm <- function(N, C, R) {
#rho <- rnorm(N, sd = sqrt(R))^2
rho <- runif(N, 0, R)
theta <- runif(N, 0, 2) * pi
x <- rho * cos(theta) + C[1]
y <- rho * sin(theta) + C[2]
```

```
l <- data.frame(x,y)
return (l)
}
N <- 500
R <- 1
centers <- matrix(c(5,2.1, 3.2,3.2, 4.8,4.1, 3.5,5.3, 5.4, 5.9, 6.7,4.0)
,nrow = 6,ncol = 2,byrow = T)
cols <- c("red", "blue", "green", "black", "orange", "maroon")
classs <- c(1, 2, 3, 4, 5, 6)
set.seed(123)
df <- data.frame()
for (row in 1:nrow(centers)) {
C <- centers[row,]
g <- discNorm(N, C, R)
col <- rep(cols[row], N)
class <- rep(classs[row], N)
g <- cbind(g, col, class)
df <- rbind(df,g)
}
# df.inicol <- df$col
# plot(df$x, df$y, xlab = 'x1', ylab = 'x2', col = as.vector
(df.inicol))
# legend(x = "topright", NULL,legend = classs,
#        lty = rep(1, length(classs)),
#        lwd = rep(2, length(classs)),
#        col=as.vector(cols))
df$col <- NULL
df$class <- factor(df$class)
}
#*********************************************************************
# Real data set 1: iris
#*********************************************************************
if (DATA_GEN == "iris") {
#library(e1071)
data(iris)
attach(iris)
df <- iris
names(df)[5] <- "class"
```

```
}
#*********************************************************************
# Real data set 2: segmentation
# Source: http://archive.ics.uci.edu/ml/datasets/Image+Segmentation
#*********************************************************************
if (DATA_GEN == "segmentation") {
dsd <- 'data/segment/' #data source directory
dsf <- 'segmentation.data' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= FALSE, sep=",")
df$class <- df[,1]
df[,1] <- NULL
}
#*********************************************************************
# Real data set 3: heart disease (Cleveland data)
# Source: http://archive.ics.uci.edu/ml/datasets/
#*********************************************************************
if (DATA_GEN == "heart") {
dsd <- 'data/heart-disease/' #data source directory
dsf <- 'processed.cleveland.data' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= FALSE, sep=",")
names(df)[length(df)] <- "class"
df$class <- factor(df$class)
}
#*********************************************************************
# Real data set 4: wine
# Source: https://archive.ics.uci.edu/ml/datasets/wine
#*********************************************************************
if (DATA_GEN == "wine") {
dsd <- 'data/wine/' #data source directory
dsf <- 'wine.data' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= FALSE, sep=",")
df$class <- factor(df[,1])
df[,1] <- NULL
}
#*********************************************************************
# Real data set 5: wine quality
# Source: https://archive.ics.uci.edu/ml/datasets/wine+quality
#*********************************************************************
```

```
if (DATA_GEN == "wine-quality") {
dsd <- 'data/wine-quality/' #data source directory
dsf <- 'winequality-red.csv' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= T, sep=";")
names(df)[length(df)] <- "class"
df$class <- factor(df$class)
}
#*********************************************************************
# Real data set 6: glass
# Source:
#*********************************************************************
if (DATA_GEN == "glass") {
dsd <- 'data/glass/' #data source directory
dsf <- 'glass.csv' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= F, sep=",")
names(df)[length(df)] <- "class"
df$class <- factor(df$class)
}
#*********************************************************************
# Real data set 7: covertype
# Source:
#*********************************************************************
if (DATA_GEN == "covertype") {
dsd <- 'data/covertype/' #data source directory
dsf <- 'covertype.csv' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= F, sep=",")
names(df)[length(df)] <- "class"
df$class <- factor(df$class)
}
#*********************************************************************
# Real data set 8: svmguide4
# Source:
#*********************************************************************
if (DATA_GEN == "svmguide4") {
dsd <- 'data/svmguide4/' #data source directory
dsf <- 'svmguide4_train.csv' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= F, sep=",")
df$class <- factor(df[,1])
```

```
df[,1] <- NULL
#remove constant columns: they are irrelevant
df <- df[,apply(df, 2, var, na.rm=TRUE) != 0]
}
#*********************************************************************
# Real data set 9: vowel
# Source:
#*********************************************************************
if (DATA_GEN == "vowel") {
dsd <- 'data/vowel/' #data source directory
dsf <- 'vowel_train.csv' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= F, sep=",")
df$class <- factor(df[,1])
df[,1] <- NULL
}
#*********************************************************************
# Real data set 10: usps
# Source: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
multiclass.html
#*********************************************************************
if (DATA_GEN == "usps") {
dsd <- 'data/usps/' #data source directory
dsf <- 'usps_test.csv' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= F, sep=",")
df$class <- factor(df[,1])
df[,1] <- NULL
}
#*********************************************************************
# Real data set 11: letter recognition
# Source: http://archive.ics.uci.edu/ml/datasets/
#*********************************************************************
if (DATA_GEN == "letter") {
dsd <- 'data/letter-recognition/' #data source directory
dsf <- 'letter-recognition.data' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= FALSE,
sep=",")
df$class <- df[,1]
df[,1] <- NULL
```

```
}
#*********************************************************************
# Real data set 12: poker
# Source: http://archive.ics.uci.edu/ml/datasets/
#*********************************************************************
if (DATA_GEN == "poker") {
dsd <- 'data/poker/' #data source directory
dsf <- 'poker.csv' #data source file
df <- read.csv(paste(dsd, dsf, sep = ''), header= FALSE, sep= ",")
class <- factor(df[,1])
df <- df[,-1]
names(df) <- c("S1", "C1", "S2", "C2", "S3", "C3", "S4", "C4", "S5",
"C5")
df$class <- class
rm(class)
}
#*********************************************************************
# Real data set 13: sensorless
# Source: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/
datasets/multiclass.html
#*********************************************************************
if (DATA_GEN == "sensorless") {
dsd <- 'data/sensorless/' #data source directory
dsf <- 'sensorless.csv' #data source file
df <- na.omit(read.csv(paste(dsd, dsf, sep = ''),header=FALSE, sep=","))
df$class <- factor(df[,1])
df[,1] <- NULL
}
#*********************************************************************
#randomly divide into p% for training, (1-p)% for testing
if (DATA_GEN != "none" && DATA_GEN != "sector") {
set.seed(2017)
p <- .8
train <- sample(nrow(df), p*nrow(df))
tst <- sample(nrow(df[-train,]), nrow(df[-train,]))
#tst <- -train
#store each fragment of data
df.train <- df[train,]
```

```
df.test <- df[tst,]
}
```

B.3 BUILDING MULTICLASS SVMS

```
# Performs one vs one voting on a given candidate.
# Input:
# svms: list of one-vs-one svm models as a list of
#       an upper triangular matrix;
#       the index of i vs j can be computed as:
#       idx <- ((i-1) * cls.n - i * (i-1)/2) + j-i
# classes: the list of classes
# it must be the case that: length(svms) = length(classes) *
# (length(classes) - 1) / 2
# candidate: the candidate to be classified;
# must be a dataframe of appropriate size and content returns voting
#results.
onevsone <- function(svms, classes, candidates) {
nclass <- length(classes)
votes <- matrix(0, nrow = nrow(candidates), ncol = length(classes))
colnames(votes) <- classes
idx <- 1
for (i in 1:(nclass-1)) {
for (j in (i+1):nclass) {
p <- predict(svms[[idx]], candidates)
if(is.list(p)) p <- p$class
p <- as.factor(p)
for (k in 1:length(p)) {
votes[k,as.character(p[k])] <- as.numeric(votes[k,as.character(p[k])])
+ 1
}
idx <- idx + 1
   }
}
ans <- c()
for (i in 1:nrow(votes)) {
max <- which.max(votes[i,])
```

```
ans[i] <- classes[max]
}
return (ans)
}
#*********************************************************************
# Let all svms vote and collect their votes according to
# expected predictions.
#*********************************************************************
maxpredict <- function(svms, classes, predictp, predictc, candidates,
minconf = 90.0) {
nclass <- length(classes)
votes <- matrix(0, nrow = nrow(candidates), ncol = length(svms))
idx <- 1
for (i in 1:(nclass-1)) {
for (j in (i+1):nclass) {
p <- predict(svms[[idx]], candidates)
if(is.list(p)) p <- p$class
p <- as.factor(p)
for (k in 1:length(p)) {
votes[k,idx] <- as.character(p[k])
}
idx <- idx + 1
    }
}
allvotes <- matrix(0, nrow = nrow(candidates), ncol = length(classes))
colnames(allvotes) <- classes
for (i in 1:nrow(candidates)) {
for (j in 1:length(classes)) {
pred <- predictp[,j+1] >= minconf
allvotes[i,j] <- as.numeric((predictc[,j+1] == votes[i,])
%*% (predictp[,j+1] * pred)) / sum(pred)
   }
}
ans <- c()
for (i in 1:nrow(allvotes)) {
max <- which.max(allvotes[i,])
ans[i] <- classes[max]
}
```

```
return (ans)
}
#*********************************************************************
# SVMDC related functions
# cls.n = number of classes
# cls.levels = list of classes
# cls.svmp = svm deciders for the plan (for creating the
#decision tree)
# cls.svmc = svm deciders for actual classification
#*********************************************************************
initSVMDC <- function(df.train, cls.n, cls.levels,cls.svmp, cls.svmc,
tshold = 5) {predictvalues <- list()
allpredict <- data.frame(matrix(ncol = length(cls.levels)+1, nrow = 0))
maxpredictp <- data.frame(matrix(ncol = length(cls.levels)+1, nrow = 0))
maxpredictc <- data.frame(matrix(ncol = length(cls.levels)+1, nrow = 0))
colnames(allpredict) <- c('svm',cls.levels)
colnames(maxpredictp) <- c('svm',cls.levels)
colnames(maxpredictc) <- c('svm',cls.levels)
for (i in 1:(cls.n-1)) {
for (j in (i+1):cls.n) {
idx <- ((i-1) * cls.n - i * (i-1)/2) + j-i
svmp <- cls.svmp[[idx]]
svmc <- cls.svmc[[idx]]
ci <- cls.levels[i]
cj <- cls.levels[j]
r <- nrow(allpredict) + 1
allpredict[r,] <- NA
maxpredictp[r,] <- NA
maxpredictc[r,] <- NA
svmname <- paste(ci," vs. ", cj)
allpredict[r,1] <- svmname
maxpredictp[r,1] <- svmname
maxpredictc[r,1] <- svmname
predictvalues[[svmname]] <- list(c('svm',ci,cj))
# predictvalues[[svmname]][['svm']] <- svmc
predictvalues[[svmname]][[ci]] <- c()
predictvalues[[svmname]][[cj]] <- c()
for (k in 1:length(cls.levels)) {
```

```
cls <- cls.levels[k]
d <- df.train[df.train$class == cls,]
p <- table(predict(svmp, d))
s <- sum(p)/100
pi <- round(p[[1]]/s, 2)
pj <- round(p[[2]]/s,2)
l <- length(predictvalues[[svmname]][[ci]])
predictvalues[[svmname]][[ci]][l + 1] <- pi
l <- length(predictvalues[[svmname]][[cj]])
predictvalues[[svmname]][[cj]][l + 1] <- pj
allpredict[r,k+1] <- paste(ci,"=", pi, "%; ", cj, "=", pj, "%",sep = "")
maxpredictp[r,k+1] <- max(pi, pj)
maxpredictc[r,k+1] <- ifelse(pi >= pj, ci, cj)
}
predictvalues[[svmname]]['score'] <- ((predictvalues
[[svmname]][[ci]][i]) + (predictvalues[[svmname]][[cj]][j])) / 2
   }
}
foo <- function(x) {
ci <- names(x)[2]
cj <- names(x)[3]
i <- which(cls.levels == ci)
j <- which(cls.levels == cj)
if (tshold < 0 ) {
ti <- x[[ci]][j]
tj <- x[[cj]][i]
x[[ci]] <- x[[ci]] >= ti
x[[cj]] <- x[[cj]] >= tj
} else {
x[[ci]] <- x[[ci]] > tshold
x[[cj]] <- x[[cj]] > tshold
}
x[[ci]][i] <- T
x[[cj]][j] <- T
x[[ci]][j] <- F
x[[cj]][i] <- F
return (x)
}
```

```
predictnodes <- lapply(predictvalues, foo)
ans <- list()
ans$pnodes <- predictnodes
ans$allpredict <- allpredict
ans$maxpredictp <- maxpredictp
ans$maxpredictc <- maxpredictc
return (ans)
}
#*********************************************************************
# creates an igraph plan
createSVMDCplan <- function(cls.n, cls.levels, pnodes, astree = F,
buildigraph = T) {
SVMDCplan <<- list()
SVMDCplan$N <<- 1
SVMDCplan$V <<- c()  #labels
SVMDCplan$E <<- c()  #edges
SVMDCplan$EL <<- c()  #edges labels
SVMDCplan$svm <<- c()
if (astree) {#tree shaped plan
createSVMDCtreeplan(NULL, rep(T, cls.n), pnodes, cls.n, cls.levels)
} else {#graph shaped plan
createSVMDCgraphplan(NULL, rep(T, cls.n), pnodes, cls.n, cls.levels)
}
SVMDCplan$N <<- SVMDCplan$N - 1
if (buildigraph) {
G <- graph(SVMDCplan$E, directed=TRUE)
G <- set_vertex_attr(G, "label", V(G), SVMDCplan$V)
#  G <- set_vertex_attr(G, "svm", V(G), SVMDCplan$svm)
} else {
G <- list()
}
dectree <- createSVMCDdectree2(SVMDCplan$E, SVMDCplan$V, cls.levels)
G$dectree <- dectree
return (G)
}
#creates plan as a graph; this function produce the list of edges (name
= label)
createSVMDCgraphplan <- function (p, m, pnodes, cls.n, cls.levels) {
```

```
min <- cls.n + 1
maxscore <- 0
dec <- NULL
idx <- 0
for (i in 1:length(pnodes)) {
x <- pnodes[[i]]
ci <- names(x)[3]
cj <- names(x)[4]
ii <- which(cls.levels == ci)
ij <- which(cls.levels == cj)
if (!m[ii] || !m[ij]) {
next
}
left <- x[[3]] & m
if (sum(left) == 0) {
next
}
right <- x[[4]] & m
if (sum(right) == 0) {
next
}
u <- sum(left & right)#seeks minimum intersection (faster)
sc <- unlist(x['score'])#seeks max score (precision)
if (sc > maxscore) {#first criterion: precision
maxscore <- sc
dec <- x
idx <- i
} else if (sc == maxscore) {
if (u < min) {#second: faster
min <- u
dec <- x
idx <- i
   }
}
#another possible approach: faster then precise if (u < min) {#faster
#   min <- u
#   dec <- x
#   idx <- i
```

```
# } else if (u == min) {
#   if (sc > maxscore) {#precision
#     maxscore <- sc
#     dec <- x
#     idx <- i
#   }
# }
}
if (idx > 0) {
#prevent using same decider again
#pnodes[[idx]][[3]] <- rep(F,length(pnodes[[idx]][[3]]))
#pnodes[[idx]][[4]] <- rep(F,length(pnodes[[idx]][[4]]))
pnodes[[idx]] <- NULL
svm <- dec[2]
#create a new vertex with label "l" and id=vertex.id
cvid <- paste(names(dec)[3],names(dec)[4],sep = '~')
nobranching <- F
if (!(cvid %in% SVMDCplan$V)) {
SVMDCplan$V[length(SVMDCplan$V) + 1] <<- cvid
SVMDCplan$svm[length(SVMDCplan$svm) + 1] <<- svm
SVMDCplan$N <<- SVMDCplan$N + 1
} else {
#print("this was already created: link to it")
nobranching <- T
}
if (!is.null(p)) {#not root node
G <- G + edge(p,vertex_attr(G,"name", length(V(G))))
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- p
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- cvid
}
if (nobranching) {
return
}
if (sum(dec[[3]] & m) <= 1) {#branch to the left
ll <- names(dec)[3]
if (!(ll %in% SVMDCplan$V)) {
SVMDCplan$V[length(SVMDCplan$V) + 1] <<- ll
SVMDCplan$svm[length(SVMDCplan$svm) + 1] <<- NA
```

```
SVMDCplan$N <<- SVMDCplan$N + 1
}
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- cvid
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- ll
} else {
pnodesL <- pnodes
createSVMDCgraphplan(cvid, dec[[3]] & m, pnodesL, cls.n, cls.levels)
}
if (sum(dec[[4]] & m) <= 1) {#branch to the right
lr <- names(dec)[4]
if (!(lr %in% SVMDCplan$V)) {
SVMDCplan$V[length(SVMDCplan$V) + 1] <<- lr
SVMDCplan$svm[length(SVMDCplan$svm) + 1] <<- NA
SVMDCplan$N <<- SVMDCplan$N + 1
}
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- cvid
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- lr
} else {
pnodesR <- pnodes
createSVMDCgraphplan(cvid, dec[[4]] & m, pnodesR, cls.n, cls.levels)
      }
   }
}
#creates plan as a tree; this function produce the list of edges and
#vertices labels
createSVMDCtreeplan <- function (p,m,pnodes, cls.n, cls.levels) {
min <- cls.n + 1
maxscore <- 0
maxb <- 0
dec <- NULL
idx <- 0
for (i in 1:length(pnodes)) {
x <- pnodes[[i]]
ci <- names(x)[2]
cj <- names(x)[3]
ii <- which(cls.levels == ci)
ij <- which(cls.levels == cj)
if (!m[ii] || !m[ij]) {
```

```
next
}
left <- x[[2]] & m
if (sum(left) == 0) {
next
}
right <- x[[3]] & m
if (sum(right) == 0) {
next
}
u <- sum(left & right)#seeks minimum intersection (faster)
sc <- unlist(x['score'])#seeks max score (precision)
b <- min(sum(left), sum(right))
if (i == 1) {
maxscore <- sc
min <- u
minb <- b
dec <- x
idx <- i
next
}
if (sc > maxscore) {#first criterion: precision
maxscore <- sc
dec <- x
idx <- i
} else if (sc == maxscore) {
if (u < min) {#second: faster
min <- u
dec <- x
idx <- i
} else if (u == min) {#thid: balance
if (b > minb) {
minb <- b
dec <- x
idx <- i
    }
  }
}
```

```
#another possible approach: faster then precise if (u < min) {#faster
#   min <- u
#   dec <- x
#   idx <- i
# } else if (u == min) {
#   if (sc > maxscore) {#precision
#     maxscore <- sc
#     dec <- x
#     idx <- i
#   }
# }
}
if (idx > 0) {
#prevent using same decider again
#pnodes[[idx]][[3]] <- rep(F,length(pnodes[[idx]][[3]]))
#pnodes[[idx]][[4]] <- rep(F,length(pnodes[[idx]][[4]]))
pnodes[[idx]] <- NULL
#svm <- dec[2]
#create a new vertex with label "l" and id=vertex.id
l <- paste(names(dec)[2],names(dec)[3],sep = '~')
cvid <- SVMDCplan$N
SVMDCplan$V[length(SVMDCplan$V) + 1] <<- l
#SVMDCplan$svm[length(SVMDCplan$svm) + 1] <<- svm
SVMDCplan$N <<- SVMDCplan$N + 1
if (!is.null(p)) {#root node
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- p
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- cvid
}
if (sum(dec[[2]] & m) <= 1) {#branch to the left
ll <- names(dec)[2]
SVMDCplan$V[length(SVMDCplan$V) + 1] <<- ll
#SVMDCplan$svm[length(SVMDCplan$svm) + 1] <<- NA
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- cvid
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- SVMDCplan$N
SVMDCplan$N <<- SVMDCplan$N + 1
} else {
pnodesL <- pnodes
createSVMDCtreeplan(cvid, dec[[2]] & m, pnodesL, cls.n, cls.levels)
```

```
}
if (sum(dec[[3]] & m) <= 1) {#branch to the right
lr <- names(dec)[3]
SVMDCplan$V[length(SVMDCplan$V) + 1] <<- lr
# SVMDCplan$svm[length(SVMDCplan$svm) + 1] <<- NA
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- cvid
SVMDCplan$E[length(SVMDCplan$E) + 1] <<- SVMDCplan$N
SVMDCplan$N <<- SVMDCplan$N + 1
} else {
pnodesR <- pnodes
createSVMDCtreeplan(cvid, dec[[3]] & m, pnodesR, cls.n, cls.levels)
       }
   }
}
#************************SVMCD method*********************************
# Creates a binary decision tree (as an array) out of the decision plan
# graph.
#**********************************************************************
createSVMCDdectree <- function (G, svms, classes) {
dectree <- rep(list(list()), length(V(G)))
treenodes <- c()
#alledges <- get.edgelist(G)
treeedges <- E(G)
#traverse the tree from root to a leaf
node <- V(G)[1]
ni <- 1
treenodes[length(treenodes)+1] <- node
while (ni <= length(treenodes)) {
node <- treenodes[ni]#retrieve current
ni <- ni + 1#move to next node in the list
dec <- strsplit(toString(vertex_attr(G, "label", node)),"~")[[1]]
#svm <- vertex_attr(G, "svm", node)[[1]]
i <- which(classes == dec[1])
j <- which(classes == dec[2])
idx <- ((i-1) * cls.n - i * (i-1)/2) + j-i
dectree[[node]]$idx <- idx
dectree[[node]]$dec <- c(dec[1], dec[2])
dectree[[node]]$cld <- c(NA,NA)
```

```
#children <- alledges[alledges[,1]==toString(node),2]
children <- treeedges[.from(node)]
if (length(children) > 0) {
children <- ends(G, children)[,2]
#compute children indexes
dec <- strsplit(toString(vertex_attr(G,"label", children[1])), "~")[[1]]
if (length(dec) > 1) {
treenodes[length(treenodes)+1] <- children[1]
#i <- which(classes == dec[1])
#j <- which(classes == dec[2])
#idx1 <- ((i-1) * cls.n - i * (i-1)/2) + j-i
#dectree[[node]]$cld[1] <- idx1
dectree[[node]]$cld[1] <- children[1]
}
dec <- strsplit(toString(vertex_attr(G, "label",children[2])), "~")[[1]]
if (length(dec) > 1) {
treenodes[length(treenodes)+1] <- children[2]
#svm <- vertex_attr(G, "svm", node)[[1]]
#i <- which(classes == dec[1])
#j <- which(classes == dec[2])
#idx2 <- ((i-1) * cls.n - i * (i-1)/2) + j-i
#dectree[[node]]$cld[2] <- idx2
dectree[[node]]$cld[2] <- children[2]
      }
   }
}
return (dectree)}
createSVMCDdectree2 <- function (E, V, classes) {
dectree <- rep(list(list()), length(V))
ni <- 1
while (ni <= length(E)) {
node <- E[ni]#retrieve current
node2 <- E[ni+1]
if (length(strsplit(V[node2], "~")[[1]]) < 2) {
node2 <- NA
}
ni <- ni + 2#move to next node in the list if (is.null(dectree[[node]]
$idx)) {
```

```
dec <- strsplit(V[node], "~")[[1]]
#svm <- vertex_attr(G, "svm", node)[[1]]
i <- which(classes == dec[1])
j <- which(classes == dec[2])
idx <- ((i-1) * cls.n - i * (i-1)/2) + j-i
dectree[[node]]$idx <- idx
dectree[[node]]$dec <- c(dec[1], dec[2])
dectree[[node]]$cld <- c(node2,NA)
} else {
dectree[[node]]$cld[2] <- node2
   }
}
return (dectree)
}
#**********************SVMDC method ***********************************
# data stored in svmdc.plan structure (an igraph tree) each node stores
# the decision svm in the "svm" attribute the node "label" attribute =
#'class1 - class2'; if a leaf, "label" = class
#*********************************************************************
svmdc.predict <- function (G, svms, classes, candidate) {
alledges <- get.edgelist(G)
#traverse the tree from root to a leaf
node <- V(G)[1]
istree <- sum(alledges[,1]==toString(node)) > 0
totnSV <- 0
totsteps <- 0
#children <- alledges[alledges[,1]==toString(vertex_attr(G,"label", node)
#),2] #children
children <- alledges[alledges[,1]==ifelse(istree, toString(node),
node$label),2] #children
while (length(children) > 0) {
dec <- strsplit(toString(vertex_attr(G, "label", node)),"~")[[1]]
#svm <- vertex_attr(G, "svm", node)[[1]]
i <- which(classes == dec[1])
j <- which(classes == dec[2])
idx <- ((i-1) * cls.n - i * (i-1)/2) + j-i
svm <- svms[[idx]]
p <- predict(svm, candidate)
```

```
totsteps <- totsteps + 1
totnSV <- totnSV + svm$tot.nSV
if(is.list(p)) p <- p$class
p <- as.factor(p)
#left <- strsplit(toString(vertex_attr(G, "label", node)), "~")[[1]][1]
if (p == dec[1]) {
node <- ifelse(istree, children[1], vertex_attr(G, "label",children[1]))
} else {
node <- ifelse(istree, children[2], vertex_attr(G, "label",children[2]))
}
#children <- alledges[alledges[,1]==toString(vertex_attr(G,"label",
node)),2]
children <- alledges[alledges[,1]==ifelse(istree,
toString(node), toString(vertex_attr(G, "label", node))),2]
}
pred <- list()
pred$vote <- toString(vertex_attr(G, "label", node))
pred$totnSV <- totnSV
pred$steps <- totsteps
return (pred)
}
#this is faster, based on binary decision tree
svmdc.predict2 <- function (dectree, svms, candidate) {
tidx <- 1
#traverse the tree from root to a leaf
totnSV <- 0
totsteps <- 0
while (!is.na(tidx)) {
dec <- dectree[[tidx]]$dec
svm <- svms[[dectree[[tidx]]$idx]]
p <- predict(svm, candidate)
totsteps <- totsteps + 1
totnSV <- totnSV + svm$tot.nSV
if(is.list(p)) p <- p$class
p <- as.factor(p)
#left <- strsplit(toString(vertex_attr(G, "label", node)), "~")[[1]][1]
if (p == dec[1]) {
tidx <- dectree[[tidx]]$cld[1]
```

```
} else {
tidx <- dectree[[tidx]]$cld[2]
   }
}
pred <- list()
pred$vote <- as.character(p)#toString(vertex_attr(G, "label", node))
pred$totnSV <- totnSV
pred$steps <- totsteps
return (pred)
}
#*********************************************************************
# Creates a radial SVM model with tunning.
#*********************************************************************
createSVMradial <- function(data, gammar, costr, eps, wts = NULL) {
tuned <- tune.svm(class~., data=data, kernel='radial', gamma=gammar,
cost=costr, class.weights = wts)
g <- tuned$best.parameters$gamma
c <- tuned$best.parameters$cost
e <- eps
s <- svm(class~., data=data, kernel='radial', gamma=g, cost= , epsilon=
e, cross=10, class.weights = wts)
return (s)
}
saveLetterSVMparam <- function() {
svms <- cls.svmr
param <- matrix(0,nrow = length(svms), ncol = 3)
eps <- 0.1
for (i in 1:length(svms)) {
s <- svms[[i]]
g <- s$gamma
c <- s$cost
e <- eps
param[i,] <- c(g,c,e)
}
write.table(param, file="data/letter-recognition/svmtune.txt", row.names
=FALSE, col.names=FALSE)
}
createLetterSVMs <- function() {
```

```
param <- read.table(file="data/letter-recognition/svmtune.txt")
svmrs <- list()
idx <- 1
for (i in 1:(cls.n-1)) {
for (j in (i+1):cls.n) {
g <- param[idx,1]
c <- param[idx,2]
e <- param[idx,3]
c1 <- cls.levels[i]
c2 <- cls.levels[j]
#select data only for these classes
cls.both <- df.train[df.train$class == c1 | df.train$class == c2,]
cls.both$class <- factor(cls.both$class)
svmrs[[idx]] <- svm(class~., data=cls.both, kernel='radial', gamma=g,
cost=c, epsilon = e, cross=10)
idx <- idx + 1
   }
}
return (svmrs)
}
```

B.4 Experimental Results: Section 5.4.1 and 5.4.2

```
# Experimental results: comparison of built in 1vs1, computed 1vs1, and
# DCSVM for ALL data sets using rbf kernel Each comparison is run 10
# times (for each set, cross-validation approach) and average accurracies
# are computed
# Results:
# - accurracies for each trial and mean accurracy
# - no. of SVM comparisons, on average, for all data samples
# - no. of support vectors used for dot products, on average
#set memory limit
memory.limit(6410241024*1024)
library(e1071)
library(igraph)
#load functions
source("utils.R")
```

```
THRESHOLD <- 2 #for dcsvm: error percentage for separation generate data:
#run with one option for a specific data set
#DATA_GEN <- "artificial1"
#DATA_GEN <- "iris"
#DATA_GEN <- "segmentation"
#DATA_GEN <- "heart"
#DATA_GEN <- "wine"
#DATA_GEN <- "wine-quality"
#DATA_GEN <- "glass"
#DATA_GEN <- "covertype"
#DATA_GEN <- "svmguide4"
#DATA_GEN <- "vowel"
#DATA_GEN <- "usps"
#DATA_GEN <- "letter"
DATA_GEN <- "sector"
#DATA_GEN <- "poker"
#DATA_GEN <- "sensorless"
alldatasets <- c(#"artificial1","iris","segmentation","heart","wine",
#"wine-quality", "glass","covertype", "svmguide4","vowel", "usps",
#"letter", "poker", "sensorless")
#store results summaries in a data frame
allresults <- data.frame(No=NA,
Dataset=NA,
N.Classes=NA,
BIacc=NA,
OVOacc=NA,
OVOdp=NA,
OVOsteps=NA,
OVOavt=NA,
DCSVMacc=NA,
DCSVMdp=NA,
DCSVMsteps=NA,
DCSVMavt=NA)
dsno <- 0
for (DATA_GEN in alldatasets) {
dsno <- dsno + 1
if (dsno > 1) {
allresults <- rbind(allresults, rep(NA,ncol(allresults)))
```

```
}
allresults[dsno,1] <- dsno
allresults[dsno,2] <- DATA_GEN
print(paste0('Loading data: ', DATA_GEN))
print(Sys.time())
source("datagen.R")
print('Data was loaded!')
print(Sys.time())
## perform classification 10 times and display information
seeds <- c(1234, 2411, 2038, 9391, 2090, 5781, 4526, 191919, 78213,
46712, 6072)
p1vs1in.true <- c()
p1vs1in.false <- c()
p1vs1co.true <- c()
p1vs1co.false <- c()
p1vs1co.time <- c()
dcsvm.true <- c()
dcsvm.false <- c()
dcsvm.time <- c()
totnSV <- 0
dcsvm.totnSV <- c()
dcsvm.totsteps <- c()
tot.samples <- 0
trials <- 1:1
if (DATA_GEN == "letter") {
trials <- 1:4
}
for (trial in trials) {
#create a list of all classes pairs in the training data set store in a
#list
# if (trial > 1) {
#   rm(cls.all)  #clear some memory
#   gc()
# }
# cls.all <- list()
cls.levels <- levels(factor(df.train$class))
cls.n <- length(cls.levels)
# idx <- 1
```

```
# for (i in 1:(cls.n-1)) {
#   for (j in (i+1):cls.n) {
#     c1 <- cls.levels[i]
#     c2 <- cls.levels[j]
#     cls.all[[idx]] <- df.train[df.train$class == c1 | df.train$class
#== c2,]
#     cls.all[[idx]]$class <- factor(cls.all[[idx]]$class)
#     idx <- idx + 1
#   }
# }
#barplot(summary(df.train$class))
#create linear svm models for all pairs of classes
#svmkernel <- 'linear'
svmkernel <- 'radial'
set.seed(1234)
gammar <- 10^(-6:-1)
costr <- 10^(-1:2)
eps <- 0.1
if (trial > 1) {
rm(cls.svmr)  #clear some memory
gc()
}
cls.svmr <- list()
#  if (DATA_GEN == "letter") {
#    #can run this for "letter" if parameters were saved
#    cls.svmr <- createLetterSVMs()
#  } else {
idx <- 1
for (i in 1:(cls.n-1)) {
for (j in (i+1):cls.n) {
c1 <- cls.levels[i]
c2 <- cls.levels[j]
#select data only for these classes
cls.both <- df.train[df.train$class == c1 | df.train$class == c2,]
cls.both$class <- factor(cls.both$class)
wts <- NULL#100 / table(cls.both$class)
#cls.svmr[[idx]] <- createSVMradial(cls.both, gammar, costr, eps, wts)
#default parameters
```

```
cls.svmr[[idx]] <- svm(class~., cls.both)
rm(cls.both)
totnSV <- totnSV + cls.svmr[[idx]]$tot.nSV
idx <- idx + 1
   }
}
#  }
#create built in 1vs1 model
cls.1vs1svm <-  svm(factor(df.train$class)~., df.train)
print('All SVMs created:')
print(Sys.time())
#*********************************************************************
#  built in 1vs1 prediction
#*********************************************************************
sample <-df.test
p1vs1 <- predict(cls.1vs1svm, sample)
#print(paste('built in 1vs1 prediction results for: "', DATA_GEN,'"',sep
= ""))
ans <- sum(sample$class == p1vs1)
p1vs1in.true[trial] <- ans
p1vs1in.false[trial] <- nrow(sample) - ans
#print(ans)
#print(sprintf("Accuracy: %2.2f%%", (ans[2]/(ans[1]+ans[2]))*100))
print(Sys.time())
#*********************************************************************
#  1vs1 prediction wifh computing votes
#*********************************************************************
#sample <-df.test
#p1vs1c <- onevsone(cls.svmr, cls.levels, sample)
p1vs1co.time[trial] <- Sys.time()
p1vs1c <- 0
for (i in 1:nrow(sample)){
# Pick a single observation for the one-vs-one classifiers to vote on
candidate = sample[i,]
pred <- onevsone(cls.svmr, cls.levels, candidate)
p1vs1c[i] <- pred
}
p1vs1co.time[trial] <- Sys.time() - p1vs1co.time[trial]
```

```
#print(paste('1vs1 prediction results for: ",DATA_GEN,'"',sep = ""))
ans <- sum(sample$class == p1vs1c)
p1vs1co.true[trial] <- ans
p1vs1co.false[trial] <- nrow(sample) - ans
#print(ans)
#print(sprintf("Accuracy: %2.2f%%", (ans[2]/(ans[1] + ans[2]))*100))
print(Sys.time())
#*********************************************************************
# DCSVM prediction
#*********************************************************************
#initialize
info <- initSVMDC(df.train, cls.n, cls.levels, cls.svmr, cls.svmr, tshold
= THRESHOLD)
print(Sys.time())
allpredict <- info$allpredict
svmdc.plan <- createSVMDCplan(cls.n, cls.levels, info$pnodes,
astree = T, buildigraph = F)
#print(Sys.time())
#svmdc.dectree <- createSVMCDdectree(svmdc.plan, cls.svmr, cls.levels)
print('DCSVM initialization completed:')
print(Sys.time())
#predict and print
dcsvm.totnSV[trial] <- 0
dcsvm.totsteps[trial] <- 0
psvmdc <- 0
dcsvm.time[trial] <- Sys.time()
for (i in 1:nrow(sample)){
# Pick a single observation for the one-vs-one classifiers
# to vote on
candidate = sample[i,]
#pred <- svmdc.predict(svmdc.plan, cls.svmr, cls.levels, candidate)
#pred <- svmdc.predict2(svmdc.dectree, cls.svmr, candidate)
pred <- svmdc.predict2(svmdc.plan$dectree, cls.svmr, candidate)
psvmdc[i] <- pred$vote
dcsvm.totnSV[trial] <- dcsvm.totnSV[trial] + pred$totnSV
dcsvm.totsteps[trial] <- dcsvm.totsteps[trial] + pred$steps
}
dcsvm.time[trial] <- Sys.time() - dcsvm.time[trial]
```

```
dcsvm.totnSV[trial] <- dcsvm.totnSV[trial] / nrow(sample)
dcsvm.totsteps[trial] <- dcsvm.totsteps[trial] / nrow(sample)
#print(paste('SVMDC prediction results for: "',DATA_GEN,'"', sep = ""))
ans <- sum(sample$class == psvmdc)
dcsvm.true[trial] <- ans
dcsvm.false[trial] <- nrow(sample) - ans
#print(ans)
#print(sprintf("Accuracy: %2.2f%%", (ans[2]/(ans[1]+ans[2]))*100))
#print(Sys.time())
#*********************************************************************
#compute errors per each class and how classes were mis-classified (the
#confusion table)
#res <- data.frame(orig = sample$class, p1vs1 = p1vs1, p1vs1c = p1vs1c,
#psvmdc = psvmdc)
#print('Confusion Table 1vs1 builtin')
#print(table(res$orig, res$p1vs1, dnn = c("Original","built-in 1vs1")))
#print('Confusion Table 1vs1 computed')
#print(table(res$orig, res$p1vs1c, dnn = c("Original","computed 1vs1")))
#print('Confusion Table DCSVM')
#print(table(res$orig, res$psvmdc, dnn = c("Original","DCSVM")))
tot.samples <- tot.samples + nrow(sample)
#split again training/test
set.seed(seeds[trial])
p <- .9
train <- sample(nrow(df), p*nrow(df))
#store each fragment of data
df.train <- df[train,]
df.test <- df[-train,]
}
print("")
#*****************print the results***********************************
print(sprintf("Number of classes: %d", cls.n))
allresults[dsno,3] <- cls.n
print(paste('Built in 1vs1 prediction results for: "', DATA_GEN,'"',sep
= ""))
s <- sprintf("%2.2f%%  ", (p1vs1in.true/(p1vs1in.true + p1vs1in.false))
*100)
print(sprintf("Trials accuracy: %s", paste(s, collapse =  "")))
```

```
print(sprintf("Mean Accuracy: %2.2f%%", (sum(p1vs1in.true)
/sum(p1vs1in.true + p1vs1in.false))*100))
allresults[dsno,4] <- (sum(p1vs1in.true)/sum(p1vs1in.true +
p1vs1in.false))*100
print("")
print(paste('Computed 1vs1 prediction results for: "',DATA_GEN,'"',sep
= ""))
s <- sprintf("%2.2f%%  ", (p1vs1co.true/(p1vs1co.true + p1vs1co.false))
*100)
print(sprintf("Trials accuracy: %s", paste(s, collapse = "")))
print(sprintf("Mean Accuracy: %2.2f%%",
(sum(p1vs1co.true) /sum(p1vs1co.true+p1vs1co.false))*100))
print(sprintf("Dot products (average SVs):%.2f", totnSV/length(trials)))
print(sprintf("Average steps: %d", (cls.n * (cls.n - 1))/ 2))
print(sprintf("Average time: %.4f", (sum(p1vs1co.time)/length(trials))))
allresults[dsno,5] <- (sum(p1vs1co.true)/sum(p1vs1co.true +
p1vs1co.false))*100
allresults[dsno,6] <- totnSV/length(trials)
allresults[dsno,7] <- (cls.n * (cls.n - 1)) / 2
allresults[dsno,8] <- sum(p1vs1co.time)/length(trials)
print("")
print(paste('DCSVM prediction results for: "',DATA_GEN,'" with threshold
= ', THRESHOLD,sep = ""))
s <- sprintf("%2.2f%%  ", (dcsvm.true/(dcsvm.true + dcsvm.false))*100)
print(sprintf("Trials accuracy: %s", paste(s, collapse = "")))
print(sprintf("Mean Accuracy: %2.2f%%", (sum(dcsvm.true)/sum(dcsvm.true+
dcsvm.false))*100))
s <- sprintf("%2.2f  ", dcsvm.totnSV)
print(sprintf("Dot products (average SV used):%s",paste(s,collapse="")))
print(sprintf("Mean SVs used:%2.2f",(sum(dcsvm.totnSV)/length(trials))))
s <- sprintf("%2.2f  ", dcsvm.totsteps)
print(sprintf("Average steps for all trials:%s",paste(s,collapse = "")))
print(sprintf("Average steps: %2.2f", (sum(dcsvm.totsteps)/
length(trials))))
print(sprintf("Average time: %.4f", (sum(dcsvm.time)/length(trials))))
print(Sys.time())
allresults[dsno,9] <- (sum(dcsvm.true)/sum(dcsvm.true+dcsvm.false))*100
allresults[dsno,10] <- (sum(dcsvm.totnSV)/length(trials))
```

```
allresults[dsno,11] <- (sum(dcsvm.totsteps)/length(trials))
allresults[dsno,12] <- (sum(dcsvm.time)/length(trials))
}
#save to file
now <- Sys.time()
allresultsfilename <- paste0("output/allresults-tshold", THRESHOLD, '_'
,format(now, "%Y%m%d_%H%M%S"), ".csv")
#write.csv(allresults, allresultsfilename, row.names = F)
```

B.5 Collecting Split Percentages

```
library(e1071)
library(igraph)
#load functions
source("utils.R")
THRESHOLD <- 2 #for dcsvm: error percentage for separation generate data:
#run with one option for a specific data set
#DATA_GEN <- "artificial1"
#DATA_GEN <- "iris"
#DATA_GEN <- "segmentation"
#DATA_GEN <- "heart"
#DATA_GEN <- "wine"
#DATA_GEN <- "wine-quality"
#DATA_GEN <- "glass"
#DATA_GEN <- "covertype"
#DATA_GEN <- "svmguide4"
#DATA_GEN <- "vowel"
#DATA_GEN <- "usps"
#DATA_GEN <- "letter"
DATA_GEN <- "sector"
#DATA_GEN <- "poker"
#DATA_GEN <- "sensorless"
alldatasets <- c("artificial1","iris","segmentation","heart","wine",
"wine-quality", "glass","covertype","svmguide4", "vowel", "usps",
"letter","poker","sensorless")
THRESHOLD <- 95
```

```
#store percentages in a list
allpercentages <- list()
for (DATA_GEN in alldatasets) {
print(paste0('Loading data: ', DATA_GEN))
print(Sys.time())
source("datagen.R")
print('Data was loaded!')
print(Sys.time())
## perform classification 10 times and display information
seeds <- c(1234, 2411, 2038, 9391, 2090, 5781, 4526, 191919,78213, 46712,
6072)
#create a list of all classes pairs in the training data set store in a
#list if (trial > 1) {
#   rm(cls.all)  #clear some memory
#   gc()
# }
# cls.all <- list()
cls.levels <- levels(factor(df.train$class))
cls.n <- length(cls.levels)
# idx <- 1
# for (i in 1:(cls.n-1)) {
#   for (j in (i+1):cls.n) {
#     c1 <- cls.levels[i]
#     c2 <- cls.levels[j]
#     cls.all[[idx]] <- df.train[df.train$class==c1|df.train$class==c2,]
#     cls.all[[idx]]$class <- factor(cls.all[[idx]]$class)
#     idx <- idx + 1
#   }
# }
#barplot(summary(df.train$class))
#create linear svm models for all pairs of classes
#svmkernel <- 'linear'
svmkernel <- 'radial'
set.seed(1234)
gammar <- 10^(-6:-1)
costr <- 10^(-1:2)
eps <- 0.1
cls.svmr <- list()
```

```
idx <- 1
for (i in 1:(cls.n-1)) {
for (j in (i+1):cls.n) {
c1 <- cls.levels[i]
c2 <- cls.levels[j]
#select data only for these classes
cls.both <- df.train[df.train$class == c1 | df.train$class == c2,]
cls.both$class <- factor(cls.both$class)
wts <- NULL#100 / table(cls.both$class)
#cls.svmr[[idx]] <- createSVMradial(cls.both, gammar, costr, eps, wts)
#default parameters
cls.svmr[[idx]] <- svm(class~., cls.both)
rm(cls.both)
idx <- idx + 1
   }
}
#****************************************************************************
# DCSVM prediction
#****************************************************************************
#initialize
info <- initSVMDC(df.train, cls.n, cls.levels, cls.svmr, cls.svmr,tshold
= THRESHOLD)
print(Sys.time())
allpercentages[[DATA_GEN]] <- sort(as.matrix(info$maxpredictp[,-1]))
}
```

B.6 PLOTTING GRAPHS

```
#read summary table
#allresults <- read.csv("output/results-tshold2.csv", header = T)[-12,]
#allresults <- read.csv("output/allresults-tshold2_20180408_063047.csv"
, header = T)
#latest results
allresults2 <- read.csv("output/allresults-tshold2_20180408_063047.csv"
, header = T)
allresults1 <- read.csv("output/allresults-tshold1_20180411_233531.csv"
, header = T)
```

```
allresults01 <- read.csv("output/allresults-tshold0.1_20180410
_195358.csv", header = T)
allresults001 <- read.csv("output/allresults-tshold0.01
_20180412_233641.csv", header = T)
allresults <- allresults2
#plots
par(mar=c(7,4,4,2)+0.1)
cnames <- colnames(allresults)
#- bar plot of triplets (build-in, one-vs-one, dcsvm) for their
# accuracies for all data sets
cols <- c(4,5,9)
# Grouped barplot
data <- t(as.matrix(allresults[,cols]))
colnames(data) <- allresults[,2]
rownames(data) <- c("Builtin", "1 vs. 1", "DCSVM")
barplot(data, col=colors()[c(28,139,100)] , border="white",font.axis=2,
beside=T, legend=rownames(data), ylab="Accuracy", ylim=c(0,140),
font.lab=2, las = 2)
#- bar plot of doubles (one-vs-one, dcsvm) for averages SVs, for all
# data sets
cols <- c(6,10)
# Grouped barplot
group2 <- c(8, 11, 12, 13, 14)
group1 <- -group2
data <- t(as.matrix(allresults[group1,cols]))
colnames(data) <- allresults[group1,2]
rownames(data) <- c("1 vs. 1", "DCSVM")
barplot(data, col=colors()[c(139,100)] , border="white", font.axis=2,
beside=T, legend=rownames(data), ylab="Average SVs", font.lab=2, las = 2)
par(mar=c(7,5,4,2)+0.1)
data <- t(as.matrix(allresults[group2,cols]))
colnames(data) <- allresults[group2,2]
rownames(data) <- c("1 vs. 1", "DCSVM")
barplot(data, col=colors()[c(139,100)] , border="white",font.axis=2,
beside=T, legend=rownames(data), ylab="", font.lab=2, las = 2)
title(ylab="Average SVs", line=4, font.lab=2, family = 'sans')
par(mar=c(7,4,4,2)+0.1)
#- bar plot of doubles (one-vs-one, dcsvm) for average number of steps,
```

```
# for all data sets
cols <- c(7,11)
# Grouped barplot
group2 <- c(10, 11, 12, 13, 14)
group1 <- -group2
data <- t(as.matrix(allresults[group1,cols]))
colnames(data) <- allresults[group1,2]
rownames(data) <- c("1 vs. 1", "DCSVM")
barplot(data, col=colors()[c(139,100)] , border="white", font.axis=2,
beside=T,legend=rownames(data),ylab="Average steps", font.lab=2, las = 2)
data <- t(as.matrix(allresults[group2,cols]))
colnames(data) <- allresults[group2,2]
rownames(data) <- c("1 vs. 1", "DCSVM")
barplot(data, col=colors()[c(139,100)] , border="white", font.axis=2,
beside=T, legend=rownames(data),ylab="Average steps",font.lab=2, las = 2)
#- bar plot of doubles (one-vs-one, dcsvm) for average time of
# predictions, for all data sets
cols <- c(8,12)
# Grouped barplot
group2 <- c(11, 12, 13, 14)
group1 <- -group2
data <- t(as.matrix(allresults[group1,cols]))
colnames(data) <- allresults[group1,2]
rownames(data) <- c("1 vs. 1", "DCSVM")
barplot(data,col=colors()[c(139,100)],border="white",
font.axis=2,beside=T, legend=rownames(data),ylab="Average time [sec]",
font.lab=2, las = 2)
#group2
data <- t(as.matrix(allresults[group2,cols]))
colnames(data) <- allresults[group2,2]
rownames(data) <- c("1 vs. 1", "DCSVM")
barplot(data,col=colors()[c(139,100)],border="white",
font.axis=2,beside=T, legend=rownames(data),ylab="Average time [sec]",
font.lab=2, las = 2)
#*********************************************************************
# aggregate info for different thresholds
#*********************************************************************
allresultsbythsold <- data.frame(No = allresults$No, Dataset =
```

```
allresults$Dataset, BIacc = allresults$BIacc, OVOacc = allresults$OVOacc)
#add threshold specific info
allresultsbythsold$DCSVMacc2 <- allresults2$DCSVMacc
allresultsbythsold$DCSVMdp2 <- allresults2$DCSVMdp
allresultsbythsold$DCSVMsteps2 <- allresults2$DCSVMsteps
allresultsbythsold$DCSVMavt2 <- allresults2$DCSVMavt
allresultsbythsold$DCSVMacc1 <- allresults1$DCSVMacc
allresultsbythsold$DCSVMdp1 <- allresults1$DCSVMdp
allresultsbythsold$DCSVMsteps1 <- allresults1$DCSVMsteps
allresultsbythsold$DCSVMavt1 <- allresults1$DCSVMavt
allresultsbythsold$DCSVMacc01 <- allresults01$DCSVMacc
allresultsbythsold$DCSVMdp01 <- allresults01$DCSVMdp
allresultsbythsold$DCSVMsteps01 <- allresults01$DCSVMsteps
allresultsbythsold$DCSVMavt01 <- allresults01$DCSVMavt
allresultsbythsold$DCSVMacc001 <- allresults001$DCSVMacc
allresultsbythsold$DCSVMdp001 <- allresults001$DCSVMdp
allresultsbythsold$DCSVMsteps001 <- allresults001$DCSVMsteps
allresultsbythsold$DCSVMavt001 <- allresults001$DCSVMavt
#dump this table in LaTeX format
library(xtable)
sink("output/tables/tables.Rnw")
cat("
\\documentclass{article}
\\usepackage{Sweave}
\\begin{document}
")
# Print a lot of tables
#invisible(
#  lapply(allresultsbythsold,
#         function(x)
#         print(xtable(table(x),caption=names(x)),table.placement="!htp"))
#)
#accuracies table
tcols <- c(2, 3, 4, 5, 9, 13, 17)
xtable(allresultsbythsold[,tcols])
#avg. steps table
tcols <- c(2, 7, 11, 15, 19)
xtable(allresultsbythsold[,tcols])
```

```
#avg. SVs table
tcols <- c(2, 6, 10, 14, 18)
xtable(allresultsbythsold[,tcols])
cat("
\\end{document}
")
sink()
cdir <- getwd()
setwd(paste0(cdir,'/output/tables/'))
Sweave("tables.Rnw")
#compilePdf("tables.Rnw")
setwd(cdir)
#plot some results
#I. accuracies for a few datasets
cols <- c(5,9,13,17)
thsholds <- c(2,1,0.1,0.01)
group <- c(10, 11, 12, 14)
colors <- colors()[c(28, 139, 100, 641)]
x <- matrix(thsholds)
y <- t(as.matrix(allresultsbythsold[group,cols]))
#colnames(data) <- allresults[group1,2]
#rownames(data) <- c("1 vs. 1", "DCSVM")
matplot(x, y, lty = 15:18, ylim = c(93,100),pch=15:18,
col= colors, xlab = "Threshold", ylab = "Accuracy") #plot
matlines(x, y, lty = 1, ylim = c(93,100),pch=15:18,col= colors, lwd = 2)
legend("topright", legend = allresultsbythsold[group,2], col=colors,
pch=15:18)
#II. accurracies for "letter", for all thresholds and all methods
cols <- c(5,9,13,17)
thsholds <- c(2,1,0.1,0.01)
group <- c(12)
colors <- colors()[c(28, 139, 100)]
tmpdf <- allresultsbythsold[group,cols]
tmpdf[2,] <- rep(allresultsbythsold[12,3],4)
tmpdf[3,] <- rep(allresultsbythsold[12,4],4)
legendlist <- c("letter", "BI", "OvsO")
x <- matrix(thsholds)
y <- t(as.matrix(tmpdf))
```

```
#colnames(data) <- allresults[group1,2]
#rownames(data) <- c("1 vs. 1", "DCSVM")
matplot(x, y, lty = 15:17, ylim = c(95,97),pch=15:17,
col= colors, xlab = "Threshold", ylab = "Accuracy") #plot
matlines(x, y, lty = 1, ylim = c(95,97),pch=15:17,col = colors, lwd = 2)
legend("topright", legend = legendlist, col=colors, pch=15:17)
#III. average steps for a few datasets
cols <- c(7,11,15,19)
thsholds <- c(2,1,0.1,0.01)
group <- c(10, 11, 12, 14)
colors <- colors()[c(28, 139, 100, 641)]
x <- matrix(thsholds)
y <- t(as.matrix(allresultsbythsold[group,cols]))
#colnames(data) <- allresults[group1,2]
#rownames(data) <- c("1 vs. 1", "DCSVM")
matplot(x, y, lty = 15:18, ylim = c(0,25),pch=15:18,col= colors, xlab =
"Threshold", ylab = "Average steps") #plot
matlines(x, y, lty = 1, ylim = c(0,25),pch=15:18,col= colors, lwd = 2)
legend("topright", legend = allresultsbythsold[group,2],col=colors,
pch=15:18)
#iV. distribution of splits in allresults table
# (must have run #experim-results-alldata2.R)
# data must be already stored in allpercentages[[DATA_GEN]]
range <- c(0, 5)
dataset <- 'letter'
dataset <- 'vowel'
dataset <- 'usps'
dataset <- 'sensorless'
x <- seq(from = range[1], to = range[2], by = 0.01)
y <- x
for (i in 1:length(x)) {
y[i] <- length(allpercentages[[dataset]][allpercentages[[dataset]] >=
(100-x[i])])
}
plot(x,y, col = colors, main = paste0("'",dataset," ' dataset"), xlab =
'Threshold', ylab = 'Separated classes')
#V. distribution of splits in allresults table
# (must have run experim-results-alldata2.R)
```

```
# data must be already stored in allpercentages[[DATA_GEN]]
# (same as above, but as percentages)
range <- c(0, 5)
dataset <- 'letter'
dataset <- 'vowel'
dataset <- 'usps'
dataset <- 'sensorless'
x <- seq(from = range[1], to = range[2], by = 0.01)
y <- x
for (i in 1:length(x)) {
l <- length(allpercentages[[dataset]])
y[i] <- length(allpercentages[[dataset]][allpercentages[[dataset]] >=
(100-x[i])]) / l * 100
}
plot(x,y, col = colors, main = paste0("'",dataset," ' dataset"), xlab =
'Threshold', ylab = 'Classes separation [%]')
#*********************************************************************
#                      Last update: 11/05/2018
#*********************************************************************
```