SVM Toolbox

Theory, Documentation, Experiments

TECHNISCHE
UNIVERSITÄT
DARMSTADT

S.V. Albrecht (sa@highgames.com)

Darmstadt University of Technology
Department of Computer Science
Multimodal Interactive Systems

Contents

List of Figures

List of Tables

1 Introduction

SVM Toolbox is a set of Matlab functions that provide access to basic SVM functionalities such as linear and non-linear separation of data points in an arbitrarily dimensional room. It was developed during a project within the lecture "Machine Learning: Statistical Approaches 2"[1], held by Prof. Bernt Schiele in winter term 2009 at Darmstadt University of Technology[2], Department of Computer Science[3].

The remainder of this document is structured as follows:

- Section 2 provides a short introduction to the theory of SVMs.
- Section 3 explains the SVM Toolbox and provides implementation details.
- Section 4 contains two experiments: one that investigates performance differences of two common multiclass approaches ("One vs. All" and "One vs. One"), and one that is aimed at outlining differences between inductive and transductive SVMs.

⇒ SVM Toolbox is publicly available under: `www.highgames.com/svm_toolbox`

[1] http://www.mis.tu-darmstadt.de/ml2
[2] http://www.tu-darmstadt.de
[3] http://www.informatik.tu-darmstadt.de

2 Support Vector Machines

Support Vector Machines (SVMs) are used to classify data (represented as real vectors) that belong to two or more distinct classes. They were first introduced by Vapnik in [5] and later in [1]. This chapter provides a very short introduction to the formal theory of SVMs. Note that no attempt has been made to cover the topic completely, nor to present it in a comprehensive manner. Furthermore, this chapter focuses on the *classification problem*, while SVMs can also be used to solve the *regression problem*, which is not covered by this document.

2.1 Problem Specification

Consider a set $X = \left\{x_i \mid i \in \{1,...,n\} \wedge x_i \in \mathbb{R}^d\right\}$ and a set $Y = \left\{y_i \mid i \in \{1,...,n\} \wedge y_i \in \{-1,+1\}\right\}$. The elements x_i in X are called *examples* and the elements y_i in Y are called *class labels*. We say x_i belongs to class y_i, or x_i is labelled with y_i.

The goal is to find a hyperplane that separates all examples such that those examples located above the hyperplane belong to one class, and those below the hyperplane belong to the other class. This hyperplane can be described by $w \cdot x + b = 0$ where $w, x \in \mathbb{R}^d$, $b \in \mathbb{R}$, w is orthogonal to the hyperplane, and $\frac{b}{\|w\|}$ is the perpendicular distance from the hyperplane to the origin (see figure 2.1).

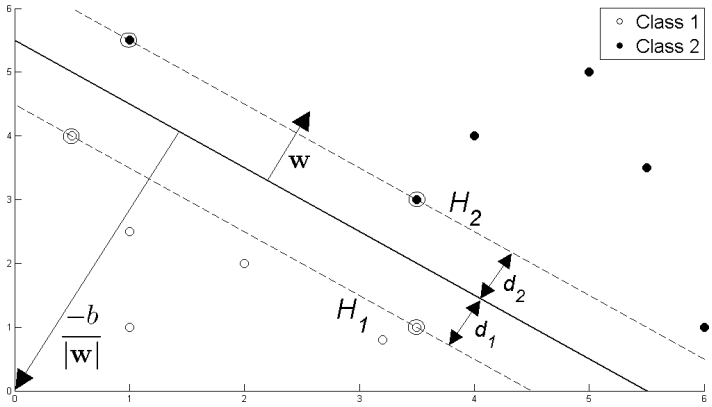

Figure 2.1: Hyperplane through two linearly separable classes (taken from [2])

2.2 Linear Separation

Consider the case where X is linearly separable. Then there exist $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that for all $i \in \{1,...,n\}$ the following holds:

$$w \cdot x_i + b \geq +1 \quad \text{if } y_i = +1$$
$$w \cdot x_i + b \leq -1 \quad \text{if } y_i = -1$$

Or, in one equation:

$$y_i(w \cdot x_i + b) - 1 \geq 0$$

In order to classify future data as accurate as possible, SVMs try to select w and b in a way that those examples closest to the hyperplane (called "Support Vectors", hence the name) lie as far away form it as possible. This approach is denoted as *maximum margin classification*.

Since the margin is equal to $\frac{1}{\|w\|}$, this is equivalent to finding

$$min\,\|w\| \quad \text{s.t.} \quad y_i(w \cdot x_i + b) - 1 \geq 0 \quad \forall i$$

which, of course, is equivalent to

$$min\,\tfrac{1}{2}\|w\|^2 \quad \text{s.t.} \quad y_i(w \cdot x_i + b) - 1 \geq 0 \quad \forall i.$$

The latter can be efficiently solved using a Quadratic Programming (QP) application.

2.3 Linear Separation with Misclassifications

Suppose X is "almost" linearly separable (as in figure 2.2), meaning just a few examples (e.g. 1 out of 10) need to be misclassified in order to achieve a linear separation of the data. We therefore introduce a new variable ξ_i (called *slack variable*) for each x_i that measures the deviation of x_i from the hyperplane in case of a wrong classification, i.e.

$$w \cdot x_i + b \geq +1 - \xi_i \quad \text{if } y_i = +1$$
$$w \cdot x_i + b \leq -1 + \xi_i \quad \text{if } y_i = -1$$
$$\text{where } \xi_i \geq 0 \ \forall i$$

Or, in one equation:

$$y_i(w \cdot x_i + b) - 1 + \xi_i \geq 0$$
$$\text{where } \xi_i \geq 0 \ \forall i$$

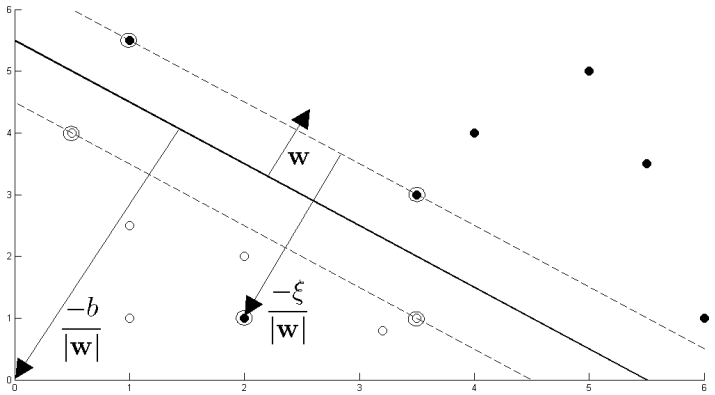

Figure 2.2: Hyperplane through two non-linearly separable classes (taken from [2])

The optimization problem now amounts to finding

$$min\,\tfrac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i \quad \text{s.t.} \quad y_i(w \cdot x_i + b) - 1 + \xi_i \geq 0 \ \forall i \ \text{ and } \ \xi_i \geq 0 \ \forall i$$

where C controls the trade-off between the slack variable penalty and the size of the margin.

2.4 Non-linear Separation with Kernels

Consider the case where X is not linearly separable, that is, there exists no hyperplane in dimension d that divides the space correctly into two classes (as on the left side in figure 2.3). In every such case, there exists at least one transformation (called *feature mapping*) $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d^+}$ where $d^+ > d$ that maps all $x \in \mathbb{R}^d$ into a higher-dimensional space such that the resulting set of transformed examples $\phi(X)$ is linearly separable through a hyperplane in $\mathbb{R}^{d^+}$ (as on the right side in figure 2.3).

Since d^+ is unbounded (i.e., $d^+ \rightarrow \infty$ is possible), it is often impractical to compute ϕ for even one example. We therefore use so-called kernel functions $k : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ that compute the dot product of two d-dimensional vectors in an implicitly higher space (i.e., $k(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$), and so we do not have to compute or even know a feature mapping ϕ (this is referred to as the *kernel trick*).

Several such kernel functions exist, some of which are more useful and some are less.

The most common kernels are:

- Linear Kernel: $k(x_i, x_j) = x_i \cdot x_j$
- Polynomial Kernel: $k(x_i, x_j) = (x_i \cdot x_j + a)^b$
- Radial Basis Function (RBF) Kernel: $k(x_i, x_j) = exp\left(\frac{-\|x_i - x_j\|^2}{2\sigma^2}\right)$
- Sigmoidal Kernel: $k(x_i, x_j) = tanh(a(x_i \cdot x_j) - b)$

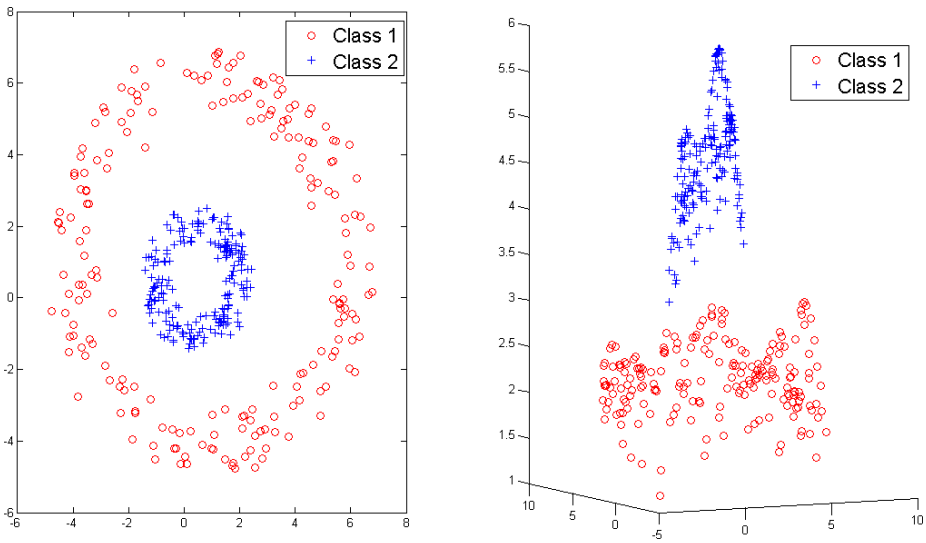

Figure 2.3: Implicitly re-mapped data using the RBF Kernel (taken from [2])

Using kernel functions in order to separate data that is not linearly separable requires substituting each dot product $x_i \cdot x_j$ in the original optimization problem with $k(x_i, x_j)$. However, if the transformation of the data is to be visualised, one has to determine an explicit feature mapping ϕ (not covered by this document).

2.5 Dual Form

Recall the minimization problem from section 2.3, which will be repeated for convenience:

$$min \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i \quad \text{s.t.} \quad y_i(w\cdot x_i + b) - 1 + \xi_i \geq 0 \ \forall i \ \text{ and } \ \xi_i \geq 0 \ \forall i$$

We introduce *Lagrange multipliers* α_i and β_i in order to put these terms into one expression:

$$\Phi_P \equiv \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i\left[y_i\left(w\cdot x_i + b\right) - 1 + \xi_i\right] - \sum_{i=1}^{n}\beta_i\xi_i$$

subject to

$$0 \leq \alpha_i \leq C \ \forall i, \ \beta_i \geq 0 \ \forall i, \ \text{and} \ \xi_i \geq 0 \ \forall i$$

This is called the *primal form* of the original problem. Minimizing Φ_P with respect to w, b, α_i and β_i is equivalent to the previous formulation. However, since we have more than one unknown variable, we need to reformulate the problem. Differentiating Φ_P with respect to w, b, and ξ_i and setting the derivatives to zero amounts to:

$$\frac{\delta\Phi_P}{\delta w} = 0 \Rightarrow w = \sum_{i=1}^{n}\alpha_i y_i x_i$$

$$\frac{\delta\Phi_P}{\delta b} = 0 \Rightarrow \sum_{i=1}^{n}\alpha_i y_i = 0$$

$$\frac{\delta\Phi_P}{\delta\xi_i} = 0 \Rightarrow C = \alpha_i + \beta_i$$

Substituting w in Φ_P reveals an equivalent formulation, known as *dual form*:

$$\Phi_D \equiv \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j k(x_i, x_j) - \sum_{i=1}^{n}\alpha_i \quad \text{s.t.} \quad 0 \leq \alpha_i \leq C \ \forall i \ \text{ and } \ \sum_{i=1}^{n}\alpha_i y_i = 0$$

where k is a kernel function (e.g. Linear Kernel, see section 2.4).

Minimizing Φ_D with respect to α_i is equivalent to minimizing Φ_P as before. The difference is that the dual form constitutes an optimization problem with merely one unknown variable, whereas the primal form has more than one. This is how SVMs are implemented in practice.

2.6 Transductive SVM

So far, SVMs are said to be a *supervised learning technique*, since each $x_i \in X$ has a corresponding label $y_i \in Y$. In many applications (e.g. speech or text recognition), it might turn out useful also to take into account those examples that have no prior label. The latter is called *transductive SVM (TSVM)*, whereas the former is called *inductive SVM*.

Let X^* be a set of examples x_i^* that have no prior label. The goal of transductive SVM is to find a label y_i^* for each x_i^* and a separating hyperplane $w\cdot x + b = 0$ such that it separates both labelled (X) and unlabelled (X^*) data with maximum margin (see figure 2.4; the dashed line corresponds to inductive SVM whereas the solid line corresponds to transductive SVM, taking the unlabelled data into account).

This corresponds to minimizing

$$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i + C^*\sum_{i=1}^{u}\xi_i^*$$

subject to

$$\forall_{i=1}^{n}\ y_i(w\cdot x_i + b) - 1 + \xi_i \geq 0$$
$$\forall_{i=1}^{u}\ y_i^*(w\cdot x_i^* + b) - 1 + \xi_i^* \geq 0$$
$$\forall_{i=1}^{n}\ \xi_i > 0$$
$$\forall_{i=1}^{u}\ \xi_i^* > 0$$

where C and C^* control the trade-off between the slack variable penalty and the size of the margin.

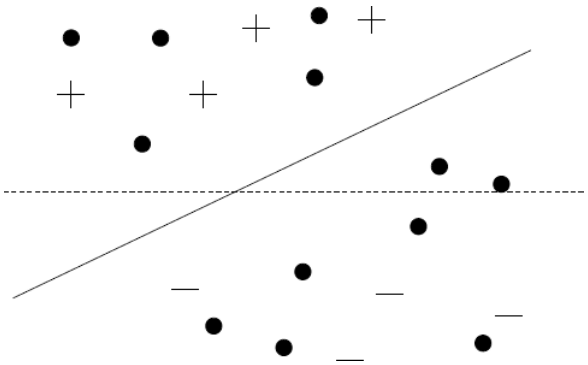

Figure 2.4: Separating hyperplane with transductive SVM (taken from [3])

2.7 Multiclass Problem

SVMs are *binary classifiers*, that is, we consider examples of two distinct classes. However, in real applications, we often face the problem of classifying more than two classes, which is referred to as the *multiclass problem*. Now, consider the case in which we have a set $L = \{L_1, ..., L_k\}$ of k classes (or *labels*). Two approaches to the multiclass problem have emerged to perform very well.

2.7.1 One vs. All

The One vs. All approach (OvA) exploits the fact that SVMs operate on continuous real values. Given a set X of training examples, OvA learns an SVM classifier $f_i(x) := w_i \cdot x + b_i$ for each L_i (positive class) against all other $L_{j\neq i}$ (assembled in the negative class). Thus, OvA reduces a k-multiclass problem into k binary problems. An example $x \in \mathbb{R}^d$ is now classified as L_i where $i = argmax_i\ f_i(x)$ (called *winner takes all strategy*).

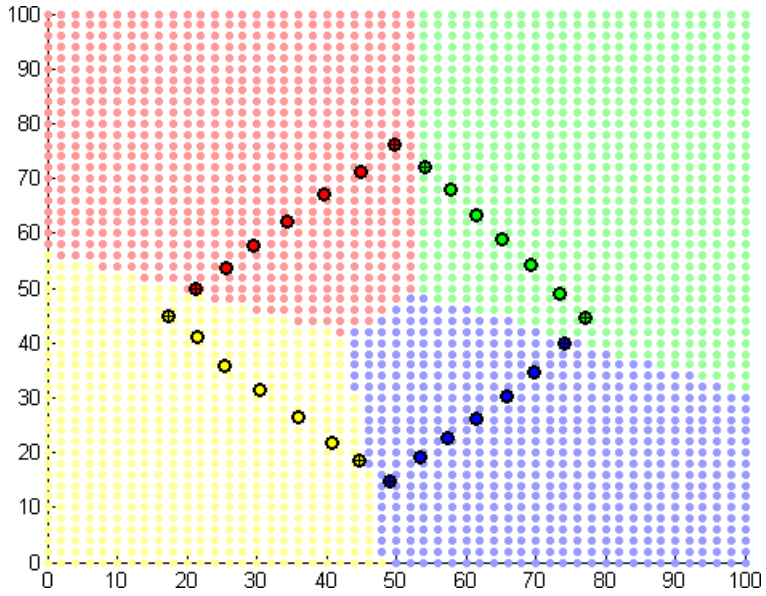

Figure 2.5: Multiclass problem with 4 classes (Linear Kernel and OvA)

2.7.2 One vs. One

The One vs. One approach (OvO) is more complex than OvA but also more accurate, and often applicable to cases where OvA performs miserably. Given a set X of training examples, OvO learns $k(k-1)/2$ SVM classifiers $f_{ij}(x) := sign\left(w_{ij} \cdot x + b_{ij}\right)$ that classify L_i (positive class) against L_j (negative class). Notice that for each f_{ij} there is no f_{ji}, i.e., there are no duplicates. Given an example $x \in \mathbb{R}^d$, OvO computes all $f_{ij}(x)$ and votes for L_i if $f_{ij}(x) > 0$, otherwise for L_j. Finally, x is classified as that L_i with the most votes (called *maximum votes strategy*).

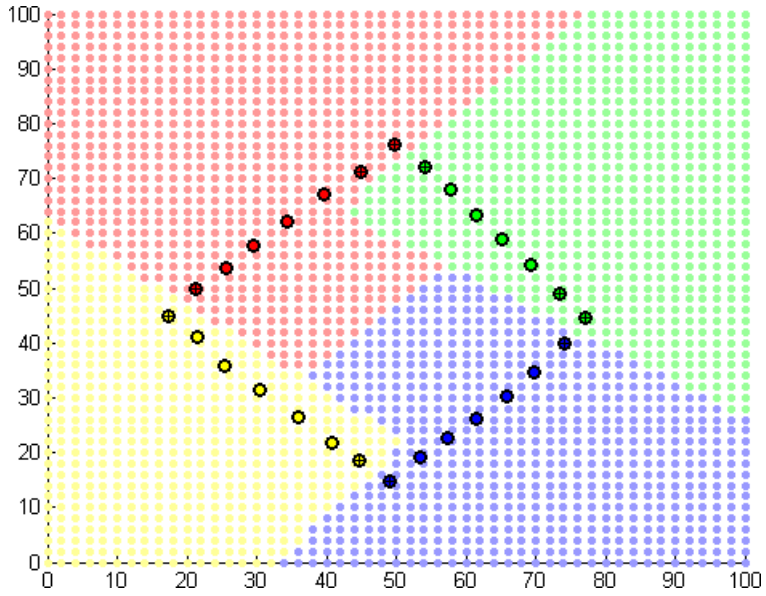

Figure 2.6: Multiclass problem with 4 classes (Linear Kernel and OvO)

3 SVM Toolbox

3.1 General Description

SVM Toolbox is a set of Matlab functions that provide access to basic SVM functionalities. It is mainly designed for educational purposes such as experimenting with the different kernel types or comparing the two multiclass approaches, but further applications may as well be possible. SVM Toolbox contains functions that implement all the "things" described in chapter 2, which is basically the following:

- Inductive learning *(see section 2.2)*
- Transductive learning *(see section 2.6)*
- Multiclass learning *(see section 2.7)*
- Non-linear separation *(see section 2.4)*
- 2D-Visualization
- Accuracy determination
- Heuristic parameter search *(see section 3.6)*

However, SVM Toolbox is neither designed to perform exceptionally well, nor does it support *regression* through SVMs. Interested programmers may feel free to add new functions or to improve existing ones. SVM Toolbox has been developed using Matlab 7.9.0 (R2009b), but former versions may work as well. Use of this software is not restricted by any license.

The remainder of this chapter gives an overview of the structure and functions in SVM Toolbox and provides implementation details of inductive and transductive learning. The chapter concludes with a discussion of a grid-based parameter search used in SVM Toolbox to find optimal or nearly optimal parameter settings.

3.2 Toolbox Structure and Functions

This section first describes the structure employed in SVM Toolbox and subsequently the most important functions provided by it. Note that each function is separately documented in its corresponding file. For more information regarding a function **`fun`** that is not nested, type "**`help fun`**" in Matlab.

3.2.1 Toolbox Structure

Consider figure 3.1. The figure shows a directed graph that contains the most important functions provided by SVM Toolbox. The graph is supposed to be read from left to right. Each column in the graph can be understood as a section where each section has immediate access to its right neighbour section, and thus implicit access to their respective right neighbours. The first section is called the *application section*, as it contains functions designed for experimental purposes only. The fifth section is called the *kernel section*, since it only consists of kernel functions.

Now, consider a path within the graph. Each path starts with a node labelled by a function of the application section and ends with a node labelled by a function of the kernel section. Such a path is called a *toolbox configuration*, it specifies all components used by a function of the application section.

Since the path is directed, each edge provides information regarding the access order, that is, if there are nodes A and B connected by an edge $A \rightarrow B$, then A calls B and later on uses its result value.

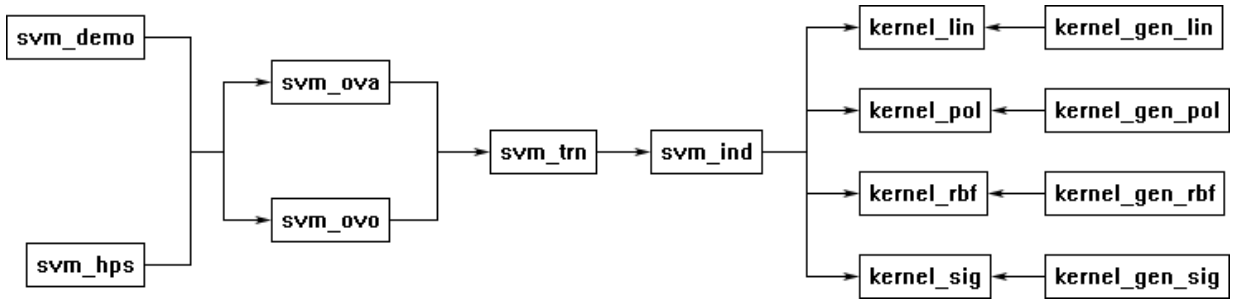

Figure 3.1: SVM Toolbox (structure)

3.2.2 Toolbox Functions

The following is a list of the most important functions contained in SVM Toolbox. This list is neither complete (but almost), nor is it intended as a replacement for the documentation in Matlab. The interested reader is encouraged to read the documentation in the corresponding function files as well.

svm_demo This function demonstrates and visualizes the functioning of (T)SVMs in a 2-dimensional coordinate system. The user is asked to enter data points (labelled and unlabelled) which will be used to determine a classifier according to the *application configuration* (see section 3.2.1). The data points can be stored in matrices for reuse with other configurations. It is also possible to define test data that can be used to determine accuracies or to find an optimal parameter setting (see `svm_hps`). For more information, type "`help svm_demo`".

svm_hps This function can be used to determine accuracies or to find an optimal parameter setting (see section 3.6). It takes as input a set X of training examples, a set Y of corresponding labels, a set N of neutral (unlabelled) data, a set X_T of test examples, a set Y_T of corresponding labels, function handles for a multiclass function and a kernel generator, argument (ranges) for the generator, the grid size, and the search depth (both for the parameter search) and computes the accuracy or an optimal parameter setting. For more information, type "`help svm_hps`".

svm_ova This function implements the multiclass approach "One vs. All" (OvA) as defined in section 2.7.1. It takes as input a set X of training examples, a set Y of corresponding labels, a set N of neutral (unlabelled) data, two misclassification penalty factors C_1 (for labelled data) and C_2 (for unlabelled data), and a kernel function and computes a classifier according to the definition of OvA. For more information, type "`help svm_ova`".

svm_ovo This function implements the multiclass approach "One vs. One" (OvO) as defined in section 2.7.2. It takes as input a set X of training examples, a set Y of corresponding labels, a set N of neutral (unlabelled) data, two misclassification penalty factors C_1 (for labelled data) and C_2 (for unlabelled data), and a kernel function and computes a classifier according to the definition of OvO. For more information, type "`help svm_ovo`".

svm_trn Implementation of *Transductive SVM* (see section 2.6). Although this function is based on the theory covered in section 2.6, its implementation follows another approach (see section 3.5). It takes as input a set X_+ of positive training examples, a set X_- of negative training examples, a set

X_0 of neutral (unlabelled) training examples, two misclassification penalty factors C_1 (for labelled data) and C_2 (for unlabelled data), and a kernel function and computes a separating hyperplane with maximum margin that involves the neutral data. For more information, type "`help svm_trn`".

svm_ind Implementation of *Inductive SVM* (see section 2.2). This function is based on the theory covered in section 2.5. It takes as input as set X of training examples, a set Y of corresponding labels, a set C of misclassification penalties for each example, and a kernel function and computes a separating hyperplane with maximum margin. For more information, type "`help svm_ind`".

kernel_lin *(nested function)* Linear kernel function as defined in section 2.4. This function is nested in `kernel_gen_lin` and can be obtained by a call of the latter. It takes as input two d-dimensional real vectors and computes the kernel value according to its definition. For more information, see documentation of `kernel_gen_lin`.

kernel_pol *(nested function)* Polynomial kernel function as defined in section 2.4. This function is nested in `kernel_gen_pol` and can be obtained by a call of the latter. It takes as input two d-dimensional real vectors and computes the kernel value according to its definition. For more information, see documentation of `kernel_gen_pol`.

kernel_rbf *(nested function)* Radial basis kernel function as defined in section 2.4. This function is nested in `kernel_gen_rbf` and can be obtained by a call of the latter. It takes as input two d-dimensional real vectors and computes the kernel value according to its definition. For more information, see documentation of `kernel_gen_rbf`.

kernel_sig *(nested function)* Sigmoidal kernel function as defined in section 2.4. This function is nested in `kernel_gen_sig` and can be obtained by a call of the latter. It takes as input two d-dimensional real vectors and computes the kernel value according to its definition. For more information, see documentation of `kernel_gen_sig`.

kernel_gen_lin This function generates a linear kernel (see `kernel_lin`). It can either be used to directly generate a kernel for `svm_demo`, or it can be passed through `svm_hps` as a function handle in order to dynamically generate new kernels during the parameter search. For more information, type "`help kernel_gen_lin`".

kernel_gen_pol This function generates a polynomial kernel (see `kernel_pol`). It can either be used to directly generate a kernel for `svm_demo`, or it can be passed through `svm_hps` as a function handle in order to dynamically generate new kernels during the parameter search. For more information, type "`help kernel_gen_pol`".

kernel_gen_rbf This function generates a radial basis kernel (see `kernel_rbf`). It can either be used to directly generate a kernel for `svm_demo`, or it can be passed through `svm_hps` as a function handle in order to dynamically generate new kernels during the parameter search. For more information, type "`help kernel_gen_rbf`".

kernel_gen_sig This function generates a sigmoidal kernel (see `kernel_sig`). It can either be used to directly generate a kernel for `svm_demo`, or it can be passed through `svm_hps` as a function handle in order to dynamically generate new kernels during the parameter search. For more information, type "`help kernel_gen_sig`".

3.3 Examples

Three examples will be shown in order to get an impression of what is possible with SVM Toolbox. The examples have been made using Matlab 7.9.0 (R2009b), but older versions may work as well.

3.3.1 Example 1: Inductive SVM

Figure 3.2 shows an inductive learning scenario with 7 distinct classes. Typing the command

```
svm_demo(10,0,@svm_ova,kernel_gen_rbf(10))
```

first opens a new figure. The user is then prompted to enter data points using the keys 1 to 9 for labelled data and 0 for neutral (unlabelled) data. Here, a face was drawn. Pressing the enter key performs SVM with the OvA multiclass approach (see section 2.7.1) using the radial basis kernel (see section 2.4) with $\sigma = 10$ and the misclassification penalty factor $C = 10$. The classification result is shown by means of the coloured areas. Pressing the space bar at an arbitrary position sets and classifies a new data point according to the learned model. Those data points marked by a plus (+) are support vectors.

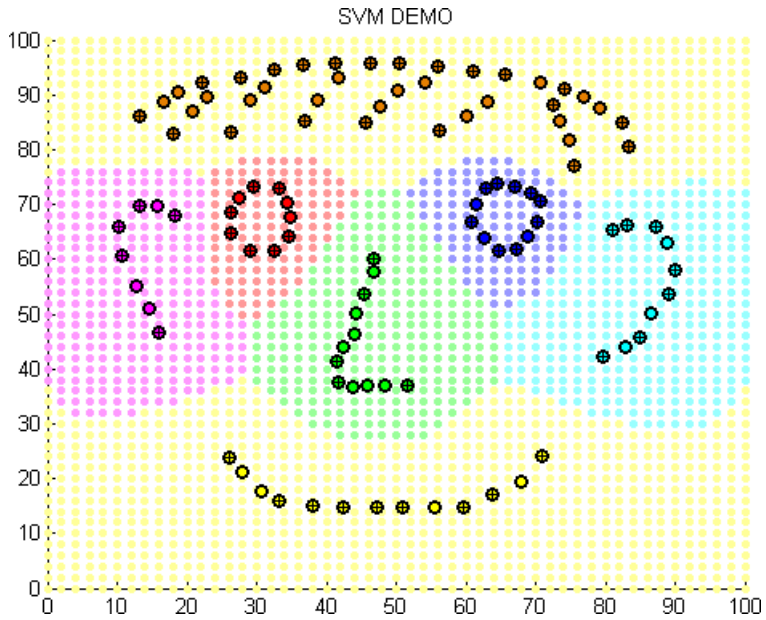

Figure 3.2: Example 1: Inductive SVM

3.3.2 Example 2: Transductive SVM

We now would like to see how transductive SVM (section 2.6) works.

We first start a new demo by typing the command

```
svm_demo(10,10,@svm_ova,kernel_gen_lin)
```

which states that the OvA multiclass approach (see section 2.7.1) and the linear kernel (see section 2.4) will be used along with the misclassification penalty factors $C_1 = 10$ (for labelled data) and $C_2 = 10$ (for unlabelled data). We then set two points as depicted in figure 3.3 and subsequently press the enter key in order to perform SVM, which results in the classification shown by the figure.

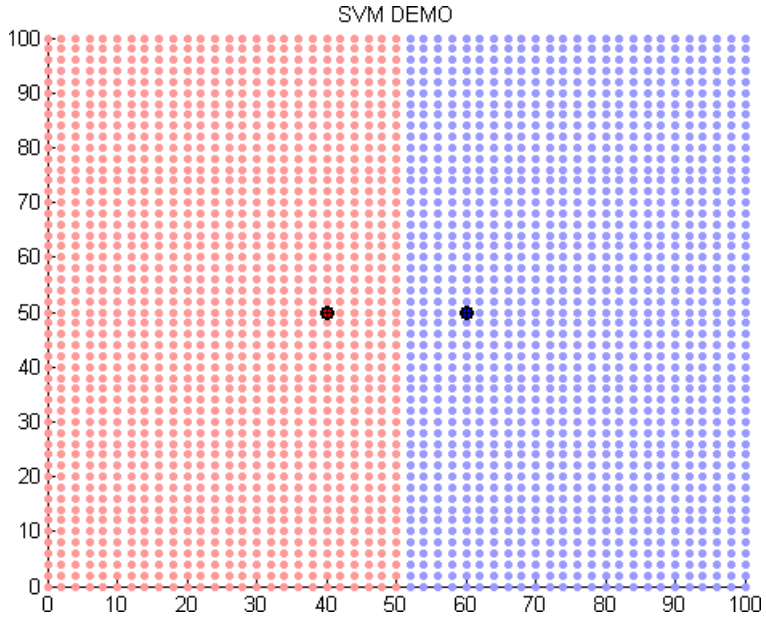

Figure 3.3: Example 2: Transductive SVM (1)

We now add some neutral data points in order to realign the separating hyperplane (as shown in figure 3.4). Neutral data points are represented by grey circles. If we now press enter and thereby learn a new classifier, we see that the new hyperplane has adjusted to the neutral data.

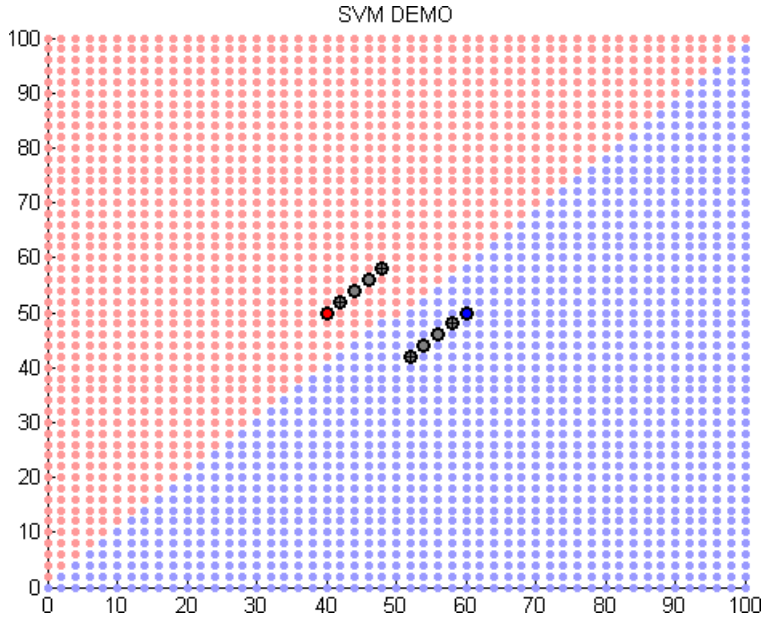

Figure 3.4: Example 2: Transductive SVM (2)

3.3.3 Example 3: Heuristic Parameter Search

Finally, let us see how the parameter search works. We begin by opening a new demo:

```
[X1,Y1,N,X2,Y2] = svm_demo(10,0,@svm_ovo,kernel_gen_pol([0,2]))
```

This time we apply the OvO multiclass approach (see section 2.7.2) and the polynomial kernel (see section 2.4) with $a = 0$ and $b = 2$ ($C_1 = 10$ and $C_2 = 0$). After adding training examples (circles) and test examples (triangles) as shown in figure 3.5, we press the enter key and thus learn a classifier. Clearly, the result is not as good as we would like it to be. We therefore close the demo by pressing the escape key and thereby store all training examples in X_1, Y_1 (labelled) and N (unlabelled), and, furthermore, all test examples in X_2 with the corresponding labels in Y_2.

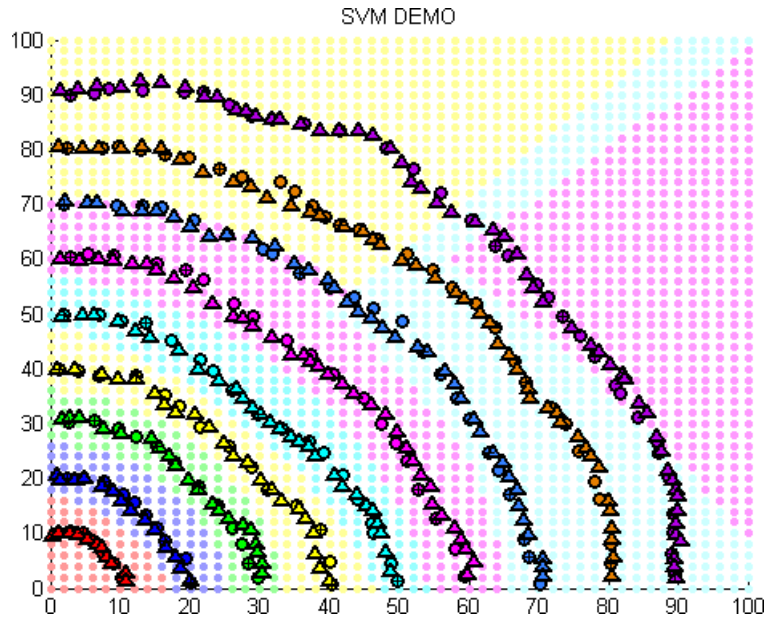

Figure 3.5: Example 3: Heuristic Parameter Search (1)

We now start a parameter search by entering the command

```
[acc,prm,t] =
svm_hps(X1,Y1,N,X2,Y2,@svm_ovo,@kernel_gen_pol,[0.1,100;0,0;0,10;0,10],3,5)
```

which states that, based on the previously stored data, an optimal parameter setting is to be searched for with the OvO multiclass approach and the polynomial kernel. The matrix `[0.1,100; 0,0; 0,10; 0,10]` means that C_1 will be restricted to the range $[10^{-1}, 10^2]$, C_2 to $[0, 0]$ (since there is no neutral data), a to $[0, 10]$, and b to $[0, 10]$. The parameter search will use a grid size of 3 and a search depth of 5 (see section 3.6). `acc` will contain the resulting accuracy based on the found parameter setting in `prm`, `t` will contain the computation time in seconds.

As a result of the parameter search, we get

```
acc = 1 and prm = [25.0750; 0; 2.5000; 2.5000]
```

that is, we get a perfect classification when using $C_1 = 25.0750$, $C_2 = 0$, $a = 2.5$, and $b = 2.5$.

Incorporating this information results in the following command:

```
svm_demo(25,0,@svm_ovo,kernel_gen_pol([2.5,2.5]),X1,Y1,N,X2,Y2)
```

Note that the stored data is appended to the command in order to reconstruct the same scenario as before. An optimal classification based on the found parameter setting is shown in figure 3.6.

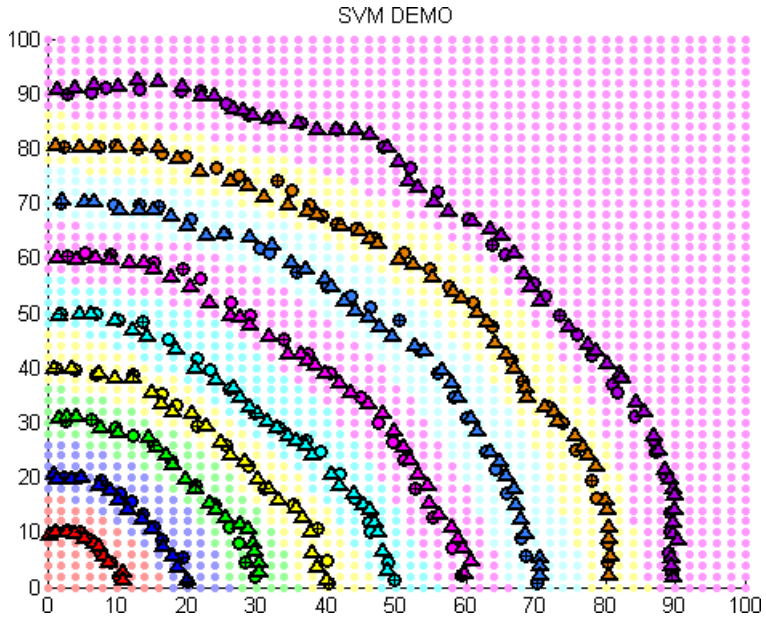

Figure 3.6: Example 3: Heuristic Parameter Search (2)

3.4 Implementation of svm_ind (Inductive SVM)

The function `svm_ind` implements *inductive SVM* based on the theory covered in section 2.5. The implementation basically follows Fletcher's paper [2]. However, since `quadprog` from the Optimization Toolbox[1] is used to solve the optimization problem, some deviations were made in order to adjust the algorithm to `quadprog`'s input specification. Furthermore, as we do not know the feature mapping ϕ (see section 2.4), the algorithm must rely on the given kernel function k only, and so the resulting classifier f is formulated as

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i k(x_i, x) + b$$

instead of the classical formulation

$$f(x) = w \cdot \phi(x) + b$$

where ϕ is an explicit feature mapping. In order to classify an example x, one needs to compute $sgn\big(f(x)\big)$. The sign function is left out, since we need the real-valued result of $f(x)$ for the OvA multiclass approach (see section 2.7.1).

[1] http://www.mathworks.de/access/helpdesk/help/toolbox/optim/

3.4.1 Input

`svm_ind` takes as input (in given order):

- A matrix $X \in \mathbb{R}^{n \times d}$ of n training examples, where each $x_i \in X$ corresponds to a row in X.
- A vector $Y \in \{+1, -1\}^n$ of corresponding class labels ($x_i \in X$ is labelled with $y_i \in Y$).
- A vector $C \in \mathbb{R}^n_{\geq 0}$ of penalty factors ($x_i \in X$ is penalized by factor $c_i \in C$ if misclassified).
- A function handle k of a kernel ($k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, $(x_i, x_j) \mapsto \phi(x_i) \cdot \phi(x_j)$).

3.4.2 Output

`svm_ind` returns as output (in given order):

- A function handle f of a classifier (defined as above).
- A matrix $S_X \in \mathbb{R}^{s \times d}$ of s support vectors (S_X "$\subseteq$" X).
- A vector $S_Y \in \{+1, -1\}^s$ of corresponding labels (S_Y "$\subseteq$" Y).
- A vector $S_\alpha \in \mathbb{R}^s_{\geq 0}$ of non-negative Lagrange multipliers ($x_i \in S_X$ is multiplied by $\alpha_i \in S_\alpha$).

3.4.3 Algorithm

`svm_ind` works as follows (in given order):

- Compute symmetric matrix $H \in \mathbb{R}^{n \times n}$, where $H_{ij} = y_i y_j k(x_i, x_j)$
- Minimize $\frac{1}{2}\alpha^T H \alpha - \sum_{i=1}^{n} \alpha_i$ in $\alpha = (\alpha_1, ..., \alpha_n)$ s.t. $0 \leq \alpha_i \leq c_i$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$ (using `quadprog`)
- Determine the matrices S_X, S_Y, and S_α such that $x_i \in S_X \Leftrightarrow x_i \in X \wedge 0 < \alpha_i < c_i$
- Set $S_i = \{i \mid x_i \in S_X\}$ (set of support vector indices)
- Compute $b = \frac{1}{|S_X|} \sum_{i \in S_i} \left(y_i - \sum_{j \in S_i} \alpha_j y_j k(x_j, x_i) \right)$
- Return classifier $f(x) = \sum_{i \in S_i} \alpha_i y_i k(x_i, x) + b$

3.5 Implementation of svm_trn (Transductive SVM)

The function `svm_trn` implements *transductive SVM* based on the theory covered in section 2.6. It is an instance of Joachims' algorithm [3], but has been slightly modified since the original algorithm, as described in section 4.1 in [3], assumes the primal form of the optimization problem (see section 2.2).

3.5.1 Input

`svm_trn` takes as input (in given order):

- A matrix $X_+ \in \mathbb{R}^{n_1 \times d}$ of n_1 positive training examples, where $x \in X_+$ correspond to rows in X_+.
- A matrix $X_- \in \mathbb{R}^{n_2 \times d}$ of n_2 negative training examples, where $x \in X_-$ correspond to rows in X_-.
- A matrix $X_0 \in \mathbb{R}^{n_3 \times d}$ of n_3 unlabelled training examples, where $x^* \in X_0$ correspond to rows in X_0.
- A real number $C_1 \in \mathbb{R}$ that serves as misclassification penalty factor for *labelled* data.
- A real number $C_2 \in \mathbb{R}$ that serves as misclassification penalty factor for *unlabelled* data.
- A function handle k of a kernel ($k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, $(x_i, x_j) \mapsto \phi(x_i) \cdot \phi(x_j)$).

3.5.2 Output

`svm_trn` returns as output (in given order):

- A function handle f of a classifier (defined as in section 3.4).
- A matrix $S_X \in \mathbb{R}^{s \times d}$ of s support vectors (S_X "$\subseteq$" $X_+ \cup X_- \cup X_0$).
- A vector $S_Y \in \{+1, -1\}^s$ of corresponding labels.
- A vector $S_\alpha \in \mathbb{R}^s_{\geq 0}$ of non-negative Lagrange multipliers ($x_i \in S_X$ is multiplied by $\alpha_i \in S_\alpha$).

3.5.3 Algorithm

`svm_trn` works as follows (in given order):

- Append X_- to X_+ and denote the resulting matrix as X ($X \in \mathbb{R}^{(n_1+n_2) \times d}$)
- Create $Y \in \{+1, -1\}^{n_1+n_2}$, where the first n_1 entries have the value $+1$, the rest -1
- Create $C \in \mathbb{R}^{n_1+n_2}_{\geq 0}$, where each entry has the value of C_1

- Learn initial classifier f by calling `svm_ind`(X, Y, C, k)
- **If** $n_3 = 0$ (i.e., X_0 is empty) **then** stop algorithm and return f
- Create $Y_0 \in \{+1, -1\}^{n_3}$, where each $y_i^* \in Y_0$ labels an $x_i^* \in X_0$ (using f)
- Append X_0 to X and Y_0 to Y
- **For** $c = -5$ to 0 **do**
- Create $C_0 \in \mathbb{R}_{\geq 0}^{n_3}$, where each entry has the value $C_2 * 10^c$
- Append C_0 to C and denote the resulting vector as C^*
- (Ω) Update classifier f using `svm_ind`(X, Y, C^*, k)
- **If** there are switchable $y_i^* \in Y_0$ and $y_j^* \in Y_0$ **then** switch them and go back to (Ω)
- **end for**
- Return classifier f

Note: $y_i^* \in Y_0$ and $y_j^* \in Y_0$ are switchable if $y_i^* \neq y_j^*$ and

$$max\left(0, 1 - y_i^* f(x_i^*)\right) + max\left(0, 1 - y_j^* f(x_j^*)\right) > max\left(0, 1 - y_j^* f(x_i^*)\right) + max\left(0, 1 - y_i^* f(x_j^*)\right)$$

3.6 Heuristic Parameter Search

In general, the quality of an SVM classifier is immediately dependent on a proper parameter setting. However, searching for optimal parameters is impractical when done without computational support, as the parameter space grows exponentially with the number of parameters. For instance, consider a learning scenario with labelled and unlabelled data, using a polynomial kernel. Here, we already have four parameters: C_1, C_2, a, and b. Other kernels might require even more parameters to be set.

Several approaches exist that try to find an optimal parameter setting with or without user interaction. SVM Toolbox provides an automatic parameter search which is based on a proposal by Carl Staelin [4]. It is a local (or *greedy*) search, since each iteration within the algorithm requires concentrating around the previous optimum. The user may set the granularity and the search depth of the algorithm, and thereby control the trade-off between quality and convergence. This section briefly explains the functioning[2] of the parameter search (henceforth *HPS*). For more details, see documentation of `svm_hps`.

[2] For a usage example, see section 3.3.3

3.6.1 Algorithm

HPS is a *grid-based* approach. For each of the n parameters, the user is required to specify a range of values in which an optimum is to be searched. Altogether, these ranges span a room in which an optimal parameter setting is to be found. HPS now fits a search pattern into this room (as in figure 3.7) and samples parameter settings at the given spots. Each setting will be evaluated using a previously defined test set. The next iteration scales the pattern down (i.e., the resolution is increased) and positions its centre to the last optimum (i.e, the search is concentrated around a promising parameter setting). This process continues until a perfect setting is found (i.e, 100% accuracy), or the maximum search depth, as specified by the user, is reached.

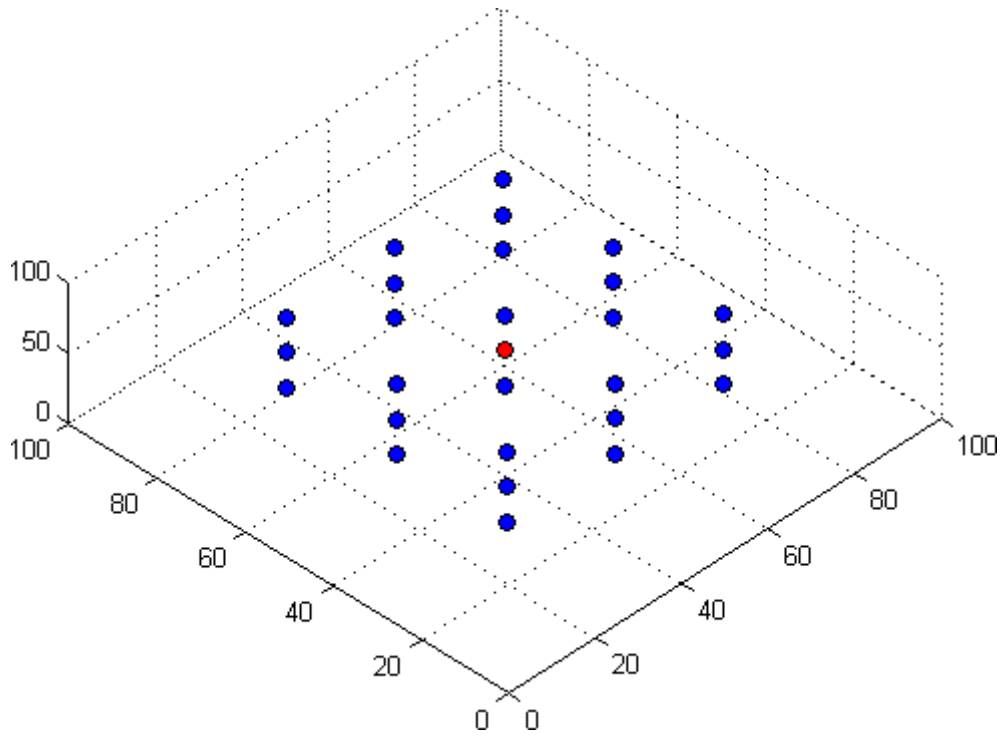

Figure 3.7: Parameter search pattern (3×3×3)

Consider figure 3.7. Here, we try to find an optimal parameter setting for three parameters. For simplicity, each parameter is restricted to the range $[0, 100]$, although arbitrary ranges are possible. We use a search pattern of grid size 3, that is, we have a 3×3×3 sampling pattern. Each spot corresponds to a specific parameter setting and therefore is a 3-dimensional vector, where each entry specifies the value of a certain parameter. The red spot marks the centre of the pattern.

4 Experiments

This section contains the results of two little (or rather tiny) experiments, that have been examined using SVM Toolbox. One experiment is concerned about finding differences between the multiclass approaches OvA and OvO (see section 2.7), the other one in differences between inductive SVM and transductive SVM (see sections 2.2 and 2.6, respectively).

Both experimental data sets have in common that none of them is comparable to what we would expect in real applications. Each data set was manually created, merely consists of 2-dimensional examples, and, furthermore, has much less examples than practical data sets do. Nevertheless, they at least give a first impression of how the considered approaches behave in different settings.

The experiments have been performed using Matlab 7.9.0 (R2009b) on an Intel machine with Windows XP (Home). The experimental data sets (and their respective results) are publicly available under `www.highgames.com/svm_toolbox/experiments`.

4.1 Multiclass Problem: OvA vs. OvO

This experiment is concerned about performance differences between the multiclass approaches OvA ("One vs. All") and OvO ("One vs. One"). A basic understanding of the theoretical foundations is recommended, the reader is therefore encouraged to read section 2.7 before continuing.

The experiment consists of 12 different data sets altogether, whereof 3 data sets contain 2, 4, 6, and 8 classes, respectively. Each data set was tested using the linear kernel, the polynomial kernel, and the RBF kernel. Additionally, each test run involved the heuristic parameter search of SVM Toolbox (see section 3.6) to find an optimal parameter setting, using a grid size of 3 and a search depth of 3. For completeness, each data set consists of at least 2 and at most 62 training examples, and of at least 85 and at most 390 test examples. The results using OvA are shown in table 4.1, those using OvO in table 4.2.

Consider the results using the OvA approach in table 4.1. Those data sets that contain 2 classes were perfectly classified in an average time of 0.1705 seconds for all kernels. Using 4 classes (or more) reveals that using the OvA approach together with the linear kernel is not a proper choice. The polynomial kernel still works fine for 4 classes, but breaks down for 6 or more classes. The RBF kernel always works perfectly in a comparingly short amount of time.

Now, consider the results using the OvO approach in table 4.2. Those data sets that contain 2 classes were perfectly classified in an average time of 0.0954 seconds for all kernels, which is clearly better than OvA. Using 4 classes (or more) shows that using the OvO approach together with the linear kernel still might be a good choice. With a proper parameter setting, this combination even achieves perfect separation in some cases. The polynomial kernel works fine for 4 classes or more, but has some exceptions as one can see. Again, the RBF kernel works perfectly with any data set and is slightly faster than OvA.

Altogether, we have a final result of OvO being about 22% more accurate than OvA, and OvO being more than 233% faster than OvA, which is unique. However, one exception must be made. The RBF kernel showed no difference for both variants except OvO being slightly faster, but this may be neglected. The reason for this can be found when considering the form of the RBF kernel. It basically selects a mid-

dle point and draws a circle around it. Since OvA uses the distance from an example to the hyperplane, its classification quality equals that of OvO, which does not gain any benefit from taking the results of multiple classifiers into account (using the RBF kernel).

The results are surprising in that OvO is generally expected to be more accurate than OvA, but also much slower, since its complexity is $O(n^2)$ (polynomial growth), whereas that of OvA is $O(n)$ (linear growth), where n is the number of classes. However, as we know from the field of Complexity Theory, there is a n_0 such that for all $n > n_0$ OvA will be faster than OvO. In this experiment, OvO is faster solely because of the little amount of classes and examples. Note that each classifier in OvO is based on an optimization problem with comparingly few examples, whereas each classifier in OvA is based on an optimization problem with comparingly many examples.

4.2 Inductive SVM vs. Transductive SVM

This experiment investigates differences between inductive SVM and transductive SVM. The reader is recommended to read the sections 2.2 and 2.6 to get a basic notion of the underlying concepts before proceeding. Henceforth, inductive SVM will be abbreviated by *SVM* and transductive SVM by *TSVM*.

Again, 12 different data sets build the foundation of this experiment. 4 of these are especially well suited for the linear kernel, 4 for the polynomial kernel, and 4 for the RBF kernel. Each data set was tested using only those kernels for which they were well suited. Here, we do only consider binary classification problems, since significant deviations may directly be generalized to the multiclass problem (as OvA and OvO both reduce the multiclass problem to multiple instances of the binary problem). Finally, we use the constant parameters $a = 0$ and $b = 2$ for the polynomial kernel (called *homogeneous quadratic kernel*), and $\sigma = 10$ for the RBF kernel. The misclassification penalty factors are set to $C_1 = 10$ (labelled data) and $C_2 = 10$ (unlabelled data). The amount of training examples ranges between 2 and 62, that of neutral examples between 8 and 41, and that of test examples between 29 and 137.

The results of the experiment are shown in table 4.3. We see that SVM and TSVM have the same accuracy, except for some cases where TSVM is slightly less accurate than SVM. This results in a 1% worse classification when using TSVM. However, the big drawback of TSVM is its time consumption, as one can immediately see. TSVM is at least 10 times (rounded), and at most 1200 times (rounded) slower than SVM. In average, TSVM is roughly 263 times slower than SVM.

The results might be somewhat confusing, as one might expect TSVM to be at least as accurate as SVM. However, one has to take into account that this experiment merely reveals TSVM being slightly less accurate in some cases, but there might as well be cases where TSVM performs better. Within this experiment, TSVM could not gain any benefit from the neutral data, but though was exceedingly slow compared to SVM. One even had to wait for more than a minute before TSVM terminated, were SVM was done in 60 milliseconds. However, TSVM is likely to outperform SVM in cases where we have little training data, but a huge amount of neutral data, as is the case in text recognition (e.g., see [3]).

Cl.	Set	lin:	Acc. (%)	Time (s)	pol:	Acc. (%)	Time (s)	rbf:	Acc. (%)	Time (s)
2	1		1	0.0709		1	0.1052		1	0.2358
	2		1	0.1019		1	0.1218		1	0.0896
	3		1	0.2408		1	0.1775		1	0.3918
	Ø		*1*	*0.1378*		*1*	*0.1348*		*1*	*0.2390*
4	1		0.3448	6.4181		1	0.6977		1	7.5143
	2		0.4944	6.3821		1	0.7592		1	1.3080
	3		0.2723	10.4974		0.9906	101.3638		1	2.0460
	Ø		*0.3705*	*7.7658*		*0.9968*	*34.2735*		*1*	*3.6227*
6	1		0.4571	19.8477		1	62.6045		1	4.6990
	2		0.5152	14.8929		0.8961	154.2432		1	4.1805
	3		0.1417	26.6774		0.3858	288.7697		1	6.7236
	Ø		*0.3713*	*20.4726*		*0.7606*	*168.5391*		*1*	*5.2010*
8	1		0.2734	40.9409		0.4775	409.9273		1	10.6384
	2		0.2769	50.5544		0.4538	530.0074		1	14.6422
	3		0.2	80.5452		0.2522	1102.70		1	9.0250
	Ø		*0.2501*	*57.3468*		*0.3945*	*680.8782*		*1*	*11.4352*
	Ø		***0.4980***	***21.4308***		***0.7880***	***220.9564***		***1***	***5.1245***

Table 4.1: Experimental results: OvA vs. OvO (OvA)

Cl.	Set	lin: Acc. (%)	Time (s)	pol: Acc. (%)	Time (s)	rbf: Acc. (%)	Time (s)
2	1	1	0.0482	1	0.0491	1	0.1253
	2	1	0.0555	1	0.0655	1	0.0598
	3	1	0.0993	1	0.0972	1	0.2592
	Ø	*1*	*0.0676*	*1*	*0.0706*	*1*	*0.1481*
4	1	0.8138	3.6786	1	0.3807	1	5.8212
	2	1	0.3993	1	0.4483	1	1.1061
	3	0.7089	5.2851	0.9859	52.7378	1	1.6709
	Ø	*0.8409*	*3.1210*	*0.9953*	*17.8556*	*1*	*2.8660*
6	1	1	0.9468	1	1.0564	1	3.1091
	2	0.9740	8.5817	0.9870	71.9098	1	2.9128
	3	0.5551	16.4394	0.6496	146.0251	1	4.5782
	Ø	*0.8430*	*8.6559*	*0.8788*	*72.9971*	*1*	*3.5333*
8	1	1	2.0686	1	1.8859	1	7.1025
	2	0.6385	27.8428	1	72.6914	1	10.1715
	3	0.5087	36.0530	0.7217	395.1113	1	10.0006
	Ø	*0.7157*	*21.9881*	*0.9072*	*156.5628*	*1*	*9.0915*
	Ø	***0.8499***	***8.4581***	***0.9453***	***61.8715***	***1***	***3.9097***

Table 4.2: Experimental results: OvA vs. OvO (OvO)

Kernel	Set	SVM:	Acc. (%)	Time (s)	TSVM:	Acc. (%)	Time (s)
lin	1		1	0.0084		1	9.0608
	2		1	0.0042		1	0.2411
	3		1	0.0540		0.9773	1.1985
	4		1	0.0145		0.9275	5.9019
	Ø		*1*	*0.0202*		*0.9762*	*4.1005*
pol	1		1	0.0285		1	0.4243
	2		1	0.0750		1	0.7328
	3		1	0.0788		1	1.1911
	4		1	0.0462		1	1.3731
	Ø		*1*	*0.0571*		*1*	*0.9303*
rbf	1		1	0.0754		1	1.4967
	2		1	0.1175		1	25.6407
	3		1	0.0609		0.9922	68.1199
	4		1	0.0337		0.9635	41.3577
	Ø		*1*	*0.0718*		*0.9889*	*34.1537*
	Ø		***1***	***0.0497***		***0.9883***	***13.0615***

Table 4.3: Experimental results: SVM vs. TSVM

Bibliography

[1] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20:273–297.

[2] T. Fletcher. Support vector machines explained. Tutorial Paper - PhD, 2008.

[3] T. Joachims. Transductive inference for text classifcation using support vector machines. *International Conference on Machine Learning (ICML)*, 1999.

[4] C. Staelin. Parameter selection for support vector machines. Technical report, HP Laboratories Israel, 2003.

[5] V. Vapnik and A. Lerner. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24:774–780, 1963.