Kernel Machines for Multi-Class Classification: A Joint Kernel Approach

Antolin Thomas Janssen

Radboud University
Institute for Computing and Information Sciences
Nijmegen, The Netherlands, 6525 AJ

2009

Supervisors

Evgeni Tsivtsivadze
Institute for Computing and Information Science
Radboud University
Nijmegen, 6525 AJ
The Netherlands

Tom Heskes
Institute for Computing and Information Science
Radboud University
Nijmegen, 6525 AJ
The Netherlands

Abstract

Pattern analysis is about the automatic detection of patterns in data, and plays an important role in many modern artificial intelligence and computer science problems. With patterns we mean any relations, regularities or structures that are present within a source of data. By detecting significant patterns in the data, a system can make predictions about new data coming from the same source. We can say that the system has acquired generalisation power by learning a pattern in the data from that source. There are many important problems that can only be solved using this approach, ranging from bio informatics to web retrieval. In recent years, pattern analysis has become a standard method for software engineering, and is used in many commercial products.

This thesis describes several algorithms and compares their performances on different datasets. This to research the possibilities that we have when we want to extract possibly interesting relationships between datapoints. The improvement of these kind of techniques can be important for future research because with increasing computational power, increasingly larger datasets can be analyzed, so that we can find relationships we never knew existed. One of these datasets is called SCOP, or the Structured Classification of Proteins, which we can analyze to better understand how our body works and how different proteins are related. To detect these relationships we use an algorithm called Regularized Least Squares. Also its performance is compared to a very similar algorithm, called SVMLight, and several other machine learning techniques, namely Naive Bayes, KStar and Random Forest. We show that the performance depends on the problem that is analyzed, where things like linear separability and the number of examples in the dataset have a notable influence. Nevertheless, the results are stable on most datasets, resulting in a prediction system that can be considered as a reliable advise-tool for making decisions about that dataset. A rapidly developing field within this research is multiclass classification, which extends the binary classification into a method that can handle problems with multiple classes. The multiclass extension for the algorithm that we consider is a joint kernel approach which also provides stable results and could be optimized to improve performance even more.

Contents

Chapter 1

Introduction

In this chapter we describe the methods and tools used during this research project. First of all there is an extended description of the technique that we implemented and its theoretical foundations. Second, there is a less extended description of the SVMLight algorithm, a similar algorithm to ours that has been optimized and has been shown to have good empirical performance on a wide variety of datasets. In the third section we propose a multi-class extension applicable to various kernel-based methods and briefly state the ideas behind that technique. To finish the first chapter the other investigated algorithms are described.

The goal of all examined algorithms is to detect patterns in datasets, or to be more exact, to determine the relations between datapoints by using their properties to calculate the similarities between those datapoints. To start our research into the field of pattern analysis, first of all we need a problem setting in which we are conducting our experiments, which is the following: There is a dataset, which consists of a number of datapoints that each have a class and a certain number of features. These features represent the important properties of the datapoint. We want to analyze the relation between the features of a datapoint and its class label, creating a model which describes the estimated predicted relationships for that dataset. After that we can use this model to predict the class label for a datapoint which we haven't seen before.

1.1 RLS - Binary

The algorithm on which most of this thesis is based is called Regularized Least Squares. It is described in details in [14] and in principle is an "extended form" of Least Squares Linear Regression. It is also known as kernelized version of Ridge Regression and has been given different names by different authors, namely LS-SVM [23] and Regularized Least-Squares [22],

[21]. This technique uses dot products (also called kernel functions) to calculate the similarity between training examples. There are some mathematical tricks to optimize this technique but the basic idea is pretty straightforward, and consists of two steps:

- Mapping into feature space
- Computing similarities

The first step makes sure that the datapoints are linearly separable by mapping them into a higher dimension, which is done by a so called kernel function. The second part consists of the learning algorithm which will do the actual pattern recognition. This goes, as said above, by creating a model using the dot products between the different datapoints. Once this is done we can run the linear algorithm on unseen datapoints and make a prediction with respect to the class of that datapoint.

1.1.1 Linear Regression

To come to a more in depth description of the idea behind the algorithm, first of all we start with Linear Regression, because its idea is similar to that of Regularized Regression but it is easier to explain because there are some more advanced mathematics involved in the latter technique. Consider our setting as mentioned above, a set of datapoints which have been mapped into a higher dimension so that they are linearly separable. Then we want to find a real valued linear function

$$g(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle = \mathbf{w}'\mathbf{x} = \sum_{i=1}^{n} (w_i \cdot x_i), \tag{1.1}$$

that best describes the relationship between the features and the class label in a given training set S=$\{(\mathbf{x}_1,y_1),...,(\mathbf{x}_l,y_l)\}$ of the feature vectors $\mathbf{x}_i$ from X $\subseteq \mathbb{R}^n$ with their corresponding labels y_i in Y $\subseteq \mathbb{R}$. The notation $\mathbf{x} = (x_1, x_2, ..., x_n)$ is used to denote the input vectors which are the vectors containing the features and their values, and $\mathbf{w}'$ is used to denote the transpose of the vector $\mathbf{w} \in \mathbb{R}^n$. Now that we have this structure we can define a pattern function that matches the predicted value $g(\mathbf{x})$ to the actual label y, which should be approximately equal to zero.

$$f((\mathbf{x}, y)) = \mid y - g(\mathbf{x}) \mid = \mid y - \langle \mathbf{w}, \mathbf{x} \rangle \mid \approx 0 \tag{1.2}$$

This is also referred to as linear interpolation, that is, fitting a hyperplane through the given n-dimensional points, which correspond with the n features of the datapoint. To be more exact, if the data that has been generated

is of the form $(\mathbf{x}, g(\mathbf{x}))$ where $g(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle$ and there are exactly $l = n$ linearly independent points, the parameters of $\mathbf{w}$ can be found by solving the equation

$$\mathbf{X}\mathbf{w} = \mathbf{y}, \tag{1.3}$$

where $\mathbf{X}$ is used to denote the matrix which rows are the transposed input vectors $(\mathbf{x}_1{}', ...\mathbf{x}_l{}')$ and $\mathbf{y}$ to denote the vector $(y_1, ..., y_l)'$ which contains the labels for each datapoint. After having said this, we can translate our error function

$$f((\mathbf{x}, y)) =\mid y - g(\mathbf{x}) \mid=\mid \xi \mid, \tag{1.4}$$

that gives the error for one particular training example, into a function where we take all training errors for that set into account. As the name of the algorithm says, the loss of a prediction is measured with the squared error of the error function, which results in the following definition

$$\mathcal{L}(g, S) = \mathcal{L}(\mathbf{w}, S) = \sum_{i=1}^{l} (y_i - g(\mathbf{x}_i))^2 = \sum_{i=1}^{l} {\xi_i}^2 = \sum_{i=i}^{l} \mathcal{L}((\mathbf{x}_i, y_i), g)), \tag{1.5}$$

where $\mathcal{L}((\mathbf{x}_i, y_i), g) = \xi^2$ is used to denote the squared error of the prediction function g for one example $(\mathbf{x}_i, y_i)$ and $\mathcal{L}(f, S)$ to denote the summed loss of the error function on the training set S. Now it becomes clear that the solution to the problem is to choose a weight vector $\mathbf{w} \in W$ that minimises the total loss of the function. If we rewrite the error function using the other notation we get the following equation

$$\boldsymbol{\xi} = \mathbf{X}\mathbf{w} - \mathbf{y}, \tag{1.6}$$

so that we can rewrite the loss function as

$$\mathcal{L}(\mathbf{w}, S) =\mid\mid \boldsymbol{\xi} \mid\mid^2 = (\mathbf{y} - \mathbf{X}\mathbf{w})'(\mathbf{y} - \mathbf{X}\mathbf{w}), \tag{1.7}$$

where $\mathbf{X}'$ denotes the transpose of $\mathbf{X}$. Now we can seek the $\mathbf{w}$ that minimises the loss by taking the derivative of this function with respect to $\mathbf{w}$ and setting it equal to the zero vector

$$\frac{\delta \mathcal{L}(\mathbf{w}, S)}{\delta \mathbf{w}} = -2\mathbf{X}'\mathbf{I}(\mathbf{y} - \mathbf{X}\mathbf{w}) = -2\mathbf{X}'\mathbf{y} + 2\mathbf{X}'\mathbf{X}\mathbf{w} = 0, \tag{1.8}$$

as we can see from equation (76) in [19]:

$$\frac{\delta}{\delta s}(\mathbf{x} - \mathbf{A}\mathbf{s})'\mathbf{W}(\mathbf{x} - \mathbf{A}\mathbf{s}) = -2\mathbf{A}'\mathbf{W}(\mathbf{x} - \mathbf{A}\mathbf{s}), \tag{1.9}$$

so that we, after simplifying, get the equation

$$\mathbf{X}'\mathbf{X}\mathbf{w} = \mathbf{X}'\mathbf{y}, \tag{1.10}$$

and if the inverse of $\mathbf{X}'\mathbf{X}$ exists we can find the solution to the minimizing of the loss by solving the following equation

$$\mathbf{w} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}. \tag{1.11}$$

One large downside of this approach, which is also called primal regression, is that solving this equation requires the computation of the inverse of a $n \times n$ matrix, that is features $\times$ features, which will be very inefficient when there are more features then datapoints. This is because $\mathbf{X}$ is of dimensions $l \times n$, which makes $\mathbf{X}'$ of dimensions $n \times l$ so that $\mathbf{X}'\mathbf{X}$ is of dimensions $n \times n$. This results in a complexity of $O(n^3)$ which means that the number of operations $t(n)$ is bounded by $t(n) \leq Cn^3$ for some constant C.

Now that the optimal $\mathbf{w}$ is known, the actual predictions can now be done by using the prediction function as mentioned above

$$g(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle . \tag{1.12}$$

To end this first section we take a look into a way how to reduce the computational complexity by rewriting equation (1.11) so that it becomes a so called dual regression problem, which is done in the following way

$$\begin{aligned} \mathbf{w} &= (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} \\ &= \mathbf{X}'\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} \\ &= \mathbf{X}'\mathbf{X}(\mathbf{X}'\mathbf{X})^{-2}\mathbf{X}'\mathbf{y} \\ &= \mathbf{X}'\boldsymbol{\alpha}, \end{aligned}$$

so that we obtain a linear combination of the training examples ($l \times l$)

$$\mathbf{w} = \sum_{i=1}^{l} \alpha_i \cdot \mathbf{x}_i. \tag{1.13}$$

Now that the principle of Regression is known, we can proceed to the next section where this algorithm will be expanded to Regularized Regression, where we add an extra parameter to restrict the choice of functions when computing a solution to the problem. The reason why we do this is because there usually isn't an exact relationship between the features and the classes, but an approximate one. To make sure that the algorithm doesn't try to match every datapoint exactly we introduce a parameter to control the number of errors on the training- vs. the number of errors on the test-set. If we optimize this parameter we get a more appropriate approximation of the relationship, by creating a smoother function that doesn't overfit the data. Overfitting means that the function tries to match each datapoint exactly, which results in a prediction function with not enough generalisation power to predict accurately.

1.1.2 Regularized Regression

As mentioned in the last part of the previous section, we introduce another parameter to restrict the functions being chosen as solutions for the problem, the so called regularisation parameter. After introducing this parameter the optimization problem that we need to solve for Regularized Regression is

$$\min_w \mathcal{L}_\lambda(\mathbf{w}, S) = \min_w \lambda|| \mathbf{w} ||^2 + \sum_{i=1}^{l}(y_i - g(\mathbf{x}_i))^2, \tag{1.14}$$

where λ is a positive r $\in \mathbb{R}$ that defines the relative trade-off between the loss and the norm and thus controls the degree of regularisation, so that the problem is reduced to an optimization problem over $\mathbb{R}^n$. The loss of the function is the measure described in the previous section, the summed squared error that the loss function returns. The (Euclidean) norm of a vector $\mathbf{z}$ is also known as

$$|| \mathbf{z} ||= \sqrt{\mathbf{z} \cdot \mathbf{z}} = \sqrt{{z_1}^2 + {z_2}^2 + ... + {z_n}^2}. \tag{1.15}$$

The loss function can be rewritten so that the relationship between Linear and Regularized Regression becomes more apparent

$$\begin{aligned} \mathcal{L}(\mathbf{w}, S) &= (\min_w \lambda|| \mathbf{w} ||^2 + \sum_{i=1}^{l}(y_i - g(\mathbf{x}_i))^2 \\ &= (\mathbf{y} - \mathbf{X}\mathbf{w})'(\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda|| \mathbf{w} ||^2 \\ &= (\mathbf{y} - \mathbf{X}\mathbf{w})'(\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda \mathbf{w}'\mathbf{w}, \end{aligned}$$

where $|| \mathbf{w} ||^2$ is equal to $\mathbf{w}'\mathbf{w}$ because of

$$|| \mathbf{z} ||^2 = \sqrt{\mathbf{z} \cdot \mathbf{z}} \cdot \sqrt{\mathbf{z} \cdot \mathbf{z}} = \langle \mathbf{z}, \mathbf{z} \rangle = \mathbf{z}'\mathbf{z}, \tag{1.16}$$

so that the derivative of $\lambda \mathbf{w}'\mathbf{w}$ is $\lambda \mathbf{w}$. Now that we know what we need to optimize, the reasoning is parallel to the one of the Linear Regression. First we need to take the derivatives of the function that we need to optimize, according to the same equation (76) in [19]:

$$\frac{\delta}{\delta s}(\mathbf{x} - \mathbf{A}\mathbf{s})'\mathbf{W}(\mathbf{x} - \mathbf{A}\mathbf{s}) = -2\mathbf{A}'\mathbf{W}(\mathbf{x} - \mathbf{A}\mathbf{s}), \tag{1.17}$$

which results in the equations

$$\mathbf{X}'\mathbf{X}\mathbf{w} + \lambda \mathbf{w} = (\mathbf{X}'\mathbf{X} + \lambda \mathbf{I}_n)\mathbf{w} = \mathbf{X}'\mathbf{y}, \tag{1.18}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix. Because $\mathbf{X}'\mathbf{X} + \lambda \mathbf{I}_n$ is always invertible if $\lambda > 0$ the solution to this problem can be calculated by solving the equation

(1.19), because the addition of $\lambda\mathbf{I}$ ensures that the matrix is not singular. If a square matrix is singular, which means that it has a determinant that is equal to zero, it is not invertible.

$$\mathbf{w} = (\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}'\mathbf{y} = \mathbf{X}\mathbf{y}\prime(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})^{-1} \tag{1.19}$$

So that the prediction function is defined by

$$g(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle = \mathbf{X}\mathbf{y}\prime(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{x}. \tag{1.20}$$

Notice that this equation again needs to invert a $n \times n$ matrix, similar to the primal solution of the Linear Regression problem. But again we can rewrite it to make sure that the inversion only needs to be done on a $l \times l$ matrix. In order to do this we must rewrite equation (1.15) in terms of $\mathbf{w}$ so that we obtain

$$\mathbf{w} = \lambda^{-1}\mathbf{X}'(\mathbf{y} - \mathbf{X}\mathbf{w}) = \mathbf{X}'\boldsymbol{\alpha}, \tag{1.21}$$

so that we again get the weightvector $\mathbf{w}$ in terms of a linear combination of the training examples $(l \times l)$

$$\mathbf{w} = \sum_{i=1}^{l} \alpha_i \cdot \mathbf{x}_i, \tag{1.22}$$

with $\boldsymbol{\alpha} = \lambda^{-1}(\mathbf{y} = \mathbf{X}\mathbf{w})$. We can then rewrite this equation as following

$$\begin{aligned}
\boldsymbol{\alpha} &= \lambda^{-1}(\mathbf{y} - \mathbf{X}\mathbf{w}) && (1.23)\\
\lambda\boldsymbol{\alpha} &= \mathbf{y} - \mathbf{X}\mathbf{X}'\boldsymbol{\alpha} && (1.24)\\
(\mathbf{X}\mathbf{X}' + \lambda\mathbf{I}_l)\boldsymbol{\alpha} &= \mathbf{y} && (1.25)\\
\boldsymbol{\alpha} &= (\mathbf{X}\mathbf{X}' + \lambda\mathbf{I}_l)^{-1}\mathbf{y} && (1.26)\\
\boldsymbol{\alpha} &= (\mathbf{G} + \lambda\mathbf{I}_l)^{-1}\mathbf{y}, && (1.27)
\end{aligned}$$

which results in the following prediction function

$$g(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle = \left\langle \sum_{i=1}^{l} \alpha_i \cdot \mathbf{x}_i, \mathbf{x} \right\rangle = \sum_{i=1}^{l} \alpha_i \langle \mathbf{x}_i, \mathbf{x} \rangle = \mathbf{y}\prime(\mathbf{G} + \lambda\mathbf{I}_n)^{-1}\mathbf{k}, \tag{1.28}$$

where $k_i = \langle \mathbf{x}_i, \mathbf{x} \rangle$. It can be seen that the matrix $\mathbf{X}\mathbf{X}'$ is denoted as $\mathbf{G}$, which is also known as the Gram matrix. Both the Gram matrix and the matrix $(\mathbf{G}+\lambda\mathbf{I}_l)$ are of dimensions $l \times l$ so that the computational complexity of the computing of its inverse will be a lot less than when computing the inverse of $(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I}_n)$ which has dimensions $(n \times n)$. This is because the number of examples is limited most of the time, but the number of features can be of that vast numbers that computing it with the primal method is a

futile job. There's one disadvantage, which is that the prediction of a new example will always be more computationally expensive ($O(nl)$) versus the $O(n)$ of the primal solution. Nevertheless it shows that the dual solution can offer huge advantages because of the reduction of required computations to calculate the vector $\boldsymbol{\alpha}$.

1.2 SVMLight

The next algorithm is very similar to ours, with only small differences, which we note when we encounter them. Again we consider a dataset with datapoints which have a set of features and a class label. One important difference to note is that this implementation solves a classification problem instead of a regression problem, which means that the predicted value is either 1 or -1 instead of being a real value with our implementation. Suppose [4] there is some hyperplane that divides the positive and negative examples:

$$\forall_i y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0, \tag{1.29}$$

where $\mathbf{w}$ is the normal to the hyperplane and $| b | / || \mathbf{w} ||$ is the perpendicular distance from the origin to the hyperplane, with $|| \mathbf{w} ||$ the Euclidean norm of w (as mentioned in the last section). Next we define the margin $margin = d^- + d^+$ where $d^+(d^-)$ stands for the distance from the closest positive(negative) example to the hyperplane separating the two classes of datapoints. We can now formulate the problem as looking for the hyperplane that has the biggest margin, so that our distance between possible predictions is as big as possible. To get a better picture of the mentioned parameters consider Figure **1.21**.

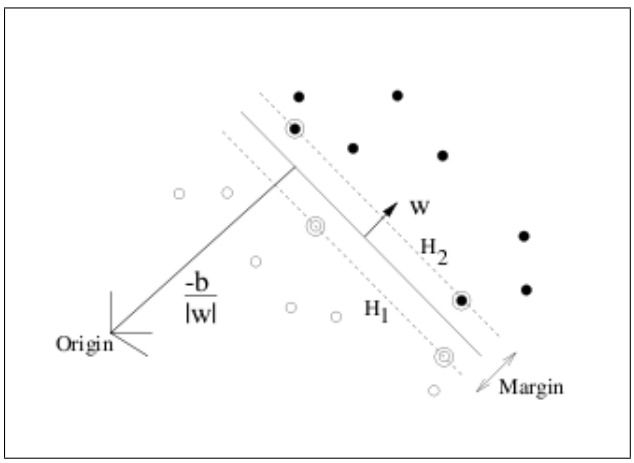


Figure **1.21** Principle of the Support Vector Machine

Now we continue to the actual definition of the algorithm. Suppose we assume that all datapoints in the training set satisfy the following equations:

$$\mathbf{x} \cdot \mathbf{w} + b \geq +1 \quad for \quad y_i = +1 \tag{1.30}$$
$$\mathbf{x} \cdot \mathbf{w} + b \geq -1 \quad for \quad y_i = -1, \tag{1.31}$$

which we can combine into the following inequality

$$\forall_i y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0, \tag{1.32}$$

where we can intuitively see that if the prediction $\mathbf{x}_i \cdot \mathbf{w} + b$ is equal to the actual class label y_i the the inequality holds, namely $(-1 \cdot -1) - 1 = (1 \cdot 1) - 1 = 0 \geq 0$. The solution to the problem now again lies in minimizing the loss with respect to the $\mathbf{w}$ and b parameters so that the margin is maximized. We can formulate this as:

$$\min_{f \in H} \frac{1}{l} \sum_{i-1}^{l} (1 - y_i f(\mathbf{x}_i))_+ + \lambda || \ f \ ||_K^2, \tag{1.33}$$

where $(k)_+ = max(k, 0)$ and $\lambda || \ f \ ||_K^2$ is the regularization parameter. If we compare this formula with the one of the Dual RLS solution we can see that they both use a similar formulation:

$$\min_{f \in H} \frac{1}{l} \sum_{i-1}^{l} V(y_i, f(\mathbf{x}_i)) + \lambda || \ f \ ||_K^2, \tag{1.34}$$

where $V(y_i, f(\mathbf{x}_i))$ is the function that is used to calculate the loss, which is the squared error in RLS ($(y - g(\mathbf{x}_i))^2$) and $(1 - y_i f(\mathbf{x}_i))_+$ in this case. This kind of problem is called a convex quadratic programming problem, of which a detailed description is not considered in this thesis. For a detailed description we refer to [4].

1.3 RLS - Multiclass

So far in the thesis we discussed binary classification problems. However, frequently the problem at hand requires considering more than two-class classification - e.g. the case when the label y is chosen from the set $\mathcal{Y}$ of cardinality $k > 2$. Several methods have been proposed to address this problem. Most of these methods aim to reduce the multi-class classification problem into a problem that deals with binary classes and then in some way combines the obtained results. For instance, we can create a binary classification problem for each of the k classes. In this situation we consider the examples $y = l_1$ to belong to a positive class and all other examples having

class labels $l_{2,\ldots,k}$ belonging to the negative class. This type of approach is usually called one-versus-all [20].

There are other possibilities to deal with the multi-class learning problem, for example the all-pairs approach. In this case [11] we consider all possible pairs of classes $l_1, l_2 \in \mathcal{Y}$. We run the algorithm for these two classes considering them as positive and negative ones, respectively. This means that $\binom{k}{2}$ hypothesis have to be generated and combined. Tibshirani calls this the all-pairs-approach.

Finally there are more general suggestions how to treat multi-class classification problems. Initially proposed in [7] and later extended by Allwein [1] the approach is know as ECOC - error correcting output codes. The main idea behind this method is to construct a coding matrix $\mathbf{M} \in \{-1, +1\}^{k \times c}$, where c is some positive integer and where every row of the coding matrix is associated with a single class $l \in \mathcal{Y}$. The binary learning algorithm is then run once for each column of the matrix on the induced binary problem in which the label of each example labeled y is mapped to $M(y, s)$. This gives a number of prediction functions for each column of the matrix $\mathbf{M}$, namely f_c. Now, given an example x, we then predict the label y for which row of the matrix is closest to $(f_1(x) \ldots f_c(x))$.

Allwein proposed a generalization of this approach, by allowing the coding matrix to contain also 0 elements in addition to $-1, +1$. By doing so they suggest that some entries in the coding matrix constructed on all data points $M(y, s)$ may be zero, indicating that we are not interested how the prediction function f_s categorizes examples with label l. For every $s = 1, \ldots l$ the algorithm $\mathcal{A}$ has labeled data in the form $(x_i, M(y_i, s))$ for all examples in the training set and omitting all examples where $M(y_i, s) = 0$. The algorithm $\mathcal{A}$ uses this information to generate the prediction functions $f_s : \mathcal{X} \to \mathbb{R}$.

We also introduce the loss function similar to Allwein [1] so that f_s on example x_i with the label $M(y_i, s) \in \{-1, +1\}$ is $L(M(y_i, s), f_s(x))$. When $M(y_i, s) = 0$ we simply want to ignore the appropriate prediction function, and for convenience can select the loss to be equal to 0. The average loss for all of the training examples can be written as follows:

$$\frac{1}{nl} \sum_{i=1}^{n} \sum_{s=1}^{l} L(M(y_i, s), f_s(x))$$

Let us by $\mathbf{M}(r)$ denote the row of the coding matrix M constructed on the training data points. Also we denote the prediction vector of the example x as follows: $\mathbf{f}(x) = (f_1(x) \ldots f_l(x))$. The question to be raised is given the predictions for a single example how can we determine the appropriate class label? To answer this one can possibly come up with several methods, however, here we follow Allwein who suggested to use so called loss-based decoding. The basic approach here is to find which row corresponding to

$\mathbf{M}(r)$ is "closest" to the prediction vector $\mathbf{f}(x)$. Formally we need to find the label which minimizes the distance $d(\mathbf{M}(r), \mathbf{f}(x))$. One way of doing this is to compute the Hamming distance between these two vectors:

$$d_H(\mathbf{M}(r), \mathbf{f}(x)) = \sum_{s=1}^{l} \frac{(1 - sign(M(r), s), f_s(x)))}{2},$$

where *sign* refers to the function $sign(k)$, which returns -1 when $k < 0$, 0 when $k = 0$ and $+1$ when $k > 0$. This would correspond to the approach proposed in [7]. However one can possibly "improve" distance metric, this by also taking into account the magnitudes of the differences between the predicted labels and the appropriate rows of the coding matrix. Formally we can write

$$d_L(\mathbf{M}(r), \mathbf{f}(x)) = \sum_{s=1}^{l} L(M(M(r), s), f_s(x)).$$

By using this approach together with joint kernel maps to deal with multi-class classification problems we are able to show notably better performance of our algorithm compared to several baseline methods.

1.3.1 Joint Kernel

In this section we briefly describe the joint kernel used in this study. We intend to describe the algorithm and the approach in details in a separate publication which is currently under preparation.

Joint kernel is a nonlinear similarity measure between input-output pairs, i.e., $J((x, y), (x', y'))$ where (x, y) and (x', y') are labeled training examples. We can write:

$$J((x, y), (x', y')) = \langle \Phi_{\mathcal{XY}}(x, y) \Phi_{\mathcal{XY}}(x', y') \rangle,$$

where $\Phi_{\mathcal{YX}}$ is the map into dot product space. The function taking form of the kernel as described above is positive definite and can be written as a dot product.

As it can be seen in the definition above, the main idea behind the joint kernel is to describe the similarity between input-output pairs by mapping pairs into a joint space. A joint kernel can encode more than just information about inputs or outputs independent of each other: it can also encode known dependencies/correlations between inputs and outputs. Joint Kernels have been already studied and several variations are proposed in [25]. In our algorithm the joint kernels are used to construct the joint similarity space between the input data point features and their multi-class encoding. One of the kernels we use in our experiments is the following modification of Gaussian:

$$J_{RBF}((x,y),(x',y')) = exp\frac{-||(x,y)-(x',y')||^2}{\sigma^2}.$$

One can notice that we could define two separate similarity measures (kernels) on inputs and the outputs (in this case the encoded labels of the data points). However, it is more difficult for the algorithm to infer which data points are related when providing this information separately. Also we suggest several ways how to further improve performance of the method by tweaking kernel function such that prior knowledge about particular dataset is incorporated into the learning problem.

1.4 Naive Bayes

In this section we take a look into the Naive Bayes algorithm, as also described in [17]. It's a technique which uses probability distributions to estimate the class of a datapoint. It does this by searching for the class for which, given a set of attributes, the probability is highest. To make this more concrete we can consider the following problem setting. Assume there is a set of datapoints which each have a class label $\mathbf{Y}$, which is also known as the output variable, and a set of attributes $\mathbf{X}_1, ..., \mathbf{X}_n$, the input variables, which we assume to be conditionally independent from each other given $\mathbf{Y}$. This means that if we have the conditional distribution $\mathbf{P}(a \mid b, c)$ and a is conditionally independent from b given c we can rewrite it to $\mathbf{P}(a|c)$, because a does not depend on the occurrence of b. The reason why we assume this is that it dramatically reduces the computational complexity by enabling the possibility to write the distribution as a summation of the distributions of its components. What this actually means will become clear later on this page. To continue our description we assume a dependency of the class label $\mathbf{Y}$ on its attributes $\mathbf{X}_1, ..., \mathbf{X}_n$ following the theory of probability estimation

$$\mathbf{P}(\mathbf{Y} \mid \mathbf{X}). \tag{1.35}$$

Finding the optimal probability and thus the correct class label can then be formulated as

$$\begin{aligned}
\mathbf{Y}^{predict} &= argmax_y \mathbf{P}(\mathbf{Y} \mid \mathbf{X}) \\
&= argmax_y \mathbf{P}(\mathbf{Y} = y \mid \mathbf{X}_1 = x_1, ..., \mathbf{X}_n = x_n) \\
&= argmax_y \frac{\mathbf{P}(\mathbf{X}_1 = x_1, ..., \mathbf{X}_n = x_n \mid \mathbf{Y} = y)\mathbf{P}(\mathbf{Y} = y)}{\mathbf{P}(\mathbf{X}_1 = x_1, ..., \mathbf{X}_n = x_n)} \\
&= argmax_y \mathbf{P}(\mathbf{Y} = y)\mathbf{P}(\mathbf{X}_1 = x_1, ..., \mathbf{X}_n = x_n \mid \mathbf{Y} = y) \\
&= argmax_y \mathbf{P}(\mathbf{Y} = y) \sum_{i=1}^{n} \mathbf{P}(\mathbf{X}_i = x_i \mid \mathbf{Y} = y).
\end{aligned}$$

The third step is done according to Bayes Rule ($\mathbf{P}(A \mid B) = \frac{\mathbf{P}(B|A)\mathbf{P}(A)}{\mathbf{P}(B)}$), the fourth step is possible because the denominator($\mathbf{P}(\mathbf{X}_1 = x_1, ..., \mathbf{X}_n = x_n)$) does not depend on $\mathbf{Y}$. The last step is the one where the conditional independence makes it possible to rewrite the problem to reduce it's computational complexity, as mentioned above.

1.5 K Star

The next algorithm that we examine is K Star, or $\mathbf{K}^*$ for short, as explained in [6], which uses entropy to calculate the similarity between two datapoints. This intuitively means computing the complexity to transform one datapoint into the other. A downside is that entropy only considers the shortest distance of all possible transformations between two datapoints, which results in a distance that is very sensitive to small changes in that instance space. To attack this problem $\mathbf{K}^*$ considers a total distance of all transformations. Having said that we come to the following definition of the algorithm:

Consider a set $\mathbf{D}$ of (possibly infinite) datapoints and $\mathbf{T}$ a finite set of transformations on $\mathbf{D}$. This means that each $t \in T$ maps datapoints to datapoints $t : \mathbf{D} : \mathbf{D}$. Also, $\mathbf{T}$ contains a so called stop symbol ($\boldsymbol{\sigma}$) which maps instances to themselves ($\boldsymbol{\sigma}(a) = a$). Let $\mathbf{C}$ then be the set of all prefix codes from $\mathbf{T}^*$ which are terminated by $\boldsymbol{\sigma}$. Now all members of $\mathbf{T}^*$ and $\mathbf{C}$ uniquely define the transformation

$$\mathbf{t}(a) = t_n(t_{n-1}(...(t_1(a))), \tag{1.36}$$

where $\mathbf{t} = t_1, ..., t_n$ on $\mathbf{D}$. Next we define a probability function $\mathbf{p}$ on $\mathbf{T}^*$ which satisfies

$$0 \leq \frac{\mathbf{p}(\mathbf{t}u)}{\mathbf{p}(\mathbf{t})} \tag{1.37}$$

$$\sum_u \mathbf{p}(\mathbf{t}u) = \mathbf{p}(\mathbf{t}) \tag{1.38}$$

$$\mathbf{p}(\Lambda) = 1, \tag{1.39}$$

so that it satisfies

$$\sum_{t \in \mathbf{C}} \mathbf{p}(\mathbf{t}) = 1. \tag{1.40}$$

Next we can define the probability function $\mathbf{P}^*$ as the probability from all paths from datapoint a to datapoint b as

$$\mathbf{P}^*(b \mid a) = \sum_{\mathbf{t} \in \mathbf{C} : \mathbf{t}(a) = b} \mathbf{p}(\mathbf{t}). \tag{1.41}$$

And because we can prove that $\mathbf{P}^*$ satisfies the properties

$$\sum_b \mathbf{P}^*(b \mid a) = 1 \tag{1.42}$$

$$0 \leq \mathbf{P}^*(b \mid a) \leq 1, \tag{1.43}$$

we can define $\mathbf{K}^*$ as

$$\mathbf{K}^*(b \mid a) = -\log_2 \mathbf{P}^*(b \mid a). \tag{1.44}$$

1.6 Random Forest

To finalize the descriptions of the used algorithms, the last one that we consider is the one called Random Forest. Because this is an algorithm that consists of a set of so called Decision Trees, we examine those first and then get to the extension of that method, the Random Forest. As described in[17], Decision Trees use a tree structure with nodes that have a test for one of the attributes of a datapoint at each non-terminal node. The branches that come from each node represent an answer to that test. For each datapoint each attribute is checked and the associated branch is chosen until it reaches a terminal node, which is also referred to as a "leaf". That leaf decides to which class the datapoint belongs. To construct this structure, we simply use a lot of datapoints from which the class labels are known and save the values of the attributes as a path in the tree leading to the leaf with that class label. As can be expected, these attributes can be of different types, from Boolean values to ranges in real values. To make the idea more apparent we consider an example as also written in [17]. Figure **1.61** shows a trained Decision Tree for a possible solution to the problem when the weather conditions are suitable to play tennis.

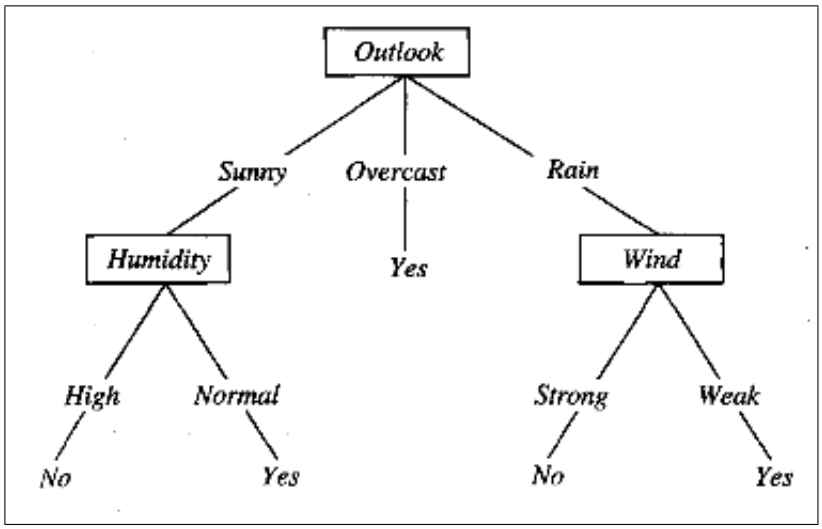


Figure **1.61** Trained Decision Tree

As you can see there are three nodes: Outlook, humidity and wind, with two answers each, yes or no, which means that this is a binary Decision Tree. If the attributes of the datapoint, or in this case, the weather condition (I.e. the set of attributes {Outlook, Humidity, Wind}) are checked and the corresponding answers are chosen it will reach the class of that datapoint, or in this case, the decision whether or not the weather is suitable to play tennis.

Now that the idea of the Decision Tree is clear we can continue to the Random forest, which uses exactly the same technique of deciding whether a datapoint belongs to a certain class or not. Like described in [3] it now simply constructs an arbitrary number of Decision Trees. It does this by generating a random vector $\boldsymbol{\theta}_k$ for each of the k trees. This vector is independent of the past generated random vectors $\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_{k-1}$ but it has the same distribution. This means that all random generated vectors are in a vector space which has some defined distribution (normal,Gaussian,etc.). Now a tree is constructed with the training set and this random vector θ_k, so that we obtain a number of trees that all used a different vector to be constructed, hence the name Random Forest. After we trained the tree we can use the classifier $h(\mathbf{x}, \boldsymbol{\theta}_k)$ for each k, to classify the unknown datapoint, also referred to as the input vector $\mathbf{x}$. When we do this for each of the k trees we can simply count votes and see which class label y is predicted most frequently for that datapoint $\mathbf{x}$.

Chapter 2

Research

In this chapter we discuss the performed experiments and the associated results, as well as the datasets being used. Because we compare the performances of several algorithms on different datasets, this chapter is structured according to those comparisons. Also there is a division between the actual algorithms and the results of the performed experiments. First we describe our algorithm and the SCOP (Structural Classification of Proteins) dataset[18]. Second, we apply the algorithm to several datasets from the LIBSVM website[5], so that we can compare our results with those of the other algorithms. In the third section we describe the Python implementation of the wrapper for the SVMLight algorithm on the same datasets. To finish the first part of this chapter we describe the algorithms from the WEKA[10] program. The second part of this chapter consists of the obtained results and the comparison between these results.

2.1 Experiments

As mentioned above, the first section of this chapter is about the experiments, datasets and implementations which this thesis is based upon. We use the programming language Python to implement the Regularized Least Squares algorithm in two different ways, the first one is for the SCOP dataset and the second to make it possible to run it on some datasets from the LIBSVM website. We also implement a Python script to run a ten fold version of the SVMLight algorithm for a fair comparison. For the WEKA we choose not to create a script because of the need for yet another programming language called Jython.

2.1.1 RLS on SCOP

The SCOP [18] database contains a detailed description of the structural and evolutionary relationships between all proteins of which the structure is known. Because almost all proteins have structural similarities with other proteins and sometimes share a common evolutionary origin, this is an interesting dataset to consider when we want to analyze relationships between the class label and the features of a set of datapoints. Furthermore, the results of these experiments can contribute to a better understanding of the evolution of proteins. Because not all relationships can be identified automatically, the SCOP is constructed manually by visual inspection and comparison of structures. There are many levels in the hierarchy, but the principal levels are family, superfamily and fold, as described below. The exact position of boundaries between these levels is to some degree subjective, but the SCOP evolutionary classification is generally conservative: where any doubt about relatedness exists, make new divisions at the family and superfamily levels.

- Family : proteins are clearly evolutionarily related
- Superfamily: Probable common evolutionary origin
- Fold: Major structural similarity

As described in [15], the dataset consists of 54 families which each have at least 10 family members (positive test examples) and 5 superfamily members outside of the family (positive train examples). Negative examples are taken from outside of the positive examples and are split into the training and test sets in the same ratio as the positive examples. A more detailed description of the SCOP dataset and the families is written in the second section of this chapter, where we discuss the results.

We now describe our implementation of the Regularized Least Squares algorithm. We start with the algorithm described in [24] and expand it so it uses cross validation to find a prediction function that isn't biased on the test set, which means that it uses information from the test set to optimize its parameters. Cross validation means that we divide the training set into two parts, a cross validation- test and training set[8]. Then we run the algorithm and save the performance measure for each chosen lambda. We repeat this for the number of folds being used and average the performance for each lambda over the number of folds. This intuitively means adding all performance measures for each lambda together and dividing them by the number of folds. Then we can use that lambda that has the highest performance measure to create the real model, i.e. training the prediction function with that one lambda on the full training set and evaluating the real performance on the real test set. To find out which number of folds works best on this kind of problem we try 5 folds, 10 folds and a pseudo-30 folds. The latter is a technique where we use 10 fold cross validation three times, and shuffle the training set before we start one of each three runs. The reason that we do this is because to calculate our performance measure (AUC) we need both positive and negative class members in the test set, and some of the SCOP families do not have enough members(≤ 11) in their training set to ensure this is the fact when we use more than 10 folds. This Area Under Curve performance measure, or AUC for short, as described in[24], is a measure that represents true positives as a function of false positives for varying classification thresholds. When the classification is perfect, the AUC will have a value of 1, if the classification is done random, the value of the AUC measure will be 0,5. There is a more extended description in the last part of this section. On the SCOP dataset we choose to use a linear kernel because of the need of optimizing yet another parameter σ, which is explained in the next section, where we use a Gaussian Kernel. The reason why we choose to do this is that the SCOP database is of that vast size that adding this extra optimization parameter results in increasing the computational complexity by a notable amount, making computation times even longer.

2.1.2 RLS on other datasets

Because of the size of the SCOP dataset, comparing our algorithm with other machine learning algorithms using this dataset would be a time consuming job. For this reason we choose to adapt our implementation so that it can use datasets from the LIBSVM website [5], so that we have access to a variety of datasets. Because of the possibility to choose smaller datasets, we can optimize our algorithm even more by using the extra optimization parameter σ as mentioned in the last section. The use of this Gaussian Kernel makes

the algorithm more able to cope with non-linearly separable problems. The difference between the linear kernel and the Gaussian kernel is that the Gaussian doesn't use the dot product $\mathbf{K}(\mathbf{x}, \mathbf{x}\prime) = \langle x, x\prime \rangle$ to compute the kernel matrix, but the equation

$$\mathbf{K}(\mathbf{x}, \mathbf{x}\prime) = e^{-\frac{(\mathbf{x}-\mathbf{x}\prime)^2}{\sigma^2}}, \tag{2.1}$$

where σ is the new parameter that controls the width of the function. To make this more apparent we consider the general Gaussian function

$$\mathbf{K}(\mathbf{x}, \mathbf{x}') = a \cdot e^{-\frac{(\mathbf{x}-\mathbf{b})^2}{2c^2}}, \tag{2.2}$$

which results in a characteristic symmetric "bell curve" shape that is shown in figure **2.1.1**. The parameter a is the height of the curve's peak(which is one in our case), b is the position of the centre of the peak(the real class value, $\mathbf{x}'$), and c ($\sigma/2$) controls the width of the "bell".

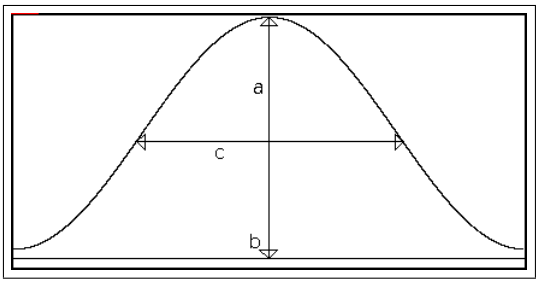


Figure **2.1.1** Gaussian curve

What this means in our situation is that $\mathbf{x}$ is the predicted value, $\mathbf{x}'$ is the real value, and as a result the similarity measure produced by this function is smaller when the value of σ is higher. This is because e^{-x} is smaller if the value of x is higher. To make a fairer comparison we choose to use several datasets, five to be exact, named australian, diabetes, fourclass, ionosphere and splice for the binary classification and six for the multiclass classification. These datasets are called DNA, iris, vehicle, vowel, wine and glass. The binary datasets are compared with all described algorithms and the multiclass datasets only with the multiclass implementation of SVMLight[13]. This is because of the good empirical performance of the SVMLight algorithm and the similarity between that algorithm and our implementation. An important note is that the AUC measure can not be used when evaluating multiclass predictions because of the form of the predictions, which we describe in the separate section about the AUC performance measure. Thus, we use another, simpler performance measure to measure the performance, called the error rate, which is simply the number of false predictions divided by the total number of predictions.

2.1.3 SVMLight

If we want to evaluate the performance of our algorithm, we need to have a similar method with which we can compare it. The one we choose, SVM-Light, is an efficient, well known algorithm, which has been shown to have good empirical performance. To make a fair comparison we need to implement the same way of evaluating the performance. To do this, we use a Python script to implement the optimizing of the parameters λ and σ, the cross validation and the Area Under Curve measure (or error rate for the multiclass) which we also use in our RLS. To make an even fairer comparison we let our algorithm create files with the same training- and test sets that it uses for it's cross validation, so that we can use exactly the same test- and training partitions for the SVMLight algorithm.

2.1.4 Bayes

WEKA is a program which contains a number of different machine learning algorithms, a full list can be found here [10]. The reason why we choose to use this program is the simplicity to use the LIBSVM datasets with different kinds of algorithms without having to implement the methods. Because of the low number of parameters on the Naive Bayes algorithm the optimizing of the algorithm is easy. The first of the two Booleans is the use of a *kernel estimation* for numeric values instead of a normal distribution. The second is the use of *supervised discretization* to convert numeric attributes to normal ones. Both are preprocessing parameters and do have an effect on the performance. These two parameters are mutually exclusive, which means that they can't both be true. So this results in three different experiments, one where both are false, one where the one is true and one where the other is true. After we conduct these experiments we can simply get the output from the program and use a script to calculate the AUC performance measure for this algorithm.

2.1.5 K*

The second algorithm we use in our experiments does have some more parameters that can be set, like the *globalBlend*, *missingMode* and *entropicAutoBlend*. The latter is a Boolean, the second is a 4 choice parameter that determines how missing attribute values are treated and the first is an integer in the range $[0, 100]$. Because of the need for another programming language to write a script to optimize the parameters, we choose not to consider every possible combination, but just optimize incrementally. This means that we first choose the best performance for the minimum choice (which is *entropicAutoBlend* or not), which results in two performance measures. When we see which gives the best performance we keep that setting

and continue optimizing the next minimum choice, which is *missing mode*. After we know the optimal choice for this parameter we continue to the optimizing of the last parameter, *globalBlend*, which we choose to do in steps of 20, because the default value of this parameter is 20. This reduction in optimization is because we conduct it manually, i.e. the entering of parameters, extracting predictions and using the same script as with Naive Bayes to compute the AUC performance.

2.1.6 Random Forest

The last algorithm that we use in the experiments is also from the WEKA program. The parameters that need to be optimized are: the *maximum depth of the trees*, the *number of features* to be used in random selection, the *number of trees* to be generated. There also is the *random number seed*, which just makes sure that one can produce the same results by entering the same random seed. Because the *maximum depth* can be unlimited and the *number of features* can be set to random (and are by default) we optimize using the *number of trees* first and then incrementally optimize the other parameters using the optimal number of trees. This means that we take a look into the effect of changing the *number of features* first and finally optimize by changing the *maximum depth*. This way we try to reach optimal performance without having to try each possible combination, which is a futile job when no script is used.

2.1.7 AUC

In this section we describe the performance measure which we use to compare the performance of the different techniques. This is because this measure has been proven to be a better performance measure than accuracy[16]. AUC[2], as mentioned above, stands for Area Under Curve. The curve that is being referred to is the Receiver Operating Characteristic curve, or ROC for short. This curve details the rate of true positives against false positives over the range of possible threshold values for a prediction function f, with the area under that curve (AUC) being the probability that a randomly chosen positive example(x^+) has a higher decision value than a randomly chosen negative example(x^-). This can also be expressed as

$$AUC(f) = P(f(x^+) > f(x^-)), \tag{2.3}$$

and refers to the true distribution of positive and negative examples. This comes down to that this comparison of decision values is done for each combination of examples. Which means that for each positive example, it is checked that it's higher than each negative example and for each negative value it's checked if it's lower than each positive example.

2.2 Results

In the last part of this chapter we describe the results that we obtained from our experiments. Similar to the last section we divide these in the way that we conduct our comparisons. First, we consider our results of the experiments on the SCOP dataset, using the different cross validation methods and the normal use (biased on the test set). Second, we compare our algorithm with the SVMLight algorithm on both the binary and multiclass classification problems and in the third section we compare our results with the other machine learning algorithms described in this thesis.

2.2.1 SCOP

Like mentioned in the last section this dataset consists of 54 families, which each are a classification problem. The detailed information about the families is described in the tables **2.2.1**a and **2.2.1**b, which can also be found in [15].

	+ TRAINING	- TRAINING	+TEST	-TEST
1.4.1.1	26	2256	23	1994
1.4.1.2	41	3557	8	693
1.4.1.3	40	3470	9	780
1.27.1.1	12	2980	6	1444
1.27.1.2	10	2408	8	1926
1.36.1.2	29	3477	7	839
1.36.1.5	10	1199	26	3117
1.41.1.2	26	3692	6	615
1.41.1.5	17	1744	25	2563
1.45.1.2	33	3650	6	663
2.1.1.1	90	3102	31	1068
2.1.1.2	99	3412	22	758
2.1.1.3	113	3895	8	275
2.1.1.4	88	3033	33	1137
2.1.1.5	94	3240	27	930
2.5.1.1	13	2345	11	1983
2.5.1.3	14	2525	10	1803
2.9.1.2	17	2370	14	1951
2.9.1.3	26	3625	5	696
2.9.1.4	21	2928	10	1393
2.28.1.1	18	1246	44	3044
2.28.1.3	56	3875	6	415
2.38.4.1	30	3682	5	613
2.38.4.3	24	2946	11	1349
2.38.4.5	26	3191	9	1104
2.44.1.2	11	3077	140	3894
2.52.1.2	12	3060	5	1275

Table **2.2.1**a SCOP Family details

	+ TRAINING	- TRAINING	+TEST	-TEST
2.56.1.2	11	2509	8	1824
3.1.8.1	19	3002	8	1263
3.1.8.3	17	2686	10	1579
3.2.1.2	37	3002	16	1297
3.2.1.3	44	3569	9	730
3.2.1.4	46	3732	7	567
3.2.1.5	46	3732	7	567
3.2.1.6	48	3894	5	405
3.2.1.7	48	3894	5	405
3.3.1.2	22	3280	7	1043
3.3.1.5	13	1938	16	2385
3.32.1.1	42	3542	9	759
3.32.1.8	40	3374	11	927
3.32.1.11	46	3880	5	421
3.32.1.13	43	3627	8	674
3.42.1.1	29	3208	10	1105
3.42.1.5	26	2876	13	1437
3.42.1.8	34	3761	5	552
7.3.5.2	12	2330	9	1746
7.3.6.1	33	3203	9	873
7.3.6.2	16	1553	26	2523
7.3.6.4	37	3591	5	485
7.3.10.1	11	423	95	3653
7.39.1.2	20	3204	7	1121
7.39.1.3	13	2083	14	2242
7.41.5.1	10	2241	9	2016
7.41.5.2	10	2241	9	2016

Table **2.2.1**b SCOP Family details

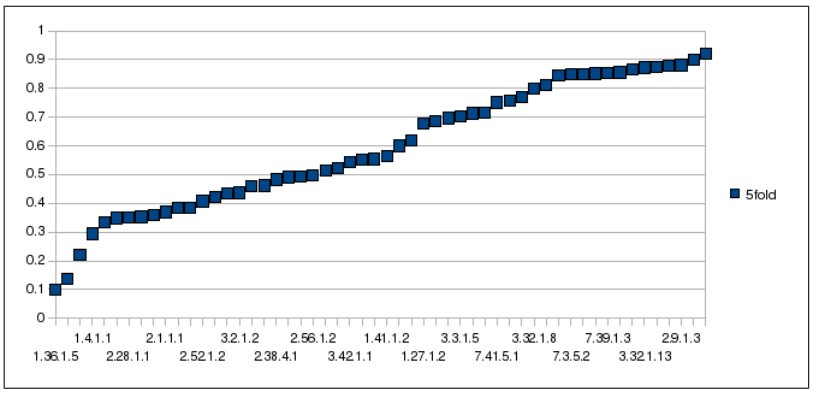
The reason we choose this dataset is that it's an extensively used one and the improvement on classifying this dataset can influence the development of better techniques for those who are active in this field of research. To achieve a fair prediction, we choose to let the machine learning algorithm be unbiased with respect to the test set. This means that the construction of the model is only dependent on the training set, of which the class labels are not known. This can result in a worse performance than when not using this technique, but it is fairer because we only consider the real relationship between the features and the class label and does not optimize with knowledge of the class label. One reason performance can significantly drop is when there is no clear link between the class and the features either due to a lot of noise in the dataset or due to the true absence of a relationship. The latter comes down to the point of looking for a relationship where it doesn't exist, which could be found if we use a biased prediction function that uses information about the class labels of the real test set to construct its model. As said in the last chapter, we implemented 30-pseudo-fold cross validation to improve performance, because we thought that the performance might stabilize when we obtain better averaged parameters by shuffling the dataset.

Unfortunately, after trying it out on a small family and seeing no improvement we dropped the idea and don't present any results about that. Instead we focused on the comparison between the biased algorithm and the five and ten fold ones. We present the results of these experiments below in table **2.2.2**. As mentioned above, a score of 0.5 means that the predictions are done randomly, and a score of 1 means the predictions are done perfectly.

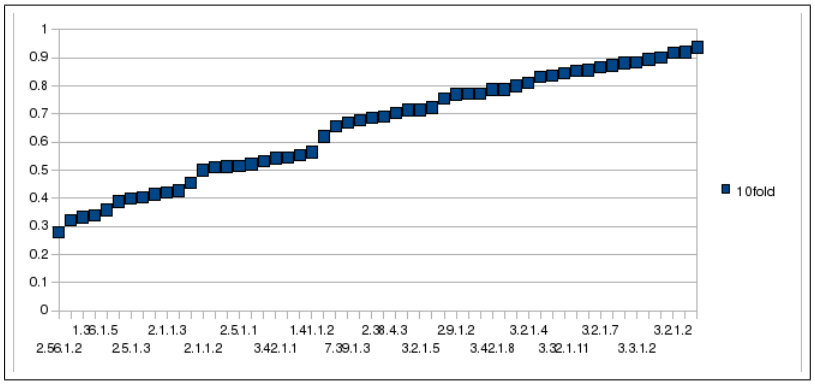
	5fold	10fold	normal		5fold	10fold	normal
1.4.1.1	0.29	0.41	0.85	**2.56.1.2**	0.5	0.28	0.83
1.4.1.2	0.36	0.36	0.95	**3.1.8.1**	0.62	0.94	0.96
1.4.1.3	0.35	0.79	0.9	**3.1.8.3**	0.92	0.92	0.89
1.27.1.1	0.55	0.55	0.89	**3.2.1.2**	0.44	0.92	0.71
1.27.1.2	0.68	0.68	0.78	**3.2.1.3**	0.71	0.71	0.66
1.36.1.2	0.33	0.32	0.87	**3.2.1.4**	0.81	0.81	0.86
1.36.1.5	0.1	0.34	0.92	**3.2.1.5**	0.72	0.72	0.79
1.41.1.2	0.57	0.57	0.9	**3.2.1.6**	0.76	0.76	0.87
1.41.1.5	0.6	0.33	0.8	**3.2.1.7**	0.87	0.87	0.95
1.45.1.2	0.88	0.83	0.92	**3.3.1.2**	0.46	0.88	0.79
2.1.1.1	0.37	0.4	0.75	**3.3.1.5**	0.7	0.7	0.67
2.1.1.2	0.49	0.5	0.93	**3.32.1.1**	0.85	0.85	0.93
2.1.1.3	0.42	0.42	0.93	**3.32.1.8**	0.8	0.8	0.71
2.1.1.4	0.46	0.51	0.83	**3.32.1.11**	0.85	0.85	0.85
2.1.1.5	0.38	0.46	0.78	**3.32.1.13**	0.87	0.87	0.73
2.5.1.1	0.51	0.51	0.81	**3.42.1.1**	0.54	0.54	0.7
2.5.1.3	0.49	0.4	0.71	**3.42.1.5**	0.44	0.72	0.83
2.9.1.2	0.77	0.77	0.83	**3.42.1.8**	0.22	0.79	0.72
2.9.1.3	0.88	0.88	0.99	**7.3.5.2**	0.85	0.86	0.93
2.9.1.4	0.14	0.84	0.94	**7.3.6.1**	0.7	0.69	0.88
2.28.1.1	0.35	0.62	0.97	**7.3.6.2**	0.85	0.77	0.9
2.28.1.3	0.39	0.53	0.7	**7.3.6.4**	0.52	0.52	0.97
2.38.4.1	0.48	0.39	0.72	**7.3.10.1**	0.85	0.89	0.86
2.38.4.3	0.35	0.69	0.73	**7.39.1.2**	0.55	0.54	0.79
2.38.4.5	0.69	0.43	0.79	**7.39.1.3**	0.85	0.67	0.83
2.44.1.2	0.88	0.51	0.87	**7.41.5.1**	0.75	0.77	0.75
2.52.1.2	0.41	0.66	0.88	**7.41.5.2**	0.9	0.9	0.81

Table **2.2.2** SCOP Results

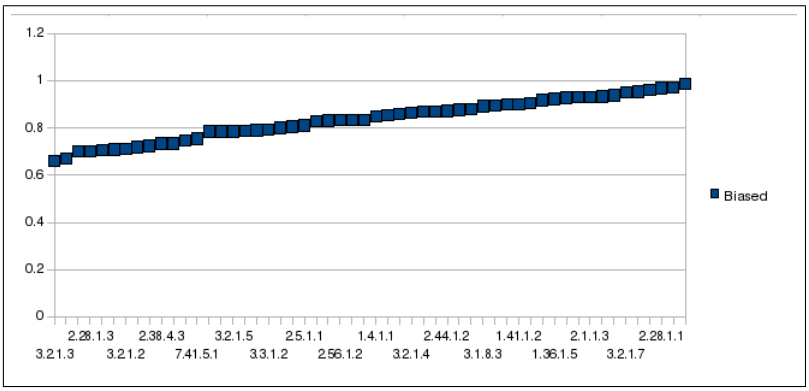
If we use a graphical representation to present the performances for each family we get the following separate plots:



SCOP 5 fold Results



SCOP 10 fold Results



SCOP biased Results

This results in a joint plot where the families are not uniquely identified but where the performance ratings are ordered in descending order to ensure a clear overview of the comparison between the different techniques, which is presented below in figure **2.2.3**.

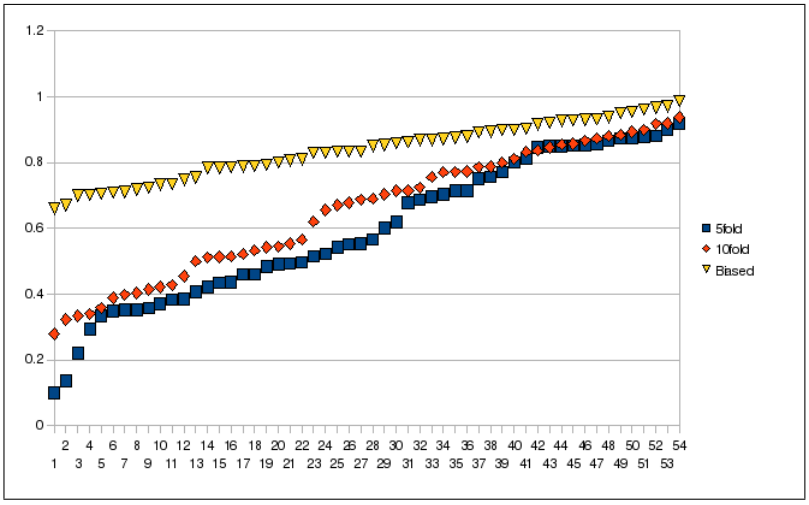


Figure **2.2.3** SCOP all Results

As we can see, the 10 fold performs significantly better than the 5 fold, but doesn't perform as well as the normal biased mode of predicting class labels. This can be accounted to the noise in that particular family or to the fact that there's no apparent relation between the features of the datapoints and their corresponding class labels. If there were enough datapoints available we could improve the performance of the cross validation approach by increasing the folds used to attain a better estimate of the prediction function. To research the true cause of this drop in performance one could use other algorithms to build a model for these families to see if it's a problem in the dataset or not. In [15], they also find an outlier family on which results are notably worse using their method (SVM-pairwise). The reason why the biased algorithm isn't affected by this is that this is optimized using the real test set, so that it recognizes relationships which aren't apparent when only considering the examples in the training set. Also, this performance could be improved by using a Gaussian Kernel, so that relationships that aren't being considered by this approach (linear kernel) are found. The reason why we did not do this, as mentioned above, is the notable increment in computational complexity, because of the need to optimize with respect to yet another parameter.

2.2.2 RLS vs. SVMLight

This section consists of two parts, one being the comparison between our binary algorithm and the SVMLight algorithm, and the other being the comparison between the proposed multiclass algorithm and the SVMMulti implementation. One parameter that isn't fair in our comparison between the multiclass algorithms is the fact that we use a Gaussian Kernel and the SVMMulti uses a linear kernel. This is because the SVMLight/SVMMulti algorithm can only use a Gaussian kernel when the problem is reformulated into multiple binary classification problems. Because of this the SVMMulti does not perform as well when considering datasets which are non-linearly separable. One could also implement a linear kernel to make the comparison more fair, but this thesis aims for an improvement in practical applicability. Because of this reason we decide to make the limitations and possibilities of each technique as clear as possible.

First we consider the binary problems where we use the Area Under Curve (AUC) to determine the performance of the two different algorithms. As mentioned before in this section, we want the value to be as close to one as possible, which means that the predictions are as precise as possible. The results of the experiments on the binary classification problems are presented in table **2.2.4**.

	RLS	SVMLight
Australian	0.9205	0.9126
Diabetes	0.8231	0.8026
Fourclass	1.0000	1.0000
Ionosphere	0.9473	0.9473
Splice	0.9490	0.9469

Table **2.2.4** AUC RLS vs. SVMLight

It can be seen that the performance of the algorithms is almost the same, with some really small variation on three of the five experiments and completely the same result on the remaining two experiments. Because of the similarity of the algorithms, we can expect to see similar results on a problem being analysed by both the Least Squares Ridge Regression and the SVMLight algorithm. These experiments confirm this expectation, so that we have a nice baseline for the comparisons with the other algorithms.

Second we consider the multi-class problems, which results in table **2.2.5**, containing the Error rates for each problem, which should be as close to zero as possible, since it is the number of false predictions divided by the total number of predictions.

	RLS	SVMMulti
DNA	0.0644	0.0811
Iris	0.0600	0.2400
Vehicle	0.3227	0.2234
Vowel	0.0341	0.3920
Wine	0.0167	0.0593
Glass	0.4	0.48

Table **2.2.5** Error rates RLS vs. SVMMulti

As you can see most of the performances of our algorithm are reasonable, except the glass and the vehicle problems. The SVMMulticlass performs notably worse on all datasets but one, which can probably be accounted to the use of a Gaussian Kernel with our method. This is because a Gaussian kernel results in an algorithm that is more able to handle with nonlinear separability in datasets. The reason why one particular dataset gets a better result on the SVMMulticlass is not completely apparent and could be researched further. The reason why the two other datasets get such bad results can probably be accounted to the strange or absent relationship between the class label and the attributes.

2.2.3 RLS vs. Rest

To finalize our research, we present the results of the experiments with the other methods that we used, again comparing them with our results on those particular datasets. Because the WEKA program doesn't provide the AUC score, but rather outputs an average performance over 10 folds, we implemented a Python script to compute the AUC values ourselves using the output predictions produced by WEKA. We consider table **2.2.5**, with the results for all algorithms. The performances in that table are optimized as described in the previous section. This again are AUC scores, which means that 0.5 is a random prediction and 1 is a perfect one.

	RLS	Bayes	Kstar	Random Forest
Australian	0.9205	0.9850	1.0000	0.9874
Diabetes	0.8231	0.5920	0.5739	0.5852
Fourclass	1.0000	0.9760	1.0000	0.9998
Ionosphere	0.9473	0.9897	0.9998	0.9936
Splice	0.9490	0.8280	0.8280	0.9225

Table **2.2.5** AUC RLS vs. Rest Optimized

It can be seen that the performance depends on the dataset that is being analyzed. This is because each algorithm uses his own methods to detect the relationships. This is exactly the reason why we try to consider as many

different datasets as possible. Another important thing to note is that the most problems from the LIBSVM website are relatively easy compared to a dataset like SCOP. This means that the relationship between the class labels and the features is mostly linearly separable. This is probably the reason that the performance of the researched algorithms is similar to the performance of the Regularized Least Squares algorithm on most datasets. On the Diabetes dataset there is a notable drop in performance probably because this problem is more difficult.

Chapter 3

Conclusion

We present an algorithm that can be used for both binary and multi-class classification. The latter is done using a joint kernel approach. We show that it has similar results as the other machine learning algorithms that we research, and even significantly outperforms them on one dataset. This is probably because of the fact that this dataset is non-linearly separable. We have also shown that it slightly outperforms SVMLight on all the chosen binary datasets but one. In terms of computation time it was hard to compare the performance of the two algorithms, mainly because we used different computers to compute the results. Next to that would the comparison not be completely fair because the SVMLight algorithm is optimized extensively and our implementation is not. One thing we can consider is the computational complexity of both algorithms, which is $O(qlf)$ for each iteration of SVMLight[12], where q is the number of rows in the Hessian matrix, l is the number of training examples and f is the maximum number of non-zero features in any of the training examples. For RLS the complexity[14] of calculating $\boldsymbol{\alpha}$ is $O(n^3)$, where n is the number of features. The disadvantage is that these complexities are not that suited for comparison because of the different variables that influence them. Nevertheless, one could run all algorithms on the same computer and compare the computation time needed for each one. With respect to the memory use one can say that the algorithms need more memory as the problems get larger, where more than 1 GB is no exception (on a SCOP family), but a precise comparison is not considered in this thesis. The comparison of the multiclass part also resulted in a notable better performance of the Regularized Least Squares algorithm. One can say that this comparison isn't completely fair because of the fact that the SVMMulti algorithm doesn't support a Gaussian kernel unless it splits up the problem in multiple binary classification problems. To counter that, one can say that the possibility to use a Gaussian kernel grants permission to use one. One could always implement our algorithm with a

linear kernel to compare those results. We also research the performance of the algorithm on the SCOP database, on which we also get stable results on most families. Some do give an AUC of under 0.5, which can probably be accounted to the non-linear separability of those families, or the absence of a real relationship between the features and the class labels. The reason why the normal version did get all AUC scores above 0.5 is probably because this algorithm is biased on the test set, meaning that it uses the test data to optimize the parameters. Another reason could be that we do not use a Gaussian kernel while researching SCOP because of the increase of computational complexity when optimizing the σ parameter. Because of the increase of computational power this could be an interesting extension to be researched in future work.

Chapter 4

Appendix

4.1 Python

In this section we give a brief description of the programming language Python, which we use to implement all our code. The goal of this section is not to give a full overview of all the functions, which can be found on[9], but to give someone who is familiar with imperative programming an idea how the language is built up.

Python is a very clean language, which is very easy to learn when one knows C++ or Java. It uses a simple syntax that's the same on all basic control structures. It also uses the modularity by enabling the possibility to use different files (filename.py) for the different classes. One downside is that debugging has to be done manually, with a lot of output to the console screen. We begin with the hello world program, which can not be more intuitive:

print "hello world" or *print 'hello world'*

As we can see there is no semicolon and Python does it with both (") and (""). To get a better idea of the complete structure next we will consider the basic statements, like the conditional statement:

if condition:
body
elif condition:
body

As we can see there is again no semicolon, the structure of the statement is only done with the indent in the body of the if statement. The condition can be an (in)equality, a Boolean or a concatenation of those using one of the keywords *and* or *or*. The *elif condition:* can also be replaced by *else:* if

you don't want to use another condition. Next we have the for-loop, which can be used in different ways making it very convenient to work with:

for element in list:
body

This just picks each element from a list, which can either be a real list or a string (which essentially is a list of characters), and assigns it to the *element* variable. Another nice thing is that Python does all the typing of the variables for you, if you assign something(y) to a variable x, that variable x will get the same type as y, overwriting the old type. The second way to use the for loop is with the index of the list, where we can again also use it to identify the index of a character in a string:

for index in range(0,len(list):
body

It's important to note that the names *index, element and list* are arbitrary and that *body* is a way to denote another statement. *len*(), on the other hand, is a function from Python. All statements that you want to be inside the for loop need the same indentation in the beginning as the last statement in that for loop. If you want to exit the loop you just begin the statement on the same indent as the for statement itself. This works the same with all functions. Another important function of a programming language, especially in our case, is to read and write files, which is also very easy in Python. One simply decides what to do with the file (read,(over)write,append) and choose the corresponding parameter for option ('w' for write, 'r' for read, 'a' for append). Write creates a file if it doesn't exist and read and append will give an error if it doesn't exist. Write will also overwrite if the file exists. Now one can simply use the function

file = open(filename, option)

and then use *lines = file.readlines()* to read or *file.write(string)* to write (a string!) to a file. The function *readlines()* returns a list with all the lines in the file.

4.2 Algorithm

This section does not consist of an overview of the code, because there is so much code that it would become a mess. Instead, the code is available on request and this section describes how to use the different scripts that we use to research the algorithms. Because all is written in Python, you will need the Python program (http://www.python.org/), as well as the NumPy package (http://numpy.scipy.org/). NumPy is the fundamental package needed for scientific computing with Python.

4.2.1 RLS on SCOP

For this script we can run the program by entering

python Run.py from to folds

Here *from* is the family where the algorithm starts, *to* is the family until where the algorithm will run and *folds* is the number of folds used. Where *from* $\in [2, 54]$, *to* $\in [3, 55]$ and *folds* is 1, 5 or 10, 1 being normal, 5 and 10 being 5 and 10 fold cross validation. The program outputs several files, on the cross validation first of all it creates a folder for the family being considered, 1 file for each fold, containing a list of AUC scores for each lambda and one file with the final lambda and the AUC score on the real test set. On the normal mode it creates a single file with all AUC scores and another script can be run to calculate all the maximum AUC scores for all families in *families.info* by typing

python AUC_curve.py

4.2.2 RLS on Rest

To run the RLS algorithm on other datasets from the LIBSVM website[5], both binary and multi-class the methods are the same. There are two ways to run the program, either using

python Run.py
or using
python Rls_auc.py x

Where *Run.py* runs it for all problems and the latter uses x, an integer variable, which in our binary script can range from 0 to 4, to choose one of five problems. In the multi-class script it can range from 0 to 5, because there are 6 multi-class problems which we consider. If one wants to use the latter way, the file *Rls_auc.py* has to be edited slightly by removing the comment symbol, *(#)*, before the line *#filechoice = int(sys.argv[1]) #first argument (choose file).*

To change the datasets that are being used one can simply edit the same file and assign other names to the *filename* variable. This script uses one input file and next to creating an output file *.out*, it also creates the input files for the SVMLight and SVMMulti scripts. As mentioned above, this works on both the binary and the multiclass classification script.

4.2.3 SVMLight and SVMMulti

The SVMLight wrapper consists of two files, one to train the algorithm and the second to run the real performance test. The first one, called by *python SVMLightWrapper.py*, runs the algorithm for all problems in the list *subprobs[]*. The second one, called by *python SVMLightWrapperTest.py*, does the same but now does it only once, and uses the parameters in the lists *lambdas* and *sigmas*, which have to be entered manually. This script uses input files created by the binary RLS script and creates one file *results_$name$* where *$name$* is the name of the problem. It also creates a *.model* file for each fold. There's also a file named *SVMLightWrapper(linear).py*, which can be used to run the SVMLight algorithm with a linear kernel. The SVMMulti wrapper works the same, but here there is no possibility to use a Gaussian kernel.

Bibliography

[1] Erin L. Allwein, Robert E. Schapire, and Yoram Singer. Reducing multiclass to binary: a unifying approach for margin classifiers. *Journal of Machine Learning Research*, 1:113–141, 2001.

[2] Ulf Brefeld and Tobias Scheffer. Auc maximizing support vector learning. In *In Proceedings ICML workshop on ROC Analysis in Machine Learning*, 2005.

[3] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[4] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[5] Chih-Chung Chang and Chih-Jen Lin. Libsvm. `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

[6] John G. Cleary and Leonard E. Trigg. K*: An instance-based learner using and entropic distance measure. In *ICML*, pages 108–114, 1995.

[7] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[8] Bradley Efron and Gail Gong. A leisurely look at the bootstrap, the jackknife, and cross-validation. *The American Statistician*, 37(1):36–48, 1983.

[9] Python Software Foundation. Python documentation. `http://docs.python.org/`.

[10] Eibe Frank, Mark Hall, and Len Trigg. Weka. `http://www.cs.waikato.ac.nz/ml/weka/`.

[11] Trevor Hastie and Robert Tibshirani. Classification by pairwise coupling. In *NIPS '97: Proceedings of the 1997 conference on Advances in*

neural information processing systems 10, pages 507–513, Cambridge, MA, USA, 1998. MIT Press.

[12] Thorsten Joachims. Making large-scale support vector machine learning practical. pages 169–184, 1999.

[13] Thorsten Joachims. A support vector method for multivariate performance measures. In Luc De Raedt and Stefan Wrobel, editors, *Proceedings of the 22nd International Conference on Machine learning*, volume 119 of *ACM International Conference Proceeding Series*, pages 377–384, New York, NY, USA, 2005. ACM Press.

[14] Shawe-Taylor John and Cristianini Nello. *Kernel Methods for Pattern Analysis*. Cambridge University Press, June 2004.

[15] Li Liao and William Stafford Noble. Combining pairwise sequence similarity and support vector machines for detecting remote protein evolutionary and structural relationships. *Journal of Computational Biology*, 10(6):857–868, 2003.

[16] Charles X. Ling, Jin Huang, and Harry Zhang. Auc: a statistically consistent and more discriminating measure than accuracy. In *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*, pages 519–524, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[17] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[18] Alexey G. Murzin, John-Marc Chandonia, Antonina Andreeva, Dave Howorth, Loredana Lo Conte, Bartlett G. Ailey, Steven E. Brenner, Tim J. P. Hubbard, and Cyrus Chothia. Scop. `www.bio.cam.ac.uk/scop/`.

[19] K. B. Petersen and M. S. Pedersen. The matrix cookbook, oct 2008.

[20] Ryan Rifkin and Aldebaro Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–141, 2004.

[21] Ryan Rifkin, Gene Yeo, and Tomaso Poggio. Regularized least-squares classification. In *Advances in Learning Theory: Methods, Models and Applications*, volume 190. IOS Press, 2003.

[22] Ryan Michael Rifkin. *Everything Old is New Again: A Fresh Look at Historical Approaches in Machine Learning*. PhD thesis, MIT, 2002.

[23] Johan A K Suykens, Tony Van Gestel, Jos De Brabanter, Bart De Moor, and Joos Vandewalle. *Least Squares Support Vector Machines*. World Scientific Publishing, Singapore, 2002.

[24] Evgeni Tsivtsivadze, Jorma Boberg, and Tapio Salakoski. Locality kernels for protein classification. In Raffaele Giancarlo and Sridhar Hannenhalli, editors, *Proceedings of the 7th International Workshop on Algorithms in Bioinformatics, (WABI 2007)*, pages 2–11. Springer, 2007.

[25] Jason Weston, Bernhard Schölkopf, and Olivier Bousquet. Joint kernel maps. In Joan Cabestany, Alberto Prieto, and Francisco Sandoval Hernández, editors, *IWANN 2005, Vilanova i la Geltrú, Barcelona, Spain, June 8-10, 2005, Proceedings*, Lecture Notes in Computer Science, pages 176–191, 2005.

ISBN ?
ISSN 1239-1883