Departament de Llenguatges i Sistemes Informatics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Master in Computing

Master of Science Thesis

An investigation into new kernels for categorical variables

Marco Antonio Villegas García

Advisor: Lluís Belanche Muñoz

January 2013

Contents

Abstract . . . 5

1 **Introduction** 7
1.1 Objectives of this thesis . . . 9
1.2 About this document . . . 9

2 **Kernel Methods** 11
2.1 About SVM . . . 11
2.1.1 The Separable Case . . . 12
2.1.2 The Non-Separable Case . . . 15
2.1.3 Nonlinear Support Vector Machines . . . 15
2.2 Kernel matrix . . . 16

3 **Existent Kernel Functions** 19
3.1 Polynomial kernels . . . 20
3.2 RBF kernels . . . 21

4 **New kernels for categorical variables** 23
4.1 Kernel functions . . . 23
4.2 Univariate functions . . . 24
4.2.1 Introducing probabilities . . . 25
4.3 Composition functions . . . 28
4.4 Transformation functions . . . 29

5 **Software implementation** 31
5.1 Computing Kernel Matrices . . . 31
5.1.1 Version 1: Building our kernel functions . . . 32
5.1.2 Version 2: Building our kernel matrices . . . 35
5.1.3 Version 3: Profiting intermediate computations . . . 38
5.2 Numerical evaluation of performance . . . 39

6 **Experimental Evaluation** 43
6.1 Data partitioning . . . 44
6.2 Cross-validation . . . 44
6.3 Artificial datasets . . . 45
6.4 Experiments with Datasets . . . 46

6.4.1 Artificial Data 47
6.4.2 Real Data 49
6.5 Experiments design and Results 50
6.5.1 GMonks 50
6.5.2 Synthetic-Poiss 52
6.5.3 PromoterGene 53

7 Conclusions and After Thoughts **57**
7.1 Future Work 58

A Demonstrations **61**
A.1 Preliminaries 61
A.2 Kernel k_0 62
A.3 Kernel k_1 64

Abstract

Kernel-based methods first appeared in the form of support vector machines. Since the first Support Vector Machine (SVM) formulation in 1995 [1] and [2], we have seen how the number of proposed kernel functions has quickly grown, and how these kernels have approached a wide range of problems and domains. The most common and direct applications of these methods are focused on continuous numeric data, given that SVMs at the end involves the solution of an optimization problem [3]. Additionally, some kernel functions have been oriented to more *symbolic* data, in problems like text analysis, or hand-written digits recognition [4]. But surprisingly, there is a gap in the area of kernel functions devoted to handle datasets with qualitative variables. One of the most common practices to overcome this lack consists on recoding the source qualitative information, making them suitable for applying numeric kernel functions.

This thesis presents the development of new kernel functions that can better model symbolic information presented as categorical variables, in a direct way, and without the need of data preprocessing methods. The proposition is based on the use of probabilistic information (probability mass distribution) to compare the different modalities of a variable. Additionally, the idea is formulated through a modular schema, combining a set of components to obtain the kernel functions, facilitating the modification and extension of single components.

The experimental results suggest an slightly improvement with respect to traditional kernel functions, in the accuracy obtained on classification problems. This progress is clearer on datasets with known probabilistic structure.

Chapter 1

Introduction

Since old centuries the man has dealt with the task of analysing data to extract patterns or regularities. The objective is always finding hints about how a certain phenomenon works, and hopefully understand it. If a full understanding is not possible, then at least a rough hypothesis is formulated. Consider, for example, the works developed by Gregor Mendel and formulated in his laws of inheritance. Between 1856 and 1863 Mendel cultivated and tested some 29,000 pea plants. After long time of thinking and analysing data, his experiments led him to make two generalizations, the *Law of Segregation* and the *Law of Independent Assortment* [5], which later became the famous Mendel's Laws of Inheritance. They express, in concise sentences, the patterns that Mendel found in his experiments.

Pattern Analysis deals with the problem of (automatically) finding and characterising patterns or regularities in data [4]. By patterns we understand any relations, regularities or structure inherent in a source of data. By detecting patterns, we can expect to build a system that is able to make predictions on new data extracted from the same source. If it works well, we say that the system has acquired *generalization power* by learning something from the data.

This approach is commonly called the *learning methodology*, because the process is focused on extracting patterns from the sample data that lead us to make generalizations about the population data. In this sense, it is a *data driven* approach, in contrast with *theory driven* approach. However, it is extremely useful to tackle complex problems in which an exact formulation is not possible, for example, recognising a face in a photo or genes in a DNA sequence.

Consider a dataset containing thousands of observations of pea plants, in the same format of Mendel's observations. It is obvious that the characters (color and size, for example) of certain pea plant generation could be predicted by using the Mendel's laws. Therefore, the dataset contains an amount of redundancy, that is, information that could be reconstructed from other parts of the data. In such cases we say that the

dataset is *redundant*.

This characteristic has an special importance for us, because the redundancy in the data leads us to formulate relations expressing such behaviours. If the relation is accurate and holds for all observations in the data, we refer to it as an *exact relation*. This is the case, for example, of the Laws of Inheritance: Mendel found that some patterns surprisingly held for all his experiments. For that reason, we say that this part of the data is also *predictable*: we can reconstruct it from the rest of the data, as well as predicting future data, like the color and size of new plants by using the current plants data.

Finding exact relations is not, by far, the general case for someone who analyses data. Certainly, the common case is finding patterns that hold with a certain probability. We call them *statistical relations*. Examples of such relations are: forecasting the total sales of a company for the next month, or inferring the credit score [6] of a new client in a bank by analysing his information.

The *science* of pattern analysis has considerably evolved from its early formulations. In the 1960's efficient algorithms for detecting linear relations were introduced. This is the case of the Perceptron algorithm [7], formulated in 1957. In the mid 1980's a set of new algorithms started to appear, making possible for the first time to detect nonlinear patterns. This group includes the backpropagation algorithm for multilayer neural networks and decision tree learning algorithms.

The emergence of the new pattern analysis approach known as kernel-based methods in mid 1990's, changed the field of pattern analysis towards a new and exciting perspective: the new approach enabled researchers to analyse nonlinear relations with the efficiency of linear algorithms via the use of kernel matrices.

Kernel-based methods first appeared in the form of support vector machines. Support Vector Machine (SVM) is a classification algorithm that quickly gained great popularity in the community for its efficiency and robustness. But the kernel approach is more than an concrete algorithm: it provides a unified framework to deal and operate on data of all types, be they vectorial, strings, or more complex structures, and enabling the analysis of a wide range of relations including correlations, rankings, clusterings...

The key component of a kernel-based method is the so called *kernel function*. Most of the efforts in the area of kernel methods are devoted to define kernel functions that capture the most of the relevant information contained in the data.

SVM was formulated in 1995 [1]; since then, many kernel functions have been proposed. The most common and direct applications of these methods are focused on numeric data, given that SVM at the end involves the solution of an optimization problem [3]. Additionally, some kernel functions have

been oriented to more *symbolic* data, in problems like text analysis, or hand-written digits recognition [4]. But surprisingly, there is a gap in the area of kernel functions devoted to handle datasets with qualitative variables. One of the most common practices to overcome this lack consists on recoding the source qualitative information, making them suitable for applying numeric kernel functions.

1.1 Objectives of this thesis

The main objective of this thesis is the development of new kernel functions that can better model symbolic information presented as categorical variables, in a direct way, and without the need of data preprocessing methods. These new functions have been formulated through a modular schema, combining a set of component to obtain the kernel function, facilitating the modification and extension of single components. In fact, this modular schema gives rise to a complete family of new kernel functions, which are focused to dealing with categorical data, but it could be regarded itself as a *framework* to define additional new kernel functions.

In particular, we defined a set of *univariate kernels* which deal with individual dimensions of the data. The *composition functions* integrate these uni-dimensional values into a single scalar. At each step of the modular schema, the *transformation functions* are optionally applied to endow additional flexibility to the method.

Concerning the processing of categorical data, we have defined *univariate kernels* that use the probabilistic information (probability mass distribution) to compare the different modalities of a variable. With this approach we overcome the problem of recoding source data, and at the time, we open the possibility of properly handle different types of data with a single kernel.

With the aim of studying the behaviour and performance of these new kernels, we have developed a set of experiments with real and artificial datasets. In consequence, it was necessary to actually implement and test the proposed functions. All this was done using the language and environment provided by R software [8]. Additionally, we have designed a new artificial dataset called `SyntheticPoiss`, which shows the convenience of using the proposed kernels in datasets with known statistical structure.

1.2 About this document

The remainder of this document is organized as follow: Chapter 2 presents founding concepts of kernel methods by means of the notions drawn from support vector machines. Chapter 3 characterizes some of the most well known kernel functions. Chapter 4 describes the foundations of the new family of kernel functions for categorical variables. Chapter 5 presents the

software environment used for experiments and details of implementation. Chapter 6 firstly introduces some common practices in data-based experiments; then, it contains a brief description of the datasets used in experiments. Finally, it presents the results of the experiments and the conclusions that we draw from them.

Chapter 2

Kernel Methods

All kernel-based methods are composed by two main components. First, the original data is embedded into a high-dimensional space, usually named *feature space*. Then, a linear method is applied to discover patterns in the embedded data. In spite of its simplicity, this modular architecture involves many advantages [4]:

(1)Detecting linear relations has been the focus of much research in statistics and machine learning, resulting in a robust and theoretically well founded set of methods, generally understood and accepted by the community of researchers.

(2) Whatever linear method we decide to apply, we should perform a embedding of the original data into a feature space, in which it is easier to find linear patterns. Again we have the freedom of selecting the appropriate mapping function, which must satisfy some conditions (presented in the next paragraphs).

And finally, (3) there is a computational short-cut which make it possible to represent linear patterns in a high dimensional feature space without explicitly computing the corresponding mapping, known as the *kernel trick*. The component that make it possible is called the *kernel function*.

The kernel function is actually the key element of the approach.

2.1 About SVM

Support Vector Machines [1] are built on the idea of linear decision boundaries to solve binary classification problems.

In this section we mainly follow the notation and formulations from [3]. Suppose we are given n observations. Each observation consists of a pair: a vector $\mathbf{x}_i \in R^d, i = 1, \ldots, n$ and the class z_i which the vector $\mathbf{x}_i$ belongs to. The base case correspond to a binary classification problem, in which $z_i \in \{-1, 1\}$. Rather than memorizing the concrete associations $\{\mathbf{x}_i, z_i\}$, we look for the underlying probability distribution from which the data are

generated, in such a way that the learned patterns are generalizable over the population of data, not only over the sample of size n.

Now assume that we have a machine whose task is to learn the associations $\{\mathbf{x}_i, z_i\}$. In fact, the machine is defined as a function of vector $\mathbf{x}$ and certain adjustable parameters α. In this sense, the associations computed by the machine will be $\{\mathbf{x}, f(\mathbf{x}, \alpha)\}$. A particular choice of α generates what we will call a ”trained machine”.

We focus now on how the Support Vector Machines manage to learn and generalize the patterns provided in the sample data.

2.1.1 The Separable Case

We start with the simplest case: a linear support vector machine trained on separable data. Again we are given the labelled training data $\{\mathbf{x}_i, y_i\}$, $\mathbf{x}_i \in R^d$, $y_i \in \{-1, 1\}$ and $i = 1, \ldots, n$. Suppose we have some hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ which separates the positive from the negative examples. $\mathbf{w}$ is a normal vector to the hyperplane, $|b|/ \parallel \mathbf{w} \parallel$ is the perpendicular distance from the hyperplane to the origin, and $\parallel \mathbf{w} \parallel$ is the Euclidean norm of $\mathbf{w}$. Let $d_+(d_-)$ be the shortest distance from the separating hyperplane to the closest positive (negative) example. The ”margin” of a separating hyperplane is defined as $d_+ + d_-$. For the linearly separable case, the support vector algorithm simply looks for the separating hyperplane with largest margin.

Assuming that the classes are perfectly separable, we would like that the hyperplane satisfies the following conditions:

$$\langle \mathbf{w}, \mathbf{x}_i \rangle + b \geq +1, y_i = +1 \tag{2.1.1}$$

$$\langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq -1, y_i = -1 \tag{2.1.2}$$

Which can be combined into the following set of inequalities:

$$y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) - 1 \geq 0 \tag{2.1.3}$$

for all $i = 1, \ldots, n$.

These inequalities require to compute the appropriate scale for $\mathbf{w}$ and b. Now consider the points for which the equality in (2.1.1) holds. These points lie on the hyperplane

$$H_1 : \langle \mathbf{w}, \mathbf{x}_i \rangle + b = +1.$$

Similarly, the points that satisfies the equality in Eq. (2.1.2) lie on the hyperplane

$$H_2 : \langle \mathbf{w}, \mathbf{x}_i \rangle + b = -1.$$

Hence $d_+ = d_- = 1/ \parallel \mathbf{w} \parallel$ and the margin is $2/ \parallel \mathbf{w} \parallel$. The hyperplanes H_1 and H_2 are parallel (they share the vector $\mathbf{w}$), and no training points fall between them. In consequence, we can find the pair of hyperplanes which gives the maximum margin by solving the following problem:

$$\begin{aligned} &\textit{minimize} && \frac{\| \mathbf{w} \|^2}{2} \\ &\textit{subject to} && \text{constraints (2.1.3).} \end{aligned}$$

We follow the intuition that having a larger margin $2/ \| \mathbf{w} \|$ will increase the generalization power of the final classifier. We expect the solution for a typical two dimensional case to have the form shown in Figure 2.1. The so called "Support Vectors" are the training points for which the equality in Eq. (2.1.3) holds, those which lie on one of the hyperplanes H_1, H_2. These are the points that actually determine the found solution: if we remove one of them, the solution will change. They are indicated in Figure 2.1 by the extra circles.

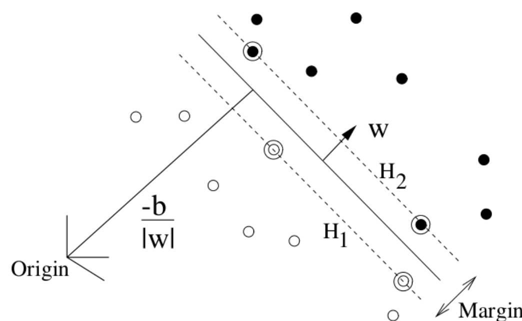


Figure 2.1: Linear separating hyperplanes in the case of separable classes. Source [3].

At this point we need to transform the problem into a Lagrangian formulation. This change has at least two advantages. Firstly, the constraints (2.1.3) will be replaced by constraints on the Lagrange multipliers themselves, which will be much easier to handle. And second, in this reformulation the training points will only appear in the form of dot products between vectors, which will be of much importance to generalize the procedure to the nonlinear case.

Then, we introduce non-negative Lagrange multipliers $\alpha_i, i = 1, \ldots, n$, one for each of the inequality constraints (2.1.3) [1]. This gives the Lagrangian:

$$L_P = \frac{1}{2} \| \mathbf{w} \|^2 - \sum_{i=1}^{n} \alpha_i y_i (\langle \mathbf{x}_i, \mathbf{w} \rangle + b) + \sum_{i=1}^{n} \alpha_i \tag{2.1.4}$$

And our problem becomes:

[1] For details about the Lagrangian reformulation of the problem see [4].

$$\begin{array}{rl} \textit{minimize} & L_P \\ \textit{with respect to} & \mathbf{w}, \mathrm{b} \\ \textit{subject to} & \alpha_i \geq 0 \end{array}$$

assuming that the derivatives of L_P with respect to all α_i vanish.

Now this is a convex quadratic programming problem, since the objective function is itself convex. This means that we can equivalently solve the following ”dual” problem:

$$\begin{array}{rl} \textit{maximize} & L_P \\ \textit{with respect to} & \alpha_i \\ \textit{subject to} & \alpha_i \geq 0 \end{array}$$

and the gradient of L_P with respect to **w** and b vanishes.

This dual formulation of the problem has the property that the solution occurs at the same values of **w**, b and α for the primal formulation.

Requiring that the gradient of L_P with respect to **w** and b vanish give the conditions:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \tag{2.1.5}$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.1.6}$$

Since these are equality constraints in the dual formulation, we can substitute them into Eq. (2.1.4) to give

$$L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \tag{2.1.7}$$

Note that now we have two different formulations: L_P and L_D, primal and dual. Both of them come from the same objective function but with different constraints. The solution is found by minimizing L_P or by maximizing L_D. Notice also that there is a Lagrange multiplier α_i for every training point. In the solution, those points for which $\alpha_i > 0$ are called ”support vectors”, and lie on one of the hyperplanes H_1, H_2. All others training points have $\alpha_i = 0$ and lie either on H_1 or H_2, satisfying the equality in Eq. (2.1.3), or on that side of H_1 or H_2 such that the strict inequality of Eq. (2.1.3) holds. The support vectors are the critical elements of the training set; they lie closest to the decision boundary; if all other training points were removed, or moved around while maintaining the class membership, and training was repeated, the same separating hyperplane would be found.

2.1.2 The Non-Separable Case

In this section we are going to extend the above algorithm in such a way it can handle non-separable data. We do that by relaxing the constraints (2.1.1) and (2.1.2), but only when is necessary. This can be done by introducing non-negative slack variables $\xi_i, i = 1, \ldots, n$ in the constraints, which then become:

$$\langle \mathbf{w}, \mathbf{x}_i \rangle + b \geq +1 - \xi_i \text{ for } y_i = +1 \tag{2.1.8}$$

$$\langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq -1 + \xi_i \text{ for } y_i = -1 \tag{2.1.9}$$

$$\xi_i \geq 0 \text{ for } i = 1, \ldots, n \tag{2.1.10}$$

The next step is to assign an extra cost for errors, by changing the objective function to be

$$\frac{\| \mathbf{w} \|^2}{2} + C \sum_{i=1}^{n} (\xi_i)^k \tag{2.1.11}$$

where C is a parameter called "cost", to be chosen by the user: a larger C corresponds to imputing a higher penalty to errors. Again this is a convex programming problem for any positive integer k, and the dual problem becomes:

Maximize

$$L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \tag{2.1.12}$$

subject to:

$$0 \leq \alpha_i \leq C, \tag{2.1.13}$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.1.14}$$

And the solution is again given by

$$\mathbf{w} = \sum_{i=1}^{N_S} \alpha_i y_i \mathbf{x}_i \tag{2.1.15}$$

where N_S is the number of support vectors. The process is represented in the Figure 2.2.

2.1.3 Nonlinear Support Vector Machines

One of the stronger assumptions for the above methods is that the class function (the underlying function that assign the "true" class) is a linear function of the data. Although they are not directly applicable to non-linear data, there is a trick showed in [9] that allow to generalize them making possible to deal with non-linear data.

The first step is to realize that the data appears in the training points (Eq. 2.1.12) only in the form of dot products $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$. Now assume that we first map the data to some other vector space H, using a mapping function ϕ.

$$\phi : \mathbf{R}^d \to H \tag{2.1.16}$$

Following this idea, as the training points already appear through dot products, after the mapping they will appear again as dot products but in the space H, on functions of the form $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$. In fact, the so called "kernel function", defined as $K(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$, allow us not to have to explicitly compute (even know) the mapping ϕ. We directly use K in the training algorithm.

This simple schema introduce a set of amazing possibilities, with minimal computational overload. For example, the space H is normally a high dimensional space, and could be even of infinite dimensions.

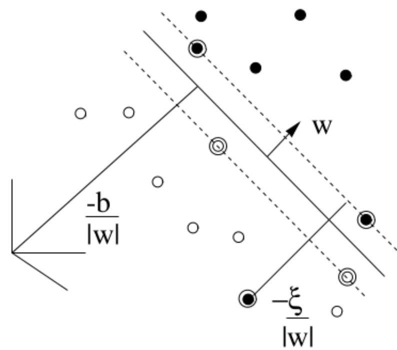


Figure 2.2: Linear separating hyperplane for the non-separable case. Source [3].

2.2 Kernel matrix

Given a kernel and a training set, we can form the matrix known as the *kernel matrix*: the matrix containing the evaluation of the kernel function on all pairs of data points. This matrix acts as an information bottleneck, as all the information available to learning method must be extracted from that matrix. In consequence, the kernel matrix plays a central role in all kernel methods, and we devote a big part of the Chapter 5 to optimize its computation.

Given a set of vectors $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the *kernel matrix* is defined as the $n \times n$ matrix K whose entries are $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, where k is a valid kernel function. The standard notation for displaying kernel matrices is:

K	1	2	$\ldots$	n
1	$k(\mathbf{x}_1, \mathbf{x}_1)$	$k(\mathbf{x}_1, \mathbf{x}_2)$	$\ldots$	$k(\mathbf{x}_1, \mathbf{x}_n)$
2	$k(\mathbf{x}_2, \mathbf{x}_1)$	$k(\mathbf{x}_2, \mathbf{x}_2)$	$\ldots$	$k(\mathbf{x}_2, \mathbf{x}_n)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	$k(\mathbf{x}_n, \mathbf{x}_1)$	$k(\mathbf{x}_n, \mathbf{x}_2)$	$\ldots$	$k(\mathbf{x}_n, \mathbf{x}_n)$

Chapter 3

Existent Kernel Functions

We have seen that the core component of a kernel-based method is the kernel function. Therefore, it is important to pay special attention in designing and selecting the proper kernel for a certain problem. There are two properties to be satisfied by all kernels: firstly, they should capture the similarity measure between objects in the particular task or application domain, and secondly, its evaluation should be simpler than explicitly computing the corresponding mapping and dot-product in the feature space.

In this section we present an overview of well known kernel functions, widely used and accepted by the research community. They are the basis for the new kernels we are going to present later on.

It is known that kernels satisfy a number of closure properties, allowing us to create more complex kernels by a combination of simpler elements. The following theorem presents a subset of the closure properties of kernels.

Theorem 3.1. *Let k_1 and k_2 be kernels over $X \times X$, $X \subset R^n$, $\alpha \in R^+$, $f(\cdot)$ a real-valued function on X, $\phi : X \rightarrow R^N$ with k_3 a kernel over $R^N \times R^N$. Then the following functions are kernels:*

- $k(x, y) = k_1(x, y) + k_2(x, y)$,
- $k(x, y) = \alpha k_1(x, y)$,
- $k(x, y) = k_1(x, y)k_2(x, y)$,
- $k(x, y) = f(x)f(y)$,
- $k(x, y) = k_3(\phi(x), \phi(y))$.

■

For details and demonstration of Theorem 3.1 see [4].

Given the mapping $\phi : \mathcal{X} \rightarrow \mathcal{H}$, the *kernel trick* consists in performing the mapping and the inner product simultaneously by defining its associated kernel function:

$$k(\mathbf{u}, \mathbf{v}) = \phi(\mathbf{u})^T \phi(\mathbf{v}) = \langle \phi(\mathbf{u})\phi(\mathbf{v}) \rangle_{\mathcal{H}}, \ \mathbf{u}, \mathbf{v} \in \mathcal{X}$$

This way it is possible to compute the hyperplane without explicitly carrying out the map into $\mathcal{H}$.

The original motivation for introducing kernels was to search for nonlinear patterns by using linear functions in a feature space using a nonlinear feature map. Among the most popular and well known kernels, we have selected the two more related with our work: the *Polynomial* and *RBF* kernels. In the following sections we present a basic description of them.

3.1 Polynomial kernels

The Theorem 3.1 give us the basic hints to see that the space of valid kernels is closed under the application of certain polynomials.

Theorem 3.2. *Let $k_1(\boldsymbol{x}, \boldsymbol{y})$ be a kernel over $X \times X$, where $\boldsymbol{x}, \boldsymbol{y} \in X$, and p is a polynomial with positive coefficients. Then the function*

$$k(x, y) = p(k_1(x, y))$$

is also a kernel. The special case where k_1 is linear, the function

$$k(\boldsymbol{x}, \boldsymbol{y}) = (\langle \boldsymbol{x}, \boldsymbol{y} \rangle + R)^m$$

is called the polynomial kernel, with $R \geq 0$ and $m \geq 1$.

■

As an example, consider the case when $\mathbf{x}, \mathbf{y} \in \mathbb{R}^2$, $R = 0$ and $d = 2$.

$$\begin{aligned}
k(\mathbf{x}, \mathbf{y}) &= \langle \mathbf{x}, \boldsymbol{y} \rangle^2 \\
&= \left[\left\langle \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \right\rangle \right]^2 \\
&= (x_1 y_1 + x_2 y_2)^2 \\
&= (x_1 y_1)^2 + 2 x_1 y_1 x_2 y_2 + (x_2 y_2)^2 \\
&= x_1^2 y_1^2 + (\sqrt{2} x_1 x_2)(\sqrt{2} y_1 y_2) + x_2^2 y_2^2 \\
&= \left\langle \begin{pmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{pmatrix}, \begin{pmatrix} y_1^2 \\ \sqrt{2} y_1 y_2 \\ y_2^2 \end{pmatrix} \right\rangle \\
&= \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{y}) \rangle
\end{aligned}$$

Therefore, $\phi : \mathbb{R}^2 \longrightarrow \mathbb{R}^3$ with $\phi(\boldsymbol{x}) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2)^T$

3.2 RBF kernels

Theorem 3.3. *Let $k_1(\boldsymbol{x},\boldsymbol{y})$ be a kernel over $X \times X$, where $\boldsymbol{x},\boldsymbol{y} \in X$. Then the function*

$$k(x, y) = exp(k_1(\boldsymbol{x},\boldsymbol{y})),$$

is also a kernel. The special case

$$k(x, y) = exp(- \parallel \boldsymbol{x} - \boldsymbol{y} \parallel^2 /(2\theta^2))$$

is called the RBF kernel.

Figure out that the RBF kernel is dependent on the Euclidean distance of **x** from **y**. In fact, one of these will be a support vector and the other will be the testing data point. The support vector will be the centre of the Gaussian and θ will determine the area of influence this support vector has over the data space. A larger value of θ will give a smoother decision surface and more regular decision boundary. This is because an RBF with large θ will allow a support vector to have a strong influence over a larger area.

Another interesting fact is that larger θ value also increases the value of the Lagrange multipliers α. When one support vector influences a larger area, all other support vectors in the area will increase in α-value to counter this influence. Hence all α-values will reach a balance at a larger magnitude.

Chapter 4

New kernels for categorical variables

In this section we introduce a new family of kernels functions especially designed for categorical variables. These are nominal or symbolic variables, and by definition, are non-numerical variables upon which no order relation has been defined, and the only meaningful relation is equality/non-equality. A categorical variable can take some discrete and unordered values, which are commonly known as *modalities*.

The proposed kernels accept categorical information in a direct way, without having to perform data-preprocessing. Since the only possible relation among categorical variables is the equality/non-equality, we propose to use *probabilistic* information to compare the modalities of a variable. This *probabilistic* information consists on the probabilistic distribution that generates the data. As we usually do not have this knowledge, we estimate it from the training data.

The remaining part of this section presents the different *components* of the proposed kernels in a modular fashion, starting with the univariate (kernel) functions, the composition functions and the transformation functions. These components are adequately combined to generate a full family of kernel functions.

4.1 Kernel functions

Let us recall the basic formulation of a kernel function:

Definition 1. *A kernel is a function K that for all $\boldsymbol{x}_i, \boldsymbol{x}_j \in X$ satisfies*

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle$$

where ϕ is a mapping from X to an (inner product) feature space F

$$\phi : \boldsymbol{x} \longmapsto \phi(\boldsymbol{x}) \in F$$

A Kernel is defined as a dot product in the feature space. In the space $\mathbb{R}^n$ we know that

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i$$

with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$.

Notice that we can decompose this operation in two *components*: first we compute the *univariate* interaction between elements of the same dimension (using simple product), and then we *integrate* this partial results using a *composition function* (in this case a sum). In this sense, we can express the dot product as follow:

$$\langle x, y \rangle = (x_1 \otimes y_1) \oplus (x_2 \otimes y_2) \oplus ... \oplus (x_n \otimes y_n)$$

with $\otimes \equiv \times$ and $\oplus \equiv +$. Naturally, the final result $\langle x, y \rangle$ depends on (1)the data we have (vectors x and y), (2)the univariate function $\otimes$ and (3)the composition function $\oplus$ we use.

The functions $\otimes$ and $\oplus$ are usually defined in the space $\mathbb{R}^n$, which means that they are only applicable to numerical (real) vectors, not categorical. If we want to apply the operation $\langle \cdot, \cdot \rangle$ to categorical variables, we need to create new functions $\oplus$ and $\otimes$ specially defined for this type of data.

In the following section we introduce a set of *univariate* functions specially defined to work with categorical variables. They are going to play the role of the function $\otimes$. After that, we will present another set of functions oriented to play the role of *composition* functions $\oplus$, which integrate the values returned by the *univariate* functions.

A word about notation. Unless we indicate, we will consider the dataset $D = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ as a $n \times d$ matrix, with n data-points $\mathbf{x}_i = \{x_{i,1}, x_{i,2}, ..., x_{i,d}\}$ of d dimensions. The response vector $Y = \{y_1, y_2, ..., y_n\}$ contains the class which each data point belongs to.

$$D_{n,d} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,d} \end{pmatrix} \quad Y_{n,1} = \begin{pmatrix} y_i \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

4.2 Univariate functions

In addition to the formal perspective, a kernel can also be regarded as a similarity measure [10] [4] between two data points. A similarity function is a function that satisfies the *Similarity principle*, which state

$$S(x,x) > S(x,y) \text{ , } x \neq y$$
$$S(x,y) \geq 0$$

As we are working with categorical variables, the more straightforward similarity measure is given by simple comparison. This simple idea is expressed with the following function:

Definition 2. *Univariate (overlap) kernel k_0*

$$k_0(x,y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases}$$

We can regard this as a similarity function: given two values (modalities), it returns "maximum similarity" if they are the equal, and "minimum similarity" if they are different. It is important to point out that this function is meant to work with single values, not vectors. Therefore, the returned value, in this case 0 or 1, is related only to the compared values, which belongs to the same dimension (column) of the dataset. In consequence, if we want to compare two categorical vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, we need to evaluate k_0 as many times as dimensions in $\mathbf{x}_i$ and $\mathbf{x}_j$, in such a way that we end up having the binary vector $\mathbf{u} = \{k_0(x_{i,1}, x_{j,1}), k_0(x_{i,2}, x_{j,2}), ..., k_0(x_{i,d}, x_{j,d})\}$ of d dimensions.

This binary vector $\mathbf{u}$ contains the univariate evaluations of kernel k_0 on data points $\mathbf{x}_i$ and $\mathbf{x}_j$. To obtain the final kernel value $k_0(\mathbf{x}_i, \mathbf{x}_j)$ it is necessary to *integrate* the vector $\mathbf{u}$ using a $composition$ function. Composition functions are presented in Section 4.3.

Theorem 4.1. *The kernel matrix generated by k_0 is positive semi-definite (p.s.d.).*

PROOF. See Appendix (A.2).

This kernel is mathematically valid but in some cases, since it is so simple, it can not capture the non-trivial relations present in the data. However, it will be our starting point to build more complex kernels.

4.2.1 Introducing probabilities

The previous function k_0 gives us very basic hints about the similarity of two values. In fact, it only detects if two values are equal or not. However, its formulation helps us to introduce a generalized form of the previous kernel function, measuring the similarity between the modalities (values) of a variable as a function of their probability: If the modalities coincide, then the similarity is higher the less probable they are. On the other hand, if the modalities do not coincide, the similarity remains zero.

Let Z be a categorical variable with Probability Mass Function (PMF) P_Z, and $P_Z(z_i)$ the probability that variable Z takes the value z_i.

Now we can define the following *univariate* kernel function:

Definition 3. *Univariate kernel k_1*

$$k_1(z_i, z_j) = \begin{cases} h(P_Z(z_i)) & \text{if } z_i = z_j \\ 0 & \text{if } z_i \neq z_j \end{cases}$$

When the values z_i and z_j coincide, we evaluate a new function $h(\cdot)$ on the value $P_Z(z_i)$ (which is equal to $P_Z(z_j)$). In fact, the function k_0 is a particular case of k_1, in which the function $h(\cdot)$ always returns 1.
The function $h(\cdot)$ has a special meaning, working as a *detector* of *relevant similarities*. Actually, the equality between two values is a fact that is more or less important depending on the probability $P_Z(z_i)$: if $P_Z(z_i)$ is very low, we are talking about an unusual and relevant event, because is more difficult to find another similar incident z_j. Conversely, if $P_Z(z_i)$ is very high, we are in presence of a common event, which is so usual that any similarity with another event lacks of interest. In consequence, the function $h(\cdot)$ should return a high value when $P_Z(z_i)$ is small, and a low value when it is big.

This way of thinking makes sense in many different scopes of human life. For example, a very common thing is that people catch a bad cold: if two persons go to the hospital with a common cold the same day, it is not a reason to rise alarms, and neither to conclude that these two persons are very similar, because everybody can catch a cold (its relative probability is high), and finding two persons with a cold is not very strange.

On the other hand, if two persons having a really odd disease go to the hospital in the same days, it should become a relevant fact because it is very difficult to find two persons suffering this rare disease. This coincidence tells us that these two persons *are* similar, or at least have something in common that makes them to develop this illness.

These arguments are encoded in the function $h(\cdot)$, which is introduced in the following definition.

Definition 4. *Inverting function $h(z)$*

$$h(z) = (1 - z^{\alpha})^{1/\alpha}.$$

with $\alpha > 0$.

■

The function $h(\cdot)$ depends on the parameter α which determines its non-linear behaviour. It is concave for $\alpha \in (0, 1)$ and convex for $\alpha \in (1, \infty)$. The reference case corresponds to $\alpha = 1$, in which $h(z) = (1 - z)$. Look at the Figure 4.1 to see different versions of function $h()$ according to the value of α.

The range of the function $h(P(z))$ is defined on the interval $(0, 1)$, since the argument $P(z) \in (0, 1)$ [1] and $\alpha > 0$.

[1]Without loss of generality, we assume that $P(z) \in (0, 1)$ excluding the values {0,1}, because $P(z) = 0$ would represent an impossible event, and $P(z) = 1$ a systematic one (with no randomness).

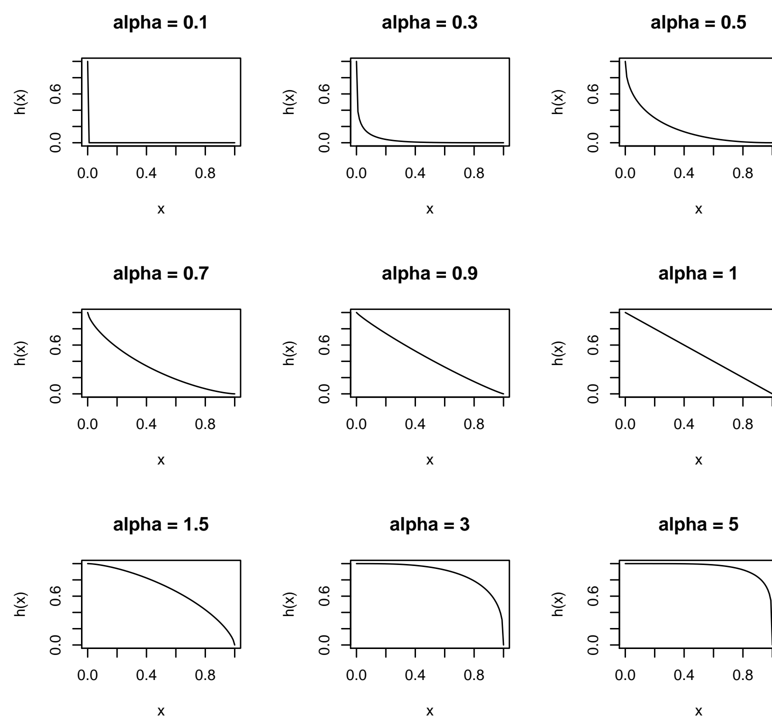


Figure 4.1: Inverting function $h(x)$

From Figure 4.1 we observe that while $P_Z(z)$ decreases, h increases. And this behaviour reflects the discussion at the beginning of this section: if the modalities coincide, then the similarity is higher the less probable they are. Additionally, the parameter α controls the way of function decreasing. These different values for the parameter α give rise to a family of kernel functions based on k_1.

Theorem 4.2. *The kernel matrix generated by k_1 is positive semi-definite (p.s.d.).*

PROOF. See Appendix (A.3).

The function k_1 (as k_0) is only an *univariate* kernel, returning a value concerning a single dimension of two data points. In consequence, if we want to compare two categorical vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, we need to evaluate k_1 as many times as dimensions in $\mathbf{x}_i$ and $\mathbf{x}_j$, in such a way that we end up having the binary vector

$$\mathbf{u} = \{k_1(\mathbf{x}_{i,1}, \mathbf{x}_{j,1}), k_1(\mathbf{x}_{i,2}, \mathbf{x}_{j,2}), ..., k_1(\mathbf{x}_{i,d}, \mathbf{x}_{j,d})\}$$

of d dimensions. ■

We have defined a set of *univariate kernels* for categorical variables, which correspond to the function $\otimes$.

In the next section we define some *composition functions* $\oplus$ to integrate the univariate results.

4.3 Composition functions

Let $\mathbf{u} = (u_1, u_2, ..., u_d)$ a vector of *univariate* evaluations. The composition function receive the vector $\mathbf{u}$ and returns a scalar value c, which will be the *complete* similarity measure between the points $\mathbf{x}_i$ and $\mathbf{x}_j$ associated with the vector $\mathbf{u}$. We define the following *composition functions*:

Definition 5. *Composition functions*

- $med(\boldsymbol{u}) = \frac{\sum_{i=1}^{n} \boldsymbol{u}_i}{n}$
- $pro(\boldsymbol{u}) = \prod_{i=1}^{n} \boldsymbol{u}_i$

The vector $\mathbf{u}$ is supposed to be the result of applying an *univariate function* on two data points $\mathbf{x}_i$ and $\mathbf{x}_j$. The *composition function* receives the vector $\mathbf{u}$ and returns a scalar value c, which is the similarity measure between $\mathbf{x}_i$ and $\mathbf{x}_j$. The resulting kernel matrix K will have the value c in the position (i, j) and (j, i), that is, $K_{ij} = K_{ji} = c$.

4.4 Transformation functions

In sections 4.2 and 4.3 we defined the two main components of the proposed kernels: *univariate functions* and *composition functions*. In this section we introduce a set of functions to provide additional flexibility to the method. The *transformation functions* are meant to be optionally applied *before* the composition step, and/or *after* the composition, as it is illustrated in Figure 4.2. These functions are:

Definition 6. *Transformation functions*

- $f_1(K(x,y)) = e^{\gamma K(x,y)}$
- $f_2(K(x,y)) = e^{\gamma[K(x,x)+K(y,y)-2K(x,y)]}$

■

By applying these functions, the property of being a valid kernel is maintained by virtue of the properties contained in Theorem 3.1.

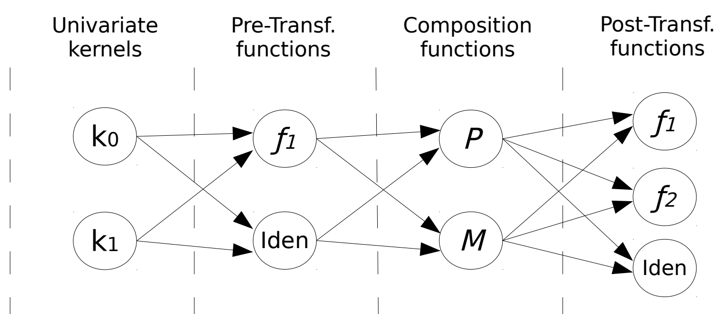


Figure 4.2: Experiments setup. *Here we can observe the modular architecture of the proposed kernels.* ***Iden*** *is the identity function,* ***P*** *and* ***M*** *are the "product" and "mean" composition functions, respectively. The selection of one component on each step will give us a new kernel function.*

In figure 4.2 we can see a simple representation of the modular architecture of the proposed kernels and the role of the transformation functions. Building a new kernel consists on selecting one function on each step, and apply it to the previous-step result. At the end of the process, the post-transformation functions will return a complete kernel matrix, according to the selected functions.

Chapter 5

Software implementation

All the experiments were developed using the R programming language. It was created by Ross Ihaka in 1995 [8], and has become one of the most used software for statistical analysis. This software can be easily extended by using *packages*, which are the main mechanism to interchange R code by the community of developers and users [11].

There are several R packages related to Kernel Methods and Support Vector Machines. Some of them are e1071[12], kernlab [13], svmpath [14] and many others [15]. For developing our experiments, we have selected kernlab package because it offers additional flexibility to accept user-defined kernel functions and works directly with kernel matrices.

In spite of having plenty of tools and functions implementing most of the SVM computation, it was necessary to develop considerable amount of code to design, integrate and summarize large set of experiments executed. In addition, we have developed efficient code for the computation of the new kernel matrices. This section contains the details of these pieces of code, how they were designed, and how they evolved to reduce more and more the space and time consumption.

5.1 Computing Kernel Matrices

We have seen that a kernel function k is a dot-product in a feature space:

$$k(x, y) = \langle \phi(x), \phi(y) \rangle \tag{5.1.1}$$

The kernel trick consist on not having to explicitly compute the mappings ϕ to evaluate the kernel k: the kernel function directly compute both the mapping and the inner-product in a feature space. One of the advantages of `KernLab` package is the possibility of building SVMs using user-defined kernel functions [13]. This feature enabled us to easily start making experiments with new ideas, by simply defining a standard R function and plugging it to `KernLab` package. However, as we will describe, this approach has a big

trouble: the execution time was extremely long, even though the applied kernels are pretty simple.

For that reason, the *implementation* of the functions k_0 and k_1 had evolved through few versions, to solve problems of large time and memory consumption.

5.1.1 Version 1: Building our kernel functions

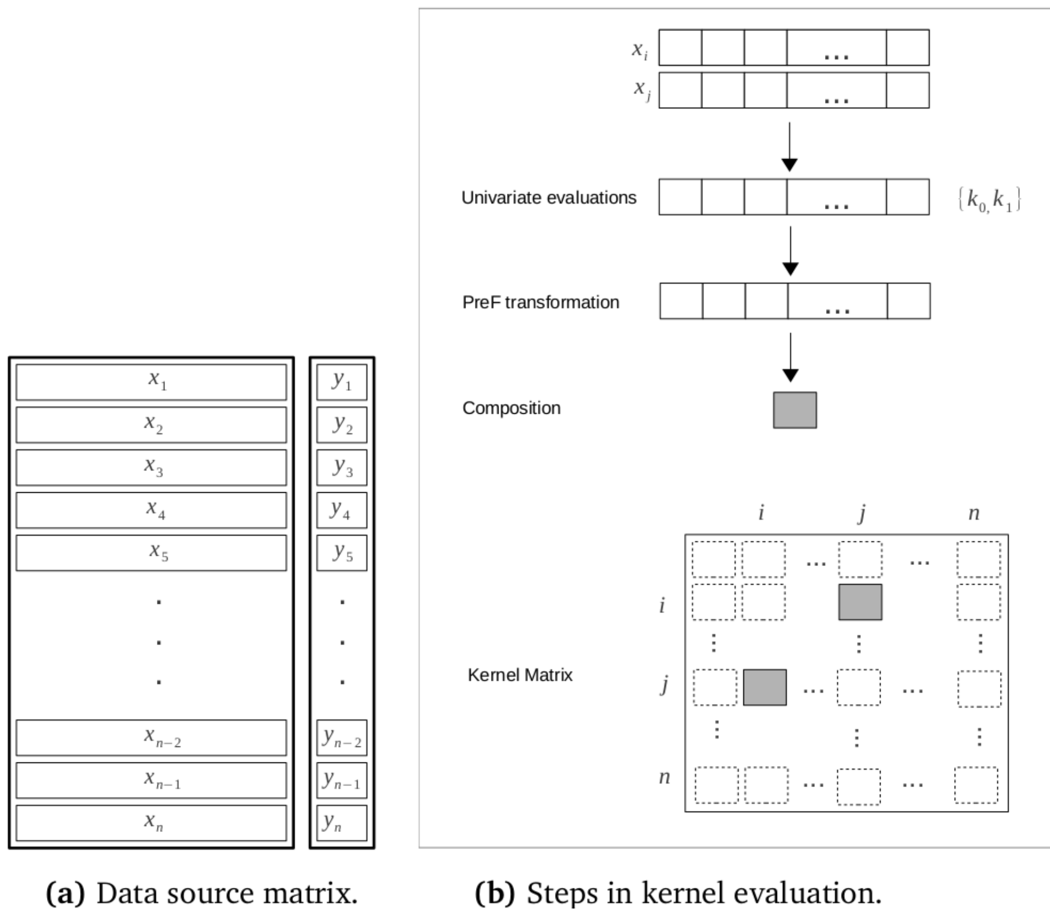
The first version of this function looked like this:

```
   Input: x and y, two data points.
   Result: Kernel evaluation k0(x, y).
 1 d ← length(x)
 2 u ← new vector of dimension d
 3 for (i = 1; i ≤ d; i + +) do
     /* go through all dimensions                          */
 4   if (x[i] = y[i]) then
 5     u[i] ← 1
 6   else
 7     u[i] ← 0
 8   end
 9 end
   /* PreF transformation                                  */
10 up ← PreF(u)
   /* Composition                                          */
11 val ← CompF(up)
12 return val
```

Algorithm 1: First version of kernel function k_0

First, let us introduce some important elements of this script. The algorithm receives two vectors of dimension d. The main loop (line 2) loops through all dimensions comparing the corresponding values. When the loop ends, in u we have a binary vector of length d containing 1 when x and y are equal, and 0 when not. A remarkable fact is that a value is always compared with another value from the same dimension. Then, it is necessary to compute the first transformation function (line 10).

It is called $PreF$ because it is applied before making the composition of vector u. This function receive an unidimensional vector and returns another one of equal size. The Section 4.4 contains the definition of the transformation functions used in this step. Then we need to apply the composition function $CompF$ to integrate the univariate results contained in vector u (which became up), in order to have a single scalar value val. This value is the very first result of evaluating kernel k_0 on data points x and y. Therefore, this will be the value that the kernel matrix K will have for the points x and y.



(a) Data source matrix. **(b)** Steps in kernel evaluation.

Figure 5.1: Kernel evaluation process. This process involves the following steps: 1) the univariate kernel is computed using vectors x_i and x_j , 2) a transformation function is applied on univariate vector, 3) the resulting vector is integrated in a scalar value by means of the Composition function, 4) the kernel matrix is updated with this value in the positions (i, j) and (j, i).

Input: x and y, two data points.
Result: Kernel evaluation $k_1(x, y)$.
Data: pmf, a 2D vector with the mass distribution of the dimensions.

```
1  d ← length(x)
2  u ← new vector of dimension d
3  for (i = 1; i ≤ d; i++) do
       /* go through all dimensions                     */
4      if (x[i] = y[i]) then
5          j ← get the index of modality x[i] in the vector pmf[i]
6          u[i] ← h(pmf[i][j])
7      else
8          u[i] ← 0
9      end
10 end
   /* PreF transformation                               */
11 up ← PreF(u)
12 val ← CompF(up)
13 return val
```

Algorithm 2: First version of kernel function k_1 .

In the Figure 5.1 you can see an schematic representation of the steps executed to evaluate the kernel in two given points. In order to obtain the complete kernel matrix corresponding to a given kernel function, it is necessary to evaluate the kernels on all possible combinations $(\mathbf{x}_i, \mathbf{x}_j)$, with $\mathbf{x}_i, \mathbf{x}_j \in X$ the data matrix. As the kernel matrix is symmetric (as well as the kernel functions), it represents a number of $\binom{n}{2} + n$ kernel evaluations. This basic approach will end up consuming too much time, and it will evolve towards version 2. But before going further, let us briefly examine the first version of kernel k_1.

According to the definition of k_1 (see Section 4.2.1), this kernel performs additional operations related to PMF information. Its implementation is pretty similar to the provided for kernel k_0, except in some tasks related to probabilities.

Notice that we have introduced the use of the new function h, which was already presented in Section 4. It is a function determined by the parameter α:

$$h(z) = (1 - z^{\alpha})^{1/\alpha}$$

A set of different values of α will have to be tested, producing several versions of h function $\{h_1, h_2, ..., h_n\}$, one for each α value. And the same applies to $PreF$ function: it actually represents a family of functions, each one determined by the value of γ parameter. It means that each computation of kernel matrices should be executed as many times as

different combinations of h and $PreF$ functions. And this explosion is another factor that caused the long time-consuming problem mentioned above.

5.1.2 Version 2: Building our kernel matrices

In the previous version of kernels implementation, we defined our kernel function, gave it to `KernLab` package, and ask it to do the rest of the work. `KernLab` did the work through the function *ksvm*, which trains and returns an SVM model with the provided parameters: the data matrix, the response vector, the kernel function and its hyper-parameters, the cost value for the SVM, and others.

One of the first tasks executed by the function *ksvm* is the building of the kernel matrix, which involves the evaluation of the given kernel k_i in all possible combinations (x_i, x_j) of data points provided in the data-source. Certainty, this task will take a long period of time, depending on how expensive is the function k_i. Since this approach did not satisfies our requirements in time consuming, we started to explore other alternatives.

A nice feature provided by $ksvm$ function is the possibility of accepting, not a kernel function, but directly a kernel matrix. In consequence, the task of building the kernel matrix is now our duty, and we will deliver it to $ksvm$ function. The point of this new approach is that we focus on building the kernel matrix in a much more efficient way, using matrix-level operations instead of traditional lopping.

The following paragraphs present how the kernels k_0 and k_1 were re-implemented using matrix-level operations. Let us start with k_0.

The kernel k_0 involves a simple comparison between corresponding dimensions in the data points.

Definition 7. *Univariate (overlap) kernel* k_0

$$k_0(z_i, z_j) = \begin{cases} 1 & \textit{if } z_i = z_j \\ 0 & \textit{if } z_i \neq z_j \end{cases}$$

But the point is that in a dataset of n rows and d dimensions, this comparison has to be done $d \times [\binom{n}{2} + n]$ times. We managed to reduce the execution time using the following idea: the whole process is controlled by a global for loop, from 1 to n. Let be X the data matrix, containing n rows (data points) $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ and d dimensions. The response variable $\{y_1, y_2, ..., y_n\}$ is excluded as it is not necessary for building the kernel. At each iteration i, a matrix $M_i = \{\mathbf{x}_i, \mathbf{x}_i, ..., \mathbf{x}_i\}$ of dimensions $[(n - i + 1) \times d]$ is formed by copying the row x_i in all rows of the matrix M_i. Then the sub-matrix $X_i = \{\mathbf{x}_i, \mathbf{x}_{i+1}, ..., \mathbf{x}_n\}$ is *compared* [1] to M_i, resulting in a binary matrix Eq_i, of same dimensions $[(n - i + 1) \times d]$.

[1]We refer to the equal operator, applied to matrices.

The matrix Eq_i contains all the univariate kernel k_0 evaluations involving the row i. The following step is to apply the transformation function $PreF$ and the composition function $CompF$, which will transform the matrix Eq_i in a vector V_i of size $n - i + 1$. This vector contains the final values of kernel matrix at positions $K_{i,i}, K_{i,i+1}, \ldots, K_{i,n-i+1}$, and $K_{i,i}, K_{i+1,i}, \ldots, K_{n-i+1,i}$ because K is a symmetric matrix.

The following algorithm summarizes this idea.

Data: Data matrix $X_{n\times d}$ with n rows $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$
Result: Kernel matrix
1 Kernel matrix initialization;
2 $K_{n\times n} \leftarrow$ empty matrix;
3 **for** $(i = 1; i \leq n; i++)$ **do**
4 $s \leftarrow n - i + 1$
5 $T_{[i]s\times d} \leftarrow$ matrix $\{\mathbf{x}_i, \mathbf{x}_{i+1}, ..., \mathbf{x}_n\}$
6 $M_{[i]s\times d} \leftarrow$ matrix $\{\mathbf{x}_i, \mathbf{x}_i, ..., \mathbf{x}_i\}$
7 $Eq_{[i]s\times d} \leftarrow M_{[i]} == T_{[i]}$
/* PreF transformation */
8 $EqF_{[i]s\times d} \leftarrow$ PreF$(Eq_{[i]})$
/* Composition */
9 $V_{[i]s\times 1} \leftarrow$ CompF$(EqF_{[i]s\times d})$
/* Now we save the vector $V_{[i]}$ in the matrix $K_{n\times n}$ */
/* Overrides the column i, rows i to n */
10 $K_{n\times n}[i][i:n] \leftarrow V_{[i]}$
/* Overrides the row i, columns i to n */
11 $K_{n\times n}[i:n][i] \leftarrow V_{[i]}$
12 **end**
13 return K

Algorithm 3: Kernel k_0 implementation, version 2.

Notice that the kernel matrix K is filled in simultaneously by rows and columns, as K is a symmetric matrix; the process is executed in n steps, that is, following the main diagonal.

In the case of kernel k_1, the implementation becomes slightly more complex: it is necessary to take the corresponding values from PMF vectors, and put them into an adequate matrix.

Definition 8. *Univariate kernel* k_1

$$k_1(z_i, z_j) = \begin{cases} h(P_Z(z_i)) & \text{if } z_i = z_j \\ 0 & \text{if } z_i \neq z_j \end{cases}$$

As you can see, another new element is the function h, which should be applied on the probability values. The following algorithm presents the approach we followed to implement the kernel k_1 using matrix-schema.

```
Data:
- Data matrix X_{n×d} with n rows {x_1, x_2, ..., x_n}
- pmf, a 2D vector with the mass distribution of the dimensions.
Result: Kernel matrix
/* The function idx returns the index of the value x_ij in the
   corresponding dimension of the pmf 2D array.                    */
/* Kernel matrix initialization                                    */
1  K_{n×n} ← empty matrix;
2  for (i = 1; i ≤ n; i++) do
3  |  s ← n − i + 1
4  |  T_{[i]s×d} ← matrix {x_i, x_{i+1}, ..., x_n}
5  |  M_{[i]s×d} ← matrix {x_i, x_i, ..., x_i}
6  |  Eq_{[i]s×d} ← M_{[i]} == T_{[i]}
   |  /* probability values for vector x_i                          */
7  |  Pv_{[i]1×d} ← {pmf[1][idx(x_i1)], pmf[2][idx(x_i2)], ..., pmf[d][idx(x_id)]}
   |  /* h function                                                 */
8  |  Ph_{[i]1×d} ← h(Pv_{[i]})
9  |  Mph_{[i]s×d} ← matrix {Ph_i, Ph_i, ..., Ph_i}
10 |  Mk_{[i]s×d} ← Mph_i × Eq_i ;          /* Element wise product  */
   |  /* PreF transformation                                        */
11 |  MkF_{[i]s×d} ← PreF(Mk_{[i]})
   |  /* Composition                                                */
12 |  V_{[i]s×1} ← CompF(MkF_{[i]s×d})
   |  /* Now we save the vector V_[i] in the matrix K_{n×n}         */
   |  /* Overrides the column i, rows i to n                        */
13 |  K_{n×n}[i][i : n] ← V_{[i]}
   |  /* Overrides the row i, columns i to n                        */
14 |  K_{n×n}[i : n][i] ← V_{[i]}
15 end
```

Algorithm 4: Kernel k_1 implementation, version 2.

The algorithms 3 and 4 directly return a kernel matrix $K_{n\times n}$, but the results actually corresponding to the selected functions $PreF$, $CompF$. Additionally, kernel k_1 also relies on function h. For that reason, in this second implementation it is necessary to execute the kernel functions as many times as different combinations of functions $PreF$ and $CompF$ (and h for kernel k_1).

In Section 6.5 you can see the different configurations of the experiments executed for each dataset. For GMonks experiments, for example, we tested six values for γ, and eight for α, which represents seven versions of $PreF$ function (plus the identity function, see Section 4.4) and eight of h function. On the other hand, there are only two $CompF$ composition functions (see Section 4.3). In consequence, the kernel k_0 has to be executed 14 times (7×2), and the kernel k_1 112 times $(7 \times 8 \times 2)$. This represents a total

number of 126 kernel matrices.

This set of kernel matrices have to suffer another operation: the $PostF$ function (see Section 4.4). These functions are applied over an already computed kernel matrix, and their purpose is helping to separate the classes contained in the matrix, as well as maintaining the property of being positive semi-definite. Considering that the number of $PostF$ is 2, the total number of working kernel matrices becomes $126 \times 2 = 252$.

Each of these matrices will be used to train one or more SVM models (depending on the particular configuration of the experiment, see Section 6.5), using the function $ksvm$ from `KernLab` package.

Let's turn back reconsidering the algorithm 4. As the total number of computed kernel matrices is considerably high, a clear working-direction to improve the performance of the whole process is to avail of the intermediate computations of the kernel. For example: given a data matrix X, the computed matrix $Eq[i]$ is the same, regardless of the selection of h, $PreF$ and $CompF$. So it is worthless to recompute matrix $Eq[i]$, it should be reused. And the same idea could be applied in the following level: once we have selected the h function, the matrix Mph_i could be profited in the rest of the computation for any combination of $PreF$ and $CompF$. This approach definitely saves a lot of computation time, but conversely, it makes more complex the implementation of the whole process. The following section will present the new implementation that embodies this improvement.

5.1.3 Version 3: Profiting intermediate computations

This version supplies two main changes. The more important consists on the profiting of the intermediate computed matrices. This change involves the *parallel* computation of several matrices: once an h function has been selected and the corresponding matrix Mph_i has been computed, it should not be deleted until all other function combinations are computed, in such a way that the matrix Mph_i is not recomputed again. However, even though this change is clearly an improvement in efficiency, it introduces the requirement of large memory resources to execute the process, due to the large number of matrices being built at the same time.

To solve this problem, now we introduce the second main change of this version: we begin to store the partially-computed matrices in secondary storage. In addition, each final matrix is stored in the form of a list of vectors, that is, $\{V_{[1]}, V_{[2]}, ..., V_{[n]}\}$, not in matrix form $K_{n \times n}$, which would be also right but inefficient, because it is a symmetric matrix.

Naturally, the first impression on adopting this solution is thinking that we are slowing down the whole execution, due to the continuous access to secondary storage. Conversely, this overload is more than compensated by not having to recompute any matrix. So the whole execution is sped up.

Keeping in mind these thoughts, the third version of kernel implementation is presented in Algorithms 5 and 6. At the end of the execution, we obtain two directory structure with all the computed kernel matrices, one file by kernel matrix.

```
Input: Data matrix X_{n×d} with n rows {x_1, x_2, ..., x_n}
ListH ← list of h functions.
ListPreF ← list of PreF functions.
ListCompF ← list of CompF functions.
Result: Kernel matrices, saved in disk.
/* Kernel matrix initialization                                    */
1  K_{n×n} ← empty matrix;
2  for (i = 1; i ≤ n; i++) do
3      s ← n − i + 1
4      M_{[i]s×d} ← matrix {x_i, x_i, ..., x_i}
5      X_{[i]s×d} ← matrix {x_i, x_{i+1}, ..., x_n}
6      Eq_{[i]s×d} ← M_{[i]} == X_{[i]}
7      foreach PreF_sub in ListPreF do
8          EqF_{[i]s×d} ← PreF_sub(Eq_{[i]})
9          foreach CompF_sub in ListCompF do              /* Composition */
10             V_{[i]s×1} ← CompF_sub(EqF_{[i]s×d})
               /* Now save vector V_{[i]} for K_{n×n}                  */
11             Save in disk vector V_{[i]s×1} associated with combination
               PreF_sub, CompF_sub
12         end
13     end
14 end
```

Algorithm 5: Kernel k_0 implementation, version 3.

The corresponding implementation of kernel k_1 is pretty similar; it just includes an additional **for** loop associated with the computation of function h.

5.2 Numerical evaluation of performance

Support vector machines have the beautiful property that training and test functions depend on the data only through the kernel functions $K(\mathbf{x}_i, \mathbf{x}_j)$. Even though it corresponds to a dot product in a space of dimension d_H, where d_H can be very large or infinite, the complexity of computing K can be far smaller. For example, for polynomial kernels $K = (\langle \mathbf{x}_i, \mathbf{x}_j \rangle + 1)^p$, a dot product in feature space H would require order $O\binom{d_L+p-1}{p}$ operations, whereas the computation of $K(\mathbf{x}_i, \mathbf{x}_j)$ requires only $O(d)$ (recall d is the dimension of the source data). It is this fact that allows us to construct

```
Input: Data matrix X_{n×d} with n rows {x_1, x_2, ..., x_n}
ListH ← list of h functions.
ListPreF ← list of PreF functions.
ListCompF ← list of CompF functions.
Result: Kernel matrices, saved in disk.
/* Kernel matrix initialization                                      */
1  K_{n×n} ← empty matrix;
2  for (i = 1; i ≤ n; i++) do
3      s ← n − i + 1
4      M_{[i]s×d} ← matrix {x_i, x_i, ..., x_i}
5      X_{[i]s×d} ← matrix {x_i, x_{i+1}, ..., x_n}
6      Eq_{[i]s×d} ← M_{[i]} == X_{[i]}
7      Pv_{[i]1×d} ← probability values for vector x_i
       /* h function                                                 */
8      foreach h_sub in ListH do
9          Ph_{[i]1×d} ← h_sub(Pv_{[i]})
10         Mph_{[i]s×d} ← matrix {Ph_i, Ph_i, ..., Ph_i}
11         Mk_{[i]s×d} ← Mph_i × Eq_i ;        /* Element wise product   */
12         foreach PreF_sub in ListPreF do
13             MkF_{[i]s×d} ← PreF_sub(Mk_{[i]})
14             foreach CompF_sub in ListCompF do       /* Composition */
15                 V_{[i]s×1} ← CompF_sub(MkF_{[i]s×d})
                   /* Now save vector V_{[i]} for K_{n×n}                */
16                 Save in disk vector V_{[i]s×1} associated with h_sub, PreF_sub, CompF_sub
17             end
18         end
19     end
20 end
```

Algorithm 6: Kernel k_1 implementation, version 3.

hyperplanes in these very high dimensional spaces with a tractable computation.

Another computational bottleneck is the kernel matrix computation. Even though we know that the computation of the kernel matrix involves $\binom{n}{2} + n$ evaluations of $K(\mathbf{x}_i, \mathbf{x}_j)$, the way of computing them makes a big difference. For example, the direct approach is to proceed one-by-one, which leads us to the number of

$$\frac{n^2 + n}{2}$$

evaluations. For small applications it is tolerable, but in large problems this approach becomes unfeasible. Actually, it is common to find alternative approaches in public implementations of SVM. For example, the package `KernLab` contains different implementations of the function `KernelMatrix` [2] depending on the provided kernel function. If the provided kernel function is defined by `KernLab` itself, `KernelMatrix` accelerate the computation by using matrix operations; if the kernel is user-defined, then `KernelMatrix` simply compute K sequentially.

This small trick in package `KernLab` hid a *mystery* in our first experiments with the new kernels: the computing of K took much more time than using standard kernels like polynomial or RBF, which are in principle more costly. When we discovered the reason of this behaviour, we confirm the necessity to compute directly the kernel matrix K, and the result is the evolution described in the previous sections. In particular, observe that in versions two and three, the algorithms contains the whole logic to compute the kernel matrix. They take advantage of two facts: firstly, matrix operations are much more efficient than direct looping, and secondly, in the successive experiments there is no need to completely recompute the matrix K, it is possible to profit part of the previous computations. As an example, consider a data matrix $D_{2000,6}$ with 2000 rows and 6 columns. Computing the kernel matrix using RBF kernel by means of `KernelMatrix` functions is rather fast (time is expressed in seconds):

```
> system.time(K <- kernelMatrix(rbf, D))
 user  system elapsed
3.468   0.108   3.578
```

This is the output of `system.time` function in `R` environment. It returns the time expended in the evaluation of the passed expression. The 'user time' is the CPU time charged for the execution of user instructions of the calling process. The 'system time' is the CPU time charged for execution by the system on behalf of the calling process. However, defining and using the new kernel $k0$ via `KernelMatrix` we obtain this very poor performance:

[2] The function `KernelMatrix` receives a kernel function and a dataset and return the corresponding kernel matrix.

```
%% Algorithm Version 1
%% Using:
%%   - composition-function:    Med.
%%   - transformation-function: Ident.
> system.time(K <- kernelMatrix(k0, D))
   user  system elapsed
110.791   0.092 110.906
```

We obtain very different performance results by computing the kernel matrix using our algorithms.

```
%% Algorithm versions 2 and 3.
%% Using:
%%   - composition-function:    Med.
%%   - transformation-function: Ident.
> system.time(K <- computeMatrix(k0, D))
   user  system elapsed
  1.656   0.040   1.697
> system.time(K <- computeMatrix(k1, D))
   user  system elapsed
  2.724   0.000   2.726
```

The performance of the kernel matrix computation is clearly improved with algorithms Version 2 and 3.

These results are even better when the scope of analysis is extended to the execution of a full battery of experiments, by virtue of the re-utilization of the intermediate computations, as it is described in Version 3 algorithms.

These algorithms may take advantage of parallel processing in several ways. First, all elements of the kernel matrix itself can be computed simultaneously. Second, each element often requires the computation of dot products, involving as many operations as dimensions of training data, which could also be parallelized. Third, a higher level of parallel computation is feasible, by training and testing on different datasets simultaneously. Actually, we have successfully used this last approach to run many experiments simultaneously.

Chapter 6

Experimental Evaluation

In the previous sections we studied the complete formulation of the new kernel functions for categorical variables. In this section we present an experimental evaluation of the behaviour and performance of the proposed kernels functions.

The new kernels are designed to work with categorical variables, and they are able to profit the probabilistic structure of the data to improve the accuracy in the prediction. Therefore, they are meant to obtain good results on datasets in which the probabilistic structure is determinant for predict the response variable.

By *probabilistic structure* we mean the probability mass function (PMF) of all variables forming the dataset. We formulate the hypothesis that, if we have a dataset in which the probabilistic structure is determinant for predicting the response (class) variable, then the new family of kernels should perform better than others. It seems reasonable because the explicit focus of kernels k_1 on PMF information.

As it was presented in Section 4, the probabilistic structure of the data becomes really important for the definition of kernels k_1. But for most of real applications, the PMF of the data is unavailable. In such case we suggest to approximate it using the available data, and it is the approach we use for all our experiments.

A desirable feature of a good predicting-model is the *generalization power*, that is, the ability of the model to predict data never seen before. To achieve this, the model first needs to be *trained* on some dataset. Then, we can ask it to predict the class of a new data-point. If the predictions are quite accurate, we say that the model has acquired *generalization power* over the data.

As you may guess, one of the main issues in the process of training models is finding data in the proper quantity and quality. In real applications usually we are given a finite dataset, and we are asked to build the best possible model with it. But in experimental domains, it is common to work with *artificial* data, and it will be presented latter.

6.1 Data partitioning

It is a common practice in the research community to separate the available data into two partitions: the *training* set and the *testing* set. Normally these partitions are stratified, that is, the classes (the modalities in the response variable) are equally represented in both partitions. As the names suggest, the *training* set is used to train the model (in our case an SVM). Once the model has been trained, we measure its prediction-power on the *testing* set. This measure is called *testing error*, and is regarded as an impartial measure of the model's accuracy, because it is taken from complete unknown data (the testing set). The *training error* measures the mistakes of the model in predicting the training data.

This data-partitioning schema is introduced to overcome, among others, the problem of *over-fitting*: if we train and test our model on the same dataset, we will clearly over-estimate its real accuracy, its actual generalization capability; the model will end up memorizing the dataset, returning good predictions on training data, but pathetic results on new data. In such cases we say that we have over-fitted the model on this data.

To solve this problem we use the data partitioning schema: the model is *tested* on a dataset never seen before, and therefore the testing error will better reflect the actual model's generalization power. But one could think that this measure is not reliable at all, because it strongly depends on the data partitioning: the testing error could be extremely good or extremely bad depending on how we select the partitions! For that reason, the data partitions should be randomly selected, and should be stratified (the class proportions are maintained in all partitions), and the whole process should be executed several times. At the end, we will have a set of testing-error values, which give us a more reliable measure of the model's generalization capability. Usually this set of values are graphically represented in a box-plot chart.

6.2 Cross-validation

The learning algorithms usually have one or more *parameters*, which are a mechanism to give more flexibility to the model to obtain even better results. The SVM algorithm, for example, has the cost parameter C which is the penalization value for each misclassification in a soft-margin SVM formulation. By properly adjusting the value C, the model could return better results than using a single random value. But this gain has a price, because the parameter should be optimized: the model has to be trained and tested using different values for the parameters. The value returning the model with minimum testing-error will be finally selected for that parameter.

This optimization procedure is usually done through a technique called *cross-validation*, which works as follow: Let $P = \{p_1, p_2, ..., p_n\}$ the set of values to be tested for a parameter ρ. We first select a value from P, let say p_1. We split the initial dataset in training and testing sets. The training dataset is again partitioned in N folds, they all randomly-selected and stratified partitions. Then, $N-1$ folds are selected to train the model using the value $\rho = p_1$, and the left fold is used for *validation*, that is, to *validate* how good is the algorithm with the value $\rho = p_1$. All possible folds combinations $\binom{N}{N-1}$ are used to train the model (with the same value $\rho = p_1$), and the left fold for validation. The obtained *validation-errors* are usually summarized in a mean value MVE. The mean validation error is computed for all $p_i \in P$, and the model with minimum MVE value is selected as the best one. Finally, the model is re-trained on the full training dataset using the winner value $\rho = p_i$, and tested on the testing set.

6.3 Artificial datasets

As we have mentioned before, one of the more common issues in the task of training models, is finding good and enough data. For that reason, an alternative that has become very common[1] is the use of *synthetic* or *artificial data*. Although the clearest advantage is that we can generate as much data as we want, the real great point is that we can determine *how* the data is generated, that is, in somehow we can *design* the dataset. This simple fact enable us to develop intensive research on determining the model's behaviour in different data-scenarios. For example, we are given a new classification algorithm and need to determine its real generalization power under different conditions. To do it we can generate a set of synthetic datasets with increasing degree of complexity, or increasing degree of noise, and study the model's performance on each case. At the end we can build a better picture of the new algorithm's behaviour.

We have performed a large number of experiments, using different dataset and different configurations, making all possible combinations of *univariate functions*, *composition functions* and *F functions*. The execution of all these combinations always ends up in a kernel matrix that represents the original data according to the selected components. In the Figure 4.2 (page 29) you can see a graphical representation of these combinations. After computing the univariate kernel, we can either apply a F function or directly integrate these results with a composition function to obtain a kernel matrix. Optionally, on this matrix we can apply again an F function to obtain another kernel matrix.

When you use traditional kernel functions like RBF and polynomial, the

[1] See for example [16].

resulting kernel matrix does not depend on the training/testing data partitions, because the kernel definition $k(x, y)$ only involves the values x and y (see section 3). However, the calculation of the new kernel functions k_1 and k_2 require the knowledge of the PMF for each variable in the dataset (section 4). But most of the cases the probability function is unknown, and need to be estimated from the data. In consequence, a proper experiment design requires that PMF is estimated using only the training data, because this knowledge influences the resulting kernel matrix and the final performance.

For that reason, the very beginning stages of our experiments consist on partitioning the data into training and testing sets. The training set is used to estimate the PMF function, and then the kernel matrix is computed, using all combinations represented in Figure 4.2 (page 29). The resulting set of kernel matrices are then used to train, validate and test the model, but respecting data partitions. The implementation details of the data selection in the kernel matrix are available in section 5.

6.4 Experiments with Datasets

The main objective of this work is to develop a new family of kernel functions for categorical variables. For that reason, all datasets we use for experiments are completely composed of categorical variables, and they all represent classification problems. The main group of experiments are developed using two synthetic datasets, *Poiss* and *GMonks*; in both we control how the data is generated, in order to test the kernel functions in different data scenarios, for example, with increasing-complexity datasets. In this sense, the stronger conclusions about the nature of the new kernels are extracted from this set of experiments. The *Poiss* dataset was designed explicitly for experiments with k_0 and k_1; *GMonks* dataset [17] is a generalization of the classic monks problems [18]. We present the details of the generation process for both of them in the next sections.

On the other hand, we also developed experiments with the real dataset *PromoterGene* [19], which it is available from UCI repository [20].

The Table 6.1 contains the main features of the datasets used for experiments.

Dataset	Dimensions	Modalities	Size	Classes	Type
Poiss	$*$	$*$	$*$	2	Synthetic
GMonks	$*$	[2,4]	$*$	2	Synthetic
Promotergene	58	4	106	2	Real

$*$: *We can select the convenient quantity.*

Table 6.1: Datasets used to perform experiments

The following sections present a summary of each of these datasets.

6.4.1 Artificial Data

Synthetic-Poiss

This artificial dataset was explicitly designed to make experiments with the new kernels k_0 and k_1, and is one of the contributions of this work. It corresponds to a binary classification problem, with a random number of variables determined by a Poisson distribution.

The idea is to generate a dataset with strong probabilistic structure, that is, a dataset where the response variable will hardly depends on the probability mass functions of the input variables. This conformation is not atypical at all; it can be found, for example, in hospitals and health centers, where diseases are normally grouped by families, and patients developing certain illness are susceptible to come down with a related disease.

The trouble comes with the fact that there are common diseases and rare diseases. The following algorithm contains the details of how the data is generated.

```
Data:
d: number of dimensions.
N: number of observations to be generated.
pmod: Poisson distribution parameter. It controls the number of
modalities in the dimensions.
Result: Dataset of N observations and d columns, and the class vector.
        Depending on the value of pmod, the dimensions will have
        more or less modalities.
1  initialization;
2  for (i = 1 to d) do
3      nmod ← generate x ~ Poiss(1, pmod) + 2
4      j ← 1
5      interval ← vector()
6      interval[1] ← 0
7      for (j = 1 to nmod) do
8          prob[j] ← e^(-pmod) / j!
9          interval[j + 1] ← prob[j] + interval[j]
10     end
       /* Now we generate the classes "+" and "-"                    */
11     Class ← [+_1, +_2, ..., +_idx, -_(idx+1), -_(idx+2), ..., -_N]
       /* Now we generate the data according to probabilities
          and classes                                                 */
12     for (r in 1 to cols) do
           /* class +                                                 */
13         auxdata_+ ← maxim index where x>prob[r][]
           /* class -                                                 */
14         auxdata_- ← maxim index where x<prob[r][]
15         datacol ← concatenate auxdata_+ and auxdata_-
16         save datacol into data
17     end
18 end
19 return data
20 return Class
```

Algorithm 7: Synthetic-Poiss data generation.

GMonks

GMonks dataset [17] is a generalization of the classic monks problems [18]. The data corresponds to a problem applied on each chunk of 6 features that take discrete, finite and unordered values (categorical features). Let n the number of chunks (group of 6 features) to be generated.

Data:
d: number of chunks (6x).
N: number of observations to be generated.
Result: Dataset of N observations and $d \times 6$ columns, and the class vector.

1 initialization;
2 **for** $(i = 1$ *to* $N)$ **do**
3 $\quad j \leftarrow 1$
4 $\quad$ **for** $(j = 1$ *to* $d)$ **do**
5 $\quad\quad$ generate a chunk $c_j = \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5, \mathsf{a}_6\}$ of 6 random variables
6 $\quad\quad$ with prob. $\begin{array}{l} \mathsf{a}_1 = \{1,2,3\} \\ \mathsf{a}_2 = \{1,2,3\} \\ \mathsf{a}_3 = \{1,2\} \\ \mathsf{a}_4 = \{1,2,3\} \\ \mathsf{a}_5 = \{1,2,3,4\} \\ \mathsf{a}_6 = \{1,2,3,4,5,6\} \end{array}$
7 $\quad\quad \mathsf{p}_1 \leftarrow (\mathsf{a}_1 = \mathsf{a}_2) \vee (\mathsf{a}_5 = 1)$
8 $\quad\quad \mathsf{p}_2 \leftarrow \#(\{\mathsf{a}_i \mid \mathsf{a}_i = 1\}) \geq 2$
$\quad\quad \mathsf{p}_3 \leftarrow (\mathsf{a}_5 = 3 \wedge \mathsf{a}_4 = 1) \mid (\mathsf{a}_5 \neq 3 \wedge \mathsf{a}_2 \neq 2)$ $\mathsf{ck}_j \leftarrow \mathsf{p}_2 \wedge \neg(\mathsf{p}_1 \wedge \mathsf{p}_3)$
9 $\quad$ **end**
10 $\quad \mathsf{class}_i \leftarrow \#\{\mathsf{ck}_j \mid \mathsf{ck}_j \equiv true, 1 \leq j \leq d\} \geq d/2$
11 $\quad \mathsf{row}_i \leftarrow$ merge vectors $(c_1, c_2, ..., c_d)$
12 **end**
13 return data matrix $(\{\mathsf{row}_1, \mathsf{row}_2, ..., \mathsf{row}_N\})$
14 return class vector $(\{\mathsf{class}_1, \mathsf{class}_2, ..., \mathsf{class}_N\})$

Algorithm 8: GMonks data generation.

Notice that we can generate as many observations as we want (N rows), and also increase the number of dimensions of the dataset ($6 \times d$ columns). In the next sections we describe how to take profit of this flexibility, in order to set up the experiments.

6.4.2 Real Data

PromoterGene [19]

This dataset contains DNA sequences of promoter and non-promoters. Promoters have a region where a protein (RNA polymerase) must make contact and theirs helical DNA sequence must have a valid conformation so that the two pieces of the contact region spatially align. The dataset consists on 106 observations and 57 features that describes the DNA sequence, coded as follows: [a] adenine, [c] cytosine, [g] guanine, [t] thymine. The response variable is categorical, with two modalities: ”+” for a promoter gene and ”-” for a non-promoter gene.

6.5 Experiments design and Results

In this section we present the design of the experiments and the obtained results. Because of the different nature of the datasets, you will see that the design changes considerably from one experiment to another. In GMonks experiments, for example, we try to capture the behaviour of the methods under different conditions. In other datasets, like Promotergene, we simply compare the accuracy and performance of the methods with standard results from RBF kernel.

The following sections presents the full set of experiments and results obtained in this work.

6.5.1 GMonks

The main advantage of using artificial data consists on that we can *configure* the conditions to generate it. More specifically, we define *configurations* by modifying a set of parameters in order to obtain the corresponding dataset. The Table 6.2 we can see a summary of the configurations executed, and the corresponding parameter values. The former four columns defines the dataset used to perform the experiment, that is, the size of the partitions used for training, testing and validation tasks, and the number of dimensions (columns) of the dataset. The following two columns concern the definition of the kernels k_0 and k_1: γ for transformation functions f_1 and α for definition of univariate kernels. Finally, the last column details the values tested for standard $cost$ parameter C in SVM.

	Train	Val.	Test	Dim	γ	α	SVM C
A1	20	10	500	6	$2^{(-3:2)}$	{0.1, 0.2, 0.3, 0.5, 0.7, 0.9, 1, 1.5}	$10^{(-1:2)}$
B1	40	20	500				
C1	80	40	800				
D1	240	120	800				

Table 6.2: Set of configurations for GMonks experiments. The notation $(a : b)$ denotes all integers c such that $a \leq c \leq b$.

Figure 6.1 contains a summary of the results grouped by configurations `A1, B1, C1, D1` and type of kernels k_0, k_1 and rbf. First a box plot for each kernel type is presented, followed by a summary plot of the mean accuracy obtained by them. We can observe how the testing error is decreased as the training set size is increased , as we expected. The results express how the new kernels slightly outperform traditional *RBF* in mean accuracy across all configurations (almost 5%). Considering the mean error, in experiments `A1` k_0 obtains 0.287, k_1 0.28055 and Rbf 0.32275. In all configurations these

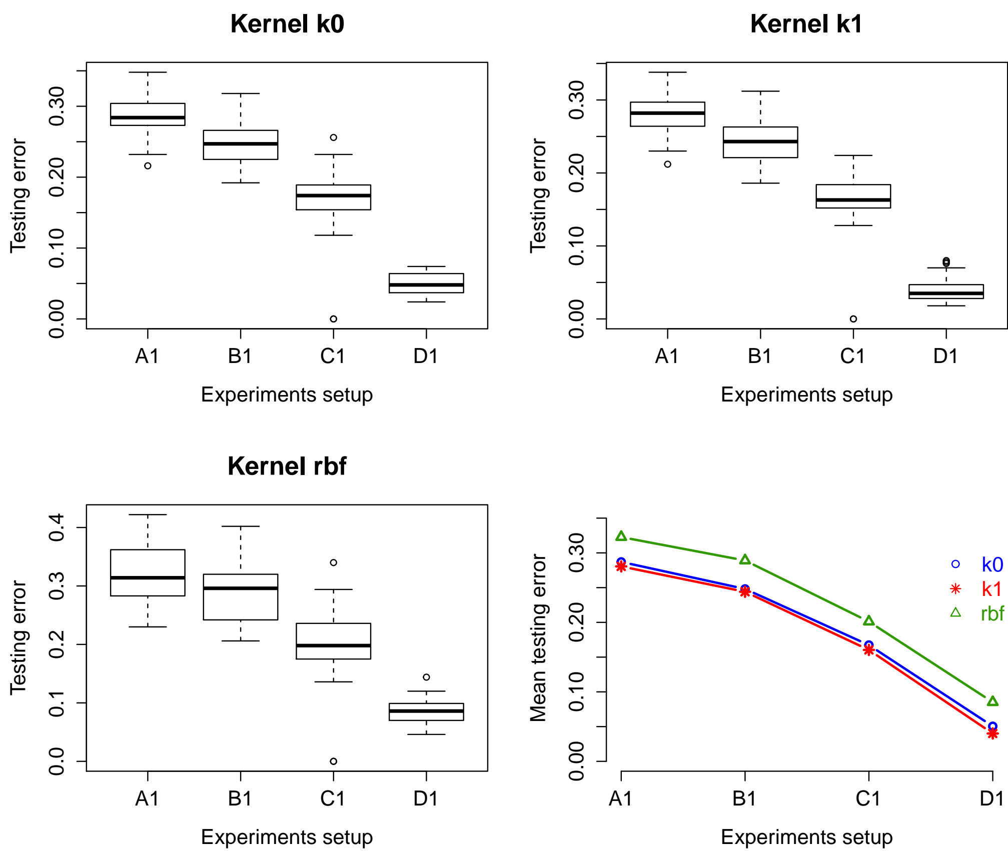


Figure 6.1: GMonks experiments. Testing error on best models of each category. Look at Table 6.2 for details of experiments setup.

relative positions are maintained, changing a little bit in experiments D1 where k_0 obtains 0.05045, k_1 0.0401 and Rbf 0.08515.

The behaviour of kernels $k0$ and $k1$ are not clearly separable, even though $k1$ is better in all cases by almost 1% of mean accuracy. This scenario confirms our initial thoughts about the new kernels: since $k0$ is a particular case of $k1$, this last should always outperform, or at least reach the same accuracy than $k0$. Additionally, considering the standard deviation of the errors, k_1 shows less dispersion than others, with the value of 0.02765349 against 0.02940434 of k_0 and 0.0468816 of Rbf in configuration A1. This proportions are also mainly maintained in the other configurations.

6.5.2 Synthetic-Poiss

The designing principles of Synthetic-Poiss dataset are based on probabilistic information of the variables. Table 6.3 contains the details of all experiments configurations. On the other hand, the experiments results presented in Figure 6.2, are a bit strange. The box plots for k_0 and k_1 are completely plain with constant value of zero, except for two cases in D1 configuration. The behaviour showed by Rbf kernel has a much higher

	Train	Val.	Test	Dim	Pmod	γ	α	SVM C
A1	20	10	500	6	4	$2^{(-3:2)}$	{0.1, 0.2, 0.3, 0.5, 0.7, 0.9, 1, 1.5}	$10^{(-1:2)}$
B1	40	20	500					
C1	80	40	800					
D1	240	120	800					

Table 6.3: Set of configurations for Synthetic-Poiss experiments. The notation $(a : b)$ denotes all integers c such that $a \leq c \leq b$.

variability. In the mean testing error plot (bottom and right) we can compare them all. It is obvious how the new kernels exceed in accuracy to Rbf kernel. However, this improvement is extremely small. For example, in configuration A1, where the difference between methods seems bigger, k_0 and k_1 obtain mean accuracy zero, and Rbf obtain 0.0033, which is more than 99% of accuracy! This situation repeated in configurations B1 and C1, where again the kernels k_0 and k_1 reach the zero value in mean error, while Rbf obtains 0.00055 and 0.00025 respectively. In configuration D1, Rbf obtains 0.00125, k_0 0.00035 and k_1 0.0002. Notice that in this problem, the Rbf kernel gives almost no range of improvement.

This fact led us to check all over again, and we found the following fact: the classification problem provided by Synthetic-Poiss algorithm was extremely easy to solve. In spite of these results, they support the ideas developed in previous chapters: the new kernels are able to take profit of the probabilistic structure embedded in the data.

Until now we have described experiments with artificial datasets. The following sections present real data experiments. We no longer use different configurations, because we can not generate new instances of the problem. We instead use the techniques described in Sections 6.1 and 6.2.

6.5.3 PromoterGene

As it was introduced in Section 6.4.2, PromoterGene dataset contains only 106 observations and 57 features. In these cases, it is recommended to execute the experiments following the common practices of data partitioning and cross-validation. In this sense, we start with a general partition, taking 2/3 of the data for training and the remaining for testing. These partitions are randomly taken always preserving the class proportions.

Then, using the training set we ran 10-folds cross validation to optimize the hiper-parameters of the models(α and γ). After that, we trained again the model on the whole training set but using the optimal hyper-parameters. Then, we tested this model on testing-data obtaining its testing-error. This process was repeated 40 times, and the results are summarized in the Figure 6.3.

The performance of k_0 and k_1 clearly outperform the one obtained by rbf. In fact, the plot presents an strong evidence that $k1$ (with mean error 0.0382) solves this problem better than Rbf (0.2125) and k_0 (0.06176471). Additionally, the kernel k_1 also shows a lower dispersion than all other kernels, having a standard deviation of (0.02993501), while Rbf has 0.1119447 and k_0 0.036359.

On the other hand, although kernels k_0 and k_1 are quite similar in accuracy, it is evident that k_1 does slightly better than k_0, which again confirm our theoretical hypothesis.

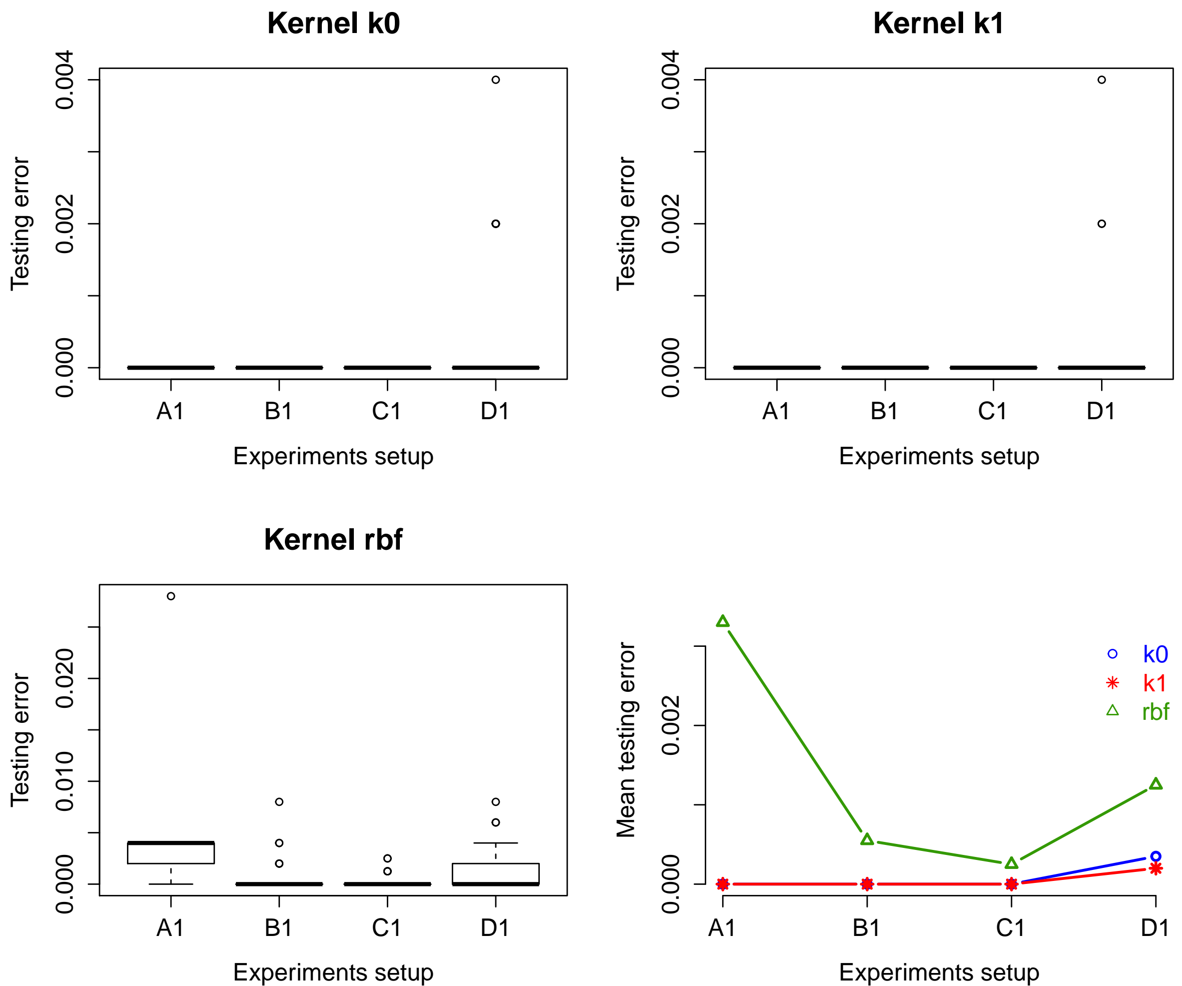


Figure 6.2: Synthetic-Poiss experiments. Testing error on best models of each category. Look at Table 6.3 for details of experiments setup.

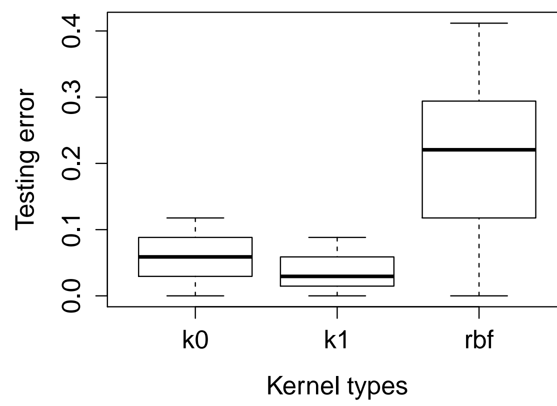


Figure 6.3: Testing error on PromoterGene dataset

Chapter 7

Conclusions and After Thoughts

This thesis deals with the development of new kernels for categorical variables. The starting idea was the hypothesis that the kernels usually applied to categorical problems, like k_0 and Rbf, do not extract all the information from the data. This deficiency is more accused on problems with only categorical variables, because the only meaningful operation in such type of data is simple comparison. The common approach in these cases consists on recoding the source data, making it suitable for applying the common numeric kernels.

The Support Vector Machines are a very versatile and adaptable framework, which allow to develop and experiment with many different classifiers by changing the kernel function. Based on its flexibility, we defined a new family of kernels k_1 oriented to take profit of the probabilistic structure hidden in the data.

With the aim of examine the behaviour and performance of these new kernels, we developed an experimental study, exploring all variants of the proposed kernels and using real and artificial datasets. The experimental results support our initial intuition, showing that k_1 outperforms in accuracy to k_0 and Rbf. This means, first, that actually there is an information that is not used by k_0 and Rbf, and second, that there are ways of profiting this information. In particular, where the results (mean errors and their variance) in standard kernels are already good, k_1 improves a little with respect to k_0, and both of them outperform to Rbf. However, where the results have still margin of improvement (as in the `PromoterGene` dataset), then the benefit of using k_1 is higher.

The main contribution of this work consist on the definition of new kernels that capture this information, and the formulation of a modular framework that facilitate their creation and extension.

7.1 Future Work

Not all our ideas have been developed into this work, so it would be interesting to continue exploring in these directions:

- Extending the experimental study by applying the new kernels in more problems. This would give a better understanding of the real potential of the proposed method.
- We have experimented with a single family of *inverting functions*. A possible extension consists on defining and testing more functions with different behaviour.
- Another possible extension consists on modifying the kernel k_1 to return a non-zero value when $x \neq y$. This value would be determined by another inverting function g.

  $$k_2(z_i, z_j) = \begin{cases} h(P_Z(z_i)) & \text{if } z_i = z_j \\ g(P_Z(z_i), P_Z(z_j)) & \text{if } z_i \neq z_j \end{cases}$$

  As you can see, the kernel k_2 would be a generalization of k_1 (in which the function g always returns zero). This new kernel has even more flexibility, because it has the possibility of returning a non-zero value (similarity) for two different objects.

  A possible application would be the following. Consider the example of normal and strange diseases (Section 4.2.1). Given two patients with *different* diseases, this extension would return a high value (similarity) if both diseases are *equally rare* (the probabilities $P_Z(z_i)$ and $P_Z(z_j)$ are very small and similar). This means that these two persons have something in common, because both of them have developed strange (although different) illnesses. On the contrary, if $P_Z(z_i)$ and $P_Z(z_j)$ are high and different, the k_2 would return a very low value.

Bibliography

[1] C. Cortes and V. Vapnik, "Support vector networks," *Machine Learning*, vol. 20/273–297, 1995.

[2] V. Vapnik, "The nature of statistical learning theory," 1995.

[3] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data Mining and Knowledge Discovery*, vol. 2, 121-167, 1998.

[4] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. 2004.

[5] G. Mendel, "Experiments in plant hybridization," 1865. Available as `http://www.esp.org/foundations/genetics/classical/gm-65.pdf`.

[6] C.-L. Huang, M.-C. Chen, and C.-J. Wang, "Credit scoring with a data mining approach based on support vector machines," *Expert Systems with Applications*, vol. 33, no. 4, pp. 847 – 856, 2007.

[7] R. F., "The perceptron - a perceiving and recognizing automaton," *Technical Report 85-460-1*, 1957.

[8] R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *Journal of Computational and Graphical Statistics*, vol. 5, no. 3, pp. 299–314, 1996.

[9] B. E. Boser, I. M. Guyon, and V. Vapnik, "A training algorithm for optimal margin classifiers," *Annual Workshop on Computational Learning Theory*, pp. 144–152, 1992.

[10] M.-F. Balcan, A. Blum, and N. Srebro, "A theory of learning with similarity functions," *Machine Learning*, vol. 72, pp. 89–112, 2008.

[11] F. Leisch, "Creating r packages: A tutorial," tech. rep., Technical report, Ludwig-Maximilians-Universität München, and R Development Core Team, 2009.

[12] E. Dimitriadou, K. Hornik, F. Leisch, D. Meyer, A. Weingessel, and M. Leisch, "The e1071 package," 2007.

[13] A. Karatzoglou, A. Smola, and K. Hornik, "kernlab: Kernel-based machine learning lab," 2009.

[14] T. Hastie, "svmpath: The svm path algorithm," 2006.

[15] A. Karatzoglou, D. Meyer, and K. Hornik, "Support vector machines in r," 2005.

[16] D. Jeske, B. Samadi, P. Lin, L. Ye, S. Cox, R. Xiao, T. Younglove, M. Ly, D. Holt, and R. Rich, "Generation of synthetic data sets for evaluating the accuracy of knowledge discovery systems," pp. 756–762, 2005.

[17] L. A. B. Muñoz and F. F. González-Navarro, "Review and evaluation of feature selection algorithms in synthetic problems," *CoRR*, vol. abs/1101.2320, 2011. Available as http://arxiv.org/abs/1101.2320.

[18] S. B. Thrun, J. Bala, and E. B. et al., "The monk's problems a performance comparison of different learning algorithms," tech. rep., 1991.

[19] "Promoter gene uci dataset," 1990. Available as http://archive.ics.uci.edu/ml/datasets/Molecular+Biology+(Promoter+Gene+Sequences).

[20] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. http://archive.ics.uci.edu/ml/.

Appendix A

Demonstrations

A.1 Preliminaries

Lemma 1. *For all $a, b > 0$,*

$$\frac{a}{b} + \frac{b}{a} \geq 2$$

PROOF. Without loss of generality, we assume $a \geq b$; then we write $a = b+\alpha$, with $\alpha \geq 0$ and:

$$\frac{a}{b} + \frac{b}{a} = \frac{b+\alpha}{b} + \frac{b}{b+\alpha} = \frac{(b+\alpha)^2 + b^2}{b(b+\alpha)} = \frac{2b^2 + 2b\alpha + \alpha^2}{b^2 + b\alpha} = 2 + \alpha^2 \geq 2$$

Lemma 2. *For all $a_1, \ldots, a_k \in \mathbb{R}$,*

$$\sum_{l=1}^{k} a_l^2 \geq 2 \sum_{l_1=1}^{k} \sum_{l_2>l_1}^{k} a_{l_1} a_{l_2}$$

is equivalent to

$$\left(a_1 - \sum_{l=2}^{k} a_l \right)^2 \geq 0$$

and therefore holds true.

PROOF. We prove this by induction on k.

- For $k = 2$ the equivalence is immediate.
- For $k \geq 2$ we want to show that:

$$\sum_{l=1}^{k+1} a_l^2 - 2\sum_{l_1=1}^{k+1}\sum_{l_2>l_1}^{k+1} a_{l_1}a_{l_2} = \left(a_1 - \sum_{l=2}^{k+1} a_l\right)^2$$

Indeed,

$$\sum_{l=1}^{k+1} a_l^2 - 2\sum_{l_1=1}^{k+1}\sum_{l_2>l_1}^{k+1} a_{l_1}a_{l_2}$$

$$= \sum_{l=1}^{k} a_l^2 + a_{k+1}^2 - 2\left(\sum_{l_1=1}^{k}\sum_{l_2>l_1}^{k+1} a_{l_1}a_{l_2} + \sum_{l_2>k+1}^{k+1} a_{l_1}a_{l_2}\right)$$

$$= \sum_{l=1}^{k} a_l^2 + a_{k+1}^2 - 2\left(\sum_{l_1=1}^{k}\sum_{l_2>l_1}^{k} a_{l_1}a_{l_2} + \sum_{l_1=1}^{k} a_{l_1}a_{k+1}\right)$$

(by induction hypothesis)

$$= \left(a_1 - \sum_{l=2}^{k} a_l\right)^2 + a_{k+1}^2 - 2a_{k+1}\sum_{l=1}^{k} a_l$$

$$= \left(a_1 - \sum_{l=2}^{k+1} a_l\right)^2$$

A.2 Kernel k_0

Theorem A.1. *The kernel matrix generated by k_0 in Definition (2) is positive semi-definite (p.s.d.).*

PROOF. Given the nature of the kernel, we develop a case analysis on the equality structure of the points $\{x_1, \ldots, x_N\}$.

1. Suppose the points $\{x_1, \ldots, x_N\}$ are all different; then

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j k_0(x_i, x_j) = \sum_{i=1}^{N} c_i c_i k_0(x_i, x_i) = \sum_{i=1}^{N} c_i^2 \geq 0$$

2. Suppose the points $\{x_1, \ldots, x_N\}$ are all equal; then

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j k_0(x_i, x_j) = \sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j = \left(\sum_{i=1}^{N} c_i\right)^2 \geq 0$$

3. Suppose there are exactly two equal points (the rest of the points are different from this pair and from one another); let i_1 and i_2 be the indexes of this pair (note that $x_{i_1} = x_{i_2}$ but in general $c_{i_1} \neq c_{i_2}$); then

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j k_0(x_i, x_j) = \sum_{i=1}^{N} c_i^2 + 2c_{i_1}c_{i_2}$$

Therefore we require that $\sum_{i=1}^{N} c_i^2 \geq 2|c_{i_1}||c_{i_2}|$. Since, in the worst possible case, all the c_i apart from $c_{i_1}c_{i_2}$ may be zero, we consider the condition[1]

$$c_{i_1}^2 + c_{i_2}^2 \geq 2|c_{i_1}||c_{i_2}|$$

Which is equivalent to:

$$|c_{i_1}||c_{i_1}| + |c_{i_2}||c_{i_2}| \geq 2|c_{i_1}||c_{i_2}|$$

Which in turn is equivalent to:

$$\frac{|c_{i_1}|}{|c_{i_2}|} + \frac{|c_{i_2}|}{|c_{i_1}|} \geq 2$$

which holds by direct application of lemma (1).

4. Suppose now there is a group of three equal points (the rest of the points are different from these and from one another); let i_1, i_2, i_3 be the indexes of this trio. An analogous reasoning leads to:

$$c_{i_1}^2 + c_{i_2}^2 + c_{i_3}^2 \geq 2(|c_{i_1}||c_{i_2}| + |c_{i_2}||c_{i_3}| + |c_{i_1}||c_{i_3}|)$$

In general case we will have a group of k equal points with indexes $i_1, \ldots, i_k$ such that $x_{i_1}, \ldots, x_{i_k}$ (with different coefficients $c_{i_1}, \ldots, c_{i_k}$); the required condition is:

$$\sum_{l=1}^{k} c_{i_l}^2 \geq 2\sum_{l_1=1}^{k}\sum_{l_2>l_1}^{k} |c_{l_1}||c_{l_2}|.$$

By lemma (2) with $a_i \equiv |c_i|$, this inequality is equivalent to:

$$\left(|c_{i_1}| - \sum_{l=2}^{k} |c_{i_l}|\right)^2 \geq 0$$

[1] In addition, if either of c_{i_1}, c_{i_2} is zero, the inequality is trivially met; this allows to assume c_{i_1}, c_{i_2} to be different from 0.

that obviously holds true.

5. If there is another group of repeated elements, a similar argument leads to affirm that their net contribution to the overall sum is again non-negative (it was assumed to be zero up to now). An iteration of this argument for every group of repeated elements leads to the desired result.

A.3 Kernel k_1

Theorem A.2. *The kernel matrix generated by k_1 in Definition (3) is positive semi-definite (p.s.d.).*

PROOF. We shall prove a more general result:
Let $h : X \to (0,1)$ any function; then

$$k_h(x,y) = h(x)\mathbb{I}_{\{x=y\}},\ x,y \in X$$

is a kernel in X, where

$$\mathbb{I}_{\{z\}} = \begin{cases} 1 & \text{if } z \text{ is true} \\ 0 & \text{if } z \text{ is false} \end{cases}$$

Given the nature of the kernel, for the sake of clarity, we develop again a case analysis on the equality structure of the points $\{x_1, \dots, x_N\}$.

1. Suppose the points $\{x_1, \dots, x_N\}$ are all different; then

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j k_h(x_i,x_j) = \sum_{i=1}^{N} c_i c_i k_h(x_i,x_i) = \sum_{i=1}^{N} c_i^2 h(x_i) \geq 0$$

2. Suppose the points $\{x_1, \dots, x_N\}$ are all equal; then

$$\sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j k_h(x_i,x_j) = \sum_{i=1}^{N}\sum_{j=1}^{N} c_i c_j h(x_i) h(x_j) = \left(\sum_{i=1}^{N} c_i h(x_i)\right)^2 \geq 0$$

3. Suppose there are exactly two equal points (the rest of the points are different from this pair and from one another); let i_1 and i_2 be the indexes of this pair. Reasoning in a way analogous to the previous proof, we arrive at:

$$\sum_{i=1}^{N}\left(c_i\sqrt{h(x_i)}\right)^2 \geq 2|c_{i_1}\sqrt{h(x_{i_1})}|\,|c_{i_2}\sqrt{h(x_{i_2})}|.$$

For a group of k equal points with indexes $i_1, \ldots, i_k$ such that $x_{i_1}, \ldots, x_{i_k}$ (with different coefficients $c_{i_1}, \ldots, c_{i_k}$); this expression generalizes to:

$$\sum_{l=1}^{k} \left(c_{i_l} \sqrt{h(x_{i_l})} \right)^2 \geq 2 \sum_{l_1=1}^{k} \sum_{l_2>l_1}^{k} |c_{l_1} \sqrt{h(x_{l_1})}|\, |c_{l_2} \sqrt{h(x_{l_2})}|.$$

By lemma (2) with $a_i \equiv |c_i \sqrt{h(x_i)}|$, this inequality is equivalent to:

$$\left(|c_{i_1} \sqrt{h(x_{i_1})}| - \sum_{l=2}^{k} |c_{i_l} \sqrt{h(x_{i_l})}| \right)^2 \geq 0$$

which is true.

4. In the general case, there will groups of two or more equal points (equal to one another and different across groups). This case is analogous to that of the previous proof.