UNIVERSITAT POLITECNICA DE CATALUNYA

Multivariate Kernel Functions for Categorical Variables

Carlos García Márquez

A thesis submitted for the degree of Informatics Engineer
Directed by Lluís Belanche Muñoz

Facultat d'Informatica de Barcelona
Departament de Llenguatges i Sistemes Informatics

December 2014

UNIVERSITAT POLITECNICA DE CATALUNYA

Abstract

Facultat d'Informatica de Barcelona
Departament de Llenguatges i Sistemes Informatics

Informatics Engineer

by Carlos García Márquez

This project investigates kernels specially designed for their use on categorical data, and introduces a new family of kernel functions grounded on the use of the probabilistic structure of the data. This new family is based on the generalisation of existing kernels for binary variables -where the inputs belong to 0, 1- to a probabilistic framework where the inputs belong to (0, 1).

Project Information

Title: **Multivariate Kernel Functions for Categorical Variables**

Name: **Carlos García Márquez**
Degree: **Enginyeria en Informàtica**
Credits: **37,5**
Director: **Lluís Belanche Muñoz**
Department: **Llenguatges i Sistemes Informtics (LSI)**

Examination board members

President: **Karina Gibert Oliveras**

Signature:

Vocal: **José M. Llaberia Griñó**

Signature:

Secretary: **Lluís Belanche Muñoz**

Signature:

Qualification

Numeric qualification:
Descriptive qualification:

Date:

Acknowledgements

I would like to thank my tutor Lluís for his guidance, and my family for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Recording, storing, and moving information is easier and cheaper than it has ever been, computers are everywhere, storage capacity keeps increasing, prices get lower, and more and more devices are connected to communication networks... today, obtaining and storing data is becoming less of a problem and the focus is shifting to finding better ways to process and use that data efficient and effectively.

There has been a recent surge in popularity in all the disciplines within computer science that deal with information, and fields like machine learning and data mining are in the spotlight, even thought they have been studied for some time. In light of this ongoing shift in technology and the surge of popularity in these fields, I believe that they will probably have to deal with some of the most interesting challenges that lay ahead in the near future. I see this project as an opportunity to learn new concepts, skills and tools related to what I studied at the Faculty of Informatics, and a way to get immersed in a very interesting subject.

1.2 Outline

This project deals with the development of kernel functions. In the context of machine learning, kernel methods have been in use since the middle nineties. They allow the use of techniques such as Support Vector Machines and Principal Component Analysis in ways that would otherwise be computationally prohibitive. The main goal of this project is to develop kernel methods that can be applied to categorical data, that is data which is qualitative in nature, rather than quantitative.

Before going into detail about kernel methods though, it would be better to have a 10,000 feet view of the scope of the whole project. The best way to do that is a list of objectives:

1. Study the state of the art in kernel functions for categorical variables.
2. Define new kernels that take advantage of probabilistic information in the data.
3. Test the performance of the different kernels with specially designed data sets.
4. Test the performance of the different kernels with data from real-life problems.
5. Analyze and compare the results of the different kernels on the different data sets.

Additionally, at the end of the project the resulting system should be:

1. Easily tweaked, extend or derive the kernels to explore the results.
2. Easily reproduce the experiments to replicate the results.

It shows from these objectives that this project is oriented towards research, but there are also technical challenges involved.

This document will cover both the theoretical side of the project and the implementation details.

The first chapters will be devoted to explaining the theoretical foundations and defining the various kernel functions, as well as analysing their application and testing the obtained results.

The second half of the document will be devoted to the technical side of the project. As with any project in Informatics Engineering, it is necessary to document the most relevant aspects in the process of analysing, designing and implementing the kernels and the tests used to measure their performance.

Chapter 2

The Basics of Kernel Methods

There are a number of books[1] and articles that provide an excellent description of kernel methods and their mathematical properties. Instead of parroting yet another explanation that doesn't add much to the subject, they will be introduced with a few simple and very straightforward examples of an algorithm that uses kernel methods. They will show where they are used, why they are useful, and an overview of how they work, to provide the necessary context to understande the kernels that will be introduced later.

2.1 Support Vector Machines

This project will focus on support vector machines (SVMs) which is a model is used in classification problems. The SVN algorithm uses kernels at its core, but it's worth pointing out that there are many other algorithms that make use of kernels: principal component analysis, canonical correlation analysis, ridge regression, and more. A kernel that is useful in one can potentially be useful in the others too.

2.1.1 A First Example of an SVM

For simplicity, let us consider a binary classification problem, where there is a list of data points with a finite number of dimensions (two in our example) and each of them is labeled one of two possible classes[2]. This labeled set is usually called the *training* set. The training set for the current example is plotted in Figure 2.1.

[1] The books Learning with Kernels [1], Kernel Methods for Pattern Analysis [2], and Kernel methods in computational biology [3] are all very recommendable introductions to the subject.

[2] It is possible to use SVM with three or more classes, but we consider the simplest scenario.

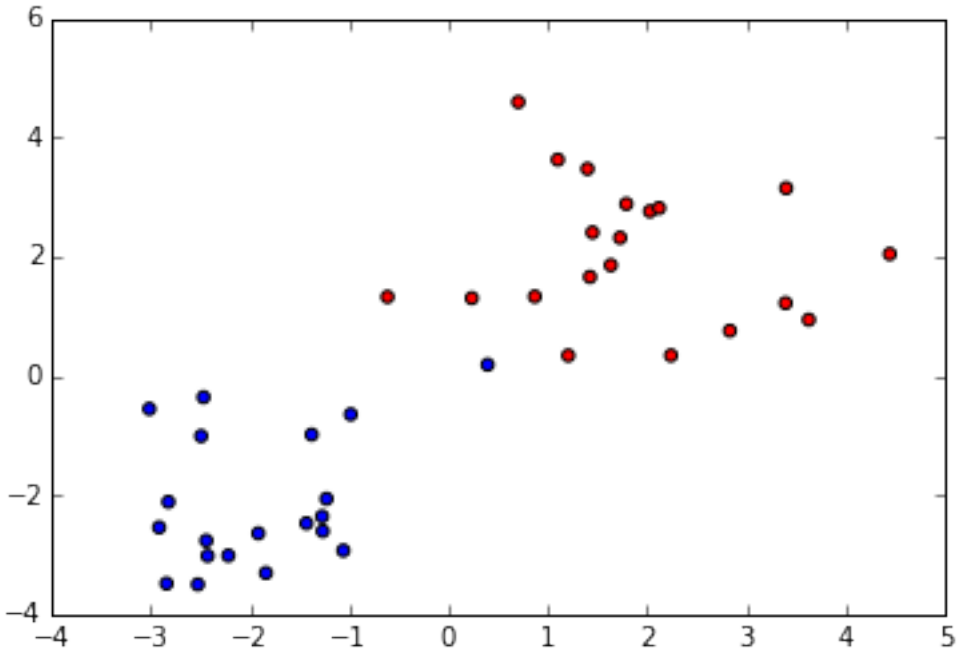


FIGURE 2.1: A linearly separable training set.

The problem that SVM try to solve is that of classifying new, unlabeled points into one of the two known classes, based on some criteria that can be inferred from the labeled data points in the training set. There are many implementations of SVM readily available, which means that one can just pick any library and easily see the solution obtained by the algorithm.

One can take the Linear Support Vector Classifier available in scikit-learn (a Python framework), create an instance, and call the *fit* method with training set from the example as an argument. This will create a classifier that implements the SVM algorithm and fit it to the example data.

```
clf = svm.SVC(kernel='rbf')
clf.fit(training_set, training_classes)
```

Now, by using the *predict* method of the classifier it is possible to query any point and get a back prediction of its label. Here is an example querying for two points (2, 2) and (-2, -2):

```
>>> clf.predict([(2, 2), (-2, -2)])
[1, 0]
```

The returned values are the lables for each point, 1 and 0 respectively.

When the fit function was called with the training set, the algorithm determined a way to classify points in space. The way SVMs do this is by finding the parameters of a line that better separates the two classes in the data set. This line is illustrated in Figure 2.2.

Let us explain what is shown in the figure. The solid line crossing the dataset is what is called the decision boundary, the side of the line on which a point falls is what determines the label it will be assigned. The dashed lines are called the margins, the SVM algorithm tries to find the decision boundary in a way that maximizes the width of these margins, as this usually gives a better classifier. Because of this maximizing of the margins is that SVM are often called wide margin classifiers. The circled points are called support vectors, these are the points that define the decision boundary and the margin. The other points in the dataset do not actually affect where the decision boundary falls.

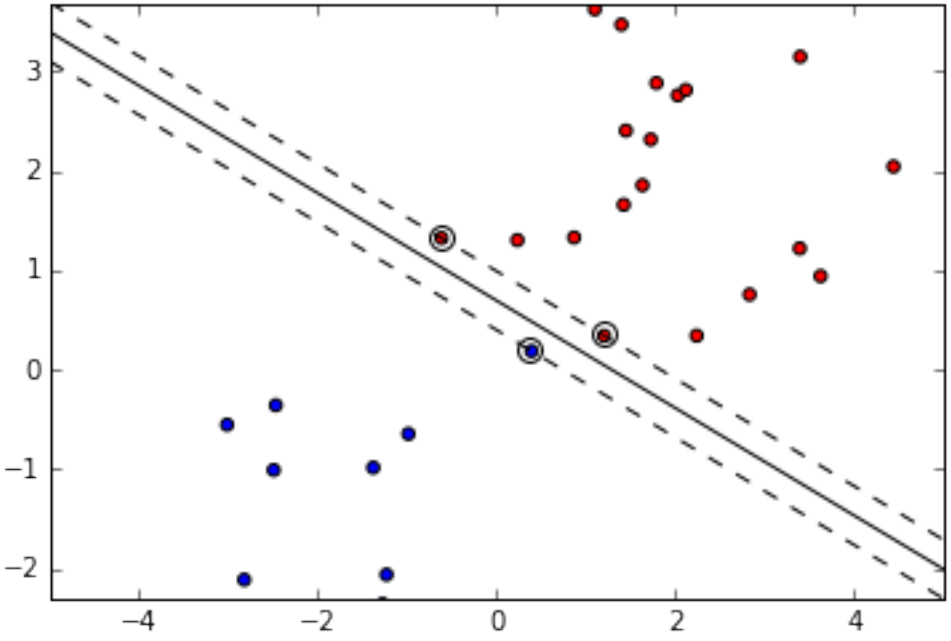


FIGURE 2.2: The SVM margin for the linearlny separable dataset.

One thing to point out is that the decision boundary is actually a hyperplane. In this example in two dimensions it is a line, but in two dimensions it would be a plane, and in general, in n dimensions it will be an $n-1$ subspace. The purpose of the SVM is to find the parameters of the hyperplane with the widest margin, so in the end an SVM is an algorithm that solves an optimization problem.

2.1.2 Non Linear Example

In practice, having a a dataset with two classes that can be directly separated by a hyperplane is unusual, most classification problems are more complex than that. Let us now look at a more interesting example, like the dataset shown in Figure 2.3.

It is apparent that this new dataset cannot be separated by simply drawing a line. However, it seems that it could easily separated with a circle. But the thing is that

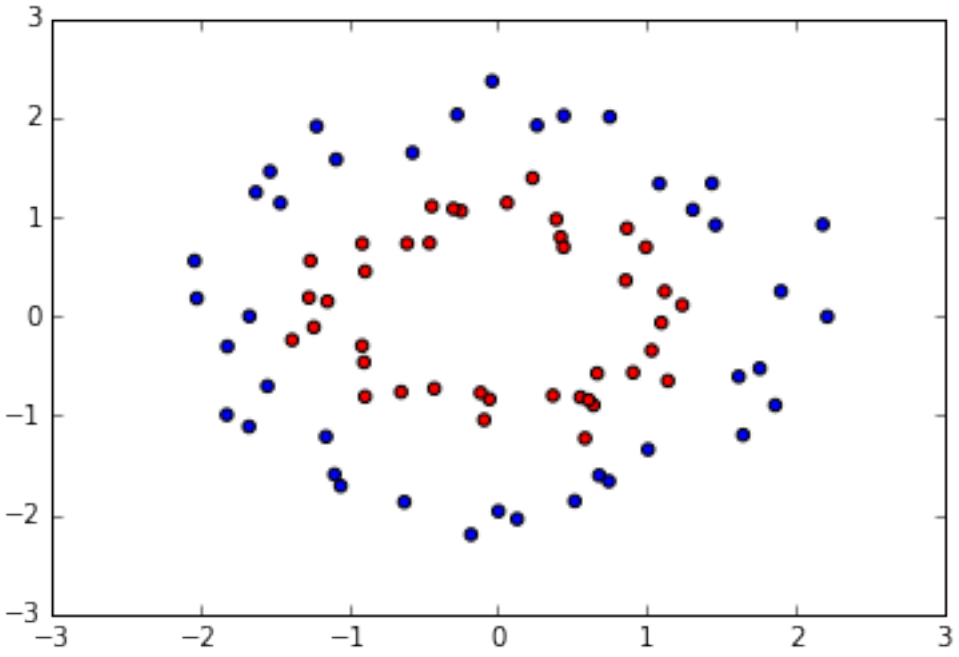


FIGURE 2.3: A not linearly separable dataset.

the optimization problem that the SVM solves is always finding a hyperplane. So what can be done when a dataset is not linearly separable? The solution is to use a mapping function:

$$\mathbf{x} \to \phi(\mathbf{x})$$

that transforms the points in the dataset and projects them into a new space where they are linearly separable. This new space is usually called the *feature space*, while the original one is called *input space*.

Back to the example. Let us transform the points and move them in to a three-dimensional space, by giving each of them a z_i coordinate with the value $x_i^2 + y_i^2$. The visualization of the dataset in this new space is shown in Figure 2.4.

Suddenly, the dataset that was not linearly separable in two dimensions is separable in the new space by a plane, which means that now an SVM can be used to find the hyperplane (which is just a plane in 3D) that separates the two labeled groups, as shown in Figure 2.5.

On a high level, that is what SVMs do. Most implementations of SVM use what is called a soft margin, and use a parameter C to allow some tolerance for outliers when finding the margin, but the idea of finding a decision boundary with the widest margin is the same for both soft margin and hard margin SVMs.

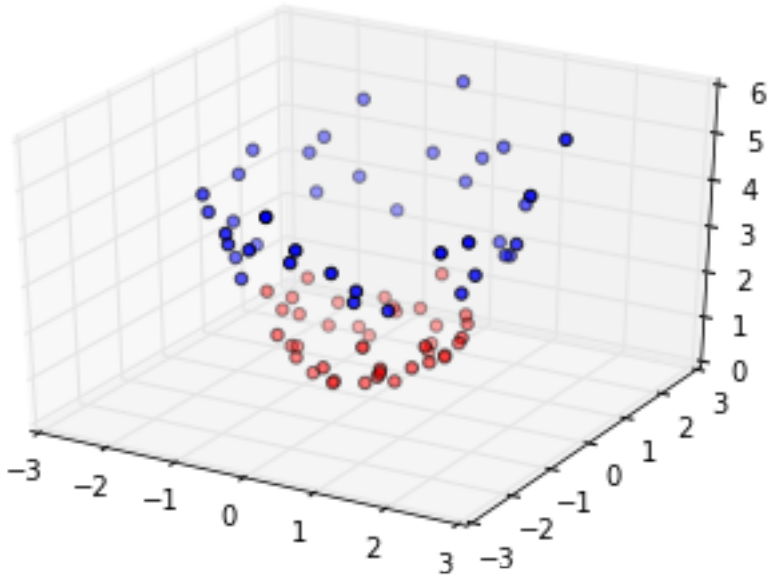


FIGURE 2.4: The dataset with the new z dimension.

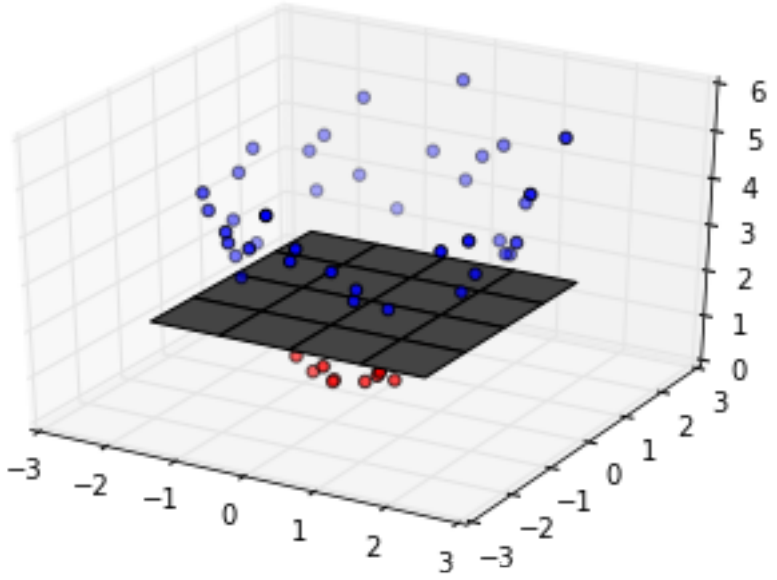


FIGURE 2.5: A hyperplane dividing the dataset.

2.2 Kernel Functions And The Kernel Trick

Most SVM implementations allow the user to choose between different available *kernels*. The most common are the linear, polynomial, RBF and sigmoid kernels, though there are many others. Most implementations also allow you to provide a custom function, and the SVM will work, as long as the function provided is a valid kernel.

But what are these *kernels*? A kernel is a function $K : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$ usually expressed as:

$$K(\mathbf{x_i}, \mathbf{x_j}) = \phi(\mathbf{x_i}) \cdot \phi(\mathbf{x_j})$$

that is, a function that takes two elements in input space, maps them to feature space and returns their dot product.

Kernels are a vital part of SVM because of the way the algorithm works. To find the decision boundary, the SVM solves a quadratic optimization problem[3] on what is called the *Gram matrix*. This is an $M \times M$ matrix obtained by calculating the dot product between all possible pairs of points in the feature space. Notice that computing the matrix is already a $\mathcal{O}(n^2)$ algorithm, which is quite expensive, and having to map each data point to a feature space before applying the dot product makes it only more so, specially when the original and feature spaces often have a high number of dimensions.

Here is where kernels come into play, thanks to something called the *kernel trick*. The trick is to say that any algorithm that can be expressed in terms of only dot products can be computed using solely kernel functions. SVMs is one of such algorithms, so are principal component analysis and others. The great advantage of using kernels is that they compute the dot product without having to explicitly map the points to feature space, which is done implicitly by the kernel. This is not only much more efficient in most cases, but it even allows to use kernels whose associated feature space is infinite-dimensional, which would be infeasible to compute. It also allows to define kernels between structures such as strings, categorical variables and even trees or graphs.

Evidently, kernel functions must have certain properties. A kernel function is equivalent to computing an inner product in a Hilbert space or inner product space. So in order to prove that a function is a valid kernel, we need to prove that the feature space has a valid dot product. This is straightforward in some cases like the linear kernel or the polynomial kernel, but more often than not finding such a space is not trivial. Kernel functions satisfy Mercer's theorem, that is, they must be continuous, symmetric, and they should have a positive semi-definite Gram matrix. Mercer's theorem guarantees that the optimization problem is convex and that unique solution will be found, so it is often used to prove that a function is a valid kernel.

Finally, an intuitive way to think about kernel functions is to consider the scalar number returned by a kernel as a measure of similarity between two points. Take for example simplest case of a linear kernel. It computes the dot product between the two vectors, therefore if the similarity is the dot product, we would be considering two perpendicular

[3]The quadratic optimization problem is quite complex but it is a solved problem and the method is exactly same for all SVM. The kernel on the other hand needs to be carefully chosen considering the problem at hand.

vectors (dot product equals 0) dissimilar and two parallel vectors the most similar as the dot product equals 1 (maximum value if the vectors are normalised). Often kernels can be understood intuitively in a similar fashion.

Chapter 3

Univariate Kernels

A fundamental part of using kernel methods is to choose an appropriate kernel function for the data at hand. Since their proposal, the number of available kernels has increased and today there are ways to work which many types of data, like vectors, graphs, strings, functional data and more. There are however, only a few basic kernels explicitly designed to work on categorical data, and there is much room for improvement in that front.

Most kernels deal with quantitative data, where the inputs are represented as vectors with values in $\mathbb{R}^n$. In contrast, when dealing with categorical data, the inputs are tuples whose values are limited to a finite set of possible values or *categories*. In some cases, data can be a mixture of categorical and quantitative data.

The treatment that categorical data receives is often very basic. In many cases it is limited to recoding the categorical variables in dummy variable form, and then perform the rest of the process as you would with quantitative data.

This chapter describes the two most common kernels used with categorical data, and then a family of kernels defined specifically for categorical data that were first introduced in *Kernel Functions for Categorical Variables with Application to Problems in the Life Sciences* [4]. All the kernels in this chapter will be uni-variate kernels, which means that they are defined only as the product of two single categorical variables. When having multiple independent variables they can be easily combine multiple uni-variate kernels, as kernels preserve their properties under some operations like addition, multiplication and the exponential of a kernel or the evaluation on a polynomial with non-negative coefficients, and a few others. [1]

[1] It is the same to say that these operation preserve positive semi-definiteness. This is demonstrated in *Functions That Preserve Families of Positive Semi-definite Matrices* [5].

3.1 Kernel Functions

3.1.1 RBF

The Gaussian Radial Basis Function (RBF) is a very popular kernel function used often in Support Vector Machines and other methods. It's based on the squared Euclidean distance between elements and is a safe choice because it usually produces reasonably good results.

As there is no Euclidean distance defined between two categorical elements, it is not possible to use this kernel directly with categorical data. As mentioned previously, the usual process is to recode the data using dummy variable form and then use the RBF kernel on the data. As it will be shown later this can work well in many cases, but it is very limited. There has been some effort to propose other ways to map categorical data into a euclidean space such as the method in *Learning dissimilarities for categorical symbols*[6], but the approach of this project is to find kernels that work on categorical variables, not recode the variables as vectors in $\mathbb{R}^N$.

Let us go on to the definition of the kernel:

$$k(\mathbf{x}, \mathbf{y}) = \exp\left(\gamma ||\mathbf{x} - \mathbf{y}||^2\right)$$

The parameter $\gamma = -1/2\sigma^2$ allows to adjust the kernel. Notice how the value of the kernel ranges between 1 (for identical elements) and 0 for distant elements (it decreases with distance). We mentioned in the previous chapter that it can be useful to think of kernels as a function that computes the similarity between two elements, if we consider the RBF kernel, two elements will be similar if the squared Euclidean distance is small, and different if the distance is large, therefore, elements are similar if they are close in space. Also, we can interpret the use of the parameter: decreasing γ (thus increasing σ would result in the value of the kernel not decreasing as much with distance, thus increasing the area of influence of each support vector. Figure 3.1 shows the decision function on the same dataset for different values of γ, illustrating what was just described.

3.1.2 Overlap

Another way to use categorical data is to come up with kernel functions that take categorical variables as input. The most common example of one such kernel is the Overlap kernel, also called Dirac kernel, which may be the simplest, most intuitive kernel for categorical variables.

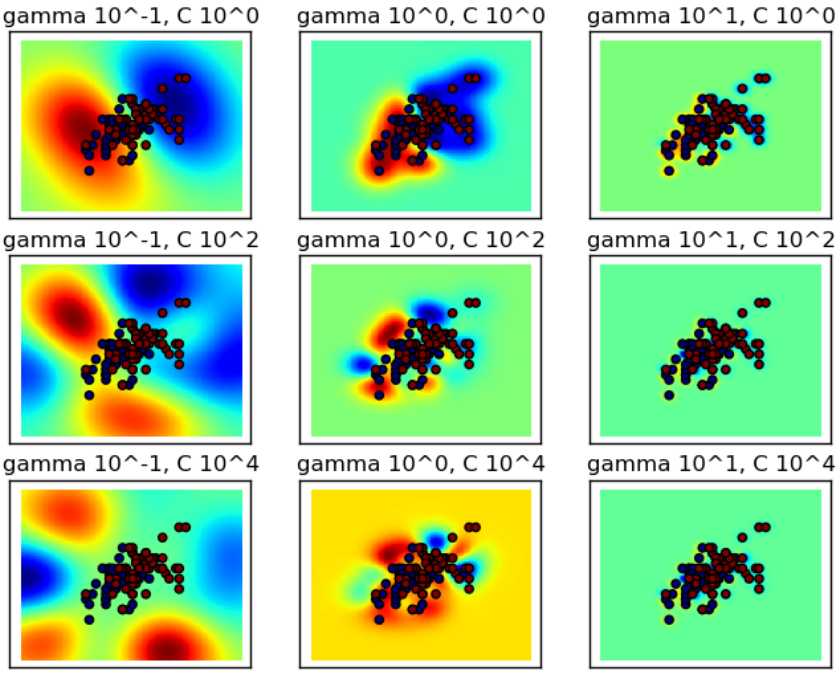


FIGURE 3.1: The decision function for different values of C and γ.

It computes the ratio of variables with matching categories between two items:

$$k(\mathbf{x}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} [x_i = y_i]$$

Where:

$$[x = y] = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

It is the number of matches over n, so when two elements have matching categories in every variable the kernel returns 1, if there are no matches the value is 0.

3.1.3 Categorical Kernel k_0

The k_0 kernel takes the Overlap kernel one step further. It's defined as:

$$k_0(\mathbf{x}, \mathbf{y}) = f_p \left(\frac{1}{n} \sum_{i=1}^{n} f_a \left([x_i = y_i] \right) \right)$$

Instead of just computing the overlap, the k_0 kernel allows to use two functions, f_a and f_p, before and after combining the multiple variables into a single value.

Notice that the Overlap kernel is a special case of k_0 when both f_a and f_p are the identity function. As such, we will not be including the Overlap kernel in our tests explicitly, but rather as another variant of the k_0 kernel.

By using different functions one can define variations of the kernel. For example, the transformation function f_1^K is defined in *Kernel Functions for Categorical Variables with Application to Problems in the Life Science* [4] as:

$$f_1^K(x, y) = e^{\gamma K(x,y)}, \quad \gamma > 0$$

Which gives raise to two new kernels, depending on whether f_1 is applied as f_a or f_p:

$$k_0'(\mathbf{x}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} exp\left(\gamma[x_i = y_i]\right)$$

$$k_0''(\mathbf{x}, \mathbf{y}) = exp\left(\frac{\gamma}{n} \sum_{i=1}^{n} [x_i = y_i]\right)$$

It is important that any transformation function used preserves the positive semi-definite property of the Gram matrix. In this case, multiplication by a constant the exponentiation of a kernel preserve positive semi-definiteness, but there are more operations that preserve the properties as demonstrated in *Functions That Preserve Families of Positive Semi-definite Matrices* [5].

3.1.4 Categorical Kernel k_1

Our purpose in this project is to investigate kernels grounded on the use of the probabilistic structure of the data. Most categorical kernels do not take the distribution of the data into account which means that a certain category that is very common will have the same consideration as a category that is a rare occurrence.

The kernel k_1, which was introduced in *Kernel Functions for Categorical Variables with Application to Problems in the Life Science* [4], builds on the idea that the distribution of the data can provide insightful information about the dataset that can be used to obtain better results.

In order to do this, the kernel uses Probablity Mass Function (PMF) of the data set to weight each category. Since the PMF of the data is not always known, in [4] it is

suggested to use the PMF of the available data, which will be the approach followed in the experiments coming in the next section.

To obtain the k_1 kernel, the univariate kernel $[x = y]$ used in the Overlap kernel is replaced with the following univariate kernel:

$$k_1^U(x, y) = \begin{cases} h_\alpha\left(P_X\left(x\right)\right) & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

Where h_α is an inverting function:

$$h_\alpha(z) = (1 - z^\alpha)^{1/\alpha}, \quad \alpha > 0$$

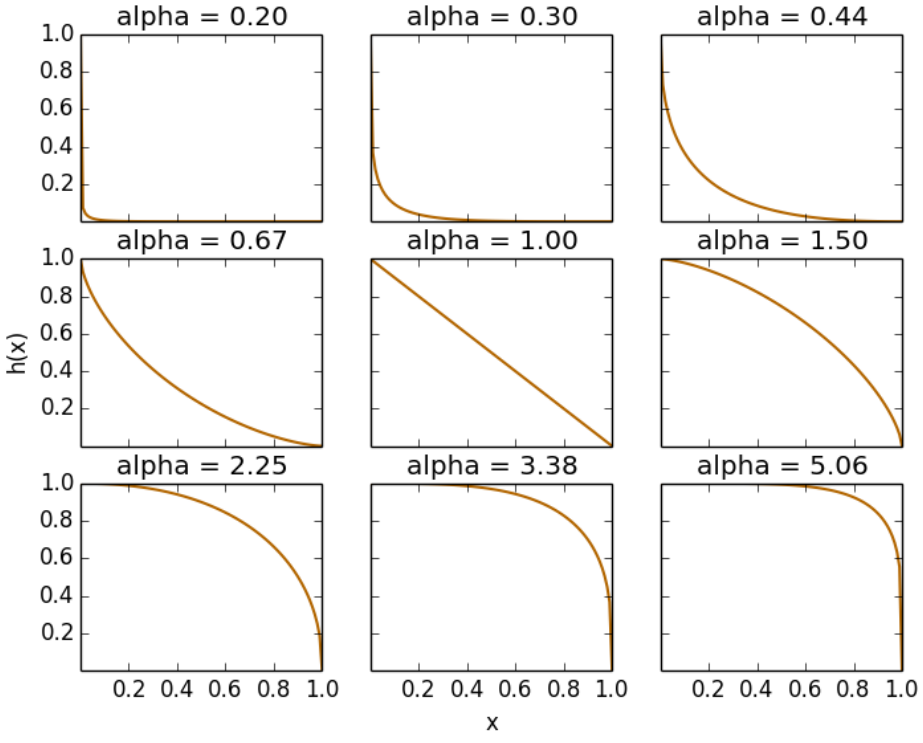


FIGURE 3.2: Shape of the function for various values of α.

k_1^U can be extended to multiple variables similarly to k_0 using the summation of the uni-variate kernels:

$$k_1(\mathbf{x}, \mathbf{y}) = f_p\left(\frac{1}{n}\sum_{i=1}^{n} f_a\left(k_1^U(x_i, y_i)\right)\right)$$

Just like before, f_a and f_p can be any function, in addition f_1, we introduce:

$$f_2^K(x,y) = e^{\gamma(2K(x,y)-K(x,x)-K(y,y))}, \quad \gamma > 0$$

The identity, f_1 and f_2 functions can be used to obtain the following four variations of the kernel, which will be used in the tests:

$$k_1(\mathbf{x},\mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} k_1^U(x_i,y_i)$$

$$k_1'(\mathbf{x},\mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} exp\left(\gamma k_1^U(x_i,y_i)\right)$$

$$k_1''(\mathbf{x},\mathbf{y}) = exp\left(\frac{\gamma}{n}\sum_{i=1}^{n} k_1^U(x_i,y_i)\right)$$

$$k_1'''(\mathbf{x},\mathbf{y}) = \exp\left(\frac{\gamma}{n}\sum_{i=1}^{n}\left[2k_1^U(x_i,y_i) - k_1^U(x_i,x_i) - k_1^U(y_i,y_i)\right]\right)$$

3.2 Evaluation

This section covers the evaluation of the described kernels using various data sets. The methodology used in all the experiments is very similar.

The first step is to obtain two data sets: one for training and another for testing. If the data is limited, the usual procedure is to randomly split the dataset in two. If the dataset is artificial and the samples can be generated randomly, then it is not necessary to split the dataset as more samples can be generated for the test set.

The second step is to leave the test set aside, and fit the model using the training set. When fitting a model it is important to choose a kernel and its parameters. Since the objective of this project is to compare kernels, the process will be to iterate the different kernels to be compared, finding the best parameters with each kernel.

A common problem with models is over-fitting to the training set. A way to avoid over-fitting when searching the parameters is to use stratified cross-validation. The training set is split in n folds, the process of finding the best parameters is then repeated n times, every time leaving a fold out of the training and evaluating the model on that left out fold. *Stratified* cross-validation simply means that the folds are separated maintaining the class proportions, to get a better estimation of the error and avoid skewing the classifier. The chosen parameters are those that perform better when evaluated on the

left out training data fold, and then a final model is fit using the best parameters and the whole training set.

Finally, to compare effectiveness of all the kernels, each model is used to predict the test set. Since this is data that was never used in the fitting of the model, the result will be a more reliable estimate of the performance of the kernel on new data. Since the error can show a high variability, depending on how the random split in training and test ended up –specially when the data set is relatively small–, the whole process is repeated as many times as possible, to get a sample of the distribution of the error instead of just a single measure.

3.2.1 Synthetic Data Set

This dataset is intended to provide a very simple setup to compare how different kernels behave when used on categorical data with some statistical properties.

The algorithm generates a series of examples with categorical attributes. The categories generated are either tied to one specific class, or not related to any class. The categories tied to classes are rare, while the unrelated are very common.

The algorithm that generates the dataset takes a parameter p, which can be used to adjust the frequency of appearance of class-related attributes with respect to common attributes. A parameter of $p = 0$ will give a null probability of class-related categories, while a parameter of $p = 1$ will mean that class-related attributes will appear as frequently as common attributes.

Figure 3.3 shows the impact of the parameter p when using RBF with default parameters on data sets with different sizes, when all the other parameters of the algorithm are constant (2 classes and 25 attributes).

The chart shows precisely how when p increases the difficulty of the classification decreases. For low values of p (< 0.3) the problem is very difficult (performance is hardly better than guessing at random), while high values of p (> 0.7) make the problem very easy (since class-related categories become more common). The most interesting results are for values of p around 0.5. It can also be noted that with bigger data sets the error is smaller, which is also expected.

The result is a very flexible algorithm. The different parameters (p, size, number of attributes and number of classes) can be used to find out how different algorithms and kernels behave on a wide array of configurations.

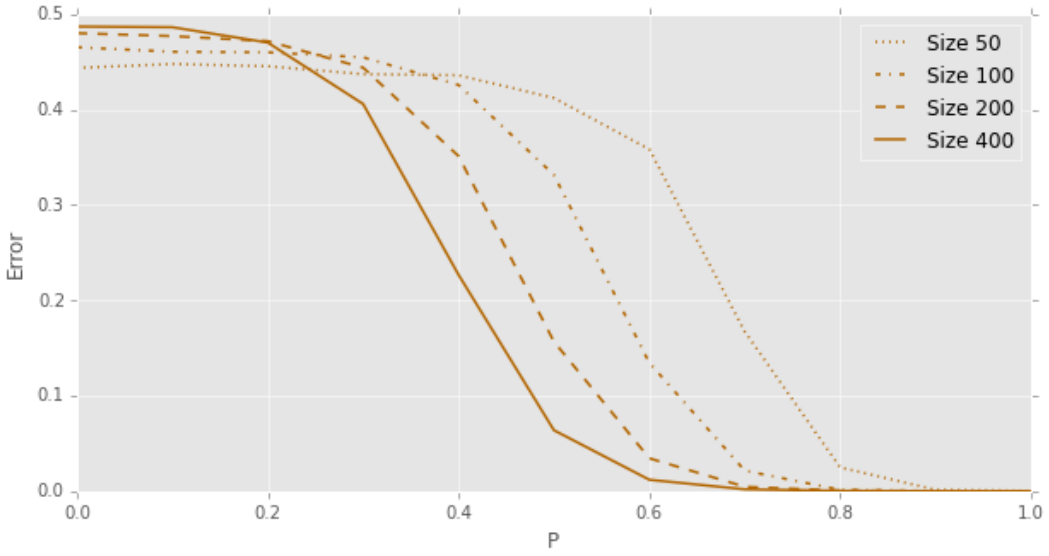


FIGURE 3.3: Effect on the error of the parameter p when using the RBF kernel.

The pseudo-code that implements the algorithm is included in Algorithm 1 and will be explained with an example.

Let us now look at table 3.1, which shows an example of data set generated by the algorithm, with 10 examples and 3 classes. It was generated with a parameter of $p = 0.75$, which results in a dataset that is fairly easy to classify:

In this example, the categories where replaced with o and x if they were common categories for attributes, while **a**, **b** and **c** replace categories related to the classes A, B and C, respectively. The actual values are generated randomly and each attribute is independent from the others.

It shows in the example that the o and x categories are the most common (80% of the values), however they don't provide any information about the class of an item, as they appear randomly. The other values, although less frequent (20%), are direct indicators of which class the example belongs to.

This is not a very complex problem, but it is enough to check which kernels pick this pattern efficiently. The reasoning behind the dataset is that the common categories generate a lot of noise which makes the classification harder, while the class-related categories give enough information to correctly classify the examples. It is expected that the more commonly used kernels will have a hard time classifying the examples and will be prone to a higher error and more variability even when training with cross-validation because they do not take into account the PMF of the different variable categories. The kernel k_1, which does take the PMF into account should be able to offer a better classification.

Data:
N: number of examples to generate.
D: number of attributes for each example.
C: number of classes.
p: difficulty parameter.
Result: Generates a matrix of attributes X and a class vector y.
$\Sigma \leftarrow$ collection of symbols to use as categories, such that $|\Sigma| \geq C + 2$.;
`/* Asign a class to each example:` */
for $n = 1..N$ **do**
  $y[n] \leftarrow n \mod C$
end
`/* Generate attributes:` */
for $d = 1..D$ **do**
  $\Sigma \leftarrow$ `Shuffle(`Σ`)`;
  `/* Pick two values to use as random categories:` */
  $a \leftarrow \Sigma[C + 1]$;
  $b \leftarrow \Sigma[C + 2]$;
  `/* And C values to use as class-related categories:` */
  $c \leftarrow \{\Sigma[i] \mid 1 \leq i \leq C\}$;
  for $n = 1..N$ **do**
    if `Random()` $> C/(C + 2) * p^2$ **then**
      `/* Common case:` */
      `/* pick either of the two random categories.` */
      if `Random()` $> 1/2$ **then**
        $X[n, d] \leftarrow a$
      else
        $X[n, d] \leftarrow b$
      end
    else
      `/* Rare case:` */
      `/* pick the class-related category.` */
      $X[n, d] \leftarrow c[y[n]]$
    end
  end
end
return X, y

Algorithm 1: SyntheticDataset

For the experiment the parameters for the data set were fixed on $p = 0.5$, 25 attributes and 4 classes.

Two data sets with these parameters were generated: one for training of size 100, another for testing of size 200. The parameters for each kernel were selected by performing a grid search with 5-fold cross validation on the training set. With the parameters fixed, each kernel was then evaluated on the test dataset.

This process was repeated 25 times with newly generated independent data sets to have more information on the expected accuracy of each kernel, and the results are shown in

-	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	Class
$\mathbf{x}_0$	o	**b**	x	o	**b**	**b**	o	**b**	B
$\mathbf{x}_1$	o	x	o	x	**c**	o	o	o	C
$\mathbf{x}_2$	x	o	o	**a**	o	x	x	x	A
$\mathbf{x}_3$	o	x	x	**a**	x	o	o	x	A
$\mathbf{x}_4$	o	o	o	**c**	x	x	o	o	C
$\mathbf{x}_5$	x	x	o	x	o	x	**c**	o	C
$\mathbf{x}_6$	o	o	o	o	**a**	**a**	o	**a**	A
$\mathbf{x}_7$	**c**	x	**c**	x	**c**	o	x	o	C
$\mathbf{x}_8$	o	o	**b**	x	o	o	o	**b**	B
$\mathbf{x}_9$	**c**	x	x	**c**	o	**c**	o	x	C

TABLE 3.1: Synthetic dataset example.

figure 3.4. Table 3.2 shows the parameter values evaluated in the grid search.

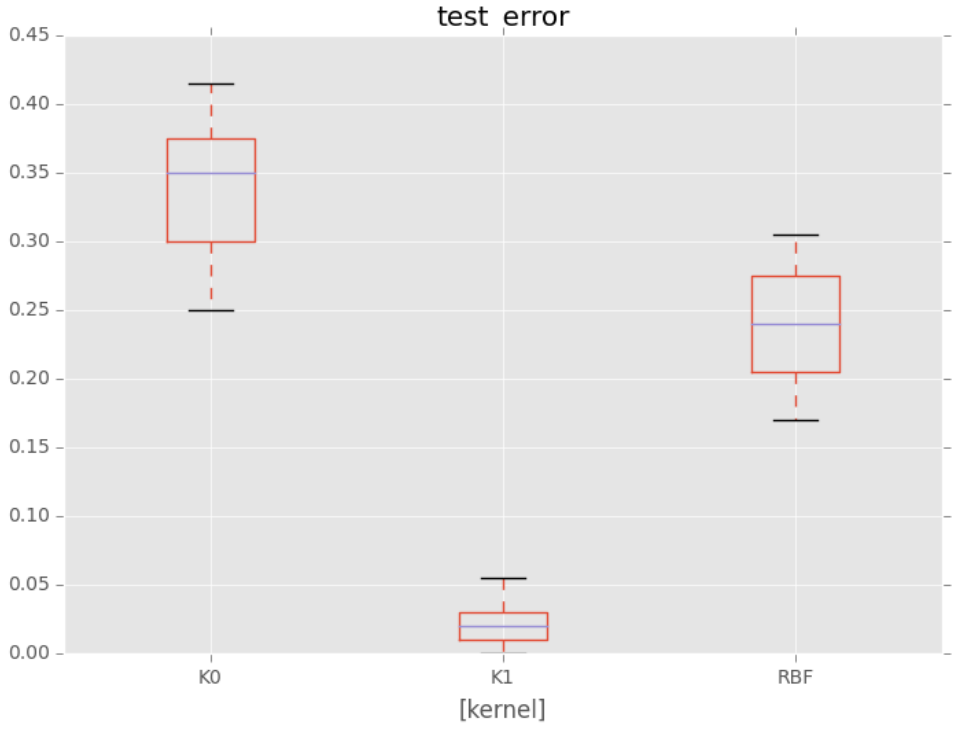


FIGURE 3.4: Test error of the univariate kernels on the synthetic dataset.

Kernel	C	α	γ
RBF	$0.1, 1, 10, 100, 1000$	-	$2^i : -11 \leq i \leq 1$
k_0	$0.1, 1, 10, 100, 1000$	-	$2^i : -3 \leq i \leq 2$
k_1	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$

TABLE 3.2: Search space for the synthetic dataset.

The results clearly confirm the initial hypothesis, see how much of a difference using the PMF information makes to the accuracy of the kernel. With the k_1 kernel clearly

surpassing both the RBF and k_0 kernels. It also shows that there is more variability in the test error of the k_0 and RBF kernels.

The synthetic data set is very artificial and designed to highlight this particular aspect of the kernels, which means that this results are not an indication of how the kernels behave in real world applications. Still, this test demonstrates that the kernel k_1 has a potential advantage in cases where the probabilistic distribution can be used to make better predictions.

As mentioned in the previous section, the Overlap kernel is not included in the test explicitly, but grouped in the k_0 kernel. One of the different configurations of the prev and post functions used in the parameter search is the overlap kernel itself. This means that the performance of k_0 is always better or equal than that of the overlap kernel.

Also as mentioned before, the PMF used by the kernel k_1 was approximated from the available data in the training set.

3.2.2 Monks Data Set

The original MONK's problem were the basis of a first international comparison of learning algorithms. The result of this comparison is summarized in *The monk's problems a performance comparison of different learning algorithms* [7]. One significant characteristic of this comparison is that it was performed by a collection of researchers, each of whom was an advocate of the technique they tested (often they were the creators of the various methods). In this sense, the results are less biased than in comparisons performed by a single person advocating a specific learning method, and more accurately reflect the generalization behavior of the learning techniques as applied by knowledgeable users.

For our tests, we use a slightly modified version of the MONK's problems.

The original MONK's problems defines a domain, based on six different attributes:

a_1: **head_shape** $\in$ round, square, octagon

a_2: **body_shape** $\in$ round, square, octagon

a_3: **is_smiling** $\in$ yes, no

a_4: **holding** $\in$ sword, balloon, flag

a_5: **color** $\in$ red, yellow, green, blue

a_6: **has_tie** $\in$ yes, no

In this domain, three different binary classification problems are defined:

1. Problem M_1:
   (head_shape = body_shape) $\vee$ (color = red)
2. Problem M_2:
   Exactly two of the six attributes have their first value.
3. Problem M_3:
   (color = green $\wedge$ holding = sword) $\vee$ (color $\neq$ blue $\wedge$ body_shape $\neq$ octagon)

In the original, the third dataset also adds noise to the problem.

In our case, we will use a different version of the problem, which combines the three variations into a single problem. We want to repeat the experiment detailed in *Kernel Functions for Categorical Variables with Application to Problems in the Life Science* [4], in order to compare the results with obtained there with additional kernels. This variation takes as a parameter an integer D that indicates the number of *chunks*. Each *chunk* is generated by picking random values for each of the six attributes a_1 to a_6, and the six attributes in a chunk are selected independently. Then, for each *chunk* the following formula is computed:

$$C_d = M_2 \wedge \neg(M_1 \wedge M_3)$$

Then, each data point is classified on one of two groups depending on whether half or more of the chunks are true.

Figure 3.5 illustrates the effect the number of chunks has on the difficulty of the problems, As the number of chunks increases, the score obtained by the RBF kernel decreases slightly. When training on bigger datasets the score increases.

The algorithm itself is pretty straightforward, the pseudo-code is shown in Algorithm 2.

Figure 3.6 shows the results obtained by the kernels RBF, k_0, k_1 and k_2 in the GMONKS test with a single chunk (6-dimensional problem), and a training and test sizes of 100 and 200, respectively.

For each kernel, the best model was found performing grid search over the parameter space shown in Table 3.3 using 5-fold cross-validation, then the model was evaluated on the test set. The results were averaged over 25 executions.

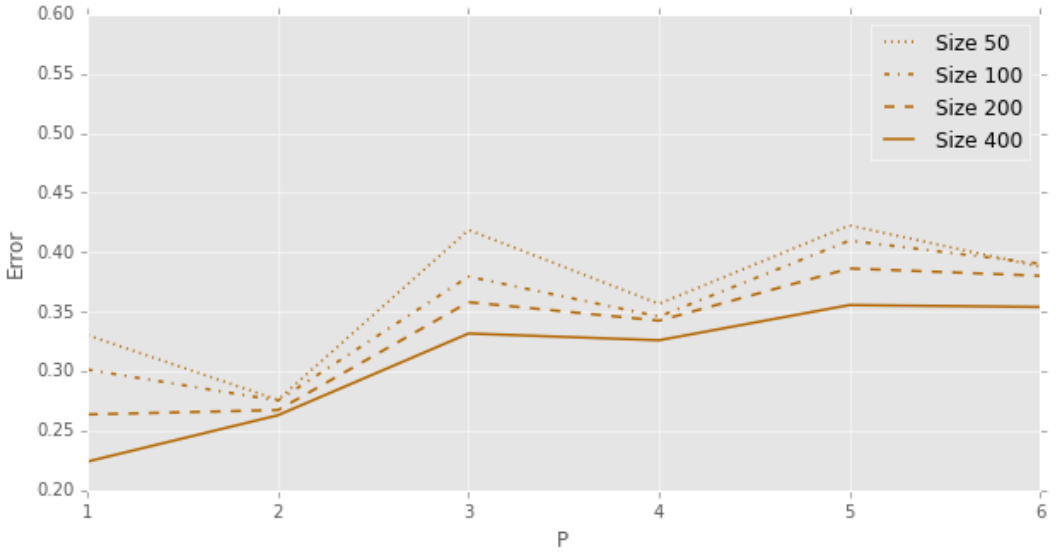


FIGURE 3.5: Effect of the number of blocks when using the RBF kernel.

Data:
N: number of examples to generate.
D: number of chunks for each example.
Result: Generates a matrix X with N examples and $D \times 6$ attributes and a class vector y.

$a_1 \leftarrow \{1, 2, 3\}$;
$a_2 \leftarrow \{1, 2, 3\}$;
$a_3 \leftarrow \{1, 2\}$;
$a_4 \leftarrow \{1, 2, 3\}$;
$a_5 \leftarrow \{1, 2, 3, 4\}$;
$a_6 \leftarrow \{1, 2, 3, 4, 5, 6\}$;
for $n = 1..N$ **do**
  for $d = 1..D$ **do**
    `/* generate a chunk` $c =< a_1, a_2, a_3, a_4, a_5, a_6 >$ `of 6 random variables by picking a random value from each set. */`
    $c \leftarrow < a_i^{(random)} \mid 1 \leq i \leq 6 >$;
    $p_1 \leftarrow (c_1 = c_2) \vee (c_5 = 1)$;
    $p_2 \leftarrow |\{c_i : c_i = 1\}| \geq 2$;
    $p_3 \leftarrow (c_5 = 3 \wedge c_4 = 1) \vee (c_5 \neq 3 \wedge c_2 \neq 2)$;
    $C_d \leftarrow p_2 \wedge \neg(p_1 \wedge p_3)$;
  end
  $X_n \leftarrow concat(\{C_d : 1 \leq d \leq D\})$;
  $y_n \leftarrow |\{C_d : C_d \equiv true, 1 \leq d \leq D\}| \geq D/2$;
end
return X, y

Algorithm 2: gMonks

Kernel	C	α	γ
RBF	$0.1, 1, 10, 100, 1000$	-	$2^i : -11 \leq i \leq 1$
k_0	$0.1, 1, 10, 100, 1000$	-	$2^i : -3 \leq i \leq 2$
k_1	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$

TABLE 3.3: Search space for the gmonks dataset.

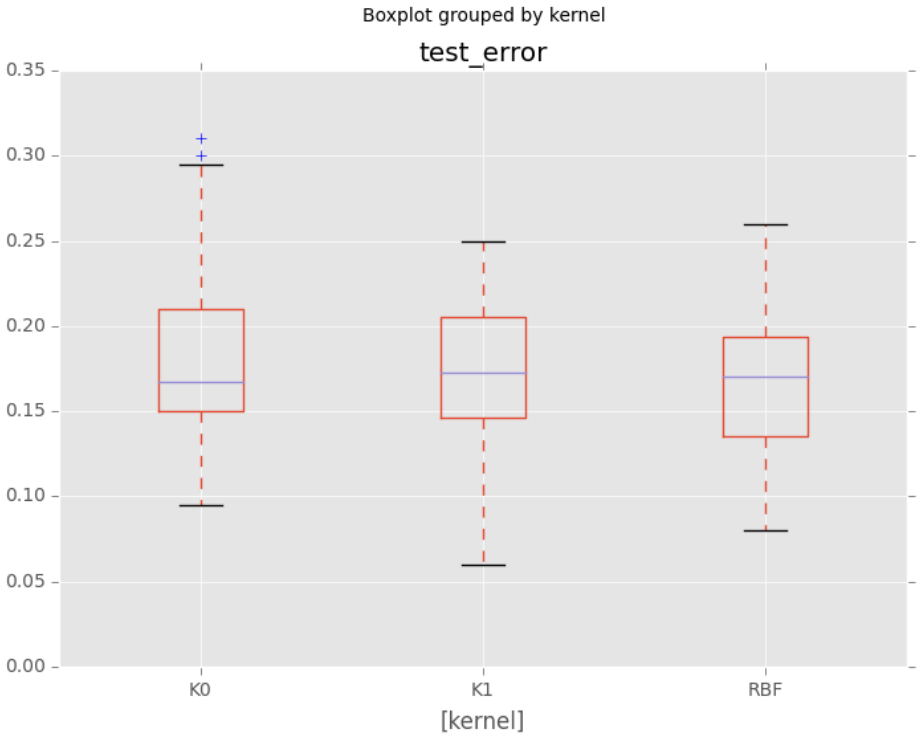


FIGURE 3.6: Test error of the univariate kernels on the gmonks dataset.

3.2.3 Other Data Sets

Unlike the previous sections, the data sets in this section are from real life problems instead of randomly generated.

3.2.3.1 Promoter Gene Data Set

This dataset is contains gene sequences from E. coli. The purpose is to differentiate between promoter and non promoter sequences.

In this case, the dataset was split 1/3 into test 2/3 into train. The parameters were found using 10-fold coss-validation. Figure 3.7 shows the distribution of the error of each kernel over 20 repetitions, and the search space in Table 3.4.

Kernel	C	α	γ
RBF	$0.1, 1, 10, 100, 1000$	-	$2^i : -11 \leq i \leq 1$
k_0	$0.1, 1, 10, 100, 1000$	-	$2^i : -3 \leq i \leq 2$
k_1	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$

TABLE 3.4: Search space for the promoters dataset.

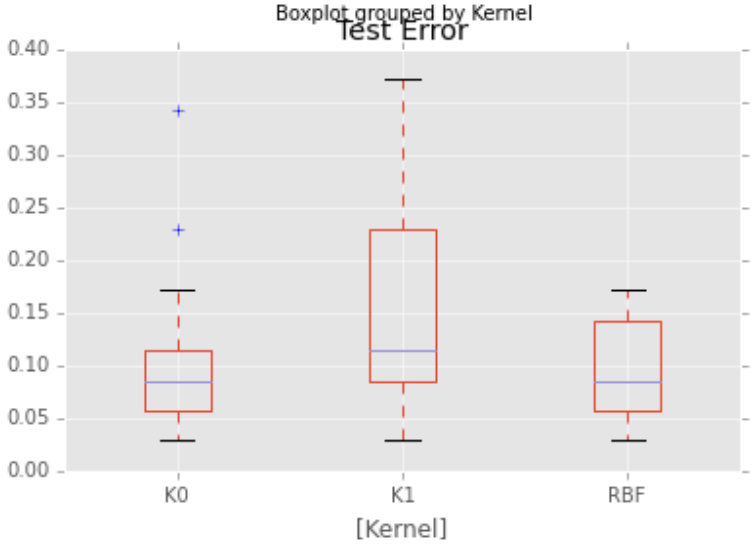


FIGURE 3.7: Test error of the univariate kernels on the promoter gene dataset.

3.2.3.2 Splice Data Set

Similar to the previous one, but with splice-junction gene sequences. Again 10-fold cross-validation and 20 repetitions. The results are show in Figure 3.8. and the search space in Table 3.5.

Kernel	C	α	γ
RBF	$0.1, 1, 10, 100, 1000$	-	$2^i : -11 \leq i \leq 1$
k_0	$0.1, 1, 10, 100, 1000$	-	$2^i : -3 \leq i \leq 2$
k_1	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$

TABLE 3.5: Search space for the splice dataset.

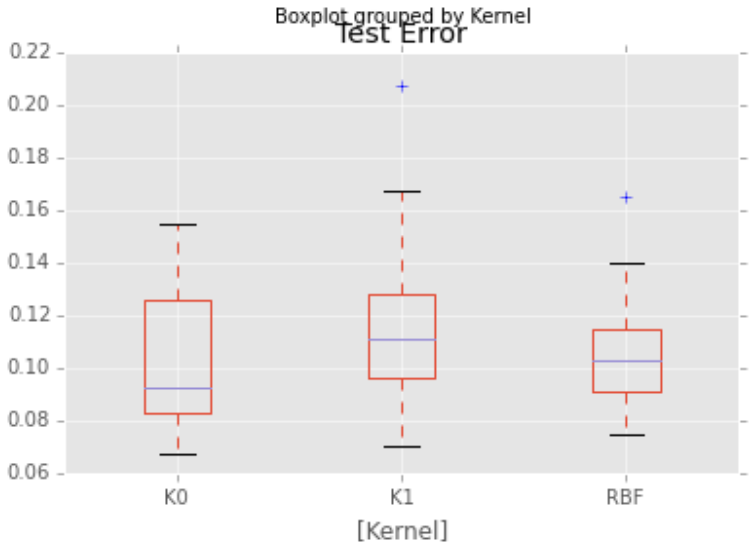


FIGURE 3.8: Test error of the univariate kernels on the splice gene dataset.

Chapter 4

Multivariate Kernels

This chapter introduces a new family of categorical kernels designed specially for data sets with multiple categorical variables.

4.1 Functions

4.1.1 Multivariate Categorical Kernels

In *Metric and euclidean properties of dissimilarity coefficients* [8], Gower and Legendre enumerate several dissimilarity measures and study some of their metric and Euclidean properties. One of the properties that they cover is whether they are positive semi-definite, which happens to be the requirement for kernel functions. This section is dedicated to taking some of the measures in [8] and showing how to use them as kernel functions.

The measures are based on binary variables with values in $\{0, 1\}$ where 1 usually denotes presence and 0 denotes absence –they are often used in fields such as biology–. The usual notation to express this measures is to use a to denote the number of $(1, 1)$ matches, b for $(1, 0)$, c for $(0, 1)$ and d for $(0, 0)$.

With that in mind, one can define the following set of rules that allow to convert a similarity measure defined in terms of a, b, c and d to a kernel:

$$
\begin{aligned}
a &\to \sum_{i:x_i=y_i} \left[f^2(x_i) + f^2(y_i)\right] = 2 \sum_{i:x_i=y_i} f^2(x_i) \\
b &\to \sum_{i:x_i\neq y_i} f^2(x_i) \\
c &\to \sum_{i:x_i\neq y_i} f^2(y_i) \\
d &\to \sum_{i:x_i\neq y_i} \left[1 - f^2(x_i) + 1 - f^2(y_i)\right]
\end{aligned}
$$

To adapt the binary matches to work with categorical variables it is necessary to define what is considered a match and mismatch when dealing with categories. A (1, 1) match is very straightforward, as matches can be considered as the cases where the categories match. For mismatches (1, 0), (0, 1) and (0, 0) there is a conceptual difference, since with binary variables there is a distinction between presence and absence, but in categorical variables all categories are equal and there is no concept of category or categories that indicate *absence* [1].

To maintain the notion that all categories are equal a mismatch can be considered as both (1, 0) and (0, 1), this is shown by the symmetry between b and c: when $x_i \neq y_i$ it can be interpreted as the presence of the category x_i in x_i and its absence from x_j, which is analog to a (1, 0) match. At the same time, the absence of y_i in x_i and its presence in x_j is analog to a (0, 1) match.

The match (0, 0) is a bit of an oddball. Since it doesn't really make sense for categorical variables it would be reasonable to leave it out, or just make d always equal to 0. However, most similarity measures for binary variables already come in two versions, one that includes d and another that doesn't. Making d equal to zero would make many of those measures redundant, so instead it was chosen to add a rule that seemed reasonable, to see if it made any difference.

Now consider the Jaccard similarity coefficient, defined in *Metric and euclidean properties of dissimilarity coeficients*[8] as:

$$S_3 = \frac{a}{a+b+c}$$

[1] Sometimes missing values in the data set are imputed to a new category, this just adds a new category that is equal to the others but still is not analog to the 0 in a binary variable. The problem here is independent from the imputation of missing values, as we are trying to finding a way measure similarity between categories

The equivalent kernel based on S_3 would be the following:

$$m_3(\mathbf{x},\mathbf{y}) = \frac{2\sum\limits_{i:x_i=y_i} f^2(x_i)}{\sum\limits_{i=1}^{n}\left[f^2(x_i)+f^2(y_i)\right]}$$

Notice that when f is any constant function, this expression is reduced to the Overlap kernel. To see this, let $f(x) = c \in \mathbb{R}, (c \neq 0)$:

$$\begin{aligned} m_3^a(\mathbf{x},\mathbf{y}) &= \frac{2\sum\limits_{i:x_i=y_i} c}{\sum\limits_{i=1}^{n}[c+c]} \\ &= \frac{2c\sum\limits_{i=1}^{n}[x_i=y_i]}{2cn} \\ &= \frac{1}{n}\sum_{i=1}^{n}[x_i=y_i] \end{aligned}$$

More interesting, is the use of the probability information $f(x) = \sqrt{h_\alpha(P(x))}$ to obtain the following kernel:

$$m_3^b(\mathbf{x},\mathbf{y}) = \frac{\sum\limits_{i:x_i=y_i} h_\alpha(P(x_i))}{\sum\limits_{i=1}^{n}\left[h_\alpha(P(x_i))+h_\alpha(P(y_i))\right]}$$

Which is the kernel that will be used in the experiments.

This same process can be repeated with each similarity coefficient to obtain different kernels:

$S_4 = \dfrac{a+d}{a+b+c+d}$

$$m_4(\mathbf{x},\mathbf{y}) = \frac{2\sum\limits_{i:x_i=y_i} f^2(x_i) + 2\sum\limits_{i:x_i\neq y_i}\left[1-f^2(x_i)+1-f^2(y_i)\right]}{\sum\limits_{i=1}^{n}\left[f^2(x_i)+f^2(y_i)\right] + 2\sum\limits_{i:x_i\neq y_i}\left[1-f^2(x_i)+1-f^2(y_i)\right]}$$

$S_5 = \dfrac{a}{a+2(b+c)}$

$$m_5(\mathbf{x},\mathbf{y}) = \frac{2\sum\limits_{i:x_i=y_i} f^2(x_i)}{2\sum\limits_{i:x_i=y_i} f^2(x_i) + 2\sum\limits_{i:x_i\neq y_i}\left[f^2(x_i)+f^2(y_i)\right]}$$

The process is analogous for the rest of the similarity measures, shown in Table 4.1. The complete expressions for all the kernels obtained from this measures are available in Appendix B.

$S_3 = \frac{a}{a+b+c}$
$S_4 = \frac{a+d}{a+b+c+d}$
$S_5 = \frac{a}{a+2(b+c)}$
$S_6 = \frac{a+d}{a+2(b+c)+d}$
$S_7 = \frac{a}{a+1/2(b+c)}$
$S_9 = \frac{a-(b+c)+d}{a+b+c+d}$
$S_{12} = \frac{a}{\sqrt{(a+b)(a+c)}}$
$S_{13} = \frac{ad}{\sqrt{(a+b)(a+c)(d+b)(d+c)}}$
$S_{14} = \frac{ad-bc}{\sqrt{(a+b)(a+c)(d+b)(d+c)}}$

TABLE 4.1: Positive semi-definite similarity measures identified by Gower and Legendre.

4.2 Experiments

4.2.1 Synthetic Data Set

We return to the first data set in the previous chapter. This time to evaluate the new multivariate kernels. The procedure followed was the same as in the previous chapter: generate a training set of size 100, find the best model using 5-fold cross-validation, measure the error of the best model on a newly generated data set, rinse and repeat 20 times to obtain a reasonable measure of the expected error of each kernel on this particular problem.

The results are shown in Figure 4.1, and the search space for the parameters in Table 4.2.

C	α	γ
$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$

TABLE 4.2: Kernel parameters searched in the Synthetic data set.

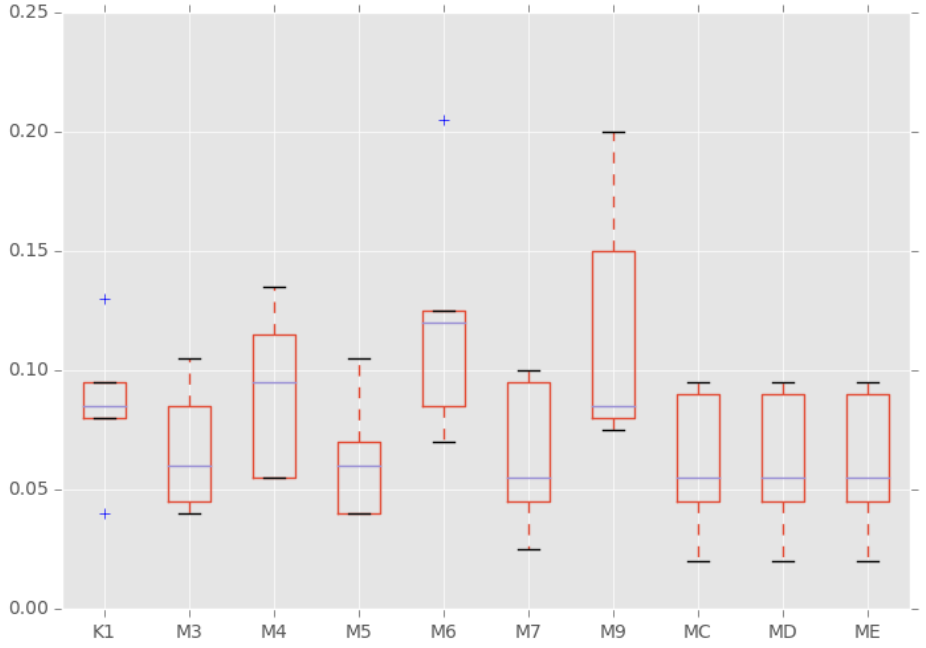


FIGURE 4.1: Test error distribution on the synthetic data set of various kernels.

4.2.2 GMonks Data Set

Likewise, same setup and procedure as the previous chapter, results in Figure 4.2, parameters in Table 4.3.

C	α	γ
$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$

TABLE 4.3: Kernel parameters searched in the GMonks data set.

4.2.3 Other Data Sets

This section revisits the real data sets from the previous chapter, and adds some other results. For simplicity, Table 4.4 shows the parameters used in every kernel for all the experiments. It would be possible to narrow down the search in most cases, but the only advantage would be that it would be faster to compute. Since the computations were done in a dedicated server, and were left to run until they were finished, it was not a problem if they took longer than necessary. It was also important to keep the comparison fair, and being too zealous in finding a good search space for one kernel could easily lead to having results that are not very representative. The search space used was complete enough to provide good reasults for all kernels in most circumstances.

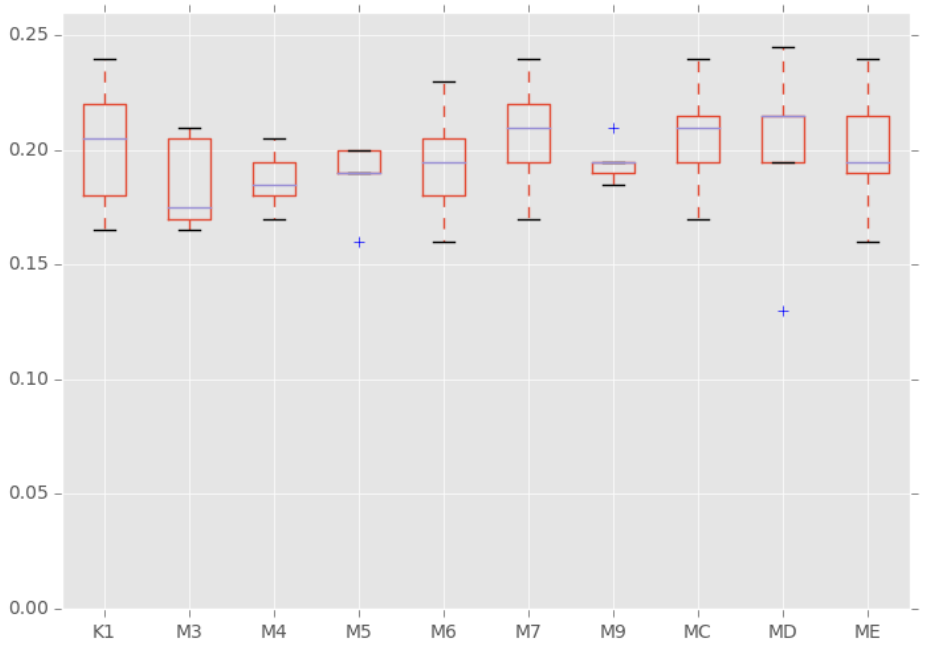


FIGURE 4.2: Test error distribution on the GMonks data set of various kernels.

Kernel	C	α	γ
RBF	$0.1, 1, 10, 100, 1000$	-	$2^i : -13 < i < 1$
k_0	$0.1, 1, 10, 100, 1000$	-	$2^i : -3 \leq i \leq 2$
k_1	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_3	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_4	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_5	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_6	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_7	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_9	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_{12}	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_{13}	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$
m_{14}	$0.1, 1, 10, 100, 1000$	$1.5^i : -3 \leq i \leq 2$	$2^i : -3 \leq i \leq 2$

TABLE 4.4: Search space for the categorical data sets.

4.2.3.1 Mushroom Data Set

A data set of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species is identified as edible or poisonous. The source states that there is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be" for Poisonous Oak and Ivy. The results for 10 repetitions are show in Figure 4.3.

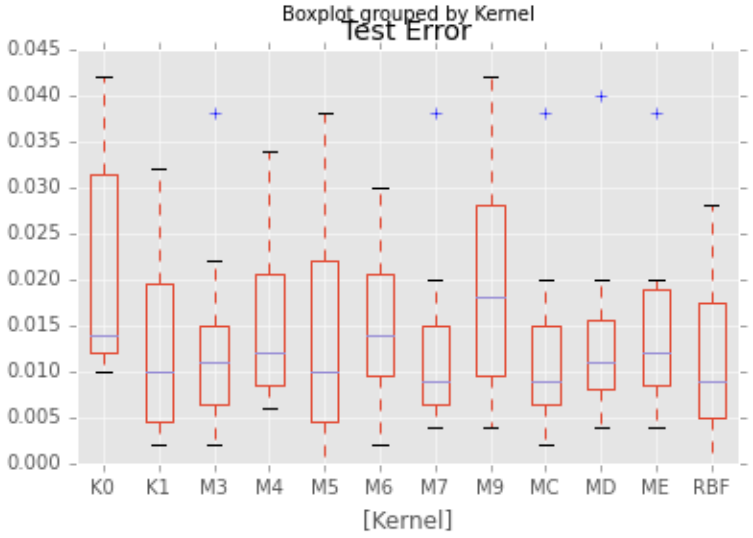


FIGURE 4.3: Test error distribution on the Mushrooms data set.

4.2.3.2 Soybean Data Set

A data set with 19 classes. There are 35 categorical attributes, some nominal and some ordered, but all were treated as categorical. The results for 5 iterations are show in Figure 4.4.

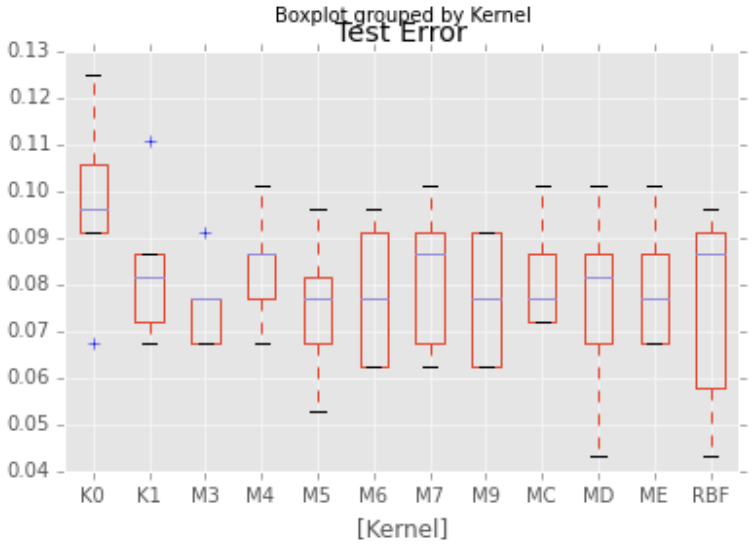


FIGURE 4.4: Test error distribution on the Soybean data set.

4.2.3.3 Car Evaluation Data Set

Car Evaluation Database was derived from a simple hierarchical decision model originally developed for the demonstration of DEX, M. Bohanec, V. Rajkovic: Expert system for decision making. The results for 5 iterations are show in Figure 4.5.

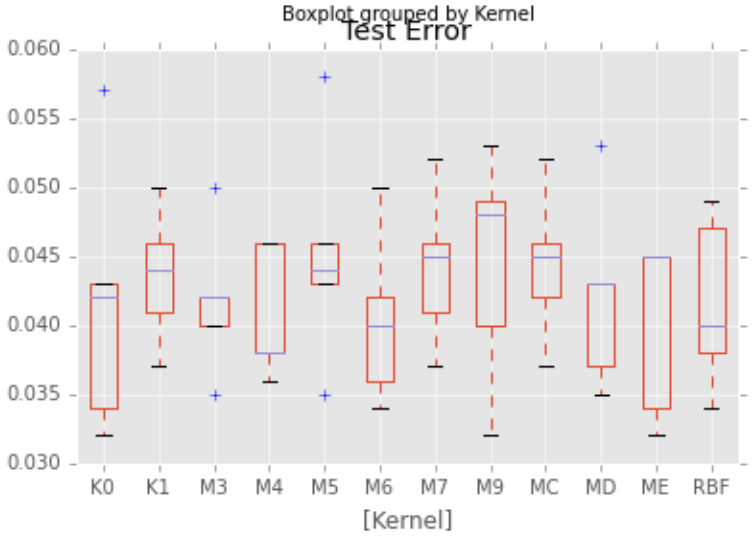


FIGURE 4.5: Test error distribution on the Car Evaluation data set.

4.2.3.4 TicTacToe Data Set

This database encodes the complete set of possible board configurations at the end of tic-tac-toe games, where "x" is assumed to have played first. The target concept is "win for x" (i.e., true when "x" has one of 8 possible ways to create a "three-in-a-row"). The variables correspond to the squares and there are three possible categories "o", "x" and "b". The results for 5 iterations are show in Figure 4.6.

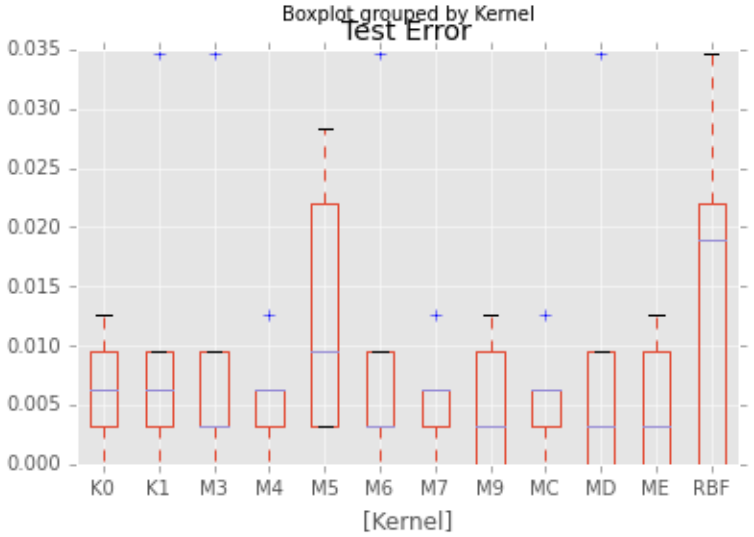


FIGURE 4.6: Test error distribution on the TicTacToe data set.

4.2.3.5 Congressional Voting Data Set

This data set includes votes for each of the U.S. House of Representatives Congressmen on the 16 key votes identified by the CQA. The CQA lists nine different types of votes: voted for, paired for, and announced for (these three simplified to yea), voted against, paired against, and announced against (these three simplified to nay), voted present, voted present to avoid conflict of interest, and did not vote or otherwise make a position known (these three simplified to an unknown disposition). This would have been more interesting if it had preserved the original nine types of votes, alas.

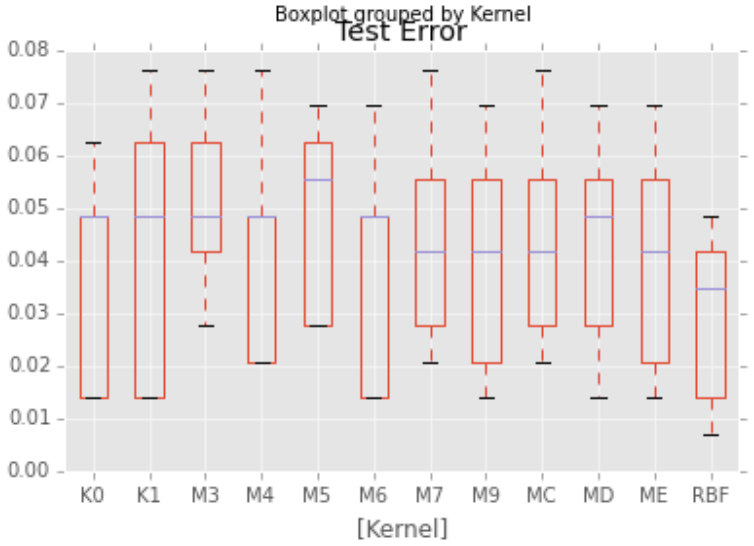


FIGURE 4.7: Test error distribution on the Congressional Voting data set.

4.2.4 Results

The categorical kernels behave quite well across all data sets, but the big advantage that was shown in the synthetic data set was not replicated in the real data sets. The kernel k_0 itself behaves very poorly in the Soybean data set and quite well in the Car Evaluation one. This pattern is repeated with all the kernels across different data sets. In fact, the apparent conclusion from the results that no kernel is strictly better than the others. One could of course add more data sets and perform a more elaborate ranking and one of the kernels will come slightly ahead of the others, and check whether there is a statistically significant difference, but while very rigorous it seems a far shot.

Most likely, the situation is the same as with the original similarity coefficients studied by Gower and Legendre. There are many variations and one needs to know the consequences of the small differences in the formulas in order to choose one or the other depending on what is more appropriate for the problem at hand.

The idea is that with these kernels, there is a wider range of choices. Instead of being limited to the common procedure of converting categories to dummy variable form, one can use a kernel that actually makes sense, and that has the potential to learn from statistical information that is ignored when using RBF or Overlap.

Chapter 5

Design and Implementation

This section covers the process and decision making involved in the development of this project, as well as the organization of the system and some distinct details of its implementation.

5.1 Planification

The following table shows the initial breakdown of tasks to develop this project and an estimation of how much time they should take to complete.

Task	**Time (hours)**
Project definition	**10**
Study and research	**120**
Learn the foundations of kernels and kernel methods	100
Study previous work on the use of kernels on categorical variables	50
Analysis and design	**150**
Develop new kernels for categorical data	90
Specify and design the system to develop the kernels	60
Implementation	**180**
Implement the kernels	50
Implement a library and software to test and analyse the kernels	130
Testing	**120**
Documentation	**90**
Total	**670**

This breakdown is the basis for the cost analysis.

5.2 Costs

Usually any project would need to account for costs in terms of hardware, licenses, personnel and such. For this project, developed as a student thesis, cost analysis becomes rather fuzzy. Thus, the approach for this section is going to be to try and estimate the costs that a similar project would have if a profit-seeking enterprise were to develop something similar, with the understanding that in that kind of scenario would require a more strict and thorough cost analysis.

All hardware required to develop this project is commonplace. The specifications for the computer used in development are the following:

1. Dual Core processor at 3.5 GHz.
2. 4GB of RAM
3. 128GB SSD
4. Motherboard, graphics card, power supply, and other components
5. Monitor, keyboard, mouse and other peripherics

The hardware available in the Faculty installations is usually more powerful than this.

In our study case scenario, an enterprise would need to provide similar hardware, as well as the working space. An estimate of €1500 per developer should be enough to cover those expenses during the development period.

Even though that would suffice for development, the development machines might not the best suited to run the test suites, specially if they need to run overnight. That may warrant buying an expensive server in some cases, but in most it would be enough to just rent a machine from an online service. Using Amazon Compute Cloud prices to gather an estimate, in our scenario we will consider that €500 would cover the costs of renting a sufficiently powerful machine for a few days.

Another source of expenses would usually be software licenses, but in our case we planned the project to be built on open source from the start, which makes the cost of software licenses equal to zero.

Some of the tools and services used, such as Github, Dropbox, or the backups system are free for personal or academic use, but they are not free for commercial use. A quick look shows that the prices of some of this services are considerably expensive – $100 a

month for GitHub, $750 a year for DropBox–, so we will add an extra €2000 of *other costs*.

The expenses would depend on the number of developers working on the project, we will go with three, the total adds up to:

Item	Amount	Cost
Desktop	3	€4500
Server	1	€500
Other costs	1	€2000
Total		€7000

Now, the other expense that needs to be taken into account is the cost of labour. You need at least one qualified person to learn about SVMs and develop custom kernels. We can give a computer scientist a couple of months of full time work to do that. The actual implementation and coding of the system can be done by any programmer that knows their algebra without dwelling into the specifics of SVMs, kernel methods and Mercers' theorem, the same goes for implementing the tests, but they would need someone with experience to supervise the work. That means that a team of one engineer and two programmers should be able to develop such a project. If the same person that designs the kernels doubles as the manager, the whole process could consist of two months of full time research by the computer scientist, then one more month of team development with the two programmers, to implement the kernels, develop the tests and document the results.

The cost would be divided as such:

Position	Hours	Hourly Rate	Cost
Research/Analisys	480	€30	€14400
Programming	320	€15	€4800
Total	800	€24	€19200

The €19200 of labour, plus the €7000 of expenses add up to €26200.

This is rather expensive, specially if done from scratch and as a one time only investment. Most of the cost is associated with having someone doing the research for three months. If someone needed to implement a learning algorithm tailored to a very specific task, it would be more efficient to find someone with a solid background in the subject, but it is debatable whether that would be any cheaper or not.

5.3 Requirements

We need to devise and implement a system that provides the user with the tools and means to cover all the objectives described in the introduction. This requirements are what defines the constraints in the design and implementation of such system.

5.3.0.1 Functional requirements

1. Provides ready to use categorical kernels (the full list is in the *kernels* section).
2. There is an easy way to modify the available kernel functions.
3. Allows to use custom kernel functions.
4. Provides tests on ready to use datasets.
5. Provides the means to easily compare and analize the results of different kernels on any dataset.
6. There is an easy way to use custom datasets.

5.3.0.2 Nonfunctional requirements

Since development time is finite, it is important to define a set of nonfunctional requirements stay focused during development. Usually the choice requires some compromise of which are the qualities that are most important for our system. In our case

1. **Maintainability**: it should be easy to isolate errors or find their cause to correct them.
2. **Extensibility**: it should be easy to make changes or add new components without having to change the already existing code.

Other requirements such as robustness, efficiency, portability, etc. should not be neglected, but will always be second to the two listed before.

5.4 Development enviroment

With a well-defined list of requirements, the next step is to choose a development environment. The first factor to take into account to make the decision are the resources

available. Since the development machine runs on Linux, the choice is quickly narrowed down to programming languages and libraries that support Linux. The bright side of this is that most environments that support Linux are also multi-platform. Another important factor is that we want it to be free software, mostly because of the non-existent budget, but also because it aligns with the desire to build a system that anyone interested can check out or modify.

Another thing to consider is familiarity, but there many frameworks in popular languages like C++, Java, Python and R. With this we narrow down the choices to a few well-known machine learning packages like OpenCV (C++), Weka (Java), Scikit-learn (Python), e1071 or Kernlab (R). As it turns out, the SVMs in most of these packages are based on LibSVM, so chosing a language ended up being a matter of personal preference.

The final choice was Python. The reasons were familiarity with the language and ease of development over Java and C++. R would have been an equally good choice for the same reasons, but the final decision was made for Python –for unrelated reasons [1]–.

5.5 Design

The specification is ideally technology independent but it is worth mentioning that, while Python is object-oriented, it is also multi-paradigm, uses dynamic typing with strong types and doesn't use interfaces. We wanted to do the specification according to what is taught at FIB, so we prepared a basic schematic in UML. UML however is better suited for more traditional object oriented languages like Java, and if you were to compare the diagrams in this project with the actual code you would realize that there are some slight differences in order to make the code more in line with the Python programming guidelines. I kept the diagram as it is because the relationships between classes are still the same and using fancy UML structures that fit better with Python is likely to end up being counter-productive. And besides, the specification in this project is very, very straightforward. It is shown in Figure 5.1.

The Kernels module will be explained first.

The scikit-learn library, which is used as base on which the custom kernels of this project are programmed, has a well defined structure. Classifiers are implemented as a class, and any instance has two main methods: `fit` which takes a data matrix and a class vector and `predict` which takes a data matrix and returns a predicted class vector. There was a minimal example of this in Chapter 1.

[1] Python is used in systems scripting, which was useful for another project that was to be developed along this one.

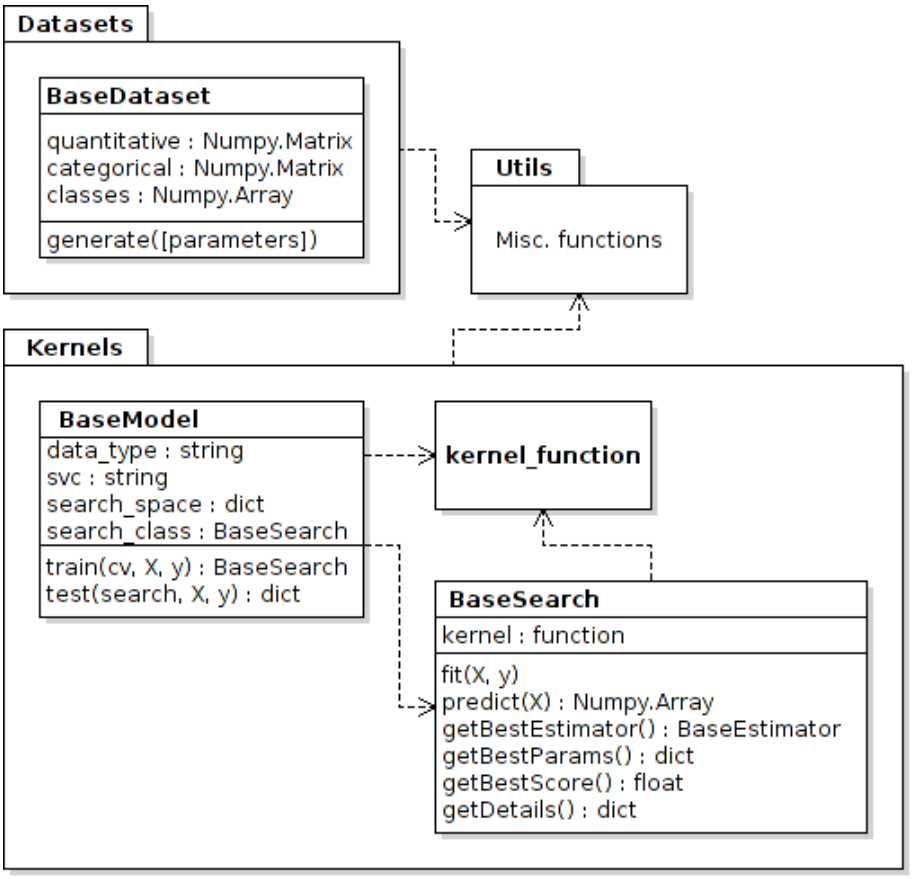


FIGURE 5.1: The design of the library.

Scikit-learn also provides a GridSearch helper that searches the best parameters for a classifier. Unfortunately, this only supports the classifiers available in scikit-learn and if to use GridSearch with custom classifiers requires to implement the grid search too.

To overcome this, a new class BaseSearch was created, based on GridSearch, but extended to support the kernels implemented in this project. This new class takes care of fitting the model with any custom kernel, performing grid search to find the best parameters using cross-validation and running predictions. It is a wrapper that combines the functionality of a predictor and the original grid search helper from scikit learn so that other kernels can be used easily.

The other class is BaseModel. This class has two functions: `train` and `test`. As it can be deduced from the names of these two functions, it abstracts the process of training a classifier using a dataset, and then testing the dataset. To use a BaseModel it is necessary to define:

1. A search class (a class that inherits from BaseSearch and can search parameters for a kernel).

2. A search space (a python dictionary with all the paremeters that are to be searched through).

3. svc and data_type (this only needs to be specified to use an original kernel from scikit learn, all the custom kernels in this project use `svc='precomputed'` and `data='categorical'`.

The module kernel function contains just the kernel functions themselves. Every kernel function is associated to a Search model, that implements the fitting and grid search for that particular kernel. The implementation can vary from some kernels to others, as different kernels have different parameters. For a more detailed description, refer to the documentation of the Python module (see section below about the documentation).

Finally, there is a Dataset class, which either generates a dataset in the case of artificial datasets, or loads the data in the case of external data sets. It allows to retrieve the information in two formats: straight as categorical data, or in dummy variable form –mainly so that it can be fed to the RBF kernel–. In the case of artificial datasets like Synthetic and GMonks, it will be necessary to pass some parameters. How to do this is also detailed in the documentation.

5.6 Kernel implementation

The kernels are implemented as Python functions that receive two data matrices $A = (a_{ik})$ and $B = (b_{jk})$, with sizes $N_A \times n$ and $N_B \times n$ respectively. A row is a vector of size n, in our case, a row is generally an example with n categorical variables. The function returns a matrix of size $N_A \times N_B$ with the value of computing the kernel function K between each pair of rows $C = (K(\mathbf{a}_i, \mathbf{b}_j), 1 \leq i \leq N_A, 1 \leq j \leq N_B)$. When A and B are the same matrix the resulting C will be the Gram matrix.

The following code is the implementation of the kernel function m_3. The code has been slightly edited. Three lines that deal with the parameters have been edited, and the names of the variables have been changed to be in line with the ones used in the rest of the document.

```
def _m3(A, B, Ap, Bp, h, prev, post):
    Na, n = A.shape
    Nb, n = B.shape
    G = np.zeros((Na, Nb))
    hAp = h(Ap)
    hBp = h(Bp)
    for r in range(Na):
        C = np.tile(A[r], (Nb, 1))
        hCp = np.tile(hAp[r], (Nb, 1))
```

```
        u = 2.0 * np.sum(prev(hBp * (C == B)), axis=1)
        v = np.sum(prev(hCp) + prev(hBp), axis=1)
        G[r, :] = u / v
    return post(G)
```

The most important parameters are:

1. A: data matrix of size N_A, each row is a vector with n categorical values.
2. B: data matrix of size N_B, each row is a vector with n categorical values.
3. Ap: probability matrix, $Ap = (P(a_{ik}))$.
4. Bp: probability matrix, $Bp = (P(b_{jk}))$.
5. h: the inverting function.
6. *prev*: the *prev* transformation function.
7. *post*: the *post* transformation function.

The algorithm broken down step by step:

1. Na is the number of rows in matrix A.
2. Nb is the number of rows in matrix B.
3. Declare an empty matrix C of size $Na \times Nb$.
4. Apply h to the matrix A_p, the resulting matrix is $h(A_p) = (h(P(a_{ik})))$.
5. Apply h to the matrix B_p, the resulting matrix is $h(B_p) = (h(P(b_{jk})))$.
6. Iterate index r from 1 to Na, we will fill C one row at a time.
7. Create $C = (c_{jk})$ of size $Nb \times n$, where each row is the r-th row of matrix A.
8. Create $h(C_p) = (h(P(c_{jk})))$, where each row is the r-th row of matrix $h(A_p)$.
9. Compute
$$\mathbf{u} = 2 \sum_{k:a_r=b_j} f^2\left(a_{rk}\right) = 2 \sum_{k=1}^{n} \left[h(P(b_{jk})) * [c_{jk} = b_{jk}]\right]$$
the result is a vector of length Nb.
10. Compute
$$\mathbf{v} = \sum_{k=1}^{n} \left[f^2\left(a_{rk}\right) + f^2\left(x_{jk}\right)\right]$$
the result is a vector of length Nb.

11. Compute $\mathbf{u}/\mathbf{v}$, set as row r in G.
12. Iterate until all rows in C are done.
13. Apply the *post* function and return the matrix.

The implementation of the other kernels is very similar to this one.

5.7 Library

One of the strong points of Python is the wide range of available libraries covering many domains of computer science and engineering. Python libraries come usually in the form of Python packages that can be readily downloaded and installed. Now, to understand what package means it is necessary to know a bit of Python.

Python code is organized into modules. A module may consist of either a single Python (.py) file, or a directory containing one or more sub-modules. Any Python file can be used and imported locally as a module. A module or group of modules that are distributed together are called a package. A package is slightly different from normal modules because it usually has a few configuration files that are used to distribute and install the package across different systems.

Python finds modules by looking into a list of folder that can be found in `sys.path`. A usual Python path will look like this:

```
>>> import sys
>>> print(sys.path)
['',
 '/usr/bin',
 '/usr/lib/python33.zip',
 '/usr/lib/python3.3',
 '/usr/lib/python3.3/plat-linux',
 '/usr/lib/python3.3/lib-dynload',
 '/usr/lib/python3.3/site-packages',
 '/usr/lib/python3.3/site-packages/IPython/extensions']
```

Those are all the directories where Python looks for modules. It seaches them in order until it finds a module that matches the import.

The path can be modified, the common way to do so is either directly (although that is not recommended) or by setting the `PYTHONPATH` environment variable.

But if you want to distribute a module so that others can use it easily with their own projects, the best way to do so is by creating a package. The purpose of a package is to make the code easy to distribute in one of two ways:

1. Using Python specific tools like `pip`.
2. Directly from the source files.

In Linux, there's also the possibility of using distribution speficic tools (such as deb, pkgbuild, rpm, etc.). But the previous ones have the advantage of being platform-indepemendent (and they are compatible with distribution specific tools).

The objective of all these methods is the same: provide an easy way to copy the Python modules into the standard module search location. In the end, a package IS a Python module, but with those additions to simplify the task of copying it to the path, dealing with dependencies and so on.

With this in mind, it was considered considered that the best approach for this project was to implement it as a Python module, and make it available as a ready-to-install package. That way it would give me a solid base to satisfy the requirements of extensibility and maintainability required in this project[2].

There are various tools available to do packaging, but for a basic library all that is needed is included as part of the Python standard libraries. The whole process is well documented and it turned out to be simpler than expected.

5.7.1 Installation

The installation process in a Linux box would be as follows:

```
$ git clone https://github.com/Alkxzv/categorical-kernels.git
$ cd categorical-kernels
$ python setup.py install
```

There is a catch: this assumes that there is a recent version of Python installed in the system, as well as the software Git. For people that work often with Python or have a modern Linux system that will not be a problem, but that may not be the case.

To cover the most general cases, there is an appendix with detailed instructions on how to install the whole setup in a fresh Ubuntu.

5.7.2 Documentation

The Python language has built-in tools for documentation, mainly through doc-strings. When defining a function, one can append a string that describes its use, parameters,

[2]It was also a perfect chance to learn how packaging works in Python, which is something that I had wanted to learn for some time, as I think it can be useful in future to write more reusable software

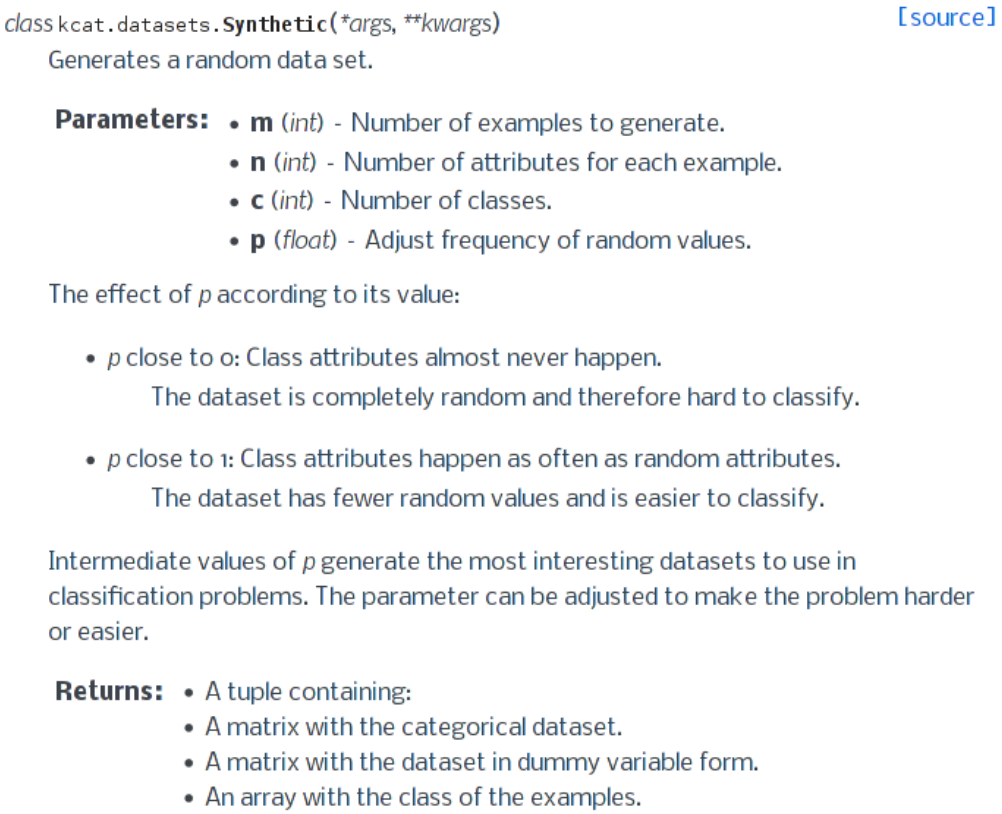
the returned values with as much detail as deemed necessary. Then there are tools like Sphinx, that parse the code and generate documentation in HTML that can be navigated, indexed and searched. This is very convenient because it allows to keep the code and documentation together in the same code base, easing the task of keeping the documentation complete and up-to-date. The fact that it's integrated with the code is extremely useful when using the library with an IDE or with Notebooks, as the users will get prompts with relevant information as they type.

Sphinx is used in the documentation of this project. Every function and class is well documented [3], the following is the doc-string for the function that generates the Synthetic dataset as it is included in the source code, the resulting HTML generated by Sphinx can be seen in Picture 5.2.

```
Generates a random data set.

Args:
    m (int): Number of examples to generate.
    n (int): Number of attributes for each example.
    c (int): Number of classes.
    p (float): Adjust frequency of random values.

The effect of *p* according to its value:

- *p* close to 0: Class attributes almost never happen.
    The dataset is completely random and therefore hard to classify.
- *p* close to 1: Class attributes happen as often as random attributes.
    The dataset has fewer random values and is easier to classify.

Intermediate values of *p* generate the most interesting datasets to
use in classification problems. The parameter can be adjusted to make
the problem harder or easier.

Return:
    - A tuple containing:
        - A matrix with the categorical dataset.
        - A matrix with the dataset in dummy variable form.
        - An array with the class of the examples.
```

The entire API documentation will not be included in this document, as it is meant more as a reference when using the library, not something that should be read in its entirety. It's also much easier to naviage the HTML, which allows you to search for a specific function and jump to the actual source code.

[3] The is available through the repository in http://github.com/alkxzv/categorical-kernels/.



class kcat.datasets.**Synthetic**(**args*, ***kwargs*) [source]

Generates a random data set.

Parameters:
- **m** (*int*) - Number of examples to generate.
- **n** (*int*) - Number of attributes for each example.
- **c** (*int*) - Number of classes.
- **p** (*float*) - Adjust frequency of random values.

The effect of *p* according to its value:

- *p* close to 0: Class attributes almost never happen.
  The dataset is completely random and therefore hard to classify.
- *p* close to 1: Class attributes happen as often as random attributes.
  The dataset has fewer random values and is easier to classify.

Intermediate values of *p* generate the most interesting datasets to use in classification problems. The parameter can be adjusted to make the problem harder or easier.

Returns:
- A tuple containing:
- A matrix with the categorical dataset.
- A matrix with the dataset in dummy variable form.
- An array with the class of the examples.

FIGURE 5.2: The documentation generated in HTML.

5.8 Tools

During the development of the project, it proved useful to write scripts to automate some tedious tasks such as running tests and generating graphics. As the project went on new functionality was added to those scripts and they ended packaged into a couple of programs that can be run through the Command Line Interface. This two applications provide a lot of functionality that, while very useful, is specific to the purpose of this project, which is why I decided not to include them in the library with the kernels and data sets, but are still in the repository.

5.8.1 Experiment Tool

This first application provides functionality automate the process of generating datasets, training models and collecting data that can later be used to compare the different kernels. The application can run experiments in batch and is parallelised, which means that the whole proces of doing experiments is reduced to executing a command, and the rest is taken care of.

It's written to integrate with the kernel library and any new kernel that is propertly added to the library can be processed without need to modify the application.

Not only that, but the program provides a number of parameters that allow to test almost any configuration that would be available with any of the synthetic datasets implemented in this project. Adding new datasets would require a bit of boilerplate code, but the process should be intuitive.

All the experiments in this project can be run using this tool, which means that every result in this document can be easily reproduced and the results can be checked anytime without need to write any code.

The interface should be intuitive. Typing `./run.py --help` will prompt the users with all the information they need, here's an extract from the program help:

```
usage: run.py [-h] [-v] [-o OUTPUT] [-i ITERATIONS] [-s RANDOM_STATE]
              [-r TRAIN_SIZE] [-e TEST_SIZE] [-f FOLDS]
              {Synthetic,GMonks,WebKB} ...

positional arguments:
  {Synthetic,GMonks,WebKB}

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         show progress
  -o OUTPUT, --output OUTPUT
                        filename where the output should be saved
  -i ITERATIONS, --iterations ITERATIONS
                        number of repetitions
  -s RANDOM_STATE, --random-state RANDOM_STATE
                        seed for the random state
  -r TRAIN_SIZE, --train-size TRAIN_SIZE
                        number of elements in the train set
  -e TEST_SIZE, --test-size TEST_SIZE
                        number of elements in the test set
  -f FOLDS, --folds FOLDS
                        number of folds to use in cross-validation
```

The user can choose one of the available datasets and can use the parameters to change the test/training size, the number of iterations to run in batch, and so on.

Dataset specific parameters can be found by typing `./run.py Dataset -h` , they allow to change the number of attributes, or classes or other parameters, depending on the dataset.

The application saves the results to a plain text file in json format.

5.8.2 Visualization Tool

This is another useful command line tool. This one takes the file produced by `run.py` and generates graphics from the data.

By default it plots the training and test error for each kernel. It has options to group the data by any attribute and filter by kernel. For example, by specifying `--group-by gamma --kernel RBF` it plots the error for each value of gamma when using the RBF kernel. There are some additional options to generate other plots, like `--synthetic` that generates a plot to visualize the effect of the parameter p, among others.

Again, all the plots in this project have been generated by this tool, making it very easy to add new kernels and compare the results.

The whole set of options is the following:

```
usage: plot.py [-h] [-o OUTPUT] [-k KERNEL] [-d DATASET] [-g GROUP_BY]
               [--synthetic] [--gmonks]
               FILENAME

positional arguments:
  FILENAME              json file containg execution data

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        directory or prefix used to save the images
  -k KERNEL, --kernel KERNEL
                        filter results to show only one kernel
  -d DATASET, --dataset DATASET
                        filter results to show only dataset
  -g GROUP_BY, --group-by GROUP_BY
                        plot results grouped by the passed attribute
  --synthetic           plot graphs for synthetic dataset
  --gmonks              plot graphs for gmonks dataset
```

5.9 Notebooks

It's common for mathematical and statistical tools such as R, Mathematica, Maple, and others, to provide an interactive environment that integrates code input, output and plots. Python's default interpreter is very basic and it can't compete with such tools in terms of ease of use and functionality.

However, there is a Python package that covers that lack of interactive environment, called IPython. IPython comes with a utility called Notebooks which provide a web-based interactive environment that allows to combine code execution, text, formulas and

plots into a single hypertext document. These notebooks are plaintext files and they can be included in version control systems and they can be shared and exported to HTML or PDF.

All the experiments in this project are stored as IPython notebooks, and they are available at http://github.com/alkxzv/categorical-kernels/, this allows to exactly reproduce the experiments and check the results, make corrections or additions by simply forking the repository.

Chapter 6

Conclusions

We declared the objectives for this project in the first chapter. To some degree all of them were satisfied:

We studied the state of the art in kernel functions for categorical variables, the most common were used in this project as a reference when comparing our results.

We defined a new family of kernels that take advantage of probabilistic information in the data. Tested the performance of the different kernels with specially designed data sets. In particular, one widely used dataset (Monks) and a newly designed (Synthetic). We also testes the performance of the different kernels with data from real-life problems, as many as we could in a reasonable amount of time, even automating the whole process.

We analyzed and compare the results of the different kernels on the different data sets.

We developed a set of tools to easily reproduce the experiments to replicate the results. And implemented them using a framework in a way hat allows to easily tweak, extend or derive the kernels to explore the results. It is hard to judge how easy it would be for someone unfamiliar with the project, but with some guidance anyone should be able to have everything installed with a few commands, make any changes they want, run the automated tests and compare their results. The whole thing should take only a few minutes, excluding the installation of Python an Scipy.

Those were the original goals stated in the introduction, and defined during the first month of the project. All in all, the project was quite focused to cover all the goals, and although it took more time than what it was originally planned everything seems to have been reasonably covered.

6.1 Future work

During the development of the project some interesting ideas came up that would be a good direction for future developments.

1. The similarity measures are intended for binary variables. When they were adapted for categorical variables, a rule for the match $(0, 0)$ in binary variables was provided based on what seemed to be a reasonable choice. However, there is no certainty as to whether the choice was the most appropriate, it would be interesting to try and compare other solutions to see how they work.

2. We only proved one function of the multivariate kernels to be a valid kernel. We provided empirical proof that the other functions work as kernels too, as we were able to train SVMs using them, and in some cases that may be enough for someone to use it in a model, but this is insufficient if we want to affirm that they are valid. We were able to build one proof, and Gower and Legendre showed that the original similarity measures are all p.s.d., so it stands to reason that it should be possible to find valid proofs for the other kernels.

Some things are not directly related to the project, but are interesting ideas nonetheless.

1. The scikit-learn library could be improved to better integrate custom kernels. The grid search functionality doesn't work out of the box with any kernel that is not a default, which would have spared us quite a bit of work.

2. There are many interesting data sets available at the *UC Irvine Machine Learning Repository*, it would be great if instead of having to download the files, they were wrapped in a library with a nice API, although this may not be viable for legal reasons.

Appendix A

Demonstration

Let

$$\{x_1, x_2, \cdots, x_N\}$$

be data vectors in V^n, being

$$V^n = V_1 \times V_2 \times \cdots \times V_n$$

$$V_i = \{v_{i1}, v_{i2}, \cdots, v_{iM_i}\}$$

Where Mi is the number of modalities in V_i.

$$m_3(\mathbf{x_i}, \mathbf{x_j}) = \frac{\sum\limits_{k:x_{ik}=x_{jk}} f(x_{ik})}{\sum\limits_{k=1}^{n} [f(x_{ik}) + f(x_{jk})]}$$

Theorem: the kernel m_3 is p.s.d.

The proof will be split in two parts. First, we define the matrices A and B of size $N \times N$, such that:

$$A = (a_{ij}) = \left(2 \sum_{k:x_{ik}=x_{jk}} f(x_{ik}) \right)$$

and

$$B = (b_{ij}) = \left(\frac{1}{\sum\limits_{k=1}^{n} [f(x_{ik}) + f(x_{jk})]} \right)$$

so that

$$m_3 = A * B$$

First, we prove that A is p.s.d.

$$\begin{aligned} a_{ij} &= 2 \sum_{k:x_{ik}=x_{jk}} f\left(x_{ik}\right) \\ &= 2 \sum_{k:x_{ik}=x_{jk}} \left[f\left(x_{ik}\right) + f\left(x_{jk}\right)\right] \\ &= \phi_f(\mathbf{x}_i)^T \phi_f(\mathbf{x}_j) \end{aligned}$$

Where ϕ_f is a function that maps a vector z in input space in the following fashion:

$$\phi_f(z) = (\widetilde{\phi}_k(z_k))_{k=1,\cdots,n}$$

$$\widetilde{\phi}_{fk}(z_n) = (0, \cdots, 0, \sqrt{2} f\left(z_k\right), 0, \cdots, 0)^T$$

$\widetilde{\phi}_{fk}$ is a map from a categorical value to a vector. To do this mapping it is necessary to define an order among the categories in V_k –any order as long as it is maintained–, then z_k is represented as a vector of zeroes, except for the position corresponding to the index assigned to z_k when ordering the categories which is equal to $\sqrt{2} f(z_k)$.

With this, ϕ_f constructs a vector that is the concatenation of the vectors resulting from mapping each category z_k.

Since the order of the categories is fixed, when computing $\phi_f(\mathbf{x}_i)^T \phi_f(\mathbf{x}_j)$ the result will be $2f(x_{ik})$ for the categories that match, and 0 for the ones that not.

This completes the proof that there is a space with a dot product, which is enough to prove that A is p.s.d..

The next part is to prove that the denominator B is p.s.d.. This is based on a demonstration by Gower and Legendre in *Metric and Euclidean properties of dissimilarity coefficients* [8].

They prove that

$$x_{ij} = 1/(x_i + x_j)$$

by showing that

$$detX = \frac{\prod_{i \neq j}^{n} \left(\frac{x_i - X_j}{x_i + x_j}\right)^2}{2^n \prod_{i=1}^{n} x_i}$$

is p.s.d. for non-negative $\mathbf{x}_i$. We can use this result to justify that B is p.s.d., but with the limitation that $f(x)$ must be non-negative, which is true for the $f(x) = h_\alpha(P(x))$ used in all kernels in this project because the PMF is obtained from the sample and is always $0 < P(x) < 1$.

Appendix B

Kernel Expansions

B.1 Multivariate Kernels form Similarity Measures

$$S_6 = \frac{a+d}{a+2(b+c)+d}$$

$$m_6(\mathbf{x},\mathbf{y}) = \frac{2\sum\limits_{i:x_i=y_i} f^2(x_i) + \sum\limits_{i:x_i\neq y_i}\left[1-f^2(x_i)+1-f^2(y_i)\right]}{2\sum\limits_{i:x_i=y_i} f^2(x_i) + 2\sum\limits_{i:x_i\neq y_i}\left[f^2(x_i)+f^2(y_i)\right] + \sum\limits_{i:x_i\neq y_i}\left[1-f^2(x_i)+1-f^2(y_i)\right]}$$

$$S_7 = \frac{a}{a+1/2(b+c)}$$

$$m_7(\mathbf{x},\mathbf{y}) = \frac{2\sum\limits_{i:x_i=y_i} f^2(x_i)}{2\sum\limits_{i:x_i=y_i} f^2(x_i) + \frac{1}{2}\sum\limits_{i:x_i\neq y_i} f^2(x_i) + f^2(y_i)}$$

$$S_9 = \frac{a-(b+c)+d}{a+b+c+d}$$

$$m_9(\mathbf{x},\mathbf{y}) = \frac{2\sum\limits_{i:x_i=y_i} f^2(x_i) - \sum\limits_{i:x_i\neq y_i} f^2(x_i) + f^2(y_i) + \sum\limits_{i:x_i\neq y_i} \left[1-f^2(x_i)+1-f^2(y_i)\right]}{\sum\limits_{i=1}^{n} f^2(x_i) + f^2(y_i) + \sum\limits_{i:x_i\neq y_i} \left[1-f^2(x_i)+1-f^2(y_i)\right]}$$

$$S_{12} = \frac{a}{\sqrt{(a+b)(a+c)}}$$

$$m_12(\mathbf{x},\mathbf{y}) = \frac{2\sum\limits_{i:x_i=y_i} f^2(x_i)}{\sqrt{\left(\sum\limits_{i=1}^{n} f^2(x_i) + \sum\limits_{i:x_i=y_i} f^2(y_i)\right)\left(\sum\limits_{i:x_i=y_i} f^2(x_i) + \sum\limits_{i=1}^{n} f^2(y_i)\right)}}$$

Appendix C

Installation

C.1 Installation in Ubuntu

The installation process depends on your operating system. The code has been tested in Xubuntu and Arch Linux. This guide details the whole setup process in a Xubuntu 13.04 fresh installation. It should be the same in other variations of Ubuntu and very similar in most Linux distributions.

The basic requirement is the package python3.3 or higher, which in the latest Ubuntu distributions is already installed.

Run the following command in a terminal to install the basic requirements:

```
apt-get install git python3-pip python3-numpy python3-scipy
```

For processing the results and generating charts, there are additional requirements:

```
apt-get install python3-pandas python3-matplotlib
```

If the user prefers to install them in a virtual environmet instead of directly to the system, all the Python packages can be found in pypi and installed with pip. This guide will not require virtualenvs for simplicity.

Now clone the project git repository:

```
git clone https://github.com/Alkxzv/categorical-kernels.git
```

Change the working directory to the new folder:

```
cd categorical-kernels
```

Install the library:

```
python3 setup.py install
```

Done, try running:

```
python3 experiments/run.py -v Synthetic
```

And you should see the first results. There is a chapter in the document that explains how to run the different tests and experiments.

C.2 Installation in Domino

Domino is a service that allows to run Python, R and Matlab code on the Amazon cloud service EC2. The advantage of using Domino is that for a small over-cost they take care of setting up and managing the servers. How much of an advantage is made obvious by showing the whole process to setup this project.

First, set up an account at domino and install the client, you can find how to do that on their site.

Next, follow this steps:

```
git clone https://github.com/Alkxzv/categorical-kernels.git
```

```
domino create domino-experiments
```

```
cp -R categorical-kernels/experiments/* domino-experiments
```

```
cd domino-experiments
```

```
domino run example.sh
```

That is all, the project will be running in the server. When the execution ends you will get an email with the results as an attachment.

Bibliography

[1] B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, 2002.

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

[3] Bernhard. Schlkopf, Koji. Tsuda, and Jean-Philippe. Vert. *Kernel methods in computational biology*. MIT Press, Cambridge, Mass., 2004.

[4] Llus A. Belanche and Marco A. Villegas. Kernel functions for categorical variables with application to problems in the life sciences. In Karina Gibert, Vicent J. Botti, and Ramn Reig Bolao, editors, *CCIA*, volume 256 of *Frontiers in Artificial Intelligence and Applications*, pages 171–180. IOS Press, 2013. ISBN 978-1-61499-320-9. URL http://dblp.uni-trier.de/db/conf/ccia/ccia2013.html#BelancheV13.

[5] Carl H. FitzGerald, Charles A. Micchelli, and Allan Pinkus. Functions that preserve families of positive semidefinite matrices. 221(1–3):83–102, May 1995. ISSN 0024-3795. URL http://www.elsevier.com/cgi-bin/cas/tree/store/laa/cas_sub/browse/browse.cgi?year=1995&volume=221&issue=1-3&aid=9300232.

[6] Jierui Xie, Boleslaw K. Szymanski, and Mohammed J. Zaki. Learning dissimilarities for categorical symbols. In Huan Liu, Hiroshi Motoda, Rudy Setiono, and Zheng Zhao, editors, *FSDM*, volume 10 of *JMLR Proceedings*, pages 97–106. JMLR.org, 2010. URL http://dblp.uni-trier.de/db/journals/jmlr/jmlrp10.html#XieSZ10.

[7] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R.S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van De Welde, W. Wenzel, J. Wnek, and J. Zhang. The monk's problems a performance comparison of different learning algorithms. Technical report, 1991.

[8] J. C. Gower and P. Legendre. Metric and euclidean properties of dissimilarity coefficients. *Journal of Classification*, 3(1):5–48, March 1986. URL http://dx.doi.org/10.1007/BF01896809.