UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

AN IMPLEMENTATION OF FLEXIBLE RBF NEURAL NETWORKS

Fernando Manuel Pires Martins

MESTRADO EM INFORMÁTICA

2009

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

AN IMPLEMENTATION OF FLEXIBLE RBF NEURAL NETWORKS

Fernando Manuel Pires Martins

DISSERTAÇÃO
Projecto orientado pelo Prof. Doutor André Osório Falcão

MESTRADO EM INFORMÁTICA
2009

Acknowledgements

I wish to acknowledge those who, either by change or simple misfortune, were involved in my MSc.

I start by acknowledging Professor André Falcão for the trust, the guidance and the patience shown for the long months this work took, never giving up on me, even when the bad times seemed to put an early end to my MSc.

I wish to acknowledge my good friend Paulo Carreira, who has been an inspiration and who has always supported and pushed me into the final goal of writing this dissertation.

A special thanks goes for my family. There are no words to express my gratitude nor to express how fantastic they were in supporting me. Without their support, this dissertation would certainly not exist.
I wish to thanks to my parents Julia and Fernando, for the support and education that allowed me to come this far, and to my grandmother Maria Teresa, for her inspiring reading and writting passion.
Finally, I have an endless debt to my beloved wife Vilma, which is responsible for, almost magically, getting me the time to work on my MSc. I also have a time debt to my daughters, Sofia and Camila, for the countless times that I had to refuse to play with in this last four years.

To my beloved family, Vilma, Sofia and Camila.

Resumo

Sempre que o trabalho de investigação resulta numa nova descoberta, a comunidade científica, e o mundo em geral, enriquece. Mas a descoberta científica *per se* não é suficiente. Para beneficio de todos, é necessário tornar estas inovações acessíveis através da sua fácil utilização e permitindo a sua melhoria, potenciando assim o progresso científico.

Uma nova abordagem na modelação de núcleos em redes neuronais com Funções de Base Radial (RBF) foi proposta por Falcão *et al.* em *Flexible Kernels for RBF Networks* [14]. Esta abordagem define um algoritmo de aprendizagem para classificação, inovador na área da aprendizagem das redes neuronais RBF. Os testes efectuados mostraram que os resultados estão ao nível dos melhores nesta área, tornando como um dever óbvio para com a comunidade científica a sua disponibilização de forma aberta. Neste contexto, a motivação da implementação do algoritmo de núcleos flexíveis para redes neuronais RBF (FRBF) ganhou novos contornos, resultando num conjunto de objectivos bem definidos: (i) integração, o FRBF deveria ser integrado, ou integrável, numa plataforma facilmente acessível à comunidade científica; (ii) abertura, o código fonte deveria ser aberto para potenciar a expansão e melhoria do FRBF; (iii) documentação, imprescindível para uma fácil utilização e compreensão; e (iv) melhorias, melhorar o algoritmo original, no procedimento de cálculo das distâncias e no suporte de parâmetros de configuração. Foi com estes objectivos em mente que se iniciou o trabalho de implementação do FRBF.

O FRBF segue a tradicional abordagem de redes neuronais RBF, com duas camadas, dos algoritmos de aprendizagem para classificação. A camada escondida, que contém os núcleos, calcula a distância entre o ponto e uma classe, sendo o ponto atribuído à classe com menor distância. Este algoritmo foca-se num método de ajuste de parâmetros para uma rede de funções Gaussianas multi-variáveis com formas elípticas, conferindo um grau de flexibilidade extra à estrutura do núcleo. Esta flexibilidade é obtida através da utilização de funções de modificação aplicadas ao procedimento de cálculo da distância, que é essencial na avaliação dos núcleos. É precisamente nesta flexibilidade e na sua aproximação ao Classificador Bayeseano Óptimo (BOC), com independência dos núcleos em relação às classes, que reside a invocação deste algoritmo.

O FRBF divide-se em duas fases, aprendizagem e classificação, sendo ambas semelhantes em relação às tradicionais redes neuronais RBF. A aprendizagem faz-se em dois passos distintos. No primeiro passo: (i) o número de núcleos para cada classe é definido

através da proporção da variância do conjunto de treino associado a cada classe; (ii) o conjunto de treino é separado de acordo com cada classe e os centros dos núcleos são determinados através do algoritmo K-Means; e (iii) é efectuada uma decomposição espectral para as matrizes de covariância para cada núcleo, determinando assim a matriz de vectores próprios e os valores próprios correspondentes. No segundo passo são encontrados os valores dos parâmetros de ajuste de expansão para cada núcleo. Após a conclusão da fase de aprendizagem, obtém-se uma rede neuronal que representa um modelo de classificação para dados do mesmo domínio do conjunto de treino. A classificação é bastante simples, bastando aplicar o modelo aos pontos a classificar, obtendo-se o valor da probabilidade do ponto pertencer a uma determinada classe. As melhorias introduzidas ao algoritmo original, definidas após análise do protótipo, centram-se: (i) na parametrização, permitindo a especificação de mais parâmetros, como por exemplo o algoritmo a utilizar pelo K-Means; (ii) no teste dos valores dos parâmetros de ajuste de expansão dos núcleos, testando sempre as variações acima e abaixo; (iii) na indicação de utilização, ou não, da escala na PCA; e (iv) na possibilidade do cálculo da distância ser feito ao centróide ou à classe.

A análise à plataforma para desenvolvimento do FRBF, e das suas melhorias, resultou na escolha do R. O R é, ao mesmo tempo, uma linguagem de programação, uma plataforma de desenvolvimento e um ambiente. O R foi seleccionado por várias razões, de onde se destacam: (i) abertura e expansibilidade, permitindo a sua utilização e expansão por qualquer pessoa; (ii) repositório CRAN, que permite a distribuição de pacotes de expansão; e (iii) largamente usado para desenvolvimento de aplicações estatísticas e análise de dados, sendo mesmo o standard *de facto* na comunidade científica estatística.

Uma vez escolhida a plataforma, iniciou-se a implementação do FRBF e das suas melhorias. Um dos primeiros desafios a ultrapassar foi a inexistência de documentação para desenvolvimento. Tal facto implicou a definição de boas práticas e padrões de desenvolvimento específicos, tais como documentação e definição de variáveis. O desenvolvimento do FRBF dividiu-se em duas funções principais, `frbf` que efectua o procedimento de aprendizagem e retorna o modelo, e `predict` uma função base do R que foi redefinida para suportar o modelo gerado e que é responsável pela classificação. As primeiras versões do FRBF tinham uma velocidade de execução lenta, mas tal não foi inicialmente considerado preocupante. No entanto, alguns testes ao procedimento de aprendizagem eram demasiado morosos, passando a velocidade de execução a ser um problema crítico. Para o resolver, foi efectuada uma análise para identificar os pontos de lentidão. Esta acção revelou que os procedimentos de manipulação de objectos eram bastante lentos. Assim, aprofundou-se o conhecimento das funções e operadores do R que permitissem efectuar essa manipulação de forma mais eficiente e rápida. A aplicação desta acção correctiva resultou numa redução drástica no tempo de execução. O processo de qualidade do FRBF passou por três tipos de testes: (i) unitários, verificando as funções individual-

mente; (ii) de caixa negra, testando as funções de aprendizagem e classificação; e (iii) de precisão, aferindo a qualidade dos resultados. Considerando a complexidade do FRBF e o número de configurações possíveis, os resultados obtidos foram bastante satisfatórios, mostrando uma implementação sólida. A precisão foi alvo de atenção especial, sendo precisamente aqui onde não foi plena a satisfação com os resultados obtidos. Tal facto advém das discrepâncias obtidas entre os resultados do FRBF e do protótipo, onde comparação dos resultados beneficiou sempre este último. Uma análise cuidada a esta situação revelou que a divergência acontecia na PCA, que é efectuada de forma distinta. O próprio R possui formas distintas de obter os vectores próprios e os valores próprios, tendo essas formas sido testadas, mas nenhuma delas suplantou os resultados do protótipo.

Uma vez certificado o algoritmo, este foi empacotado e submetido ao CRAN. Este processo implicou a escrita da documentação do pacote, das funções e classes envolvidas. O pacote é distribuído sob a licença LGPL, permitindo uma utilização bastante livre do FRBF e, espera-se, potenciando a sua exploração e inovação.

O trabalho desenvolvido cumpre plenamente os objectivos inicialmente definidos. O algoritmo original foi melhorado e implementado na plataforma standard usada pela comunidade científica estatística. A sua disponibilização através de um pacote no CRAN sob uma licença de código aberto permite a sua exploração e inovação. No entanto, a implementação do FRBF não se esgota aqui, existindo espaço para trabalho futuro na redução do tempo de execução e na melhoria dos resultados de classificação.

Keywords: Funções de Base Radial, Redes Neuronais, Núcleos Flexíveis, R

Abstract

This dissertation is focused on the implementation and improvements of the *Flexible Radial Basis Function Neural Networks* algorithm. It is a clustering algorithm that describes a method for adjusting parameters for a Radial Basis Function neural network of multivariate Gaussians with ellipsoid shapes. This provides an extra degree of flexibility to the kernel structure through the usage of modifier functions applied to the distance computation procedure.

The focus of this work is the improvement and implementation of this clustering algorithm under an open source licensing on a data analysis platform. Hence, the algorithm was implemented under the R platform, the *de facto* open standard framework among statisticians, allowing the scientific community to use it and, hopefully, improve it. The implementation presented several challenges at various levels, such as inexistent development standards, the distributable package creation and the profiling and tuning process. The enhancements introduced provide a slightly different learning process and extra configuration options to the end user, resulting in more tuning possibilities to be tried and tested during the learning phase. The tests performed show a robust implementation of the algorithm and its enhancements on the R platform.

The resulting work has been made available as a R package under an open source licensing, allowing everyone to used it and improve it. This contribution to the scientific community complies with the goals defined for this work.

Keywords: Radial Basis Function, Neural Network, Flexible Kernels, R

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Whenever scientific work results in a new discovery, the scientific community, and the world in general, becomes richer. But the scientific discovery by itself is not sufficient, it must be accessible and easily usable by everyone, so that people take advantage of such innovations. Making the scientific breakthroughs accessible to everyone is therefor a major contribution to the scientific community since it provides a way to everyone use it, test it and, ultimately, improve it.

An approach for modeling kernels in Radial Basis Function (RBF) networks has been proposed in *Flexible Kernels for RBF Networks* [14] by Falcão *et al.*. This approach focus on a method for adjusting parameters for a network of multivariate Gaussians with ellipsoid shapes and provides extra degree of flexibility to the kernel structure. This flexibility is achieved through the usage of modifier functions applied to the distance computation procedure, essential for all kernel evaluations.

This new algorithm was an innovation within the neural networks learning area based on RBF neural networks. A concept proof implementation of this architecture has proved capable of solving difficult classification problems with good results in real life situations. This was a stand alone implementation with the specific goal to prove the concept and, therefor was available only to the research team members. Consequently, making this work accessible to everyone was the next logical step for this new algorithm.

In this context, an implementation of the Flexible kernels for RBF neural network (FRBF) algorithm under a widely spread scientific platform arise. A widely used platform by the scientific community should be targeted, hence the R platform has been chosen, since it is the open source *de facto* standard statistical platform. The resulting implementation was also packed and distributed under open source licensing, allowing anyone to modify it and, eventually, improve it. Some enhancements were performed on the original algorithm, some focused on the algorithm parameterization and others on the algorithm itself. The usage of the available base R functions that were, themselves, already parameterized helped on this task and, as a result, a high number of possible configurations to the end user was delivered.

This dissertation is organized as follows: Chapter 1 introduces this dissertation, Chapter 2 describes the Radial Basis Function neural networks, details the flexible kernels clustering algorithm, which is the genesis of this work, and the improvements performed to it. The implementation framework R is covered in the Chapter 3 and the implementation of the new algorithm is detailed in Chapter 4. Finally, Chapter 5 concludes this dissertation and resumes the goals achieved.

1.1 Motivation

Having obtained such good results with the FRBF tests, it was obvious that it should be made available to everyone. Hence, the main stimulus behind this work was to provide an easy way for the scientific community to use FRBF.

The proof of concept implementation was developed as a stand alone application, so it was a very specific computer program that served a single purpose and was not ready, nor meant, to be used in any other way. Hence, it did not served the purpose of distribution nor integration with frameworks, or other applications, making it a non eligible solution.

There was also a second motivation for this work, focused on the enhancement of the algorithm. It early became clear that the original algorithm could be improved and a new implementation was the perfect scenario for such task, since it provided the chance to perform the enhancements.

Hence, the need of a new FRBF implementation emerged. The motivation of this work was to (i) provide an easy to use implementation to the scientific community, (ii) integrate with, or within, a framework, (iii) improve the original algorithm and (iv) be open to receive improvements from others.

1.2 Goals

The motivation resulted in the set of specific goals. The main goal of this work was to deliver a new, open and integrated, implementation of the FRBF and a second goal was to improve the original algorithm.

These goals have been established after the identification of (i) the need of an FRBF implementation that would be integrated with, or within, a framework and (ii) the opportunity of enhance the original algorithm with some improvements. In detail, the goals for this work have been set as:

Integration. The implementation of FRBF only made sense if it could be integrated with, or within, a framework or a third party application. The selected platform was R since R is the *de facto* open standard among statisticians. R is also an integrated suite of software facilities for statistical computing, data manipulation, calculation

and graphical display. All this makes R a perfect target for this new FRBF implementation. Delivering the FRBF implementation as a R expansion package complies with this goal.

Open source. In order to allow others to expand and improve FRBF the source code had to be made open for the public. Thus, the resulting implementation was delivered under an open source licensing, allowing anyone to access the source code, explore it and even improve it.

Documentation. The implementation process followed the usual software development good practices. This means, among other things, that everything is documented. The entire source code is documented, the distributed R package is documented and the improvements are also documented. Since there are several distinct documentation levels involved here, the documentation itself comes in different formats but is, in general, easily accessible. This provides an easy way to the understanding of the FRBF implementation to anyone willing to go deeper in the subject.

Enhancements. The enhancements of the original algorithm were defined as improvements to the distance calculation procedure and the support for more configuration options. The new implementation also had to support the original algorithm specification, meaning disabling the improvements, a feature that also comes up as a configuration option. In practice, this means that the end user has more power and flexibility to configure the algorithm when searching for the best classification model for a given domain.

The goals stated above fully respond to the initial motivation identified on the precedent section. The achieve of these goals resulted in an easy way to the scientific community to use FRBF on a well known and standard platform.

1.3 Contribution

Regarding the previously stated goals in the previous section, the main contribution of this work is the deliver of an improved FRBF implementation to the scientific community.

The enhancements performed over the original algorithm are a small contribution to the RBF neural network learning algorithms. The improvements included in this implementation provide the end user more power and flexibility when parameterizing the learning task. This results in a much wide number of possibilities available when searching for the best classification model for a given problem.

The implementation of the FRBF as a R expansion package is, by itself, a contribution to the *de facto* standard statistical platform used by the scientific community. The packaging of the algorithm provides a standard way to distribute, use and document the

FRBF algorithm on this widely used platform. Finally, the usage of an open source licensing model allows anyone to explore and extend it to their own needs, opening a way for future improvements and an yet better implementation or classification algorithm.

Chapter 2

Flexible Kernels for RBF Networks

This chapter describes the Radial Basis Functions (RBF) briefly, explains the RBF neural networks, details the Flexible RBF neural networks algorithm and the correspondent enhancements introduced to the original version.

The *Flexible Kernels for RBF neural networks* algorithm, defined by Falcão *et al.* in [14], was a breakthrough in the RBF neural networks. It is a learning algorithm used for classification that provides adjustment of parameters, allowing extra flexibility to the kernel structure. The tests performed proved that this algorithm is effective with real life data.

2.1 Radial Basis Functions

A Radial Basis Function is a function whose value depends on the distance from a point x to a center point c, so that

$$\phi(\mathbf{x}, \mathbf{c}) = \phi(\|\mathbf{x} - \mathbf{c}\|) \tag{2.1}$$

The norm is to use the Euclidean distance, but other distance functions can be used.

RBF neural networks are typically used to build up function approximations. This means that a RBF neural network is used as a function that closely matches, or approximates, or describes, a target function on a specific domain. The target function itself may actually be unknown. But, in such cases, there is usually enough data from the target function domain from which one can learn, and use that knowledge to define an approximate function.

The sum of the RBF is commonly used to approximate given functions. This can be interpreted as a rather simple one layer type of artificial neural network (NN) that can be expressed by the equation

$$g(x) = \sum_{u=1}^{N} w_u \phi(||x - c_u||) \tag{2.2}$$

where the approximating function $g(x)$ is represented as a sum of N radial basis functions, each associated with a different center c_u, and weighted by an appropriate coefficient w_u. The w_u coefficient is a weight that can be estimated using any of the standard iterative methods for neural networks, like the least squares function. In this case, the Radial Basis Functions are the activation functions of the neural network. RBF are covered in detail by Hastie *et al.* in [23] and by Buhmann in [6].

2.2 Radial Basis Functions Neural Networks

RBF neural network, as introduced in the prior section, is a type of artificial neural network constructed from a function distance. The function distance is obtained from the known domain data, called training data, which means the RBF neural network is a learning method that will try to find patterns in the training data and model it as a network. In particular, the distance function is used to determine the weight of each known data point, the training example, and it is called Kernel function. The work of Yee *et al.* in [59] and Hastie *et al.* in [23] cover RBF neural networks in detail.

Learning with RBF neural networks is therefor an approach to function approximation, which is closely related to distance weighted regression and to artificial neural networks. The term *regression* is widely used by the statistical learning community to refer the problem of approximating real valued functions, while *weighted distance* refers to the contribution that each training example has, by calculating the weight of its distance to a center point. This subject is widely studied by the scientific community, some examples are [41, 5, 22, 4, 36, 24, 35, 23, 59, 58]. In particular, Park *et al.* in [40] studies universal approximation using RBF neural networks.

As specified in detail by Mitchell in [36], in the RBF neural network approach the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \tag{2.3}$$

where k is a parameter provided by the user that specifies the number of kernel functions to be included, x is the point being classified, each x_u is an instance from X, the training data, and $K_u(d(x_u, x))$ is the kernel function, that depends on a distance function.

It is easy to understand that the distance function is essential for all kernel evaluations. As previously stated, the kernel function is actually the distance function that is used to determine the weight of each training example. In Equation 2.3 above, it is defined so that it decreases as the distance $d(x_u, x)$ increases.

Even though $\hat{f}(x)$ is a generic approximation to $f(x)$, the function that correctly classifies each instance, the contribution from each of the kernel terms is located in a region near the x_u point. It is common to choose each kernel function to be a Gaussian function

centered at the point x_u with some variance σ_u^2, so that

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)} \tag{2.4}$$

This equation is the common Gaussian kernel function for RBF neural networks, but other kernel functions can be used. The kernel functions have been widely studied and is easy to find literature about it, for instance, Hastie *et al.* in [23] describes the Gaussian RBF and Powell in [42] details RBF approximation to polynomial functions.

2.2.1 Neural Architecture

The function in Equation 2.3 can be viewed as describing a two layer network where the first layer computes the values of the various $K_u(d(x_u, x))$, and the second layer computes a linear combination of the unit values calculated in the first layer. In the basic form, all inputs are connected to each hidden unit. Each hidden unit produces an activation determined by a Gaussian function, or any other function used, centered at some instance x_u. Therefor, its activation will be close to zero unless the input x is near x_u. The output unit produces a linear combination of the hidden unit activations. An example of a RBF neural network is illustrated in Figure 2.1.

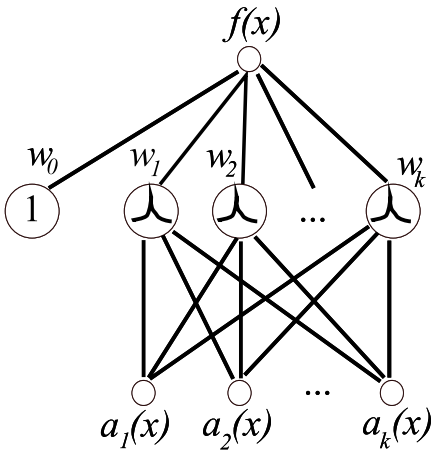


Figure 2.1: A RBF neural network from Mitchell [36].

In the neural network terminology, the variables of Equation 2.3 are called differently, though they mean the same. In particular, k is the number of neurons in the hidden layer, x_u is the center vector for neuron u, and w_u are are weights of the linear output neuron. An example of a RBF neural network with the neural network terminology is illustrated in Figure 2.2. The work of Haykin in [24] discusses neural networks in detail while Hartman *et al.* in [22] focus on neural networks with Gaussian hidden units as universal approximations, and more recently, Zainuddin *et al.* in [60] discusses function approximation using artificial neural networks.

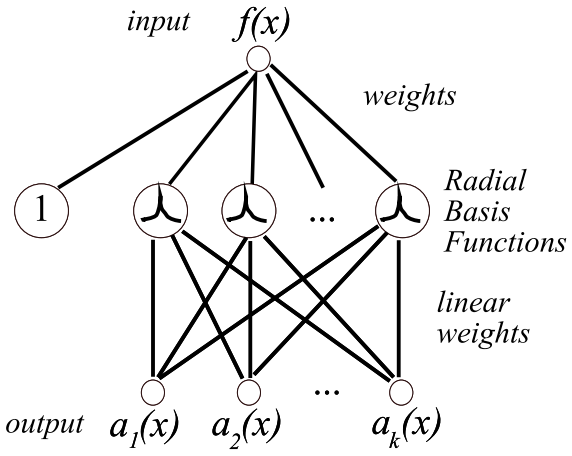


Figure 2.2: A RBF neural network with NN terminology adapted from Mitchell [36].

2.2.2 Radial Basis Function Network Training

RBF neural networks are built eagerly from local approximations centered around the training examples, or around clusters of training examples, since all that is known is the set of the training data points. Hence, the training data set is used to build the RBF neural network, which is achieved over two consecutive stages. The first stage selects the centers and set the deviations of the neural network hidden units. The second stage optimizes the linear output layer of the neural network.

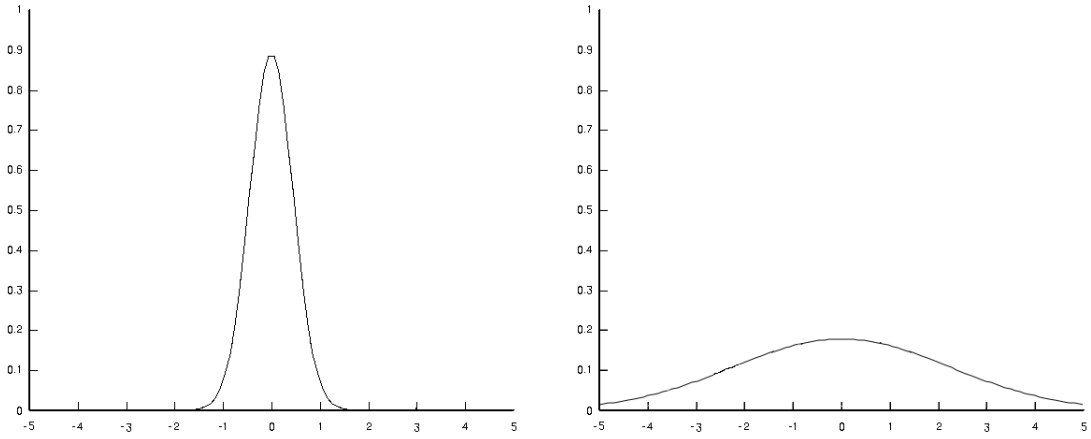
First Stage

As previously stated, on the first stage the centers must be selected. The center selection should be assigned to reflect the natural data clustering. The selection can be done uniformly or non-uniformly. The non-uniform selection is especially suited if the training data points themselves are found to be distributed non-uniformly over the testing data. The most common methods for center selection are:

Sampling: use randomly chosen training points. Since they are randomly selected, they will represent the distribution of the training data in a statistical sense. However, if the number of training data points is not large enough, it may actually be a poor representation of the entire data domain.

K-Means: use the K-Means algorithm, explained by MacQueen and Bishop in [34, 4], to select an optimal set of points that are placed at the centroids of clusters of training data. Given a k number of clusters, it adjusts the positions of the centers so that (i) each training point belongs to the nearest cluster center, and (ii) each cluster center is the centroid of the training points that belong to it. The EM algorithm, explained in detail by Dempster *et al.* in [13], can also be used for this task.

Once the centers are assigned, it is time to set the deviations. The size of the deviation determines how spiky the Gaussian functions are. If the Gaussians are too spiky, the network will not interpolate between known points, and thus loses the ability to generalize. On the other end, if the Gaussians are very broad the network loses fine detail. This is actually a common manifestation of the fitting dilemma, over-fitting is as bad as under-fitting. For an example of such Gaussian shapes, see Figure 2.3.



Spiky Gaussian at the left and broad Gaussian on the right.

Figure 2.3: An example of a spiky and a broad Gaussian, adapted from Wikipedia [57].

To obtain a good result, the deviations should typically be chosen so that Gaussians overlap with a few nearby centers. The most common methods used for such task are:

Explicit: the deviation is defined by a specific value, for instance a constant.

Isotropic: the deviation is the same for all units and is selected heuristically to reflect the number of centers and the volume of space they occupy.

K-Nearest Neighbor: where each unit deviation value is individually set to the mean distance to its K nearest neighbors. Hence, deviations are smaller in tightly packed areas of space, preserving detail, and higher in sparse areas of space. The work of Cover *et al.* in [11] and Haykin in [24] give a detailed insight.

Second Stage

Once the centers and deviations have been set, the second stage takes place. In this stage, the output layer can be optimized using a standard linear optimization technique, the Singular Value Decomposition algorithm (SVG) as described by Haykin in [24].

The singular value decomposition is an important way of factoring matrices into a series of linear approximations that expose the underlying structure of the matrix. This allows faster computation since patterns are used instead of the entire data itself.

The training of the RBF neural network is concluded when this stage finishes. The result is a trained neural network that defines a model for the domain of the training data.

2.2.3 Classification with Radial Basis Function Network

Classification using the learned RBF neural network is quite simple. Once the RBF neural network is defined through the learning procedure, as described in the previous section, it holds a model that can be applied to new, unseen, data of the same domain as the training data. Hwang *et al.* in [26] describe an efficient method to construct a RBF neural network classifier.

When the model is applied, the artificial neural network will be able to classify, hopefully correctly, the new data by calculating the distance of each new data point to each of the centers of the model. This is performed through the activation of the hidden units, as in any other artificial neural network. When a new data point x is being classified, it will activate the hidden unit x_u, where x_u is the nearest center to the point x. Following Figure 2.4, the classification of x results in (i) 20% of probability to belong to the first cluster of class A, (ii) 10% of probability to belong to the second cluster of class A, (iii) 40% of probability to belong to class B, and (iv) 30% of probability to belong to class C.

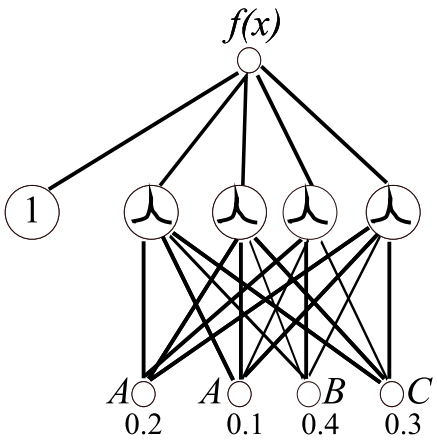


Figure 2.4: A RBF neural network classification example adapted from Mitchell [36].

To summarize, when the obtained RBF neural network model is applied to a data point x it will return the x_u that is the nearest center to x, and classification happens.

2.3 Flexible Kernels for RBF Neural Networks

The original *Flexible Kernels for RBF neural networks* algorithm, developed by Falcão *et al.* in [14], was a breakthrough in the RBF neural networks area. This learning algorithm is used for classification and distinguishes itself from other RBF neural network algorithms by introducing extra flexibility to the kernel and by its approximation to the Bayes

Optimal Classifier (BOC) with independent kernel regarding the classes. The work is the genesis of this dissertation. It follows the earlier works of Albrecht *et al.* and Bishop *et al.* in [1, 4] and will be explained in the next sections.

As described by Hastie *et al.* in [23], kernel methods achieve flexibility by fitting simple models in a region local to the target point x_u. Localization is achieved via a weighting kernel K_u, and individual observations receive weights $K_u(d(x_u, x))$.

Different models of RBF neural networks can be found in the literature and several methods for fitting parameters on this type of artificial neural network have already been studied. Thus, introducing flexibility to the kernel function *per se* is not a new idea, for instance Bishop in [3, 4] and Jankowski in [29] have done it previously.

2.3.1 Flexible Kernels

This approach truly innovates by using modifier functions applied to the distance computation procedure, which is essential for all kernel evaluations as seen previously in Section 2.2. But this approach also distinguishes from others because it will sum only the kernels from the same class, which means it is approximated to the Bayes Optimal Classifier, described in [36] by Michell, since it preserves class independence. This is achieved by using the training data information for constructing separate sets of kernels for each class present in the data domain. Using this principle, the classification procedure is straightforward.

Continuing to follow closely the work of Falcão *et al.* in [14], and stated in a more formal form, a class C_i, belonging to the class set C, is attributed to a given pattern x according to the sum of all the kernels that have been adjusted for each class

$$\arg\max_{C_i \in C} \sum_j w_i^j K_{ij}(x) \tag{2.5}$$

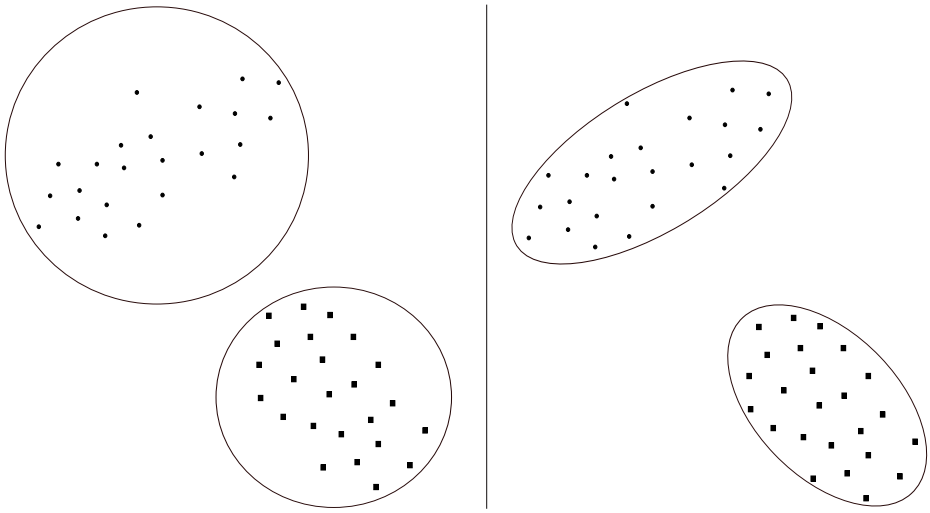
where K_{ij} is a generic kernel function and the w_i^j parameter leverages the importance of a given kernel. Note that this equation is not equivalent to Equation 2.2. The equations have only one slight difference in the sum, but that difference results in two very distinct approaches. While the traditional RBF neural network sums all the RBF, in this equation the sum is selectively applied per class. This selective application of the sum per class isolates the kernels and allows class independence.

Using the common Gaussian model as the choice for kernel functions, as presented in Section 2.2, $K_{ij}(x)$ can be rewritten as

$$K_{ij}(x) = exp(-(x - c_{ij})\Sigma^{-1}(x - c_{ij})^t/2) \tag{2.6}$$

where c_{ij} corresponds to the kernel location parameter and Σ to a covariance matrix of a data set. Using the inverse of the covariance matrix allows the captured of the correlations between different variables, which provides a *n*-dimensional ellipsoid shape to

each kernel against the common circular shape. As exemplified in Figure 2.5, the Gaussian model is a better representation of the clusters than the common circular shape. Note that Equation 2.6 is not equivalent to Equation 2.4, since the distance function used is no longer the usual Euclidean distance.



On the left the common circular shape and on the right the ellipsoid shape.

Figure 2.5: Example of shapes.

The inverse of the covariance matrix was selected because it is generally more versatile than using the simple distance to the kernel centroid that assumes strict variable independence. But his approach has some drawbacks. Namely, if the covariance matrix is singular or very ill conditioned, the use of the inverse can produce meaningless results or strong numerical instability. Removing the highly correlated components may seem a good ideal. Unfortunately, that is infeasible since these may vary among kernels and among classes, thus not allowing an unique definition of the set that builds the highly correlated components for removal.

To better understand this problem, a spectral decomposition can be applied to the covariance matrix, producing the following eigensystem

$$\Sigma = P \wedge P^t \tag{2.7}$$

in which P is the matrix composed of the eigenvectors and $\wedge$ is the respective eigenvalues in a diagonal matrix format. Spectral decomposition, or eigenvalue decomposition, has been widely studied as [19, 4, 35, 23] are examples of.

The use of a more generic strategy is suggested in order to effectively use the information conveyed by the eigenstructure produced for each kernel. This is achieved by not limiting the model to the inverse function but instead, consider a generic matrix function $M(\wedge)$ coupled with a scalar multiplier parameter s used to weight the distance modified

by the operator $M(\cdot)$, the diagonal matrix function. $K_{ij}(x)$ can now be rewritten as

$$K_{ij}(x) = exp(-(x - c_{ij})PM(\wedge)P^t(x - c_{ij})^t s) \tag{2.8}$$

This equation is thus more generic than the general Gaussian kernel model in Equation 2.6. In fact, the new s parameter value is a generalization of the $1/2$ constant. Larger values of s correspond to more spiky Gaussians, and smaller ones to broader shapes that decrease slowly towards 0. These Gaussians shapes can be seen in Figure 2.3.

It was found that using the parameter inside the Gaussian kernel, instead of considering it as a common weight multiplier, has a strong positive effect in the discrimination capabilities of the model.

A variety of models have been tested by Falcão in [14] *et al.* for the diagonal matrix function $M(\cdot)$, listed on Table 2.1. Model (0) stands for the simplest RBF kernel, which does not account for the correlation between variables, since it provides circular shapes for kernels instead of ellipsoids as all the remaining models. The traditional Gaussian kernel of Equation 2.6 corresponds to function model (3), the Mahalanobis functions detailed by Mardia *et al.* in [35], and, like in model (6), a constant ε, with a small value like 0.01, is added to the expression to ensure that numerical problems do not arise.

(0)	1	(1)	$(1 - \lambda)$	(2)	$(1 - \lambda)^2$
(3)	$1/(\lambda + \varepsilon)$	(4)	$exp(1 - \lambda)$	(5)	$exp(1 - \lambda)^2$
(6)	$(1 - log(\lambda + \varepsilon))$	(7)	$(1 - \lambda)/(1 + \lambda)$	(8)	$((1 - \lambda)/(1 + \lambda))^2$

Table 2.1: Distance weighting function models from Falcão *et al.* [14].

The learning procedure for this, described in the following two sections, is the same for all the models. It is performed over two stages and is quite similar to the procedure explained in Section 2.2.2.

Stage One of Flexible Kernels Learning Procedure

On this initial stage, the classifier0 uses an unsupervised method to construct the network topology where the kernel positions and the global shapes are set. Most of the network parameters required are defined without any customization from the user. Actually, only two parameters are required from the user (i) the total number of kernels in the model and, (ii) the distance modifying function type.

The algorithm starts by defining the number of kernels in each class. This is done through the proportion of the total variance in the training data set associated with that class. The variance is used to adjust the number of kernels per class regarding how the class is spread on the domain data. Then, the training data is separated according to each class, and the kernel centers are determined through the usual K-Means clustering algorithm. After the clustering procedure, the kernel location parameters correspond to the centroids of the resulting clusters, and the w_{ij} parameter of Equation 2.5 is set to the

number of patterns that are included in each cluster. Finally, the spectral decomposition is performed for the covariance matrices of each kernel to determine the eigenvector matrix and the corresponding eigenvalues. An overview of this first stage can be seen in Algorithm 2.1.

Algorithm 2.1 Stage One of Flexible Kernels Learning Procedure

Require: Number of kernels, instance modifying function type, training data.

1: $var \leftarrow variance(classes, total\_kernels, training\_data)$
2: **for all** $class$ in $classes$ **do**
3: $num\_kernels \leftarrow adjustKernels(total\_kernels, var, class, training\_data)$
4: $kernels[class] \leftarrow kmeans(training\_data, num\_kernels)$
5: **for all** $kernel$ in $kernels[class]$ **do**
6: $eigen[kernel] \leftarrow pca(kernel)$ {Spectral decomposition for the covariance matrices to find the eigenvector matrix and the corresponding eigenvalues.}
7: **end for**
8: **end for**

Stage Two of Flexible Kernels Learning Procedure

After the conclusion of stage one, the classes of each pattern in the training data are used to learn the adjustment of appropriate spread parameters for all kernels, i.e. the widths s of each kernel are determined.

This part of the algorithm starts by assigning a common low value for all clusters of all classes. This value is increased iteratively by a fixed amount, checking the classification error rate at every iteration. Typically, as this parameter increases, the error decreases down to a local minimum and is then used as an initial estimate. At this point, a simple Hill-Climbing greedy algorithm starts to individually adjust estimates for each kernel. The Hill-Climbing algorithm can be easily found in the literature, as [36, 45, 20] are examples of.

The parameter adjustment is also performed iteratively per cluster by incrementing and reducing the spread parameter value and testing its accuracy. First, increment the value and check the classification accuracy. If the accuracy result is better, then keep that value as the current best spread parameter for the current cluster. Otherwise, reduce the value and check the classification accuracy. Again, if the accuracy result is better, then keep that value as the current best spread parameter for the current cluster. This method uses two parameters (i) d, the parameter that will make vary s, and (ii) α, a constant that will make vary d. The d parameter starts with a, somewhat, large value, implying larger modifications to the s parameter for each kernel. But, as the algorithm proceeds, the d parameter decreases, and the change in s approaches zero. This procedure is performed until a maximum number of thirty iterations is achieved or there are no changes in the classification accuracy for three consecutive iterations. An overview of this second stage

can be seen in Algorithm 2.2.

Algorithm 2.2 Stage Two of Flexible Kernels Learning Procedure

```
1: s[] ← random() {Attribute a random initial value to all clusters.}
2: d ← 0.23
3: no_changes ← 0
4: number_iterations ← 30
5: while number_iteration > 0 do
6:    number_iterations ← number_iterations − 1
7:    for all cluster in random(clusters) do
8:       s[cluster]′ ← s[cluster] × (1 + d) {Increase the testing spread.}
9:       if betterAccuracy(s, s′) = s′ then
10:         s[cluster] ← s′ {Keep this value for the current cluster.}
11:         no_changes ← 0
12:      else
13:         s[cluster]′ ← s[cluster] × (1 − d) {Decrease the testing spread.}
14:         if betterAccuracy(s, s′) = s′ then
15:            s[cluster] ← s′ {Keep this value for the current cluster.}
16:            no_changes ← 0
17:         else
18:            no_changes ← no_changes + 1
19:            if no_changes = 3 then
20:               number_iterations ← 0 {No changes for 3 iterations, so stop.}
21:            end if
22:         end if
23:      end if
24:      d ← α × d {Update d multiplying it by a constant.}
25:   end for
26: end while
```

Finding the s value by testing the classification accuracy constitutes the final classifier, thus resulting on a RBF neural network ready for classification.

Classification using the Flexible Kernels

After the learning procedure has taken place, the classification is straightforward. Since the kernels are isolated, the classes are independent, which means that the classification of any given point is performed by measuring the distance between the point and the sum of the centroid distances of each class. The point will belong to the nearest class, *i.e.* it will be classified as a point of the class that is nearest to it.

Following the example of Figure 2.6, the FRBF will return (i) 0.3 when tested for class A, (ii) 0.4 when tested for class B, and (iii) 0.3 when tested for class C. Thus, x belongs to B since it is the class to which x has the higher probability to belong to.

Note that this is different from the traditional RBF neural networks, where the sum is applied to all the kernels. Simply stated, (i) in the traditional approach, an instance x is

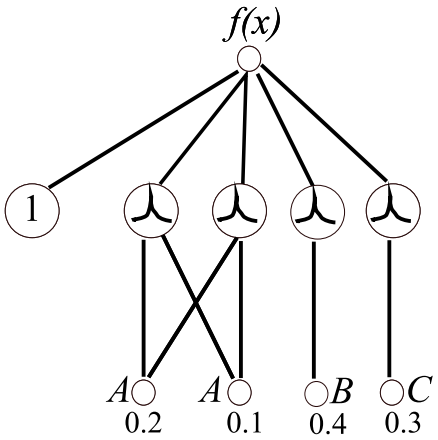


Figure 2.6: A FRBF classification example adapted from Mitchell [36].

applied to the RBF neural network and it returns the probability of x to be part of each class cluster, as seen in Figure 2.4; while (ii) in this approach, an instance x is applied to the sum of the kernels of a class, and returns the probability of x to be part of that class. This difference is easily viewed by comparing Figures 2.4 and 2.6.

2.4 Proof of Concept Prototype

Following the procedures described for the FRBF in the previous section, a prototype has been implemented in order to prove the algorithm concept. It was codenamed Remora.

The prototype was developed using the C programming language and the Microsoft Visual C++[1] development environment. To perform the spectral decomposition, it made use of the Principal Components Analysis (PCA), detailed by Murtagh *et al.* [38] and Mardia *et al.* in [35]. An implementation of the PCA standard ANSI C library, which has been developed by Murtagh, was used for this.

The implementation followed the algorithm strictly, which means that the parameters described in the algorithm are constants, except for the identified user parameters, even when a parameter could be parameterized by the user, such as the d parameter listed in Algorithm 2.2. During the development of the prototype, a bug in the PCA library was found and promptly fixed.

The prototype resulted in a stand alone Microsoft Windows[2] application that executed from the command line. The application works in three distinct modes and the command line parameters depend on the selected run mode. The application supports the following command line parameters:

Function Type: the index of the function type to use, from the Table 2.1.

[1]Microsoft and Visual C++ are registered trademarks of Microsoft Corporation.
[2]Microsoft Windows is a registered trademark of Microsoft Corporation.

Model File: the resulting learned model output file, only for run modes **C** and **V**.

Run Mode: the run mode: **A** (Mode Adjust), to adjust the RBF neural network model; **C** (Mode Classify), to perform the classification, it outputs the data point index and the class prediction; and **V** (Mode Validate), to perform the validation, it performs the classification, outputs the data point index and the class prediction, and displays the accuracy and the confusion matrix.

Input File: the input file that holds the testing data, when in run mode **A**, or the data to classify, on the other run modes, **C** and **V**.

Number of Kernels: the total number of kernels in the model, only for run mode **A**.

Clustering Function: the clustering function to use, only for run mode **A**: **1** (Cluster K-Means), to use the K-Means; **2** (Cluster EM), to use the EM algorithm; and **3** (Cluster AHCL), to use the complete linkage.

Output File: the resulting classification output file name, only for **C** and **V** run modes.

Verbosity: the type of verbosity during execution: **1**, verbose, or **0**, no verbose.

In order to be used, the application has to be called twice. Once in the **A** run mode, for training, and a second execution in the run mode **C**, for classification, or **V**, for classification and accuracy validation. The execution in the training mode **A** results in the writing of a model file, named `output.rem`, that holds the artificial neural network model. This file will later be used when the application is executed for the classification task, in the **C** and **V** run modes. There are two main difference between the **C** and **V** run modes. They both perform classification on the input data, read from the CSV input file, and they both output the result into a CSV file, containing the data points index and the class to which they belong to. But in the **C** run mode the input file cannot contain the class column while in the **V** run mode the input file must contain the class column, plus it displays the accuracy and the confusion matrix. Figure 2.7 exemplifies the prototype application usage.

The prototype was tested with several of the most common data sets from the StatLog [47] repository with very good results, as stated in Table 2.2.

Dataset	Kernel Function	Train Accuracy (%)	Test Accuracy (%)
dna	(1)	98.30	95.62
letter	(3)	98.95	95.95
satimage	(9)	94.88	90.35
shuttle	(7)	99.90	99.85

Table 2.2: StatLog results using the prototype, adapted from Falcão *et al.* [14].

Training:
```
remora.exe 1 A "d:/thesis/dataset/train.csv" 6 1 1
```

Classification:
```
remora.exe 1 C output.rem "d:/thesis/dataset/classify-nc.csv"
"d:/mestrado/dataset/result.txt" 1
```

Validation:
```
remora.exe 1 V output.rem "d:/thesis/dataset/classify-wc.csv"
"d:/mestrado/dataset/result.txt" 1
```

Figure 2.7: Prototype usage example.

Despite the results, unfortunately the prototype was unable for redistribution since it was a very specific stand alone application. It had not been developed to be included, or used by, other applications or frameworks, therefor it had one unique usage and a single specific purpose.

2.5 Improvements

As stated before, the enhancement of the original algorithm, as described in Section 2.3, is one of the focus of this dissertation. Several improvements have been introduced, mostly related to the algorithm parameterization.

Despite the changes described here, the original algorithm was preserved unchanged. The only exception is the testing of the spread values, where the increase and the decrease of values are always both tested. Apart from this exception, the resulting implementation, as described in the following Chapter 4, allows the use of the original version. In fact, the default configuration reflects the original version of the algorithm.

The enhancements that have been introduced are described in the following sections.

Parameterization

One obvious and simple improvement was to allow the user to customize the parameters that were declared static, such as the d parameter listed in Algorithm 2.2. Hence, all parameters that could be defined by the user moved from constants values into user defined values. Namely: (i) d, the initial value of d; (ii) ε, required by some models; (iii) $niter$, the maximum number of iterations to perform when finding s; and (iv) $niter\_changes$, the number of iterations without changes that can occur when finding s. Note that the α parameter in Algorithm 2.2 from the Stage Two of learning procedure in Section 2.3.1, is not parameterized. The reason for such design option came from the prototype results that indicated that the α parameter could be automatically inferred with very good results, thus removing the need for the user to tune this parameter. Hence, the variation of d per

iteration occurs from the following formula $d = d\_start + iter niter * (d\_end - d\_start)$ where (i) d_start is the initial d value; (ii) $iter$ is the current iteration; (iii) $niter$ is, as seen before, the maximum number of iterations to perform when finding s; and (iv) d_end is the lower threshold for d, meaning d will never be lower than 0.01.

Testing Spread Values

As described in the Stage Two of the learning procedure algorithm, in Algorithm 2.2 a s' greater value is tested, but a s' lower value is only tested if the greater value did not yield a better accuracy value than the one found up to that moment. This has been changed to always test the increase and the decrease of the spread parameter. This means that a lower value will always be tested even if the greater value resulted in a better accuracy than the one found up to that moment.

This is the only modification that cannot be parameterized to allow the execution of the algorithm with the original behavior. This means that the algorithm will always execute using this improvement.

PCA Scale Variance

The Principal Components Analysis (PCA), used for the spectral decomposition, can be scaled to have unit variance before the analysis takes place. The scaling will be performed by dividing the centered columns by their root-mean-square, as Becker *et al.* states in [2]. In practice, this means that the spectral decomposition will be performed only after all values have been scaled.

Evaluate Each Cluster Individually

In the original algorithm, the classification of a given point is calculated using the sum of the centroid distances per class, as previously described. But it can also be calculated using just the individual centroid distance.

In the original version, the distance of all the centroids is summed per each class, and a point is classified against the distance of the class. With this enhancement, a point can be classified by calculating the distance against an individual class centroid.

Chapter 3

R

This chapter describes R and why it was selected as the target platform for the new implementation of the *Flexible Kernels for RBF neural networks* algorithm. Other solutions have been considered, such as Java[1] and .Net[2] frameworks, but since they have not been selected as the target development platform, they are not mentioned here. This chapter also covers the official repository, that holds the packages that can be used to expand R, and the mechanisms provided for development.

3.1 What is R?

R is a *programming language*, a *development framework* and a *software environment* for statistical computing, modeling and data visualization. It was created by Ross Ihaka and Robert Gentleman [27] at the University of Auckland, in New Zealand, and it implements the S programming language, developed at Bell Laboratories[3] by Rick Becker, John Chambers and Allan Wilks.

R is currently a GNU[4] project developed by the R Development Core Team and can be regarded as an open source implementation of the S language, providing an easy and accessible route to research in statistics. This makes R very similar to S, it even supports much code from S allowing it to be executed unaltered, and therefor almost all literature targets both systems.

R is a language and cross platform environment that uses a command line interface. Pre-compiled binary versions are provided for various operating systems and there are graphical user interfaces available on some of those systems. It is highly extensible, provides graphical techniques and a wide variety of statistical computing, like linear and nonlinear modeling, classical statistical tests, time-series analysis, neural networks, classification and clustering. Some of R strengths include:

[1] Java is a registered trademark of Sun Microsystems.
[2] .Net is a registered trademark of Microsoft Corporation.
[3] Formerly AT&T, now Lucent Technologies.
[4] GNU is a registered trademark of the Free Software Foundation.

- an effective data handling and storage facility;
- a suite of operators for calculations on arrays, in particular matrices;
- a large, coherent, integrated collection of intermediate tools for data analysis;
- support for much S code, allowing it to be executed unaltered;
- ease to produce well-designed publication-quality plots, including mathematical symbols and formulas where needed;
- graphical facilities for data analysis and display either on-screen or on hard copy;
- an extension mechanism that allows contributions;
- a well-developed, simple and effective programming language which includes special operators, conditionals, loops, user-defined recursive functions and input and output facilities.

For all the stated reasons, R is widely used for statistical software development and data analysis, making it the *de facto* standard among statisticians.

3.2 R Language

Due to the similarity between R and S, the R language and its natural evolution follows S. There is a set of books that characterize the language, namely:

1. *The New S Language*, which is the basic reference for R and was written by Becker *et al.* [2],
2. *Statistical Models in S*, that details the features included in the early nineties and was written by Chambers [7], and
3. *Programming with Data*, that describes the formal methods and classes of the methods package and was also written by Chambers [8].

Despite these S references, there is a specific *R Language Definition* [52] that defines the R language. There is also a frequently asked questions (FAQ) [25] that covers the basics and is a good starting point for all new R users.

The language syntax has a superficial similarity with the C programming language, but the semantics are of the functional programming language variety with stronger affinities with the Lisp and APL programming languages. In particular, it allows ”computing on the language”, which makes possible to write functions that take expressions as an input, a feature that is common and often useful when applied to statistical modeling and graphics.

3.3 R Workspace

The R workspace is the working environment that includes the command prompt and the user defined objects such as vectors, matrices, data frames, lists, functions and variables. This means that the workspace is composed by a working area, that includes all objects currently in memory, and a command prompt where the user can give commands such as (i) a call to any defined function; (ii) a variable manipulation, like an assignment; or (iii) a specific R console command, such as terminate the session or clear the workspace. The management of objects in the workspace memory is dynamic. This means that, for instance, a library, function or variable, can be loaded into, or removed from, the workspace at any time.

During a R session, it is possible to save the state of any object from the workspace into an external file, and load it again from the file into the workspace. At the end of a R session, the user can save an image of the current state of the workspace, that includes all the objects, that will be automatically reloaded the next time R is started. It is also possible to save the current workspace state and loaded it at any time. This feature is extremely useful to everyone that needs to keep a restore point or wants to keep a specific state of the current work for sharing or later usage.

R comes with both a command line text console and a graphical console that provides user friendly interaction such as a set of common R console commands and easy access to R packages. There are also other third party R environments that potentiate the usage of R workspace, for instance by combining the console with a script editor. But this is not the only way to interact with R. It is possible to execute an R script by calling the R from the system command prompt and the script as a parameter. That will make R to execute the specified script and, when finished, it will return to the system command prompt.

3.4 Comprehensive R Archive Network

R comes with a set of pre-installed packages that form its basis. In order to expand these basic capabilities, other packages can be obtained from a centralized repository, the Comprehensive R Archive Network (CRAN). CRAN is a family of Internet sites that hold a very wide range of modern statistical packages.

A package is a library, usually about a specific topic, area or functionality, that contains a set of functions, data, and the correspondent documentation. The data present in the packages is optional and is usually used to support, test, or illustrate the functions of the package.

Each package expands R by providing new functions to it. The packages are usually available to the scientific community through the R centralized repository CRAN. Obtaining and using a package is performed by downloading the package from CRAN and then

loading it into the workspace. The graphical console assists the user in this task, making it quite easy and straightforward.

As stated before, one of this dissertation goals is to provide the FRBF, described in the following Chapter 4, as a package in the CRAN repository as a contribution to the scientific community.

3.5 R Development

Simply stated, R allows development through the definition of user functions and class objects. For that, R provides the usual basic programming language mechanisms [52], like control structure, class definition, basic data types, operators, etc.. R is interpreted, which means slower executions when compared with similar compiled code. Nevertheless, using R is actually quite efficient, even when it comes to working with complex operations and large data sets. When packed, a development may be distributed and shared with others.

3.5.1 Objects

R supports two object systems, known as the S3 and the S4 objects.

Simply stated, S3 objects, classes and methods have been available in R since its inception and are very informal. For instance, it is not required to define any data type for its slots, commonly known in Object Oriented Paradigm (OOP) as properties or members.

The S4 objects are the new generation that tries to eliminate the weak S3 OOP support. It requires more attention from the developer. In particular, it forces the explicit declaration of slots with a data type and the `new()` function must be used to create a S4 object.

3.5.2 Function Overloading

R supports function overloading based on data types. More precisely, a function behavior can be defined based on the data type that it receives as an argument. For instance, the `print` function, that displays a variable value on the console, changes its behavior depending on the variable data type. The printing of a matrix is displayed differently than an integer or an array.

The overloading mechanism is quite simple. R interprets the function name concatenated with the data type by a dot, preserving the original function signature. Figure 3.1 shows how this mechanism can be declared.

This mechanism is very useful when custom implemented classes need a pretty way to display its values to the user.

```
<function>.<data type> <- function(argument1, ...  )  {
 [...]
}
```

Figure 3.1: Function overloading definition.

3.5.3 Application Programming Interface

R has an Application Programming Interface (API) that allows it to be included as, or to include an, external library. This feature makes R a perfect companion for other applications, since they can make use of R and its functions and packages. This feature also makes R a preferred target platform, since it supports communication with other external functions allowing it to be expanded without recurring to the package mechanism.

In particular, R can interface with the C and Fortran [52] programming languages. It can be called from both C and Fortran, and it can execute compiled C and Fortran 77 code, or any other language which can generate C interfaces, for example C++.

In particular, C implementations are common for performance reasons. Some packages actually have its functions implemented in the C programming language. This allows a boost in performance when compared with the same implementation in R, that would have to be interpreted.

3.5.4 Debug

Since R runs on a command line interface, debugging in R is performed via the call to debug functions. This is true even when using the graphical console, since it does not provide any special or extra interface for this task.

The usual debug mechanisms like break points, call stack trace, variable querying, and manipulation of data in memory are available.

Since the usage of the debug functionality is performed via written commands, it turns out to be verbose and script intrusion. For instance, to mark a break point on a function, the `browser()` function call must be included precisely on the line where the execution should be paused and the debug command line should be prompted. When the `browser()` function call is executed, the control returns to the R console with a special debug prompt that allows the user to interact with the program in its current state. The user has access to all information in scope, allowing to query and modify the data as required.

3.5.5 Why R?

As seen along the sections of the current chapter, R provides many features that make it a great choice for the development of the new implementation of the *Flexible Kernels for RBF neural networks* algorithm.

R was chosen for a number of reasons, but in particular it was selected because (i) the R platform is quite open and extensible, allowing anyone to use it and extend it in several ways, (ii) the CRAN repository is a great way to distribute a package to the scientific community, and finally, (iii) R itself is widely used for statistical software development and data analysis. In fact, it is the *de facto* standard among statisticians.

Hence, selecting R for this task helped achieving some of the goals of this dissertation.

Chapter 4

FRBF Implementation

This chapter describes the new implementation of the *Flexible Kernels for RBF neural networks* algorithm, including its enhancements, as described in the previous Sections 2.3 and 2.5. All aspects of its implementation in R are covered here, namely, the development process, the packaging, the documentation, the problems found, the challenges overcame and the resulting work. The resulting work is known as FRBF, the acronym of Flexible RBF that is also the name of the R function that implements the algorithm and the distributable R package.

4.1 Development Environment

The implementation of FRBF was performed on several distinct environments under the Windows XP[1] [10] and Kubuntu[2] [33] operating systems. This was necessary because some development tasks were easier, and faster, to accomplish under certain specific environments.

Regardless of the number of environments used, there was a common version control system that served them all. A Subversion (SVN) [9] version control server running on Debian[3] [28] has been used for this task. On the client side, the correspondent SVN client command line tools and the operating systems specific GUI tools, TortoiseSVN [55] in Windows XP and KSVN [39] in Kubuntu, have also been used.

Despite the number of operating systems and different software involved, there was only one laptop used in the implementation of FRBF. The laptop hardware has a single core Pentium[4] M processor at 2GHz, 2GB of RAM and 100GB of hard disk. This particular limitation of using only one laptop, forced to boot between systems whenever it was necessary to perform tasks that specifically required a certain environment. Each of the software tools used had specific goals and some were only really useful when combined

[1] Microsoft XP is a registered trademark of Microsoft Corporation.
[2] Kubuntu is a Linux distribution and a registered trademark of Canonical Ltd.
[3] Debian is a registered trademark of Software in the Public Interest, Inc.
[4] Pentium is a registered trademark of Intel Corporation.

with others. For this reason, some automatic combinations of software tools have been implemented through scripting. The usage of two distinct operating systems and such a variety of software increase the implementation complexity.

4.1.1 R Development Environment

The implementation of the algorithm was mainly performed in Windows XP using the Tinn-R [15] editor. But the algorithm has also been developed in Kubuntu using the JGR [43] as both the text editor and the R console.

Regardless of the operating system, the official R tools have been widely used for script execution, algorithm debugging, FRBF function testing, packaging, documentation and build. There was one exception related with the packaging that ended being totally performed under the Linux environment for productivity reasons.

4.1.2 Documentation Development Environment

The documentation of the FRBF package has been entirely performed on Linux. This was because Kubuntu had almost all the necessary tools already available on the system, and the missing ones were extremely easy to obtain and use with little or none configuration. This task could also have been accomplished using Windows, as describe by Rossi in [44], but there was an enormous overhead, with little gain compared to the adopted solution, related with tool gathering, installation, configuration and a certain lack of documentation and support on it.

Hence, the development of the packaging documentation was performed on Kubuntu, using LaTeX, as the official documentation [53] refers, Kate [49] editor and the official R tools for compilation and packaging.

However, the development of this dissertation has been performed on Windows XP using the WinShell [12], MiKTeX [46], Ghostscript [48] and Inkscape [54] tools.

4.1.3 Packaging Development Environment

The packaging development was also entirely performed on Linux since the development of documentation and packaging is bound. In fact, the package documentation is one of the steps of the packaging procedure, as stated in the official documentation [53]. Thus, the reasons for choosing Linux are the same stated in the previous section.

The packaging procedure was automated employing scripts. To do so, a R packaging script and a shell script, which can be seen in Appendix E, were written specifically for this task. Again, the JGR and Kate editors have been used for this task.

4.2 Implementation

As explained before in the previous chapter, R has been chosen as the target platform for the implementation of FRBF. First of all, R had to be learned, since the the author was unfamiliar with it. Fortunately, there is much literature about this subject [2, 8, 25, 52, 56]. R is quite easy to understand and it has a fast learning curve.

R provides the basic development mechanisms required for this kind of task, but it suffers from the fact that it is less used and widespread than other more common programming languages.

Since R is less used, the number of tools available and their functionalities cannot be compared with the ones available for more common and widespread programming languages. In particular, the development process and the debugging task are somewhat raw. R does provide the basic mechanisms but there are no fancy tools available to leverage these mechanisms and make them more user friendly or more productive.

Another real problem faced was the absence of documentation aimed for the development procedure. This forced the need for common tasks, like code documentation, to be specifically defined for the scope of this implementation.

4.2.1 Development

R does not have a real development manual procedure where standards, code documentation, good practices, design and organization are defined. The R Development Team provides a manual of R Internals [51] but it focus on tools for writing code outside R, like in C and Fortran, rather than focusing on a coding standard. For this reason, the author used his experience and a set of general good practices of software development to define specific development procedures for this implementation. Hence, a small standard has been defined from some easy and generic good practices to replace the missing R development procedure:

Declaration: all variables had to be declared and initialized;

Documentation: all the functions, classes and static constants had to be documented;

Structure: the code had to be organized through a logical domain group.

Declaration

R does not require for explicit variable declaration, it is enough to assign a value to a new variable in order to create it. Doing so will make R to automatically infer a data type and assigned it to the variable. For this reason, all variables used in the implementation of FRBF were initialized prior to its first use, and the explicit data type declaration was usually omitted. But in some particular cases it was necessary to explicitly define a data

type, usually by creating the variable with the correspondent data type constructor, or to enforce a data type, usually done through a data type conversion. The need for such data type conversions was specially common between the `matrix` and `data.frame` data types.

As a convention, names always started with a small letter. When it is required to compound words, both the camel case writing and the word separation with an underscore were accepted. The variables, constants, function names, function parameters, class names, and class slots all use this convention. An example this convention can be seen in Figure 4.1.

```
#
# Get Number of Clusters.
# Retrieves the number of clusters depending on
# each class variance.
#
# @param training_matrix:  the training data matrix
# @param classes:  array of classes
# @param config:  the algorithm configuration
# @return array of clusters per class
#
getNumberOfClusters <- function(training_matrix,
                                classes, config) {
  [...]
}
```

Figure 4.1: An example of code declaration and documentation.

This rule was also applied to the static constants declaration. R does not provide a specific static constant declaration, so a variable and a constant are only distinguished by the way the variable name is written. Constants are written with all capital letters and use an underscore as a word compound symbol.

Resulting from this rule, only S4 classes, as described in Section 3.5.1, were used. This kind of class enforces the declaration of class slots with a data type. The class constructor guarantees that all class slots are initialized. Sometimes default values are used for slots, allowing the user to omit it. If a class slot value is missing and there is no default value defined for its initialization, the constructor will issue an error and the FRBF execution will be halted.

Code Documentation

The resulting code of FRBF is entirely documented, not only because it is a good practice but also because of the algorithm complexity. A template has been defined and used for documenting the code. A function or class is documented in the code by having its defi-

nition preceded by a description and, when applied, an explanation about its signature. A comment line starts by an #, the R standard comment symbol. The function signatures use `@param <parameter>[:] <parameter description>` to document a parameter and `@return <result description>` to describe the result returned by the function. The descriptions are all free form. The Figure 4.1 illustrates this through a code snippet of a function documentation.

Structure

The implementation of the algorithm was organized following the steps described in Section 2.3.1, namely the two learning stages and the classification step. That resulted in five distinct R source code files, each containing specific definitions about a common domain. The script files are listed bellow by dependency order:

1. *Classes*, that holds the class definition, the object constructors and the required static values;

2. *Common*, that contains the functions which are auxiliary or commonly used in any part of the algorithm;

3. *Model*, which contains the functions responsible for the learning procedure that builds the RBF neural network model;

4. *Predict*, that holds the prediction function, responsible for the classification step; and

5. *Main*, which has the main functions, in particular the functions that are available to the end user.

4.2.2 Functions and Operators Used

One of R strongest points is the set of operators and functions available for mathematical calculus. The functions and operators are usually optimized and are able to perform complex calculations quite efficiently. The FRBF implementation took advantage of this by using the basic R functions and operators whenever possible. FRBF does not make use of any function outside the basic R installation, making it totally independent of external package.

Some of the most interesting functions used in the implementation relate with clustering, spectral decomposition and matrix computation and manipulation. Some of the functions and operators usage can be seen in the sample code of Appendix B.

Function diag

The `diag` function extracts or replaces the diagonal of a matrix, or constructs a diagonal matrix. In this implementation, it is used to construct a diagonal matrix. This function is very useful since it eliminates the need for a custom implementation of this utility.

Function kmeans

The `kmeans` function executes the K-Means clustering algorithm on a data matrix. The input data matrix is clustered by the K-Means method, which aims to partition the points into k groups such that the sum of squares from points to the assigned cluster centers is minimized.

The algorithm to use for clustering can be specified by the user. If no algorithm is specified, the function default algorithm will be used.

Function max.col

The `max.col`, also referred as `maxCol`, is a very useful function that finds the maximum position for each row of a matrix.

This function is used to find which column of a matrix holds the biggest numerical value per row. It performs very fast and discards the need for a custom implementation of such utility.

Function prcomp

The `prcomp` function performs a PCA on a given data matrix and returns a S3 class object. This function is used to obtain the eigenvalues, from the `sdev` slot, and the eigenvector, from `rotation` slot.

The calculation is done by a singular value decomposition of the data matrix, possibly scaled as it is one of the enhancements introduced and previously referred, and not by using the `eigen` function on the covariance matrix. This is generally the preferred method for numerical accuracy.

The standard deviations of the principal components, slot `sdev`, is the square roots of the eigenvalues of the covariance/correlation matrix. The calculation is actually done with the singular values of the data matrix.

The `rotation` holds the matrix whose columns contain the eigenvectors.

As stated in the manual [50], the signs of the columns of the rotation matrix are arbitrary, and so may differ between different programs for PCA, and even between different builds of R.

Function sd

The `sd` function computes the standard deviation of a matrix. As a result, a vector of the standard deviation of the columns is returned.

This function is used to quickly calculate the standard deviation of a matrix. It performs very well and discards the need for a custom implementation of this formula.

Function sum

The `sum` function sums all the values passed as an argument. Because of its flexibility, in this implementation it is used to sum distinct data types, mainly matrices and vectors.

This function performs a fast sum, regardless of the data type used as argument, and eliminates the need for several custom implementations of sum functions.

Operators

R provides a set of operators that allow the execution of complex, or tedious, operations on a simple and straightforward way.

For instance, the operator `%*%`, also referred as `matmult`, multiplies two matrices. If one of the operands is a vector, it will be promoted to either a row or column matrix to make the two arguments conformable. If both are vectors it will return its inner product as a matrix. The `[` and `[[` operators, and the corresponding closing brackets, are used for indexing and serve for structures like matrices, vectors and data frames. For instance, when working with matrices, a row or column can be fully specified by not specifying and index or by explicitly specifying a set of indexes. Another example is the `%in%` operator that yields true or false when a value belongs, or not, to a set.

Using such powerful operators makes the execution of certain tasks very easy. Figure 4.2 shows an example of the usage of such operators. Note that the omission of the row index selects all the rows, but the columns index specifically exclude one particular column.

```
data_matrix[ , !(colnames(data_matrix) %in% column_name)]
```

This results in the `data_matrix` without the `column_name` column.

Figure 4.2: An example of the operators usage.

4.2.3 Model

The FRBF implementation, following the prototype implementation described earlier, uses a model that can be saved for future usage. This potentiates the share of models trained with specific data sets.

The model keeps all the information that was used in the learning procedure plus the result of that same learning procedure. The `RemoraModel` is an S4 class with the following slots:

config is the configuration used to build the model, holds the S4 class object `RemoraConfiguration`;

model is the matrix with the model data information, in particular the class centroids and related information such as the spread parameter s;

lambda is the precalculated matrix values of Equation 2.7 per cluster; and

kernels is the list of the `RemoraKernels` S4 class object with the kernels that resulted from the K-Means.

The definition of the classes referred above can be found in Appendix A.2.

The configuration parameters are only used in the training procedure, hence it may seem odd to include the configuration as part of the model. But its inclusion is actually very useful because it helps to document and explain how the model was built. There is one exception to this though, the `verbose` configuration parameter, that can hold any of the `VERBOSE_OPTIONS` value described in Appendix A.1, is used in the classification procedure. This is totally dispensable as the user does not require to have any feedback about the classification procedure as it goes.

The model itself is the result of the training procedure of the FRBF algorithm. The function responsible for returning the model to the user is the `frbf` function which will be detailed later on this chapter.

4.2.4 Print

Since the FRBF implementation is based on classes created with the purpose to support the model, the `print` function has been overloaded. This mechanism is used to allow complex and compound data types to be displayed in a more user friendly output, whenever the user needs to inspect its values. Thus, the following classes are supported by the `print` function:

RemoraConfiguration prints all the configuration slot values;

RemoraKernels prints all the kernel slot values iterating whenever the values are lists; and

RemoraModel calls the print function for all the slots, dispatching the print to the correspondent class print function.

The complete list of the FRBF classes is available in Appendix A.2.

4.2.5 Learning

The learning procedure occurs through the `frbf` function. This function is responsible for the two learning stages of the FRBF algorithm, as described in the earlier Section 2.3.1.

The `frbf` function receives all the required parameters to build the model. Namely, it will receive the FRBF configuration parameters and the training data. When finished, it will return a model to the user, as already described in this chapter.

The function is actually quite simple. Following Algorithm 4.1, the first step is to build the learning configuration object from the user input values, which happens in the first line. The FRBF algorithm really starts by performing the K-Means in line 2. Then, the spectral decomposition takes place in line 3. Next, the best spread parameter values for each cluster are found in line 4. An information table per centroid is built from lines 5 to 7. Then, the kernel function values, here called lambda, are calculated in line 8. Based on the information calculated, the model is built in line 9, and returned to the user in the last line.

Algorithm 4.1 Overview of the `frbf` function steps

Require: Training data and parametrization values.

1: $config \leftarrow remoraConfiguration(user\ learning\ parameters)$
2: $kernels \leftarrow getKMeans(training\_matrix, config)$
3: $kernels \leftarrow getPCA(kernels, config)$
4: $s\_values \leftarrow findS(training\_matrix, kernels, model\_lambda, config)$
5: **for all** $centroid$ in $kernels$ **do**
6: $\quad centroid\_table[centroid] \leftarrow buildModel(kernels, s\_value[centroid], centroid)$
7: **end for**
8: $model\_lambda \leftarrow findLambda(training\_matrix, kernels, config)$
9: $model \leftarrow remoraModel(config, centroid\_table, model\_lambda, kernels)$
10: **return** $model$

This function is one of the user interface functions and its real implementation can be seen in Appendix B.2. The coming Section 4.4.1 will describe its signature, the user acceptable values and its usage.

4.2.6 Prediction

The classification occurs through the R `predict` function. This function has been overloaded in order to be used with the FRBF model and, therefor, it behaves just like the common R user expects. The overloading of the `predict` function is exemplified in Figure 4.3 and it follows the overloading mechanism described earlier.

The `predict` function will perform the classification for a given data set using the model data calculated in the learning procedure, as described in the classification task of Section 2.3.1. As stated before, the information contained in the `config` slot that is

used in this stage resumes itself to the `verbose` parameter. Its code is fully available in Appendix B.4 and, being one of the user interface functions, it can be seen in detail in the coming Section 4.4.2.

```
#
# Remora predict function.
#
# @param object remora model
# @param data_matrix the data to use to train
#       the algorithm or the data to use to classify
# @return prediction
#
predict.RemoraModel <- function(object, data_matrix, ...)  {
 [...]
}
```

Figure 4.3: Remora predict function overloading.

4.2.7 Tuning

The first R versions of the FRBF algorithm had a slow execution performance. This was mainly derived from the fact that the author, at the time, was still unfamiliar with many of the R functions and operators that are optimized to perform certain operations. For instance, matrix manipulation using an iterative process like looping for each column and row is way too slow when compared with the matrix function manipulation and index selection mechanisms.

Even with a not very fast execution, the performance was not an initial concern and therefor, it was actually one of the last modifications introduced to the FRBF implementation. But the performance became a critical issue. When some of the tests took too much time to build a model, it clearly became a problem that needed attention. In particular, the learning procedure for the *Satellite Image*, also known as *satimage*, data matrix from the StatLog [47] repository, with 4435 data points, 36 features and 6 classes, was taking over 19 hours to construct the model. Such problem clearly required a tuning process.

The tuning process was iterative and started by profiling the functions that had the most number of invocations during the learning procedure, and later evolved to all the important functions of the implementation. R does provide an amazing way to find where inefficiencies are in the code though the functions `Rprof` and `summaryRprof`. But the R profiling mechanism was not really useful since it is not able to trace uses of loops, like for, while and repeat. Consequently, it does not identify loops as the cause of inefficiency of the code. Hence, this task was performed with a naive approach by collecting timestamps in specific function points and analyzing where they where slow. Some of this

performance information is still available and can be seen by setting the configuration `verbose` parameter with the `debug` value.

The profiling showed that the functions were usually spending too much time performing object manipulations like inspecting the values of a matrix and seeking the column that had the biggest value per row. After collecting this profiling information, the tuning process began. Learning advanced R techniques was the main focus by the time, since it was necessary to acquire specific expertise to eliminate slow object manipulation. Each time a new technique was learned and applied, the performance got better. Learning advanced R techniques payed its profits and the earlier referred data matrix learning procedure start getting faster. When the tuning task was concluded, the model from the previously referred *satimage* data set was built in around 5 minutes.

Since this implementation is entirely coded in R, it will never perform as fast as a C coded version. This happens because R is interpreted and C is natively compiled, as described in the earlier Section 3.5.3.

4.2.8 Problems Found

Several challenges have been overcome during the FRBF implementation. Some of them have already been described in the previous sections.

One of the first problems encountered was the inexistence of an Integrated Development Environment (IDE) that allowed an easy and fast way to write, debug and refactor the R code. Later, a performance problem raised and the same happened with the tuning process, where the profiling had to be performed through naive techniques has explained in the previous section. All this caused a longer and slower development and evolution of the algorithm. Compared with current common IDEs and profiling tools available for other programming languages, this can be seen as a productivity issue.

Another problem found was the missing of a R development manual with the definitions for standards, code documentation, good practices, design and structure. This was overcame by the definition of specific rules for this development, has already describred.

The FRBF algorithm complexity was also a challenge. During its implementation several bugs raised from the algorithm interpretation and its enhancements. This is a common situation of the development process, but coding the algorithm correctly was not achieved at the first try. In particular, the adjustment of appropriate spread parameters detailed in Algorithm 2.2, the `findS` function, visible in Appendix B.1, required special detailed attention. The values returned by it were very disparate from the ones obtained by the prototype, which clearly indicated that the function had problems. Many debug sessions were performed around this function before it become correct.

4.3 Tests

The tests of the FRBF were performed with several testing methods and data sets. The testing methods made use of distinct approaches, namely (i) unit testing, to test functions individually; (ii) black box testing, to test the algorithm learning and classification procedures; and (iii) accuracy testing, to assert about the quality of its results and in which the data sets were particularly relevant.

The unit testing focused only on critical functions that required special attention. The tests were performed by calling the function with a specific set of input arguments and confronting the output result with the expected correct result. This method allowed to test the functions individually.

In the black box testing, the goal was to test the functions integration. This test was performed by calling the available user interface functions, described in the upcoming Section 4.4, and checking if it behaved correctly. Figure 4.4 exemplifies the output result of a black box test script, available in Appendix C.

The accuracy testing aimed at certifying that the algorithm implementation was returning good results. This was achieved by confronting the R implementation results with the prototype results for the same configuration and data sets. In this test, several data sets have been used, like the classical *iris* and the StatLog [47] repository data sets. The data sets assume a particular importance in this test, since they are widely known and used by the scientific community. Table 4.1 characterizes some of the most interesting data sets used in the accuracy tests.

Dataset	# Training Patterns	# Testing Patterns	# Features	# Classes
iris	125	25	4	3
wdbc	569	80	30	2
satimage	4435	2000	36	6
shuttle	43500	14500	9	7

Table 4.1: FRBF testing data sets.

4.3.1 Execution Behaviors Observed

The tests revealed some specific behaviors about the FRBF execution.

An expected behavior that was observed during the tests was that the FRBF execution is CPU bound. This happens because the FRBF is computation intensive, since it performs many calculus with matrices. On the other hand, it is memory efficient, since it does not require much RAM to perform the calculations with big data sets. This happens even when several structures loaded with thousands of data points are loaded in memory. For instance, a matrix with 4435 data points and 37 columns, *i.e.* with 164095 singular values, requires approximately 32MB of RAM, including the R environment.

```
Performing full tests on frbf using wdbc.
[...]
Configuration [1]:  function is euclidean, algorithm is
   Hartigan-Wong, scale variance is TRUE, perform sum is TRUE.
Accuracy [1]:  0.7594937
Train Accuracy:  0.7820738
Model [1]:  1.57995 minutes.
Prediction [1]:  0.001316667 minutes.
Configuration [2]:  function is euclidean, algorithm is
   Hartigan-Wong, scale variance is TRUE, perform sum is FALSE.
Accuracy [2]:  0.949367
Train Accuracy:  0.8857645
Model [2]:  1.5703 minutes.
Prediction [2]:  0.001316667 minutes.
[...]
Configuration [52]:  function is mahalanobis, algorithm is
   Hartigan-Wong, scale variance is FALSE, perform sum is FALSE.
Accuracy [52]:  0.9746835
Train Accuracy:  0.913884
Model [52]:  0.5627667 minutes.
Prediction [52]:  0.001033334 minutes.
Configuration [53]:  function is mahalanobis, algorithm is
   Lloyd, scale variance is TRUE, perform sum is TRUE.
Accuracy [53]:  0.721519
Train Accuracy:  0.775044
Model [53]:  2.29115 minutes.
Prediction [53]:  0.001316667 minutes.
[...]
Configuration [141]:  function is normalized_difference_sq,
   algorithm is MacQueen, scale variance is TRUE,
   perform sum is TRUE.
Accuracy [141]:  0.9367089
Train Accuracy:  0.8945518
Model [141]:  0.9513 minutes.
Prediction [141]:  0.001300001 minutes.
Configuration [142]:  function is normalized_difference_sq,
   algorithm is MacQueen, scale variance is TRUE,
   perform sum is FALSE.
Accuracy [142]:  0.949367
Train Accuracy:  0.8927944
Model [142]:  1.001033 minutes.
Prediction [142]:  0.001300001 minutes.
[...]
```

Figure 4.4: Black box test script execution example.

The tests also revealed one particular behavior about the adjustment of the spread parameters, detailed in Algorithm 2.2. After all clusters have been stabilized, it was common for the `findS` function, visible in Appendix B.1, to grab one cluster and successively lower its s value, making a broad Gaussian as visible in Figure 2.3, in order to expand that same cluster and thus capture more data points. This happens while a better accuracy is obtained, even if between iterations only one single data point is better classified. Figure 4.5 exemplifies this behavior, the • that belongs to both clusters was caught because the • cluster expanded in a broad way, otherwise it would have not been caught by it and would have been classified as a ■.

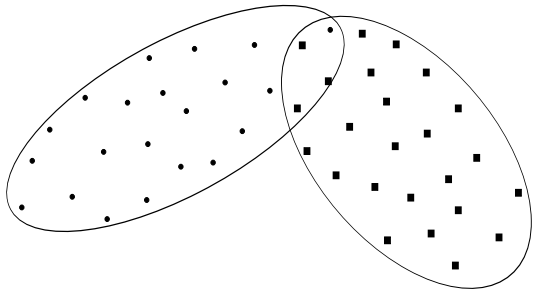


Figure 4.5: A FRBF cluster grab making a broad Gaussian example.

4.3.2 Results

Overall, after solving the problems, the result of the tests were very satisfying. Considering the complexity involved and the number of configuration possibilities available, the tests showed a solid implementation of the FRBF algorithm and of its enhancements.

Accuracy

A special attention was given to the accuracy test results, the only test where the author was not entirely satisfied with the results obtained.

During this test, some discrepancies appeared in the results obtained from the R and the prototype implementations. First, from the fact that some functions had bugs, as previous described. And later, after the bug fixing, from the fact that the spectral decomposition is performed differently in the prototype and in this R implementation. In fact, R itself has distinct ways to obtain the eigenvalues and eigenvectors. The manual [50] states that `prcomp` yields better results than the `eigen` function, but even though, tests were performed with both to check which returned better results. The `prcomp`, described earlier, was the original choice and it was kept, since when it was used the algorithm returned a better accuracy. But even using this function, the results from the prototype were usually more accurate. A detailed analysis was performed to confirm that they diverged precisely

in the PCA, but no changes were made to substitute the internal R function `prcomp`. Table 4.2 shows a comparison of the best accuracy obtained using the prototype and the FRBF implementation for the same data set and parameterization.

Dataset	FRBF Accuracy		Prototype	
	Train (%)	Classification (%)	Train (%)	Classification (%)
satimage	84.23	84.46	87.46	85.30
wdbc	92.44	97.46	100	100
shuttle	98.39	98.43	100	100

Table 4.2: FRBF and prototype accuracy comparison results.

An example of the best results from the accuracy tests are shown in Table 4.3. The table shows some of the most interesting data sets used, part of the configuration applied to it, the learning procedure computation time and the accuracy results obtained for the training and the classification data sets. The training accuracy is calculated by applying the learned model to the training data set.

Dataset	Kernel		Train		Classification
	Function	#	Time (min.)	Accuracy (%)	Accuracy (%)
satimage	(0)	6	5.40	79.32	79.9
satimage	(3)	6	4.40	84.23	84.46
wdbc	(0)	3	1.55	89.98	97.46
wdbc	(3)	4	0.15	92.44	97.46
iris	(4)	3	0.07	93.6	96
shuttle	(3)	7	153.64	98.39	98.43

Table 4.3: FRBF training and testing accuracy results.

One curious result was the classification accuracy obtained with the *wdbc* data set using the Mahalanobis and the Euclidean kernel functions. Using distinct configurations resulted in the same classification accuracy. This was a rare case, since the tests indicate that, typically, the Mahalanobis function performs better than the Euclidean function, as exemplified by the *satimage* test results on Table 4.3. The learning procedure is usually not time consuming, but that was not the case with the *shuttle* data set, where the learning procedure time was a lot longer than in the rest of the examples. This happened because the training data set is much bigger than the rest of the presented data sets, as visible in Table 4.1, and thus a lot more calculations are performed. As an example, for finding the spread parameter s for *satimage*, each iteration performs 783000 calculations, which is clearly computation intensive and time consuming.

Iterations

The tests revealed that the number of iterations to find the adjustments of the spread parameters was actually very low, no matter how big the training data set was.

The default value is set to 5% of the number of data points involved in the training. But the tests showed that the s value was usually found with much less iterations, most of the times with less than 25% of the default value.

Parameter Influence

There is a set of parameters that clearly influence the accuracy more than other parameters.

When using the same configurations, the tests showed that the Mahalanobis kernel function usually performs better than the other available functions. The number of clusters is, obviously, one of the parameters that directly influences the accuracy result. This is because it has a direct impact in the K-Means accuracy result, which is one of the most important factors of the learning procedure.

On the other hand, the selected K-Means algorithm, performing the sum, or not, of the centroids distance per class and changing the PCA scale parameter seems to have less influence in the final result. Usually the gain of changing one of these parameters is residual, but that is not always the case, where the differences in accuracy can be high. The test example in Figure 4.4 shows some of these variations. In the example, the Configuration 1 and 2, and the Configuration 141 and 142, only differ in the perform sum value. While the first case as a difference of 19%, the second case only differs by 1%. The combination of several of these parameters can result in greater differences, like in the case of Configuration 52 and 53, that differ by 25%.

All this shows that FRBF is extremely flexible and allows the user to test many different parameterizations in the search of the best model.

4.4 User Interface

Once the testing phase has been concluded the FRBF was ready for usage. The interaction between the user and the FRBF is performed through a set of two functions. These functions work as the user interface and can be used just like any other R function. If exported they can be seen as an Application Programming Interface (API). The following two sections describe these user interface functions in detail.

4.4.1 FRBF

The `frbf` is the function responsible for the learning procedure. It implements the Stage One and Stage Two of the Flexible Kernels Learning Procedure, as detailed in the Algorithms 2.1 and 2.2 from the earlier Section 2.3.1. It also implements the algorithm improvements described in the Section 2.5. As a result, the `frbf` function returns a model, an object of the `RemoraModel` S4 class as already described.

The `frbf` function has the following signature:

data_matrix: the training data, must be a matrix or a data frame;

number_clusters: the total number of clusters, may be adjusted during the execution and will be used by the K-Means algorithm, see the `kmeans` R function in [50];

class_name: the name, or index, of the column that holds the class of the training data matrix;

weighting_function: the name of the kernel function, if none is specified the Euclidean function will be used, the allowed values for this parameter can be seen in Table 4.4;

scale_variance: specifies if the scale should be performed for the principal components analysis, default is True, see the `prcomp` R function in [50];

s_value: the initial s value to use to find the kernels sigma value, the spread parameter adjustment, it has a default value of 0.2;

d: the initial d value to use to find the s value, it will use the default value of 0.23 if no value is specified;

epsilon: the ε value for the functions that require it, if none is specified a default value of 0.01 will be used;

niter: the maximum number of iterations to perform to find s, if no value is provided, a default will be calculated based on the number of training data points;

niter_changes: the number of iterations without changes that can occur, if this number is reached without any change, the iteration will stop, if no value is specified 5 will be used by default;

perform_sum: specifies if the sum of the centroids per cluster should be applied, or not, default is True;

clustering_algorithm: specifies which of the K-Means algorithms should be used, if none is specified, the default K-Means algorithm will be used, see the `kmeans` R function in [50];

verbose: specifies the algorithm verbosity during the execution, if nothing is specified it will be silent, the allowed values for this parameter can be seen in Table 4.5; and finally it

Returns: the FRBF neural network model.

The R code implementation of the `frbf` function can be seen in Appendix B.2. A later section shows how this user interface function can be used.

#	Value	Formula
(0)	euclidean	1
(1)	one_minus	$(1-\lambda)$
(2)	one_minus_sq	$(1-\lambda)^2$
(3)	mahalanobis	$1/(\lambda+\varepsilon)$
(4)	exp_one_minus	$exp(1-\lambda)$
(5)	exp_one_minus_sq	$exp(1-\lambda)^2$
(6)	exp_one_log	$(1-log(\lambda+\varepsilon))$
(7)	normalized_difference	$(1-\lambda)/(1+\lambda)$
(8)	normalized_difference_sq	$((1-\lambda)/(1+\lambda))^2$

Table 4.4: `weighting_function` parameter values, following Falcão *et al.* [14].

no	yes	detail	debug

Table 4.5: Acceptable values for `verbose` parameter.

4.4.2 Predict

The `predict` function is responsible for the classification procedure. It implements the Classification using the Flexible Kernels, as described in the earlier Section 2.3.1. As previous stated, this function overloads the basic `predict` function for the `RemoraModel` S4 object class. As a result, the `predict` function acts just as the user expects, it receives a model and the data to classify and returns a prediction about the classification.

The `predict` function has the following signature:

object: remora model, obtained from the learning procedure;

data_matrix: the matrix, or data frame, containing the data to classify; and it

Returns: prediction through an array containing the class of each data point.

The R code implementation of the `predict` function can be seen in Appendix B.4. The next section shows how this user interface function can be used.

4.4.3 Usage

Having both functions available, its usage is quite easy. As already referred, the model can be saved for later usage, this is also exemplified in this section.

Function frbf

First, the user must train the algorithm with a training data set so that it can learn and build the artificial neural network and return its model. This is performed by calling the `frbf` function.

For instance, `model <- frbf(training_matrix, class_name="class", weighting_function="mahalanobis", number_clusters = 7)` builds a model of 7 clusters by applying the `mahalanobis` kernel function over the training data matrix `training_matrix`. The model is returned and assigned to the `model` variable. Figure 4.6 exemplifies this procedure combined with the classification task.

Function predict

After the RBF neural network model has been built, it will be used to make predictions about the class of new, unseen, data. This is performed by calling the `predict` function.

For instance, `classification <- predict(model, data_matrix)` will apply the model `model`, learned from the application of the `frbf` function, to classify the data set `data_matrix`. Figure 4.6 exemplifies this procedure combined with the learning task.

Object save and load

The model obtained from the training procedure can be saved for latter usage, or for distribution. This is a very interesting feature that R provides to its users. Using R basic functions `save` and `load` makes this task quite easy.

For instance, the `save(model, file = "model.Rdata")` function will save the model object into the `model.Rdata` file, while the `load("model.Rdata")` function will load the model object back from the `model.Rdata` file into the current workspace. Figure 4.6 exemplifies how the model can be saved for later usage after the learning task has been concluded.

4.5 R Packaging

The final stage of the FRBF implementation is its packaging for distribution. Packaging the FRBF includes its source code, the user documentation and a test script that is executed for validation. As previously stated, this has been entirely performed under Linux, but following Rossi in [44] allows anyone to set up a Windows system to accomplish this task.

The packaging procedure is well documented in the R official documentation [53], and several easy to follow documents about this subject are available, such as the tutorials from Gómez-Rubio [18] and Gentleman [16].

The execution of this task involves several distinct steps, (i) the creation of the package structure for the FRBF, (ii) the inclusion of the help files that document the package, and (iii) the build of the distributable package file.

```
# Load the training data from a file
training_matrix <- read.csv(file=
        '/thesis/datasets/wdbc/wdbc_train.csv', header=TRUE)

# Train the RFB network and get the resulting model
model <- frbf(training_matrix, class_name="class",
        weighting_function="mahalanobis", number_clusters = 7)

# Save the model for later usage
save(model, file = "/thesis/models/wdbc_model.Rdata")

# Load the data to classify from a file
data_matrix <- read.csv(file=
        '/thesis/datasets/wdbc/wdbc_unknown.csv', header=TRUE)

# Predict the classification using the model
classification <- predict(model, data_matrix)

# Show the classification
print(classification)
```

Figure 4.6: Example of the FRBF functions usage.

4.5.1 Package Structure

The first step of the packaging procedure is to gather the functions that will be packed and build a structure, in a local directory, that provide the basis for the other upcoming two steps. R has two ways to perform the packaging, (i) pack the information that is in memory, and (ii) pack from a given set of files. The second approach has been used since the first one seemed incompatible with the S4 object class, as detailed in an upcoming section. A small build script, detailed in Appendix E.1, was written specially for this purpose. The packaging structure created by the execution of this script is a set of directories and files bellow the `<package name>` directory:

DESCRIPTION: the information description of the package, such as the name and license;

NAMESPACE: the list of functions and classes to export, meaning the ones that will be visible to the user;

man/<**package**>**-package.Rd:** the package help file that documents the package;

man/<**class_name**>**-class.Rd:** the classes help files, one per each class;

man/<**function_name**>**.Rd:** the functions help files, one per each function;

R/ the directory of the R source code, since the file list method is being used it simply copies the specified files into this directory;

4.5.2 Help Files

Once the structure has been created, it is time to document the package and its functions through the creation of the correspondent help files.

The help files go under the `man/` directory and are R documentation files, with the `.Rd` extension, and are actually LaTeX files in its essence. Murdoch in [37] describe the technical side of these files for documentation purpose and Gentleman in [17] introduces how to document the functions by writing this user help files for packaging purposes. More generic documentation and books about writing LaTeX, for both beginners and advanced users, is quite easy to obtain, as [32, 30, 31] are examples of.

A special note about the section named `examples` in the `.Rd` files. It contains R code that is used to show an example of the function usage. But this R code is also used in the latter packaging step to validate that the user will have a correct running example. These LaTeX files will be compiled to provide the documentation when the user requests the help for the FRBF package functions.

Each time the packaging mechanism runs, it will overwrite the existing files, so, once created, the `.Rd` files are kept in a safe place and will be copied into the `man/` directory whenever the automatic packaging shell script is executed. The packaging shell script is available in Appendix E.2.

4.5.3 Distribution File

The final step of the packaging mechanism is to create the distribution file. This is achieved by compiling the information contained in the package structure and collect it into a single tarball gziped file with the `.tar.gz` file extension.

The package must be validated before the distribution file can be built. This is performed by the `R CMD check <package>` system command. The R code provided in the `examples` section of the help files is executed to validate that the user will end up with a correct running example. If the R code provided does not execute correctly, the check command will not validate the package. This command creates a structure under the `<package name>.Rcheck` directory that contains information about this validation, like the check log file.

Finally, the distributable source package file can be built. This is done by running the system command `R CMD build <package>`, that creates the `<package>.tar.gz` file for CRAN submission. A binary distribution for a specific platform can be created through the `R CMD --binary --use-zip build <package>` command. The main differences from this two distribution packages come from the fact that the first does not

contain the files compiled, meaning they will be compiled latter, either by the CRAN for distribution or at install time in a system that has all the necessary tools to make the build..

The FRBF has been built using the `frbf` as the package name and the GNU LGPL as the usage license, originating the creation of the `frbf_1.0.tar.gz`, where 1.0 indicates the package version. The package has been submitted to the CRAN repository and is already available, allowing the scientific community to download and explore it.

4.5.4 Problems Found

The packaging brought up some challenges of itself.

As previously stated, R has two ways to perform the packaging (i) pack the information that is in memory, and (ii) pack from a given set of files. The first approach used the packaging mechanism that gathers the information from the current environment. It issued some warnings but the installation package was created. The problem came when trying to install the package, a critical error with little information halted the installation. This was a serious problem, since there was not much information about it. There were many people complaining about this error, specially in the R developing groups, but only a couple of hints to solve it that did not work. The investigation to understand the cause of the problem and, consequently solve it, began. It took some time to find what was causing the problem. It was related with the S4 class objects, that somehow seemed to be unfriendly with this packaging method. So, the later packaging mechanism had to be used and a small build script, as detailed in Appendix E, was written specially for that purpose. With this new mechanism in practice, all the packaging warnings and installation errors disappeared.

The creation of the help files using Latin characters was also a challenge. The support of a specific encoding is not always easy to perform, since the `.Rd` files are specifically processed and are not directly compiled by the usual LaTeX process. To overcame this, a specific encoding command must be included in the `.Rd` files, but unfortunately it did not solved the problem. Hence, the solution was to replace the Latin characters with standard non-accentuated ASCII characters.

4.5.5 Installing and Uninstalling

The installation and removal of an R package is quite easy, therefor installing and uninstalling the `frbf` package is straightforward.

The installation of the package can be performed through the system command `R CMD INSTALL frbf_1.0.tar.gz`. This command will install the package in the current R installation. Unless the package is already compiled, R will need to compile the package before installing it. This means that the system must have all the required tools for this task, as stated in the official documentation [53]. The package can be loaded like any

other package though the usage of the `library` function. The deletion of the installed package is done through the system command `R CMD REMOVE frbf`.

A simpler way to perform this actions is to use the CRAN repository, that already has the FRBF package compiled for all the platforms. Using the graphical R interfaces eases the execution of this task. Once installed, the `frbf` package can be immediately loaded and its functions used.

Chapter 5

Conclusions

As stated in the introduction of this dissertation, the author was motivated by the opportunity to provide an easy way for the scientific community to use the FRBF approach proposed by André Falcão *et al.* in *Flexible Kernels for RBF Networks* [14]. Hence, the main focus was set in providing an implementation of the FRBF that could be easily used by everyone and integrated with, or within, a framework or third party applications. This was also a great opportunity to perform some enhancements to the original algorithm such as providing a wider range of parameterization.

This chapter concludes this dissertation and covers the work performed, the scientific contribution and the future work on FRBF.

5.1 Work Performed

The work performed to achieve the goals started by understanding the FRBF algorithm and identifying the enhancements that could be included in this new implementation.

After that, the selection of the platform for the new implementation took place. The R framework has been selected because of its great features. It is an open and extensible platform with a repository that allows an easy way to distribute the new implementation as an expansion package. It is also widely used for statistical software development and data analysis, making it *de facto* standard among statisticians.

Once the platform has been selected, the implementation took place. First of all, it was necessary to learn R and define a set of development standards. Then, the FRBF algorithm development, and the identified improvements, took place. A profiling and tuning process was performed because of some execution inefficiencies that were detected. There was a set of tests, using distinct methods and data sets, that covered the development of the algorithm and its accuracy in order to certify that it was ready for distribution.

After the FRBF algorithm has been developed, the next step was to write the user documentation and build the distribution package.

5.2 Release

As stated before, an implementation of the FRBF was developed in the R framework and distributed as an expansion package for the scientific community. This package is licensed to the end user under LGPL, allowing anyone to make use of the FRBF algorithm in a wide range of usages and, eventually, improve it.

Delivering this FRBF improved implementation as an open source R expansion package fully covers the goals of this work.

5.3 Future Work

As a future work some improvements can be made to the FRBF.

For instance, the performance could be improved. Developing this FRBF implementation in C, or C++, would certainly allow much faster computations, allowing an obvious time reduction for the learning procedure.

A special attention was payed to the accuracy results, the only test where the author was not entirely satisfied because of the R PCA calculation differences when compared with the prototype implementation. A different spectral decomposition to provide a better accuracy would be much appreciated, since the accuracy tests have identified that there is space for improvement in this area.

Appendix A

Static Definitions

The definitions used in the implementation of FRBF are detailed in the following sections. This covers both the constants and the classes used on the code.

A.1 Constant Definition

The constants are used all over the implementation.

```
#
# Remora weighting functions available
#
# 0 Euclidean (default)
# 1 One Minus
# 2 One Minus Squared
# 3 Mahalanobis
# 4 Exp One Minus
# 5 Exp One Minus Sq
# 6 Exp One Log
# 7 - Unimplemented
# 8 Normalized Difference
# 9 Normalized Difference Sq
FUNCTION_REMORA_EUCLIDEAN <- "euclidean"
FUNCTION_REMORA_ONE_MINUS <- "one_minus"
FUNCTION_REMORA_ONE_MINUS_SQ <- "one_minus_sq"
FUNCTION_REMORA_MAHALANOBIS <- "mahalanobis"
FUNCTION_REMORA_EXP_ONE_MINUS <- "exp_one_minus"
FUNCTION_REMORA_EXP_ONE_MINUS_SQ <- "exp_one_minus_sq"
FUNCTION_REMORA_EXP_ONE_LOG <- "exp_one_log"
FUNCTION_REMORA_NORMALIZED_DIFFERENCE <- "normalized_difference"
FUNCTION_REMORA_NORMALIZED_DIFFERENCE_SQ <-
                                        "normalized_difference_sq"
FUNCTIONS_REMORA = c(FUNCTION_REMORA_EUCLIDEAN,
                     FUNCTION_REMORA_ONE_MINUS,
                     FUNCTION_REMORA_ONE_MINUS_SQ,
                     FUNCTION_REMORA_MAHALANOBIS,
```

```
                      FUNCTION_REMORA_EXP_ONE_MINUS,
                      FUNCTION_REMORA_EXP_ONE_MINUS_SQ,
                      FUNCTION_REMORA_EXP_ONE_LOG,
                      FUNCTION_REMORA_NORMALIZED_DIFFERENCE,
                      FUNCTION_REMORA_NORMALIZED_DIFFERENCE_SQ)


#
# Verbose
#
# No verbose, means silent
VERBOSE_NO <- "no"
# Display some information
VERBOSE_YES <- "yes"
# Displays detailed information
VERBOSE_DETAIL <- "detail"
# Displays debug information
VERBOSE_DEBUG <- "debug"
VERBOSE_OPTIONS <- c(VERBOSE_NO, VERBOSE_YES,
                     VERBOSE_DETAIL, VERBOSE_DEBUG)
```

A.2 Class Definition

The S4 class definitions used in the FRBF implementation are presented below. Note that the class name it actually declared as a constant, which have been covered here and not in the previous Appendix A.1 for easiness of understanding. This was a design option that allows less reference errors and an easier way to identify a reference in the code.

```
#
# Class definitions.
# S4 implementation.
#

#
# Class distance definition.
#
CLASS_REMORA_DISTANCE <- "RemoraDistance"
class_distance <- setClass(CLASS_REMORA_DISTANCE,
  representation(index = "numeric", distance = "numeric",
                 className = "character", point = "list"),
  prototype = list(index=numeric(), distance=numeric(),
                   className=character(), point=list()))

#
# Class configuration definition.
#
CLASS_REMORA_CONFIGURATION <- "RemoraConfiguration"
class_configuration <- setClass(CLASS_REMORA_CONFIGURATION,
  representation(number_clusters = "numeric",
```

```
                  class_name = "character",
                  weighting_function="character",
                  clustering_algorithm="character",
                  scale_variance="logical", s="numeric",
                  d="numeric", epsilon = "numeric",
                  niter="numeric", niter_changes="numeric",
                  perform_sum="logical", verbose="character"),
  prototype = list(number_clusters=numeric(),
                    class_name=character(),
                    weighting_function=character(),
                    clustering_algorithm=character(),
                    scale_variance=logical(), s=numeric(),
                    d=numeric(), epsilon=numeric(),
                    niter=numeric(), niter_changes=numeric(),
                    perform_sum=logical(), verbose=character()))

#
# Class remora kernels
#
CLASS_REMORA_KERNELS <- "RemoraKernels"
class_kernels <- setClass(CLASS_REMORA_KERNELS,
  representation(class_name="character", eigen_values = "list",
                  eigen_vector = "list", clusters = "numeric",
                  cluster_points = "data.frame",
                  points_per_cluster = "list",
                  centroids="matrix", size="numeric"),
  prototype = list(class_name=character(), eigen_values=list(),
                    eigen_vector=list(), clusters = numeric(),
                    cluster_points = data.frame(),
                    points_per_cluster = list(),
                    centroids = matrix(), size = numeric()))


#
# Class model definition.
#
CLASS_REMORA_MODEL <- "RemoraModel"
class_kernels <- setClass(CLASS_REMORA_MODEL,
  representation(config=CLASS_REMORA_CONFIGURATION,
                  model="data.frame", lambda="list",
                  kernels="list"),
  prototype = list(config=new(CLASS_REMORA_CONFIGURATION),
                    model=data.frame(), lambda=list(),
                    kernels=list()))
```

Appendix B

FRBF Code Sample

The FRBF is open source, thus all its code is available. Nevertheless some of the most interesting FRBF implemented code is available in the sections that follow.

B.1 Find S

The `findS` function is one of the crucial functions in the FRBF algorithm. It is responsible for finding the best spread s values for each cluster and makes use of the `initS` function to initialize the s values.

```
#
# Finds the s value for each cluster.
#
# @param training_matrix: training data matrix
# @param kernels: the kernels (found on stage one)
# @param model_lambda: the previously calculated lambda
#                      function values
# @param config: the remora configuration
# @return s parameter per cluster
#
findS <- function(training_matrix, kernels, model_lambda,
config) {
  if (verbose.showDetail(config@verbose)) {
    cat('\tFinding S values:\n')
    flush.console()
  }

  number_points <- nrow(training_matrix)
  test_matrix <- as.matrix(getUnclassedMatrix(
                           training_matrix, config@class_name))
  class_names <- names(kernels)
  best_kernels_s <- list()
  test_kernels_s <- list()
  kernels_s_up <- list()
  kernels_s_down <- list()
```

```
# if necessary calculates the niter parameter
if (config@niter < 0) {
  config@niter <- round(nrow(training_matrix) * 0.05)
  warning("niter parameter has been calculated,
           value is ", config@niter)
}
# if necessary adjusts the niter parameter to a minimum value
if (config@niter < 10) {
  config@niter <- 10
  warning('niter was too low, it has been
           redefined to ', config@niter)
}

# initialize values
distance_table <- buildDistanceTable(test_matrix, kernels,
                                     model_lambda, config)
d <- config@d
d_start <- d # prototype uses 0.23
d_end <- 0.01

iter <- 1
random_index = vector()
flat_index <- list()
# structure is passed into a flat structure and
# point cluster index is added
accuracy_matrix <- training_matrix
new_cluster_column_index <- ncol(training_matrix) + 1
for (classes in sample(names(model_lambda))) {
 for (lambda in sample(c(1:length(model_lambda[[classes]])))) {
  flatIndex <- getFlatIndex(classes, lambda)
  flat_index[flatIndex] <- flatIndex

  points_per_cluster <- kernels[[classes]]@points_per_cluster
  for (ppc_idx in (c(1:length(points_per_cluster)))) {
   ppc <- points_per_cluster[[ppc_idx]]
   for (idx in ppc["point_index"]) {
    accuracy_matrix[idx, new_cluster_column_index] <-ppc_idx;
   }
  }
 }
}
names(accuracy_matrix) <- c(names(training_matrix),
                            "cluster_index")
best_kernels_s <- initS(distance_table, kernels,
                        accuracy_matrix, config)
kernels_s_up <- best_kernels_s
kernels_s_down <- best_kernels_s
last_change <- config@niter_changes
```

```
best_hit <- 0

if (verbose.showDetail(config@verbose)) {
  cat('\t\tMaximum number of iteractions: ')
  cat(config@niter)
  cat('\n\t\tIteractions: ')
  flush.console()
}

# iter

time_start <- unclass(Sys.time())
while (last_change >= 0 && iter < config@niter) {
  if (verbose.showDebug(config@verbose)) {
  cat('\n\t\t#')
    cat(iter)
    cat(':\n')
    flush.console()
  }
time_start_iter <- unclass(Sys.time())
  # iterate over random indexs
  for (index in sample(flat_index)) {
    #
    # try s up
    #
    test_kernels_s <- best_kernels_s
    kernels_s_up[index] <- as.numeric(kernels_s_up[index])
                           * (1 + d)
    test_kernels_s[index] <- kernels_s_up[index]

    # get distances for new s up value
    dst_up <- distances(distance_table, kernels,
                        test_kernels_s, config)

    # classification for the distances found
    class_up <- buildClassification(dst_up, config)

    # accuracy for the distances
    hit <- accuracy(accuracy_matrix, class_up, config,
                    FALSE)

    if (hit > best_hit) {
      # new, better s value found
      if (verbose.showDebug(config@verbose)) {
        cat('\t\t\ts upper value ')
        cat(test_kernels_s[[index]])
        cat(' is better for ')
        cat(index)
        cat(' with hit of ')
```

```
      cat(hit)
      cat(';\n')
      flush.console()
    }
    best_kernels_s[index] <- test_kernels_s[index]
    best_hit <- hit
    last_change <- config@niter_changes
  }

  #
  # try s down
  #
  test_kernels_s <- best_kernels_s
  kernels_s_down[index] <- as.numeric(kernels_s_down[index])
                           * (1 - d)
  test_kernels_s[index] <- kernels_s_down[index]

  # get distances for new s up value
  dst_down <- distances(distance_table, kernels,
                        test_kernels_s, config)

  # classification for the distances found
  class_down <- buildClassification(dst_down, config)

  # accuracy for the distances
  hit <- accuracy(accuracy_matrix, class_down, config,
                  FALSE)

  if (hit > best_hit) {
    # new, better s value found
    if (verbose.showDebug(config@verbose)) {
      cat('\t\t\ts lower value ')
      cat(test_kernels_s[[index]])
      cat(' is better for ')
      cat(index)
      cat(' with hit of ')
      cat(hit)
      cat(';\n')
      flush.console()
    }
    best_kernels_s[index] <- test_kernels_s[index]
    best_hit <- hit
    last_change <- config@niter_changes
  }

}
d <- d_start+iter/config@niter*(d_end-d_start);
if (d < d_end) {
  cat("\t\tunexpected d < d_end\n")
```

```
      d <- d_end
    }
    last_change <- last_change - 1
    iter <- iter + 1
  time_end_iter <- unclass(Sys.time())
    if (verbose.showDebug(config@verbose)) {
      cat('\t\t#')
      cat(iter-1)
      cat(': ')
    cat(time_end_iter - time_start_iter)
    cat(' seconds\n')
      flush.console()
    }
  }

  time_end <- unclass(Sys.time())

  if (verbose.showDetail(config@verbose)) {
iter_time <- time_end - time_start
    cat('\n')
    cat('\tNumber of iteractions performed: ')
    cat(iter-1)
    cat('\n')
   cat('\tTime spent: ')
   cat(iter_time/60)
   cat(' minutes.\n')
   cat('\tAverage time per iteraction: ')
   cat(iter_time/iter)
   cat(' seconds.\n')
   cat('\tFinal sigma values:\n')
   print(best_kernels_s)
    flush.console()
  }

  best_kernels_s
}
```

B.2 FRBF

The learning procedure of the FRBF algorithm is encapsulated in the `frbf` function. This function is the learning procedure algorithm entry point for the user. It returns the model, in a `RemoraModel` class, as describe in Appendix A.2, which will be used by the `predict` function, described in the Appendix B.4.

```
#
# Remora model function.
# The learning procedure of the frbf algorithm.
#
# @param data_matrix: the data to use to train the algorithm
```

```
# @param number_clusters: is the number of clusters to use in
#           the training part
# @param class_name: is the name, or index, of the training
#           data matrix class column
# @param weighting_function: is the name of the weighting
#           function to use in the classification process
# @param scale_variance: specifies if the scale should be
#           performed for the principal components analysis,
#           default is True (@see prcomp)
# @param s_value: is the initial s value to use to find the
#           kernel sigma value
# @param d: is the initial d value to use to find the s value
# @param epsilon: is the epsilon value for function, only for
#           functions that require it
# @param niter: is the maximum number of iterations to perform
#           to find s, if no value is provided, a default will
#           be calculated based on the number of training data
#           points
# @param niter_changes: is the number of iteration without
#           changes that can occur, if the number of
#           niter_changes is reached without any change, the
#           iteration will stop, a default value will be used
#           if none is specified
# @param perform_sum: specifies if the sum of the centroids
#           per cluster should be applied, or not
# @param clustering_algorithm: specifies which of the k-means
#           algorithm should be used, if none specified, the
#           default k-means algorithm will be used (@see kmeans)
# @param verbose: specifies the algorithm verbosity during it's
#           execution (runtime implementation specific
#           parameter)
#
# @return model
#
frbf <- function(data_matrix, number_clusters, class_name,
         weighting_function = FUNCTION_REMORA_EUCLIDEAN,
         scale_variance=TRUE, s_value = 0.2, d = 0.23,
         epsilon = 0.01, niter=-1, niter_changes=5,
         perform_sum = TRUE, clustering_algorithm = '',
         verbose=VERBOSE_NO) {
  # train
  if (verbose.show(verbose)) {
    cat('Model phase...\n')
    flush.console()
  }
  config <- remoraConfiguration(number_clusters, class_name,
                weighting_function, scale_variance, s_value, d,
                epsilon, niter, niter_changes, perform_sum,
                clustering_algorithm, verbose)
```

```
  model <- remora.model(data_matrix, config)
  model
}
```

B.3 Get PCA

The `getPCA` function performs the spectral decomposition using the `prcomp` function. This allows an easy way to get the eigenvectors and the eigenvalues.

```
#
# Finds the Principal Components recurring to PCA.
#
# @param kernels is the kernels
# @param config is the configuration
# @return principal component analysis
#
getPCA <- function(kernels, config) {
  if (verbose.showDetail(config@verbose)) {
    cat('\tPerforming PCAs...\n')
    flush.console()
  }

  pca_result <- list()

  # iterates for clusters/kernels
  for (cluster_name in names(kernels)) {

    # get cluster specific information
    k <- kernels[[cluster_name]]
    points_per_cluster <- k@points_per_cluster
    for (cluster_id in c(1:length(points_per_cluster))) {
      cluster_points <- points_per_cluster[[cluster_id]]

      pca <- prcomp(cluster_points[,
                        c(2:length(cluster_points))],
                    scale. = config@scale_variance)
      stddev  <- pca$sdev
      rotation <- pca$rotation
      k@eigen_values[cluster_id] <- list(stddev)
      k@eigen_vector[cluster_id] <- list(rotation)
    }
    pca_result[cluster_name] <- k
  }

  pca_result
}
```

B.4 Predict

The `predict.RemoraModel` funcion overloads the `predict` function to support the trained model contained in the `RemoraModel` class, defined in Appendix A.2 and built by the `frbf` function of Appendix B.2.

This function is the FRBF classification stage entry point for the user. It receives the model and classifies, *i.e.* predicts, to which class each of the given data points belong to.

```
#
# Remora predict function.
#
# @param object: remora model, obtained from
#               the learning procedure
# @param data_matrix: the data to classify
# @return prediction
#
# @see frbf
#
predict.RemoraModel <- function(object, data_matrix, ...) {
  model <- object

  # classify
  if (verbose.show(model@config@verbose)) {
    cat('Classification phase...\n')
    flush.console()
  }
  data_matrix <- as.matrix(getUnclassedMatrix(data_matrix,
                    model@config@class_name))
  if (verbose.showDebug(model@config@verbose)) {
    classification <- remora.predict(model, data_matrix)
  } else {
    classification <- remora.classify(model, data_matrix)
  }
  classification
}
```

Appendix C

Tests

The FRBF implementation in R has been subjected to several tests. The tests have been automated through R scripting, thus allowing its execution and validation without human assistance. The following function exemplifies one of the scripted tests. When called, regardless of argument set or not, it will make use of the *wdbc* data set and iterate through all the kernel functions available, all the K-Means algorithms available and will still test the PCA scaling factor and the sum, or not, of the centroids per class. Overall, it will perform 144 tests. During its execution, the configuration used, the accuracy and the time spent for training and classification procedures will be printed.

```
#
# Black box test.
# Uses the WDBC data and iterates over
#  all the kernel functions,
#  all the K-Means algorithms,
#  the PCA scaling factor true and false values, and
#  the perform sum true and false values.
#
# @param clusters: the number of clusters to use
# @result returns the average accuracy for all iterations
#
test_frbf_wdbc <- function(clusters = 4) {
 cat('\nPerforming full tests on frbf using wdbc.')

 test_data_file <- "wdbc.csv"
 classify_data_file <- "wdbc-for-classification.csv"

 cat("\n\tUsing test data",test_data_file)
 mtrx <- read.csv(file=paste(
                 'd:/mestrado/thesis/R/datasets/wdbc/',
                 test_data_file, sep=""), header=TRUE)
 mtrx <- getUnclassedMatrix(mtrx, "ID")
 cat("\n\tUsing classification data",classify_data_file)
 classify <- read.csv(file=paste(
                 'd:/mestrado/thesis/R/datasets/wdbc/',
                 classify_data_file, sep=""), header=TRUE)
 classify <- getUnclassedMatrix(classify, "ID")
```

```
matrix_class_name <- "Diagnosis"
total_accuracy <- 0
iterations <- 0
cat('\n\nBase configuration:',clusters,
    'clusters, no verbosity.\n')
for (fn in FUNCTIONS_REMORA) {
 for (ca in c("Hartigan-Wong", "Lloyd", "Forgy", "MacQueen")) {
  for (sv in c(TRUE, FALSE)) {
   for (psum in c(TRUE, FALSE)) {

    iterations <- iterations + 1
    cat('\nConfiguration [')
    cat(iterations)
    cat(']: function is ')
    cat(fn)
    cat(', algorithm is ')
    cat(ca)
    cat(', scale variance is ')
    cat(sv)
    cat(', perform sum is ')
    cat(psum)
    cat('. ')
    flush.console()

    time_startm <- unclass(Sys.time())
    model <- frbf(mtrx, weighting_function=fn,
                  clustering_algorithm = ca,
                  niter = nrow(mtrx)/12,
                  class_name=matrix_class_name,
                  number_clusters = clusters,
                  scale_variance = sv,
                  perform_sum = psum,
                  verbose=VERBOSE_NO)
    time_endm <- unclass(Sys.time())

    # TRAIN ACCURACY
    train_prediction <- predict(model, mtrx)

    # CLASSIFICATION
    time_startp <- unclass(Sys.time())
    classification <- predict(model, classify)
    time_endp <- unclass(Sys.time())

    accuracy <- getAccuracy(classify, matrix_class_name,
                classification)
    total_accuracy <- total_accuracy + accuracy

    cat('\nAccuracy [')
    cat(iterations)
```

```
      cat(']: ')
      cat(accuracy)
      cat('\nTrain ')
      showAccuracy(mtrx, matrix_class_name, train_prediction,
                   FALSE)
      cat('\nModel [')
      cat(iterations)
      cat(']: ')
      cat((time_endm - time_startm)/60)
      cat(' minutes.')
      cat('\nPrediction [')
      cat(iterations)
      cat(']: ')
      cat((time_endp - time_startp)/60)
      cat(' minutes.\n')
      flush.console()
    }
   }
  }
 }

 (total_accuracy / iterations)
}
```

Appendix D

Documentation

The documentation of FRBF is provided through R documentation help files, which are LaTeX files in its essence, that are packed in the distribution package.

Once the distribution package has been installed, they can be invoked using the `help` command, or its shortcut `?<function>`. Thus, executing the help command in the R console for a FRBF package item, will bring up the correspondent help documentation. The following example is from a development version of the `predict.RemoraModel` function.

```
\name{predict.RemoraModel}
\alias{predict.RemoraModel}
\title{ Predict Classification }
\description{
The predict.RemoraModel funcion overloads the predict function
to support the trained model contained in the
\code{\link[frbf:RemoraModel-class]{RemoraModel}} class.
This function receives the model, a data matrix to classify, and
classifies, i.e. predicts, to which class each of the given data
points belong to.
}
\usage{
predict.RemoraModel(object, data_matrix, ...)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{object}{ the
                \code{\link[frbf:RemoraModel-class]{model}},
                obtained from the learning procedure
                (see \code{\link[frbf:frbf]{frbf}}) }
  \item{data_matrix}{ the data to classify }
  \item{...}{ additional arguments affecting the predictions
              produced }
}
\details{
  The \code{data_matrix} can be a \code{matrix} or
  \code{data.frame}.
  It can have the class column if it has the same name, or
  index, as the training matrix used in
```

```
  \code{\link[frbf:frbf]{frbf}}).
  In such case it will be automatically ignored, otherwise the
  class column cannot be present in the data set.
}
\value{
  The result is a prediction list containing the class name
  of each data point. The position of the data point in result
  is the same as the position in the matrix given for
  classification.
}
\references{Andre O. Falcao, Thibault Langlois and
 Andreas Wichert (2006) \emph{Flexible kernels for RBF
 networks}. Jornal of Neurocomputing, volume 69, pp 2356-2359.
 Elsevier. }
\author{ Fernando Martins and Andre Falcao }
%\note{ }
\seealso{ \code{\link[frbf:frbf]{frbf}}
\code{\link[frbf:RemoraModel-class]{RemoraModel}}
}
\examples{
# infert data is composed by 248 points and will be split
data(infert)
# the training matrix will be use the first 100 points
training_matrix <- infert[c(1:100) ,]
# the matrix to classify will use all the other points
classification_matrix <- infert[c(101:248) ,]

# create the model
model <- frbf(training_matrix, class_name = "education",
              number_clusters = 10, scale_variance = FALSE)

# predict
classification <- predict(model, classification_matrix)

# the classification points for the last
print(classification)
}
% Add one or more standard keywords, see file 'KEYWORDS' in the
% R documentation directory.
\keyword{ classif }% __ONLY ONE__ keyword per line
```

Appendix E

Packaging

The implementation of the FRBF algorithm required a set of scripts in order to build it automatically. The following R scripts are used to pack the algorithm.

E.1 R Packaging Script

The packaging procedure is actually quite simple. All that is required to create the packaging structure is to call the `package.skeleton` function with the `frbf` as the package name and the file list that contains the FRBF code implementation. This is precisely what the following R script does. This script, saved on a file named `0_pack_remora.r`, is used by the shell script shown in Appendix E.2

```
#
# Packaging Script
#

#
# Version 1, September 2009
# Fernando Martins
# fmp.martins@gmail.com
# http://www.vilma-fernando.net/fernando
#

cat('Packing Remora...\n')

file_lst <- character(5)
file_lst[1] <- '/home/fmm/thesis/R/src/1_classes.r'
file_lst[2] <- '/home/fmm/thesis/R/src/2_common.r'
file_lst[3] <- '/home/fmm/thesis/R/src/3_model.r'
file_lst[4] <- '/home/fmm/thesis/R/src/4_predict.r'
file_lst[5] <- '/home/fmm/thesis/R/src/5_main.r'

package.skeleton(name = "frbf", force = TRUE,
                 namespace = TRUE, code_files = file_lst)

cat('\nDone.\n')
```

E.2 Shell Packaging Script

To automate the entire packaging procedure, the following shell script was created. It guarantees that there is no previous packaging files nor directories, runs the R packaging script from Appendix E.1, complements the packaging structure with the specific FRBF documentation files from Appendix D, validates the package and, finally, builds a `.tar.gz` file ready for CRAN submission. CRAN will then validated it and, if approved, compile it to all the available systems, making it available for distribution.

```
#!/bin/sh
rm frbf_*.tar.gz
rm -Rf frbf/
rm -Rf frbf.Rcheck/
R -f 0_pack_remora.r
rm -Rf frbf/man/*.Rd
rm frbf/Read-and-delete-me
cp ../pack_files/* frbf/.
cp ../pack_files/man/* frbf/man/.
R CMD check frbf
R CMD build frbf
```

Abreviations

ANSI	American National Standards Institute
API	Application Programming Interface
APL	A Programming Language
ASCII	American Standard Code for Information Interchange
BOP	Bayes Optimal Classifier
CPU	Central Processor Unit
CRAN	Comprehensive R Archive Network
CSV	Comma Separated Values
EM	Expectation Maximization algorithm
FAQ	Frequently Asked Questions
FRBF	Flexible RBF Network
GB	Giga Byte
GHz	Giga Hertz
GNU	GNU's Not Unix
GUI	Graphical User Interface
IDE	Integrated Development Environment
LGPL	Lesser General Public License
MB	Mega Byte
NN	Neural Network
OOP	Object Oriented Paradigm
PCA	Principal Component Analysis
RAM	Random Access Memory
RBF	Radial Basis Function
SVD	Standard Value Decomposition
SVN	Subversion

Bibliography

[1] S. Albrecht, J. Busch, M. Kloppenburg, F. Metze, and P. Tavan. Generalized radial basis function networks for classification and novelty detection: self-organization of optimal bayesian decision. *Neural Networks*, 13(10):1075–1093, 2000.

[2] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The new S language: a programming environment for data analysis and graphics*. Wadsworth and Brooks/Cole Advanced Books & Software, Monterey, CA, USA, 1988.

[3] Chris Bishop. Improving the generalization properties of radial basis function neural networks. *Neural Computation*, 3(4):579–588, 1991.

[4] M. C. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, UK, 1995.

[5] D. Broomhead and D. Lowe. Multivariate functional interpolation and adaptative networks. *Complex Systems*, 2:321–355, 1988.

[6] Martin D. Buhmann. *Radial Basis Functions: Theory and Implementations*. Cambridge University Press, 2003.

[7] John M. Chambers. *Statistical Models in S*. CRC Press, Inc., Boca Raton, FL, USA, 1991.

[8] John M. Chambers. *Programming with Data: A Guide to the S Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[9] CollabNet and Tigris. Subversion. http://subversion.tigris.org/, 2009.

[10] Microsoft Corp. Windows xp. http://www.microsoft.com/windows/windows-xp/, 2009.

[11] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

[12] Ingo H. de Boer. Winshell. http://www.winshell.org/, 2009.

[13] Arthur Dempster, Nan Laird, and Donald Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[14] André O. Falcão, Thibault Langlois, and Andreas Wichert. Flexible kernels for RBF networks. *Neurocomputing*, 69:2356–2359, 2006.

[15] J. Faria, P. Grosjean, and E Jelihovschi. Tinn-R - GUI/Editor for R language and environment statistical computing. http://sourceforge.net/projects/tinn-r, 2008.

[16] Robert Gentleman. An introduction to the R package mechanism, 2002.

[17] Robert Gentleman. R Functions: Writing, Using and Documenting, 2002.

[18] Virgilio Gómez-Rubio. Introduction to the R packaging system, 2008.

[19] Paul Halmos. What does the spectral theorem says? *Americal Mathematical Monthly*, 70(3):241–247, 1963.

[20] Jiawei Han and Micheline Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.

[21] David Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. The MIT Press, 2001.

[22] Eric Hartman, James D. Keeler, and Jacek M. Kowalski. Layered neural networks with gaussian hidden units as universal approximations. *Neural Computation*, 2(2):210–215, 1990.

[23] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, 2001.

[24] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, Upper Saddle River, NJ, 1999.

[25] Kurt Hornik. The R FAQ, 2009.

[26] Young-Sup Hwang and Sung-Yang Bang. An efficient method to construct a radial basis function neural network classifier. *Neural Networks*, 10(9):1495–1503, 1997.

[27] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[28] Software in the Public Interest Inc. Debian. http://www.debian.org/, 2009.

[29] Norbert Jankowski. Approximation and classification with rbf-type neural networks using flexible local and semi-local transfer functions. *4th Conference on Neural Networks and Their Applications*, pages 77–82, 1999.

[30] Helmut Kopka and Patric Daly. *A guide to LaTeX: Document preparation for begginers and advanced users*. Addison-Wesley Professional, 1999.

[31] Helmut Kopka and Patric Daly. *Guide to LaTeX*. Addison-Wesley Professional, 2003.

[32] Leslie Lamport. *LaTeX: A document preparation system*. Addison-Wesley Professional, 1994.

[33] Canonical Ltd. Kubuntu. http://www.kubuntu.org/, 2009.

[34] James MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[35] K. Mardia, J. Kent, and J. Bibby. *Multivariate Analysis*. Academic Press, 2000.

[36] Tom M. Michell. *Machine Learning*. McGraw-Hill, 1997.

[37] Duncan Murdoch. Parsing Rd files, 2009.

[38] Fionn Murtagh and André Heck. *Multivariate Data Analysis*. Kluwer Academic, 1987.

[39] openDesktop.org. Ksvn. http://sourceforge.net/projects/ksvn/, 2009.

[40] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3(2):246–257, 1991.

[41] J. D. Powell. *Radial basis functions for multivariable interpolation: a review*. Clarendon Press, New York, NY, USA, 1987.

[42] J. D. Powell. Radial basis function approximations to polynomials. *Numerical analysis 1987*, pages 223–241, 1988.

[43] RoDuSa. JGR - Java GUI for R. http://jgr.markushelbig.org/, 2009.

[44] Peter Rossi. Making R packages under Windows: A tutorial, 2006.

[45] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 2003.

[46] Christian Schenk. Miktex. http://miktex.org/, 2009.

[47] D. Shi, D. Yeung, and J. Gao. Sensitivity analysis applied to the construction of radial basis function networks. *Neural Networks*, 7(18):951–957, 2005.

[48] Artifex Software. Ghostscript. http://pages.cs.wisc.edu/˜ghost/, 2009.

[49] Kate Development Team and Others. Kate. http://kate-editor.org/, 2009.

[50] R Development Core Team. R: A language and environment for statistical computing, 2009.

[51] R Development Core Team. R Internals, 2009.

[52] R Development Core Team. R Language Definition, 2009.

[53] R Development Core Team. Writing R Extensions, 2009.

[54] The Inkscape Team. Inkscape. http://www.inkscape.org/, 2009.

[55] Tigris. Tortoisesvn. http://tortoisesvn.tigris.org/svn/tortoisesvn/, 2009.

[56] Luis Torgo. *A Linguagem R: Programação para Análise*. Escolar Editora, 2009.

[57] Wikipedia. Gaussian functions. http://en.wikipedia.com/wiki/Gaussian_function, 2009.

[58] Ian Witten and Frank Eibe. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[59] Paul V. Yee and Simon Haykin. *Regularized Radial Basis Function Networks: Theory and Applications*. John Wiley, 2001.

[60] Zarita Zainuddin and Ong Pauline. Function approximation using artificial neural networks. *WSEAS Trans. Math.*, 7(6):333–338, 2008.

Index