UNIVERSIDAD DE GRANADA

MODELOS AVANZADOS DE INTELIGENCIA COMPUTACIONAL PARA APROXIMACIÓN FUNCIONAL Y PREDICCIÓN DE SERIES TEMPORALES EN ARQUITECTURAS PARALELAS

TESIS DOCTORAL

Ginés Rubio Flores

2010

Departamento de Arquitectura y Tecnología de Computadores

Editor: Editorial de la Universidad de Granada
Autor: Ginés Rubio Flores
D.L.: GR 3525-2010
ISBN: 978-84-693-5225-0

UNIVERSIDAD DE GRANADA

MODELOS AVANZADOS DE INTELIGENCIA COMPUTACIONAL PARA APROXIMACIÓN FUNCIONAL Y PREDICCIÓN DE SERIES TEMPORALES EN ARQUITECTURAS PARALELAS

Memoria presentada por

Ginés Rubio Flores

Para optar al grado de

DOCTOR EUROPEO EN INGENIERÍA INFORMÁTICA

Fdo. Ginés Rubio Flores

D. Héctor Emilio Pomares Cintas, **D. Ignacio Rojas Ruiz**, Profesores Titulares de Universidad y **D. Alberto Guillén Perales** Ayudante Doctor del Departamento de Arquitectura y Tecnología de Computadores

CERTIFICAN

Que la memoria titulada: **"Modelos avanzados de inteligencia computacional para aproximación funcional y predicción de series temporales en arquitecturas paralelas"** ha sido realizada por **D. Ginés Rubio Flores** bajo nuestra dirección en el Departamento de Arquitectura y Tecnología de Computadores de la Universidad de Granada para optar al grado de Doctor Europeo en Ingeniería Informática.

Granada, a DÍA de MES de 2010

Fdo. Héctor Pomares Cintas	Ignacio Rojas Ruiz	Alberto Guillén Perales
Director de la Tesis	Director de la Tesis	Director de la Tesis

AGRADECIMIENTOS

Quisiera expresar mi agradecimiento a los directores de este trabajo, por haber sido mis guías en éste el complicado mundo de la investigación.

A Luis Javier Herrera, por el apoyo y ánimos prestados así como por la orientación y estupendos consejos.

A Manuel Rodríguez Álvarez y Francisco Illeras García por ser tan serviciales, eficaces y eficientes en su trabajo.

Y en general, a todos mis compañeros del grupo de investigación y del departamento.

A los miembros del grupo de CAFPE de la Universidad de Granada, especialmente a Antonio Bueno y a Julio Lozano.

Al personal del Edinburgh Parallel Computing Centre (Chris, Cathy, Mario, etc.) por su hospitalidad. A Ben Paechter y su grupo de investigación por su acogida durante mi estancia en Edimburgo.

A mis amigos por compartir los momentos de diversión y saber escuchar mis problemas.

Finalmente, también quiero agradecer a mis padres (sin los cuales sin duda esta tesis no hubiese sido posible) y a mis hermanos el apoyo.

La realización de este trabajo no habría sido posible sin la Beca de Formación de Personal Investigador del Ministerio de Educación y Ciencia que disfruté durante los años 2005-2009, y sin la subvención económica del proyectos del Ministerio de Ciencia y Tecnología TIN2004-01419.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ALGORITMOS

LISTA DE ACRÓNIMOS

ARMA modelo Autoregresivo de Medias Móviles, *AutoRegressive Moving Average* model

ANN Red de Neuronas Artificiales, *Artificial Neural Networks*

MLP Perceptrón Multicapa, *MultiLayer Perceptron*

RBF Función de Base Radial, *Radial Basis Function*

RBFNN Red Neuronal de Funciones de Base Radial, *Radial Basis Function Neural Network*

GA Algoritmo Genético, *Genetic Algorithm*

GP Programación Genética, *Genetic Programming*

SVM Máquina de Vectores de Soporte, *Support Vector Machine*

SVC Vectores de Soporte para Clasificación, *Support Vector Classification*

SVR Vectores de Soporte para Regresión, *Support Vector Regression*

LSSVM Máquinas de Vectores de Soporte de Mínimos Cuadrados, *Least Square Support Vector Machines*

KWKNN K-Vecinos más Cercanos con distancia basada en *Kernel*, *Kernel based distance Weighted K-Nearest Neighbours*

PKWKNN KWKNN Paralelo, *Parallel KWKNN*

HPC Computación de Altas Prestaciones, *High Performance Computing*

OpenMP Multiprocesamiento Abierto, *Open Multi-Processing*

MPI Interfaz de Paso de Mensajes, *Message Passing Interface*

GPU Unidad de Procesamiento Gráfico, *Graphics Processing Unit*

CUDA Arquitectura Unificada de Dispositivo de Cómputo, *Compute Unified Device Architecture*

MSE Error Cuadrático Medio, *Mean Square Error*

RMSE Raíz Cuadrada del Error Cuadrático Medio, *Root Mean Square Error*

NRMSE Raíz Cuadrada Normalizada del Error Cuadrático Medio, *Normalized Root Mean Square Error*

1. INTRODUCCIÓN (INTRODUCTION)

En este capítulo se diserta acerca del problema de la predicción de series temporales y la aplicación de los denominados métodos inteligentes para aproximación funcional, centrándose en las familias de las Redes Neuronales Artificiales (ANN) y los métodos *kernel*. Las ANN están inspiradas en las redes neuronales biológicas, mientras que los métodos *kernels* pertenecen a la rama de Aprendizaje de Máquinas de la Inteligencia Artificial. A pesar de sus distintos orígenes, ANN y métodos *kernels* están muy relacionadas, y en este capítulo se hablará de dicha relación.

This chapter discusses the time series prediction problem and the application of the so-called intelligent methods for functional approximation, focusing in Artificial Neural Networks (ANN) paradigm and Kernel Methods. ANN are inspired by biological neural networks, whereas Kernel methods belong to Machine Learning branch of Artificial Intelligence Methods. Despite their different origins, ANN and Kernel Methods are very related, discussing in this Chapter this relation.

1.1. Aproximación funcional y predicción de series temporales

La aproximación funcional es un tipo de aprendizaje supervisado en el que se pretende crear un modelo para una función $f(x)$ desconocida de la cual tenemos una serie de muestras $(x_1, y_1), \cdots, (x_N, y_N)$ de la misma. El modelado de series temporales es un tipo de aproximación funcional en la que la función a modelar varía en el tiempo $y = f(t)$. De esta función tenemos en la versión más simple del problema (sin variables exógenas) unas muestras $y_1, \cdots, y_t, \cdots, y_N$ y el objetivo es predecir futuros valores de la función. La predicción de series temporales es un campo de gran importancia científica y económica, pero de una gran dificultad, que aumenta conforme la predicción es a mayor distancia en el futuro.

La teoría de modelos lineales para predicción de series temporales está bien especificadas [Box and Jenkins, 1976], pero su campo de aplicación está limitado a series temporales con dependencias lineales entre las muestras o aquellas series que puedan ser transformadas para cumplir con dicho requisito. La inmensa mayoría de las series temporales en la práctica (industriales, económicas, naturales, etc) son altamente no lineales, y para modelarlas se recurren a modelos de aproximación funcional generales como Redes Neuronales Artificiales [Balkin and Ord, 2000; Coulibaly et al., 2001; Teräsvirta et al., 2006; Alves da Silva et al., 2008; Shiblee et al., 2009] y en particular de éstas las Funciones de Base Radial [Rojas et al., 2000d; Rojas et al., 2000a], métodos de Aprendizaje de Máquina [Van Gestel et al., 2001; Xu, 2005; Zhou et al., 2008; Misra et al., 2009; Sapankevych and Sankar, 2009] e hibridaciones de estas y otras técnicas [Jang, 1993; Hsieh and Tang, 1998; Jursa and Rohrig, 2008; Zhang, 2003]. En todo caso, hay que empezar analizando la serie para deducir cómo se relacionan entre sí las muestras, de forma que se pueda obtener a partir de las muestras $y_1, \cdots, y_t, \cdots, y_N$ una serie de vectores de entrada/salida que puedan ser utilizados por las técnicas de aproximación funcional convencionales, definiendo el valor futuro que se desea predecir: el horizonte de predicción. Por ejemplo, si se desea conocer el valor de la series para un muestra en el futuro, y se deduce que hay una fuerte dependencia del valor de la serie en t con su valor 2 muestras antes se obtendría el conjunto de vectores $(y_1, y_3), (y_2, y_4), (y_3, y_5), \cdots, (y_{t-2}, y_t), \cdots, (y_{N-2}, y_N)$. Con los datos ya listos para ser utilizados por un método de aproximación funcional, se entrena el modelo como con cualquier otro problema de aproximación funcional. Donde varía la aplicación del método es a la hora de la predicción, si se desea un modelo para los 50 futuros valores de una serie, existen 2 alternativas básicas:

1. Predicción directa: crear un modelo para cada horizonte de predicción, lo cual supone en este caso entrenar 50 modelos.

2. Predicción recursiva: crear un modelo para cada el horizonte de predicción $t+1$, e ir aplicando recursivamente el modelo para predecir los 50 requeridos.

Ambas alternativas tienen sus problemas. En el caso de la predicción directa el coste computacional se dispara con el número de horizontes que se desee predecir y no se obtiene realmente un modelo para la función. En cambio con la predicción recursiva el problema es la propagación y acumulación del error en la predicción conforme se avanza en el tiempo, ya que se utilizan las predicciones del modelo para un tiempo como entrada para predecir las siguientes. Para predicción a largo plazo, se suele utilizar más la predicción recursiva.

Dejando de lado el método de predicción, la mayor parte del problema de modelado se centra en encontrar los valores anteriores que deben de usarse para crear los vectores de entrada salida: los regresores. En el caso de las series lineales, aunque no se creen vectores de entrada/salida, hay un método estadístico establecido de análisis que guía la selección de los mismos. Pero para series no lineales, no hay una métodología claramente vencedora, aunque existen criterios que se pueden aplicar para elegirlos, como la estimación de la Información Mutua [Kraskov et al., 2004; B.V. Bonnlander and A.S. Weigend, 2004; Ji et al., 2005] o la varianza del error de salida [Sorjamaa et al., 2005; Lendasse et al., 2006; Jones et al., 2007]. Unas características muy importantes de las series temporales son su estacionaridad (es decir, el que sus parámetros estadísticos, especialmente media y varianza, no dependan del tiempo) y estacionalidad (es decir, que la serie varíe regularmente en el tiempo), que se pueden intentar hallar de varias maneras: inspección visual, métodos estadísticos, etc. (ver [Dutilleul, 2001], [Vlachos et al., 2005] y [Ahdesmäki et al., 2005] para algunos ejemplos de la literatura sobre el tema).

1.2. Los métodos inteligentes para regresión

Si nos vamos a lo fundamental de un sistema inteligente que aprende una función a partir de muestras de la misma, se concluye que hay una serie de suposiciones en que se basan el funcionamiento de todos:

1. Un problema viene descrito por un conjunto de datos separado en entradas y salidas. Las entradas son una descripción suficiente y veraz del punto concreto del espacio de entrada del cual dar una medida (esto último no es normalmente garantizable, y da lugar al importantísimo problema de la selección de características). Es muy común suponer que los datos disponibles son independientes y están idénticamente distribuidos en el espacio de entrada (estas suposiciones no son válidas en el caso de las series temporales).

2. Dadas 2 vectores de entrada, cuanto más similares son más similares serán sus salidas.

3. Los datos del problema pueden contener errores, tanto en las entradas como en las salidas, pero siempre habrá mayor proporción de datos sin error o con

un error acotado (es decir, hay pocos outliers). Para aproximación funcional, el concepto de error en la salida no es discreto (es decir, no hay datos "erróneos" sino con mayor o menor error), y se suele suponer que el error en la salida es una función aditiva que sigue una distribución gaussiana de media 0 y con una determinada varianza σ_e^2, ya que de todas formas se puede normalizar adecuadamente los datos para ello.

Claramente, el concepto clave es que se debe de poder dar una medida de similitud entre los puntos de entrada. Los problemas de clasificación y regresión (o aproximación funcional) se pueden ver cada cual como caso particular del otro, pero se han mantenido separado. Un motivo básico es la diferencia en las medidas de error de los modelos para ambos tipos: en los problemas de clasificación las medidas de error son discretas (suele utilizarse el porcentaje de acierto de clasificación de ejemplos) mientras que en los de regresión en cambio son funciones de rango continuo. Otra gran diferencia es el concepto de distancia entre las salidas, es decir, la clase 1 no tiene por que estar a priori más cercana a la clase 2 o la 3, pero en cambio cláramente en regresión se prefiere una predicción de 2 si el valor real es 1. Ambos factores además afectan decisivamente el concepto o medida de generalización de los modelos, que es muy diferente según el tipo de problema.

Los métodos inteligentes están basados en analogías de procesos del mundo real, inspirados en procesos biológicos, como las Redes Neuronales Artificiales [Haykin, 1994], Algoritmos Genéticos [De Jong, 1985; Michalewicz et al., 1990], algoritmos basado en Sistema Inmune [Farmer et al., 1986; Hunt and Cooke, 1996], etc. Se han logrado grandes éxitos en este campo, especialmente con las Redes Neuronales Artificiales. Los 2 ejemplos importantes de las mismas son los modelos de Perceptrón Multicapa (MLP) y las Redes de Funciones de Base Radial (RBFNN). Estos modelos así como sus métodos y algoritmos asociados de aprendizaje, se han desarrollado sobre todo en base a prueba y error, deduciendo las bases formales a posteriori. En cambio, el Aprendizaje de Máquina comprende métodos que se basan en teoría matemática. Como ejemplos representativos tenemos los métodos *kernels*, y dentro de ellos las Máquinas de Vectores de Soporte (SVM). La sólida base teórica de estos métodos, y el enorme éxito de las SVM para clasificación han convertido a los métodos *kernels* en un tema de interés para la investigación. Algunos autores, como [Neal, 1996], han proclamado incluso el fin de las redes neuronales al demostrarse ser estos métodos unas generalizaciones de las mismas MLP y RBFNN. Ahora bien, es una declaración exagerada que no ha venido avalada por la práctica en el caso de los métodos *kernels* aplicados a regresión. Así, los modelos de más éxito son diferentes en cada caso:

1. Para problemas de clasificación las ANN han obtenido excelentes resultados con diversos modelos y particularmente tanto con MLP como con RBFNN.

Pero actualmente los modelos estado del arte para clasificación son las SVM (de Aprendizaje de Máquina), ya que superan con creces el rendimiento de los demás modelos.

2. Para problemas de regresión las RBFNN son ampliamente aplicadas y que se han estudiado tratando de mejorarlas desde hace años. En Aprendizaje de Máquina, las técnicas de Regresión con Procesos Gaussianos y Máquinas de Vectores de Soporte de Mínimos Cuadrados, *Least Square Support Vector Machines* (LSSVM), una variante de SVM, no acaban de imponerse a pesar de sus excelentes precisiones.

1.3. Redes Neuronales Artificiales vs métodos kernel

En las Redes Neuronales Artificiales, las neuronas funcionan como unidades que miden la similitud entre las entradas y calculan una función de salida. Centrándonos en los métodos *kernels*, la función de *kernel* es una función de similitud, y la salida para un punto es calculada por combinación de las salidas de los puntos de entrenamiento, ponderada por la similitud del punto de entrada a los puntos de entrenamiento y, normalmente, un factor de importancia asociado a cada uno (ver los multiplicadores de Lagrange en la formulación de SVM y LSSVM). Esto lleva a algo bien conocido: si se observa un MLP y una RBFNN con un centro por punto de entrenamiento y un SVM con *kernel* lineal y un LSSVM con *kernel* Gaussiano, los modelos son matemáticamente idénticos (ver formulación en el capítulo 2). En el caso de RBFNN, este límite de un centro por punto de entrenamiento es equivalente, además, a usar infinitas neuronas, lo que llevó a la demostración de [Neal, 1996] y su afirmación de que las ANN estaban superadas.

Ahora bien, no se afirma aquí que el trabajo con métodos *kernels* es redundante, ni mucho menos. En el caso de modelos SVM para clasificación (SVC), sus características los hacen ahora mismo los mejores modelos existentes. Son los mejores por rendimiento (computacional, precisión y generalización), tener un algoritmo de aprendizaje bien fundamentado e independiente del *kernel* y especialmente (por lo que es el talón de Aquiles de los métodos *kernels* para regresión): los modelos resultantes son dispersos, es decir, tras el entrenamiento se descarta parte de los puntos de entrada. Si observamos las SVM para clasificación como unas ANN, cosa que hemos visto que podemos, resulta que tenemos redes de una capa de neuronas cuyo entrenamiento obtiene un máximo compromiso entre generalización y precisión (controlado por un factor de regularización) y además con el mínimo número de neuronas posible para cualquier tipo de función de activación de neurona (*kernel*). Los algoritmos de aprendizaje de ANN para clasificación muchas veces son específicos a la función de transferencia, y la obtención del mínimo número de neuronas a la vez de conservar resultados óptimos es algo que no está garantizado, además de ser muy pesado computacionalmente [Chen et al.,

1992; Musavi et al., 1992; Branke, 1995; Coulibaly et al., 2000; Guillén et al., 2005]. Es decir, ahora mismo, las SVC han superado completamente las ANN de una sóla capa con función de transferencia común para todas las neuronas y neurona de salida con combinación lineal.

Ya se ha nombrado que los métodos *kernels* para regresión más comunes (SVM para regresión (SVR), LSSVM y Procesos Gaussianos) tienen una grave problema: la falta de dispersión de los modelos, es decir, todo modelo incluye todos los puntos de entrenamiento. Esto se ha tratado de remediar extendiendo el algoritmo de aprendizaje con métodos de poda tanto dentro como a posteriori del aprendizaje, pero, desgraciadamente, esto añade un gran coste computacional al aprendizaje. Esto es especialmente grave, y lleva al otro problema común de todos los métodos *kernels*: el crecimiento del coste de aprendizaje con el número de datos de entrenamiento.

Otro grave problema es que, salvo en caso de Procesos Gaussianos, los algoritmos de aprendizaje están definidos para obtener parámetros lineales (LSSVM) o de un problema convexo (SVM) de los modelos, suponiendo fijados los parámetros no lineales de los mismos (usualmente, factor de regularización y parámetros del *kernel*). Si hay que optimizar los parámetros no lineales, el entrenamiento de un modelo se vuelve mucho más complejo y costoso, implicando un problema de optimización global (en [Suykens et al., 2002], se ve como se puede abordar por niveles la optimización, pero sigue siendo muy costoso). Los modelos de clasificación suelen ser mucho menos sensibles a los parámetros no lineales que los de regresión, lo que supone otro problema añadido a los métodos *kernels* para regresión.

En resumen, aunque para clasificación los métodos *kernels* tienen un modelo estado del arte no superado hasta el momento, SVC, para regresión en cambio no acaban de imponerse porque, si nos centramos en el caso más común, es decir un modelo SVR o LSSVM con *kernel* Gaussiano o RBF, son equivalentes a las RBFNN, especialmente si a posteriori se aplica la poda [de Kruif and de Vries, 2003; Zeng and wen Chen, 2005]. Ahora bien, hay un elemento de ventaja para los mismos: las excelentes precisiones obtenidas sin duda y el factor de regularización que adecuadamente fijado garantiza buena propiedades de generalización.

Como último apunte, hay que decir que aunque los métodos *kernels* son generalizaciones de algunas ANN, las ANN en sí son mucho más generales que los métodos *kernels*. La definición de los modelos de las ANN es más flexible al no estar supeditadas sus estructuras ninguna base formal demasiado rígida, lo que en ocasiones las hace mucho más deseables.

1.4. Estructura de la memoria

En este, el Capítulo 1, se ha realizado una breve introducción sobre los problemas de regresión con métodos inteligentes (concretamente con las redes neuronales artificiales y los métodos *kernels*).

Capítulo 2: expone los fundamentos de los temas estudiados en esta memoria.

Capítulo 3: presenta un estudio acerca del entrenamiento del método *kernel* de Máquinas de Vectores de Soporte de Mínimos Cuadrados (*Least Square Support Vector Machines*, LSSVM) para regresión centrándose en obtener métodos que mejoran de su eficiencia y la precisión del resultado.

Capítulo 4: presenta KWKNN (*Kernel Weighted K-Nearest Neighbours*), un nuevo (creado por el autor) método *kernel* para regresión, describiendo sus características y ventajas frente a otros métodos *kernels* más clásicos. También se expone una versión del KWKNN que hace uso de programación paralela para afrontar conjuntos de datos de gran tamaño, usualmente no abordables con otros métodos *kernels*, y otra implementada sobre GPU cuyo beneficio es un incremento de velocidad notable.

Capítulo 5: aborda un novedoso estudio para obtención de *kernels* específicos para series temporales mediante dos enfoques: uno basado en el estudio de las propiedades de los datos con asistencia de un experto y otro basado en Programación Genética que no requiere conocimiento previo acerca del problema que se modela.

Capítulo 6: muestra dos librerías creadas específicamente con el objetivo de proveer acceso a dos plataformas paralelas distintas (*cluster* de computadores y unidades de procesamiento gráfico) desde Matlab.

Capítulo 7: se exponen las principales conclusiones de esta memoria, así como la enumeración de las aportaciones realizadas.

Capítulo 8: se enumeran las publicaciones derivadas de este trabajo.

1.5. Functional Approximation and time series prediction

The functional approximation is a kind of supervised learning problem where the main goal is to create a model for an unknown function $f(x)$ from a set of its samples $(x_1, y_1), \cdots, (x_N, y_N)$. The time series prediction is a subtype of functional approximation where the function to be modeled depends mainly upon time $y = f(t)$. Of this function we have, in the simplest version of the problem (without exogenous variables), we have a series of samples $y_1, \cdots, y_t, \cdots, y_N$, and the objective is to predict the next values. The time series prediction has a great scientific and economic importance, but also presents lot of issues, than become more difficult larger the prediction is further in the future.

The theory of linear models for time series prediction is well specified [Box and Jenkins, 1976], but its field of application is limited to time series with linear dependencies between their samples or those series that can be transformed to fulfill this requirement. The most of the time series (in industry, economics, nature, etc) are highly non linear, and to model them general functional approximation methods are used, like Artificial Neural Networks[Balkin and Ord, 2000; Coulibaly et al., 2001; Teräsvirta et al., 2006; Alves da Silva et al., 2008; Shiblee et al., 2009] and particularly Radial Basis Functions [Rojas et al., 2000d; Rojas et al., 2000a], Machine Learning Methods [Van Gestel et al., 2001; Xu, 2005; Zhou et al., 2008; Misra et al., 2009; Sapankevych and Sankar, 2009] and hybridizations of theses and other techniques[Jang, 1993; Hsieh and Tang, 1998; Jursa and Rohrig, 2008; Zhang, 2003]. In any case, the first step is the analysis of the dependencies between samples of the series, in order to obtain from $y_1, \cdots, y_t, \cdots, y_N$ a set of input/output vectors that can be used by the conventional techniques for functional approximation, by defining the desired future value to predict: the prediction horizon. For instance, if we wish to the value of the series for a step sample in the future and it is deduced a strong dependency between a sample at t with the sample two steps ago, it will be obtained the set of vectors $(y_1, y_3), (y_2, y_4), (y_3, y_5), \cdots, (y_{t-2}, y_t), \cdots, (y_{N-2}, y_N)$. With the data conveniently formatted to be used by a conventional method for functional approximation, a model is training likes with any normal functional approximation problem. There is 2 basic alternatives to use a generated model for prediction of, for instance, 50 future values:

1. Direct Prediction: creating a model by prediction horizon, which implies training 50 models.

2. Recursive Prediction: creating a model for the horizon $t + 1$, and applies it recursively to predict the 50 values.

Both approaches have their advantages and disadvantages. In the direct prediction case the computational cost grows with the number of horizons, and a unique mo-

del for the function is not obtained. In the other hand, with the recursive prediction the issue is the propagation and accumulation of errors in prediction within time, as predictions of the model are used as inputs to predict next ones. For large scale prediction, recursive prediction is usually more used.

Without taking into account the prediction, the main difficulty of the modeling is focused on finding the previous values to be used to create the input/output vectors: the predictor variables also known as regressors. In case of linear series modeled with linear models, although no input/output vectors are created, there is an established stastical methodology for the analysis that guides the selection. For non linear series, there is no a clear choice, although there are a number of criteria that can be used, likes estimation of the Mutual Information [Kraskov et al., 2004; B.V. Bonnlander and A.S. Weigend, 2004; Ji et al., 2005] or the variance of output error [Sorjamaa et al., 2005; Lendasse et al., 2006; Jones et al., 2007]. Important characteristics of the time series are their stationarity (i.e., their statistical properties, specially their average and variance, do not depend of time) and seasonality (i.e, the regular variations of the series in time), that can be apprehended in various ways: visual inspection, stastical methods, etc. (see [Dutilleul, 2001], [Vlachos et al., 2005] and [Ahdesmäki et al., 2005] for some examples about the topic in literature).

1.6. Intelligent Methods for regression

There are a number of fundamental suppositions when learning a function from a set samples:

1. A problem is described by a data set separated in inputs and outputs. Inputs are a enough and reliable description of the given point of the input space of which a measure will be given (this is not usually guaranteed, and it is the origin of the very important problem of feature selection). It is very common to suppose that the data are independent and identically distributed in input space (this is not a valid supposition when working with time series).

2. Given 2 inputs, more similar they are, more similar are their outputs.

3. The data can have errors, both in inputs or outputs, but there are always a greater proportion of non erroneous data or data with controled error (i.e, there are few outliers). For functional approximation, the concept of erroneous output is continuous (there is not erroneous data, but with more or less error), and it is often supposed that the errors at outputs follow an additive Gaussian distribution with mean and a fixed variance.

The key concept is clearly that a similarity measure between inputs points must be defined. The classification and regression problems can be seen each one

as a particular case of the other one, but they have remained separated. A basic motivation is the difference in the error measures for models of each type: in the classification problems the error measures are discrete (the percentage of correct answers on the samples is an usual measure) meanwhile in regression problems they are continuous functions. The other important difference is the concept of distance between outputs, i.e., the class *1* have not to be closer to the classes *2* or *3*, but in regression it is certainly preferable a prediction of *2* if the real output for the input is *1*. Both factors have a decisive influence in the concept and measurement of the generalization capability of models, which are very different given the particular problem type.

Intelligent Methods are inspired in analogies of the real world, biological processes, as Artificial Neural Networks [Haykin, 1994], Genetic Algorithms [De Jong, 1985; Michalewicz et al., 1990], immune system-based algorithms [Farmer et al., 1986; Hunt and Cooke, 1996], etc. Great achievements have been obtained in this field, specially with Artificial Neural Networks. Two important examples of these are the MultiLayer Perceptron (MLP) and the Radial Basis Function Neural Networks (RBFNN). These models, as well as their learning methods and algorithms, have been developed in base of trial-and-error, being the formal basis deduced a posteriori. In the other hand, Machine Learning methods are based in mathematical theory. Kernel methods are representative examples of these ones, and particularly the Support Vector Machines (SVM). The well-founded theoretical basis of these methods and the outstanding success of SVM for classification have made of kernel methods an topic of interest for research. Some authors, as [Neal, 1996], have even proclaimed the end of Artificial Neural Networks due to the demonstration that some of these methods are generalizations of MLP and RBFNN. However, this excessive declaration has no practical supports in the case of kernel methods for regression. The most successful models are quite different for each class of problems:

1. For classification problems, ANN have obtained excellent results with number of models and particularly with MLP and RBFNN. But nowadays, the state-of-the-art for classification are SVM models, as their performances outperform the rest of models.

2. For regression problems, for years RBFNN models have been widely applied and extensively studied with the aim of improving them. From Machine Learning, the Gaussian Process and Least Square Support Vector Machines (a variant of SVM) for regression have not a such outstanding advantage despite their excellent accuracy.

1.7. Artificial Neural Networks vs. kernel methods

In Artificial Neural Networks, a neuron works as a similarity measurement unit between inputs that computes an output function. In kernel methods, a kernel function is a similarity function and, given an input, a model computes the output as a combination of the outputs of the training points, weighted by both the similarity with each training input and a relevance factor associated to each input point (i.e., the Lagrange Multiplier of SVM and LSSVM models). This is a well-known conclusion: a MLP model and RBFNN model with a center by training point have respectively an equivalent formulation of a SVM model with linear kernel and a LSSVM model with Gaussian kernel (as can be seen in Chapter 2). In RBFNN case, a center per input point is equivalent to using an infinite number of neurons, leading to the demonstration of [Neal, 1996] and its affirmation about the end of ANN.

Nevertheless, in this work it is not said that research with kernel method is redundant. obviously. SVM for classification (SVC) have features that make it nowadays the best classification technique. They have the best performances (computationally, in accuracy and generalization capabilities), a well-founded training algorithm and no dependent of a particular kernel, and, specially, their models are sparse, i.e., they do not use all the training points inside the generated models (this is one of the main drawback of most kernel methods applied to regression). From the ANN point of view, a SVC model is a neural networks which training algorithm achieves the best possible compromise between generalization and accuracy (controlled by a regularization factor), and, moreover, using the minimum number of neurons regardless their particular transfer function (i.e., the kernel). Learning algorithms for ANN are most of time *transfer function-dependent*, and do not guarantees to obtain a network with the minimum number of neurons preserving optimum performances (moreover, this implies additional high computational costs [Chen et al., 1992; Musavi et al., 1992; Branke, 1995; Coulibaly et al., 2000; Guillén et al., 2005]). Nowadays, SVC models outperform ANN models with an input layer, an hidden layer of neurons of unique transfer function and an output neuron that performs an output based in linear combination, which is one of the most used architectures in ANN.

As previously commented, the most common of kernel methods (SVM for regression (SVR), LSSVM and Gaussian Processes) have a serious problem: they generate non-sparse models, i.e., every model formulation includes all the samples of the training set. The problem as been addressed by extending the learning algorithms with pruning procedures both before or after the actual training with samples, but, unfortunately, these procedures increase the computational costs of training. This issue arises the other widespread weakness of kernel methods: the great increase of computational costs of training with the number of training data.

Summarizing, despite the fact that kernel methods have for classification at present day a non beaten state-of-the-art model, SVC, for regression, in return, they have not such a clear advantage. The reason is that SVR and LSSVM with Gaussian kernels, the most commonly used models, are equivalent to RBFNN, specially if pruning is applied [de Kruif and de Vries, 2003; Zeng and wen Chen, 2005]. Nevertheless, they present characteristics that can make them preferable: the excellent accuracy of generated models and the regularization factor, which suitably fixed can guarantees good generalization capabilities.

As last appointment it is worth to make one thing clear, whereas some of kernel methods are generalizations of some ANN, ANN models are much more general than kernel methods. ANN model definition is more flexible as it is not subject to a rigid formal basis, that can make them in some cases more desirable than models from kernel method.

1.8. Structure of the document

In the present Chapter 1, a brief introduction about the intelligent methods (specifically Artificial Neural Networks and kernel methods) for regression and their problems has been carried out.

Chapter 2: Exposes the fundamental topics studied in this document.

Chapter 3: Presents a study about the training of Least Square Support Vector Machines (LSSVM) for regression, aiming to obtain procedures to improve the efficiency of training and the accuracy of resulting models.

Chapter 4: Presents a new kernel method for regression (created by the author), *Kernel Weighted K-Nearest Neighbours* (KWKNN) and its characteristics and advantages versus some classical kernel methods. It is also presented a parallel version of KWKNN for cluster platform that can deal with large data sets, that can not be afforded with other kernel methods, as well as another implementation for Graphics Processing Units (GPU) that exhibits a great improvement in speed of computation.

Chapter 5: Addresses a new study for the creation of specific-to-problem kernels for time series using two approaches: one based in the study of the properties of the data with assistance an expert and another based in Genetic Programming, that not requires of previous knowledge about the modeled problem.

Chapter 6: Shows two libraries specifically created with the aim of providing access to two parallel computing platforms (computer clusters and graphics processing units) from Matlab.

Chapter 7: Exposes the main conclusions of this work, as well as enumerates the main contributions made.

Chapter 8: The publications derivated from this study are enumerated.

2. FUNDAMENTOS

En este capítulo se presentan las técnicas que se consideran de referencia básica para contextualizar el trabajo expuesto en la memoria sobre los métodos *kernel* en el ámbito de los métodos inteligentes para regresión y la computación de altas prestaciones. En primer lugar, se describen los modelos Autorregresivos de Media Móvil (*AutoRegressive Moving Average models*, ARMA) dado que son los modelos lineales más básicos de la literatura para modelado y predicción de series temporales, y aún ampliamente usados en la actualidad. Posteriormente, se describen las Redes Neuronales Artificiales, modelos clásicos y aún aplicados con éxito en la práctica para diversos problemas y concretamente para predicción de series temporales. Específicamente, se presentan el Perceptrón Multicapa y las Redes de Funciones de Base Radial. Seguidamente, se hace una breve introducción de los métodos *kernel* de la literatura más relevantes que se nombran en esta memoria: las Máquinas de Vectores de Soporte para Regresión (SVR o SVM para regresión), las Máquinas de Vectores de Soporte de Mínimos Cuadrados para Regresión (LSSVM) y la Regresión mediante Procesos Gaussianos. Finalmente, se habla de programación paralela y de las librerías MPI (*Message Passing Interface*) y CUDA (*Compute Unified Device Architecture*), que son usadas para aprovechar las posibilidades que proporcionan las arquitecturas de cómputo de altas prestaciones (*High Performance Computing*, HPC) utilizadas en este trabajo: cluster de computadores y procesadores gráficos (*Graphics Processing Unit*, GPU).

2.1. Modelos lineales para modelado y predicción de series temporales

Los modelos lineales han sido usados tradicionalmente para realizar predicción de series temporales y, aún considerando sus limitaciones, siguen siendo ampliamente utilizados [Gaur et al., 2005][Francq et al., 2005][Torres et al., 2005].

La teoría de modelos lineales para predicción de series temporales está bien especificada, pero su campo de aplicación está limitado a series temporales con dependencias lineales entre las muestras [Box and Jenkins, 1976]. Los modelos lineales básicos (AR, MA y ARMA) deben aplicarse a series estacionarias dado

que éstas conservan sus propiedades estadísticas constantes en el tiempo, concretamente: la media, la varianza y la covarianza.

Box y Jenkins definen un modelo estocástico lineal general como el que produce una salida con una entrada que es ruido blanco ε_t y una suma ponderada de valores anteriores ε_t. Matemáticamente puede expresarse como sigue:

$$\tilde{Y}_t = \mu + \varepsilon_t - \varphi_1\varepsilon_{t-1} - \varphi_2\varepsilon_{t-2} - \cdots - \varphi_q\varepsilon_{t-q} - \cdots \tag{2.1}$$

donde μ es la media de un proceso estacionario, y φ_t con $t = 1, 2, \cdots$, son coeficientes que satisfacen:

$$\sum_{i=1}^{\infty} \varphi_i^2 < \infty \tag{2.2}$$

donde ε_t es una variable aleatoria independiente con media cero y varianza constante σ_ε^2. Sin embargo, la Ecuación (2.1) no es práctica ya que tiene un número de términos infinito. Así, se expresa la Ecuación (2.1) en términos de un número finito de componentes Autoregresivos y/o de Medias Móviles. Si se define Y_t como $\tilde{Y}_t - \mu$, un modelo Autoregresivo de orden p, AR(p), para una serie temporal $\{Y_t\}_1^N$ tiene la forma:

$$Y_t = -\sum_{i=1}^{p} \phi_i Y_{t-i} + \varepsilon_t. \tag{2.3}$$

donde $\phi_i,\ i = 1, \ldots, p$ son los parámetros del modelo y ε_t el error en cada muestra t.

Un modelo de medias móviles de orden q, $MA(q)$, para una serie temporal $\{Y_t\}_1^N$ tiene la forma:

$$Y_t = \varepsilon_t + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i}. \tag{2.4}$$

donde $\theta_i,\ i = 1, \ldots, q$ son los parámetros del modelo y ε_t el error en cada muestra t.

Un modelo $ARMA(p,q)$ (*AutoRegressive Moving Average model*) consiste en 2 partes, una parte Autoregresiva (*AutoRegressive*, AR) de orden p y una de medias móviles (*Moving Average*, MA) de orden q :

$$Y_t = -\sum_{i=1}^{p} \phi_i Y_{t-i} + \varepsilon_t + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i}. \tag{2.5}$$

que se puede escribir cómo:

$$\Phi_p \vec{Y}_t = \Theta_q \vec{\varepsilon}_t. \tag{2.6}$$

donde $\Phi_p = (1, \phi_1, \ldots, \phi_p)$, $\vec{Y}_t = (Y_t, Y_{t-1}, \ldots, Y_{t-p})^T$, $\Theta_q = (\theta_1, \ldots, \theta_q)$ y $\vec{\varepsilon}_t = (\varepsilon_t, \varepsilon_{t-1}, \ldots, \varepsilon_{t-q})^T$.

Siguiendo la metodología clásica para crear un modelo lineal que ajuste una serie temporal dada, un primer paso es imponer la estacionaridad a una serie temporal. Esto implica eliminar la tendencia y la estacionalidad (si se presenta) mediante diferenciación. Diferenciar una serie $\{Y_t\}_1^N$ consiste en crear una nueva serie $\{\delta Y_t\}_1^{N-1}$ donde $\delta Y_i = Y_{i+s} - Y_i$, y s es el orden de diferenciación, por lo que se pierden datos en el proceso. Seguidamente, hay que buscar los valores p y q, mediante inspección visual de las gráficas de unos estadísticos: para estimar el valor de p se suele utilizar la gráfica de la función de autocorrelación parcial y para estimar q la de la función de autocorrelación.

El coeficiente de la función de autocorrelación (*Autocorrelation Coefficient Function*, ACF) de orden k para una serie se obtiene con la siguiente expresión:

$$r_k = \frac{\sum_{t=1}^{N-k} (Y_t - \mu)(Y_{t+k} - \mu)}{\sum_{i=1}^{N} (Y_t - \mu)^2} \tag{2.7}$$

donde r_k es el coeficiente de de autocorrelación de orden k y μ la media de la serie temporal $\{\delta Y_t\}_1^N$.

El coeficiente de la función de autocorrelación parcial (*Partial Autocorrelation Coefficient Function*, PACF) de orden k se define como el último coeficiente de la siguiente ecuación:

$$w_t = \Phi_{1k} w_{t-1} + \Phi_{2k} w_{t-2} + \cdots + \Phi_{kk} w_{t-k} + \varepsilon_t \tag{2.8}$$

donde $w_t = (Y_t - \mu)$. Planteando y resolviendo el sistema de ecuaciones se pueden obtener los coeficientes de PACF $\{\Phi_{ii}\}_{i=1}^k$.

Un modelo $AR(p)$ (o $ARMA(p,0)$) presenta una función de autocorrelación que decrece rápidamente hacia cero, bien de forma regular, sinusoidal o alternando valores positivos y negativos, junto con una función de autocorrelación parcial con tantos valores distintos de cero como orden del autorregresivo. Esto se puede apreciar en las Figura 2.1 y 2.2 para una serie artificial $AR(4)$. Una regla simétrica puede extraerse para los modelos $MA(q)$ (o $ARMA(0,q)$): Un modelo $ARMA(0,q)$ presenta una función de autocorrelación parcial que decrece rápidamente hacia cero, bien de forma regular, sinusoidal o alternando valores positivos y negativos, junto con una función de autocorrelación con tantos valores distintos de cero como orden de la media móvil.

Los modelos $ARMA(p,q)$ se pueden ajustar mediante regresión de mínimos cuadrados para optimizar los valores de los parámetros con respecto al error. Apli-

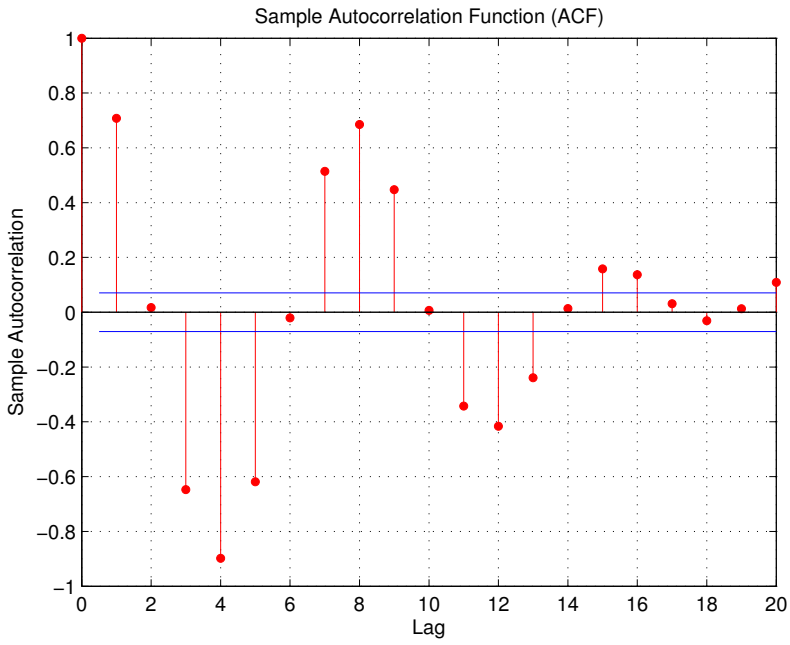


Fig. 2.1: Coeficientes de la función de autocorrelación para una serie $AR(4)$

cando sistemáticamente los pasos anteriores se pueden obtener un cierto número de modelos ARMA para una misma serie, por lo que el último paso es escoger uno de ellos. Es generalmente una buena práctica trabajar con los p y q más bajos posibles pero que proporcionen resultados adecuados, aunque también se suelen utilizar otros criterios como el de información de Akaike (*Akaike's Information Criterion*, AIC) [Akaike, 1974] o el criterio de longitud de descripción mínima (*Minimum Description Length*, MDL) [Rissanen, 1978].

La metodología expuesta requiere intervención de un experto, ya que para aplicar algunos pasos de la misma se necesita de un importante nivel de conocimiento. Hay trabajos que han presentado sistema basados en *soft-computing* (sistema difusos) que automatizan el proceso [Valenzuela et al., 2008][Rojas et al., 2008].

2.2. Redes Neuronales Artificiales

Las Redes de Neuronas Artificiales [Smith, 1993] (*Artificial Neural Networks*, ANN) son modelos de aprendizaje inspirados en el sistema nervioso de los animales. Se componen de una serie de unidades, denominadas neuronas, que de forma simplificada simulan la funcionalidad de las neuronas biológicas. Una neurona recibe una serie de entradas a través de interconexiones y emite una salida. Esta salida viene dada por tres funciones:

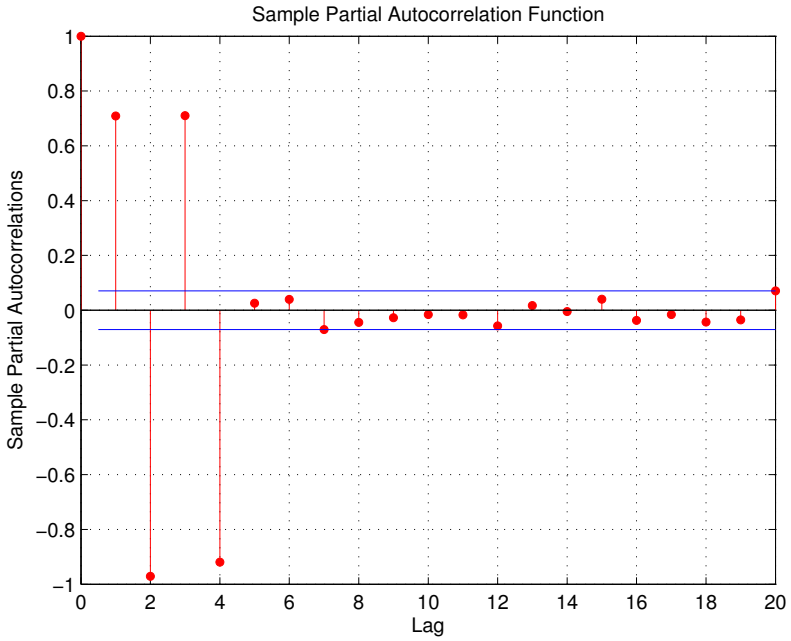


Fig. 2.2: Coeficientes de la función de autocorrelación parcial para una serie $AR(4)$

1. Función de propagación: suele calcular una combinación de cada entrada modificada por el peso de su interconexión (una suma ponderada, por ejemplo).

2. Función de activación: recibe como entrada a la anterior, aunque puede no utilizarse.

3. Función de transferencia, aplicada al valor devuelto por la función de activación.

Las neuronas artificiales se interconectan en una red y colaboran para producir una salida ante un estímulo de entrada (como ejemplo, veanse las Figuras 2.3 y 2.4). El mecanismo de aprendizaje de las ANN se basa en la simulación de la reorganización de las conexiones sinápticas biológicas mediante un mecanismo de pesos, que son ajustados durante la fase de aprendizaje. Con una ANN ya entrenada, el conjunto de los pesos contiene el conocimiento de la red y tiene la propiedad de resolver el problema.

Estos modelos son extremadamente flexibles, pudiendose obtener un número virtualmente infinito de modelos redes distintos al variar el tipo de neurona, la forma de interconectarlas en red y el mecanismo de aprendizaje. Del gran número de tipos de Redes Neuronales Artificiales que se pueden hallar en la literatura citaremos unos pocos a continuación:

1. Perceptrón y Perceptrón Multicapa [Haykin, 1998]

2. Redes de Funciones de Base Radial [Broomhead and Lowe, 1988]

3. Mapas Autoorganizados, Redes de Kohonen [Kohonen, 1997]

4. Redes de Hopfield [Hopfield, 1988]

5. Máquina de Boltzmann [Hinton, 1983]

Las ANN se utilizan tanto para aprendizaje supervisado [Bishop, 1995] como no supervisado [Kohonen, 1997], aunque en esta memoria nos interesa centrarnos esencialmente en los modelos para el aprendizaje supervisado. Se escogieron el Perceptrón Multicapa y las Redes de Funciones de Base Radial como modelos más relevantes en nuestro caso dado que éstos son ámpliamente usados en diversos contextos: aplicaciones médicas [Seidel et al., 2007], meteorología [Hontoria et al., 2005], cálculo numérico [Dehghan and Shokri, 2008] y, por supuesto, predicción de series temporales [Lin and Chen, 2005][Agirre-Basurko et al., 2006][Harpham and Dawson, 2006][Chattopadhyay, 2007].

2.2.1. Perceptrón Multicapa

Un Perceptrón Multicapa (*MultiLayer Perceptron*, MLP) es un modelo de red neuronal artificial formada por múltiples capas de neuronas, a diferencia del Perceptrón (también llamado Perceptrón simple) que consta de una única neurona y que además tan sólo puede abordar problemas lineales [Haykin, 1998]. Es seguramente uno de los modelos más veteranos de Redes Neuronales Artificiales, y por ende, de los modelos inteligentes, pero a pesar de ello sigue siendo de los más usados en un sin fín de aplicaciones prácticas incluso en la actualidad: reconocimiento de patrones en análisis médicos para detección de cáncer [Seidel et al., 2007], obtención de un mapa de radiación solar [Hontoria et al., 2005], predicción de niveles de ozono y nitrógeno atmosférico [Agirre-Basurko et al., 2006], etc.

La arquitectura de un Perceptrón Multicapa consta de varias capas de neuronas (perceptrones simples) en las que la salida de una capa es la entrada de la siguiente. Una primera distinción es que un MLP puede ser totalmente o localmente conectado. En la Figura 2.3 se muestra una red donde cada salida de una neurona de la capa i es entrada de todas las neuronas de la capa $i+1$, mientras que el segundo, cada neurona de la capa i es entrada de una serie de neuronas (región) de la capa $i+1$.

Típicamente, las conexiones entre neuronas tienen un valor de ponderación o peso, y cada neurona tiene una función de activación con la que genera su salida en función de sus entradas. Las capas pueden clasificarse en tres tipos:

1. Capa de entrada: Constituida por aquellas neuronas que introducen las variables de entrada en la red. En estas neuronas no se produce procesamiento.
2. Capas ocultas: Formada por aquellas neuronas cuyas entradas provienen de capas anteriores y las salidas pasan a neuronas de capas posteriores.
3. Capa de salida: Neuronas cuyos valores de salida se corresponden con las salidas de toda la red.

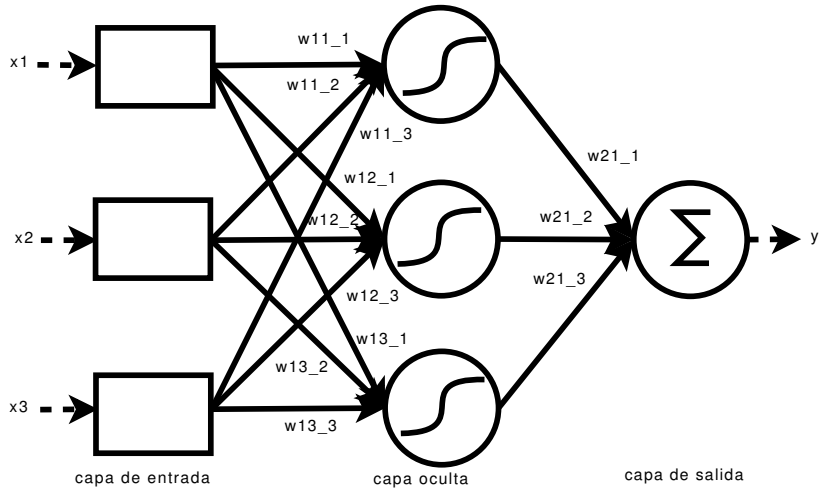


Fig. 2.3: Perceptrón Multicapa totalmente conectado de una capa oculta de 3 neuronas con 3 entradas y una salida

La popularidad de la arquitectura MLP se debe al hecho de que un MLP con una única capa oculta puede aproximar cualquier función continua en un intervalo hasta el nivel deseado, cuestión demostrada por [K.I. Funahashi, 1989] y que proporciona una base sólida al campo de las redes neuronales. Sin embargo, esta resultado no proporciona información alguna sobre el número de nodos ocultos necesarios para llevar a cabo la aproximación. Por tanto, usualmente se utiliza una arquitectura de red totalmente conectada con 1 capa oculta con N neuronas con funciones de activación sigmoidal (2.9) y 1 neurona en la capa de salida con función de agregación lineal. El número de neuronas N se puede determinar mediante diversos métodos, por ejemplo validación cruzada. La salida de la red se puede caracterizar mediante la ecuación 2.10.

$$f(\vec{x},\vec{w}) = \tanh(\langle \vec{x},\vec{w} \rangle) \tag{2.9}$$

$$F(\vec{x}) = \sum_{i=1}^{N} p_i f(\vec{x},\vec{w}_i) \tag{2.10}$$

ENTRADA: X, conjunto de puntos de entrada
ENTRADA: Y, salidas asociadas a las entradas de X
ENTRADA: N, número de neuronas en la capa oculta

SALIDA: Perceptrón Multicapa entrenado

Inicializar la red y sus pesos (aleatoriamente, con un algoritmo rápido, etc.)

repetir
  para $x \in X$ ($y \in Y$ es la salida asociada a y) **hacer**
    Calcular la salida de la red con x, $\hat{y}$
    Calcular el error de la salida de la red con con respecto a la salida real y,
    Aplicar la regla delta para todos los pesos entre la capa oculta y la capa de salida
    Aplicar la regla delta para todos los pesos entre la capa de entrada y la oculta
    Actualizar los pesos de la red
  fin para
hasta Criterio de parada (convergencia, alcanzar un error mínimo, etc)
Devolver la red entrenada

Alg. 2.1: Algoritmo de retropropagación para un Perceptrón Multicapa con una capa oculta

donde f es la función de activación sigmoidal, $\vec{x}$ es un vector de entrada a la red, N es el número de neuronas en la red, $\vec{w}_i$ es el vector de pesos de las entradas y p_i es el peso de la salida la i-ésima neurona de la red.

Para el aprendizaje de este tipo de ANN se suele usar el algoritmo de la *Propagación Hacia Atrás* (también conocido como *Retropropagación del Error* o *Regla Delta Generalizada*, *Backpropagation*) [Werbos, 1974][D.E. Rumelhart, 1986]. Por normalmente usarse este algoritmo en el entrenamiento de estas redes, las MLP son también conectadas como como redes de retropropagación. En el Algoritmo 2.1 se puede ver el pseudocódigo del mismo para MLP con una capa oculta (como la de la Figura 2.3).

El algoritmo de retropropagación técnicamente realiza una optimización estocástica no lineal basada en una evaluación del gradiente de error de las salidas con respecto a los pesos de la red. La velocidad de convergencia de este método es grande pero generalmente acaba cayendo en un mínimo local, es decir, este algoritmo tiene el problema de caer en soluciones suboptimales. En la literatura se pueden hallar modificaciones [Bello, 1992][Murray and Edwards, 1993] tanto para el caso general como para problemas específicos [B.B., 2000] así como mé-

todos alternativos basados en otros modelos inteligentes más recientes, como las Máquinas de Vectores de Soporte [Suykens and Vandewalle, 1999].

2.2.2. Redes Neuronales de Funciones de Base Radial

Dentro del amplio abanico de tipos de redes neuronales, existe un tipo de red cuyo nombre completo es red neuronal de funciones de base radial (*Radial Basis Function Neural Network*, RBFNN) [Broomhead and Lowe, 1988]. Estos aproximadores universales [Park and Sandberg, 1991] se definen matemáticamente como:

$$\mathcal{F}\left(\vec{x}_k; C, R, \Omega\right) = \sum_{i=1}^{m} \phi(\vec{x}_k; \vec{c}_i, r_i) \cdot \Omega_i \tag{2.11}$$

donde m es el número de neuronas en la capa oculta, $C = \{\vec{c}_1, ..., \vec{c}_m\}$ son los centros de cada una de las RBF, $R = \{r_1, ..., r_m\}$ son los valores de los radios de las RBF, $\Omega = \{\Omega_1, ..., \Omega_m\}$ es el conjunto de los pesos correspondientes a cada una de las unidades de procesamiento y $\phi(\vec{x}_k; \vec{c}_i, r_i)$ representa a una RBF. Dentro de las posibles funciones de activación de base radial, las funciones Gaussianas (Eq. 2.12), se muestran particularmente apropiadas para ser usadas por una red neuronal para aproximar funciones [Bors, 2001; Rojas et al., 1998; Poggio and Girosi, 1990; Park and Sandberg, 1991].

$$e^{-\frac{||\vec{x}_k - \vec{c}_i||^2}{r_i^2}}. \tag{2.12}$$

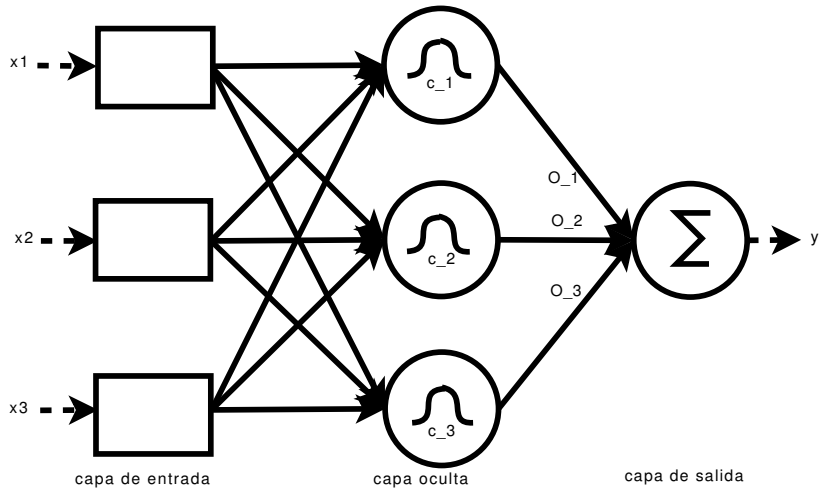


Fig. 2.4: Red Neuronal de Funciones de Base Radial

Para poder crear una RBFNN (ver Figura 2.4) hay que conocer cuantas RBF deben componer la red, dónde estarán centradas las RBF (la mayoría de los autores han usado algoritmos de *clustering* para inicializar sus centros) [Moody and Darken, 1989][Karayiannis and Mi, 1997a][Uykan and Güzelis, 1997][Gomm and Yu, 2000][González et al., 2002][Guillén et al., 2007], cuál será la amplitud de cada una [Benoudjit and Verleysen, 2003] y, por último, conocer los pesos que ponderan las salidas de las RBF. La razón de haber mencionado el cálculo de los pesos en último lugar no es casual puesto que es el único de todos los parámetros cuyo valor puede ser calculado de forma óptima. Una vez establecidos los valores y el número de RBF, la obtención del valor de los pesos puede plantearse como un sistema de ecuaciones lineal entre la salida de la función a aproximar y la matriz de activación de las RBF (P):

$$\vec{y} = P\Omega \tag{2.13}$$

donde P es una matriz de dimension $n \times m$ en la que P_{ki} representa el valor de la i-ésima función Gaussiana cuando se le proporciona como entrada el vector de entrada $\vec{x}_k$, siendo n igual al número de vectores de entrada. Hay muchas formas de resolver este sistema de ecuaciones. Entre las más usadas se encuentran la descomposición de Cholesky [Pomares et al., 2000], la descomposición en valores singulares (*Singular Vale Decomposition*, SVD) [Kanjilal and Banerjee, 1995] y el método de mínimos cuadrados ortogonales (*Orthogonal Least Squares*,OLS) [Chen et al., 1991].

Existe extensa literatura acerca del entrenamiento de las RBF [Karayiannis and Mi, 1997b] [Rojas et al., 2000c] [González et al., 2003] [Hatanaka et al., 2003]. Pero dado que sólo serán utilizadas como modelos de referencia, sólo nombraremos el siguiente método:

1. Para un número dado de RBF: se calculan los centros mediante el algoritmo ICFA [Guillén et al., 2007] y a continuación los radios mediante el algoritmo de Levenberg-Marquardt.

2. Para fijar el número de RBF: partiendo de un número bajo de RBF, se calcula el error de validación cruzada de orden 2 sobre los datos de entrenamiento (utilizando el método del primer punto). Se va incrementando el número de RBF y calculando el error de validación cruzada de orden 2 sobre los datos de entrenamiento hasta que no se obtenga mejora.

2.3. Métodos kernel

Desde que las Máquinas de Soporte de Vectores (*Support Vector Machine*, SVM) se desarrollaran para clasificación (SVC), se han creado y aplicado multitud de técnicas basadas en el llamado *Kernel Trick* [Müller et al., 2001][Scholkopf

and Smola, 2001]. Toda esta familia de técnicas ha recibido el nombre de *Métodos de Kernel*. Las SVMs tiene una sólida base matemática e ignoran la llamada *maldición de la dimensionalidad* con respecto a las entradas, escalando su coste computacional con respecto al número de datos de entrenamiento. El *Kernel Trick* se ha utilizado para obtener extensiones no lineales de otros algoritmos, cómo por ejemplo el Análisis de Componentes Principales [Wu et al., 1997], algoritmos de clustering [Zhang and Chen, 2003] o los k-vecinos más cercanos [Rubio et al., 2008], ya que es inmediato de aplicar una vez se ha formulado un método en función de los valores de una función sobre pares de entrada en vez de directamente sobre las entradas [Müller et al., 2001][Scholkopf and Smola, 2001]. Los métodos así obtenidos son independientes de la estructura concreta de la entrada y, por lo tanto, una vez definida la función de *kernel* no padecen la llamada *maldición de la dimensionalidad*, es decir, su complejidad computacional es independiente del número de variables de entradas, pues están definidos en base a valores resultados de la evaluación de una función sobre pares de datos de entrada.

Dado el éxito de las SVC, pronto se propuso una adaptación para aproximación funcional de las SVM, las SVR, pero éstas presentan bastantes desventajas con respecto a la original. Las Máquinas de Vectores de Soporte de Mínimos Cuadrados (*Least Square SVM*, LSSVM) son un modelo que arregla algunos de los problemas de las SVR.

La regresión mediante Procesos Gaussianos es una técnica que parte de otra técnica de geo-estadística conocida como *kriging*. En los años 90, el artículo Neal, 1996 hizo que se levantara un cierto interés en la comunidad de aprendizaje de máquina por este método, debido a que demostraba que las Redes de Funciones de Base Radial (RBFNN) son equivalentes a Procesos Gaussianos cuando el número de neuronas en la capa oculta se hacía infinito. Algunos de los mayores problemas de la técnica son la complejidad de su formulación y su alto coste computacional [MacKay, 1998].

Como se verá en los siguientes apartados, todas estas técnicas tienen problemas similares: altos costes en cómputo para entrenar los modelos cuando crece el número de datos de entrenamiento (pero no la dimensionalidad de los mismos) y para ajustar los parámetros, así como dar lugar a modelos *voluminosos* y escasamente interpretables.

2.3.1. Máquinas de Vectores de Soporte para Clasificación

La totalidad de los métodos *kernel* para clasificación se basan en el trabajo de Vapnik de generalización del clasificador hiperplanar para casos no lineales [Scholkopf and Smola, 2001]. El clasificador hiperplanar binario es una técnica de clasificación que busca el *mejor* hiperplano que separe los puntos de dos clases distintas en un espacio d-dimensional. El *mejor* hiperplano separador define un clasificador que separa los puntos de las dos clases con más generalización, en-

tendiendo que la generalización mayor es aquella obtenida con el hiperplano que quede a mayor distancia de los conjuntos de puntos que separan. La citada distancia se denomina *margen* del hiperplano separador. Más formalmente, dado un conjunto de ejemplos (x_i, y_i), $i = 1, \ldots, N$, donde $x_i \in \mathbb{R}^d$ e $y_i \in \{-1, 1\}$ (etiqueta de la clase del punto), se busca resolver:

$$\begin{array}{rcl} \min\limits_{w\in\mathbb{R}^d,\ b\in\mathbb{R}} \tau(w,b) & = & \frac{1}{2}\|w\|^2, \\ con & & \\ y_i(\langle w, x_i\rangle + b) & \geq & 1,\ \ \forall i = 1, \ldots, N. \end{array} \tag{2.14}$$

donde τ es la función a minimizar con respecto al vector w y el valor b, que definen el hiperplano que separa las clases con mayor distancia a las mismas (es decir, con margen máximo). Es posible que este problema no tenga solución (es decir, que no exista un hiperplano separador para un conjunto dado de puntos), por lo que se relaja la condición permitiendo que se clasifiquen mal algunos ejemplos, quedando el problema como:

$$\begin{array}{rcl} \min\limits_{w\in\mathbb{R}^d,\ b,\zeta_i\in\mathbb{R}} \tau(w,b,\zeta) & = & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\zeta_i, \\ con & & \\ \zeta_i & \geq & 0 \\ y_i(\langle w, x_i\rangle + b) & \geq & 1-\zeta_i,\ \ \forall i = 1, \ldots, N. \end{array} \tag{2.15}$$

donde, con respecto al problema planteando en la Ecuación 2.14, se introducen las variables de holgura $\zeta_i,\ \forall i = 1, \ldots, N$ y el parámetro $C > 0$, que controla la proporción error/precisión (mayores valores de C implica menores tolerancias al error).

Este es un problema de optimización de programación cuadrática resoluble mediante el método de los multiplicadores de Lagrange. Aplicando el método de los multiplicadores de Lagrange obtenemos el problema en forma dual:

$$\begin{array}{rcl} \max\limits_{\alpha_i\in\mathbb{R}} J(\alpha) & = & \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j\rangle, \\ con & & \\ \sum_{i=1}^{N}\alpha_i y_i & = & 0 \\ \alpha_i & \in & [0, C],\ \ \forall i = 1, \ldots, N. \end{array} \tag{2.16}$$

donde α_i es el multiplicador de Lagrange asociado al ejemplo de entada (x_i, y_i). Resolviendo este problema puede formularse el clasificador como:

$$f(x) = \sum_{i=1}^{N}\alpha_i y_i \langle x_i, x\rangle + b \tag{2.17}$$

donde x es el punto de entrada que se va a clasificar. Como puede observarse, todo queda formulado en término de productos escalares de los vectores de entrada. Es

notable que sólo los puntos de la entrada que estén en el margen tienen $\alpha_i > 0$ por lo que el resto puede ignorarse en el modelo final.

Las Máquinas de Vectores de Soporte (SVM) se basan en aprovechar esto para definir una función *kernel* $k(x,x') = \langle \Phi(x), \Phi(x) \rangle$ donde Φ es una función que mapea los puntos del espacio de entrada a un espacio de características de mayor dimensionalidad en el cual se define el hiperplano [Schölkopf et al., 1999]. Este proceso se denomina *Kernel Trick*. El mapeado Φ no tiene por que definirse explícitamente, ya que puede utilizarse como *kernel* cualquier función k que cumpla el teorema de Mercer [Scholkopf and Smola, 2001]. Una función $k(u,v)$ verifica el teorema de Mercer si para toda función g de cuadrado integrable:

$$\int_{u,v} k(u,v)g(u)g(v)dudv > 0 \tag{2.18}$$

Ejemplos de funciones que cumplen con este son los denominados *kernel* lineal, polinomial y Gaussiano (también RBF):

$$k(x_i,x_j) = \langle x_i, x_j \rangle. \tag{2.19}$$

$$k(x_i,x_j) = (\langle x_i, x_j \rangle + 1)^d,\ d \in \mathbb{N} \tag{2.20}$$

$$k(x_i,x_j) = \exp\left[-\frac{1}{\sigma^2} \|x_i - x_j\|^2 \right]. \tag{2.21}$$

Es notable que algunas funciones *kernel* utilizadas en la literatura no cumplen el teorema de Mercer, como el *kernel* Sigmoide (también MLP, por su similitud a las redes neuronales MLP con función de activación sigmoide):

$$k(x_i,x_j) = \tanh\left(s\langle x_i, x_j \rangle + t \right). \tag{2.22}$$

La complejidad computacional de SVC escala con el número de puntos de entrenamiento lo que dificulta su aplicación a grandes conjuntos de datos, aunque hay diversos trabajos que tratan de solucionarlo, por ejemplo aplicando programación paralela [Graf et al., 2005].

2.3.2. Máquinas de Vectores de Soporte para Regresión

Las SVM para regresión son una modificación del modelo para clasificación. Dado un conjunto de muestras de una función $\{(x_1,y_1),\ldots,(x_N,y_N)\} \subset X \times \mathbb{R}$, donde X es un conjunto en el que hay definido una operación de producto escalar entre sus elementos, la técnica de regresión ε-SVR busca una función $f(x)$ tan suave cómo sea posible cuyos valores difieran de cada salida real y_i no más de ε para todo $x_i \in X$ (ésta se conoce como la función de error ε-insensible). La función $f(x)$ se define como:

$$f(x) = \langle w,x \rangle + b,\ w,x \in X,\ b \in \mathbb{R}. \tag{2.23}$$

Como en el caso de SVM para clasificación, el problema puede ser no resoluble, por lo que una cantidad de error mayor que ε debe ser tolerada. Se introducen las variables de holgura $\varsigma_i, \varsigma_i^*$:

$$\begin{array}{rcl} \min\limits_{w \in X,\ b, \varsigma_i, \varsigma_i^* \in \mathbb{R}} \tau(w,b,\varsigma,\varsigma*) & = & \frac{1}{2}\|w\|^2 + C\sum\limits_{i=1}^{N}(\varsigma_i + \varsigma_i^*), \\ subject\ to & & \\ y_i - (\langle w,x_i \rangle + b) & \leq & \varepsilon + \varsigma_i \\ (\langle w,x_i \rangle + b) - y_i & \leq & \varepsilon + \varsigma_i^* \\ \varsigma_i, \varsigma_i^* & \geq & 0,\ \ \forall i = 1,\ldots,N. \end{array} \tag{2.24}$$

La constante $C > 0$ representa el compromiso entre suavidad y el número de errores mayores que ε de la función. Al igual que en el caso de SVM para clasificación (SVC), es un problema de optimización de programación cuadrática resoluble mediante el método de los multiplicadores de Lagrange. Pasando el problema a forma dual, para resolverlo solo resta hallar los multiplicadores de Lagrange α_i y el coeficiente b. La solución (2.23) se escribe en términos de los α_i, α_i^* y b, así como de productos escalares que pueden ser reemplazados por una función de *kernel* k (como se vió en el apartado 2.3). La función de regresión queda:

$$f(x) = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)k(x,x_i) + b. \tag{2.25}$$

Las mayores desventajas de las SVR frente a las SVC son:

- La introducción de un parámetro adicional ε.
- El cálculo del doble de multiplicadores de Lagrange (α_i, α_i^*, es decir, 2 por cada x_i).
- El modelo resultante no es disperso, es decir, se utilizan todos los puntos de entrenamiento en el modelo.

A pesar de sus serios inconvenientes frente a SVC (y, como se verá, otras técnicas *kernel* ya para regresión), las SVR han sido empleadas con éxito en algunos trabajos para modelado de series temporales, por ejemplo, prediciendo cargas de redes eléctricas [Pai and Hong, 2005], o series financieras [Tay, 2001]. Hoy día se usan menos ya que las LSSVM obtienen mejores resultados y con menor complejidad computacional (ver Sección 2.3.3).

2.3.3. Las Máquinas de Vectores de Soporte de Mínimos Cuadrados para Regresión

Las Máquinas de Vectores de Soporte de Mínimos Cuadrados (LSSVM) son una modificación de la formulación estándar de SVM para regresión que lleva a entrenar el modelo mediante la resolución de un sistema lineal en vez de plantear un problema de optimización de programación cuadrática [Suykens et al., 2002]. Las LSSVM están muy relacionadas con las redes de regularización y los Procesos Gaussianos, pero enfatizan y explotan su interpretación desde el punto de vista de la teoría de optimización. En las Máquinas de Vectores de Soporte de Mínimos Cuadrados para Regresión se sustituye la función de error ε-insensible de la formulación de SVR por una de error cuadrático, planteando el problema:

$$\begin{aligned} \min_{w\in X,\ b,e_i\in\mathbb{R}} \tau(w,b,e) &= \tfrac{1}{2}\|w\|^2+\gamma\tfrac{1}{2}\sum_{i=1}^{N} e_i^2, \\ \text{con} & \\ (y_i-(\langle w,x_i\rangle+b)) &= e_i,\ \ \forall i=1,\ldots,N. \end{aligned} \tag{2.26}$$

donde $\gamma>0$ tiene una función similar al parámetro C en la formulación de SVM. Como en el caso de las SVR (ver apartado 2.3.2), se puede resolver dicho problema mediante el método de los multiplicadores de Lagrange, pero al final se obtiene un sistema lineal en vez de un problema de optimización de programación cuadrática:

$$\left[\begin{array}{c|c} 0 & 1_N^T \\ \hline 1_N & \Omega+I/\gamma \end{array}\right]\left[\begin{array}{c} b \\ \hline \alpha \end{array}\right]=\left[\begin{array}{c} 0 \\ \hline y \end{array}\right] \tag{2.27}$$

donde $\Omega_{ij}=\langle x_i,x_j\rangle$ e I es la matriz identidad. Si aplicamos el *Kernel Trick*, $\Omega_{ij}=k(x_i,x_j)$, al final puede escribir la función resultante en términos de los coeficientes α_i, b y valores de una función de *kernel* k:

$$f(x)=\sum_{i=1}^{N}\alpha_i k(x,x_i)+b. \tag{2.28}$$

Las mayores ventajas de las LSSVM para regresión con respecto a SVR, aparte de la resolución más sencilla del problema planteado, es que desaparece el parámetro ε y disminuyen a la mitad el número de multiplicadores de Lagrange. El modelo también se ha aplicado a clasificación, pero frente a SVC tiene la desventaja de no generar modelos dispersos (*non-sparse*).

2.3.4. Regresión mediante Procesos Gaussianos

Un Proceso Gaussiano se define como una distribución Gaussiana de probabilidad definida sobre un espacio de funciones. Si consideramos nula la media,

un Proceso Gaussiano queda caracterizado por su función de covarianza $C(x,x')$, que puede verse como una función *kernel*. Dado un conjunto de observaciones de una función $\{(x_1,y_1),\ldots,(x_N,y_N)\} \subset X \times \mathbb{R}$, se denota por Σ la matriz de covarianza generada de los datos X, $\Sigma_{ij} \equiv C(x_i,x_j)$, y por Y el vector de salidas de entrenamiento $Y = (y_1,\ldots,y_N)^T$. Para una nueva entrada $\hat{X}$, se puede calcular la distribución de probabilidad de la predicción, que será también Gaussiana, mediante su media y su varianza:

$$\begin{array}{rcl} \hat{Y} & = & K^T\Sigma^{-1}Y. \\ \hat{\sigma}^2 & = & diag(\hat{\Sigma} - K^T\Sigma^{-1}K). \end{array} \tag{2.29}$$

donde $K_{ij} = C(x_i,\hat{x}_j)$, $\hat{\Sigma}_{ij} = C(\hat{x}_i,\hat{x}_j)$ y $\hat{x}_i \in \hat{X}$. La media de la distribución de predicción $\hat{Y}$ para las nuevas entradas se interpreta como las salidas predecidas y la desviación estándar $\hat{\sigma}$ como las cotas de error.

Dada una función de covarianza $C(x,y;\Theta)$, se denominan $\Theta = (\theta_1,\ldots,\theta_m)$ a sus hiperparámetros. En un entorno de trabajo Bayesiano, los hiperparámetros pueden ser ajustados minimizando el Negativo del Logaritmo de la Probabilidad Marginal (*Negative Log Marginal Likelihood*, NLML):

$$NLML = -logP(Y|X,\Theta) = \frac{N}{2}\log(2\pi) + \frac{1}{2}\log(|\Sigma|) + \frac{1}{2}Y^T\Sigma^{-1}Y. \tag{2.30}$$

La principal ventaja de este criterio es que existen expresiones analíticas para la derivadas parciales con respecto a los hiperparámetros, por lo que se pueden aplicar técnicas de descenso en gradiente para su optimización [MacKay, 1998; Rasmussen and Williams, 2005]. Hay más criterios además del NLML [Rasmussen and Williams, 2005]. Por ejemplo, la validación cruzada es un método costoso pero muy adecuado para tratar con malas especificaciones del modelo (covarianza) o ruido en los datos. Para Procesos Gaussianos hay disponibles, a cuanbio de un pequeño esfuerzo computacional extra, expresiones para una versión *Dejando un valor fuera* (*Leave-One-Out*) del criterio NLML: el *Leave-One-Out Negative Log Predictive Likelihood* (NLPL-LOO), así como otra que sólo tiene en cuenta el error, *Leave-One-Out Mean Square Error*, (MSE-LOO) que se evalúa a partir de la inversa de la matriz de covarianza:

$$NLPL - LOO = \frac{1}{2}\left(-N\log(2\pi) + \sum_{i=1}^{N}\log\left([\Sigma^{-1}]_{ii}\right) - \frac{[\Sigma^{-1}Y]_i^2}{[\Sigma^{-1}]_{ii}}\right). \tag{2.31}$$

$$MSE - LOO = \frac{1}{2}\sum_{i=1}^{N}\frac{[\Sigma^{-1}Y]_i^2}{[\Sigma^{-1}]_{ii}^2}. \tag{2.32}$$

Volviendo al entorno Bayesiano, pueden expresarse preferencias acerca de los valores de los hiperparámetros mediante un término que se añada al criterio (*prior over hyperparameters*). Por ejemplo, para una regularización (entendida como suavidad de la función) es muy común preferir valores pequeños de parámetros, esto puede expresarse sumando al criterio este término:

$$prior(\Theta) = \|\Theta\|^2. \tag{2.33}$$

La formulación de Procesos Gaussianos permite también expresar dentro de la función de covarianza mediante un término un modelo de ruido que se suponga en los datos. Por ejemplo, si suponemos ruido Gaussiano aditivo de media 0 y varianza σ_e^2 puede añadirse a la diagonal de la matriz de covarianza el término siguiente [Neal, 1996]:

$$\Sigma_{i,j} = C(x_i, x_j) + \delta_{i,j}\sigma_e^2. \tag{2.34}$$

Cuando σ_e^2 se fija a priori es equivalente a término *jitter*, es decir, un valor añadido a la diagonal de la matriz para mejorar sus características de cara a los cálculos (evitando que sea singular, por ejemplo). Sin embargo, aunque matemáticamente sean idénticos, conceptualmente son distintos y además se puede incorporar a los hiperparámetros al ajustar σ_e^2.

La única restricción que tiene una función para ser función de covarianza es que debe de generar matrices definidas positivas, por ejemplo:

$$C(x_i, x_j; \{\theta_1, \theta_2\}) = \theta_1 \exp\left[-\frac{1}{{\theta_2}^2}\|x_i - x_j\|^2\right]. \tag{2.35}$$

donde $\{\theta_1, \theta_2\}$ son los hiperparámetros de la función.

El coste de esta flexibilidad a la hora de embeber preferencias y conocimiento previo en el modelo es su compleja especificación, la alta influencia de dichos conocimientos previos y su especificación en el rendimiento de los modelos. Por estos motivos, los Procesos Gaussianos no son muy populares, aunque han sido aplicados con éxitos a diversos problemas y particularmente a la predicción de series temporales [Brahim-Belhouari and Bermak, 2004][Rubio et al., 2007]. Otro problema que tienen los Procesos Gaussianos son su alto coste computacional con una evaluación de NLML de $O(N^3)$, aunque hay métodos para reducir algo dicho coste [MacKay, 1998][Rasmussen and Williams, 2005][Leithead and Zhang, 2007][Zhang and Leithead, 2007] incluso habiendose creado implementaciones paralelas de los mismos [Keane et al., 2002]. Finalmente, comparten con SVR y LSSVM el problema de no generar modelos dispersos, lo que supone que si se desea reducir el tamaño de un modelo hay que aplicar a posteriori un método de poda, añadiendo tiempo de cómputo [Bo et al., 2006].

2.4. Algoritmos Genéticos

Los algoritmos genéticos [Holland, 1975][Forrest, 1993][Goldberg, 1994][Michalewicz, 1996][Rojas et al., 2001] son algoritmos de búsqueda basados en la selección natural [Darwin, 1859] y en las leyes de la genética.

Aunque sean aleatorios, los algoritmos genéticos son mucho más potentes que una búsqueda ciega, ya que explotan información histórica para especular sobre la localización de nuevas zonas del espacio de soluciones con una mayor adaptación que las que están siendo explotadas actualmente. Sus objetivos fueron claramente definidos por Holland en [Holland, 1975]:

- abstraer y explicar rigurosamente los procesos adaptativos de los sistemas naturales, y
- diseñar sistemas artificiales que mantengan los mecanismos importantes de los sistemas naturales.

Formalmente, un algoritmo genético es un algoritmo probabilístico[1] que en un instante t mantiene una población de cromosomas $\mathcal{P}^t = \left\{\mathfrak{l}_1^t, ..., \mathfrak{l}_j^t, ..., \mathfrak{l}_\eta^t\right\}$. Cada uno de estos cromosomas representa una solución potencial para un problema dado y se encuentra codificado mediante una cadena de bits. Los bits de dicha cadena se deben interpretar de acuerdo a una estructura de datos D para identificar los valores que toman cada una de las características que definen la solución representada por la cadena binaria. La bondad de las soluciones codificadas en cada uno de los cromosomas de la población se mide de acuerdo a una función de adaptación f que asigna a cada cromosoma un grado de adaptación que será mejor cuanto mejor sea el cromosoma. Una vez que los cromosomas han sido evaluados, se seleccionan los más aptos para formar una nueva población $\mathcal{P}^{t+1}$ en la que sólo los cromosomas que han sido seleccionados podrán reproducirse para generar nuevas soluciones para el problema. El proceso de reproducción se lleva a cabo mediante la aplicación de operadores genéticos que pueden ser de mutación o de cruce. Los operadores de mutación μ_k son unarios, es decir, se aplican sobre una cadena de bits, y crean una nueva solución aplicando un cambio aleatorio a un cromosoma ya existente en la población:

$$\mu_k : D \rightarrow D \tag{2.36}$$

En cambio, los operadores de cruce χ_j combinan la información de varios cromosomas para formar varios descendientes que posiblemente heredarán las mejores cualidades de cada uno de sus progenitores:

[1] Su comportamiento es no determinista. En dos ejecuciones distintas no tiene por que encontrar la misma solución.

$t \leftarrow 0$
Crear la población inicial $\mathcal{P}^t$
Evaluar $\mathcal{P}^t$
repetir
  $t \leftarrow t+1$
  Seleccionar $\mathcal{P}^t$ tomando los cromosomas más aptos de $\mathcal{P}^{t-1}$
  Reproducir $\mathcal{P}^t$
  Evaluar $\mathcal{P}^t$
hasta No se cumpla la *condición de parada*
Devolver el cromosoma más apto de $\mathcal{P}^t$

Alg. 2.2: Algoritmo genético básico

$$\chi_j : D \times ... \times D \rightarrow D \times ... \times D \tag{2.37}$$

Esta secuencia de pasos se conoce como generación, y su repetición lleva a que la población converja a una solución que se espera que sea casi-óptima. Los pasos a seguir en un algoritmo genético básico se detallan en el Algoritmo 2.2.

Debido a su estrecha relación con la genética natural, se han acuñado diversos términos de ésta para referirse a las estructuras de datos, la información o los procesos en el ámbito de los algoritmos genéticos. A cada una de las posibles soluciones para el problema se las conoce como *cromosomas*. En cada uno de ellos se pueden diferenciar dos partes, la estructura de datos que le da soporte, conocida como *genotipo*, que en el caso de los algoritmos genéticos es una cadena de bits, y el *fenotipo*, que es la información genética en sí, independientemente de la forma en la que codifique. Cada cromosoma está compuesto por unidades mínimas de información llamadas *genes* que describen las distintas características atómicas de la solución. Los genes relacionados con ciertas características están colocados en una posición determinada y cada uno de los estados o valores que pueden contener se llama *alelo*. El conjunto de posibles soluciones que se almacena en el momento actual se conoce como *población*, y cada una de las iteraciones del proceso evolutivo se llama *generación*.

Desde su introducción, han aparecido multitud de artículos y disertaciones que establecen su aplicabilidad a multitud de áreas como la aproximación funcional, el control de procesos, la optimización de rutas, horarios, consultas a bases de datos, problemas de transporte, etc. [Bäck, 1995][Bennet et al., 1991][Booker, 1982][De Jong, 1975][De Jong, 1985][Fogel, 1995][Goldberg, 1989a][Michalewicz et al., 1990][Michalewicz, 1996][Vignaux and Michalewicz, 1991]. Las razones para su popularidad están bien claras: son algoritmos de búsqueda muy poderosos al posibilitar la explotación / exploración de múltiples soluciones, en contraposición

con los algoritmos de búsqueda tradicionales, se pueden aplicar a cualquier tipo de problemas, son capaces de manejar restricciones sobre el espacio de soluciones válidas o aceptables, no están limitados por la topología del espacio de búsqueda (pueden buscar en espacios no derivables, no continuos, multimodales, etc.) y son computacionalmente sencillos y fáciles de implementar.

Los algoritmos genéticos son altamente configurables. Sus múltiples grados de libertad permiten ajustar el proceso de búsqueda a las necesidades de cada problema, pero si no se controlan adecuadamente puede ser que no se llegue as encontrar una solución aceptable. Para resolver un problema particular, hay que estudiar detalladamente cómo se va a implementar cada una de las siguientes componentes del algoritmo:

- la codificación usada para representar las posibles soluciones,
- la creación de la población inicial,
- el diseño de la función de evaluación,
- el proceso de reproducción,
- el proceso de selección,
- el valor de los parámetros (probabilidades de aplicación de operadores y tamaño de la población), y
- el criterio de parada.

2.5. Programación Genética

En [Koza, 1992], Koza propuso una clase de algoritmos evolutivos llamada Programación Genética (GP), cuya idea es evolucionar programas de computadora para la creación automática de nuevos programas de computadora. Aunque en principio fue diseñada para ello, no sólo se ha utilizado para generar programas, sino que cualquier otro tipo de soluciones cuya estructura sea similar a la de un programa, por ejemplo fórmulas matemáticas o circuitos electrónicos. La Programación Genética es más general y, de hecho, incluye los algoritmos genéticos. Las aplicaciones de estos algoritmos son muy diversas:

- Integración, derivación e inversión simbólicas.
- Predicción de secuencias.
- Compresión de datos con pérdidas (imágenes...).
- Descubrimiento de leyes a partir de datos empíricos (leyes de Kepler...).

- Demostración de identidades matemáticas.
- Diseño en ingeniería (civil, etc...).
- Estrategias en robots (hormigas artificiales).
- Regresión simbólica.
- Diseño de circuitos digitales (multiplexor...).
- Comportamiento emergente (hormigas artificiales, agentes, robots...).
- Diseño de controladores y automatismos (péndulo invertido, parquear un trailer...).
- Diseño de estrategias óptimas en juegos (Pac-Man).
- Diseño de arquitecturas internas (para *path planning* de robots...).
- Generación de secuencias seudoaleatorias.
- Clasificación de datos (*Clustering*, *Data Mining*...).

La Programación Genética es computacionalmente muy costosa lo que limitó sus primeras aplicaciones a resolver problemas relativamente simples. Pero, tras varías mejoras en la tecnología y al aumento de la potencia de cómputo, la Programación Genética ha obtenidos resultados muy competitivos. Por ejemplo, hay una lista de casi 40 resultados de este tipo de algoritmos[2] que se han clasificado como competitivos con el ser humano, en áreas como la computación cuántica, diseño electrónico, juegos, ordenación, etc. Se han replicado algunas invenciones realizadas tras el año 2000, e incluso producido invenciones patentables.

En su definición más tradicional, hecha por Koza, los programas que se desarrollan están codificados en forma de árboles de análisis sintáctico (*parse trees* en inglés). El uso de árboles de expresiones (*parse trees*) tiene las ventajas de prevenir errores sintácticos, que pueden llevar a individuos no válidos, y la jerarquía en el árbol previene de problemas con la precedencia de operadores. Hay que definir los nodos del árbol convenientemente, es decir, el conjunto de terminales que está compuesto por las entradas posibles al individuo, constantes y funciones de aridad 0, y el conjunto de funciones está compuesto por los operadores y funciones que pueden componer a un individuo. Por ejemplo podemos definir los siguientes nodos para árboles que representen un sencillo lenguaje imperativo:

- Funciones booleanas: *AND, OR, NOT, XOR*.

[2] `http://www.genetic-programming.com/humancompetitive.html`

- Funciones aritméticas: *PLUS, MINUS, MULT, DIV*.
- Sentencias condicionales: *IF, THEN, ELSE, CASE, SWITCH*
- Sentencias para iteraciones: *WHILE, FOR, REPEAT..UNTIL*

El conjunto de terminales y funciones elegidos para resolver un problema particular debe ser, obviamente, suficiente para representar una solución al problema. Por otro lado, no es conveniente usar un número grande de funciones, debido a que esto aumenta el tamaño del espacio de búsqueda. Además es deseable que las funciones puedan manejar todos los argumentos que eventualmente podrían llegar a tener.

Uno de los parámetros principales para un algoritmo de Programación Genética es el tamaño máximo de un programa. Este límite puede estar impuesto sobre el número de nodos o sobre la profundidad del árbol. Con esto en mente, también es importante tener una población inicial diversa para obtener buenos resultados, por lo que usualmente se utilizan métodos para generar poblaciones iniciales con árboles de diversos tamaño [Koza, 1992]:

Método grow: crea un individuo de como mucho el tamaño máximo permitido.

Método full: crea un individuo del tamaño máximo permitido.

Método ramped half and half: está diseñado para garantizar una población inicial con alto grado de diversidad de tamaño de árboles. Si suponemos D el tamaño máximo definido para un árbol, se divide la población en $D-1$ grupos con tamaño máximo $2,..,D$ en los que la mitad de cada grupo se crea usando el método *grow* y la otra mitad con el *full*.

Los algoritmos de Programación Genética constan, al igual que los algoritmos genéticos generales, de operadores genéticos de cruce y mutación. El operador de cruce (*crossover*) clásico de la Programación Genética es el operador de intercambio de sub-árboles. Éste elige al azar un nodo de cada árbol y luego se intercambian los subárboles bajo estos nodos. Hay mayor diversidad de operadores de mutación, a continuación se listan algunos:

Mutación puntual: sólo un nodo es intercambiado por otro de la misma clase

Permutación: los argumentos de un nodo son permutados

Levantamiento: un nuevo individuo es generado a partir de un subárbol

Expansión: un terminal es cambiado por un árbol generado al azar

Mutación de Subárbol: un subárbol es reemplazado por otro generado al azar.

La mayor parte de la complejidad (tanto en coste computacional como en dificultad de diseño) de los algoritmos de Programación Genética está concentrada en la evaluación de la *función de adaptación*. La adaptación debe medir cómo de buena es una solución y ésta depende claramente del tipo de problema. Por ejemplo, en un problema de clasificación en bases de datos, la adaptación puede estar medida por el número de ejemplos bien clasificados. Evidentemente, no es tan obvio evaluar un elemento activo como es una fórmula matemática o un programa, y puede resultar en que la función de adaptación no sea determinística.

2.6. Programación Paralela

La potencia de los ordenadores ha ido aumentando con los años, pero está limitida por motivos físicos. Algunos de los más destacables de estos factores físicos son la frecuencia de reloj de los procesadores y la densidad de integración de los componentes electrónicos en procesadores, que se están acercando a los extremos de lo técnicamente posible, y en el que el calor disipado por la operación de los sistemas los puede llegar a destruir. La velocidad de los programas secuenciales están fuertemente limitados por este factor. La programación paralela se basa en el principio de dividir un algoritmo en tareas que se ejecutan en paralelo o concurrentemente en distintos procesos o procesadores, obteniendo una mejora de la eficiencia algorítimica que revierte en una ganancia de velocidad de ejecución.

Hay 2 tipos básicos de paralelismo que se pueden explotar para paralelizar un algoritmo secuencial que llevan a 2 clases extremas de programas paralelos:

- *Paralelismo de datos*: se basa en dividir la entrada de un programa en trozos a los cuales se aplica el mismo trabajo pero de forma de independiente en distintos procesos/procesadores.

- *Paralelismo funcional*: también conocido como *Paralelismo de tareas* o *Paralelismo de control*, consiste en asignar las distintas tareas de un algoritmo a distintos procesos/procesadores. El problema de hallar la mejor partición de tareas para ejecución paralela o concurrente es un problema NP-completo.

En la práctica la mayoría de los algoritmos presentan potencialmente ambos tipos de paralelismo. Esto implica que puede llegar a haber una cierta cantidad de posibles diseños paralelos para un mismo algoritmo, cuya eficiencia se verá también afectada por factores propios de la arquitectura donde se ejecutarán. De todas maneras, hay un límite a lo que puede mejorar un algoritmo al paralelizarse que viene dado por la llamada **Ley de Amdahl** [Amdahl, 1967]: *la ganancia de velocidad que se consigue al mejorar un recurso de un computador en un factor*

igual a p está limitada por la cota $p/(1+f(p-1))$*, donde* f *es la porción del tiempo en que dicha mejora no se utiliza en las aplicaciones que se ejecutan en el computador*. De esta forma, por mucho que mejoremos un recurso (es decir, en nuestro caso, por mucho que incrementemos el número de procesadores p), la mejora de velocidad no va a ser mayor que $1/f$.

Los programas paralelos necesitan de *computadores paralelos* para poder ser ejecutados. Un computador paralelo puede definirse de forma muy general como un conjunto de procesadores que trabajan cooperativamente para la resolución de problemas. Se pueden organizar diversas *arquitecturas paralelas* con múltiples procesadores comunicados entre sí mediante una red de interconexión (bus de un mismo ordenador, red ethernet, etc). Se pueden diferenciar 2 tipos básicos de arquitecturas paralelas:

- *Arquitecturas de memoria compartidas*: aquellas en las cuales los procesadores o procesos tienen un sólo espacio de memoria que comparten.
- *Arquitecturas de memoria distribuidas*: aquellas en las cuales los procesadores o procesos tienen distintos espacios de memoria independientes.

Otra clasificación para las arquitecturas de computadores viene dada por la *taxonomía de Flynn* [Flynn, 1972], que es independiente de la anterior división:

1. **SISD** (*Single Instruction, Single Data*, único flujo de instrucciones, único flujo de datos): son las máquinas con un único procesador que operan con un único flujo de instrucciones sobre datos almacenados en una memoria. Correspondientes a la arquitectura de von Neumann.

2. **MIMD**: (*Multiple Instruction, Multiple Data*, múltiples flujos de instrucciones, múltiples flujos de datos): son máquinas cuyos procesadores pueden ejecutar diferentes instrucciones sobre distintos conjuntos de datos.

3. **SIMD**: (*Single Instruction, Multiple Data*, único flujo de instrucciones, múltiples flujos de datos): son máquinas cuyos procesadores ejecutan la misma secuencia de instrucciones sobre distintos conjuntos de datos. Son arquitecturas especialmente diseñadas para soportar aplicaciones con alto nivel de paralelismo de datos, siendo un ejemplo notorio de este tipo de arquitectura son las GPU actuales que operan los mismos algoritmos sobre los píxeles de una imágen de forma independiente (ver Sección 6.1.1).

4. **MISD**: (*Multiple Instruction, Single Data*, múltiples flujos de instrucciones, único flujo de datos): son máquinas cuyos procesadores ejecutan diferentes instrucciones sobre el mismo conjuntos de datos. Son el menos común de los tipos de arquitectura de esta taxonomía, ya que los computadores MIMD y

SIMD suelen ser los más adecuados para la mayoría de aplicaciones y una arquitectura MISD puede ser emulada fácilmente por una MIMD.

El tipo concreto de arquitectura influye en el modo en que se debe de diseñar un programa paralelo para que se ejecute de forma eficiente en ella. Los tipos de arquitecturas no son incompatibles, obteniendo arquitecturas mixtas. El tipo de arquitectura influye en la eficiencia del mecanismo de comunicación entre tareas o procesos paralelos.

La comunicación entre tareas es fundamental a la hora de diseñar un programa paralelo. Hay 2 mecanismos básicos de comunicación entre tareas:

- *Uso de espacios de memoria compartidas*: sea mediante un espacio compartido de memoria real o simulación del mismo.

- *Paso de mensajes*: que se basa en el paso explícito de datos entre procesos/procesadores.

Al igual que en el caso de las arquitecturas paralelas, ambos tipos de comunicación no son incompatibles entre sí. La eficiencia del mecanismo de comunicación es un factor clave en las prestaciones de un programa paralelo.

Todo lo que hemos expuesto hasta el momento, unido a la dificultad añadida a la hora de depurar, convierte la programación paralela en una tarea que puede llegar a ser extremadamente árdua. Se suele aplicar una metodología de 4 pasos para el diseño de un algoritmo paralelo para un problema:

1. *Particionamiento*: el cómputo que se realiza y los datos sobre los cuales se aplica son descompuestos en pequeñas tareas. El objetivo es hallar el máximo paralelismo potencial. Se suele distinguir entre *particionamiento de dominio* y *particionamiento funcional*, respectivamente correspondientes a buscar el paralelismo de datos y el funcional.

2. *Comunicación*: se determina la comunicación necesaria entre las tareas para que se coordinen, diseñándose una estructura y algoritmos de comunicación adecuados. Las comunicaciones se clasifican según 4 criterios: locales/globales, estructuradas/no estructuradas, estáticas/dinamicas, y síncronas/asíncronas.

3. *Aglomeración*: se evalúan los costes de implementación y rendimiento de la estructura de tareas y comunicación obtenida en los pasos anteriores. En caso de ser necesario, diversas tareas individuales son combinadas en una única con el fin de mejorar el rendimiento o hacer menos costosa la implementación.

4. *Mapeado*: cada tarea se asigna a un procesador de manera que se maximice el uso de los procesadores a la vez que se minimicen los costes de comunicación. El problema de hallar el mejor mapeado de tareas para un conjunto de procesadores es un problema NP-completo.

El resultado de aplicar esta metodología en su forma más pura son algoritmos que crean y destruyen dinámicamente tareas. Un paradigma muy aplicado a la hora de crear programa paralelos es el SPMD (*Single Program Multiple Data*, único programa, múltiples datos) en el que cada proceso ejecuta el mismo programa pero con diferentes conjuntos de datos. Es uno de los más populares al sólo requerir escribir un único programa, siendo además muy adecuado para su uso en arquitecturas distribuidas usando como mecanismo de comunicación el paso de mensaje. El estándar MPI, ver Sección 2.7, es especialmente adecuado para desarrollara aplicaciones usando este paradigma.

2.7. MPI: la Interfaz para Paso de Mensajes

MPI son las siglas en inglés para Interfaz para Paso de Mensajes (*Message Passing Interface* y define un estándar para la comunicación entre procesos, siendo éste paradigma uno de los más utilizados a la hora de diseñar aplicaciones paralelas. Fue definida en los 80, aunque desde su creación se han llevado a cabo numerosos estudios de rendimiento tanto de cara a su implementación en las más diversas plataformas como para su mejora [Walker, 1994] [Bruck et al., 1995] [Gropp et al., 1996] [Foster et al., 1998] [Fagg et al., 2005]. El estándar es especialmente adecuado para máquinas de memoria distribuida aunque también puede utilizarse en máquinas de memoria compartida. Hay una gran cantidad de implementaciones tanto en el ámbito comercial como en iniciativas de código abierto: (Sun MPI[3], Intel MPI[4]) (OpenMPI[5], LAM[6], MPICH[7] y MPICH2[8]). Es un estándar *de facto* a la hora de programar aplicaciones paralelas, como puede atestiguar el hecho de que, aunque en principio la interfaz fue implementada en *C/C++* y *Fortran*, en la actualidad se puede hacer uso de ella desde la mayoría de los lenguajes de alto nivel que puede llegar a ser utilizados por la comunidad científica como *Python* [Dalcin et al., 2005] [Dalcín et al., 2008], *Perl*[9], *C#* [Willcock et al., 2002] [Gregor and Lumsdaine, 2008] o *Java* [Baker and Carpenter, 2000] [Baker et al., 2006]. Es más, la intefaz fue creada para un modelo de programación

[3] http://www.sun.com/software/products/clustertools/
[4] http://www.intel.com/cd/software/products/asmo-na/eng/308295.htm
[5] http://www.open-mpi.org/
[6] http://www.lam-mpi.org/
[7] http://www-unix.mcs.anl.gov/mpi/mpich1/
[8] http://www.mcs.anl.gov/research/projects/mpich2/
[9] http://search.cpan.org/~josh/Parallel-MPI-0.03/MPI.pm

un programa múltiples datos (*Single Process, Multiple Data* o *Single Program, Multiple Data*, SPMD), pero se está expandiendo para soportar operaciones otros modelos de programación paralela.

MPI trabaja siempre a nivel de procesos, que pueden ser mapeados o no a distintos procesadores, proporcionando capacidad para:

- **Organizar los procesos en grupos y organizar dichos grupos en topologías (rejillas, por ejemplo)**. Esto básicamente se consigue a través del concepto de *comunicador* que es una abstracción de un conducto a través de los cuales los procesos se comunican. MPI define funciones para la creación y gestión de comunicadores.

- **Sincronizar los procesos**. MPI proporciona funciones para sincronización de procesos tanto específicas como con operaciones síncronas.

- **Comunicar los procesos**. MPI proporciona una gran cantidad de funciones de comunicación entre procesos, tanto *proceso a proceso* como *globales* (entre un grupo de procesos) así como de funcionamiento *síncrono* como *asíncrono*.

Como se ha dejado entrever antes, MPI es un estándar definido independientemente del lenguaje en que se va a usar, por lo que la interfaz también incluye una definición de los tipos de datos para las comunicaciones independiente de la implementación específica y funcionalidad para definición y gestión de nuevos tipos de datos para los mensajes que se van a comunicar los procesos entre sí.

3. ENTRENAMIENTO DE LSSVM PARA REGRESIÓN

Las Máquinas de Vectores Soporte de Mínimos Cuadrados (*Least Square Support Vector Machines*, LSSVM) [Suykens et al., 2002] representan el estado del arte en métodos *kernel* para problemas de aproximación funcional o regresión. Surgieron como alternativa a las Máquinas de Vectores de Soporte para Regresión (SVR), solucionando algunos de sus problemas de eficiencia, de entrenamiento y de tamaño de los modelos. Estos modelos están íntimamente relacionados con los Procesos Gaussianos para Regresión, aunque con un enfoque mucho más basado en la teoría de optimización, lo que lleva a un planteamiento más práctico de los mismos. Sin embargo, las LSSVM adolecen también de diversos problemas, algunos generales a todos los métodos *kernel* y otros específicamente suyos. En este capítulo se presenta una metodología para una optimización más eficiente los modelos LSSVM para regresión con respecto al error de validación cruzada tanto en tiempo de ejecución como precisión. Esta metodología se basa en la presentación de expresiones para las derivadas parciales del error de validación cruzada con respecto a los parámetros de los modelos LSSVM, lo que permite la aplicación de algoritmos como el Gradiente Conjugado, que no son aplicados normalmente para optimización de estos modelos. Además, y también como parte fundamental de la metodología, se presentan heurísticas para inicializar a valores razonablemente buenos los parámetros de los modelos LSSVM, lo que mejora la eficiencia de la optimización. El buen rendimiento de la metodología es demostrado frente a otros algoritmos propuestos en la bibliografía para entrenar una LSSVM en sus distintos aspectos.

3.1. Métodos kernel para regresión

Aunque las Máquinas de Vectores Soporte (*Support Vector Machine*, SVM) fueron concebidas y desarrolladas para resolver problemas de clasificación, otras técnicas han aparecido en la literatura que hacen uso del llamado *kernel trick*

[Scholkopf and Smola, 2001] tanto para clasificación como para regresión. Hoy día estas técnicas son referidas en general como métodos *kernel*. Los métodos *kernel* se definen por operaciones sobre los valores de una función de *kernel* sobre los datos, ignorando la estructura de la entrada, y evitando así el problema de la maldición de la dimensionalidad [Bellman, 1966]. Su mayor problema es la eficiencia cuando el número de datos crece.

Para problemas de regresión, el primer método *kernel* propuesto en la bibliografía fue el de los Vectores Soporte para Regresión (*Support Vector Regression*, SVR) [Scholkopf and Smola, 2001]. Las Máquinas de Vectores Soporte de Mínimos Cuadrados (*Least Square Support Vector Machines*, LSSVM) son una modificación de la formulación de SVR estándar creada para superar sus principales inconvenientes [Suykens et al., 2002]. Concretamente, el problema de optimización resultante tiene la mitad de parámetros que cuando se utilizan SVRs, y el modelo se puede optimizar resolviendo un sistema de ecuaciones lineal en vez de un problema de optimización cuadrática.

Tanto SVR como LSSVM han sido aplicados a predicción de series temporales con resultados prometedores como puede verse en [Müller et al., 2000], [Tay, 2001] y [Thiessen and van Brakel, 2003] para SVR, y en [Van Gestel et al., 2001] y [Xu, 2005] para el caso de LSSVM. Sin embargo, ambas técnicas tienen algunos problemas comunes:

1. **No existe ningún método para seleccionar la función de *kernel* óptima para un problema concreto.** Aunque existen numerosos trabajos en la bibliografía que investigan sobre los métodos *kernel*, la mayor parte de ellos usan el *kernel* Gaussiano o *kernels* basados en él para problemas de regresión y predicción de series temporales, debido a la interpolación suave que obtienen.

2. **La optimización de los parámetros es costosa computacionalmente.** El proceso de optimización requiere de la evaluación de algún procedimiento de validación cruzada (CV) [Ying and Keong, 2004] [An et al., 2007] o algún criterio Bayesiano [Van Gestel et al., 2001] con una complejidad $O(N^3)$, donde N es el número de puntos de entrenamiento. En el caso de SVR, un interesante estudio en [Cherkassky and Ma, 2004] da una guía de cómo fijar los valores de los hiperparámetros. Para el caso de LSSVM, un reciente estudio en [Liitiäinen et al., 2007] presenta una interesante aproximación al entrenamiento de LSSVM con *kernel* Gaussiano basada en una estimación no paramétrica del ruido (*Nonparametric Noise Estimation*, NNE), la cual trata de estimar un límite superior a la precisión de aproximación que puede ser obtenida por cualquier modelo para unos datos concretos.

3. **Los modelos generados pueden ser grandes,** ya que incluyen todos los datos de entrenamiento en su interior. Para aliviar este gran problema, un método de

poda para reducir el número de ejemplo retenidos podría ser usado a posteriori sobre los modelos generados [de Kruif and de Vries, 2003], [Cawley and Talbot, 2002].

En la literatura, las funciones *kernel* usadas son casi todas variantes de las Funciones de Base Radial (RBF) [Rojas et al., 2000b] y el análisis del problema se centra principalmente en la selección de características, es decir, qué características deben de tenerse en cuenta para el problema [Müller et al., 2000] [Rubio et al., 2008]. En la mayoría de los casos, los parámetros son optimizados con respecto al error de validación cruzada de los modelos, aunque otros criterios se pueden utilizar como la verosimilitud Bayesiana (Bayesian Likelihood) [Scholkopf and Smola, 2001] [Van Gestel et al., 2001] o alguna cota al error de entrenamiento [Lendasse et al., 2005]. En la implementación más popular de LSSVM para Matlab, LSSVMlab[1], tanto la optimización con respecto al error de validación cruzada como la verosimilitud Bayesiana están disponibles. En el primer caso, el software busca en el espacio de parámetros por medio de un algoritmo de búsqueda en rejilla (grid), que es muy costoso computacionalmente para rejillas de grandes tamaños, valores altos de orden de validación cruzada y gran número de parámetros. La excepción es el caso especial del error de validación cruzada Leave-One-Out (LOO), que está implementado en base a la inversión de la matriz de *kernel* para los datos de entrenamiento. En [Ying and Keong, 2004] y [An et al., 2007] se presentan expresiones para la evaluación del error de validación cruzada eficientes y de complejidad computacional independiente del orden de la validación cruzada.

En este capítulo se presenta:

1. Un estudio en el que se enfatiza en la optimización precisa y eficiente de los parámetros de LSSVM con respecto al error de validación cruzada de orden arbitrario haciendo uso de las derivadas parciales.

2. Un método para estimar, tanto el parámetro σ del *kernel* Gaussiano, como el hiperparámetro de regularización del proceso de optimización de un modelo LSSVM, basado en el uso de un método de estimación no paramétrica del ruido e información extraída de los datos a modelar.

Los problemas a los que se aplican los métodos en los ejemplos son de predicción de series temporales, debido a su gran relevancia, aunque realmente se puede hacer uso de los métodos descritos en cualquier problema de regresión.

[1] `http://www.esat.kuleuven.ac.be/sista/lssvmlab/`

3.2. Máquinas de Vectores de Soporte de Mínimos Cuadrados (LSSVM)

Dado un conjunto de muestras de una función $\{(x_1,y_1),\ldots,(x_N,y_N)\} \subset \mathbb{R}^d \times \mathbb{R}$, donde N es el número de muestras y d es la dimensionalidad de la entrada, el modelo LSSVM clásico intenta aproximar una función que relaciona las entradas X con las salidas Y resolviendo el siguiente problema de optimización:

$$\begin{array}{rcl} \min\limits_{w\in\mathbb{R}^d,\ b,e_i\in\mathbb{R}} \tau(w,b,e) & = & \frac{1}{2}\|w\|^2+\gamma\frac{1}{2}\sum\limits_{i=1}^{N} e_i^2, \\ (y_i-(\langle w,x_i\rangle+b)) & = & e_i,\ \ \forall i=1,\ldots,N \end{array} \tag{3.1}$$

donde τ es la función a optimizar, la cual depende del vector de pesos $w \in \mathbb{R}^d$ y el valor b, que son los parámetros de aproximador lineal; $\gamma > 0$ es un factor de regularización y e_i es el error cometido al aproximar la i-ésima muestra ($e = \{e_1,e_2,\cdots,e_N\}$). Es el típico problema de optimización con restricciones de una función derivable que puede afrontarse usando el método de los Multiplicadores de Lagrange y que conduce a la resolución del siguiente sistema lineal:

$$\left[\begin{array}{c|c} 0 & 1_N^T \\ \hline 1_N & \Omega+I/\gamma \end{array}\right]\left[\begin{array}{c} b \\ \hline \alpha \end{array}\right]=\left[\begin{array}{c} 0 \\ \hline y \end{array}\right] \tag{3.2}$$

donde $\Omega_{ij} = \langle x_i,x_j\rangle$ es el producto escalar entre pares de puntos de entrada, I la matriz identidad y α es el vector de multiplicadores de Lagrange. Si el producto escalar en el *espacio de entrada* es sustituido por un producto escalar en un *espacio de características* dado por la función de *kernel* $k(x,x';\Theta) = \langle\phi(x;\Theta),\phi(x';\Theta)\rangle$, donde ϕ es la función que mapea puntos del espacio de entrada a puntos del espacio de características, entonces $\Omega_{ij} = k(x_i,x_j;\Theta)$ y la función modelada se puede expresar como:

$$f(x) = \sum_{i=1}^{N} \alpha_i k(x,x_i) + b. \tag{3.3}$$

3.2.1. Evaluación del error de validación cruzada de orden l

Con el objetivo de evitar que un modelo LSSVM sobreajuste a los datos de los cuales aprende, es importante medir su capacidad de generalización, es decir, su rendimiento cuando aproxima datos que no han sido vistos en el entrenamiento. Uno de los métodos más comunes para obtener un modelo con buena capacidad de generalización consiste en minimizar el error de validación cruzada en vez del error de entrenamiento. El error de validación cruzada de orden l para un modelo se obtiene dividiendo el conjunto de datos en l subconjuntos de aproximadamente igual tamaño, y utilizando, de forma iterativa, cada uno de esos l subconjuntos

como datos de validación, dejando el resto como datos de entrenamiento. Por lo tanto, un total de l modelos son entrenados, y la media de los l errores de validación (por cada subconjunto) es el error de validación cruzada de orden l.

En [An et al., 2007] se presenta un método de coste reducido para la evaluación del error de validación cruzada de orden l para LSSVM. Para calcular el error de validación cruzada de orden l se puede usar la siguiente expresión:

$$MSE_{l\text{-fold}} = \frac{1}{N}\sum_{m=1}^{l}\sum_{j=1}^{|\beta^{(m)}|}{\beta_j^{(m)}}^2 \tag{3.4}$$

donde $\beta = [\beta^{(1)}, \beta^{(2)}, \cdots, \beta^{(l)}]^T$, y cada $\beta^{(m)}$ es definido como $\beta^{(m)} = C_{mm}^{-1}\alpha^{(m)}$. Es imporatnte destacar que aquí, $\alpha = [\alpha^{(1)}, \alpha^{(2)}, \cdots, \alpha^{(l)}]^T$ no corresponde al de la Ecuación (3.2), sino que viene dado por:

$$\alpha = Cy \tag{3.5}$$

$$C = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1l} \\ C_{12}^T & C_{22} & \cdots & C_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ C_{l2}^T & C_{2l}^T & \cdots & C_{ll} \end{bmatrix} = K_\gamma^{-1} + \frac{1}{d}K_\gamma^{-1}1_n 1_n^T K_\gamma^{-1} \tag{3.6}$$

$$d = -1_n^T K_\gamma^{-1} 1_n \tag{3.7}$$

$$K_\gamma = K + I/\gamma \tag{3.8}$$

$$K_{ij} = k(x_i, x_j; \Theta) \tag{3.9}$$

donde I es la matriz identidad, Θ es el vector de parámetros del *kernel* k y γ el factor de regularización. Para facilitar la lectura, se definen la matriz C, el valor d, la matriz K_γ y la matriz K en las Ecuaciones (3.6), (3.7) (3.8) y (3.9). La matriz C se particiona en l^2 submatrices de tamaño $N/l \times N/l$, que son las utilizadas para el cálculo de los $\beta^{(m)}$ de la Ecuación (3.4). El coste computacional es independiente del orden de validación cruzada l, siendo dominado por el coste de inversión de la matriz K_γ.

3.2.2. Derivadas parciales del error de validación cruzada de orden l

En las sub-secciones previas hemos declarado que nuestro principal objetivo a la hora de optimizar un modelo LSSVM es minimizar el error de validación cruzada para obtener buenas capacidades de generalización. Con el fin de conseguir dicha minimización, necesitamos las expresiones de las derivadas parciales del error de validación cruzada de orden l (Ecuación 3.4) con respecto a un parámetro p dado (que puede ser el factor de regularización γ o un parámetro del *kernel*). Se obtuvieron las siguientes expresiones:

$$\frac{\partial MSE_{l\text{-fold}}}{\partial p} = \frac{2}{N}\sum_{m=1}^{l}\sum_{j=1}^{|\beta^{(m)}|}\beta_j^{(m)}\left[\frac{\partial \beta^{(m)}}{\partial p}\right]_j \tag{3.10}$$

$$\frac{\partial \beta^{(m)}}{\partial p} = \frac{\partial C_{mm}^{-1}}{\partial p}\alpha^{(m)} + C_{mm}^{-1}\frac{\partial \alpha^{(m)}}{\partial p} \tag{3.11}$$

$$\frac{\partial \alpha^{(m)}}{\partial p} = \frac{\partial C_{mm}}{\partial p}y^{(m)} \tag{3.12}$$

$$\frac{\partial C_{mm}^{-1}}{\partial p} = -C_{mm}^{-1}\frac{\partial C_{mm}}{\partial p}C_{mm}^{-1} \tag{3.13}$$

$$\frac{\partial C}{\partial p} = \begin{bmatrix} \frac{\partial C_{11}}{\partial p} & \frac{\partial C_{12}}{\partial p} & \cdots & \frac{\partial C_{1l}}{\partial p} \\ \frac{\partial C_{12}^T}{\partial p} & \frac{\partial C_{22}}{\partial p} & \cdots & \frac{\partial C_{2l}}{\partial p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C_{l2}^T}{\partial p} & \frac{\partial C_{2l}^T}{\partial p} & \cdots & \frac{\partial C_{ll}}{\partial p} \end{bmatrix} \tag{3.14}$$

$$\frac{\partial C}{\partial p} = \frac{\partial K_\gamma^{-1}}{\partial p} + \frac{\partial d^{-1}}{\partial p}K_\gamma^{-1}1_n 1_n^T K_\gamma^{-1}\cdots \tag{3.15}$$

$$+\frac{1}{d}\left(\frac{\partial K_\gamma^{-1}}{\partial p}1_n 1_n^T K_\gamma^{-1} + K_\gamma^{-1}1_n 1_n^T\frac{\partial K_\gamma^{-1}}{\partial p}\right) \tag{3.16}$$

$$\frac{\partial d^{-1}}{\partial p} = \frac{1}{d^2}1_n^T\frac{\partial K_\gamma^{-1}}{\partial p}1_n \tag{3.17}$$

$$\frac{\partial K_\gamma^{-1}}{\partial p} = -K_\gamma^{-1}\frac{\partial K_\gamma}{\partial p}K_\gamma^{-1} \tag{3.18}$$

Por lo tanto, y como era de esperar, la derivada parcial del error de validación cruzada de orden l con respecto al parámetro p depende de $\frac{\partial K_\gamma}{\partial p}$, la derivada parcial de $K_\gamma = K + I/\gamma$. Para cada $p = \theta \in \Theta$, la derivada depende de la función *kernel* k, pero para $p = \gamma$ es $\frac{\partial K_\gamma}{\partial p} = -I/\gamma^2$.

La evaluación de la derivada parcial implica la inversión de la matriz K_γ, lo que tiene una complejidad computacional de $O(N) = N^3$ con métodos exactos, y depende también de la derivada parcial de la función *kernel* con respecto a sus parámetros, la cual debe de ser proporcionada para cada *kernel* en particular.

Las ecuaciones anteriores, en su conjunto, pueden ser utilizadas en un esquema de Gradiente Conjugado para optimizar tanto el factor de regularización γ como los parámetros del *kernel* de un modelo LSSVM. A este respecto, es importante mencionar que en [Suykens et al., 2002] se recomienda optimizar en diferentes niveles los parámetros del *kernel* y el factor de regularización γ. Como la

convergencia al óptimo global no está garantizada, se recomienda realizar varias optimizaciones partiendo de distintos puntos de inicio. Además, en la Sección 3.4 se proporcionan algunas sugerencias heurísticas para obtener los puntos de inicio con el fin de tener una ruta más rápida al óptimo global o, al menos, a un *buen* óptimo.

3.3. LSSVM sin sesgo

Frecuentemente se prefiere usar una versión sin sesgo de LSSVM para aproximación funcional. La eliminación del sesgo (b en la Ecuación 3.1) simplifica las expresiones de LSSVM si se considera que la función a modelar tiene media cero, una condición que puede ser impuesta sin pérdida de generalidad. Dado un conjunto de muestras de una función $\{(x_1,y_1),\ldots,(x_N,y_N)\} \in \mathbb{R}^d \times \mathbb{R}$, donde N es el número de muestras y d es la dimensionalidad de la entrada, el modelo LSSVM sin sesgo intenta aproximar una función que relaciona las entradas X con las salidas Y resolviendo el siguiente problema de optimización:

$$\begin{array}{rcl} \min\limits_{w\in\mathbb{R}^d, e_i\in\mathbb{R}} \tau(w,e) & = & \frac{1}{2}\|w\|^2 + \gamma\frac{1}{2}\sum\limits_{i=1}^{N} e_i^2, \\ (y_i - (\langle w, x_i\rangle)) & = & e_i, \ \ \forall i = 1,\ldots,N \end{array} \tag{3.19}$$

lo que lleva a usar nuevamente los multiplicadores de Lagrange α y, finalmente, a resolver el sistema lineal:

$$[\Omega + I/\gamma]\,[\alpha] = [y] \tag{3.20}$$

donde $\Omega_{ij} = \langle x_i, x_j \rangle$ es el producto escalar entre pares de puntos de entrada e I la matriz identidad. Si el producto escalar en el *espacio de entrada* es sustituido por un producto escalar en un *espacio de características* dado por la función de *kernel* $k(x,x';\Theta) = \langle \phi(x;\Theta), \phi(x';\Theta)\rangle$, donde ϕ es la función que mapea puntos del espacio de entrada a puntos del espacio de características, entonces $\Omega_{ij} = k(x_i,x_j;\Theta)$ y la función modelada se puede expresar como:

$$f(x) = \sum_{i=1}^{N} \alpha_i k(x,x_i) \tag{3.21}$$

3.3.1. Evaluación del error de validación cruzada de orden l

El método de coste reducido para la evaluación del error de validación cruzada de orden l para el caso especial de LSSVM sin sesgo es:

$$MSE_{l\text{-fold}} = \frac{1}{N}\sum_{m=1}^{l}\sum_{j=1}^{|\beta^{(m)}|}{\beta_j^{(m)}}^2 \tag{3.22}$$

$$\beta^{(m)} = C_{mm}^{-1}\alpha^{(m)} \tag{3.23}$$

$$C = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1l} \\ C_{12}^T & C_{22} & \cdots & C_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ C_{l2}^T & C_{2l}^T & \cdots & C_{ll} \end{bmatrix} \tag{3.24}$$

$$C = K_\gamma^{-1} \tag{3.25}$$

$$K_\gamma = K + I/\gamma \tag{3.26}$$

$$K_{ij} = k(x_i, x_j; \Theta) \tag{3.27}$$

$$\alpha = \left[\alpha^{(1)}, \alpha^{(2)}, \cdots, \alpha^{(l)}\right]^T \tag{3.28}$$

donde α es el mismo de la Ecuación (3.20), las matrices K_γ y K son las mismas que las de las Ecuaciones (3.8) y (3.8), pero la matriz C, en cambio, se define en este caso como la inversa de K_γ, ver Ecuación (3.25).

3.3.2. Derivadas parciales del error de validación cruzada de orden l

De nuevo, para el caso especial de LSSVM sin sesgo, las derivadas parciales del error de validación cruzada de orden l (3.22) con respecto a un parámetro p (que puede ser γ o un parámetro del *kernel*) quedan como:

$$\frac{\partial MSE_{l\text{-fold}}}{\partial p} = \frac{2}{N}\sum_{m=1}^{l}\sum_{j=1}^{|\beta^{(m)}|}\beta_j^{(m)}\left[\frac{\partial \beta^{(m)}}{\partial p}\right]_j \tag{3.29}$$

$$\frac{\partial \beta^{(m)}}{\partial p} = \frac{\partial C_{mm}^{-1}}{\partial p}\alpha^{(m)} + C_{mm}^{-1}\frac{\partial \alpha^{(m)}}{\partial p} \tag{3.30}$$

$$\frac{\partial \alpha}{\partial p} = \frac{\partial K_\gamma^{-1}}{\partial p}y \tag{3.31}$$

$$\frac{\partial K_\gamma^{-1}}{\partial p} = -K_\gamma^{-1}\frac{\partial K_\gamma}{\partial p}K_\gamma^{-1} \tag{3.32}$$

3.4. Método heurístico para la estimación de los parámetros de LSSVM

El objetivo cuando se trata de optimizar los parámetros de un modelo a partir de un conjunto dado de muestras de entrada/salida es optimizar la capacidad de

generalización del modelo, y un camino para conseguir esto es minimizar el error de validación cruzada de orden l. Para ello, en esta sección se hace uso de un método de búsqueda local como es el Gradiente Conjugado. Este método es capaz de encontrar el mínimo más cercano a una asignación tentativa inicial de los parámetros a optimizar, usando la información de las derivadas parciales del modelo. En el caso de las máquinas de vectores soporte de mínimos cuadrados, LSSVM, estos parámetros son el factor de regularización, también llamado hiperparámetro γ y los parámetros de la función de *kernel*. La bondad del mínimo encontrado depende en gran medida de los valores iniciales de búsqueda. En esta sección se propone un método heurístico, obtenido a partir de nuestra experiencia, para asignar dichos valores. Dicho método está basado en una estimación no parámetrica del error de aproximación (ruido), concepto que abordaremos a continuación.

3.4.1. Estimación no paramétrica del ruido: Tests Delta y Gamma

La estimación del ruido existente en un conjunto de pares de entrada/salida se ha comprobado ser especialmente útil para problemas de regresión ya que proporciona una cota mínima al error de entrenamiento, el cual puede ser usado para determinar cuánto de preciso puede llegar a ser un modelo entrenado para dichos datos. En otras palabras, cuando un modelo entrenado para unos datos obtiene una precisión superior al del ruido presente en dichos datos, se puede concluir que el modelo esta sobreajustado a los datos de entrenamiento y, por tanto, debe descartarse. Esta estimación del ruido también puede ser usada como criterio de selección de las entradas más relevantes para el modelo, como se realiza en [Lendasse et al., 2006]. En esta sección se describe brevemente los dos métodos no paramétricos de estimación del ruido más conocidos.

El Test Delta para un conjunto de muestras de entrada/salida de una función $(x_1, y_1), \ldots, (x_N, y_N) \subset \mathbb{R}^d \times \mathbb{R}$, donde N es el número de muestras y d es la dimensionalidad de la entrada, puede ser definido como:

$$\delta_{N,k} = \frac{1}{2N} \sum_{i=1}^{N} (y_i - y_{nn[i,k]})^2 \tag{3.33}$$

donde $nn[i,k]$ es el índice del k-ésimo vecino más cercano a x_i usando una medida de distancia, usualmente la distancia Euclídea. Dado que $\delta_{N,1} \approx \sigma_e^2$, donde σ_e^2 es la varianza del ruido en la salida, es posible usar $\delta_{N,1}$ como estimación del mejor error cuadrático medio que puede ser obtenido por un modelo sobre el conjunto de datos de entrenamiento sin sobreajuste.

En [Liitiäinen et al., 2007] se describen las limitaciones del Test Delta para datos de alta dimensionalidad (vectores de entrada de dimensionalidad superior a 4) en cuando a robustez y confiabilidad del método. Los autores recomiendan usar el Test Gamma en esos casos, el cual es un método más complejo para la

estimación del ruido en conjuntos de datos de entrada/salida. Usando la misma notación de más arriba, definamos Γ como el momento empírico de grado ρ de los datos para el k-ésimo vecino, definido como:

$$\Gamma_{N,\rho,k} = \frac{1}{N}\sum_{i=1}^{N} \|x_i - x_{nn[i,k]}\|^{\rho} \tag{3.34}$$

Entonces, el estimador del Test Gamma de la varianza del ruido de la salida σ_e^2 para $k \geq 2$ puede calcularse como:

$$\sigma_e^2 \approx E_k[\delta_{N,l}] - \frac{E_k[\Gamma_{N,2,l}]\sum_{l=1}^{k}(\Gamma_{N,2,l} - E_k[\Gamma_{N,2,l}])(\delta_{N,l} - E_k[\delta_{N,l}])}{\sum_{l=1}^{k}(\Gamma_{N,2,l} - E_k[\Gamma_{N,2,l}])^2} \tag{3.35}$$

donde $E_k[\delta_{N,l}] = \frac{1}{k}\sum_{l=1}^{k}\delta_{N,l}$, $E_k[\Gamma_{N,2,l}] = \frac{1}{k}\sum_{l=1}^{k}\Gamma_{N,2,l}$ y $\delta_{N,l}$ se definió en la Ecuación (3.33). Como puede verse, la definición del Test Gamma depende de la del Test Delta.

Se puede demostrar que ambos tests convergen al valor σ_e^2 al aumentar N [Pi and Peterson, 1994][Evans, 2002]. Por lo tanto, se necesita un número suficientemente alto de muestras de entrada/salida con el fin de obtener un valor fiable en el resultado de los tests.

3.4.2. Valor inicial para el hiperparámetro γ basado en la Estimación No-Paramétrica del Ruido

El factor de regularización γ de las Ecuaciones (3.1) y (3.19), también conocido como el hiperparámetro γ, es el que define el compromiso entre suavidad y precisión del modelo. Por suavidad de un modelo se entiende que el valor de sus derivadas parciales no es demasiado alto. Un modelo en el que la derivada parcial con respecto a una variable sea muy alto, sufrirá cambios bruscos en su salida con respecto al cambio en dicha variable. Cuanto mayor sea el factor de regularización mayor será la importancia que se le da en el proceso de minimización al error del modelo. Valores excesivamente grandes de γ provocan que el modelo se sobreajuste a los datos de entrenamiento (derivadas altas, es decir, modelo poco suave), mermando su capacidad de generalización. Por lo tanto, es de vital importancia dar un valor adecuado al factor de regularización.

Con el fin de proporcionar una expresión general para el factor de regularización, la varianza del ruido debe ser tenida en cuenta. También debemos tener en cuenta que, si se multiplican los datos de entrenamiento por un factor de escala, los pesos obtenidos en la Ecuación (3.19) se escalarán en concordancia. Por otro lado, para el término con el error del modelo, es deseable tener una estimación de su valor óptimo. Es evidente que para un rango de salida (factor de escala) un modelo creado de datos sin ruido puede obtener mejor precisión que uno obtenido de

datos muy ruidosos. Y, precisamente, tanto el Test Delta como el Gamma pueden ayudarnos en dicha estimación.

Por lo tanto, proponemos el siguiente valor heurístico inicial γ_h para el factor de regularización:

$$\gamma_h = \frac{\sigma_y^2}{\hat{\sigma}_e^2} \tag{3.36}$$

donde $\hat{\sigma}_e^2$ se puede estimar de los datos usando NNE:

$$\hat{\sigma}_e^2 = \begin{cases} \text{usar Ecuación (3.33)} & \text{si } \textit{dimensionalidad de } X \leq 4 \\ \text{usar Ecuación (3.35)} & \text{en otro caso} \end{cases} \tag{3.37}$$

El problema de optimización para un modelo LSSVM dado por la Ecuación (3.19) puede reescribirse como:

$$\begin{array}{rcl} \min\limits_{w,e_i} \tau(w,e) & = & \frac{1}{2}\frac{1}{\sigma_y^2}\|w\|^2 + \frac{1}{2}\frac{1}{\hat{\sigma}_e^2}\sum\limits_{i=1}^{N} e_i^2, \\ y_i - \langle w, \phi(x_i;\Theta)\rangle & = & e_i, \ \ \forall i = 1,\ldots,N \end{array} \tag{3.38}$$

En el peor caso, los datos son puro ruido, por lo que $\hat{\sigma}_e^2 = \sigma_y^2$ y se puede fijar el valor mínimo heurístico de γ como $\gamma_{min} = 1$. Un rango de búsqueda de γ podría ser $[\gamma_{min}, \gamma_h]$, pero se pueden usar valores cercanos a γ_h si se confía en el resultado del NNE. En [Jones et al., 2007] se demuestra que el ruido en los vectores de entrada provoca que el NNE aproxime la varianza del error efectivo en vez de la varianza del error en la salida. La varianza del error efectivo es mayor que la varianza del ruido, pero una mejor estimación no puede ser realizada.

Debido a que en el caso de series temporales se tiene que generar los vectores de entrada/salida de las propias series, los vectores de entrada siempre vendrán contaminados por el mismo ruido que hay en la salida. Por lo tanto, es recomendable buscar los valores de γ en un rango centrado en γ_h.

3.4.3. Estimación del valor inicial para el parámetro σ del *kernel* Gaussiano basado en los datos de entrenamiento

Consideraremos aquí el caso más común de modelo LSSVM para regresión, es decir, el modelo LSSVM con *kernel* Gaussiano, cuya ecuación es:

$$k(x,x';\sigma) = \exp\left(-\frac{1}{\sigma^2}\|x-x'\|^2\right) \tag{3.39}$$

El valor de σ está relacionado con las distancias entre pares de puntos de entrenamiento y la suavidad de la interpolación del modelo [Rojas et al., 2000b]. Como regla general, cuanto mayor sea σ, más suave será la interpolación entre 2 puntos

consecutivos. Un rango de búsqueda razonable para σ es $[\sigma_{min}, \sigma_{max}]$, donde σ_{min} es la distancia mínima (no nula) y σ_{max} es la distancia máxima entre 2 puntos de entrenamiento.

3.5. Experimentos

3.5.1. Experimentos para la comprobación de las heurísticas de inicialización del parámetros

Con el fin de hacer un estudio comparativo justo y riguroso, se ha seleccionado un conjunto diverso de series temporales para cubrir la mayor cantidad de situaciones practicas:

- Series temporales deterministas no lineales discretas y continuas.
- Series temporales estocásticas lineales y no lineales.
- Series temporales reales.
- Series temporales de competiciones.

Una serie determinista (es decir, no estocástica) no lineal tiene que presentar algún comportamiento caótico para no ser un problema de aproximación trivial (es decir, no debe diverger, ni converger a un valor o un ciclo). Hay muchos ejemplos de series deterministas en la literatura, y cuatro han sido seleccionadas para la comparativa:

- *Hénon*: el mapa de Hénon es uno de los sistemas dinámicos más estudiados. El mapa de Hénon canónico recorre puntos en el plano siguiendo la siguiente Ecuación (3.40) [Hénon, 1976].

$$\begin{array}{rcl} x_{n+1} & = & y_n + 1 - 1{,}4x_n^2 \\ y_{n+1} & = & 0{,}3x_n \end{array} \tag{3.40}$$

- *Logistic*: el mapa logístico es un modelo demográfico que fue popularizado por [May, 1976] como ejemplo de un sistema no lineal simple que exhibe un complejo comportamiento caótico. Se obtiene de la Ecuación (3.41).

$$y_t = 4y_{t-1}(1 - y_{t-1}) \tag{3.41}$$

- *Mackey-Glass*: la serie de Mackey-Glass se aproxima a partir de la Ecuación diferencial (3.42) [Mackey and Glass, 1977]. Es un benchmark ampliamente usado para medir la capacidad de generalización de los métodos de predicción de series temporales. La serie es continua y se obtiene integrando la Ecuación

(3.42) con un método numérico como el Runge-Kutta de cuarto orden. Los datos de esta serie se obtuvieron del fichero `mgdata.dat`, incluido en la *Fuzzy Logic Toolbox*[2] del software Matlab.

$$\frac{dx(t)}{dt} = \frac{0{,}2x(t-\tau)}{1+x^{10}(t-17)} - 0{,}1x(t) \tag{3.42}$$

Las series estocásticas lineales y no lineales del presente estudio consisten en series deterministas con algún ruido asociado:

- *AR(4)*: una serie auto-regresiva de cuarto orden[3].
- *S.T.A.R.*: un modelo *Smooth Transition AutoRegressive* consiste en dos modelos AR enlazados por una función de transición. La serie no lineal se ha generado de la Ecuación (3.43), de la misma forma que se realiza en [Berardi and Zhang, 2003], con una varianza del ruido de $\sigma^2 = 5.0\text{e-}2$.

$$\begin{aligned} y(t) = {} & 0{,}3y(t-1) + 0{,}6y(t-2) + \\ & + \frac{0{,}1-0{,}9y(t-1)+0{,}8y(t-2)}{1+e^{-10y(t-1)}} + N(0,\sigma^2) \end{aligned} \tag{3.43}$$

Las series reales son difíciles de modelar, presentando normalmente mucho ruido y propiedades no lineales:

1. *Sunspots*: el número de manchas solares por mes desde 1700 a 2005[4]. Esta serie caótica ha sido muy estudiada, pero tiene muchas propiedades locales, ruido e incluso zonas no predecibles con el conocimiento previo.
2. *London*: temperatura media mensual en Londres, desde enero de 1659 a octubre de 2007[5].
3. *Electric*: datos de la carga del tendido eléctrico en California[6] (los datos originales están muestreados por horas).

Finalmente, las series temporales de las competiciones son frecuentemente usadas como benchmark en la literatura. Un par de series de la competición de Santa Fe [Weigend and Gershenfeld, 1994] fueron añadidas a los experimentos: *Laser* y *Computer*.

[2] ver `http://www.mathworks.com/products/fuzzylogic/?BB=1`
[3] disponible en `http://www.robjhyndman.com/TSDL/misc/simar4.dat`
[4] en `ftp://ftp.ngdc.noaa.gov/STP/SOLAR_DATA/SUNSPOT_NUMBERS/MONTHLY.PLT`
[5] datos de `http://hadobs.metoffice.com/hadcet/cetml1659on.dat`
[6] proporcionados en `http://www.ucei.berkeley.edu/CSEM/datamine/ISOdata/`

Como paso de preprocesamiento, la tendencia se eliminó cuando era necesario y los datos fueron normalizados para que tuvieran media 0 y varianza 1. Con el fin de aplicar LSSVM al modelado de series temporales, es necesario obtener un conjunto de vectores de E/S a partir de la serie. Para ello, es necesario conocer los regresores significativos para cada conjunto de datos. Para las series de las cuales se conocía el proceso de generación, el conjunto óptimo de regresores fue directamente usado (*Hénon, Logistic, AR(4), S.T.A.R.*). Para el resto de series se aplicó un método de selección de características, basado en el test Gamma [A.Sorjamaa et al., 2007], que consiste en la evaluación de los vectores E/S creados con los regresores de $t-1$ a $t-20$, y la selección del subconjunto más prometedor, es decir, el de menor varianza del ruido. El uso de otras herramientas de selección de variables podría haber aportado otras propiedades útiles de la series, pero debemos recordar que la selección de características no es nuestra prioridad en estos experimentos. En la Tabla 3.1 se resume las acciones llevadas a cabo para cada serie, las columnas son:

1. F.S.: selección de características. *"No"*, significa que el modelo se obtuvo de la literatura y *"Sí"*, significa que una selección de características basada en el test Gamma fue llevada a cabo.

2. Tend.: diferenciación de la serie para eliminar tendencia. *"Sí"* significa que la serie tuvo que diferenciarse para quitarle la tendencia y *"No"* que no fue necesario.

3. Datos: número de datos de entrenamiento usados en el experimento.

En cada caso, 500 valores de la serie se usaron para entrenamiento, excepto en el caso de Sunspots que no tiene suficientes muestras. En el caso de *Hénon*, *Logistic* y *Mackey-Glass*, que son series limpias de ruido, se añadió de forma controlada ruido Gaussiano aditivo de varianza 0,01.

Los experimentos fueron realizados para probar las heurísticas propuestas para inicializar el hiperparámetro γ y el parámetro σ del *kernel* Gaussiano en el modelo LSSVM sin sesgo para regresión. Para cada conjunto de datos, y un valor de σ fijado a la mitad del rango de búsqueda recomendado, el error de validación cruzada de orden 10 para modelos LSSVM fue calculado para un amplio rango de valores de γ. Las gráficas resultantes del logaritmo del error de validación cruzada frente al logaritmo de γ se muestran de la Figura 3.1 a la 3.5. En el caso de los conjuntos de *Hénon*, *Logistic* y *S.T.A.R.* (ver las Figuras 3.1 y 3.2), hay un amplio rango de valores de γ casi óptimos, y el valor proporcionado por la heurística, marcado con una línea discontinua, se queda en el borde de dicho rango. En el caso de los datos *Mackey-Glass*, *Sunspots*, *London*, *Electric*, *Computer* y *Laser* (Figuras 3.2, 3.3, 3.4 y 3.5), hay un claro valor óptimo, y el valor heurístico propuesto lleva a

Tabla 3.1: Conjuntos de datos usados en los experimentos del Capítulo 3

Serie	Modelo	F.S.	Tend.	Datos
Hénon	$y_{t+1} = F(y_t, y_{t-1})$	No	No	500
Logistic	$y_{t+1} = F(y_t)$	No	No	500
Mackey-Glass	$y_{t+1} = F(y_{t-5}, y_{t-11}, y_{t-17}, y_{t-23})$	No	No	500
AR(4)	$y_{t+1} = F(y_t, y_{t-1}, y_{t-2}, y_{t-3})$	No	No	500
S.T.A.R.	$y_{t+1} = F(y_t, y_{t-1})$	No	No	500
Sunspots	$y_{t+1} = F(y_t, ..., y_{t-12})$	Si	No	306
London	$y_{t+1} = F(y_t, ..., y_{t-13})$	Si	No	500
Electric	$y_{t+1} = F(y_t, ..., y_{t-16})$	Si	Si	500
Computer	$y_{t+1} = F(y_t, ..., y_{t-20})$	Si	No	500
Laser	$y_{t+1} = F(y_t, ..., y_{t-6})$	Si	No	500

un punto muy cercano al mismo del espacio. Los resultados para *AR(4)* (Figura 3.2) son los peores de los experimentos realizados, pero, sin embargo, el valor es adecuado como punto de partida del proceso de optimización. Como puede observarse, γ_h es en general una elección razonable para empezar la búsqueda del valor óptimo de γ.

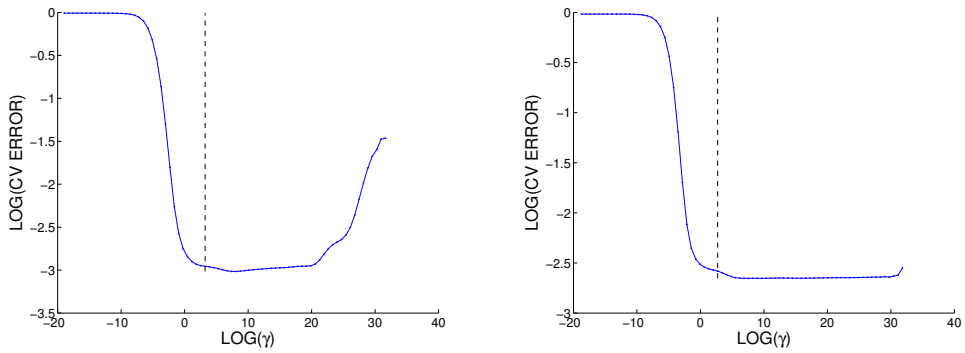


Fig. 3.1: Error de validación cruzada de orden 10 de LSSVM frente a γ (γ_h marcado con línea discontinua) para un σ fijo para *Hénon* (izquierda) y *Logistic* (derecha)

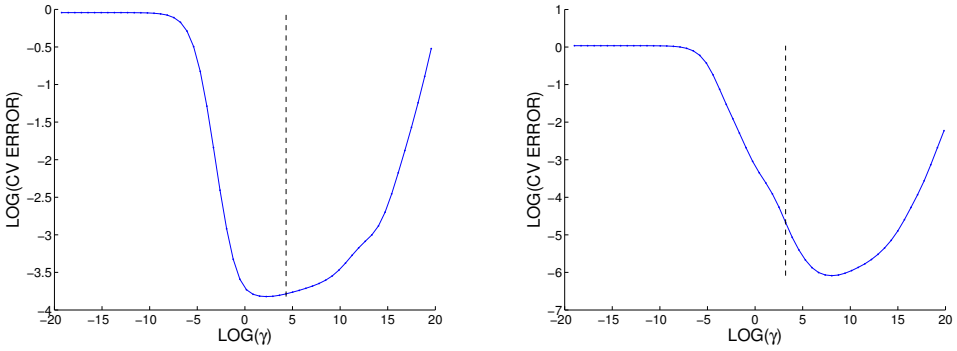


Fig. 3.2: Error de validación cruzada de orden 10 de LSSVM frente a γ (γ_h marcado con línea discontinua) para un σ fijo para *Mackey-Glass* (izquierda) y *AR(4)* (derecha)

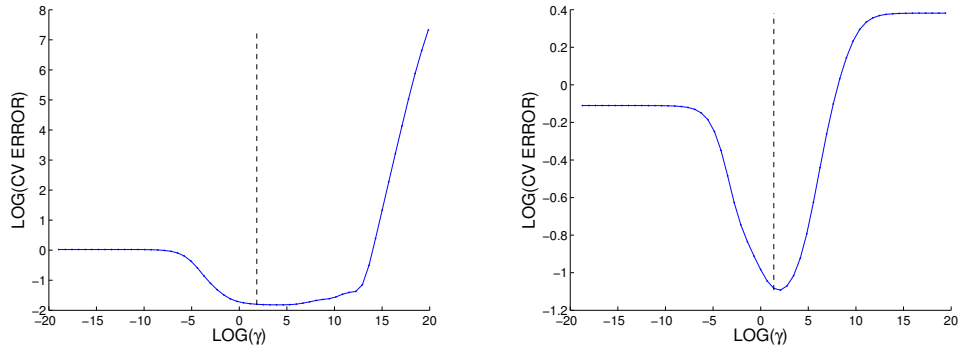


Fig. 3.3: Error de validación cruzada de orden 10 de LSSVM frente a γ (γ_h marcado con línea discontinua) para un σ fijo para *S.T.A.R.* (izquierda) y *Sunspots* (derecha)

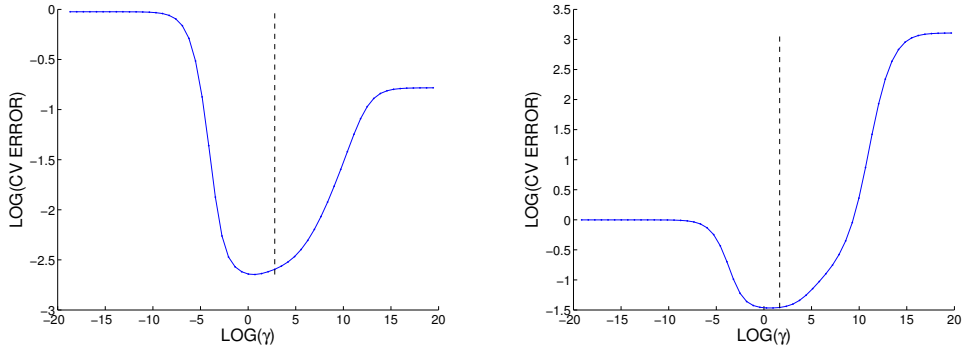


Fig. 3.4: Error de validación cruzada de orden 10 de LSSVM frente a γ (γ_h marcado con línea discontinua) para un σ fijo para *London* (izquierda) y *Electric* (derecha)

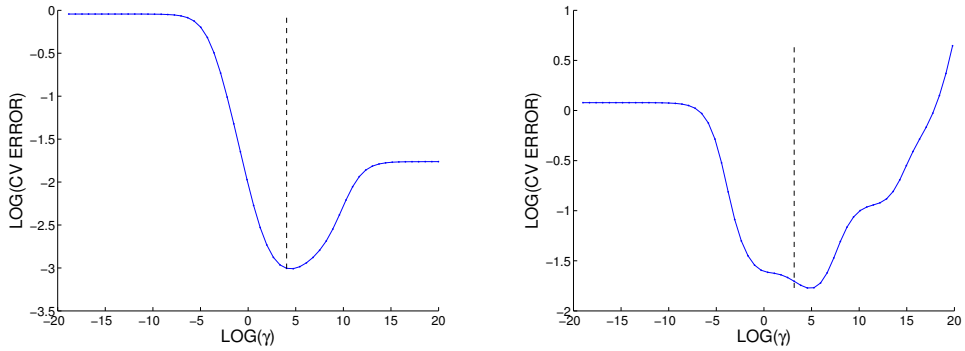


Fig. 3.5: Error de validación cruzada de orden 10 de LSSVM frente a γ (γ_h marcado con línea discontinua) para un σ fijo para *Computer* (izquierda) y *Laser* (derecha)

Con el fin de completar el estudio, un conjunto de experimentos fueron llevados a cabo para probar los dos valores iniciales heurísticos propuestos en la Sección 3.4. Para ello, se obtuvieron 500 pares de muestras (γ, σ) de cada una de las siguientes distribuciones bidimensionales a las que denotaremos *rangos* $R_0, \cdots, R_5$:

R_0: $\gamma \sim \gamma_h + N(0, (\gamma_h - \gamma_{min})/100)$ y $\sigma \sim U(\sigma_{min}, \sigma_{max})$

R_1: $\gamma \sim U(\gamma_{min}, \gamma_h)$ y $\sigma \sim U(1/2 \cdot \sigma_{min}, 2 \cdot \sigma_{max})$

R_2: $\gamma \sim U(\gamma_{min}, \gamma_h)$ y $\sigma \sim U(1/4 \cdot \sigma_{min}, 4 \cdot \sigma_{max})$

R_3: $\gamma \sim U(\gamma_{min}, \gamma_h)$ y $\sigma \sim U(1/6 \cdot \sigma_{min}, 6 \cdot \sigma_{max})$

R_4: $\gamma \sim U(\gamma_{min}, \gamma_h)$ y $\sigma \sim U(1/8 \cdot \sigma_{min}, 8 \cdot \sigma_{max})$

R_5: $\gamma \sim U(\gamma_{min}, \gamma_h)$ y $\sigma \sim U(1/10 \cdot \sigma_{min}, 10 \cdot \sigma_{max})$

donde N es la distribución normal y U la distribución uniforme. El error de validación cruzada de orden 10 para modelos LSSVM que usan valores de parámetros sacados de cada rango es calculada con la Ecuación (3.22). El primer *rango* es el recomendado por nuestra heurística para conseguir buenos valores iniciales de parámetros; los siguientes 5 son rangos que sucesivamente aumentan el espacio de búsqueda para cada parámetro.

Para evaluar el rendimiento de la heurística para cada conjunto de datos, primero un valor pseudo-óptimo de γ y σ con respecto al error de validación cruzada de orden 10 para modelos LSSVM fue calculado mediante múltiples ejecuciones de un procedimiento de gradiente conjugado (CG), usando las Ecuaciones de (3.22) a (3.32), partiendo de los valores generados como puntos iniciales.

Para cada conjunto de datos, las distancias desde estos valores óptimos de parámetros calculados a los obtenidos en cada rango se resumen en la Tabla 3.2. Adicionalmente, en las Figuras de la 3.6 a la 3.15 se representan en diagramas de cajas ordenados por rango los valores obtenidos del error de validación cruzada de orden 10. La representación dibuja una caja delimitada por los cuartiles menor, mediano y mayor cuyos extremos son 1,5 veces el rango intercuartil desde los finales de la caja. Los valores fuera de rango *Outliers* son denotados por signos $+$.

Los resultados obtenidos se muestran para la serie de *Hénon* en la Figura 3.6, para *Logistic* en la Figura 3.7, para *Mackey-Glass* en la Figura 3.8, para *S.T.A.R.* en la Figura 3.10, para *Sunspots* en la Figura 3.11, y para *Electric* en la Figura 3.13. Todos ellos confirman la validez de la heurística ya que muestran que el valor medio del error de validación cruzada crece con el tamaño del rango, a pesar del número relativamente alto de *outliers*. Los mejores resultados se obtienen en el primer rango, el recomendado según la heurística propuesta.

Los resultados para *AR(4)*, en la Figura 3.9, y para *London*, en la Figura 3.12, muestran que los valores de la heurística no son tan críticos en estos casos. Es importante apuntar que la influencia de los rangos en el error de validación cruzada en estos casos es menor que con otras series, pero sigue presente. El motivo puede ser el comportamiento altamente lineal de estas series, y es posible que el valor del parámetro σ no sea tan relevante en este caso.

Para las dos series altamente no lineales de la competición de Santa Fe (*Computer*, Figura 3.14, y *Laser*, Figura 3.15), los resultados demuestran que el primer rango consigue mejor valores de parámetros. Sin embargo, el resto de rangos tienen escasa relevancia. Los resultados para el valor medio del error de validación cruzada presentan un gran número de outliers, pero siguen confirmando la validez de la heurística.

De las figuras mencionadas anteriormente y la Tabla 3.2 se ha podido comprobar cómo los valores de parámetros generados por el primer rango son, en media, más cercanos al óptimo calculado y, que sus errores de validación cruzada de orden 10, son menores. Con el fin de hacer una comparativa estadística (distancia y error de validación cruzada) para todas las series, los resultados fueron normalizados por medio del peor caso (último rango) y analizado mediante el test de Kruskal-Wallis. Se obtuvo que los rangos influyen con una probabilidad muy alta ($p \ll 0{,}001$) en los resultados y que el rango R_0 es el mejor para la solución del problema (el diagrama de cajas puede verse en la Figura 3.16).

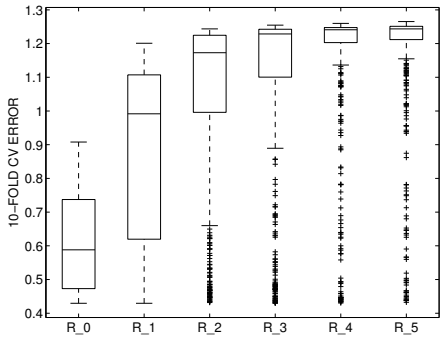


Fig. 3.6: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *Hénon*

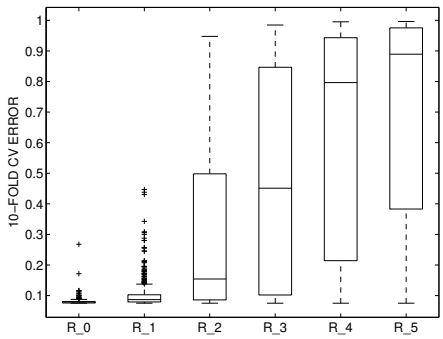


Fig. 3.7: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *Logistic*

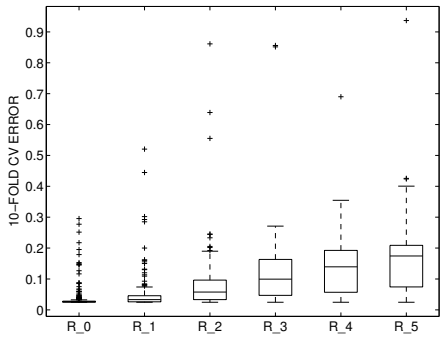


Fig. 3.8: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *Mackey-Glass*

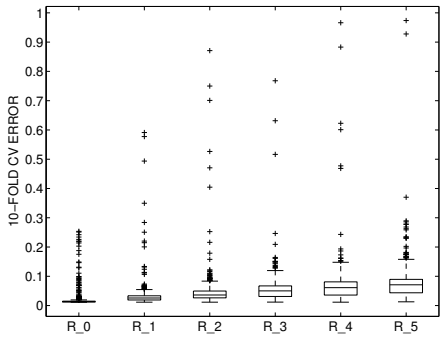


Fig. 3.9: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *AR(4)*

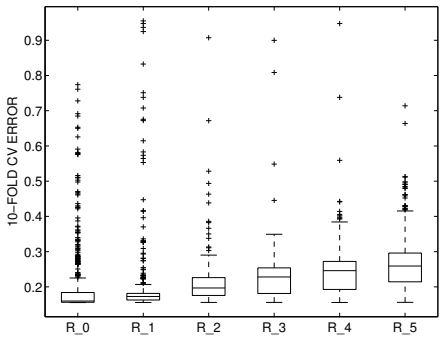


Fig. 3.10: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *S.T.A.R.*

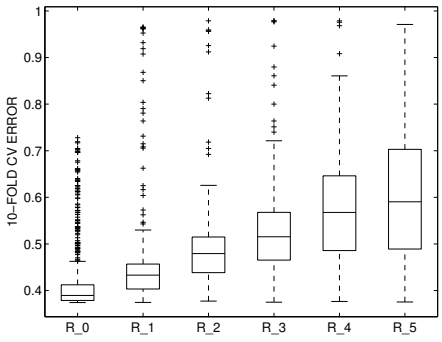


Fig. 3.11: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *Sunspots*

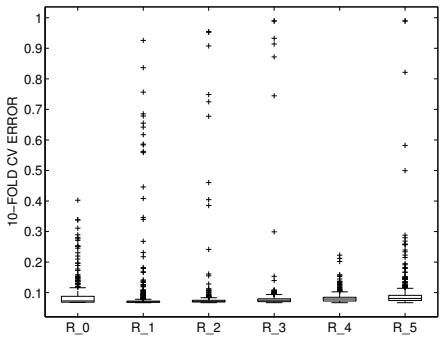


Fig. 3.12: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *London*

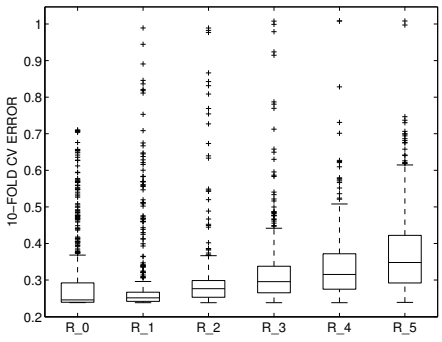


Fig. 3.13: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *Electric*

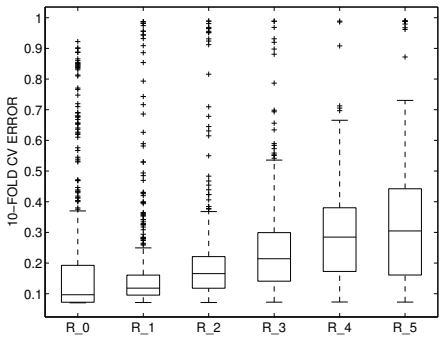


Fig. 3.14: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *Computer*

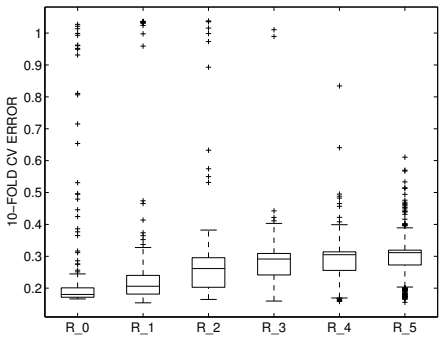


Fig. 3.15: Diagrama de cajas del error de validación cruzada de orden 10 por rango para *Laser*

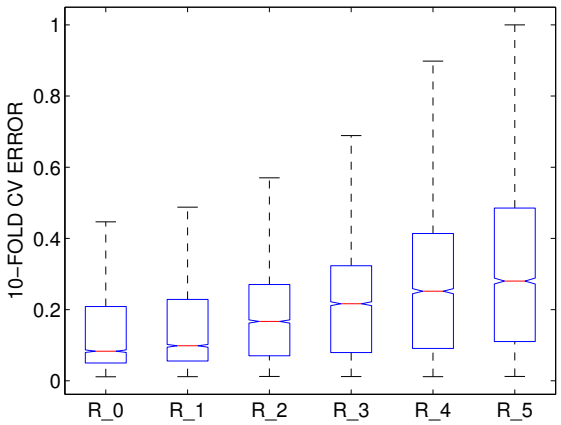


Fig. 3.16: Diagrama de cajas del error de validación cruzada de orden 10 por rango para todas las series

Tabla 3.2: Media y desviación estándar de la distancia al óptimo encontrado por rango (experimentos del capítulo 3)

DATASET		
Hénon	R_0: 1.68e+00 ± 3.83e-01	R_1: 2.00e+00 ± 6.70e-01
	R_2: 2.66e+00 ± 6.98e-01	R_3: 2.99e+00 ± 7.67e-01
	R_4: 3.23e+00 ± 8.95e-01	R_5: 3.50e+00 ± 8.55e-01
Logistic	R_0: 5.28e+00 ± 2.40e-01	R_1: 6.05e+00 ± 6.40e-01
	R_2: 6.13e+00 ± 6.45e-01	R_3: 6.27e+00 ± 6.76e-01
	R_4: 6.29e+00 ± 6.34e-01	R_5: 6.31e+00 ± 6.18e-01
Mackey-Glass	R_0: 1.68e+00 ± 1.88e-01	R_1: 1.43e+00 ± 4.73e-01
	R_2: 1.89e+00 ± 5.48e-01	R_3: 2.18e+00 ± 6.58e-01
	R_4: 2.42e+00 ± 7.08e-01	R_5: 2.62e+00 ± 7.39e-01
AR(4)	R_0: 2.85e+01 ± 2.51e-01	R_1: 2.91e+01 ± 7.01e-01
	R_2: 2.89e+01 ± 7.74e-01	R_3: 2.88e+01 ± 7.33e-01
	R_4: 2.87e+01 ± 7.68e-01	R_5: 2.86e+01 ± 7.31e-01
S.T.A.R.	R_0: 4.03e+00 ± 2.90e-01	R_1: 4.65e+00 ± 5.84e-01
	R_2: 4.67e+00 ± 5.10e-01	R_3: 4.75e+00 ± 5.16e-01
	R_4: 4.78e+00 ± 4.99e-01	R_5: 4.86e+00 ± 5.25e-01
Sunspots	R_0: 1.95e+00 ± 2.28e-01	R_1: 2.35e+00 ± 3.52e-01
	R_2: 2.46e+00 ± 3.29e-01	R_3: 2.56e+00 ± 3.49e-01
	R_4: 2.71e+00 ± 3.80e-01	R_5: 2.78e+00 ± 4.29e-01
London	R_0: 2.06e+00 ± 2.43e-01	R_1: 1.46e+00 ± 5.47e-01
	R_2: 1.64e+00 ± 5.00e-01	R_3: 1.87e+00 ± 5.64e-01
	R_4: 2.11e+00 ± 5.51e-01	R_5: 2.22e+00 ± 5.89e-01
Electric	R_0: 8.83e-01 ± 5.14e-01	R_1: 8.65e-01 ± 5.28e-01
	R_2: 1.11e+00 ± 5.25e-01	R_3: 1.39e+00 ± 5.61e-01
	R_4: 1.58e+00 ± 6.02e-01	R_5: 1.78e+00 ± 6.21e-01
Computer	R_0: 8.01e+00 ± 2.19e-01	R_1: 8.54e+00 ± 5.63e-01
	R_2: 8.50e+00 ± 5.88e-01	R_3: 8.43e+00 ± 5.94e-01
	R_4: 8.43e+00 ± 5.72e-01	R_5: 8.43e+00 ± 5.68e-01
Laser	R_0: 1.71e+01 ± 1.56e-01	R_1: 1.79e+01 ± 7.21e-01
	R_2: 1.79e+01 ± 7.43e-01	R_3: 1.79e+01 ± 7.40e-01
	R_4: 1.78e+01 ± 7.43e-01	R_5: 1.78e+01 ± 7.95e-01

3.5.2. Experimentos para la comprobación de los métodos de optimización de parámetros

Para comprobar el rendimiento del método de optimización y las heurísticas de inicialización de parámetros para modelos LSSVM con *kernel* RBF propuestas, vamos a utilizar los mismos conjuntos de datos que en el Apartado 3.5.1. Así, en estos experimentos se cogieron 1000 puntos de cada serie, $y_1, ..., y_{1000}$, que fueron normalizados con media zero y varianza uno, usando los 500 primeros valores para entrenamiento y los siguientes 500 para test (con la excepción del conjunto de datos *Sunspots* que sólo tiene 306 datos para la cual también se coge la primera mitad para entrenamiento y la siguiente para test). En el caso de las series determinísticas sin ruido *Hénon*, *Mackey-Glass* y *Logistic*, se hicieron pruebas añadiendo ruido Gaussiano aditivo con varianza controlada $\sigma_e^2 = \{0, 1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$ a la secuencia de datos $\hat{y}_i = y_i + N(0, \sigma_e^2)$, con el fin de poner a prueba el estimador de ruido no paramétrico y confirmar que la presencia de ruido en la entrada puede hacer que la estimación no sea la deseada. Los vectores de entrada/salida se generaron usando los mismos regresores de la Tabla 3.1. Para cada modelo LSSVM, se aplica la heurística y el modelo se optimiza con el procedimiento de evaluación de bajo coste del error de validación cruzada de orden 10 (este valor de orden es uno de los más usado en la literatura):

1. Directamente: fijando los valores de los parámetros a los valores dados por las heurísticas descritas $(\gamma_h, \overline{\sigma})$.

2. Procedimiento de búsqueda en rejilla: el mismo usado en LSSVMlab con el rango sugerido en la Sección 3.4.

3. Procedimiento de búsqueda local: usando el método de optimización de Broyden, Fletcher, Goldfarb y Shanno, BFGS [Broyden, 1970]. Éste es un método para resolución de sistema no lineales sin restricciones, derivado del método de Newton. Se aplica el procedimiento cogiendo como punto inicial el dado por las heurísticas $(\gamma_h, \overline{\sigma})$.

4. Procedimiento de búsqueda global: dado que los métodos de optimización locales, como el BFGS, normalmente devolverán en problemas de optimización globales un mínimo local, que no tiene por qué ser la solución óptima, también se aplica un Búsqueda Local Iterativa (con BFGS como búsqueda local). La idea es: una vez detectado el haber llegado a un mínimo local con la búsqueda local y si quedan iteraciones de búsqueda, se vuelve a aplicar dicha búsqueda local empezando desde el punto alcanzado, pero tras hacerle sufrir una perturbación con el fin de que el procedimiento tome otro camino. Se aplica el procedimiento cogiendo como punto inicial el dado por las heurísticas $(\gamma_h, \overline{\sigma})$.

Se hizo uso de la toolbox LSSVMlab. Los errores de entrenamiento y test se proporcionan independientes de escala usando el *Error Cuadrático Medio Normalizado* [Herrera et al., 2005] ($NRMSE = \sqrt{MSE/\sigma_y^2}$, donde MSE es el error cuadrático medio y σ_y^2 la varianza de la función que se modela). El mejor $NRMSE$ de entrenamiento evaluado del estimador no paramétrico del ruido se muestra también. Todo el software está implementado en Matlab, y los experimentos ejecutados en un computador personal con una CPU Intel Core 2 Quad a 2.83GHz y 8 GB de RAM. El número máximo de evaluaciones del procedimiento de validación cruzada es de 100 en todos los experimentos, que es un número suficiente para alcanzar la convergencia en la mayoría de los casos.

Los resultados obtenidos se muestran en las Tablas 3.3 a 3.12. El significado de cada columna es el siguiente:

1. σ_e^2: varianza del ruido Gaussiano añadido.
2. *NNE*: varianza del ruido evaluada con el estimador de ruido no paramétrico.
3. $NRMSE_{NNE}$ mejor error de entrenamiento que puede obtenerse según el NNE evaluado.
4. Método:
   a) *Heuristic*, usando $\gamma_h, \overline{\sigma}$
   b) *GS*, búsqueda en rejilla, usando la evaluación de coste reducido del error de validación cruzada de orden 10 en el rango $[0{,}5 \cdot \gamma_h, 2 \cdot \gamma_h]$.
   c) *BFGS* aplicado desde $\gamma_h, \overline{\sigma}$.
   d) *ILS*, búsqueda local iterativa (de BFGS) aplicada desde $\gamma_h, \overline{\sigma}$
   e) *LSSVMlab*, toolbox LSSVMlab aplicada desde $\gamma_h, \overline{\sigma}$
5. T: tiempo en segundos.
6. f.e.: número de evaluaciones del error de validación cruzada realizadas.
7. 10-CVE: valor final del error de validación cruzada de orden 10 alcanzado.

Los mejores valores de error de validación cruzada de orden 10 están destacados en negrita en las tablas.

Como puede verse, el procedimiento para evaluar el error de validación cruzada de [An et al., 2007] es más rápido que la implementación inmediata (ver las filas etiquetadas *GS* y *LSSVMlab*). Se ve también que la evaluación de las derivadas parciales (usadas en *BFGS*, y por lo tanto también en *ILS*) añaden tiempo

a la evaluación del error de validación cruzada pero permite alcanzar un óptimo local con menor número de evaluaciones. El punto inicial para el procedimiento de optimización tiene una influencia importante en el resultado. El procedimiento de búsqueda global (*ILS*) sólo alcanza en unos pocos casos una solución mejor que la hallada por la búsqueda local (*BFGS*) desde el punto de inicio dado por la heurística: *Hénon* (Tabla 3.3), *Logistic* (Tabla 3.4) y *Laser* (Tabla 3.12). Desde el punto de vista de los resultados, la búsqueda local *CG* es el método de optimización que mejor rendimiento ha tenido de todos los empleados, seguido de la búsqueda global (ILS), la búsqueda en rejilla *GS* (este último ha conseguido el mejor resultado de error de validación cruzada sólo en una ocasión para *Logistic*) y finalmente *LSSVMlab*. El número de evaluaciones requeridas por *BFGS* es mucho menor que el máximo permitido en la mayor parte de los casos, obteniendo una convergencia en un número de evaluaciones del error validación cruzada y derivada del mismo muy inferior a 100. Es notable reseñar los relativamente buenos resultados obtenidos solamente con la heurística de inicialización de parámetros de modelos (filas etiquetadas *Heuristic*). Estos resultados avalan tanto la validez de la heurística de inicialización de parámetros como el uso de la optimización basada en gradiente.

Tabla 3.3: Optimizaciones con kernel RBF usando distintos métodos para la serie *Hénon*.

		Method	Training	Test	T	f.e.	10-CVE
		Heuristic	1.50e-04	1.68e-04	4.11e-01	0	3.25e-08
σ_e^2	0.00e+00	GS	5.14e-05	6.49e-05	2.99e+01	100	5.02e-09
NNE	1.00e-06	CG	1.50e-04	1.68e-04	1.06e+00	1	3.25e-08
$NRMSE_{NNE}$	1.00e-03	ILS	5.75e-05	6.49e-05	6.92e+01	100	**4.15e-09**
		LSSVMlab	1.46e-04	1.64e-04	6.48e+02	100	3.25e-08
		Heuristic	1.37e+00	1.41e+00	3.59e-01	0	1.93e+00
σ_e^2	1.00e+00	GS	1.37e+00	1.41e+00	2.88e+01	100	1.92e+00
NNE	1.74e+00	CG	1.37e+00	1.41e+00	7.98e+00	12	**1.92e+00**
$NRMSE_{NNE}$	1.32e+00	ILS	1.37e+00	1.41e+00	1.12e+02	100	1.92e+00
		LSSVMlab	1.37e+00	1.42e+00	6.35e+02	100	1.93e+00
		Heuristic	6.04e-01	6.01e-01	3.62e-01	0	3.79e-01
σ_e^2	1.00e-01	GS	5.82e-01	6.05e-01	2.88e+01	100	3.65e-01
NNE	3.54e-01	CG	5.80e-01	6.07e-01	9.40e+00	14	**3.64e-01**
$NRMSE_{NNE}$	5.95e-01	ILS	5.80e-01	6.07e-01	6.86e+01	100	3.64e-01
		LSSVMlab	6.05e-01	6.02e-01	6.37e+02	100	3.79e-01
		Heuristic	2.22e-01	2.19e-01	3.60e-01	0	5.24e-02
σ_e^2	1.00e-02	GS	2.04e-01	2.07e-01	2.89e+01	100	4.89e-02
NNE	3.61e-02	CG	1.93e-01	2.05e-01	1.01e+01	15	**4.65e-02**
$NRMSE_{NNE}$	1.90e-01	ILS	1.93e-01	2.05e-01	7.78e+01	100	4.65e-02
		LSSVMlab	2.22e-01	2.19e-01	6.42e+02	100	5.24e-02
		Heuristic	7.33e-02	7.26e-02	3.62e-01	0	5.77e-03
σ_e^2	1.00e-03	GS	6.85e-02	7.03e-02	2.89e+01	100	5.39e-03
NNE	4.11e-03	CG	6.85e-02	7.02e-02	1.40e+01	21	**5.39e-03**
$NRMSE_{NNE}$	6.41e-02	ILS	6.85e-02	7.02e-02	3.14e+01	100	5.39e-03
		LSSVMlab	7.32e-02	7.25e-02	6.46e+02	100	5.77e-03
		Heuristic	2.37e-02	2.36e-02	3.60e-01	0	6.16e-04
σ_e^2	1.00e-04	GS	2.32e-02	2.32e-02	2.89e+01	100	6.09e-04
NNE	2.50e-04	CG	2.32e-02	2.32e-02	6.71e+00	10	**6.09e-04**
$NRMSE_{NNE}$	1.58e-02	ILS	2.32e-02	2.32e-02	2.40e+01	100	6.09e-04
		LSSVMlab	2.37e-02	2.36e-02	6.46e+02	100	6.16e-04

Tabla 3.4: Optimizaciones con kernel RBF usando distintos métodos para la serie *Logistic*.

		Method	Training	Test	T	f.e.	10-CVE
		Heuristic	8.43e-05	8.70e-05	4.08e-01	0	7.97e-09
σ_e^2	0.00e+00	GS	3.29e-05	3.38e-05	2.98e+01	100	1.32e-09
NNE	1.00e-06	CG	8.43e-05	8.70e-05	1.06e+00	1	7.97e-09
$NRMSE_{NNE}$	1.00e-03	ILS	1.93e-07	2.11e-07	4.72e+01	100	**4.39e-14**
		LSSVMlab	8.41e-05	8.69e-05	8.09e+02	100	7.97e-09
		Heuristic	1.38e+00	1.40e+00	3.51e-01	0	1.93e+00
σ_e^2	1.00e+00	GS	1.38e+00	1.40e+00	2.86e+01	100	1.93e+00
NNE	1.96e+00	CG	1.38e+00	1.40e+00	6.60e+00	10	**1.93e+00**
$NRMSE_{NNE}$	1.40e+00	ILS	1.38e+00	1.40e+00	6.77e+01	100	1.93e+00
		LSSVMlab	1.39e+00	1.40e+00	8.00e+02	100	1.93e+00
		Heuristic	6.83e-01	6.83e-01	3.57e-01	0	4.77e-01
σ_e^2	1.00e-01	GS	6.79e-01	6.77e-01	2.87e+01	100	4.72e-01
NNE	4.51e-01	CG	6.79e-01	6.77e-01	1.20e+01	18	**4.72e-01**
$NRMSE_{NNE}$	6.71e-01	ILS	6.79e-01	6.77e-01	4.71e+01	100	4.72e-01
		LSSVMlab	6.85e-01	6.85e-01	8.01e+02	100	4.77e-01
		Heuristic	2.71e-01	2.54e-01	3.59e-01	0	7.52e-02
σ_e^2	1.00e-02	GS	2.63e-01	2.47e-01	2.87e+01	100	7.12e-02
NNE	7.02e-02	CG	2.62e-01	2.46e-01	1.97e+01	30	7.10e-02
$NRMSE_{NNE}$	2.65e-01	ILS	2.62e-01	2.46e-01	6.72e+01	100	**7.06e-02**
		LSSVMlab	2.72e-01	2.54e-01	8.01e+02	100	7.52e-02
		Heuristic	9.41e-02	9.54e-02	3.60e-01	0	9.04e-03
σ_e^2	1.00e-03	GS	9.26e-02	9.14e-02	2.89e+01	100	8.86e-03
NNE	9.08e-03	CG	9.41e-02	9.54e-02	8.00e+00	12	9.04e-03
$NRMSE_{NNE}$	9.53e-02	ILS	9.12e-02	9.07e-02	5.58e+01	100	**8.72e-03**
		LSSVMlab	9.41e-02	9.54e-02	8.08e+02	100	9.04e-03
		Heuristic	3.02e-02	2.94e-02	3.58e-01	0	9.45e-04
σ_e^2	1.00e-04	GS	3.03e-02	2.95e-02	2.88e+01	100	**9.42e-04**
NNE	8.26e-04	CG	3.02e-02	2.94e-02	4.04e+00	6	9.44e-04
$NRMSE_{NNE}$	2.87e-02	ILS	3.02e-02	2.94e-02	6.44e+01	100	9.44e-04
		LSSVMlab	3.02e-02	2.94e-02	8.08e+02	100	9.45e-04

Tabla 3.5: Optimizaciones con kernel RBF usando distintos métodos para la serie *Mackey-Glass*.

		Method	Training	Test	T	f.e.	10-CVE
		Heuristic	5.58e-03	4.95e-03	3.88e-01	0	1.38e-04
σ_e^2	0.00e+00	GS	5.51e-03	4.88e-03	2.87e+01	100	1.37e-04
NNE	1.00e-06	CG	4.49e-03	4.03e-03	1.47e+01	22	**1.25e-04**
$NRMSE_{NNE}$	1.00e-03	ILS	4.49e-03	4.03e-03	6.48e+01	100	1.25e-04
		LSSVMlab	5.61e-03	4.97e-03	6.21e+02	100	1.38e-04
		Heuristic	1.22e+00	1.24e+00	3.42e-01	0	1.59e+00
σ_e^2	1.00e+00	GS	1.22e+00	1.24e+00	2.74e+01	100	1.59e+00
NNE	1.61e+00	CG	1.22e+00	1.24e+00	5.09e+00	8	**1.59e+00**
$NRMSE_{NNE}$	1.27e+00	ILS	1.22e+00	1.24e+00	1.89e+01	100	1.59e+00
		LSSVMlab	1.24e+00	1.24e+00	6.05e+02	100	1.59e+00
		Heuristic	4.42e-01	4.42e-01	3.42e-01	0	2.28e-01
σ_e^2	1.00e-01	GS	4.47e-01	4.42e-01	2.73e+01	100	2.25e-01
NNE	1.89e-01	CG	4.49e-01	4.42e-01	8.16e+00	13	**2.25e-01**
$NRMSE_{NNE}$	4.35e-01	ILS	4.49e-01	4.42e-01	8.00e+01	100	2.25e-01
		LSSVMlab	4.43e-01	4.42e-01	6.08e+02	100	2.28e-01
		Heuristic	1.45e-01	1.47e-01	3.43e-01	0	2.58e-02
σ_e^2	1.00e-02	GS	1.42e-01	1.45e-01	2.74e+01	100	2.56e-02
NNE	2.07e-02	CG	1.40e-01	1.43e-01	9.47e+00	15	**2.55e-02**
$NRMSE_{NNE}$	1.44e-01	ILS	1.40e-01	1.43e-01	6.70e+01	100	2.55e-02
		LSSVMlab	1.45e-01	1.47e-01	6.16e+02	100	2.58e-02
		Heuristic	5.24e-02	5.52e-02	3.43e-01	0	3.70e-03
σ_e^2	1.00e-03	GS	4.50e-02	5.26e-02	2.73e+01	100	3.51e-03
NNE	1.30e-03	CG	4.55e-02	5.25e-02	1.01e+01	16	**3.50e-03**
$NRMSE_{NNE}$	3.61e-02	ILS	4.55e-02	5.25e-02	2.69e+01	100	3.50e-03
		LSSVMlab	5.25e-02	5.52e-02	6.19e+02	100	3.70e-03
		Heuristic	1.60e-02	2.00e-02	3.40e-01	0	6.01e-04
σ_e^2	1.00e-04	GS	1.58e-02	1.99e-02	2.73e+01	100	5.95e-04
NNE	1.00e-06	CG	1.61e-02	1.97e-02	8.20e+00	13	**5.79e-04**
$NRMSE_{NNE}$	1.00e-03	ILS	1.61e-02	1.97e-02	6.33e+01	100	5.79e-04
		LSSVMlab	1.60e-02	2.00e-02	6.17e+02	100	6.01e-04

Tabla 3.6: Optimizaciones con kernel RBF usando distintos métodos para la serie *AR(4)*.

		Método	Entrenamiento	Test	T	f.e.	10-CVE
		Heuristic	1.25e-01	9.92e-02	2.43e-01	0	2.00e-02
σ_e^2	0.00e+00	GS	8.84e-02	6.83e-02	1.67e+01	100	1.32e-02
NNE	4.54e-02	CG	3.50e-02	3.47e-02	9.08e+00	23	1.34e-03
$NRMSE_{NNE}$	2.13e-01	ILS	3.56e-02	3.48e-02	5.11e+01	100	**1.31e-03**
		LSSVMlab	1.32e-01	1.06e-01	4.09e+02	100	2.00e-02

Tabla 3.7: Optimizaciones con kernel RBF usando distintos métodos para la serie *STAR*.

		Método	Entrenamiento	Test	T	f.e.	10-CVE
		Heuristic	4.00e-01	3.98e-01	4.08e-01	0	1.69e-01
σ_e^2	0.00e+00	GS	3.99e-01	3.98e-01	2.97e+01	100	1.67e-01
NNE	1.67e-01	CG	3.93e-01	3.95e-01	1.13e+01	16	**1.66e-01**
$NRMSE_{NNE}$	4.08e-01	ILS	3.93e-01	3.95e-01	7.50e+01	100	1.66e-01
		LSSVMlab	4.00e-01	3.98e-01	6.39e+02	100	1.69e-01

Tabla 3.8: Optimizaciones con kernel RBF usando distintos métodos para la serie *Sunspots*.

		Método	Entrenamiento	Test	T	f.e.	10-CVE
		Heuristic	4.41e-01	7.23e-01	7.22e-02	0	3.16e-01
σ_e^2	0.00e+00	GS	3.76e-01	7.28e-01	1.41e+00	100	2.92e-01
NNE	4.19e-01	CG	3.21e-01	7.41e-01	9.12e-01	9	**2.80e-01**
$NRMSE_{NNE}$	6.47e-01	ILS	3.21e-01	7.41e-01	4.29e+00	100	2.80e-01
		LSSVMlab	4.18e-01	7.33e-01	6.59e+01	100	3.16e-01

Tabla 3.9: Optimizaciones con kernel RBF usando distintos métodos para la serie *London*.

		Método	Entrenamiento	Test	T	f.e.	10-CVE
		Heuristic	2.20e-01	2.73e-01	4.05e-01	0	7.07e-02
σ_e^2	0.00e+00	GS	2.43e-01	2.64e-01	2.93e+01	100	6.76e-02
NNE	5.42e-02	CG	2.43e-01	2.64e-01	9.04e+00	13	**6.76e-02**
$NRMSE_{NNE}$	2.33e-01	ILS	2.43e-01	2.64e-01	7.27e+01	100	6.76e-02
		LSSVMlab	2.21e-01	2.73e-01	6.46e+02	100	7.07e-02

Tabla 3.10: Optimizaciones con kernel RBF usando distintos métodos para la serie *Electric*.

		Método	Entrenamiento	Test	T	f.e.	10-CVE
		Heuristic	3.85e-01	4.91e-01	3.86e-01	0	2.60e-01
σ_e^2	0.00e+00	GS	4.31e-01	4.86e-01	2.80e+01	100	2.49e-01
NNE	2.29e-01	CG	4.33e-01	4.86e-01	8.01e+00	12	**2.49e-01**
$NRMSE_{NNE}$	4.78e-01	ILS	4.33e-01	4.86e-01	4.81e+01	100	2.49e-01
		LSSVMlab	3.86e-01	4.91e-01	6.42e+02	100	2.60e-01

Tabla 3.11: Optimizaciones con kernel RBF usando distintos métodos para la serie *Computer*.

		Método	Entrenamiento	Test	T	f.e.	10-CVE
		Heuristic	2.87e-04	2.72e-01	3.94e-01	0	1.25e-01
σ_e^2	0.00e+00	GS	5.65e-05	2.64e-01	2.85e+01	100	1.11e-01
NNE	1.00e-06	CG	1.20e-01	1.62e-01	2.79e+01	40	**2.74e-02**
$NRMSE_{NNE}$	1.00e-03	ILS	1.20e-01	1.62e-01	6.17e+01	100	2.74e-02
		LSSVMlab	4.80e-04	2.77e-01	8.17e+02	100	1.25e-01

Tabla 3.12: Optimizaciones con kernel RBF usando distintos métodos para la serie *Laser*.

		Método	Entrenamiento	Test	T	f.e.	10-CVE
		Heuristic	8.73e-02	1.83e-01	3.91e-01	0	5.47e-02
σ_e^2	0.00e+00	GS	3.92e-02	1.73e-01	2.87e+01	100	4.86e-02
NNE	9.41e-03	CG	1.51e-02	1.79e-01	1.16e+01	17	4.47e-02
$NRMSE_{NNE}$	9.70e-02	ILS	7.75e-02	1.43e-01	7.75e+01	100	**4.38e-02**
		LSSVMlab	7.95e-02	1.76e-01	8.05e+02	100	5.47e-02

4. KWKNN UNA TÉCNICA DE *KERNEL* DE BAJO COSTE COMPUTACIONAL.

Las Máquinas de Vectores Soporte de Mínimos Cuadrados (*Least Square Support Vector Machines*, LSSVM) [Suykens et al., 2002] y los Procesos Gaussianos son el estado del arte en métodos *kernel* para aproximación funcional. Sin embargo, siguen teniendo un problema de difícil solución: el alto coste computacional de su entrenamiento, que crece en función del número de puntos implicados. En este capítulo se presenta una nueva técnica de *kernel* con un menor coste computacional: los K vecinos ponderados con distancia basada en *kernel* (*Kernel based distance Weighted K-Nearest Neighbours*, KWKNN). Amén de los menores tiempos de cómputo, esta técnica tiene la ventaja adicional de ser más fácilmente paralelizable que los método de *kernel* clásicos, y se presenta en este capítulo un estudio de una implementación de la misma (*Parallel KWKNN*, PKWKNN). La paralelización proporciona una reducción del tiempo requerido para ejecutar el algoritmo. Además, gracias a esta paralelización, se pueden afrontar problemas que, debido al gran tamaño del conjunto de datos, no serían asumibles con el KWKNN secuencial ni con otros modelos como LSSVM.

Por tanto, en este capítulo se describe en primer lugar el modelo KWKNN. Una vez demostrada la eficacia del modelo, se propone un algoritmo paralelo que permite reducir significativamente el tiempo de cómputo y, por tanto, el poder afrontar problemas de gran tamaño en lo que al conjunto de datos se refiere.

4.1. Motivación

El coste computacional de los métodos *kernel* escalan con el número de puntos de entrenamiento de tal modo que requieren mucha capacidad de cómputo a partir de unos pocos miles de datos. Esto hace que se vuelvan totalmente impracticables a medida que sigue aumentando el número de datos. Tomando LSSVM como referencia al ser el más eficiente, tenemos que el entrenamiento básico (lo que en

Suykens denomina primer nivel de optimización [Suykens et al., 2002]) tiene una complejidad de $O(N^2)$, donde N es el número de puntos de entrenamiento. En esta medida no se considera la optimización de los parámetros del *kernel* ni de los hiperparámetros del modelo. La optimización de los parámetros del *kernel* e hiperparámetros en dichos modelos aumenta todavía más el coste computacional, y típicamente tiene una complejidad algorítmica de $O(N^3)$. El hallar buenos valores de parámetros para un problema es crítico de cara a obtener buenos resultados con el modelo diseñado.

En el Capítulo 3 se ha presentado un enfoque para mejorar la optimización de parámetros de LSSVM basado en el uso de heurísticas para inicialización de los parámetros y el uso del gradiente conjugado para buscar los valores óptimos de los mismos. En el presente capítulo, en cambio, se parte del siguiente hecho: un *kernel* define un *espacio de características* que es independiente del tipo concreto de modelo en el que se usará (LSSVM, SVR o Proceso Gaussiano, por ejemplo). De este modo, se puede adaptar el algoritmo de los K vecinos más cercanos para que trabaje sobre dicho espacio mediante la re-definición de la función de distancia que se utilizará. Esta idea se aplicó concretamente al algoritmo de los K vecinos ponderados para regresión y problemas de predicción de series temporales. No obstante, sería fácilmente aplicable a otros algoritmos basados en los K vecinos más cercanos para otros usos (como clasificación).

El algoritmo de los K vecinos ponderados con distancia basada en *kernel* (*Kernel Weighted K-Nearest Neighbours*, KWKNN) tiene un rendimiento aceptable y un menor coste computacional que las LSSVM o los Procesos Gaussianos. En cualquier caso, para conjuntos de datos con un gran número de muestras (varias decenas de miles) un computador personal normalmente ya no podrá afrontar los costes computacionales y de memoria necesarios. La paralelización del algoritmo es algo deseable, al distribuir datos y cómputo sobre un cierto número de nodos (por ejemplo, en una plataforma cluster) permitiendo obtener tiempos de ejecución razonables. En la literatura, hay ejemplos de implementaciones paralelas de la búsqueda de los vecinos más cercanos para clasificación como en [Aparício et al., 2006] y [Bosch and Sloot, 2007]. En [Aparício et al., 2006] se hace una descripción de una implementación para el proyecto de cómputo Grid EGEE [1] que paraleliza el algoritmo a distintos niveles usando tanto hebras [Drepper and Molnar, 2005] como paso de mensajes (MPI[2]). En [Bosch and Sloot, 2007] se describe una implementación que llega a obtener ganancias super-lineales, pero sólo en algunos casos y que es muy dependiente de la representación de los puntos de entrada y la distancia usada (Euclídea).

[1] *Enabling Grids for E-sciencE*, `http://public.eu-egee.org/`

[2] `http://www.mcs.anl.gov/research/projects/mpi/`

4.2. Kernel K vecinos ponderados

Los K vecinos más cercanos (K-NN) es un algoritmo que ha sido aplicado normalmente para tareas de clasificación [Dasarathy, 1990] aunque también ha sido utilizado para problemas de regresión [Lora and Santos, 2002; Sorjamaa et al., 2005]. Es un algoritmo simple y eficiente comparado con los más usuales y refinados, obteniendo en algunos casos una precisión similar. En esta sección, se describe la variante de los K vecinos más cercanos ponderados para regresión y su extensión para usar *kernels*, lo que permite presentarla como método *kernel*.

Dado un conjunto de muestras de una función $\{(x_1,y_1),\ldots,(x_N,y_N)\} \subset \mathbb{R}^d \times \mathbb{R}$, donde N es el número de muestras y d es la dimensionalidad de la entrada, el método de los K vecinos más cercanos ponderados para regresión calcula la salida para un nuevo punto de entrada $\hat{x}$ como:

$$f(\hat{x}) = \frac{\sum_{i=1}^{K} y_i^{\hat{x}} w^K\left(\hat{x}, x_i^{\hat{x}}\right)}{\sum_{i=1}^{K} w^K\left(\hat{x}, x_i^{\hat{x}}\right)}, \tag{4.1}$$

$$\text{donde } w^K\left(\hat{x}, x_i^{\hat{x}}\right) = \left[1 - \frac{D\left(\hat{x}, x_i^{\hat{x}}\right)^2}{D\left(\hat{x}, x_{K+1}^{\hat{x}}\right)^2}\right]^2 \tag{4.2}$$

donde D es una función de distancia definida sobre $\mathbb{R}^d$, $x_i^{\hat{x}}$ es el punto de entrenamiento i-ésimo vecino más cercano a $\hat{x}$ de acuerdo a D (usualmente, la distancia Euclídea), $y_i^{\hat{x}}$ su salida correspondiente y $w^K\left(\hat{x}, x_i^{\hat{x}}\right)$ es su peso asignado. Como se puede ver en la formulación, el método depende exclusivamente de los valores de la función de distancia, y por lo tanto es independiente de la estructura de la entrada. Esta última propiedad hace de los K vecinos ponderados (y, de hecho, al igual que la mayoría de los algoritmos basados en K-NN) un método claramente "kernelizable" ya que se puede definir la distancia en el espacio de características gracias al kernel.

Para definir los K vecinos más cercanos ponderados con una distancia basada en *kernel*, se necesita una función de distancia en el espacio de características. La única información disponible sobre dicho espacio es la función *kernel* k que, formalmente, es el producto escalar de las entradas tras aplicar una proyección Φ de los puntos del *espacio de entrada* al *espacio de características* [Scholkopf and Smola, 2001]: $k(x,x') = \langle \Phi(x), \Phi(x') \rangle$. Se cumple la siguiente relación entre la función de distancia y la norma cuadrática en el espacio de características: $D(x,x')^2 = \|\Phi(x) - \Phi(x')\|^2$. La norma cuadrática puede escribirse entonces como:

$$\begin{aligned} \|\Phi(x)-\Phi(x')\|^2 &= \langle\Phi(x),\Phi(x)\rangle + \langle\Phi(x'),\Phi(x')\rangle - 2\langle\Phi(x),\Phi(x')\rangle \\ &= k(x,x)+k(x',x')-2k(x,x') \end{aligned} \tag{4.3}$$

Por lo tanto, la norma cuadrática de dos puntos en el espacio de características puede ser escrita en términos de valores de una función de *kernel*, y no hay necesidad de conocer explícitamente la función Φ. Consecuentemente, la adaptación de los K vecinos más cercanos ponderados para trabajar en espacio de características es bastante sencilla y se obtiene definiendo el cuadrado de la función de distancia en la Ecuación (4.2) como:

$$D\left(x,x'\right)^2 = k(x,x)+k(x',x')-2k(x,x') \tag{4.4}$$

La formulación nos permite usar KWKNN como cualquier otro método *kernel*, teniendo que ajustar los valores de los parámetros del *kernel* en el modelo para cada problema específico.

4.3. Evaluación del error de validación cruzada para KWKNN

Un modelo KWKNN debe de ajustarse a un conjunto de datos en particular, al igual que con cualquier otro método *kernel*, fijando adecuadamente los parámetros de su *kernel* (esto es similar a lo que se vió para LSSVM en el Capítulo 3). Con el fin de obtener los valores de parámetros para un *kernel* arbitrario, se pueden estimar los valores óptimos por medio del error de validación cruzada.

El error de validación cruzada de orden l (l-CV) de KWKNN, y en particular el caso del "Dejar Uno Fuera" (*Leave-One-Out*, LOO), puede ser derivado muy eficientemente de las distancias entre puntos de entrenamiento. Se debe recordar que para calcular el error de validación cruzada de orden l se debe:

1. dividir el conjunto de datos en l subconjuntos de igual tamaño
2. para cada uno de los l subconjuntos
   - *a*) crear un modelo que use como conjunto de entrenamiento el resto del conjunto original
   - *b*) usar dicho modelo para predecir las salidas del subconjunto seleccionado

La media de los l errores (por cada subconjunto) es el error de validación cruzada de orden l. Un caso extremo es el error LOO, "Dejar Uno Fuera", que es cuando el número de subconjuntos iguala el de elementos del conjunto de entrenamiento, por lo que cada conjunto de test tiene un único elemento. La implementación del validación cruzada de orden l para modelos KWKNN con N muestras

implica l evaluaciones de las distancias entre pares de muestras, tarea cuya complejidad es

$$O\left((N-N/l)\cdot N/l\right) \tag{4.5}$$

además de N/l (por cada muestra) búsquedas para encontrar los $K+1$ vecinos más cercanos:

$$O\left((N-N/l)\cdot (K+1)\right) \tag{4.6}$$

Por lo tanto, la evaluación del error de validación cruzada de orden l para KWKNN tiene una complejidad total de:

$$\begin{array}{l} O\left(l\cdot N/l\cdot (N-N/l)\cdot (K+1)\right) = \\ O\left(N\cdot (N-N/l)\cdot K\right) = \\ O\left(N^2\cdot K\right) \end{array} \tag{4.7}$$

La complejidad algorítmica de la evaluación del error de validación cruzada de KWKNN es $O\left(N^2\cdot K\right)$ y si consideramos $K \ll N$ incluso puede considerarse $O\left(N^2\right)$, lo cual es un orden de complejidad inferior a la de un modelo LSSVM o un Proceso Gaussiano, que es $O\left(N^3\right)$ (ver Capítulo 3).

4.4. Optimización de parámetros de la función *kernel*

En nuestro caso, dado que no es posible derivar la función del error de l-CV respecto a los parámetros del *kernel* para que éstos sean optimizados mediante el Gradiente Conjugado, se usa una variante de la Búsqueda de Vecindario Variable (*Variable Neighbour Search*, VNS) [Mladenović and Hansen, 1997]. El VNS se basa en tres hechos simples:

1. Un mínimo local con respecto a una estructura de vecindad no lo es necesariamente con respecto a otra.

2. Un mínimo global es un mínimo local con respecto a todas las posibles estructuras de vecindades.

3. En muchos problemas el mínimo local con respecto a una o varias estructuras de vecindad están relativamente cerca.

El VNS propuesto funciona del siguiente modo: en cada iteración se explora el vecindario de la mejor solución actual, es decir, se genera un número de soluciones creadas mediante la aplicación de una perturbación de la mejor solución actual. En caso de no hallar en el vecindario una solución mejor que la mejor solución actual, se incrementa la modificación usada para generar las soluciones vecinas, es decir,

se incrementa el tamaño del vecindario. En caso de hallar en el vecindario una solución mejor que la mejor solución actual, ésta se convierte en la nueva mejor solución actual y se reduce el tamaño del vecindario.

El VNS propuesto tiene 2 parámetros: el número de iteraciones del algoritmo y el número de vecinos generados en cada iteraciones. El producto de ambos valores es el número total de evaluaciones del error de validación cruzada que se realizan en una ejecución del VNS, y puede usarse para controlar el tiempo de ejecución. El tamaño del vecindario determina el componente de explotación del algoritmo mientras que el número de iteraciones el de exploración. Este algoritmo, aparte de trabajar con un criterio no derivable, es especialmente adecuado para afrontar un problema de búsqueda global, al ser la meta-heurística en que se basa específicamente diseñada para no quedarse atrapada en mínimos locales.

4.5. Optimización del valor K

Análogamente al factor de regularización γ para LSSVM [Suykens et al., 2002], el número de vecinos más cercanos K puede ser considerado el hiperparámetro a optimizar para un modelo KWKNN. El criterio respecto al cual optimizar K es el mismo usado en este trabajo para los parámetros del *kernel*: el error de validación cruzada. Por la propia formulación del KWKNN, no tiene sentido buscar K demasiado altos, pues el peso asignado por la distancia disminuye para los más lejanos drásticamente. En la práctica a partir de un K-ésimo vecino los restantes no influirán significativamente en los resultados. Hay varias formas de afrontar la optimización de K en un rango de valores $[2, K_{max}]$, por ejemplo:

- Optimización de K como un parámetro adicional al mismo nivel que los parámetros del *kernel*.
- Optimización de los parámetros del *kernel* para un valor fijo de K, iterando para los distintos valores de K.
- Optimización de K para cada conjunto de valores de parámetros del *kernel* considerados.

La primera aproximación no parece muy aconsejable, ya que mezcla los valores enteros de K con los parámetros del *kernel* que toman valores reales y se mueve en rangos de búsqueda muy distintos. La segunda opción es más adecuada que la primera al separar K de los parámetros del *kernel*, pero el problema que tiene es que para un K fijo el espacio a explorar de valores de parámetros del *kernel* es enorme. La tercera opción tiene como ventaja principal sobre las otras dos ser mucho más eficiente, ya que una vez fijados los valores de parámetros del *kernel* y un valor K máximo, K_{max}, se pueden calcular los K_{max} vecinos más cercanos

para dichos valores de parámetros del *kernel* y se puede proceder a calcular el criterio de validación cruzada para cada K en el rango de búsqueda sin necesidad de recalcular los vecinos más cercanos cada vez. Uno de los aspectos más tenidos en cuenta en este trabajo es la reducción de costes computacionales, por lo que la aproximación escogida para optimización del K fue la tercera, al ser de la tres la que se espera tenga menos costes computacionales. En los Algoritmos 4.1 y 4.2 está descrito el proceso de optimización utilizado.

ENTRADA: k, *kernel* $k(x,x';\Theta)$
ENTRADA: Θ_0, valores iniciales de los parámetros del *kernel*
ENTRADA: X,Y, X es m puntos d dimensionales de entrada con sus correspondientes m salidas asociadas en Y
ENTRADA: $K_{\text{máx}}$, valor máximo del parámetro del K-nn
ENTRADA: it, número de iteraciones
ENTRADA: it_e, número de vecinos generados por vecindario
ENTRADA: l, orden del error de validación cruzada con respecto al cual se optimiza

SALIDA: Θ, valores finales de los parámetros del *kernel*
SALIDA: K, valor final del parámetro del K-nn
SALIDA: $coste$, valor del error de validación cruzada validación del KWKNN

$coste = +\infty$
$\Theta = \Theta_0$
$K = 2$
para $K_i = 2$ to $K_{\text{máx}}$ **hacer**
  $(\Theta_i, coste_i)$ = VNS de optimización KWKNN para K fijo $(k, \Theta_0, X, Y, K_i, it, it_e, l)$
  si $coste > coste_i$ **entonces**
    $\Theta = \Theta_i$
    $K = K_i$
    $coste = coste_i$
  fin si
fin para

Alg. 4.1: VNS para optimización KWKNN

ENTRADA: k, *kernel* $k(x,x';\Theta)$
ENTRADA: Θ_0, valores iniciales de los parámetros del *kernel*
ENTRADA: X,Y, X es m puntos d dimensionales de entrada con sus correspondientes m salidas asociadas en Y
ENTRADA: K, parámetro del K-nn
ENTRADA: it, número de iteraciones
ENTRADA: it_e, número de vecinos generados por vecindario
ENTRADA: l, orden del error de validación cruzada con respecto al cual se optimiza

SALIDA: Θ, valores finales de los parámetros del *kernel*
SALIDA: $coste$, valor del error de validación cruzada validación del KWKNN

constantes $e_{\text{máx}}$, $e_{\text{mín}}$ tamaños máximo y mínimo del vecindario, e tamaño actual inicializado a $e_{\text{mín}}$
$coste = KWKNN - ERROR_{l-fold}(k,\Theta_0,X,Y,K)$
$\Theta = \Theta_0$
para $i = 1$ to it **hacer**
  $mejora = false$
  para $j = 1$ to it_e **hacer**
    $vecino = \Theta + norm(0,e)$, donde $norm(0,e)$ es la distribución normal de media 0 y varianza e
    $coste_{vecino} = KWKNN - ERROR_{l-fold}(k,vecino,X,Y,K)$
    si $coste > coste_{vecino}$ **entonces**
      $coste = coste_{vecino}$
      $\Theta = vecino$
      $mejora = true$
    fin si
  fin para
  si $mejora$ **entonces**
    $e = e \times 2$
    si $e > e_{\text{máx}}$ **entonces**
      $e > e_{\text{máx}}$
    fin si
  si no
    $e > e_{\text{mín}}$
  fin si
fin para

Alg. 4.2: VNS de optimización KWKNN para K fijo

4.6. Experimentos con KWKNN

La Figura 4.1 muestra los tiempos de optimización de modelos KWKNN y LSSVM con un mismo *kernel* usando VNS. Se han limitado a 500 las evaluaciones del error de validación cruzada, tanto para error 10-CV como LOO trabajando un problema con tamaños de conjunto de entrenamiento de entre 500 y 2000 muestras (el parámetro K de KWKNN se optimiza en el rango $[2, 20]$). De esta figura se infiere claramente la diferencia en complejidad computacional entre KWKNN y LSSVM. Para conjuntos con varios miles de muestras, LSSVM se vuelve impracticable mientras KWKNN sí puede seguir usándose dada su menor complejidad computacional.

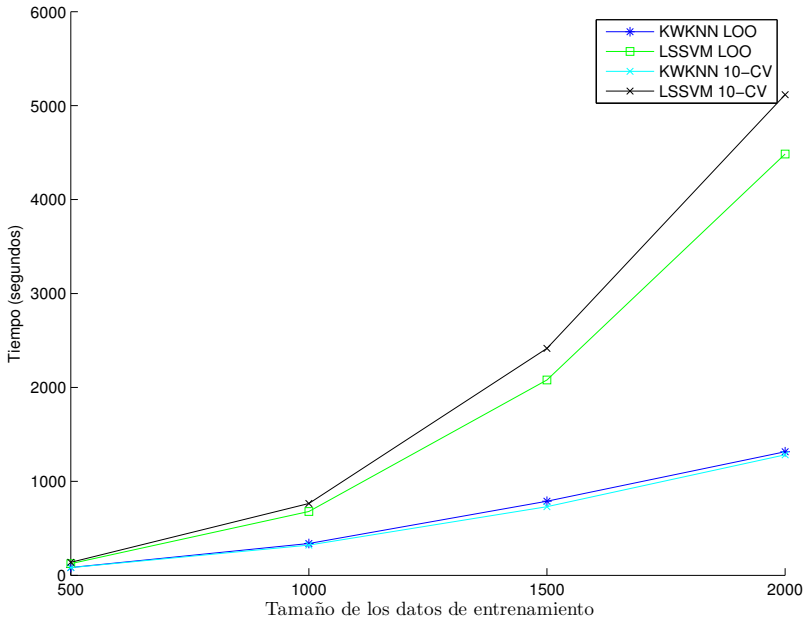


Fig. 4.1: Tiempos (en segundos) de optimización para modelos KWKNN y LSSVM con el mismo *kernel* y datos frente al número de muestras.

4.7. Paralelización del KWKNN para grandes conjuntos de datos

Los grandes problemas cuando se trabaja con grandes conjuntos de datos son el coste computacional y los requisitos de memoria que implica el KWKNN y, concretamente, el error l-CV. En la versión paralela diseñada del KWKNN (que referiremos como PKWKNN), las muestras del conjunto de entrenamiento son distribuidas entre los procesos paralelos[3], ver Figura 4.2. Para N muestras y p

[3] cada proceso se mapea a un procesador o nodo

procesos, cada proceso tiene N/p muestras (en caso de no ser exactamente divisible N entre p, los procesos $0, \cdots, p-2$ tendrán $[N/p]$ muestras, N/p redondeado hacia arriba, y el proceso $p-1$, $N-(p-1)[N/p]$ muestras).

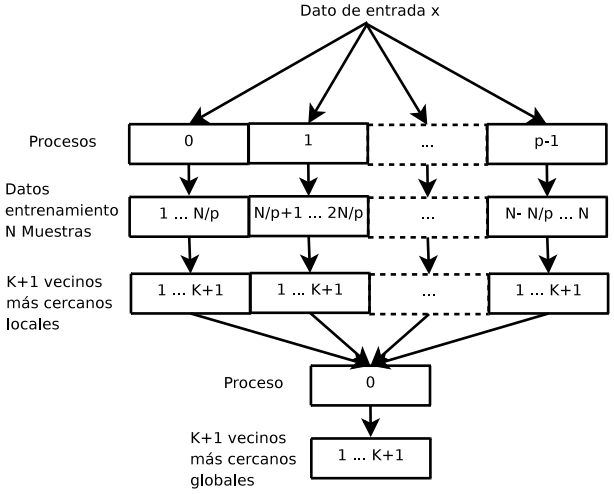


Fig. 4.2: Distribución de los K vecinos más cercanos

Dado un nuevo punto de entrada $\hat{x}$, PKWKNN calcula su salida $\hat{y}$ como se describe en el Algoritmo 4.3. Primero, cada nodo calcula los $K+1$ puntos más cercanos a $\hat{x}$ de sus datos locales en términos de distancia basada en *kernel*. Seguidamente, todos los nodos mandan sus vecinos más cercanos, con sus salidas asociadas, al nodo 0 (que participa en la operación), siendo éste el encargado de calcular los $K+1$ vecinos más cercanos a $\hat{x}$ globales a partir de los $K+1$ vecinos más cercanos locales en cada nodo. Finalmente, el nodo 0 aplica la Ecuación (4.2) para calcular $\hat{y}$, la salida de $\hat{x}$. Sin tener en cuenta los tiempos de comunicación, la complejidad viene determinada por:

$$O\left(\left\lceil\frac{N}{p}\right\rceil \cdot \log(\left\lceil\frac{N}{p}\right\rceil) + p\cdot(K+1)\cdot\log(p\cdot(K+1))\right) \tag{4.8}$$

y si consideramos $p(K+1) < \left\lceil\frac{N}{p}\right\rceil$, el término correspondiente al cómputo de los $K+1$ vecinos más cercanos globales puede ser despreciado:

$$O\left(\left\lceil\frac{N}{p}\right\rceil \cdot \log(\left\lceil\frac{N}{p}\right\rceil)\right) \tag{4.9}$$

La influencia del segundo término de la Ecuación 4.8 no es apreciable hasta que $N/p \simeq p\cdot(K+1)$. En nuestros experimentos con PKWKNN, como se verá, usamos un conjunto de datos con N sobre 30000 y modelos con $K=10$, lo que

supone que no se notaría la influencia del búsqueda de los vecinos más cercanos hasta usar al menos unos $p \simeq 55$ procesadores.

Este esquema es básico y supone un cuello de botella serio al tener que comunicar los p conjuntos de $K+1$ vecinos más cercanos locales al proceso 0 para que éste calcule los $K+1$ vecinos más cercanos globales. Una mejora de PKWKNN se basa en distribuir el cálculo de los $K+1$ vecinos más cercanos. Dado un nuevo punto de entrada $\hat{x}$, PKWKNN calcula su salida $\hat{y}$ de la siguiente manera: Primero, cada nodo calcula los $K+1$ puntos más cercanos a $\hat{x}$ de sus datos locales en términos de distancia basada en kernel. Seguido, todos los nodos participan en un árbol de reducción binario con raíz en el nodo 0 (que participa en la operación) con el fin de hallar los $K+1$ vecinos más cercanos a $\hat{x}$ globales a partir de los $K+1$ vecinos más cercanos locales en cada nodo así como sus salidas. Finalmente, el nodo 0 aplica la Ecuación (4.2) para calcular $\hat{y}$, la salida de $\hat{x}$. Sin tener en cuenta los tiempos de comunicación, la complejidad viene determinada por:

1. hallar los $K+1$ vecinos más cercanos a los datos locales en cada nodo
2. hallar los $K+1$ vecinos más cercanos a los datos globales mediante un árbol binario de reducción en $\log_2(p)-1$ pasos, ver Figura 4.3.

Teniendo en cuenta los pasos mencionados, la evaluación del PKWKNN en p nodos para un nuevo punto $\hat{x}$ con N muestras de entrenamiento es:

$$O\left(N/p \cdot (K+1) + (\log_2(p)-1) \cdot 2 \cdot (K+1)\right) = O\left(N/p \cdot K + \log_2(p) \cdot K\right) \tag{4.10}$$

Lo cual supone una mejora substancial con respecto a la primera versión.

Al igual que en el caso de modelos KWKNN, para PKWKNN los parámetros del *kernel* deben de optimizarse para un conjunto de datos, y el error de validación cruzada de orden l es la opción elegida. Dadas una serie de muestras distribuidas entre p procesos, la evaluación del error de validación cruzada de orden l (l-CV) para un l arbitrario es bastante compleja en términos de comunicaciones, ya que los datos en cada partición pueden llegar a estar repartidos por varios procesos. Pero hay 2 casos simples y eficientes: el error p-CV, haciendo coincidir el orden de validación cruzada l con el número de procesos p, y el error LOO. La estructura de comunicaciones coincide en ambos casos. El Algoritmo 4.4 lo ilustra en un pseudo-código que cubre ambos casos.

4.8. Experimentos con PKWKNN

La serie temporal número 3 de la competición del ESTSP 2008, Figura 4.4[4], tiene un gran número de muestras (31614 valores). La serie fue especialmente

[4] `http://www.estsp.org/`

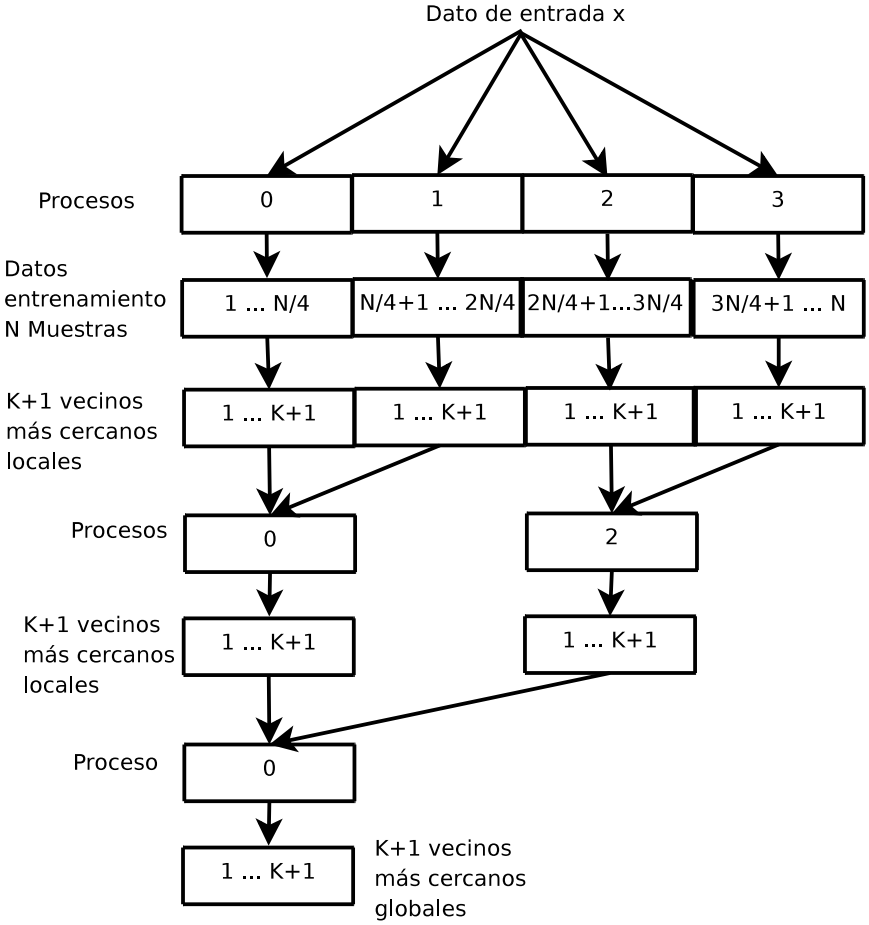


Fig. 4.3: Ejemplo de árbol de reducción binario para cálculo de los vecinos más cercanos con 4 procesos

escogida para penalizar el uso de LSSVM o Procesos Gaussianos en la competición, por consiguiente, es un conjunto de datos ideal para demostrar los beneficios de la paralelización del algoritmo [Rubio et al., 2008]. Hemos usado los índices de las muestras como entrada, $X = 1, ..., 31614$, y como salida la serie temporal $Y = y_1, .., y_{31614}$. El *kernel* de los modelos es el de la Ecuación (4.13), y es un *kernel* específicamente diseñado para el problema abordado usando el método descrito en el Capítulo 5. Es destacable su alto número de parámetros del *kernel*, 7. Se fijó el número de vecinos más cercanos usados por PKWKNN a $K = 10$, este valor de K se eligió al ser un compromiso adecuado entre un espacio de búsqueda suficiente para K y un tiempo de ejecución estimado razonable para el algoritmo.

ENTRADA: X, m puntos de entrada (en el proceso 0)
ENTRADA: p, número de procesos
ENTRADA: r, el número del proceso local
ENTRADA: $S^{(r)} = [X^{(r)}, Y^{(r)}]$, muestras de entrenamiento locales al proceso r
ENTRADA: N, número global de muestras de entrenamiento
ENTRADA: $n = |X^{(r)}|$, número local de muestras de entrenamiento
ENTRADA: K, parámetro del KNN
ENTRADA: $D^2(x,x';\Theta)$, norma cuadrática basada en *kernel* entre los puntos x,x', donde Θ son los valores de los parámetros del *kernel*

SALIDA: $\hat{Y}$, las salidas calculadas para los puntos de X (en el proceso 0)

(1) Transmisión de los puntos de entrada de X a todos los procesos desde el proceso 0
(2) Calcular $D^2(X_j, X_i^{(r)};\Theta)$ donde $j = 1,\cdots,m, i = 1,\cdots,n$
(3) Calcular los $K+1$-ésimos vecinos más cercanos a cada X_j ($nn[X_j, X^{(r)}]_i$, $i = 1,\cdots,K+1$ es el índice del i-ésimo vecino más cercano $X^{(r)}$ a X_j)
(4) Fijar $Kn^{(r)} = \{ (D^2(X_j, X^{(r)}_{nn[X_j,X^{(r)}]_i};\Theta),\ Y^{(r)}_{nn[X_j,X^{(r)}]_i}),\quad j = 1,...,m,\quad i = 1,...,K+1 \}$
(5) Recoger en el proceso 0 todos los $Kn^{(r)}$ y unirlos en $Kn^{(X)}$
si $r = 0$ **entonces**
  (7) Calcular los $K+1$-ésimos vecinos más cercanos globales a las muestras para cada X_j usando la información de distancia en $Kn^{(X)}$
  (8) Calcular la salida $\hat{Y}_j$ para cada X_j usando la Ecuación (4.2) con la distancia e información de salida en $K^{(X)}$
fin si

Alg. 4.3: PKWKNN

$$k_{\lambda-periodic}(x_i,x_j;\{\theta_1,\theta_2\}) = \theta_1 \exp\left(-2\frac{sin(\frac{\pi}{\lambda}(x_i-x_j))^2}{\theta_2^2}\right) \tag{4.11}$$

$$k_{ratquad}(x_i,x_j;\{\theta,\alpha,\beta\}) = \theta\left(1+\frac{\|x_i-x_j\|^2}{2\alpha\beta^2}\right)^{-\alpha} \tag{4.12}$$

$$
\begin{aligned}
k(x_i,x_j;\Theta) = & k_{ratquad}(x_i,x_j;\{\theta_1,\alpha,\beta\}) + \\
& k_{7-periodic}(x_i,x_j;\{\theta_2,\theta_3\}) + \\
& k_{8736-periodic}(x_i,x_j;\{\theta_4,\theta_5\}) \\
\Theta = & \{\theta_1,\alpha,\beta,\theta_2,\theta_3,\theta_4,\theta_5\}
\end{aligned}
\tag{4.13}
$$

El algoritmo PKWKNN fue implementado directamente en Matlab. Se uso la biblioteca de acceso a la Interfaz de Paso de Mensajes (*Message Passing Interface*, MPI) en su implementación MPICH2 [Guillén, 2005] [Guillen et al., 2008a], y el programa compilado como aplicación independiente de Matlab con el fin de ejecutarlo en un plataforma cluster (cada nodo con un procesador AMD Opteron de 64 bits a 2.6 GHz con 2 GB de RAM). El programa se ejecutó en 2, 4, 8 y 16 procesadores (ver Tabla 4.1).

Es necesario recordar que el lenguaje de Matlab tiene características que no hacen aconsejable algunas formas de implementación que son comunes en otro lenguajes, como por ejemplo C. Concretamente, no se recomienda el uso de estructuras iterativas y en cambio se fomenta el uso de operaciones sobre matrices para obtener códigos eficientes. Estas características fueron tenidas en cuenta en la implementación, por ejemplo, a la hora de calcular los $K+1$ vecinos más cercanos: se obtienen las distancias entre pares de puntos usando operaciones matriciales y después se ordenan los puntos, en vez de ir procesando cada par de puntos e ir insertando la distancia en un vector ordenado (una opción más eficiente en otros lenguajes).

La Figura 4.5 muestra la media de los tiempos de 10 ejecuciones de PKWKNN frente al número de muestras para distintos números de procesadores. Se puede observar que aumentando el número de procesadores claramente mejoran los tiempos de ejecución. Como puede ser apreciado en la Figura 4.6, la ganancia calculada usando 10240 muestras[5] escala linealmente con el número de procesadores
.

El cálculo de la validación cruzada es crítico para la aplicación de PKWKNN, ya que es usado para la optimización de los parámetros del *kernel*. La Figura 4.7 revela que la media de los tiempos de evaluación de la validación cruzada esta en función del número de procesadores implicados. La Figura 4.7, donde se muestra la media de los tiempos de evaluación del error p-CV y LOO, revela que estos difieren muy poco: sobre el 4 % en el caso de 2 procesadores, 3 % en el caso de 4 u 8, y sobre un 2 % en el caso de 16, representando sobre 53, 16, 6 y 1,5 segundos

[5] no se utilizó el conjunto completo, ya que el algoritmo secuencial (KWKNN) no puede ser ejecutado con 31614 muestras por los excesivos requisitos de memoria

respectivamente. En términos absolutos, la diferencia de tiempos de evaluación entre la implementación del error p-CV y LOO para PKWKNN se vuelve prácticamente despreciable cuando se usa un número mayor de procesadores.

Una aplicación típica del método implica primero la optimización del error de validación cruzada con respecto a los parámetros del *kernel* por medio del VNS y la posterior utilización del modelo optimizado para predecir la salida de nuevos datos de entrada. En la Figura 4.8 se muestran los tiempos de ejecución del VNS usando error LOO (10 iteraciones con vecindario de tamaño 10, 100 evaluaciones del error de validación cruzada) frente al número de muestras de entrenamiento para 8 procesadores y en la Figura 4.9 el tiempo de predicción para los modelos generados frente al tamaño de entrada. Junto a las gráficas de tiempos se muestran las curvas de la complejidad computacional inferida en la Sección 4.7 ajustadas a los datos. Los resultados confirman la complejidad computacional calculada. En las Figuras 4.10 y 4.11 se reproducen estos resultados para 16 procesadores.

ENTRADA: P, número de procesos
ENTRADA: r, el número del proceso local
ENTRADA: $S^{(r)} = [X^{(r)}, Y^{(r)}]$, muestras locales al proceso r samples (puntos y salidas)
ENTRADA: N, número global de muestras
ENTRADA: $n = |X^{(r)}|$, número local de muestras
ENTRADA: K, parámetro del K-nn
ENTRADA: $D^2(x,x';\Theta)$, norma cuadrática basada en *kernel* entre los puntos x, x', done Θ son los valores de los parámetros del *kernel*

SALIDA: Error de validación cruzada de orden P/LOO para las muestras globales con PKWKNN
para $p = 1, ..., P$ **hacer**
(1) Transmisión de los datos de entrada a todos los procesos de $S^{(p)}$ desde el proceso p
si $r \neq p$ **entonces**
(2) Calcular $D^2(X_j^{(p)}, X_i^{(r)}; \Theta)$
(3) Calcular los $K+1$-ésimos vecinos más cercanos a cada $X_j^{(p)}$ ($nn[X_j^{(p)}, X^{(r)}]_i,\ i = 1, \cdots, K+1$ es el índice del i-ésimo vecino más cercano de $X^{(r)}$ a $X_j^{(p)}$)
(4) Fijar $Kn^{(p,r)} = \{\ (D^2(X_j^{(p)}, X^{(r)}_{nn[X_j^{(p)}, X^{(r)}]_i}),\ Y^{(r)}_{nn[X_j^{(p)}, X^{(r)}]_i})\}$
(5) Recoger $Kn^{(p,r)}$ en el proceso p
si no
si LOO error **entonces**
(6) Calcular $D^2(X_j^{(p)}, X_i^{(p)}; \Theta)$
(7) Calcular los $K+1$-ésimos vecinos más cercanos a cada $X_j^{(p)}$ ($nn[X_j^{(p)}, X^{(p)}]_i \neq j,\ i = 1, \cdots, K+1$ es el índice del i-ésimo vecino más cercano de $X^{(p)}$ a $X_j^{(p)}$)
(8) Fijar $Kn^{(p,p)} = \{\ (D^2(X_j^{(p)}, X^{(p)}_{nn[X_j^{(p)}, X^{(p)}]_i}),\ Y^{(r)}_{nn[X_j^{(p)}, X^{(r)}]_i}\ )\}$
fin si
(9) Recoger los $Kn^{(p,r)}$ en el proceso p y unirlos en $Kn^{(p)}$
(10) Calcular los $k+1$-ésimos vecinos más cercanos globales para cada muestra $X_j^{(p)}$ usando la información de distancia en $Kn^{(p)}$
(11) Calcular la salida $\hat{Y}_j$ para cada $X_j^{(p)}$ usando la Ecuación (4.2) con la distancia e información de salida en $Kn^{(p)}$
Calcular la suma del error cuadrático entre $\hat{Y}_j$ y $Y^{(p)}$ como $SSE^{(p)}$
fin si
fin para
(12) Recoger los $SSE^{(p)}$ en el proceso 0 como SSE
(13) Calcular en el proceso 0 el error de validación cruzada de orden P/LOO como $\sum SSE/N$

Alg. 4.4: Error de validación cruzada de orden P/LOO para PKWKNN

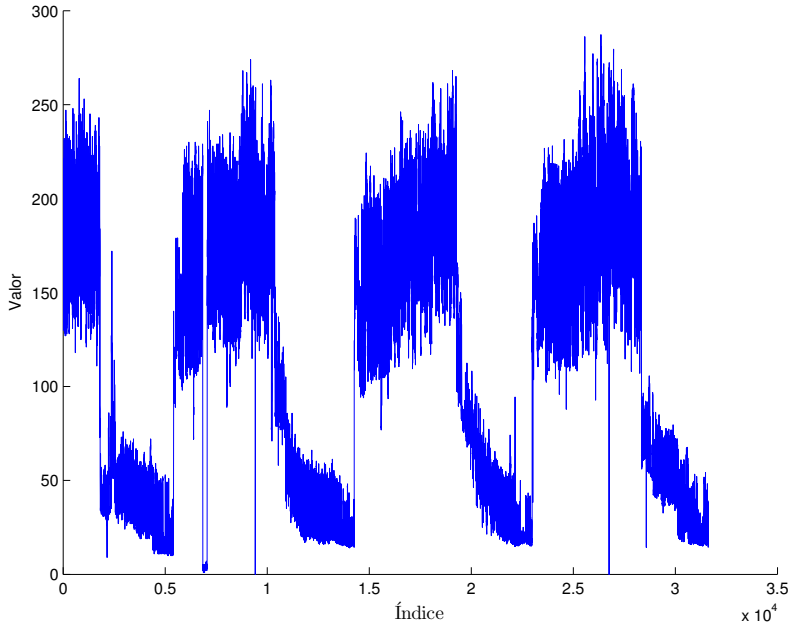


Fig. 4.4: Serie temporal 3 ESTSP.

Tabla 4.1: Número de muestras por proceso para los datos de los experimentos

Procesadores	índice del proceso → número de muestras
2	0,1 → 15807
4	0,..,2 → 7904, 3 → 7902
8	0,..,6 → 3952, 7 → 3950
16	0,..,14 → 1976, 15 → 1972

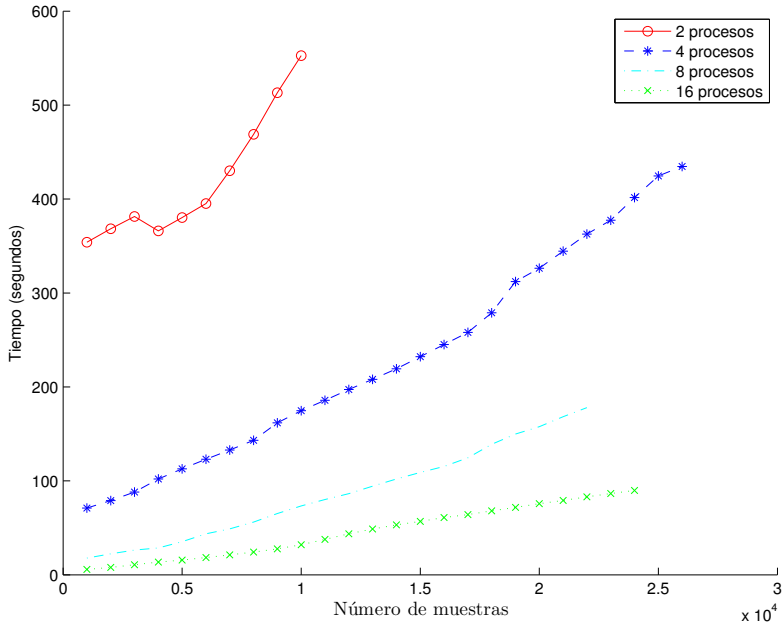


Fig. 4.5: Media de los tiempos de ejecución (10 repeticiones) de PKWKNN frente al Número de Muestras de entrada.

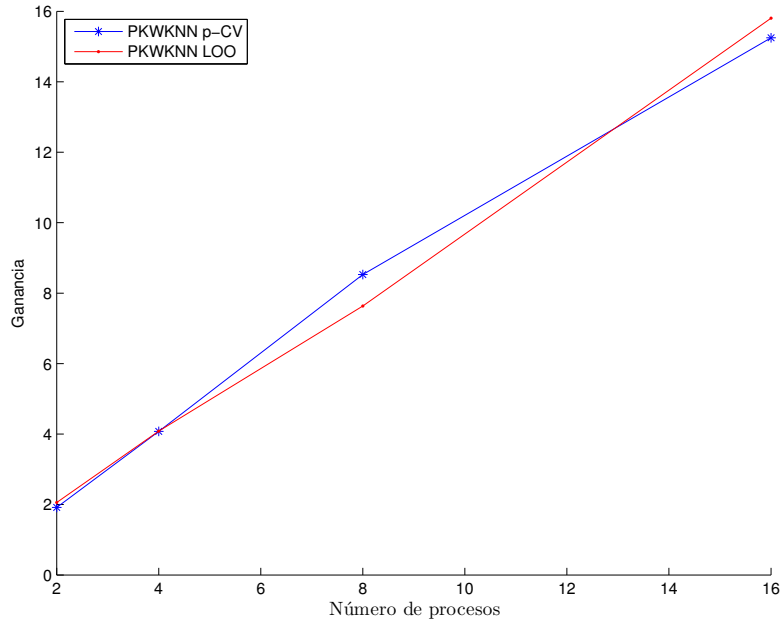


Fig. 4.6: Ganancia (*Speed-up*) obtenida por PKWKNN *p*-CV y PKWKNN LOO con 10240 muestras.

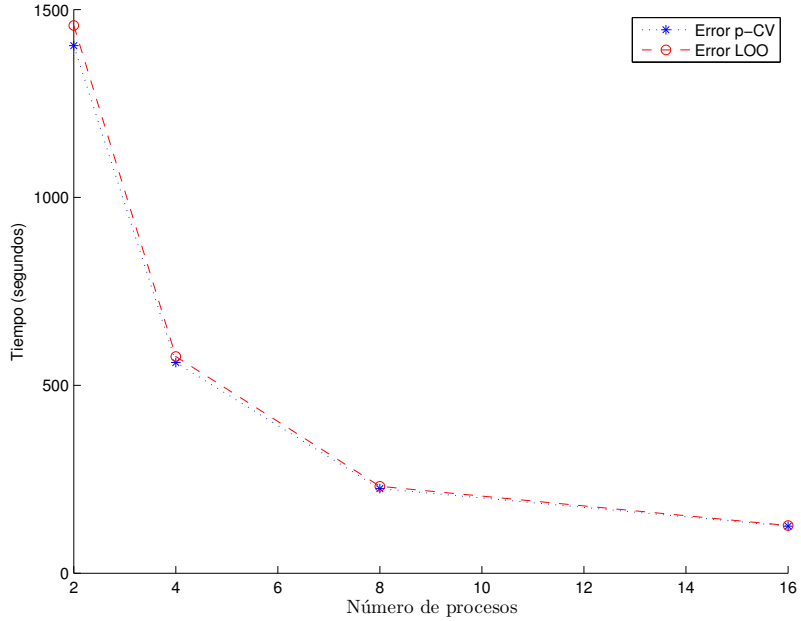


Fig. 4.7: Media de los tiempos de evaluación del error de validación cruzada para PKWKNN frente al Número de Procesos.

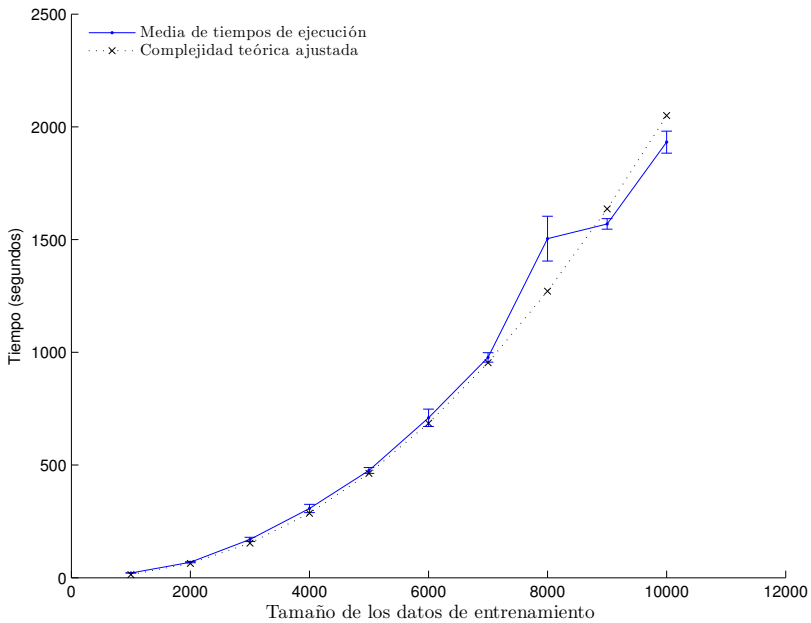


Fig. 4.8: Media y desviación típica de los tiempos para VNS frente al Número de muestras de entrenamiento (ejecución con 8 procesadores)

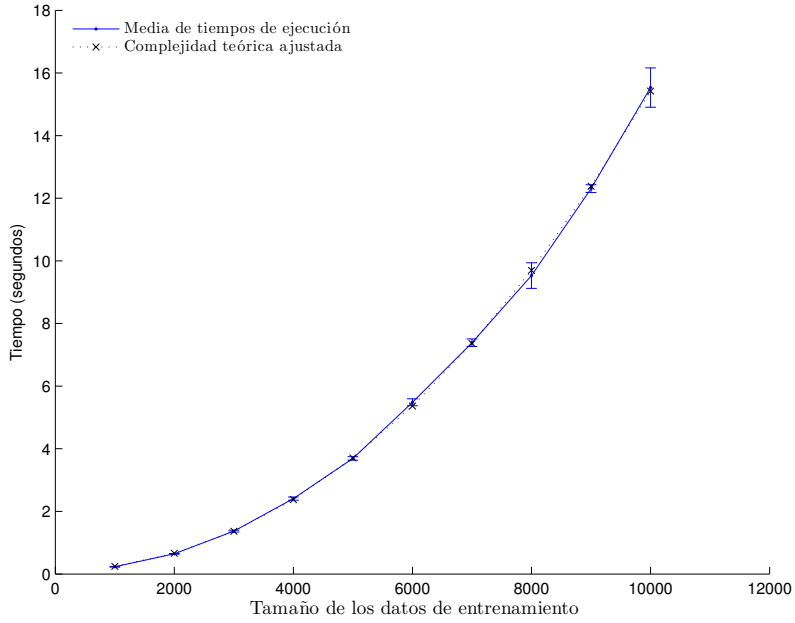


Fig. 4.9: Media y desviación típica de los tiempos para PKWKNN frente al Número de muestras de entrenamiento (ejecución con 8 procesadores)

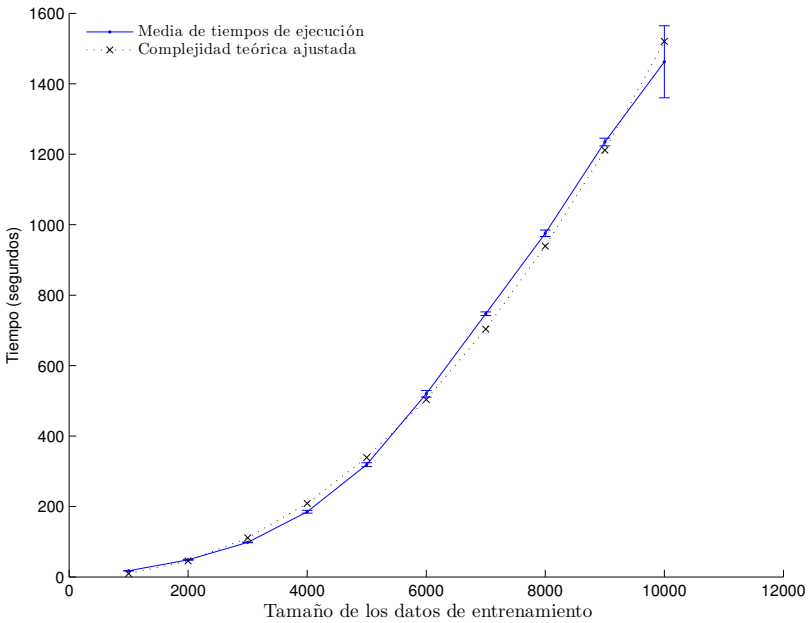


Fig. 4.10: Media y desviación típica de los tiempos para VNS frente al Número de muestras de entrenamiento (ejecución con 16 procesadores)

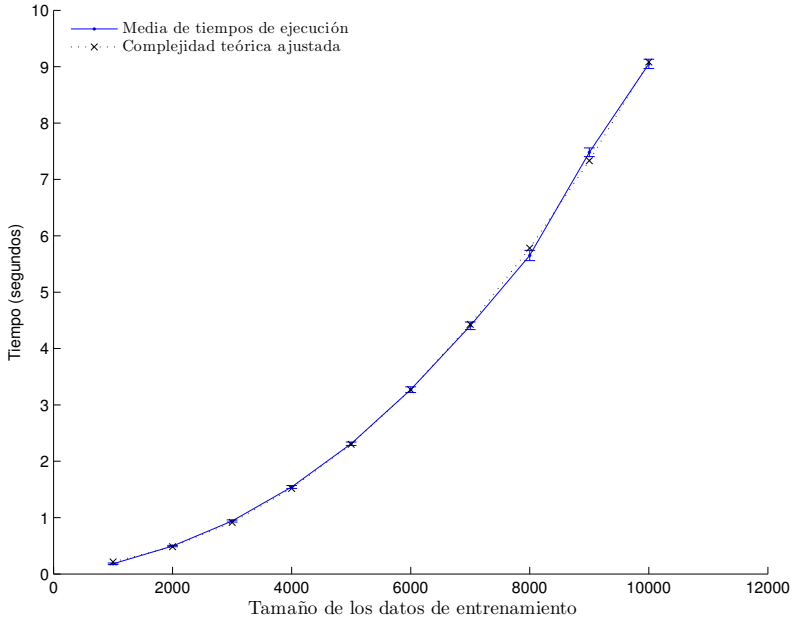


Fig. 4.11: Media y desviación típica de los tiempos para PKWKNN frente al Número de muestras de entrenamiento (ejecución con 16 procesadores)

4.9. Optimización del KWKNN bajo Matlab utilizando CUDA

De cara a implementar técnicas de *kernel* con CUDA, KWKNN se puede beneficiar en gran medida ya que, para clasificación, ya hay estudios en la literatura que aprovechan las características de este tipo de procesamiento aplicado al algoritmo K-NN [Garcia et al., 2008]. En cambio, la implementación de LSSVM es mucho más problemática pues necesita de funciones de inversión de matrices o la descomposición de Cholesky, ambas funcionalidades no contempladas en la librería CUDA y muy difíciles de implementar eficientemente por las propias características de la arquitectura [Volkov and Demmel, 2008].

El diseño de la paralelización de KWKNN en CUDA se centra la paralelización de grano fino de los pasos más costosos del mismo, es decir, la evaluación de los K vecinos más cercanos.

Lo más costoso computacionalmente en el uso de KWKNN es la optimización de los parámetros del *kernel* y el hiperparámetro K. La optimización se realiza con respecto al error de validación cruzada. Por eficiencia, se implementaron funciones específicas para la evaluación del *kernel*, el error de validación cruzada y la búsqueda de los K vecinos más cercanos de un punto. Esto implica una importante diferencia con respecto a [Garcia et al., 2008], ya que para optimizar los parámetros con respecto al error de validación cruzada se va a evaluar repetidas veces el K-NN con los mismos datos. Esto es debido a que el K-NN básico no tiene propiamente dicho entrenamiento, salvo que se desee optimizar el parámetro K. En cambio, en el caso de KWKNN por cada conjunto de valores concretos de parámetros del *kernel* se obtiene un espacio de características diferente con sus propia función de distancia.

Una buena práctica de desarrollo en CUDA es intentar trasladar todo el cómputo posible a la tarjeta. En el caso de la implementación del K-NN para clasificación en [Garcia et al., 2008] la función de distancia es fija, la distancia Euclídea, pero en el caso de KWKNN la función de distancia está basada en el *kernel* usado. Esto es especialmente importante dado que el KWKNN proporciona los mejores resultados cuando su salida es una suma ponderada de *kernels*. Por tanto, es necesario utilizar la GPU para:

- La evaluación del la distancia basada en *kernel*.
- La evaluación del KWKNN.

Con estas premisas básicas, los datos más voluminosos que hay que trasladar a la GPU son las muestras de entrenamiento X_t, Y_t y los nuevos puntos a evaluar X, mientras que el resto de datos usados, que serán los que podrán variar: número de vecinos K del KWKNN, descripción del *kernel* y valores de hiperparámetros.

Denominaremos CUDA-KWKNN a la nueva implementación realizada, que además está desarrollada bajo Matlab extendiendo CUBLASMEX (descrito en el Capítulo 6) con las funciones necesarias que no estén en ella.

4.9.1. Kernels

Como se puede ver en la Sección 4.2 de esta memoria, una función *kernel* $k(x,x')$ es un producto escalar definido en un *espacio de características* del cual no se necesita más descripción en principio. El KWKNN se basa en que se puede definir la norma cuadrática D^2 de 2 puntos del *espacio de características* a partir del *kernel* k mediante la siguiente relación:

$$D^2(x,x') = k(x,x) + k(x',x') - 2k(x,x') \tag{4.14}$$

En el Capítulo 5 se ve que los *kernels* más efectivos que se han utilizado son aquellos resultantes de realizar sumas ponderadas de los siguientes "*kernels* básicos"de la literatura:

$$k_{linear}(x_i,x_j;\{\sigma\}) =< x_i,x_j >) \tag{4.15}$$

$$k_{gauss}(x_i,x_j;\{\sigma\}) = \exp\left(-\frac{1}{\sigma^2}\|x_i-x_j\|^2\right) \tag{4.16}$$

$$k_{ratquad}(x_i,x_j;\{\rho,\beta\}) = \left(1+\frac{\|x_i-x_j\|^2}{2\rho\beta^2}\right)^{-\rho} \tag{4.17}$$

$$k_{\lambda-periodic}(x_i,x_j;\{\theta\}) = \exp\left(-2\frac{sin(\frac{\pi}{\lambda}(x_i-x_j))^2}{\theta^2}\right) \tag{4.18}$$

que son respectivamente el *kernel* Lineal, Gaussiano, Racional Cuadrático y λ-periódico, donde σ,ρ,β,θ son parámetros a optimizar de dichos *kernels* mientras que el parámetro del periodo λ es fijo, por lo que hay realmente un *kernel* λ-periódico distinto por valor de periodo.

Un *kernel* suma ponderada k_{suma} de M de los anteriores *kernels* toma la forma:

$$k_{suma}(x,x') = \sum_{i=1}^{M} \alpha_i k_i(x,x') \tag{4.19}$$

donde $\alpha_i,\ i = 1,\cdots,M$ son los factores de escala asociado a cada *sub-kernel*, los cuales deben ser optimizados para cada k_{suma} junto con los parámetros de cada sub-*kernel*. Por ejemplo, un *kernel*:

$$k_{suma}(x,x') = \alpha_1 k_{gauss}(x_i,x_j;\{\sigma_1\}) + \alpha_2 k_{\lambda_2-periodic}(x_i,x_j;\{\theta\}) \tag{4.20}$$

tendrá como parámetros a optimizar $\alpha_1, \sigma_1, \alpha_2, \theta$ y como parámetro fijo λ_2.

4.9.2. Implementación de Kernels y evaluación de la distancia basada en kernels en CUDA

La implementación en GPU de la evaluación de *kernels* para KWKNN no tiene la misma flexibilidad que la que está implementada en Matlab, no pudiendo definirse *kernels* con componentes arbitrarios, además, hay que tener en cuenta que la memoria de la GPU es más pequeña que la un PC (p.e. la máquina de pruebas tiene 8 GB de RAM mientras que la tarjeta gráfica dispone de 512 MB, es decir, es 16 veces menor). En la implementación de Matlab, cuya interfaz de llamadas se quiere mantener en la medida de lo posible, se evalúa la norma cuadrática basada en un *kernel* de entrada arbitrario entre 2 conjuntos de puntos, y se utilizan la operaciones sobre matrices de Matlab durante la evaluación para obtener una mayor eficiencia. Este enfoque básico se mantiene pero la implementación realizada en GPU tiene en cuenta las limitaciones de memoria y flexibilidad de la siguiente manera:

1. Los *kernels* con los que se trabajarán serán del tipo *kernels* suma ponderada, cuyos componentes podrán ser el *kernel* Gaussiano, el *kernel* λ-periódico y el *kernel* Lineal (que es el producto escalar de las entradas).

2. Dada la limitación de memoria de la GPU, se usará el mínimo de memoria posible en la evaluación, sólo trabajando con los datos de entrada (2 conjuntos de puntos) y la matriz de salida.

3. La evaluación de la norma cuadrática es independiente entre cada componente de la matriz de salida (es decir, pares de de puntos de entrada), por lo que se le asignará una hebra a cada una. Las hebras se organizan en una rejilla de 2 dimensiones de $N_1 \times N_2$.

Un *kernel* suma ponderada queda descrito por 3 vectores:

1. El vector con los índices de los *kernels* componentes de la suma.

2. El vector con los parámetros de los *kernels* componentes de la suma (factor de escala + parámetros propios del sub-*kernel*).

3. El vector con el parámetro λ de cada *kernel* componente de la suma, que sólo se utiliza en el caso de que el componente sea un *kernel* λ-periódico.

El pseudo-código de la función CUDA de evaluación de la norma cuadrática basada en *kernels* se puede ver en el Algoritmo 4.5.

ENTRADA: X_1, conjunto de N_1 puntos de entrada ($N_1 \times d$, memoria global de la GPU)
ENTRADA: X_2, conjunto de N_2 puntos de entrada ($N_2 \times d$, memoria global de la GPU)
ENTRADA: k_{suma}, vector de índices de *kernels* componentes de la suma (memoria global de la GPU)
ENTRADA: Θ, vector de parámetros del *kernel* suma (memoria global de la GPU)
ENTRADA: Λ, vector de parámetros λ de los *kernels* componentes de la suma (memoria global de la GPU)

SALIDA: D, norma cuadrática basada en el *kernel* k_{suma} con los parámetros Θ y Λ de los conjuntos X_1 e X_2 ($N_1 \times N_2$, memoria global de la GPU)

Determinación de los índices (I,J) de la hebra en la rejilla.
$D(I,J) = 0$
$x_1 = X_1(I,:)$
$x_2 = X_2(J,:)$
para $k_i \in k_{suma}$ **hacer**
  si k_i es el *kernel* Lineal **entonces**
    Obtención del factor de escala α_i.
    $k_1 =< x_1,x_1 >$, $k_2 =< x_2,x_2 >$, $k_3 =< x_1,x_2 >$
  fin si
  si k_i es el *kernel* Gaussiano **entonces**
    Obtención del factor de escala α_i y del parámetro σ_i de Θ.
    $k_1 = k_{gauss}(x_1,x_1;\sigma_i)$, $k_2 = k_{gauss}(x_2,x_2;\sigma_i)$
    $k_3 = k_{gauss}(x_1,x_2;\sigma_i)$
  fin si
  si k_i es el *kernel* Racional Cuadrático **entonces**
    Obtención del factor de escala α_i y del parámetro ρ_i, β_ide Θ.
    $k_1 = k_{ratquad}(x_1,x_1;\rho_i,\beta_i)$, $k_2 = k_{ratquad}(x_2,x_2;\rho_i,\beta_i)$
    $k_3 = k_{ratquad}(x_1,x_2;\rho_i,\beta_i)$
  fin si
  si k_i es el *kernel* λ-periódico **entonces**
    Obtención del factor de escala α_i y
    del parámetro θ_i de Θ,
    y del periodo λ_i de Λ.
    $k_1 = k_{\lambda_i-periodic}(x_1,x_1;\theta_i)$, $k_2 = k_{\lambda_i-periodic}(x_2,x_2;\theta_i)$
    $k_3 = k_{\lambda_i-periodic}(x_1,x_2;\theta_i)$
  fin si
  $D(I,J) = D(I,J) + \alpha_i(k_1 + k_2 - 2k_3)$.
fin para

Alg. 4.5: Evaluación de la norma cuadrática basada en *kernels* (función CUDA)

4.9.3. Cálculo de los vecinos más cercanos y evaluación del KWKNN en CUDA

Vamos a suponer X un conjunto de M puntos a evaluar con el K-NN, X_T el conjunto de N muestras que entrenan al modelo, k_{suma} el vector de índices de *kernels* componentes de la suma, Θ el vector de parámetros del *kernel* suma y Λ el vector de parámetros λ de los *kernels* componentes de la suma (todos en memoria global de la GPU). Una vez calculada la norma cuadrática basada en *kernel* entre los puntos de los 2 conjuntos (con la función anteriormente descrita en 4.5) que supondremos en la matriz D, $M \times N$, hay que obtener los K puntos de X_T más cercanos a cada punto $x \in X$. Esto implica obtener los índices de dichos puntos en X_T. Este cálculo es independiente para cada punto $x \in X$, pudiendo asignar a cada hebra la obtención de los K vecinos para cada punto x, organizando las hebras en un rejilla de 1 dimensión $M \times 1$ (una hebra por punto a evaluar). El algoritmo escogido simplemente realiza K búsquedas lineales $O(N)$ para obtener índices de los puntos, por lo que se deduce que cada hebra ejecuta un procedimiento de complejidad $O(KN)$, pero al ser normalmente $K \ll N$ se puede considerar que realmente es $O(N)$.

Dado un conjunto de muestras de una función $\{(x_1, y_1), \ldots, (x_N, y_N)\} \in \mathbb{R}^d \times \mathbb{R}$, donde N es el número de muestras y d es la dimensionalidad de la entrada, el KWKNN calcula la salida para un nuevo punto de entrada $\hat{x}$ como:

$$f(\hat{x}) = \frac{\sum_{i=1}^{K} y_i^{\hat{x}} w^K\left(\hat{x}, x_i^{\hat{x}}\right)}{\sum_{i=1}^{K} w^K\left(\hat{x}, x_i^{\hat{x}}\right)}, \tag{4.21}$$

$$donde\ w^K\left(\hat{x}, x_i^{\hat{x}}\right) = \left[1 - \frac{D\left(\hat{x}, x_i^{\hat{x}}\right)^2}{D\left(\hat{x}, x_{K+1}^{\hat{x}}\right)^2}\right]^2 \tag{4.22}$$

donde D es una función de distancia basada en *kernel*, $x_i^{\hat{x}}$ es el punto de entrenamiento i-ésimo vecino más cercano a $\hat{x}$ de acuerdo a la distancia basada en *kernel* D, $y_i^{\hat{x}}$ su salida correspondiente y $w^K\left(\hat{x}, x_i^{\hat{x}}\right)$ es su peso asignado.

En los apartados anteriores se ha descrito nuestra implementación del cálculo de la norma cuadrática basada en *kernel* y de la obtención de los K vecinos más cercanos por puntos en la GPU, por lo que tendremos ya calculadas la norma cuadrática basada en *kernel* entre $\hat{x}$ y las muestras así como los K vecinos más cercanos a las mismas. En la implementación realizada no se calcula punto a punto de entrada el KWKNN sino para un conjunto X. Este cálculo es independiente para punto $x \in X$, pudiendo asignar a cada hebra la evaluación del KWKNN para

cada punto x, organizando las hebras en un rejilla de dimensión $M \times 1$ (una hebra por punto a evaluar).

4.9.4. Error de validación cruzada del KWKNN en CUDA

La implementación del cálculo del error de validación cruzada del KWKNN en GPU se basa exclusivamente en una modificación del cálculo de la norma cuadrática basada en *kernel*. Las mayores diferencias del Algoritmo 4.6 con respecto al cómputo normal descrito en el Algoritmo 4.5 son que:

1. Los conjuntos de puntos a evaluar y el de muestras son el mismo.

2. No hay que tener en cuenta los puntos de su misma partición de un punto como posibles K vecinos más cercanos.

El método elegido para no tener en cuenta los puntos de una misma partición de un punto como posibles K vecinos más cercanos es definiendo como *infinita* la norma cuadrática de un punto con respecto a los puntos de su misma partición. Se controlo la pertenencia o no a la partición mediante los índices en la matriz de distancia. Con esto, y aplicando posteriormente las funciones para cálculo de los K vecinos más cercanos y de salida KWKNN se consigue la salida de validación cruzada de orden l. Para hacer más eficiente el procedimiento, se implementó también el cálculo final del error en GPU (usando las funciones de CUBLAS *cublasSaxpy* y *cublasSdot*).

4.9.5. Experimentos

La máquina utilizada en los experimentos fue un INTEL CORE 2 QUAD 2.83GHZ, con 8 GB de memoria RAM DDR, con una tarjeta GEFORCE 9800 GTX (128 procesadores de flujo a 1688 MHz, 512 MB de memoria GDDR3). Debido a la propias capacidades de la tarjeta, los cálculos en punto flotante en la misma serán realizada en simple precisión (32 bits). Los experimentos consistieron en comparar la implementación de KWKNN sólo en Matlab con otra idéntica que utiliza funciones-cuda para el cálculo de la distancia basada en *kernel*, los K vecinos más cercanos y la evaluación del KWKNN. Los experimentos fueron una repetición parcial de los de la Sección 5.2.2 para los conjunto de datos de carga eléctrica en California y del río Snake.

El conjunto de datos de carga eléctrica usado en estos experimentos corresponde a la carga eléctrica en California desde el año 1998 los primeros 33 días del 2003[6]. Los datos tienen un total de 1858 valores (unos 5 años), y nuestro objetivo es predecir los últimos 365 valores (1 año). Los *kernels* suma ponderada usados en estos experimentos fueron:

[6] `http://www.ucei.berkeley.edu/CSEM/datamine/ISOdata/`

ENTRADA: X, conjunto de N puntos de entrada ($N \times d$, memoria global de la GPU)
ENTRADA: k_{suma}, vector de índices de *kernels* componentes de la suma (memoria global de la GPU)
ENTRADA: Θ, vector de parámetros del *kernel* suma (memoria global de la GPU)
ENTRADA: Λ, vector de parámetros λ de los *kernels* componentes de la suma (memoria global de la GPU)

SALIDA: D, norma cuadrática basada en el *kernel* k_{suma} con los parámetros Θ y Λ del conjunto X ($N \times N$, memoria global de la GPU) para error de validación cruzada de orden l
Determinación de los índices (I,J) de la hebra en la rejilla.
$D(I,J) = 0$
$x_1 = X(I,:)$
$x_2 = X(J,:)$
si los índices I y J no pertenencen a la misma partición **entonces**
  para $k_i \in k_{suma}$ **hacer**
    si k_i es el *kernel* Lineal **entonces**
      Obtención del factor de escala α_i.
      $k_1 =< x_1, x_1 >$, $k_2 =< x_2, x_2 >$, $k_3 =< x_1, x_2 >$
    fin si
    si k_i es el *kernel* Gaussiano **entonces**
      Obtención del factor de escala α_i y del parámetro σ_i de Θ.
      $k_1 = k_{gauss}(x_1, x_1; \sigma_i)$, $k_2 = k_{gauss}(x_2, x_2; \sigma_i)$
      $k_3 = k_{gauss}(x_1, x_2; \sigma_i)$
    fin si
    si k_i es el *kernel* Racional Cuadrático **entonces**
      Obtención del factor de escala α_i y del parámetro ρ_i, β_ide Θ.
      $k_1 = k_{ratquad}(x_1, x_1; \rho_i, \beta_i)$, $k_2 = k_{ratquad}(x_2, x_2; \rho_i, \beta_i)$
      $k_3 = k_{ratquad}(x_1, x_2; \rho_i, \beta_i)$
    fin si
    si k_i es el *kernel* λ-periódico **entonces**
      Obtención del factor de escala α_i y
      del parámetro θ_i de Θ,
      y del periodo λ_i de Λ.
      $k_1 = k_{\lambda_i - periodic}(x_1, x_1; \theta_i)$, $k_2 = k_{\lambda_i - periodic}(x_2, x_2; \theta_i)$
      $k_3 = k_{\lambda_i - periodic}(x_1, x_2; \theta_i)$
    fin si
    $D(I,J) = D(I,J) + \alpha_i(k_1 + k_2 - 2k_3)$.
  fin para
si no
  $D(I,J) = +\infty$.
fin si

Alg. 4.6: Evaluación de la norma cuadrática basada en *kernels* para error de validación cruzada de orden l (función CUDA)

$$k_1(x_i,x_j;\{s_1,s_2,s_3,\sigma_g,\sigma_7,\sigma_{364}\}) = \begin{array}{l} s_1\cdot k_{gauss}(x_i,x_j;\sigma_g) \\ +s_2\cdot k_{7-periodic}(x_i,x_j;\sigma_7) \\ +s_3\cdot k_{364-periodic}(x_i,x_j;\sigma_{364}) \end{array}$$

$$k_2(x_i,x_j;\{s_1,s_2,s_3,\alpha,\beta,\sigma_7,\sigma_{364}\}) = \begin{array}{l} s_1\cdot k_{ratquad}(x_i,x_j;\alpha,\beta) \\ +s_2\cdot k_{7-periodic}(x_i,x_j;\sigma_7) \\ +s_3\cdot k_{364-periodic}(x_i,x_j;\sigma_{364}) \end{array}$$

La serie temporal del flujo mensual del río Snake[7] tiene un total de 1092 valores (1092 meses = 91 años), la primera mitad de los mismos se usaron para entrenamiento y el resto para test (546 valores en cada caso $\simeq$ 45 años). Los *kernels* suma ponderada usados en estos experimentos fueron:

$$k_1(x_i,x_j;\{s_1,s_2,\sigma_g,\sigma_{12}\}) = \begin{array}{l} s_1\cdot k_{gauss}(x_i,x_j;\sigma_g) \\ +s_2\cdot k_{12-periodic}(x_i,x_j;\sigma_{12}) \end{array}$$

$$k_2(x_i,x_j;\{s_1,s_2,\alpha,\beta,\sigma_{12}\}) = \begin{array}{l} s_1\cdot k_{ratquad}(x_i,x_j;\alpha,\beta) \\ +s_2\cdot k_{12-periodic}(x_i,x_j;\sigma_{12}) \end{array}$$

Estos *kernels* se utilizan en modelos KWKNN optimizados mediante el algoritmo VNS expuesto en la Sección 4.2. Se generaron modelos optimizados tanto frente al error 10-CV como al LOO. El parámetro K de cada KWKNN óptimo se buscó en un rango $[2,20]$ para cada conjunto de valores de parámetros del *kernel*, añadiendo un tiempo mínimo a la evaluación de cada conjunto de valores de parámetros del *kernel* considerado como se expuso en la Sección 4.2. El algoritmo VNS realizó 50 iteraciones con vecindades de tamaño 10, lo que implica 500 evaluaciones de la validación cruzada.

Los resultados obtenidos usando error de validación cruzada LOO están recogidos en la Tabla 4.2 y usando error de validación cruzada de orden 10 en la Tabla 4.3. Los tiempos de entrenamiento reflejan la mayor complejidad de cálculo del segundo *kernel* (el que usa el *kernel* Racional Cuadrático) frente al primero para ambas series. Esto se puede apreciar especialmente en los tiempos de entrenamiento para una misma serie con la implementación de Matlab. Queda reflejado cómo el cómputo en GPU del error de validación cruzada de orden 10 es mucho más eficiente que la del error LOO, ya que la implementación realiza muchos menos cálculos, algo que, aunque algoritmicamente no se refleja, sí tiene incidencia al requerir menos hebras que manejar en un momento dado. Se ha obtenido una importante mejora en la velocidad de cálculo, sobre todo con el conjunto de datos de carga eléctrica en California, manteniendo a la vez unos resultados de

[7] `http://www-personal.buseco.monash.edu.au/hyndman/TSDL/hydrology.html`

precisión aceptable. Los factores limitantes son la función CUDA de cálculo los K vecinos más cercanos y las transferencias de entre memoria principal y de dispositivo. Para conjuntos de datos de unos pocos miles de datos (aquí el límite está marcado por la memoria disponible en el dispositivo), se pueden esperar grandes mejoras de tiempos utilizando esta implementación de KWKNN con respecto a la de Matlab pura (coincidiendo con los resultados de [Garcia et al., 2008] para K-NN para clasificación).

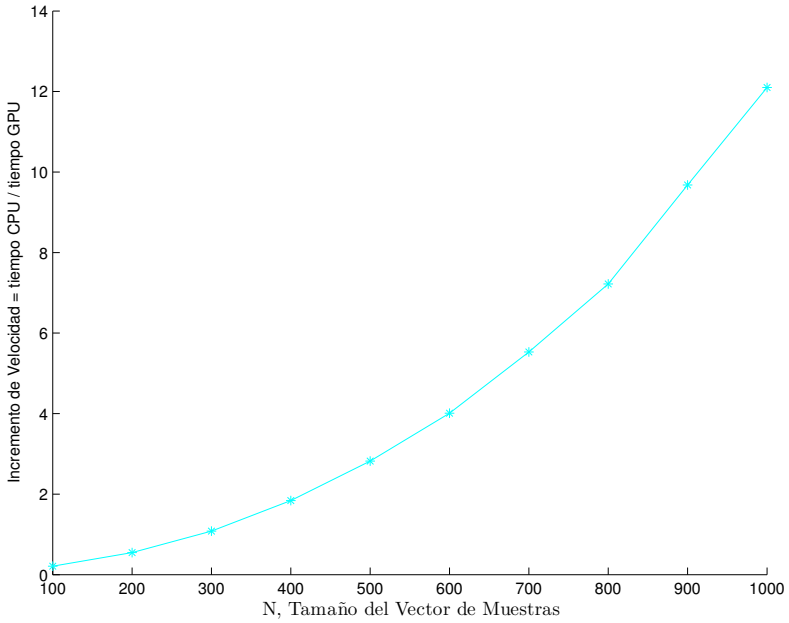


Fig. 4.12: Ganancia obtenida al implementar los K vecinos más cercanos en GPU (entradas: 2 conjuntos de igual tamaño N)

Tabla 4.2: Resultados obtenidos utilizando *kernels* específicos al problema con la optimización de K y error de LOO usando el algoritmo KWKNN implementado en Matlab y Matlab+CUDA

Río Snake				
Impl. + *kernel*	opt. tiempo	rmse-loo	K	rmse test
CUDA 1	1.50e+01	9.09e+02	20	8.67e+02
MATLAB 1	3.49e+01	1.19e+03	3	9.51e+02
Ganacia	2.32e+00			
CUDA 2	1.60e+01	9.05e+02	20	8.62e+02
MATLAB 2	6.15e+01	8.67e+02	20	7.77e+02
Ganacia	3.84e+00			
Carga eléctrica				
Impl. + *kernel*	opt. tiempo	rmse-loo	K	rmse test
CUDA 1	3.80e+01	2.40e+04	13	3.52e+04
MATLAB 1	4.31e+02	2.40e+04	13	3.52e+04
Ganacia	1.13e+01			
CUDA 2	4.30e+01	2.40e+04	13	3.52e+04
MATLAB 2	5.91e+02	2.45e+04	20	3.60e+04
Ganacia	1.37e+01			

Tabla 4.3: Resultados obtenidos utilizando *kernels* específicos al problema con la optimización de K y error de validación cruzada (l=10) usando el algoritmo KWKNN implementado en Matlab y Matlab+CUDA

Río Snake				
Impl. + *kernel*	opt. tiempo	rmse-10-CV	K	rmse test
CUDA 1	1.55e+01	1.79e+03	16	9.15e+02
MATLAB 1	3.89e+01	9.06e+02	19	8.72e+02
Ganacia	2.51e+00			
CUDA 2	1.72e+01	1.79e+03	16	8.22e+02
MATLAB 2	6.00e+01	8.11e+02	20	7.77e+02
Ganacia	3.49e+00			
Carga eléctrica				
Impl. + *kernel*	opt. tiempo	rmse-10-CV	K	rmse test
CUDA 1	1.89e+01	6.71e+04	8	3.79e+04
MATLAB 1	4.10e+02	4.06e+04	20	3.44e+04
Ganacia	2.17e+01			
CUDA 2	1.97e+01	6.71e+04	8	3.78e+04
MATLAB 2	6.08e+02	4.06e+04	20	3.43e+04
Ganacia	3.09e+01			

5. KERNELS ESPECÍFICOS PARA UN PROBLEMA DE REGRESIÓN.

Los *kernels* son el núcleo (si se permite la redundancia) de los métodos *kernel*. Aunque hay una amplia bibliografía relacionada con métodos *kernel*, y específicamente para métodos *kernel* aplicados a aproximación funcional, en pocos trabajos se prescinde del *kernel* Gaussiano o de Funciones de Base Radial (RBF). Estos *kernels* presentan buenas propiedades para este tipo de problemas debido a su capacidad de generalización y de interpolación. Los modelos LSSVM y los Procesos Gaussianos con *kernel* Gaussiano son equivalentes a Redes de Funciones de Base Radial (RBFNN) con un centro por punto y con radio único. Su único elemento diferenciador con respecto a RBFNN es el método de entrenamiento, que permite controlar el sobreajuste mediante un factor de regularización. Esto no tiene por qué ser negativo, aunque traslade la mayor parte del peso del entrenamiento a la selección de características, dejando el resto de la dificultad en el ajuste de parámetros del modelo y del kernel.

Este capítulo enfoca el kernel como parte fundamental de un modelo basado en kernel, incidiendo especialmente en la importancia de adaptarlos a un problema dado. Se describe cómo abordar el proceso de creación de un kernel específico tanto para un problema del que no tenemos conocimiento previo como para problemas en los que sí tenemos dicho conocimiento previo. Se mostrarán una serie de conclusiones interesantes acerca de este difícil problema.

5.1. Kernels específicos: discusión

Se puede demostrar que las dos principales métodos *kernel* para regresión, LSSVM y Procesos Gaussianos, son realmente equivalentes (ver [Suykens et al., 2002]). Sin embargo, el enfoque Bayesiano de Procesos Gaussianos y su incidencia en la optimización de los parámetros del *kernel* parece diferenciarlas. Esto

no es más que aparente, ya que el enfoque Bayesiano también se ha aplicado a LSSVM [Suykens et al., 2002], si bien es cierto que su formulación típica es más cercana al entorno de optimización. En la práctica, el proceso de trabajo con todos los métodos es el mismo:

1. Se tiene que plantear un *kernel* adecuado al problema.

2. Se tiene que hacer una estimación del error en los datos o la precisión alcanzable y reflejarla en los hiper-parámetros a optimizar: ε y C en el caso de SVR, γ en el caso de LSSVM, y el *jitter* o el componente de ruido Gaussiano en el caso de Procesos Gaussianos, ver Ecuación (2.34).

3. Se tiene que hacer una estimación de los parámetros del *kernel*, bien mediante el error validación cruzada del modelo sobre los datos de entrenamiento (existen fórmulas de coste reducido para calcular esto, para LSSVM ver [An et al., 2007]) o usando un criterio Bayesiano para estimar la verosimilitud (*likelihood*) del modelo.

Una vez entrenados una serie de modelos, debe elegirse el mejor, atendiendo a algún criterio. Todos los modelos presentan los mismos problemas principales:

1. Su coste computacional aumenta en gran proporción con el aumento del número de ejemplos de entrenamiento.

2. El modelo resultante es *pesado*, o mejor dicho *voluminoso*, en el sentido de que incluye todos los puntos de entrenamiento.

Estas son las cuestiones en las que más se ha incidido en la bibliografía. Se pueden encontrar excelentes análisis en [Scholkopf and Smola, 2001], [Suykens et al., 2002], [MacKay, 1998] y [Rasmussen and Williams, 2005].

Sin embargo hay un problema del entrenamiento que ha sido sistemáticamente ignorado, excepto tal vez en la literatura de Procesos Gaussianos: la adaptación del *kernel* al problema concreto. En la inmensa mayoría de los casos siempre se utilizan las mismas funciones de *kernel* para SVR y LSSVM: el *kernel* Gaussiano o RBF (ver 2.21). En el caso de Procesos Gaussianos, las elecciones por excelencia también son variantes del *kernel* RBF [Brahim-Belhouari and Bermak, 2004]:

$$C(x_i,x_j;\{\theta_1,\rho_{1\cdots d},\sigma_l,\sigma_e\}) = \theta_1 \exp\left[-\sum_{k=1}^{d}\frac{1}{\rho_k^2}(x_{ik}-x_{jk})^2\right] + \sigma_l^2\langle x_i,x_j\rangle + \delta_{ij}\sigma_e^2. \tag{5.1}$$

$$C(x_i,x_j;\{\theta_1,\theta_2,\sigma_l,\sigma_e\}) = \theta_1 \exp\left[-\frac{1}{{\theta_2}^2}\|x_i - x_j\|^2\right] + \sigma_l^2\langle x_i,x_j\rangle + \delta_{ij}\sigma_e^2. \quad (5.2)$$

donde x_i y x_j son dos puntos de entrada (usualmente vectores), $(\theta_1,\theta_2,\rho_{1\cdots d},\sigma_l,\sigma_e)$ son parámetros de los *kernels* y δ_{ij} es la función delta de Dirac.

La utilización de *kernels* específicos para problemas de regresión apenas se ha tratado en la literatura. Ha habido algunos intentos de crear modelos con funciones *kernel* compuestas en forma de combinación lineal ponderada: [Cristianini et al., 2002], [Crammer et al., 2002], [Ong et al., 2005], [Sonnenburg et al., 2005]. También se ha intentado aplicar Programación Genética para obtener funciones *kernels* adaptadas a problemas específicos de clasificación: [Howley and Madden, 2005], [Gagné et al., 2006], [Methasate and Theeramunkong, 2007] y [Sullivan and Luke, 2007]. Las conclusiones fueron que aunque se pueden obtener modelos más ajustados, el coste computacional hace prohibitivo en la mayoría de los casos su aplicación sistemática.

En este Capítulo se van a presentar 2 enfoques para la creación de *kernels* específicos para problemas de regresión:

1. Basado en conocimiento experto: se presenta una metodología específicamente para series temporales que ha dado excelentes resultados en predicción a largo plazo.

2. Sin conocimiento experto: se presenta un estudio de la aplicación de Programación Genética para obtención de *kernels* específicos para problemas de regresión.

5.2. Diseño Experto de Kernels específicos para series temporales

5.2.1. Descripción

Un *kernel* específico a un problema debe embeber en su formulación alguna información relevante acerca del fenómeno que genera los datos de los que se dispone. En los problemas de predicción de series temporales en los cuales la serie presenta un cierto comportamiento estacional, la capacidad de predicción a corto y largo plazo de los modelos puede mejorarse mediante la incorporación de la estacionalidad como información adicional en el propio modelo. Como puede verse, en [Herrera et al., 2007b] para modelos LSSVM, incluir la información estacional de la serie en el modelo para predicción recursiva puede mejorar el rendimiento del predictor, e incrementar tanto la capacidad de generalización como la interpretabilidad del mismo.

Nuestro enfoque está basado en la creación de *kernels* específicos al problema mediante suma ponderada de *kernels* clásicos que se pueden encontrar en la

literatura, y que tengan en cuenta la información estacional. La propiedad de clausura de *kernels* con respecto a la suma y producto [Scholkopf and Smola, 2001] asegura que la suma ponderada de *kernels* válidos es un *kernel* válido. El factor de escala se convierte en un parámetro adicional que pondera la importancia de cada sub-*kernel* en la suma. Siguiendo el mismo procedimiento que en [Rasmussen and Williams, 2005], usaremos en este caso el índice de tiempo de la serie ($t = 1,\ldots,N$) como entrada. De esta forma, no es necesario realizar selección de regresores en este caso, confiando en la información incrustada en la función de *kernel* y la optimización de sus parámetros (en nuestro caso, con respecto al error de validación cruzada). Otra característica interesante de este enfoque es que el error no se propaga de un horizonte de predicción hacia otros como ocurre en el caso de usar predicción recursiva.

En este trabajo concretamente, dada una serie temporal $\{Y_t\}_1^N$ de la que se utiliza un cierto número de sus primeros valores para entrenamiento y el resto para test, la metodología que aplicamos es la siguiente:

1. Como primer paso, se deben identificar todos los periodos existentes en la serie a modelar. Esta tarea se puede realizar tanto mediante simple inspección visual como mediante métodos estadísticos o un análisis del espectro (ver [Dutilleul, 2001], [Vlachos et al., 2005] y [Ahdesmäki et al., 2005] para algunos ejemplos de la literatura sobre el tema).

2. Se crea un término base fijo para los *kernels* que se crearán: para cada periodo de longitud λ, incorporamos un término $\lambda - periodic$ (Ecuación (5.3)). La fórmula del *kernel* λ-Periódico es:

$$k_{\lambda-periodic}(t_1,t_2;\{\sigma\}) = \exp\left(-\frac{1}{\sigma^2}\sin\left(\frac{\pi}{\lambda}(t_1-t_2)\right)^2\right) \tag{5.3}$$

   donde t_1 y t_2 son los índices de tiempo de entrada, σ es el único parámetro del *kernel* y λ es un periodo de longitud fija.

3. Con el término base fijo creado, se crean combinaciones del mismo con otros. Aquí utilizaremos los *kernels* Gaussianos y el Racional Cuadrático [1]. Por lo tanto cada serie tiene dos *kernels* candidatos:

$$k_{gauss}(t_1,t_2;\{\sigma\}) = \exp\left(-\frac{1}{\sigma^2}\|t_1-t_2\|^2\right) \tag{5.4}$$

[1] En el trabajo [Rubio et al., 2008] se usa también un *kernel* lineal. Sin embargo, aunque pueda presentar un comportamiento interesante para series con tendencia lineal, no ha probado ser especialmente de ayuda para series estacionarias, al desaparecer su influencia en la suma ponderada en la optimización.

$$k_{ratquad}(t_1,t_2;\{\alpha,\beta\}) = \left(1+\frac{\|t_1-t_2\|^2}{2\alpha\beta^2}\right)^{-\alpha} \tag{5.5}$$

donde t_1 y t_2 son los índices de tiempo de entrada, σ es el único parámetro del *kernel* Gaussiano y α,β son los dos parámetros del *kernel* Racional Cuadrático.

4. Se optimizan los parámetros de cada *kernel* en la sección de entrenamiento de la serie con respecto al error de validación cruzada. El valor del error de validación cruzada se utilizará como criterio de selección de *kernels*.

Esta metodología es realmente una simplificación que se puede generalizar en varios aspectos como, por ejemplo, ampliando el conjunto de los *kernels* usados para las combinaciones o usando otro criterio de valoración de *kernels* distinto a la validación cruzada, como puede ser el criterio Bayesiano para LSSVM de [Suykens et al., 2002]. La metodología generalizada para creación y selección de *kernels* específicos para series temporales basados en estacionalidad viene dada en el Algoritmo 5.1.

Dada la serie temporal $\{Y_t\}_1^N$
Dado K, un conjunto de *kernels* conocidos que no puede contener el *kernel* λ-Periódico

Hallar los periodos $\lambda_1,\lambda_2,\cdots,\lambda_l$ que se identifiquen en la serie
Crear la componente base de los *kernels* como $k_{base}(t_1,t_2) = \sum_{i=1}^{l}\alpha_i k_{\lambda_i-periodic}(t_1,t_2;\{\sigma_i\})$
para Cada posible combinación en suma ponderada de *kernels* en K, k_b **hacer**
  Crear un *kernel* específico $k_j = k_b + k_{base}$
  Evaluar k_j sobre la serie con algún criterio de adaptación
fin para

Devolver el conjunto de *kernels* específicos creados $\{k_j\}_1^{\cdots}$ junto con sus valores del criterio de adaptación

Alg. 5.1: Algoritmo para creación y selección de *kernels* específicos para series temporales basados en estacionalidad

5.2.2. Experimentos

En esta sección, se utilizan tres ejemplos de problemas de predicción a largo plazo de series temporales para poner a prueba la metodología propuesta. En [Rubio et al., 2008] utilizamos una aproximación básica con KWKNN usando *kernels*

específicos al problema con tres series desconocidas para la competición del ESTSP'08. En los experimentos presentados en esta sección no se utilizan las dos primeras series de la competición, sino otras con el fin de proporcionar un análisis más significativo al conocer su origen y significado. La primera de ellas es del consumo eléctrico en California, ampliamente usada en la literatura. La segunda, es el flujo mensual del río Snake, también de uso extendido en la literatura de predicción de series temporales a largo plazo [Herrera et al., 2007b][Herrera et al., 2007a]. La tercera serie es una de las series del ESTSP'08, cuyo gran número de muestras es un ejemplo de problema no puede ser modelado ni con LSSVM ni con KWKNN.

Dada la importancia que tiene el problema concreto en la creación de un *kernel* específico, se describen a continuación tanto los conjuntos de datos utilizados como los *kernels* creados para ellos con la metodología propuesta y comentarios sobre los resultados.

La ejecuciones de los experimentos se hicieron sobre un cluster de 32 nodos con procesadores AMD Opteron de 64 bits a 2.6 GHz con 8 GB de RAM. Sólo se utilizó computación paralela para la tercera serie, que por su complejidad computacional requería el uso de PKWKNN (ver apartado 4.7). Todo el software de KWKNN se implementó partiendo de una implementación propia de LSSVM con el fin de compartir los *kernels* utilizados. La toolbox MPIMEX [Guillen et al., 2008a] se usó en la implementación de PKWKNN para acceder a las funciones necesarias de la librería MPI.

5.2.2.1. Serie temporal de consumo eléctrico en California

La predicción del consumo en una red eléctrica es un problema de gran importancia desde el punto de vista económico, energético y social. El conjunto de datos de consumo eléctrico usado en estos experimentos corresponde a la red eléctrica en California desde el año 1998 hasta los primeros 33 días del 2003[2]. Los datos tienen un total de 1858 valores (unos 5 años), y nuestro objetivo es predecir los últimos 365 valores (1 año).

La serie tiene claros periodos semanales y anuales (ver Figura 5.1). Las periodicidades anteriormente nombradas fueron confirmadas usando un método similar al expuesto en [Kourentzes and Crone, 2008]. Usando la metodología expuesta en la sección 5.2.1, se crearon dos *kernels* a medida cuyas entradas son índices de tiempo:

$$\begin{aligned} k_1(t_1,t_2;\{s_1,s_2,s_3,\sigma_g,\sigma_7,\sigma_{364}\}) &= s_1 \cdot k_{gauss}(t_1,t_2;\sigma_g) \\ &\quad + s_2 \cdot k_{7-periodic}(t_1,t_2;\sigma_7) \\ &\quad + s_3 \cdot k_{364-periodic}(t_1,t_2;\sigma_{364}) \end{aligned}$$

[2] http://www.ucei.berkeley.edu/CSEM/datamine/ISOdata/

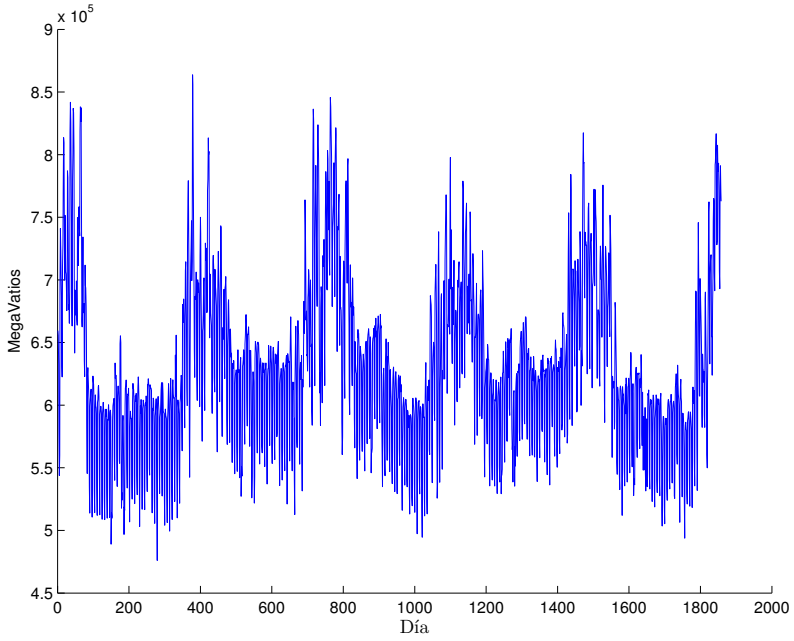


Fig. 5.1: Datos de Consumo eléctrico en California

$$\begin{aligned} k_2(t_1,t_2;\{s_1,s_2,s_3,\alpha,\beta,\sigma_7,\sigma_{364}\}) &= s_1 \cdot k_{ratquad}(t_1,t_2;\alpha,\beta) \\ &\quad +s_2 \cdot k_{7-periodic}(t_1,t_2;\sigma_7) \\ &\quad +s_3 \cdot k_{364-periodic}(t_1,t_2;\sigma_{364}) \end{aligned}$$

donde t_1 y t_2 son los índices de tiempo de entrada, $\{s_1,s_2,s_3,\sigma_g,\sigma_7,\sigma_{364}\}$ son los parámetros del *kernel* k_1 y $\{s_1,s_2,s_3,\alpha,\beta,\sigma_7,\sigma_{364}\}$ son los parámetros del *kernel* k_2.

Se utilizaron ambos *kernels* para generar modelos KWKNN optimizados mediante el algoritmo VNS expuesto en la Sección 4.2. Se generaron dos modelos por cada *kernel* al optimizarse tanto frente al error 10-CV como al LOO. El algoritmo VNS realizó 50 iteraciones con vecindades de tamaño 10, lo que implica 500 evaluaciones de la validación cruzada, que duraron entre 13,4 y 17,5 minutos en este caso. El parámetro optimizado K de cada KWKNN se busca en un rango $[2,20]$ para cada conjunto de valores de parámetros del *kernel*, añadiendo un tiempo mínimo a la evaluación de cada conjunto de valores de parámetros del *kernel* considerado como se expuso en la Sección 4.2.

Los resultados del error de entrenamiento (ver Tabla 5.1) muestran que los modelos optimizados con error LOO padecen de sobreajuste al conjunto de entrenamiento repercutiendo en mayores errores en el conjunto de test que los modelos optimizados con 10-CV. Este resultado es de esperar ya que para problemas de

predicción a largo plazo es más adecuado minimizar el error de aproximación con respecto a bloques de valores de un cierto tamaño con el fin de obtener modelos con buenas capacidades de generalización. Todo esto hace preferible en nuestro caso optimizar con respecto al error 10-CV (u otro de menor orden) que con respecto al error LOO.

Los *kernels* específicos son independientes del tipo de modelo, por consiguiente, se han entrenado tanto modelos KWKNN como LSSVM con fines comparativos. La optimización de los parámetros de los modelos LSSVM se realizó, al igual que en el caso de los modelos KWKNN, frente al error 10-CV y al LOO, aunque el algoritmo de optimización usado fue el Gradiente Conjugado, que también se limitó a 500 evaluaciones del error validación cruzada. Los tiempos de ejecución en este caso son mucho mayores, como se puede apreciar en la Tabla 5.2 oscilando entre 2,75 y 5 horas. La diferencia en rendimiento entre modelos optimizados frente a LOO y 10-CV no es claramente apreciable en este caso.

Para poder aplicar el enfoque clásico de predicción recursiva, se tuvo que analizar la serie para escoger los regresores (ver apartado 1.1). Se usó el enfoque basada en estimación no paramétrica del ruido como en [Lendasse et al., 2006] para la selección de un grupo de regresores, siendo el modelo resultante el siguiente:

$$y(t) = F(y(t-1), y(t-2), y(t-7), y(t-364)) \tag{5.6}$$

Los resultados con este enfoque se muestran en la Tabla 5.3. Como puede observarse, los modelos con *kernels* específicos consiguen menor error de test que los modelos para predicción recursiva. El tiempo de optimización es de unos 45 minutos, lo cual es menos que en el caso de modelos LSSVM con *kernels* específicos. Esto es debido a que para aplicar el Gradiente Conjugado se evalúa la derivada del error de validación cruzada frente a cada parámetro del *kernel*, y los *kernels* específicos tienen entre 6 y 7 parámetros frente al único parámetro del *kernel* Gaussiano.

Tabla 5.1: Resultados de KWKNN con *kernels* específicos al problema usando optimización del K para el Consumo Eléctrico en California

LOO					
kernel	opt. time	rmse-cv	K	rmse entrenamiento	rmse test
1	8.12e+02	2.40e+04	7	3.00e+02	3.78e+04
2	1.04e+03	2.40e+04	7	3.26e+02	3.77e+04
10-CV					
kernel	opt. time	rmse-cv	K	rmse entrenamiento	rmse test
1	8.27e+02	4.03e+04	20	2.94e+04	3.44e+04
2	1.05e+03	4.01e+04	20	2.98e+04	3.45e+04

Tabla 5.2: Resultados de LSSVM con *kernels* específicos al problema para el Consumo Eléctrico en California

LOO				
kernel	opt. time	rmse-cv	rmse entrenamiento	rmse test
1	1.61e+04	1.36e+04	6.66e+03	3.35e+04
2	1.84e+04	1.35e+04	5.83e+03	4.04e+04
10-CV				
kernel	opt. time	rmse-cv	rmse entrenamiento	rmse test
1	1.32e+04	1.86e+04	1.78e+04	3.42e+04
2	2.84e+03	9.93e+03	9.74e+03	3.49e+04

Tabla 5.3: Resultados de LSSVM usando predicción recursiva para el Consumo Eléctrico en California

l-fold	opt. time	rmse-cv	rmse tr	rmse test
LOO	2.52e+03	2.62e+04	2.03e+04	4.80e+04
10-CV	2.68e+03	2.79e+04	2.17e+04	5.36e+04

En las Figuras de la 5.2 a la 5.7 se dibujan las predicciones a un año vista con los mejores modelos KWKNN, LSSVM con *kernel* específicos y LSSVM para predicción recursiva. El mejor modelo en cada caso se escoge de acuerdo al valor final de validación cruzada en la optimización. En el caso de los modelos con *kernels* específicos, se trata de elegir entre uno de los dos *kernels* usados; en el caso LSSVM para predicción recursiva no hay selección. Se puede apreciar que en las predicciones obtenidas por los modelos con *kernels* específicos hay una influencia mayor de los periodos de la serie mientras que la predicción dada por el modelo recursivo tiende más a la media de la serie marcando menos el periodo anual. Comparando la salida de test de los modelos con *kernels* específicos, se puede ver que LSSVM produce una función mas suave, algo esperable de un modelo que incide tanto en el concepto de regularización [Suykens et al., 2002].

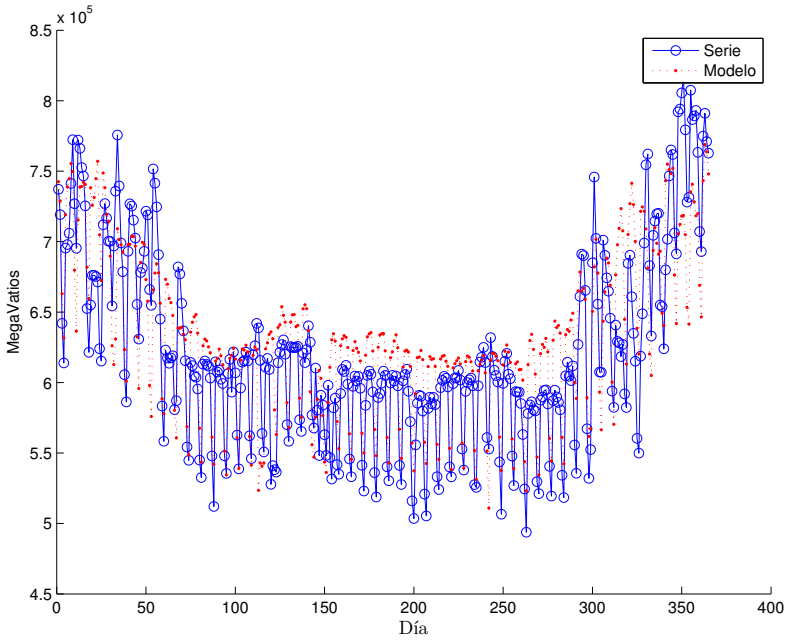


Fig. 5.2: Predicción del último año de los datos de Consumo Eléctrico en California, mejor modelo KWKNN usando un *kernel* específico-al-problema de acuerdo al error LOO

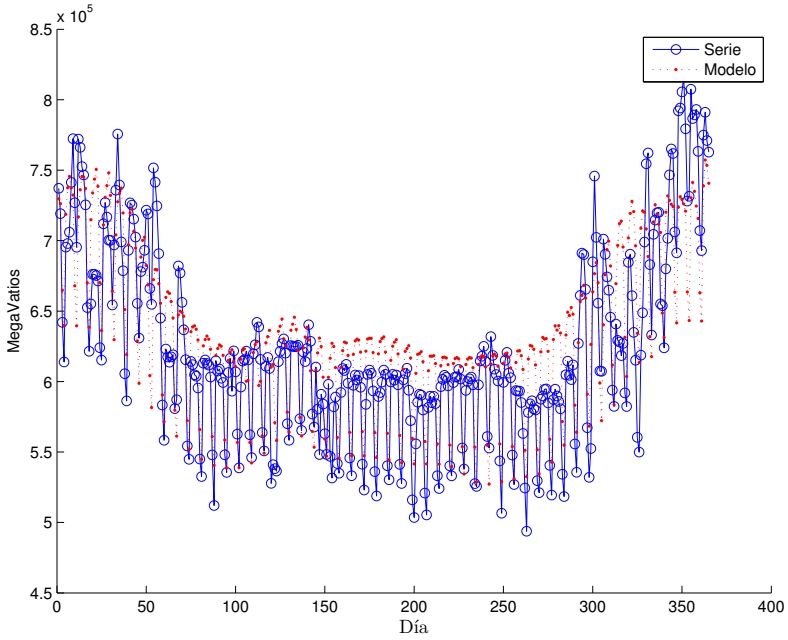


Fig. 5.3: Predicción del último año de los datos de Consumo Eléctrico en California, mejor modelo KWKNN usando un *kernel* específico-al-problema de acuerdo al error 10-CV

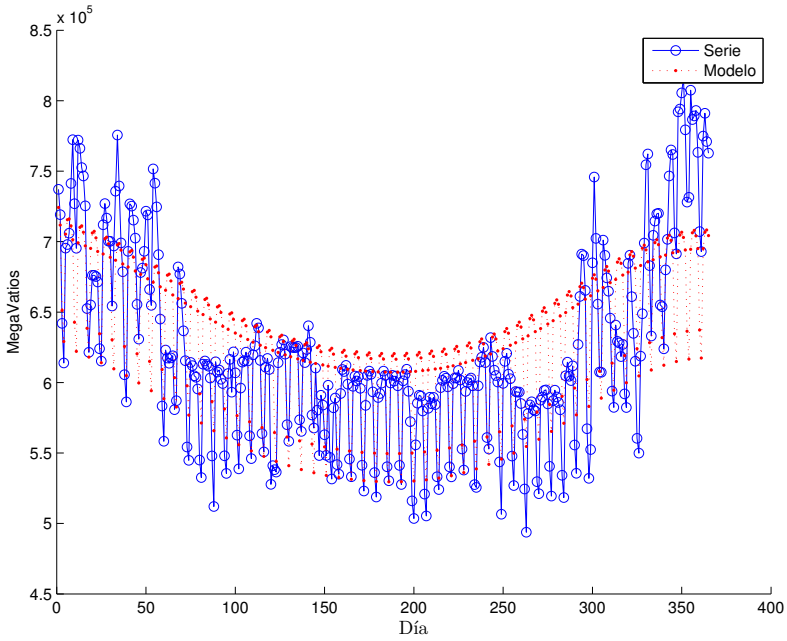


Fig. 5.4: Predicción del último año de los datos de Consumo Eléctrico en California, mejor modelo LSSVM usando un *kernel* específico-al-problema de acuerdo al error LOO

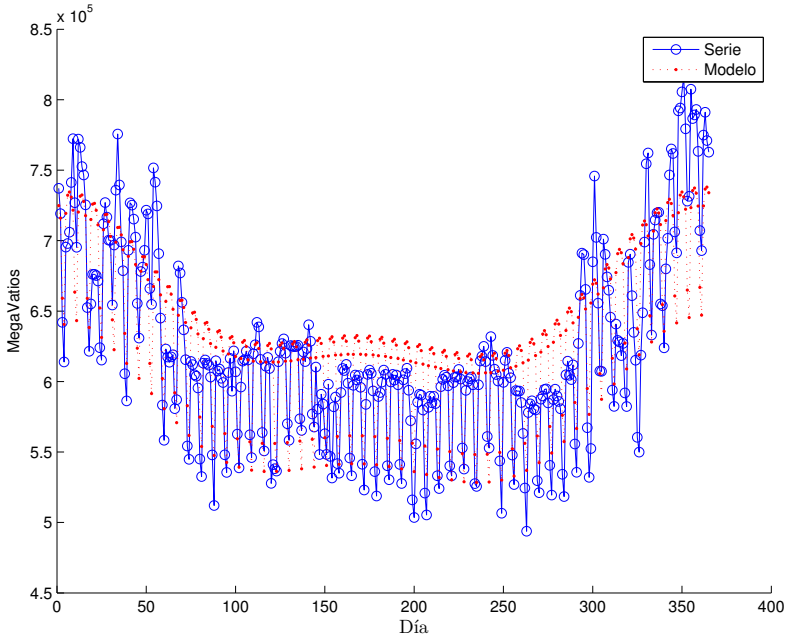


Fig. 5.5: Predicción del último año de los datos de Consumo Eléctrico en California, mejor modelo LSSVM usando un *kernel* específico-al-problema de acuerdo al error 10-CV

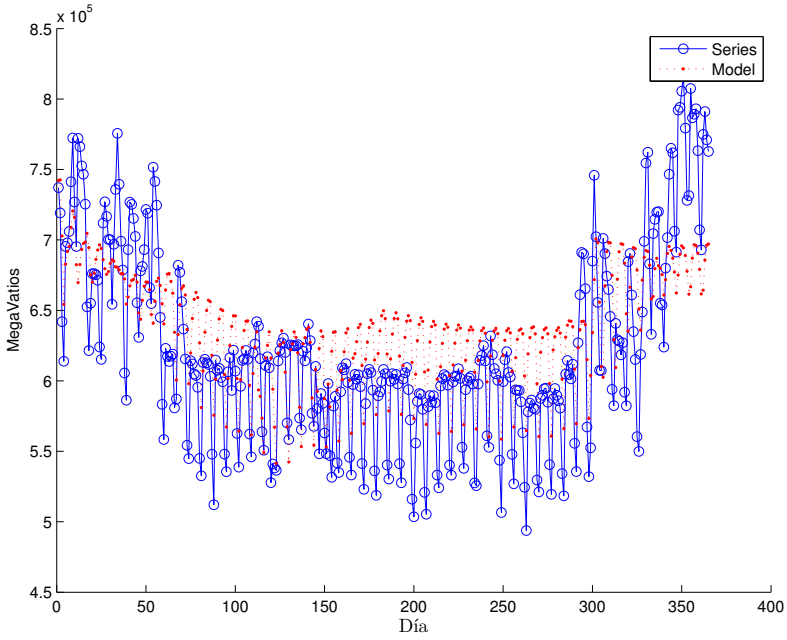


Fig. 5.6: Predicción del último año de los datos de Consumo Eléctrico en California, predicción recursiva con modelo LSSVM optimizado con error LOO

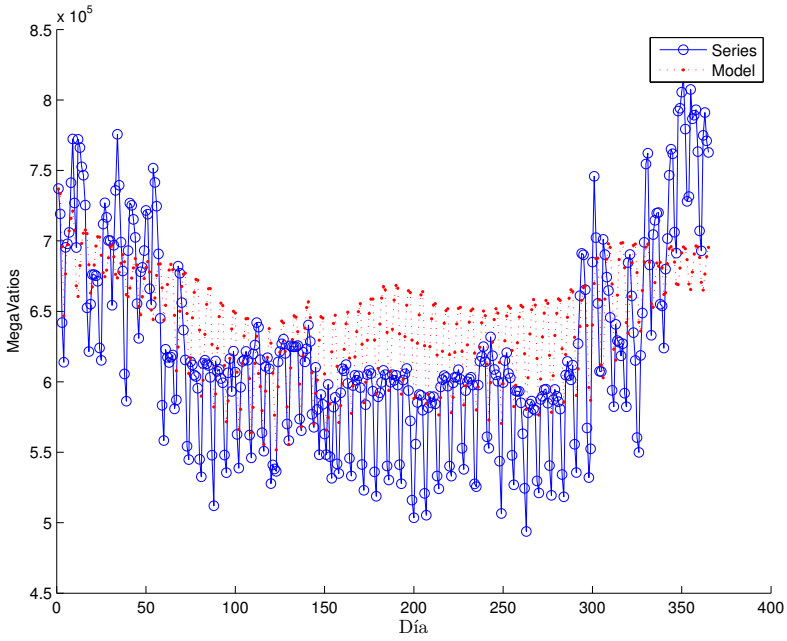


Fig. 5.7: Predicción del último año de los datos de Consumo Eléctrico en California, predicción recursiva con modelo LSSVM optimizado con error 10-CV

5.2.2.2. Serie temporal del río Snake

La serie temporal del flujo mensual del río Snake[3] es una serie que ha sido también utilizada en trabajos para recalcar la importancia de las periodicidades en los datos para la predicción de series temporales a largo plazo [Herrera et al., 2007b][Herrera et al., 2007a]. Presenta una gran cantidad de variaciones estocásticas y un patrón periódico anual (ver Figura 5.8). Los datos tienen un total de 1092 valores (1092 meses = 91 años), y, al igual que en los trabajos mencionados, la primera mitad de los mismos se usaron para entrenamiento y el resto para test (546 valores en cada caso $\simeq$ 45 años). Usando los mismos pasos aplicados en el caso anterior se obtienen los siguientes *kernels* específicos:

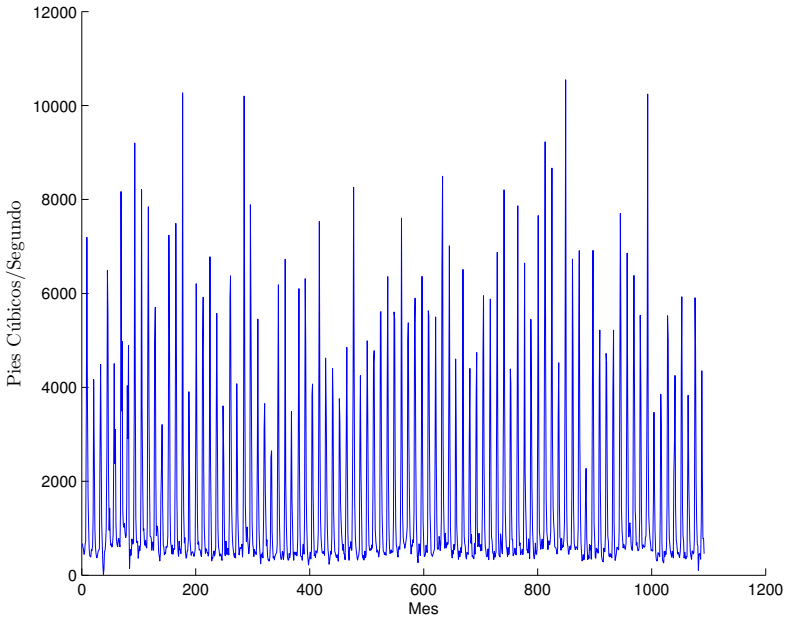


Fig. 5.8: Serie temporal del flujo mensual del río Snake

$$
\begin{aligned}
k_1(t_1,t_2;\{s_1,s_2,\sigma_g,\sigma_{12}\}) &= s_1 \cdot k_{gauss}(t_1,t_2;\sigma_g) \\
&\quad + s_2 \cdot k_{12-periodic}(t_1,t_2;\sigma_{12}) \\
k_2(t_1,t_2;\{s_1,s_2,\alpha,\beta,\sigma_{12}\}) &= s_1 \cdot k_{ratquad}(t_1,t_2;\alpha,\beta) \\
&\quad + s_2 \cdot k_{12-periodic}(t_1,t_2;\sigma_{12})
\end{aligned}
$$

donde t_1 y t_2 son los índices de tiempo de entrada, $\{s_1, s_2, \sigma_g, \sigma_1 2\}$ son los parámetros del *kernel* k_1 y $\{s_1, s_2, \alpha, \beta, \sigma_1 2\}$ son los parámetros del *kernel* k_2.

Los regresores para el modelo para predicción recursiva fueron los utilizados en el trabajo [Herrera et al., 2007b]:

[3] `http://www-personal.buseco.monash.edu.au/hyndman/TSDL/hydrology.html`

$$y(t) = F(y(t-3), y(t-6), y(t-9), y(t-12)) \tag{5.7}$$

Las optimizaciones de los modelos generados (KWKNN, LSSVM con *kernel* específicos y LSSVM para predicción recursiva) se hicieron con los mismos parámetros que en el ejemplo anterior. Los resultados se pueden ver en las Tablas 5.4, 5.5 y 5.6. A diferencia del caso anterior, no se observa una clara ventaja de los modelos con *kernel* específicos optimizados con 10-CV error frente a los optimizados con LOO. Los tiempos son menores que en el caso anterior, al tener menos datos la serie: algo menos de 2 minutos para KWKNN, unos 15 para LSSVM con *kernels* específicos y sobre 6 minutos para LSSVM para predicción recursiva. Los resultados son coherentes con los obtenidos en el ejemplo anterior, siendo KWKNN con *kernels* específicos el que obtiene mejores errores de test. En [Herrera et al., 2007b] y [Herrera et al., 2007a] se entrenaron modelos para estos mismos datos de idéntica manera. En [Herrera et al., 2007a] se entrenaron un modelo LSSVM y uno *Neurofuzzy*, mientras que en [Herrera et al., 2007b] un modelo LSSVM al que se incorporó información sobre periodos en los datos mediante la introducción de una variable adicional a la entrada. Se aplicó un esquema de predicción recursiva en ambos casos, y adicionalmente se entrenó los modelos con un método presentado por el autor para mejorar dicho esquema de predicción a largo plazo. Los mejores errores de predicción (RMSE) obtenidos en cada caso fueron respectivamente de 1054 (modelo LSSVM), 1041 (modelo TaSe) y 810 (modelo LSSVM con predicción a largo plazo mejorada). Estos valores han sido mejorados por los modelos con *kernels* específicos presentados en el presente trabajo.

Tabla 5.4: Resultados de KWKNN con *kernels* específicos al problema usando optimización del K para el flujo mensual del río Snake

LOO					
kernel	opt. time	rmse-cv	K	rmse entrenamiento	rmse test
1	7.83e+01	8.51e+02	20	7.61e+02	7.84e+02
2	1.08e+02	9.43e+02	20	3.69e+01	7.73e+02
10-CV					
kernel	opt. time	rmse-cv	K	rmse entrenamiento	rmse test
1	8.21e+01	8.11e+02	20	7.59e+02	7.72e+02
2	1.08e+02	8.15e+02	20	2.40e+02	7.77e+02

5.2.2.3. Serie temporal nº 3 de la competición del ESTSP 2008

Los datos utilizados son los de la más compleja de las tres series propuestas en la competición del ESTSP 2008, que se puede descargar de la página web `http://www.estsp.org/`. En este caso, no hay más información aparte de los datos de las series en sí. Para la serie considerada, 31614 valores fueron dados con

Tabla 5.5: Resultados de LSSVM con *kernels* específicos al problema para el flujo mensual del río Snake

LOO				
kernel	opt. time	rmse-cv	rmse entrenamiento	rmse test
1	7.62e+02	8.02e+02	5.19e+02	7.89e+02
2	9.68e+02	8.01e+02	5.65e+01	7.90e+02
10-CV				
kernel	opt. time	rmse-cv	rmse entrenamiento	rmse test
1	8.45e+02	8.54e+02	6.92e+02	7.88e+02
2	9.81e+02	8.44e+02	5.95e+02	7.88e+02

Tabla 5.6: Resultados de LSSVM usando predicción recursiva para el flujo mensual del río Snake

l-fold	opt. time	rmse-cv	rmse tr	rmse test
LOO	3.25e+02	1.02e+03	7.57e+02	9.21e+02
10-CV	3.49e+02	1.01e+03	7.72e+02	1.10e+03

la intención de predecir los siguientes 200, ver Figura 5.9. Los datos fueron analizados como en los casos anteriores, mostrando un periodo principal de longitud 8736 y una gran cantidad de periodos menores interdependientes (un interesante análisis puede hallarse en [Kourentzes and Crone, 2008]). Se decidió usar los periodos de longitud 8736 y 7, de entre los numerosísimos periodos determinados mediante inspección visual de la serie [Rubio et al., 2008].

Siguiendo la metodología del trabajo se obtuvieron los siguientes *kernels*:

$$\begin{aligned} k_1(t_1,t_2;\{s_1,s_2,s_3,\sigma_g,\sigma_7,\sigma_{8736}\}) &= s_1\cdot k_{gauss}(t_1,t_2;\sigma_g) \\ &\quad +s_2\cdot k_{7-periodic}(t_1,t_2;\sigma_7) \\ &\quad +s_3\cdot k_{8736-periodic}(t_1,t_2;\sigma_{8736}) \end{aligned}$$

$$\begin{aligned} k_2(t_1,t_2;\{s_1,s_2,s_3,\alpha,\beta,\sigma_7,\sigma_{8736}\}) &= s_1\cdot k_{ratquad}(t_1,t_2;\alpha,\beta) \\ &\quad +s_2\cdot k_{7-periodic}(t_1,t_2;\sigma_7) \\ &\quad +s_3\cdot k_{8736-periodic}(t_1,t_2;\sigma_{8736}) \end{aligned}$$

donde t_1 y t_2 son los índices de tiempo de entrada, $\{s_1,s_2,s_3,\sigma_g,\sigma_7,\sigma_{8736}\}$ son los parámetros del *kernel* k_1 y $\{s_1,s_2,s_3,\alpha,\beta,\sigma_7,\sigma_{8736}\}$ son los parámetros del *kernel* k_2.

La creación de modelos KWKNN o LSSVM en este caso no es posible debido al gran número de muestras en los datos. La Figura 5.10 muestra para tamaños de entre 500 y 2000 muestras, los tiempos de optimización de modelos KWKNN y LSSVM con un *kernel* específico al problema usando VNS, limitando a 500 las evaluaciones del error de validación cruzada, tanto para error 10-CV como LOO

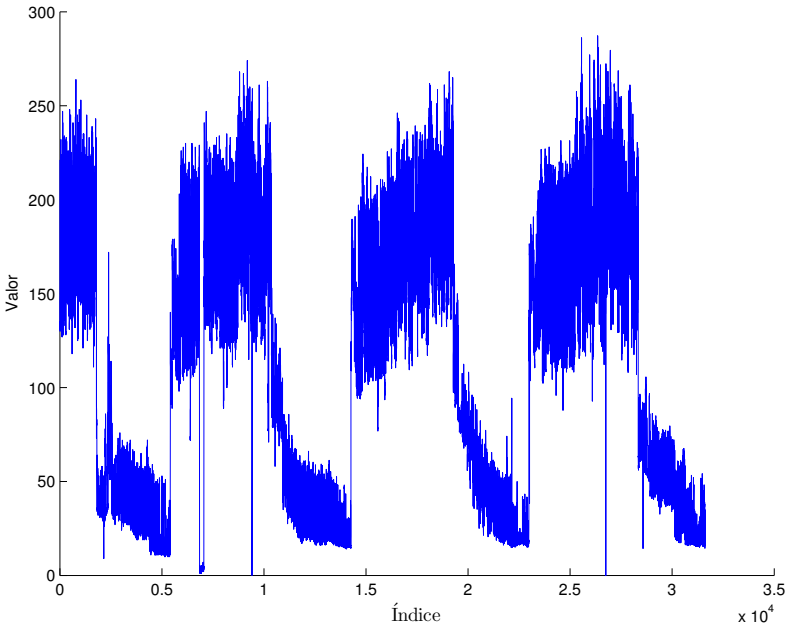


Fig. 5.9: Serie temporal 3 ESTSP.

(el parámetro K de KWKNN se optimiza en el rango $[2,20]$). De esta figura se infiere claramente la diferencia en complejidad computacional entre KWKNN y LSSVM. Para conjuntos con miles de muestras, usar LSSVM se vuelve impracticable. PKWKNN sin embargo puede ser usado para este problema. Aparte de los costes computacionales anteriormente nombrados, es importante recordar que también los requisitos de memoria hacen obligatorio aplicar PKWKNN en este caso.

La optimización de los modelos KWKNN con VNS se realiza con los mismos valores de parámetros que en los ejemplos anteriores, y el número de procesadores usado fue de 32. En este caso la optimización se hace con respecto al error 32-CV y LOO (ver Sección 4.7 acerca de la evaluación del error de validación cruzada en datos distribuidos). Los resultados mostrados en la Tabla 5.7 reflejan tiempos de optimización entre 7,5 y 12 horas. Debe recordarse que el incremento de velocidad teórico de PKWKNN es lineal, por lo que de tener un computador con suficiente memoria como para ejecutar KWKNN, los tiempos habrían sido 32 veces mayores. Los errores de entrenamiento y test muestran que los modelos optimizados respecto al error 32-CV presentan un menor sobreajuste que los optimizados con LOO. Las Figuras 5.11 y 5.12 representan las predicciones obtenidas con los mejores modelos (según el error de validación cruzada final de la optimización).

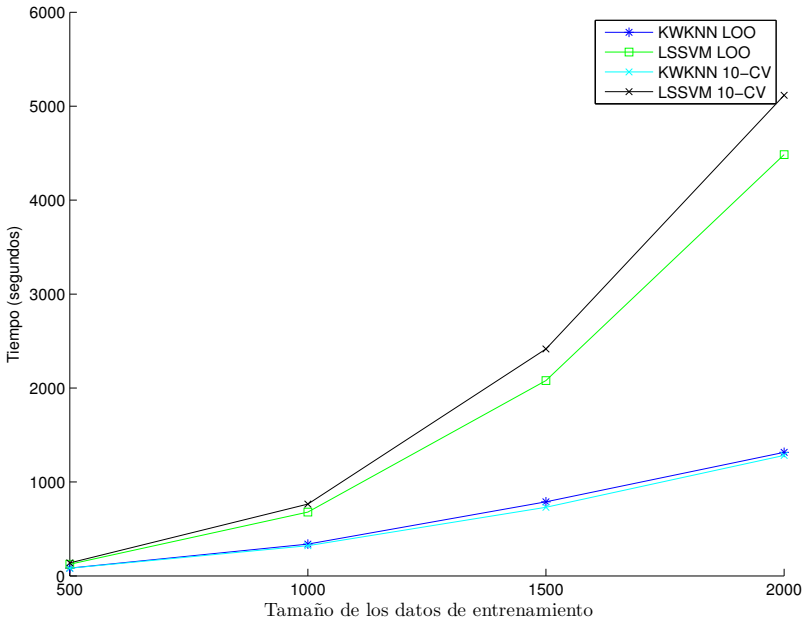


Fig. 5.10: Tiempos de optimización para modelos KWKNN y LSSVM con un *kernel* específico-al-problema (en segundos) frente al número de muestras.

Tabla 5.7: Resultados de KWKNN con *kernels* específicos al problema usando optimización del K para la serie ESTSP 3

LOO					
kernel	opt. time	rmse-cv	K	rmse entrenamiento	rmse test
1	3.24e+04	6.98e+00	2	3.01e-04	1.27e+01
2	4.24e+04	6.98e+00	2	1.66e-05	1.29e+01
10-CV					
kernel	opt. time	rmse-cv	K	rmse entrenamiento	rmse test
1	3.21e+04	3.06e+01	5	1.33e+01	1.13e+01
2	4.25e+04	3.00e+01	2	2.78e-02	9.58e+00

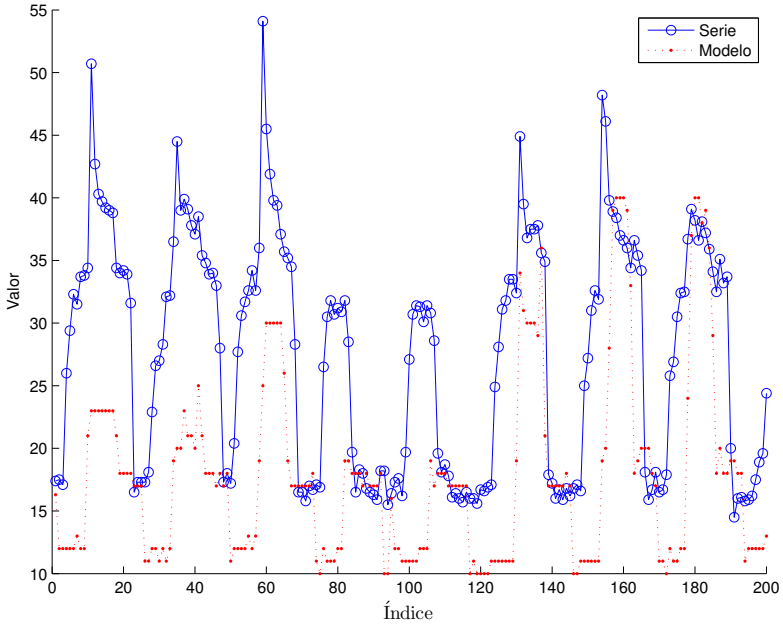


Fig. 5.11: Predicción de la serie ESTSP 3, mejor modelo PKWKNN con error LOO

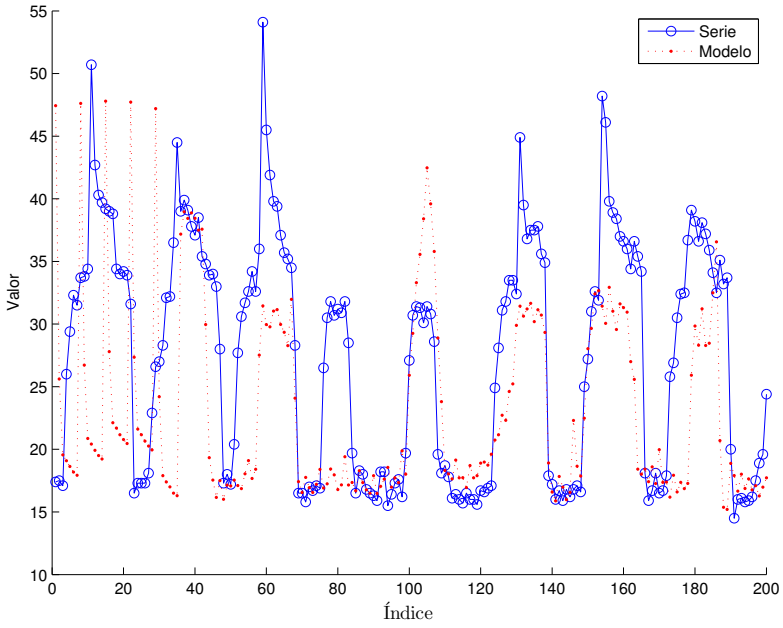


Fig. 5.12: Predicción de la serie ESTSP 3, mejor modelo PKWKNN con error 32-CV

5.3. Programación Genética y Kernels específicos

En este apartado se aborda el problema de creación de *kernels* específicos a un problema dado, pero intentando hallar una metodología que permita realizar esta tarea sin conocimiento previo ni intervención de un experto. Este objetivo, aunque se restrinja al caso de series temporales, es ciertamente muy díficil. Como quedó demostrado en el apartado anterior es el conocimiento sobre una serie en concreto embebido en la expresión de un *kernel* lo que permite obtener buenos resultados.

5.3.1. Precedentes

En [Koza, 1992], Koza propuso una clase de algoritmos evolutivos llamada Programación Genética (GP), que puede ser usada para evolucionar programas de computadora para la resolución de un amplio rango de problemas. Los programas que se desarrollan están codificados en forma de árboles de análisis sintáctico (*parse trees* en inglés). El uso de *parse trees* tiene las ventajas de prevenir errores sintácticos, que puedan llevar a individuos no válidos, y también que la jerarquía en el árbol previene de problemas con la precedencia de operadores.

La tarea de obtener un *kernel* es la de obtener una función simétrica que cumpla el teorema de Mercer (ver apartado 2.3.1), un requisito básico de los métodos *kernel* clásicos. Aunque los precedentes en la literatura de aplicación de GP para creación de *kernels* específicos se limitan a problemas de clasificación, de los distintos enfoques se extrajeron algunas conclusiones útiles para nuestra particular tarea, a partir de las siguientes referencias:

- En [Howley and Madden, 2005] los errores de entrenamiento de modelos SVM son optimizados, evitando el sobreajuste mediante un estimador del margen del propio modelo (es decir, una aproximación a su capacidad de generalización). La condición del teorema de Mercer se ignora debido a su complejidad computacional, algo que no es demasiado grave ya que existen en la literatura *kernels* que no lo cumplen que han dado buenos resultados, cómo por ejemplo el *kernel* sigmoidal de la Ecuación (5.8).

$$k_{sigmoid}(x_i, x_j; \{\kappa, \rho\}) = \tanh(\kappa\langle x_i, x_j\rangle + \rho) \tag{5.8}$$

- En [Gagné et al., 2006] el modelo optimizado es llamado regla K-NN de clasificación, y la función de adaptación está inspirada en el concepto de margen de un clasificador. El teorema de Mercer también se ignora porque el modelo usado es especialmente tolerante a *kernels* que no lo cumplen.

- En [Methasate and Theeramunkong, 2007] y [Sullivan and Luke, 2007] se optimiza el error de validación cruzada de modelos SVM. Los nodos operadores

y terminales usados se escogieron para que siempre se generaran *kernels* que satisfagan el teorema Mercer.

- En [Diosan et al., 2007] el error de un conjunto de validación es usado para modelos SVM. Los *kernels* creados pueden ser incluso no simétricos, pero las condiciones son evaluadas después de la creación de cada uno de ellos.

La estrategia usada en este trabajo es diferente, y se resume en la Figura 5.13. Aparte de aplicar por primera vez GP en la creación de *kernels* a medida para problemas de regresión, otra novedad es el uso de una adaptación específicamente creada del estimador no paramétrico del ruido test Delta [Lendasse et al., 2006] para evaluación de los *kernels* (ver la Sección 5.3.2). De esta manera la calidad de cada *kernel* se evalúa con independencia del modelo en particular en el cual puedan ser usados (SVM para regresión, LSSVM, Procesos Gaussianos, etc). Los *kernels* creados son probados usando KWKNN (ver apartado 4.2), que es un modelo bastante robusto con respecto a los *kernels* que no cumplen el teorema de Mercer.

Por lo tanto, el trabajo presentado en este apartado ilustra una aplicación de un método *kernel* para regresión, especializando las funciones *kernel* al problema concreto mediante Programación Genética.

5.3.2. Adaptación del test Delta en espacio de características

La estimación del ruido existente en un conjunto de pares de entrada/salida se ha comprobado ser especialmente útil para problemas de regresión. El ruido en la salida establece un limite a la precisión de un modelo entrenado para dichos datos. En otras palabras, cuando un modelo entrenado para unos datos obtiene una precisión superior al del ruido presente en dichos datos, se puede concluir que el modelo está sobreajustado a los datos de entrenamiento y debe descartarse. Esta estimación del ruido también puede ser usada para realizar selección de entradas relevantes para el modelo, como se ve en [Lendasse et al., 2006]. En nuestro caso se usará para evaluar la *adaptación* en sí de un *kernel* para los datos de un problema particular.

El test Delta se basa en la definición del vecino más cercano a un punto. Para definir los vecinos más cercanos en espacio de características, se debe de definir la distancia en el espacio de características en base al *kernel*. La única información disponible sobre dicho espacio es la función *kernel* k que formalmente es el producto escalar aplicado a las entradas tras aplicar una proyección Φ de los puntos del *espacio de entrada* al *espacio de características* [Scholkopf and Smola, 2001] $k(x,x') = \langle \Phi(x), \Phi(x') \rangle$. Se cumple la siguiente relación entre la función de distancia y la norma cuadrática en el espacio de características: $D(x,x')^2 = \|\Phi(x) - \Phi(x')\|^2$. La norma cuadrática puede escribirse entonces como:

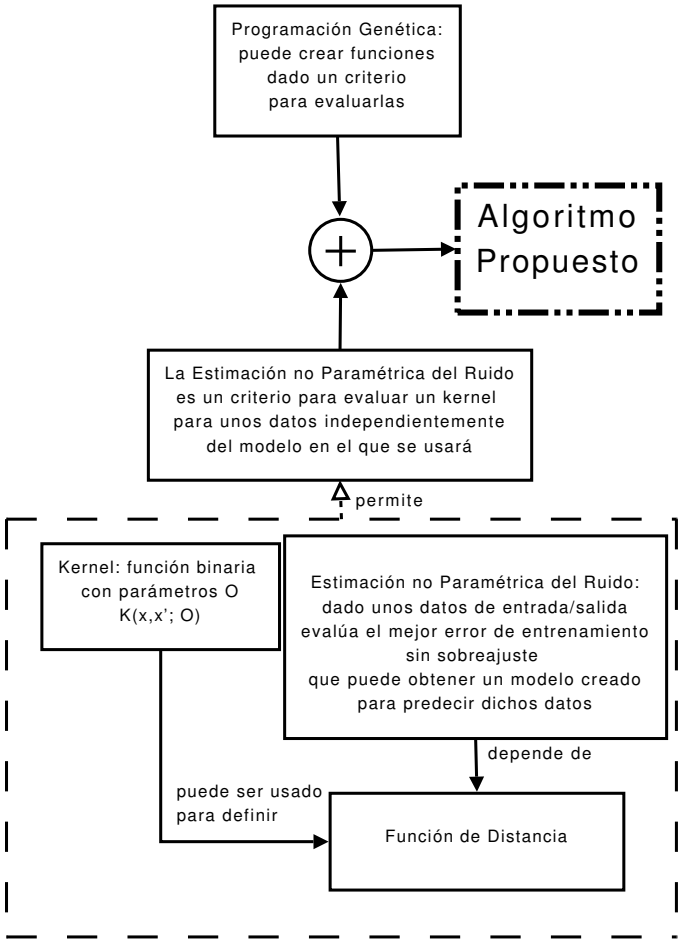


Fig. 5.13: Bases del diseño del algoritmo GP presentado

$$\begin{aligned} \|\Phi(x)-\Phi(x')\|^2 &= \langle\Phi(x),\Phi(x)\rangle + \langle\Phi(x'),\Phi(x')\rangle - 2\langle\Phi(x),\Phi(x')\rangle \\ &= k(x,x)+k(x',x')-2k(x,x') \end{aligned} \tag{5.9}$$

Por lo tanto, la norma cuadrática de dos puntos en el espacio de características puede ser escrita en términos de valores de una función de *kernel*, y no hay necesidad de conocer explícitamente la función Φ. Consecuentemente, la adaptación del test Delta para trabajar en espacio de características es inmediata al calcular el vecino más cercano en función de la distancia de *kernel* en vez de usando la distancia Euclídea.

5.3.3. Algoritmo de Programación Genética

El primer parámetro a definir de un algoritmo de Programación Genética (GP) es la representación de las soluciones. En el caso de los *kernels*, la clásica representación de árboles sintácticos (o *parse trees*) es adecuada, centrando el problema en escoger los nodos operadores y terminales que se usarán para construir los árboles (es decir, las expresiones que codifican los *kernels*). Un *kernel* $k(x_i,x_j)$ según la teoría debe ser simétrico, $k(x_i,x_j)=k(x_j,x_i),\forall x_i,x_j$, y cumplir el teorema de Mercer. Si vemos los precedentes en la literatura:

- En [Methasate and Theeramunkong, 2007] [Sullivan and Luke, 2007], los nodos fueron escogidos para que se generaran *kernels* Mercer, usando como nodos terminales *kernels* Mercer conocidos y operadores que garantizaran que los resultados siguiesen siendo *kernels* Mercer (reglas de clausura de *kernels*).
- En [Howley and Madden, 2005] la función obtenida $k(x,z)$ se usa para crear un *kernel* $K(x,z)=k(x,z)k(z,x)$, asegurando la simetricidad del resultado.
- En [Gagné et al., 2006] la simetría se asegura usando nodos terminales que son, ellos mismos, funciones simétricas de las entradas.
- En [Diosan et al., 2007] las condiciones de simetría y del teorema de Mercer se comprueban después de la creación de cada *kernel*.

En nuestro caso, como en [Gagné et al., 2006], los nodos se escogieron para asegurar la creación de funciones que fueran simétricas con respecto de las entradas pero sin preocuparnos por cumplir con el teorema de Mercer. Los nodos pueden consultarse en la Tabla 5.8. La aridad de los operadores (es decir, el número de hijos/operandos) es de 1 o de 2 para todos los nodos con excepción de los nodos 2 y 4 (los operadores suma y producto), cuyo número de hijos puede variar y llegar hasta un máximo fijado por el parámetro *max. arity* (ver Tabla 5.9). En la Figura 5.14 está un árbol con nodos del conjunto que codifica el *kernel* Gaussiano.

Tabla 5.8: Operadores usados en el algoritmo GP. p_i denota un parámetro de un nodo, $input_i$ es el resultado de la evaluación de i-ésimo hijo de un nodo, x_1, x_2 son las entradas de X.

Índice	Nodo	Índice	Nodo
1	p_1	10	$log(input_1)$
2	$\sum_j^n input_j$	11	$exp(input_1)$
3	$input_1 - input_2$	12	$tanh(input_1)$
4	$\prod_j^n input_j$	13	$sin(input_1)$
5	$(input_1)^{input_2}$	14	$p_1(x_1 + x_2)$
6	$par_1 \cdot input_1$	15	$p_1 \|\|x_1 - x_2\|\|^2$
7	$(input_1)^{par_1}$	16	$p_1 x_1 \cdot x_2$
8	$-input_1$	17	$p_1 sin(\pi \cdot p_1 \cdot (x_1 - x_2))^2$
9	$(input_1)^2$		

Un gran problema asociado a la representación de expresiones en árboles, es que el espacio de búsqueda es incluso superior al caso de los algoritmos genéticos clásicos. Más aún, una variación en la estructura de un árbol por un operador lo convierte en uno que codifica una expresión muy diferente, lo que supone grandes saltos en el espacio de búsqueda. Con el fin de limitar el espacio de búsqueda, cada vez que se crea un árbol se le aplica un procedimiento de **simplificación** y otro de **normalización**. El procedimiento de simplificación convierte un árbol en otro que codifica un *kernel* equivalente pero con menos nodos. El tipo de transformación depende del operador raíz del sub-árbol al que se aplica (ver Tabla 5.8):

1. Eliminación de sub-árboles constantes, es decir, aquellos en los que todos los nodos hoja son constantes. Se sustituyen por un nodo terminal constante (Nodo índice 1 de la Tabla 5.8). Este operador no se aplica a sub-árboles que contengan un nodo que sea un operador de negación (Nodo índice 8 de la Tabla 5.8). El motivo es que al optimizarse los parámetros en escala logarítmica en la implementación, es uno de los únicos modos de obtener valores negativos en las fórmulas.

2. Si los sub-árboles de un nodo suma o producto (nodos de índices 2 y 4 de la Tabla 5.8) son idénticos, el nodo se sustituye por una operación de escalado adecuada (Nodos de índices 6 y 7 de la Tabla 5.8) sobre uno de los sub-árboles.

3. Si hay varios nodos de operación de escalado en cadena (Nodos de índices 6 y 7 de la Tabla 5.8) se reducen a un único nodo.

4. Si se aplica a un nodo terminal un nodo operador de escala (Nodo de índice 6 de la Tabla 5.8), se elimina este último.

5. Sub-árboles sólo con operadores suma o producto y nodos terminales se simplifican a un sólo nodo suma o producto aplicado dichos terminales.

El procedimiento de normalización ordena de manera recursiva los hijos de los nodos que sean operadores conmutativos de un árbol mediante el orden definido en el Algoritmo 5.2. Un ejemplo de simplificación y normalización para un árbol creado de forma aleatoria se muestra en las Figuras 5.15, 5.16 y 5.17.

T1, T2: 2 árboles de índices
si $profundidad(T1) < profundidad(T2)$ **entonces**
  $T1 < T2$
si no si $profundidad(T1) > profundidad(T2)$ **entonces**
  $T1 > T2$
si no si $numero\_nodos(T1) < numero\_nodos(T2)$ **entonces**
  $T1 < T2$
si no si $numero\_nodos(T1) > numero\_nodos(T2)$ **entonces**
  $T1 > T2$
si no si $indice\_raiz(T1) < indice\_raiz(T2)$ **entonces**
  $T1 < T2$
si no si $indice\_raiz(T1) > indice\_raiz(T2)$ **entonces**
  $T1 > T2$
si no
  $T1 == T2$
fin si

Alg. 5.2: Orden de normalización

Un algoritmo de GP, dejando aparte la representación de los individuos de la población y su evaluación, se define por los mismos elementos que un algoritmo genético común: inicialización de la población, selección de individudos para reproducción y sustitución, operadores genéticos de cruce y mutación, función de adaptación, etc. A continuación se exponen la configuración particular del algoritmo presentado:

Initialización: la población se inicializa usando el procedimiento *grow*, como en [Howley and Madden, 2005] [Diosan et al., 2007], que asegura una variedad de profundidad de árboles en la población inicial.

Selección: los padres son escogidos con la clásica selección por ruleta [Goldberg, 1989b].

Cruce: el clásico operador de GP de cruce por intercambio de subárboles (*subtree exchange cross-over*) es usado [Koza, 1992]. Este operador escoge al azar un subárbol de cada padre y los intercambia.

Mutación: poda de subárbol (*subtree prune mutation*), se sustituye un subárbol escogido aleatoriamente por un nodo terminal.

Sustitución: los peores individuos de la población son los reemplazados en cada generación.

Función de adaptación: cada individuo es un *kernel* cuya función de adaptación está basada en el test Delta. De hecho, el valor de adaptación de un individuo se calcula como la raíz cuadrada del test Delta en el espacio de características aplicado a los datos de entrada dividido por la varianza de la salida. Esta normalización hace que el valor sea independiente del problema concreto, y permite realizar comparativas; de hecho está relacionada con el *Error Cuadrático Medio Normalizado* [Herrera et al., 2005] ($NRMSE = \sqrt{MSE/\sigma_y^2}$, donde MSE es el error cuadrático medio y σ_y^2 la varianza de la función que se modela). Esta función de adaptación no es determinista, al depender de los valores que se asignen a los parámetros del *kernel* que codifica un individuo. Cada *kernel* tiene sus propios parámetros, estos son optimizados con respecto al test Delta usando, al igual que en el apartado 4.4, Búsqueda de Vecindario Variable (VNS) [Mladenović and Hansen, 1997], y se justifica el no usar otros métodos como el Gradiente Conjugado al hecho de que el test Delta no es derivable.

Una preocupación importante en GP es el crecimiento de los árboles en una población sin mejora del valor de adaptación de los individuos de la misma. El operador de cruce hace que crezcan los árboles, por lo que se escogió una mutación que compensara dicho efecto reduciendo el tamaño de los árboles. Además, se definieron los parámetros de máxima profundidad, número de nodos y aridad de nodos de los árboles válidos en una ejecución, siendo evaluados y usados en reproducción sólo aquellos árboles que cumplan con todas las restricciones de tamaño.

El software fue desarrollado bajo Matlab en un ordenador tipo PC y se utilizó compilación cruzada para poder ejecutarlo en plataforma cluster. Se hizo uso de la librería MPI, utilizando la toolbox MPImex [Guillen et al., 2008a] para poder acceder a ella desde Matlab. La mayor parte de los costes computacionales del algoritmo se concentran en la evaluación del individuo, concretamente en la optimización de los parámetros del *kernel* con respecto al test Delta. Para reducir tiempos de ejecución se implementó un esquema de balanceo de carga: un proceso mantiene la población y ejecuta el bucle principal (es decir: las generaciones, creación de individuos, etc.), mientras el resto ejecutan un bucle esperando recibir individuos/*kernels* del proceso principal para aplicar la optimización VNS con el fin de optimizar los parámetros del *kernel*.

5.3.4. Experimentos

En esta sección se describe la configuración usada en los experimentos que se han realizado, así como los *kernels* que han sido obtenidos por nuestro algoritmo de programación genética. El fin de estos experimentos es confirmar si es posible obtener *kernels* específicos para usar en modelos para predicción de series temporales con la misma estructura que los creados para los experimentos del apartado 5.2. En los experimentos con *kernels* basado en conocimiento previo, el único factor que se tuvo en cuenta fue la periodicidades de las series, con estos experimentos se intenta también ver si los *kernels* creados con programación genética pueden embeber tanto éste como otro tipo de conocimiento, desconocido a priori.

5.3.4.1. Parámetros del algoritmo

Los valores de los parámetros del algoritmo GP usado en los experimentos son los mostrados en la Tabla 5.9. Estos fueron escogidos para evitar tiempos de ejecución demasiado grandes pero sin limitar en exceso la exploración del espacio de búsqueda (es decir, *kernels* que pudiesen ser generados). Si no se limitara el número de nodos en un árbol, en el peor de los casos se podrían obtener con esta los operadores y terminales que se utilizan árboles con *15625* nodos. Evidentemente, esto es excesivo si se desea analizar dichos árboles. Dado nuestro interés en ver las expresiones generadas, para obtener árboles legibles se limitaron a 10 el número máximo de nodos por árbol. Para fomentar la explotación del espacio, se substituye la mitad peor de la población (30 de 60) cada generación con hijos de los 2 mejores individuos.

Tabla 5.9: Valores de los parámetros del GP en los experimentos

Parámetro	Valor	Parámetro	Valor
población	60	prof. máx. de un árbol	4
generaciones	200	nodos máx. de un árbol	10
substituciones	30	aridad máx. de un nodo	5

5.3.4.2. Series temporales

Las series utilizadas fueron las mismas que en los experimentos del apartado 3.5.1 más el conjunto de datos del río Snake: las series caóticas *Hénon* [Hénon, 1976], *Logistic* [May, 1976] y *Mackey-Glass* [Mackey and Glass, 1977], las lineales *AR(4)* y *S.T.A.R.* [Berardi and Zhang, 2003], las reales *Sunspots*, *London*, *Electric* y *Snake* [Herrera et al., 2007b][Herrera et al., 2007a], y las sacadas de la competición de Santa Fe *Computer* y *Laser* [Weigend and Gershenfeld, 1994]. De todas las series utilizadas, se esperaba obtener *kernels* a medida para las series reales utilizadas, que presentan todas periodos. Para todos los casos se cogen 1000

valores de las series, la primera mitad para entrenamiento y la siguiente para test (menos en el caso de la serie *Sunspots*, que sólo tiene 306 datos).

5.3.4.3. Resultados

Se ejecutó 6 veces el algoritmo para cada serie con los parámetros anteriormente nombrados (y expuestos en la Tabla 5.9). Hay una clara mejora del valor de adaptación del mejor individuo así como del valor mediano de la adaptación de la población como se puede ver en la Tabla 5.10 o las gráficas de valores medios de adaptación por generación de las Figuras 5.18 a la 5.26. En la Tabla 5.11 están los errores de entrenamiento y test de los modelos KWKNN que usan el mejor de los *kernels* generados por el algoritmo de programación genética (en cada caso se optimizan con 500 evaluaciones del error de validación cruzada de orden 10, con el parámetro K de KWKNN en el rango $[2,20]$, como en los experimentos del apartado 5.2). Se proporcionan los valores del error independientes de escala usando el *Error Cuadrático Medio Normalizado* [Herrera et al., 2005] ($NRMSE = \sqrt{MSE/\sigma_y^2}$, donde MSE es el error cuadrático medio y σ_y^2 la varianza de la función que se modela). Un valor de NRMSE de *1* significa una aproximación igual de buena que la media. Como puede verse, al coger la población con el individuo con mejor valor de adaptación de los creados y entrenar modelos KWKNN con dichos *kernels* sólo se obtienen resultados mejores que la media para las series *London* y *Snake*. Los *kernels* correspondientes se muestran en la Tabla 5.12, y se observa que su principal característica es la componente que modela la periodicidad de dichas series. En las pruebas sin embargo no se lograron modelar los periodos presentes en las series *Sunspots* y *Electric*, como se esperaba en un principio. En el caso de *Snake* tenemos resultados del apartado 5.2 con modelos KWKNN con *kernels* específicos creados por un experto en la Tabla 5.4, con unos errores (RMSE) de entrenamiento y test de $3{,}69e+01$ y $7{,}73e+02$, en el peor de los casos, frente a los resultados con el mejor *kernel* creados por el algoritmo de programación genética de $8{,}10e+02$ y $9{,}36e+02$. En cuanto a *London*, se utilizó la metodología del apartado 5.2 para crear *kernels* específicos obteniendo resultados de entrenamiento y test de NRMSE de $5{,}70e-02$ y $3{,}49e-01$ frente a los de $2{,}79e-01$ y $5{,}06e-01$ con el mejor de los *kernels* obtenidos con el algoritmo de programación genética. Los tiempos de ejecución medios (junto sus desviaciones típicas) vienen reflejados en la Tabla 5.13 para cada serie usada en los experimentos, y demuestran el elevado coste computacional del algoritmo, el cual repercute en tiempos bastantes elevados aún a pesar de usar el test Delta modificado y la paralelización de la evaluación de los individuos.

Tabla 5.10: Media del valor de adaptación mediano y mejor de las poblaciones de 6 ejecuciones del GP para cada conjunto de datos

Serie	mediana de la adaptación		mejor valor adaptación	
	inicio	fin	inicio	fin
Henon	8.425e-01	7.260e-01	5.471e-01	4.689e-01
Logistic	9.689e-01	8.673e-01	6.687e-01	5.605e-01
Mackey-Glass	7.439e-01	1.097e-01	1.021e-01	7.571e-02
AR4	9.089e-01	4.904e-01	4.410e-01	2.962e-01
STAR	7.395e-01	3.675e-01	3.441e-01	2.959e-01
Sunspots	7.970e-01	4.340e-01	3.356e-01	2.407e-01
London	8.670e-01	2.514e-01	2.514e-01	1.917e-01
Electric	7.073e-01	4.000e-01	3.442e-01	3.308e-01
Snake	9.252e-01	5.207e-01	4.417e-01	3.361e-01
Computer	7.423e-01	2.216e-01	1.732e-01	1.657e-01
Laser	1.043e+00	4.967e-01	4.268e-01	3.599e-01

Tabla 5.11: Modelos entrenados con los *kernels* de la población con el mejor valor de adaptación de las ejecuciones

Datos	error entr.	error test
Henon	8.78e-01 $\pm$ 1.01e-01	1.05e+00 $\pm$ 1.66e-02
Logistic	1.01e+00 $\pm$ 1.32e-03	9.95e-01 $\pm$ 2.39e-04
Mackey-Glass	4.68e-01 $\pm$ 6.94e-02	1.01e+00 $\pm$ 3.45e-02
AR4	7.42e-01 $\pm$ 6.95e-02	1.22e+00 $\pm$ 1.02e-01
STAR	8.48e-01 $\pm$ 4.89e-02	1.07e+00 $\pm$ 7.04e-03
Sunspots	2.69e-01 $\pm$ 3.82e-03	1.43e+00 $\pm$ 4.85e-02
London	2.84e-01 $\pm$ 5.11e-03	5.33e-01 $\pm$ 3.51e-01
Electric	3.35e-01 $\pm$ 3.80e-02	9.87e-01 $\pm$ 4.98e-03
Computer	5.48e-01 $\pm$ 6.18e-02	1.36e+00 $\pm$ 2.97e-03
Laser	4.43e-01 $\pm$ 3.07e-01	1.22e+00 $\pm$ 1.15e-01
Snake	4.02e-01 $\pm$ 8.96e-02	7.64e-01 $\pm$ 4.09e-01

Datos	mejor error entr.	mejor error test
Henon	6.89e-01	1.05e+00
Logistic	1.01e+00	9.95e-01
Mackey-Glass	4.46e-01	1.00e+00
AR4	5.92e-01	1.14e+00
STAR	9.87e-01	1.09e+00
Sunspots	2.80e-01	1.29e+00
London	2.79e-01	5.06e-01
Electric	4.43e-01	9.73e-01
Computer	5.67e-01	1.36e+00
Laser	9.23e-02	1.34e+00
Snake	4.45e-01	5.15e-01

Tabla 5.12: *Kernels* a medida creados por el algoritmo de programación genética para London y Snake

Datos	mejor *kernel*
London	$K(u,v;\{p_1,p_2,p_3\}) = \exp(\tanh((p_1)+(p_2 sin(\pi p_3(u_1-v_1)))))$
Snake	$K(u,v;\{p_1,p_2,p_3,p_4\}) = -\left((p_1)^{\left((p_2 sin(\pi p_3(u_1-v_1)))^{(p_4)}\right)}\right)$

5.3.4.4. Conclusiones

La programación genética es capaz de generar *kernels* específicos a un problema dado a un alto coste computacional, pero sin embargo no hay garantía de obtener una expresión comprensible o siquiera que mejore resultados de *kernels* comunes y menos los creados por un experto. En nuestro caso se ha intentado reducir en alguna medida los costes computacionales evaluando la adaptación de los *kernels* mediante un estimador basado en el test Delta, aún así los resultados no han mejorado substancialmente los obtenidos con los *kernels* creados con la metodología expuesta en el apartado 5.2. Es más, la esperanza inicial de modelar características de las series más allá de los periodos no se ha cumplido. La conclusión es que este enfoque necesita más investigación para obtener mejores resultados y ser práctico.

Tabla 5.13: Tiempos de ejecución de los experimentos con GP

Serie	Tiempos de ejecución
Henon	1.9 horas $\pm$ 13 minutos
Logistic	2.2 horas $\pm$ 29 minutos
Mackey-Glass	2.9 horas $\pm$ 50 minutos
AR4	1.5 horas $\pm$ 27 minutos
STAR	1.8 horas $\pm$ 20 minutos
Sunspots	0.27 horas $\pm$ 4.3 minutos
London	1.8 horas $\pm$ 33 minutos
Electric	2.2 horas $\pm$ 56 minutos
Computer	1.7 horas $\pm$ 24 minutos
Laser	2.1 horas $\pm$ 15 minutos
Snake	2.1 horas $\pm$ 26 minutos

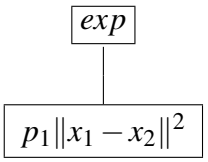


Fig. 5.14: Árbol codificando el *kernel* Gaussiano

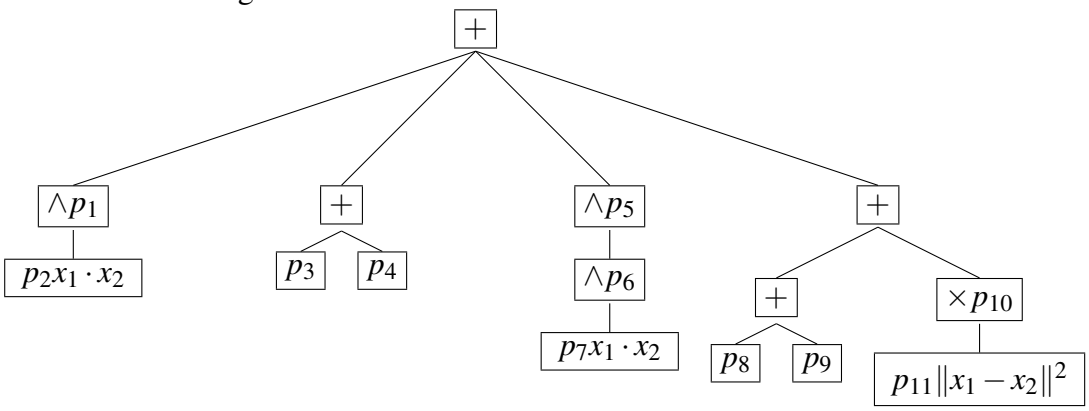


Fig. 5.15: Árbol creado aleatoriamiente sin simplificar ni normalizar: entradas x_1, x_2 y 11 parámetros $p_1, \cdots, p_{11}$

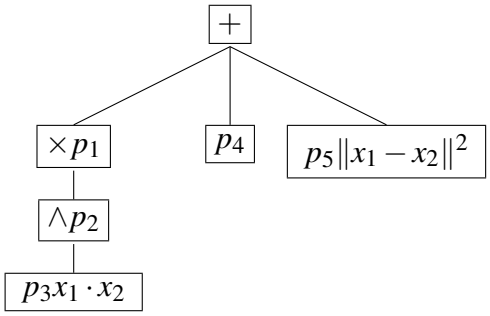


Fig. 5.16: Árbol simplificado pero sin normalizar: entradas x_1, x_2 y 5 parámetros $p_1, \cdots, p_5$

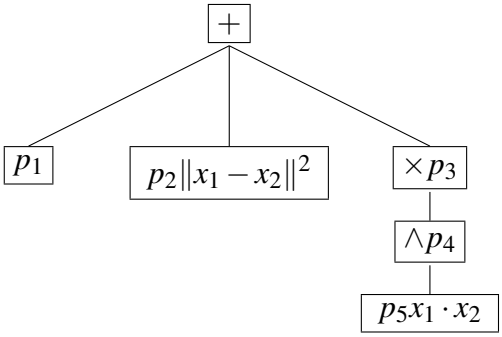


Fig. 5.17: Árbol simplificado y normalizado: entradas x_1, x_2 y 5 parámetros $p_1, \cdots, p_5$

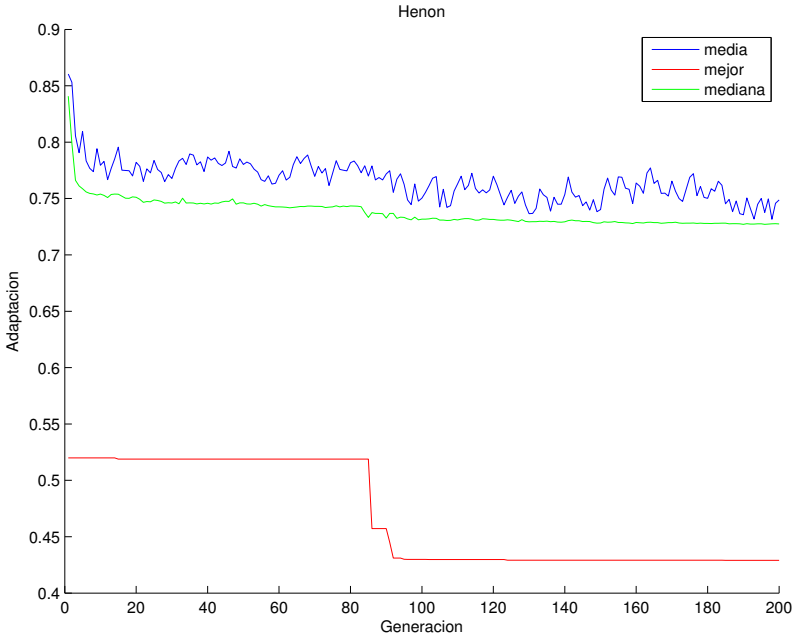


Fig. 5.18: Evolución de la adaptación de la población del GP para Henon

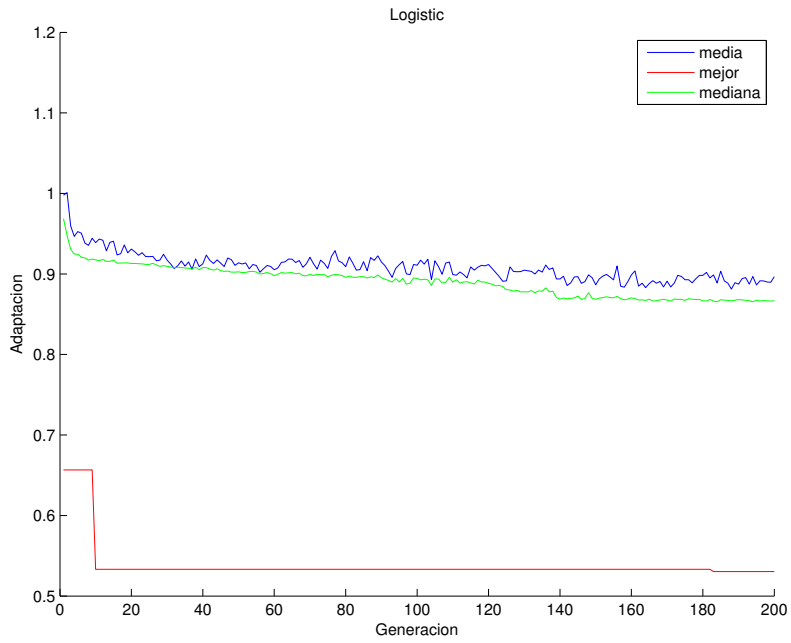


Fig. 5.19: Evolución de la adaptación de la población del GP para Logistic

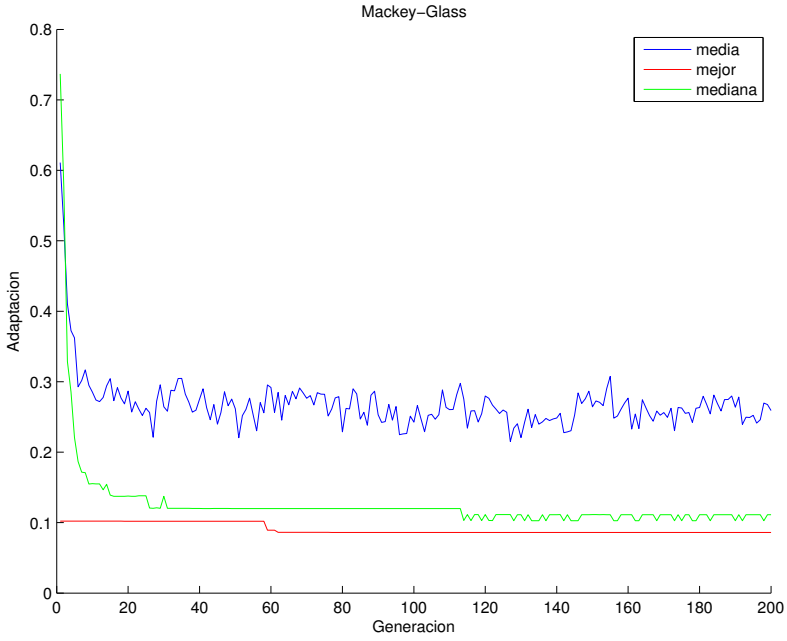


Fig. 5.20: Evolución de la adaptación de la población del GP para Mackey-Glass

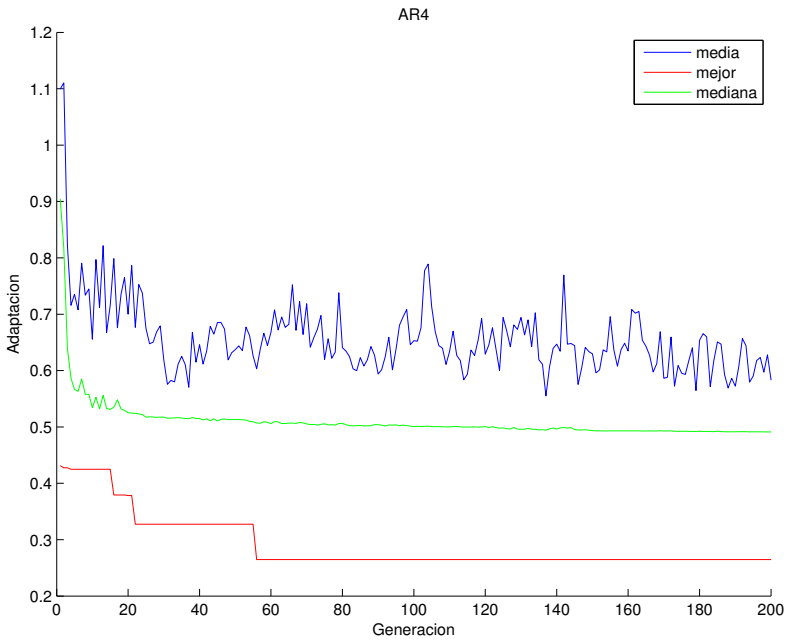


Fig. 5.21: Evolución de la adaptación de la población del GP para AR4

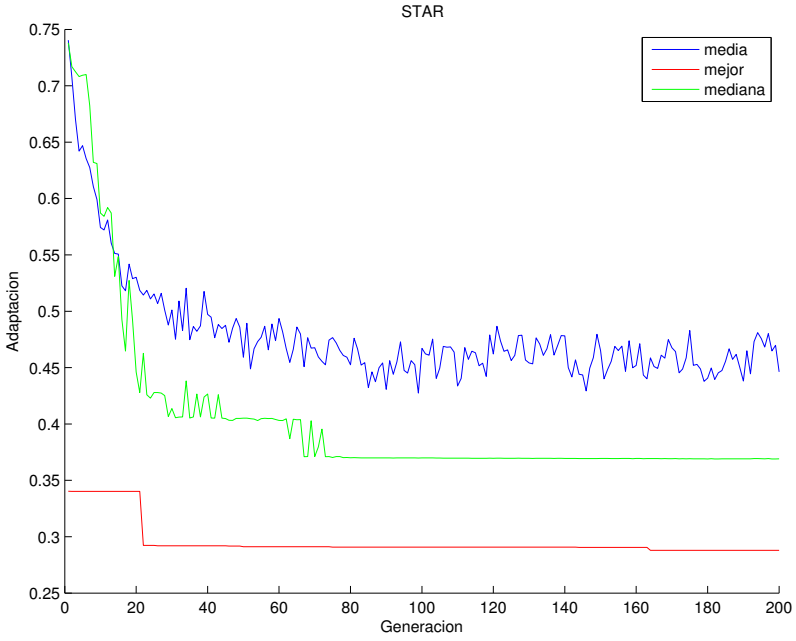


Fig. 5.22: Evolución de la adaptación de la población del GP para STAR

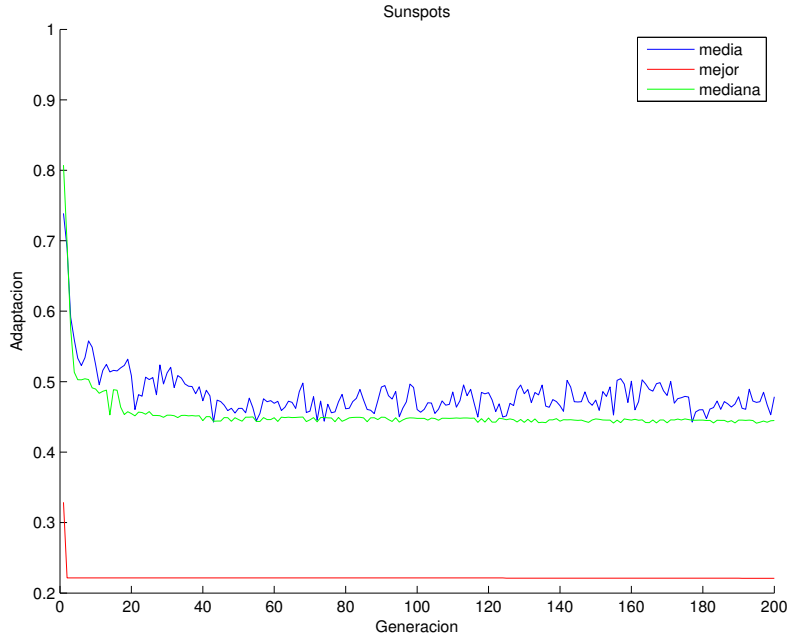


Fig. 5.23: Evolución de la adaptación de la población del GP para Sunspots

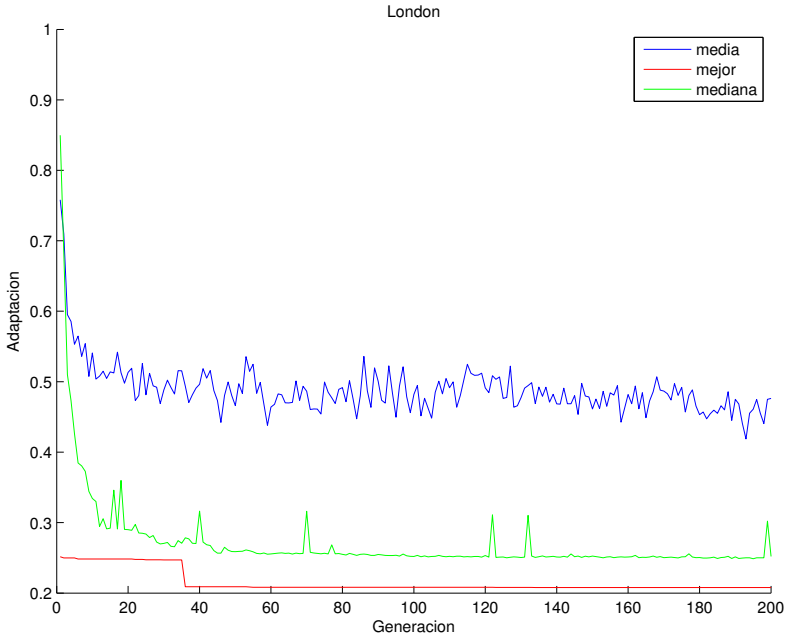


Fig. 5.24: Evolución de la adaptación de la población del GP para London

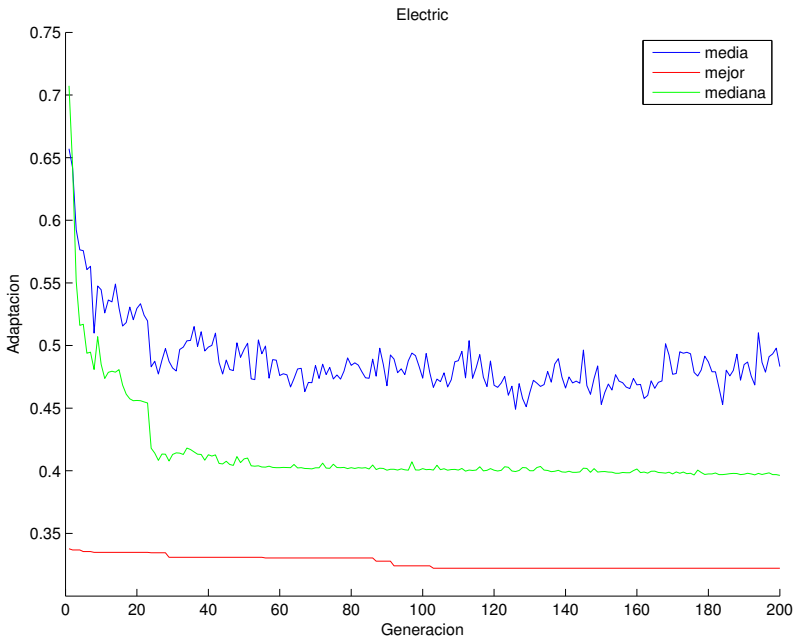


Fig. 5.25: Evolución de la adaptación de la población del GP para Electric

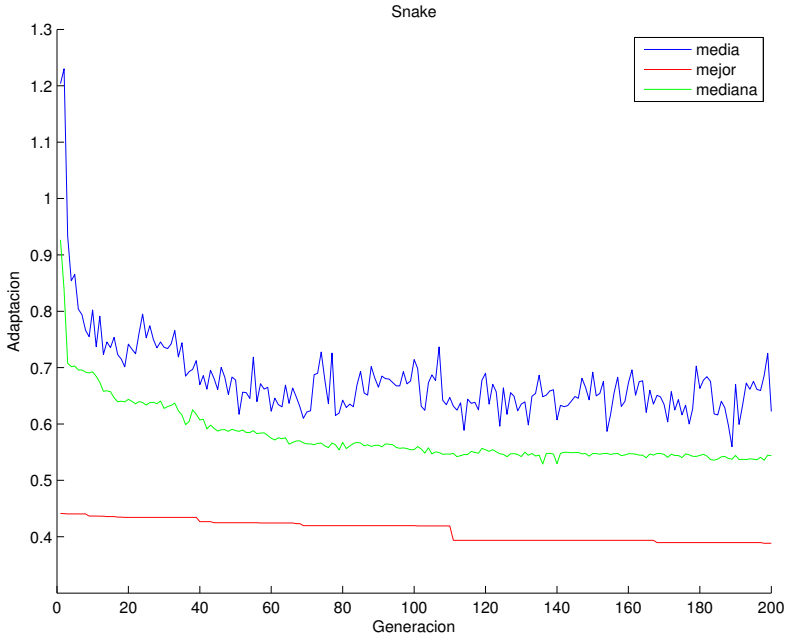


Fig. 5.26: Evolución de la adaptación de la población del GP para Snake

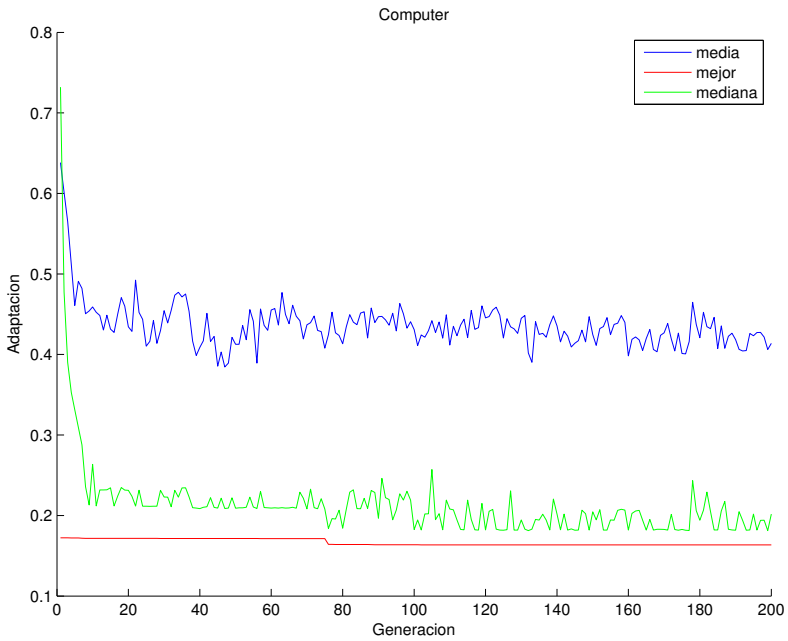


Fig. 5.27: Evolución de la adaptación de la población del GP para Computer

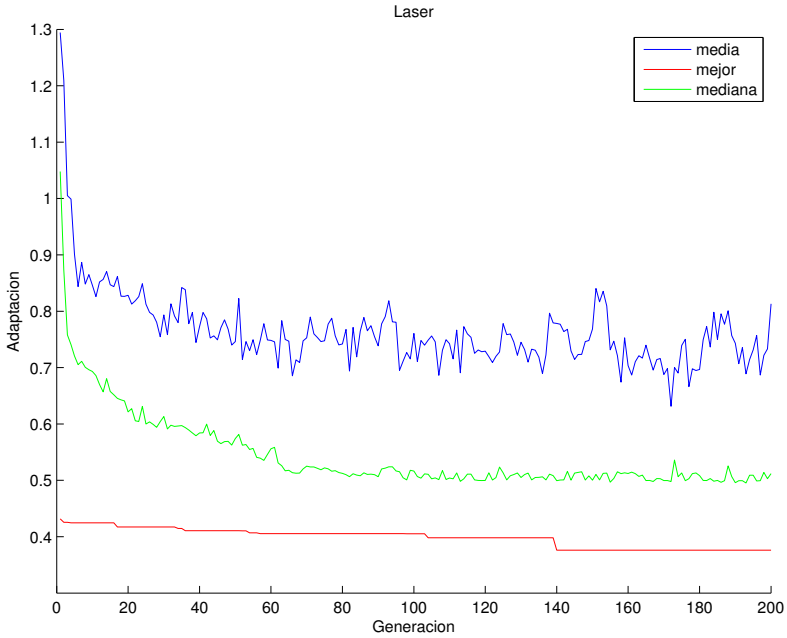


Fig. 5.28: Evolución de la adaptación de la población del GP para Laser

6. HPC EN MATLAB

Matlab es un programa ampliamente utilizado en ámbitos científicos y técnicos, debido a que es un entorno de trabajo agradable, sencillo y potente, contando con un lenguaje interno sencillo a la par que flexible y con muchas funcionalidades permitiendo tanto ejecución interpretada como compilación de aplicaciones. A lo anterior hay que sumar además la gran ventaja que supone su amplísimo soporte para todo tipo de aplicaciones científicas y técnicas en módulos de código denominados *toolboxes* (*cajas de herramientas*). Debido a su éxito, es natural que se haya intentado dar soporte para usar Matlab en entornos de computadores de alto rendimiento (*High Performance Computing*, HPC) tanto por parte de iniciativas comerciales como no comerciales. En este capítulo se hablan de librerías para acceso a funcionalidades de cómputo de alto rendimiento desde Matlab. En una primera parte se describe una implementación de una interfaz software para uso de las capacidades de cómputo de las tarjetas gráficas Nvidia (CUDA) en Matlab. En una segunda parte se comparan las opciones disponibles para usar MPI desde Matlab, entre ellas 2 desarrolladas en el departamento de Arquitectura y Tecnología de la Universidad de Granada, MPITB y MPIMEX (en el desarrollo de esta última participa el autor de esta memoria).

6.1. CUDA: Compute Unified Device Architecture

6.1.1. Antecedentes: GPUs

La unidad de procesamiento gráfico o GPU (*Graphics Processing Unit*) es un coprocesador que descarga de gran parte de las tareas relacionadas con los gráficos a la CPU. Hoy en día las GPU son muy potentes y pueden incluso superar la frecuencia de reloj de una CPU antigua (más de 500MHz). De hecho, las tarjetas gráficas en que se incluyen poseen su propia memoria y a todos los efectos son un sistema embebido especializado. La potencia de las GPU y su dramático ritmo de desarrollo reciente se deben a dos factores diferentes. El primer factor es la alta especialización en tareas de procesamiento gráfico. Por ejemplo, las GPU actuales están optimizadas para cálculo con valores en coma flotante, predominantes en los gráficos 3D. Por otro lado, la mayor parte de los algoritmos gráficos que ejecutan

conllevan un alto grado de paralelismo inherente, al ser sus unidades fundamentales de cálculo (vértices y píxeles) completamente independientes. Los modelos actuales de GPU suelen tener una media docena de procesadores de vértices, y hasta dos o tres veces más procesadores de fragmentos o píxeles. De este modo, con una frecuencia de reloj inferior a las de las CPUs, se obtiene en una potencia de cálculo mucho mayor gracias a su arquitectura en paralelo.

Al inicio, la programación de la GPU se realizaba con llamadas a servicios de interrupción de la BIOS. Tras esto, la programación de la GPU se empezó a hacer en el lenguaje ensamblador específico a cada modelo. Posteriormente, se situó un nivel más entre el hardware y el software, diseñando las API (*Application Program Interface*), que proporcionaban un lenguaje más homogéneo para los modelos existentes en el mercado. El primer API usado ampliamente fue estándar abierto OpenGL (*Open Graphics Language*), tras el cuál Microsoft desarrolló *DirectX*.

Tras el desarrollo de APIs, se decidió desarrollar lenguajes de alto nivel para gráficos. Por ejemplo, "*OpenGL Shading Language*"(GLSL) es un lenguaje estándar de alto nivel, asociado a la biblioteca OpenGL, implementado en principio por todos los fabricantes. La empresa californiana NVIDIA creó un lenguage propietario llamado *Cg* (*C for graphics*), con mejores resultados que GLSL en las pruebas de eficiencia. En colaboración con NVIDIA, Microsoft desarrolló su *High Level Shading Language*, HLSL, prácticamente idéntico a Cg, pero con ciertas incompatibilidades menores.

En los últimos años se intenta aprovechar la gran potencia de cálculo de las GPU para aplicaciones no relacionadas con los gráficos, en lo que desde recientemente se viene a llamar GPGPU, o GPU de propósito general (*General Purpose GPU*). Nombrando 2 proyectos de GPGPU:

1. OpenCL (*Open Computing Language*): API y lenguaje de programación que juntos permiten crear aplicaciones con paralelismo a nivel de datos y de tareas que pueden ejecutarse tanto en GPUs como CPUs. El lenguaje está basado en el C99, eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales. Apple creó la especificación original y la propuso al Grupo Khronos para convertirla en un estándar abierto y libre. El 16 de junio de 2008 Khronos creó el *Compute Working Group* para llevar a cabo el proceso de estándarización.

2. CUDA (*Compute Unified Device Architecture*): hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA, que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPUs de nVidia.

En esta memoria se hace uso de la última para realizar una implementación sobre GPU del KWKNN.

6.1.2. CUDA

CUDA (siglas de *Compute Unified Device Architecture*, Arquitectura Unificada de Dispositivo de Cómputo) en palabras de la empresa desarrolladora NVIDIA es descrita como sigue:

La tecnología NVIDIA CUDA (TM) es el único entorno basado en el lenguaje C que permite a los programadores escribir software para resolver problemas computacionales complejos en menos tiempo aprovechando la gran capacidad de procesamiento paralelo de las GPU multinúcleo. Miles de programadores están utilizando ya las herramientas gratuitas de desarrollo de CUDA (válidas para millones de GPU ya instaladas en el mercado) a fin de acelerar todo tipo de aplicaciones, desde herramientas de codificación de audio y vídeo, hasta software para la exploración de gas y petróleo, el diseño de productos, la generación de imágenes en medicina o la investigación científica.

Desde el punto de vista de arquitectura de computadoras, la biblioteca permite acceder a las capacidades de la tarjeta gráfica (o GPU) que actúa como un sistema embebido que es en sí una arquitectura multinúcleo de memoria compartida. La biblioteca da acceso a diversas funciones para acceder a las capacidades de la tarjeta, a la vez que implementa una serie de extensiones al C estándar. Dichas extensiones permiten la definición de funciones que se ejecutarán en la tarjeta gráfica (de ahora en adelante **dispositivo**), que se denominan en su documentación **kernels**. Este término se puede confundir con el concepto de *kernel* asociado a los métodos *kernel* a los que se hace continua referencia a lo largo de esta memoria, por lo que las denominaremos aquí por claridad **función/funciones-gpu**.

El modelo que impone CUDA fomenta el paralelismo de grano fino, y facilita la implementación de aplicaciones SIMD, *Single Instruction, Multiple Data*, es decir, un mismo código que se aplica muchos datos distintos. El modelo de programación de CUDA está diseñado para que se creen aplicaciones que de forma transparente escalen su paralelismo al incrementar el número de núcleos computacionales sin modificar los programas. Este diseño se basa en tres abstracciones, que son:

1. La jerarquía de grupos de hebras.
2. Las memorias compartidas.
3. Las barreras de sincronización.

Las hebras que ejecutan una función-cuda se organizan en una **rejilla** de hasta 3 dimensiones en la cual cada hebra pertenece a un grupo denominado **bloque** de

hasta 512 componentes, y posee un identificador único (y accesible) tanto a nivel de rejilla como de bloque. La única abstracción de control global entre hebras está definido a nivel de un bloque, y es una barrera de sincronización. Las hebras se comunican entre sí a través la memoria compartida del dispositivo que se organiza en una jerarquía de 3 niveles básicos:

- memoria privada de la hebra
- memoria compartida del bloque, visible sólo para las hebras pertenecientes a un mismo bloque (denominada como *shared memory* en la documentación)
- memoria global del dispositivo, visible para todas las hebras (denominada como *global memory* en la documentación). La memoria global es persistente entre llamadas de funciones-cuda, a diferencia de los 2 niveles inferiores.

Adicionalmente, existen 2 espacios de memoria global de solo lectura: la memoria constante y la de textura. Cada espacio de memoria global está optimizado para un uso específico. En el caso de la memoria de textura hay incluso diferencias en el modo de direccionamiento y formato de los datos.

Las aplicaciones que hacen uso de CUDA tienen partes que se ejecuta en el computador (**host**) y otras que se ejecutan en el *dispositivo*, las funciones-cuda. Así pues, la tarjeta gráfica actúa como un coprocesador, aunque la memoria del dispositivo y del host se mantienen separadas. CUDA proporciona funciones para la reserva y liberación de memoria en el dispositivo y para la transferencia de datos en esta y la memoria de host (la memoria RAM del computador).

CUDA es especialmente atractiva para la computación científica ya que arquitectura económica, accesible, *estándar* (entre tarjetas de la compañía NVIDIA) y escalable para la programación paralela [Nickolls et al., 2008]. Es especialmente adecuada para aplicaciones con gran paralelismo de datos [Garland et al., 2008]. La compañía distribuye, incluida con CUDA, una implementación muy parcial de LAPACK, CUBLAS, que carece de las funciones más complejas como la descomposición LU, QR y de Cholesky, aunque ya hay estudios de implementaciones de las mismas [Volkov and Demmel, 2008].

Una consulta rápida de la literatura de los últimos años mostrará aplicaciones científicas (astronomía [Belleman et al., 2008], bioinformática [Manavski and Valle, 2008], física [Liu et al., 2008], etc) y de ingeniería que se han beneficiado con la implementación en CUDA, aunque en esta memoria destacaremos 2: la implementación de SVM para clasificación de [Catanzaro et al., 2008] y de los K vecinos más cercanos de [Garcia et al., 2008].

Ahora bien, CUDA presenta una serie de problemas y limitaciones:

1. Sólo esta disponibles para tarjetas gráficas de la marca NVIDIA, a diferencia de otras alternativas, como OpenCL[1].

2. El ancho de banda entre la memoria del host y la del dispositivo puede ser un cuello de botella.

3. El soporte de los tipos de datos flotantes presenta problemas. Sólo las tarjetas más recientes soportan flotantes de doble precisión (64 bits) y su implementación no se ajusta del todo al estándar IEEE 754. En cuanto al soporte de flotantes de simple precisión (32 bits) tampoco se ajusta del todo a la especificación IEEE, y presenta problemas asociados a los redondeos y a la carencia de soporte para los números desnormalizados o NaNs (*Not a Number*, constantes para codificar un valor indefinido resultado de un error).

4. El código de las funciones-cuda no puede ser recursivo, ni hacer uso de punteros a funciones así como otras restricciones asociadas al modelo de programación[2].

5. Las aplicaciones que se alejen del esquema SIMD son fuertemente penalizadas en su rendimiento y facilidad de implementación.

6.1.3. Las capacidades de cómputo de CUDA y soporte para flotantes

Las tarjetas NVIDIA con controladores que soporten CUDA, dejando aparte de sus distintas especificaciones de memoria y GPU, no tienen las mismas capacidades de cómputo en CUDA. Hay definidos en este momento 3 niveles capacidades de cómputo: `1.0`, `1.1` y `1.3`. Cada nivel implementa las capacidades del anterior y añade a su vez nuevas funcionalidades. Estas especificaciones se pueden consultar en el documento [CUD, 2007], y se refieren sobre todo capacidades de cómputo paralelo (número máximo de hebras manejables, en ejecución, límites de organización en rejilla, etc), operaciones implementadas (sobre todo referencias a las funciones atómicas en memoria principal) y, lo que es muy relevante en nuestro caso, el soporte de tipos coma flotantes. Los números reales y sus operaciones son soportados a partir del nivel `1.0`, pero sólamente los de simple precisión (32 bits). Para tener soporte de flotantes de doble precisión (64 bits) hay que disponer de una tarjeta de nivel `1.3`. Este es un fuerte factor limitador, amén de que la implementación realizada de los tipos puntos flotantes en las GPU no sigue exactamente el estándar IEEE-754, teniendo una serie de desviaciones que enumeramos a continuación:

[1] `http://www.khronos.org/opencl/`

[2] ver `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`

- El modo de redondeo no es dinámicamente configurable, sin embargo la mayor parte de las operaciones soportan los modos de redondeo IEEE, vía funciones de dispositivo.

- No existe mecanismo para detección de las excepciones de punto flotante que ocurran, y todas las operaciones se comportan como si la excepciones IEEE fueran enmascaradas, notificando una respuesta enmascarada como definido en IEEE-754 si ocurre un evento generado de excepción. Por el mismo motivo, aunque el código SNaN (*Signalling Not A Number*, notificación de un resultado erróneo) está soportado, no hay notificación.

- Valores absolutos y negaciones no siguen el IEEE-754 con respecto a los valores NaN, que se devuelven sin cambios.

- Sólo para flotantes de simple precisión (32 bits):

  - Sumas y productos son combinadas con frecuencia en una sóla operación de *producto y suma* (*multiply-add instruction*, FMAD), que trunca el resultado intermedio del producto.
  - La división se implementa vía el recíproco de un modo que no sigue el estándar IEEE 754.
  - La raíz cuadrada se implementa vía el recíproco de un modo que no sigue el estándar IEEE 754.
  - Los únicos redondeos soportados para sumas y productos son el *redondeo al par más cercano* y el truncamiento (*round-to-nearest-even* y *round-towards-zero*), pero vía modos estáticos, los redondeos a valores infinitos tampoco se soportan.
  - Los número desnormalizados no son soportados, en la aritmética de punto flotante y las operaciones de comparación se convierten los valores desnormalizados de operandos a zeros antes de aplicación.
  - Resultados con subdesbordamiento (*underflow*) son convertidos a zero.
  - El resultado de una operación que implique uno o más operandos con valores NaN es un NaN con patrón de bit 0x7fffffff.

- Sólo para flotantes de simple precisión (64 bits):

  - El *redondeo al par más cercano* (*round-to-nearest-even*) es el único modo IEEE de redondeo soportado para el recíproco, la división y la raíz cuadrada.

6.1.4. CUBLASMEX

Es posible usar CUDA desde Matlab a través de la interfaz mex[3]. CUBLASMEX es una toolbox que actúa como interfaz entre Matlab y las funciones de CUBLAS. Aunque no es novedoso de por si, existiendo por ejemplo el producto comercial *Jacket* de la empresa AccelerEyes[4] sí es una contribución valiosa al ponerse disponible a la comunidad científica para su estudio y uso.

CUBLAS[5] hace disponible desde C un subconjunto de las funcionalidades BLAS y LAPACK que se ejecuta en la GPUs de NVIDIA. CUBLASMEX proporciona una vez compilado como un fichero mex una serie de funciones para inicializar el entorno CUDA, trasladar hacia y desde la GPU variables de Matlab (matrices bidimensionales densas de flotantes de doble y simple precisión), y utilizar las funciones disponibles de CUBLAS desde el propio Matlab. Este diseño básico ha sido usado para ampliar funcionalidades adicionales que no fueran cubiertas por CUBLAS, por ejemplo funciones de apoyo para la implementación del KWKNN (como se puede ver en el Apartado 4.9).

A continuación se listan algunas de las principales funciones de la toolbox que ha sido implementada a raíz del trabajo descrito:

```
check_value=CUBLASmex('Init');
init CUBLASmex environment, must be used before all the other
CUBLASmex functions if success then check_value equals 0, else
check_value equals 1.

check_value cublasMatrix]=CUBLASmex('Matrix',rows,cols);
allocate a matrix of rows x cols in device memory;
rows, cols > 1, the number of matrices than can be allocated
in device memory is limited by CUBLASmex implementation and
the size of the memory device if success then check_value
equals 0, else check_value equals 1.

check_value = CUBLASmex('Free',cublasMatrix);
free from device memory the input cublasMatrix allocated by
Matrix, if success then check_value equals 0, else check_value
equals 1.

check_value = CUBLASmex('SetMatrix',cublasMatrix,Matrix);
```

[3] http://developer.download.nvidia.com/compute/cuda/1_0/Accelerating\%20Matlab\%20with\%20CUDA.pdf

[4] http://www.accelereyes.com/

[5] http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0.pdf

```
copy to the input cublasMatrix in device memory the values of
the input Matrix in host memory matrices must have the same
size if success then check_value equals 0, else check_value
equals 1.

check_value = CUBLASmex('GetMatrix',cublasMatrix,Matrix);
copy to the input Matrix in host memory the values of the input
cublasMatrix in device memory matrices must have the same size
if success then check_value equals 0, else check_value equals 1.

check_value = CUBLASmex('Finalize');
end CUBLASmex environment, must be used after the end of the
program that use CUBLASmex functions, and not before other call,
if success then check_value equals 0, else check_value equals 1.

check_value = CUBLASmex('Scopy', n, x, incx, y, incy );
y(1:incy:incy*n) = x (1:incx:incx*n)0,
if success then check_value equals 0, else check_value equals 1.

check_value = CUBLASmex( 'Sgemm',
                         'n'|'t',
                         'n'|'t',
                         alpha,
                         A, B, beta, C );
'n' 'n' -> alpha*A*B+ beta *C
't' 'n'-> alpha*A'*B+ beta *C
'n' 't'-> alpha*A*B'+ beta *C
't' 't'-> alpha*A'*B'+ beta *C
if success then check_value equals 0, else check_value equals 1.

check_value = CUBLASmex( 'Ssymm', 'l'|'r', 'u'|'l',
                         alpha, C, B, beta, C );
A symmetric, 'u' upper diagonal 'l' lower diagonal0
'l' -> alpha*A*B+ beta *C
'r' -> alpha*B*A+ beta *C
if success then check_value equals 0, else check_value equals 1.

check_value = CUBLASmex( 'Strsv', 'l'|'u', 'n'|'t', 'u'|'n',
 T, x );
Solve system T x = b0
'u' upper diagonal 'l' lower diagonal0
'n' normal 't' transpose
```

```
'u' unit 't' non-unit
 T triangular matrix
 x vector.initialized to b, after call contains solution
if success then check_value equals 0,
else check_value equals 1.

...
```

6.1.5. CUBLASMEX: rendimiento

La interfaz CUBLASMEX proporciona operaciones básicas de transferencia de matrices entre memoria del PC y el dispositivo así como el acceso a las operaciones de BLAS y LAPACK de CUBLASMEX. La forma de trabajar más eficiente de utilizar CUBLASMEX es:

1. Al inicio del programa, trasladar el máximo posible de variables al dispositivo.
2. Hacer el máximo posible de cómputo en la tarjeta evitando si no es estrictamente necesario traslados de datos entre memoria y dispositivo.
3. Finalmente, recoger los resultados trasladándolos del dispositivo a la memoria.

Se muestran a continuación una serie de pruebas que miden el rendimiento de los movimientos de datos entre memoria y dispositivos, así como la mejora del tiempo de cómputo que se consigue con 3 funciones que proporciona la interfaz: *saxpy*, *dgemv* y *sgemm*.

La máquina utilizada en este trabajo es un PC con un procesador INTEL CORE 2 QUAD 2.83GHZ, con 8 GB de memoria RAM DDR, con una tarjeta GEFORCE 9800 GTX (128 procesadores de flujo a 1688 MHz, 512 MB de memoria GDDR3).

6.1.5.1. Traslado de datos

CUBLASMEX utiliza un buffer de memoria reservado con la función *cudaMallocHost* para optimizar las transferencias de datos entre memoria del PC y tarjeta gráfica (reserva una zona de memoria accesible desde la tarjeta), como se indica en la documentación de CUDA [CUD, 2007]. Las operaciones correspondientes de CUBLASMEX son, respectivamente, *SetMatrix* y *GetMatrix*. Se han medido los tiempos de traslado de 100 vectores columnas ($N \times 1$) y matrices cuadradas ($N \times N$) con $N = 1000, 2000, \cdots, 8000$. Los resultados se pueden ver en las Figuras 6.1 y 6.2. El coste ha resultado ser idéntico tanto en el sentido de memoria del PC al dispositivo como en el sentido contrario, y los tiempos aumentan conforme aumentan el valor N de $8{,}872000e-05$ segundos con $N = 1000$ a $1{,}227500e-04$ segundos con $N = 8000$ para vectores y de $3{,}416820e-03$ segundos con $N = 1000$ a $1{,}947926e-01$ segundos con $N = 8000$ para matrices.

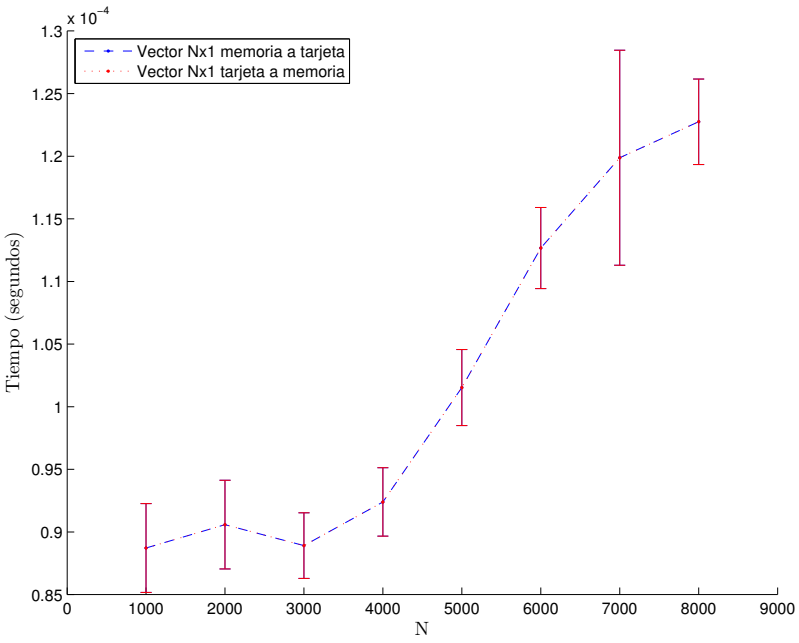


Fig. 6.1: Tiempos de transferencia de vectores $N \times 1$ entre memoria y dispositivo con CUBLASMEX.

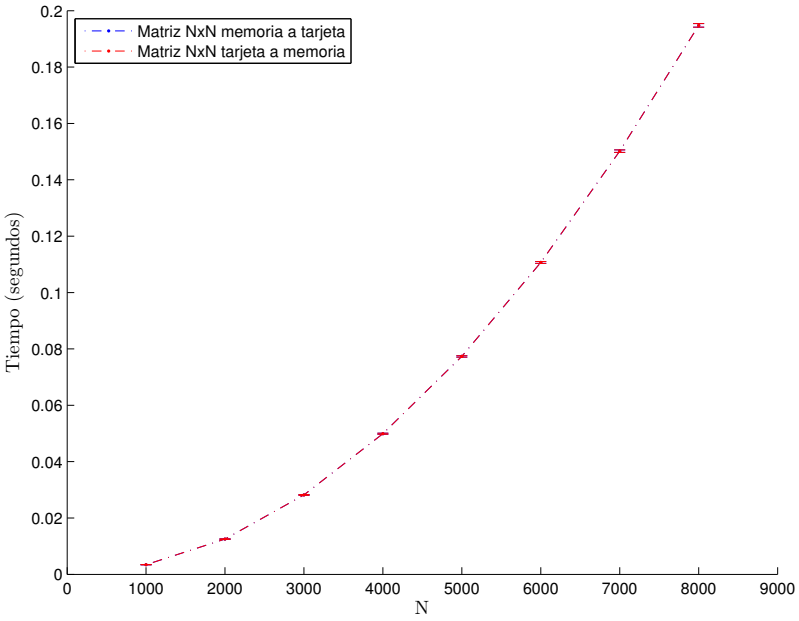


Fig. 6.2: Tiempos de transferencia de matrices $N \times N$ entre memoria y dispositivo con CUBLASMEX.

6.1.5.2. Pruebas con saxpy

La operación de BLAS de nivel 1 *saxpy* es una combinación de producto con escalar y suma de vectores de flotantes de simple precisión (32 bits):

$$\mathbf{y} = \alpha \cdot \mathbf{x} + \mathbf{y} \tag{6.1}$$

donde α es un escalar y x e y son vectores de N elementos.

Se ha ejecutado 100 veces la operación con vectores aleatorios de valores en el rango $[0,10]$, de dimensiones $N \times 1$ con $N = 1000, 2000, \cdots, 100000$, y se ha sacado la ganancia con respecto a la operación *daxpy* de BLAS en el PC de pruebas. La diferencia entre *daxpy* y *saxpy* es que la primera opera sobre flotantes de doble precisión (64 bits) que son los que por defecto utiliza Matlab. Esto sirve para comparar el rendimiento de tiempo no sólo contra la operación equivalente que realmente se usará, sino también para medir los errores que se cometerán debido a las distintas precisiones. En la Figura 6.3 se puede observar como la ganancia aumenta con N, mientras que la media del valor absoluto de la diferencia de resultados se mantiene casi inapreciable, como se puede ver en la Figura 6.4.

6.1.5.3. Pruebas con sgemv

La operación de BLAS de nivel 2 *sgemv* es una combinación de producto matriz-vector y suma de vectores de flotantes de simple precisión (32 bits):

$$\mathbf{y} = \alpha \cdot \mathbf{A} \cdot \mathbf{x} + \beta \cdot \mathbf{y} \tag{6.2}$$

donde α y β son escalares, A es una matriz $M \times N$ y x e y son vectores de M elementos.

Al igual que en el caso anterior, la operación ejecuta 100 veces con matrices y vectores aleatorios de valores en el rango $[0,10]$, de dimensiones $N \times N$ y $N \times 1$ con $N = 1000, 2000, \cdots, 8000$, y se ha sacado la ganancia con respecto a la operación *dgemv* de BLAS en el PC de pruebas. La diferencia entre *dgemv* y *sgemv* es que la primera opera sobre flotantes de doble precisión (64 bits) que son los que por defecto utiliza Matlab, y al igual que en los experimentos con *saxpy*, sirve para comparar el rendimiento de tiempo no sólo contra la operación equivalente que realmente se usará, sino también para medir los errores que se cometerán debido a las distintas precisiones. En la Figura 6.5 se puede observar como la ganancia aumenta con N, al igual que la media del valor absoluto de la diferencia de resultados en la Figura 6.6. De hecho, se puede observar la ganancia es muy superior al caso *saxpy* que apenas llega a una ganancia de 7 con $N = 10000$ frente a la ganancia de casi 2000 con $N = 8000$ obtenida con *sgemv*. Sin duda, al implicar matrices, la operación se beneficia más de las características de cómputo de la tarjeta al poder paralelizar más los cálculos, aunque, por desgracia, aquí sí que se hacen patentes y aumentan con N los errores de redondeo.

6.1.5.4. Pruebas con sgemm

La operación de BLAS de nivel 3 *sgemm* es una combinación de producto y suma matriz-matriz de matrices de flotantes de simple precisión (32 bits):

$$\mathbf{C} = \alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta \cdot \mathbf{C} \tag{6.3}$$

donde α y β son escalares, A es una matriz $M \times L$, B es una matriz $L \times N$, A es una matriz $M \times N$, .

Siguiendo con lo visto hasta ahora, se ha ejecutado 100 veces *sgemm* con matrices aleatorias de valores en el rango $[0, 10]$, de dimensiones $N \times N$ con $N = 1000, 2000, \cdots, 6000$. No se ha podido probar con matrices de mayor tamaño debido a la memoria de la tarjeta gráfica disponible en el PC de pruebas (512 MB, 3 matrices de 7000×7000 de flotantes de simple precisión ocupan ≈ 560 MB). Al igual que en los experimentos con *saxpy* y *sgemv*, se utiliza la versión para flotantes de doble precisión *dgemm* en Matlab, midiendo tanto rendimiento de tiempo y como errores debidos a las distintas precisiones. En la Figura 6.7 se puede observar como la ganancia aumenta con N, al igual que la media del valor absoluto de la diferencia de resultados en la Figura 6.8. De hecho, y siguiendo con una clara tendencia, se puede observar como la ganancia es muy superior comparada tanto con el caso *saxpy* como *sgemv*, con un valor cerca $16e5$ para $N = 6000$. Por desgracia, también aumentan los errores implicados con respecto a *sgemv*.

6.1.5.5. Conclusiones de las pruebas

El uso de CUBLASMEX permite obtener una ganancia de velocidad importante en los cómputos siempre que impliquen suficientes operaciones con matrices de tamaño adecuado, de al menos $N \times N$ con $N >= 1000$, y sin perder de vista la memoria disponible en la tarjeta gráfica y los tiempos consumidos en trasladar datos entre memoria del ordenador al dispositivo. Además hay que tener presente que, como en nuestro caso, si hay que pasar de flotante de doble precisión de Matlab (64 bits) a flotantes de simple precisión (32 bits) por las características de la tarjeta se irán acumulando errores con las operaciones.

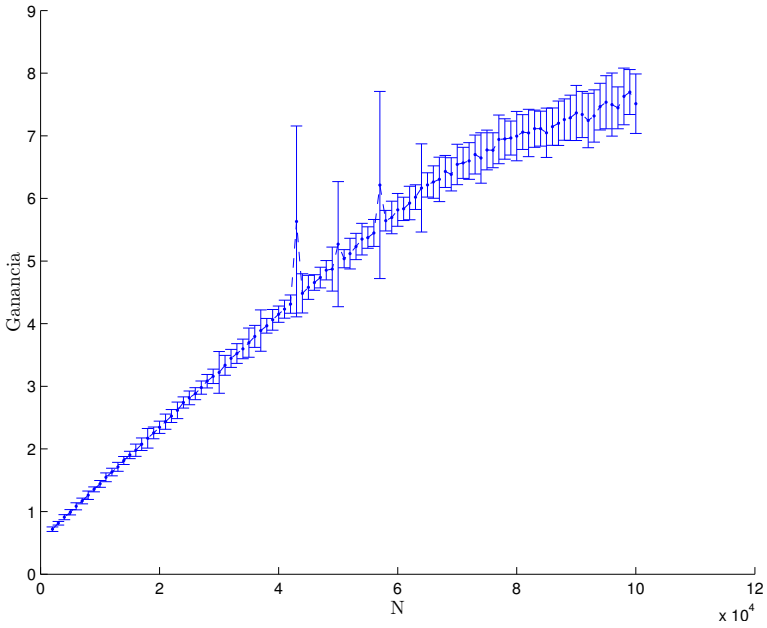


Fig. 6.3: Ganancia de velocidad de *saxpy* en CUBLASMEX frente a *daxpy* de BLAS.

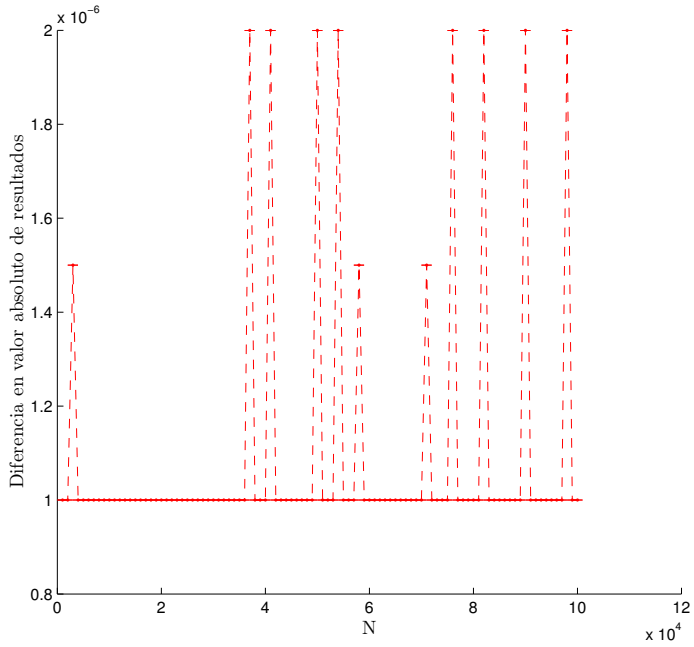


Fig. 6.4: Media del valor absoluto de la diferencia de resultados entre *saxpy* en CUBLASMEX frente a *daxpy* de BLAS.

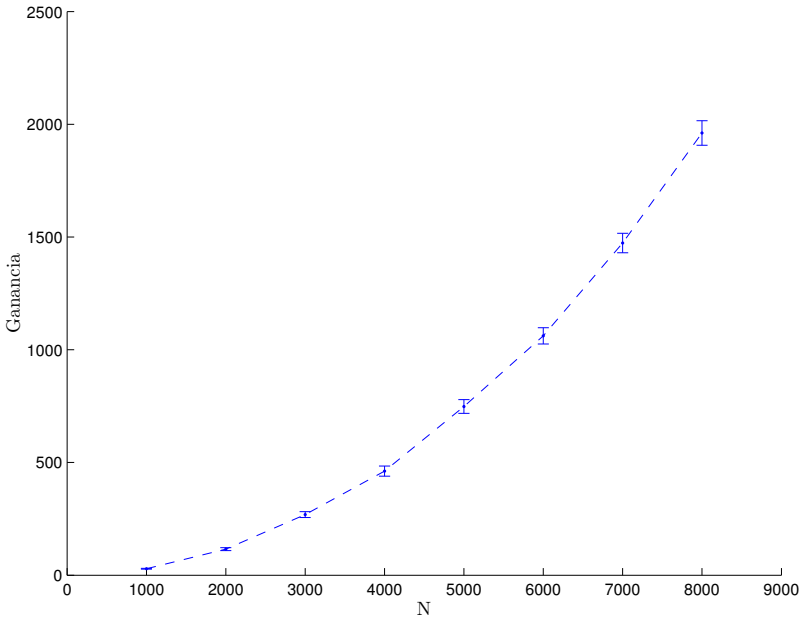


Fig. 6.5: Ganancia de velocidad de *sgemv* en CUBLASMEX frente a *dgemv* de BLAS.

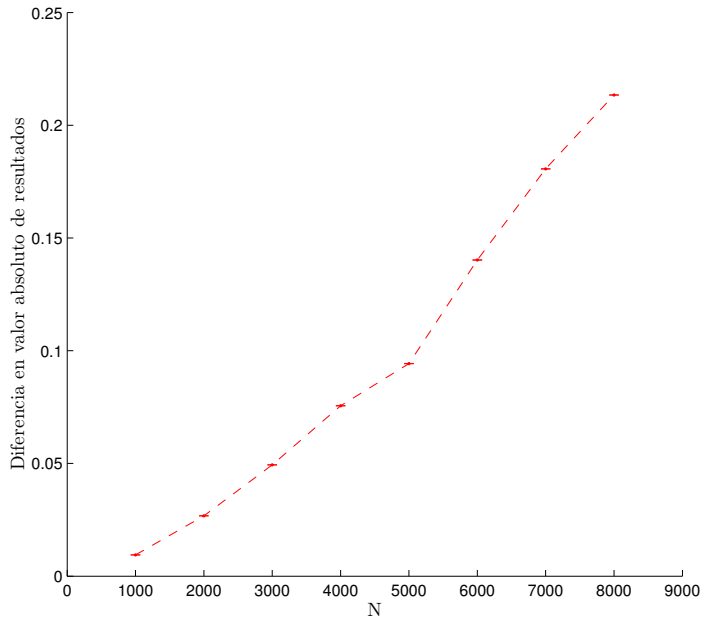


Fig. 6.6: Media del valor absoluto de la diferencia de resultados entre *sgemv* en CUBLASMEX frente a *dgemv* de BLAS.

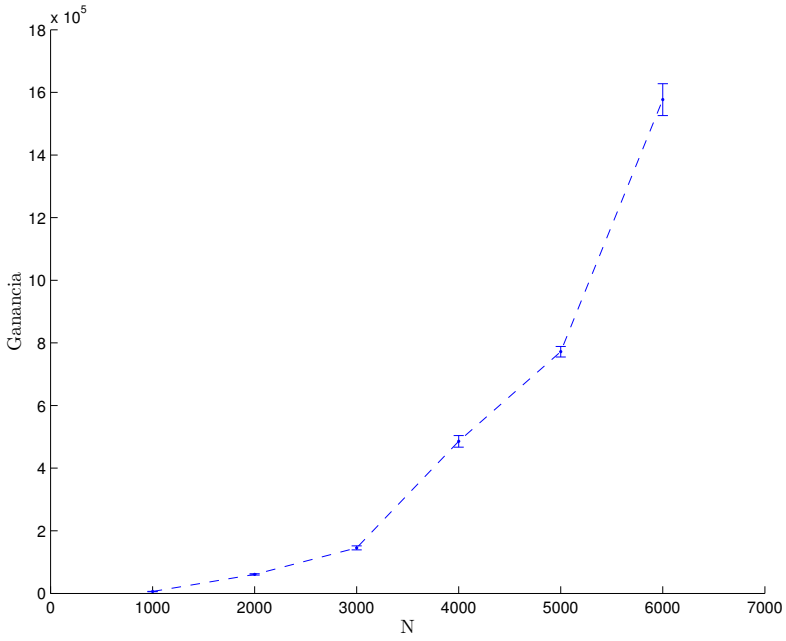


Fig. 6.7: Ganancia de velocidad de *sgemm* en CUBLASMEX frente a *dgemm* de BLAS.

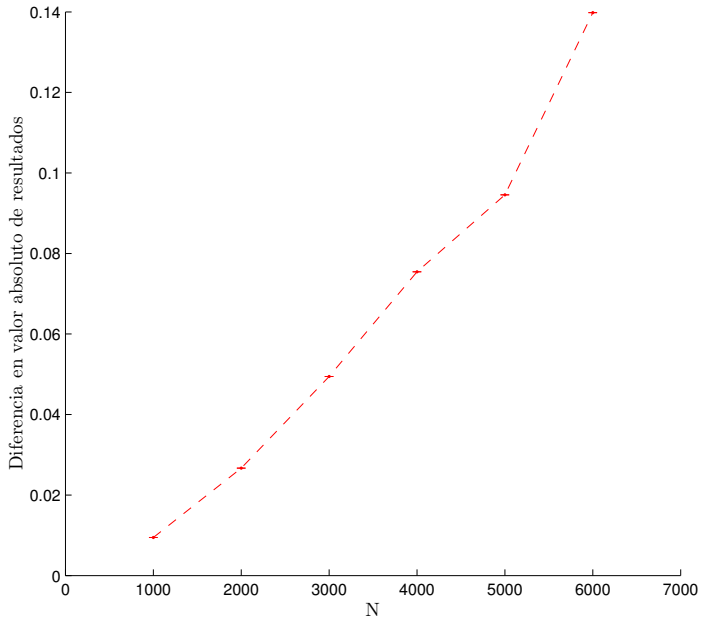


Fig. 6.8: Media del valor absoluto de la diferencia de resultados entre *sgemm* en CUBLASMEX frente a *dgemm* de BLAS.

6.2. Estudio y comparativa de *toolboxes* para MPI

El empleo de multiprocesadores y multicomputadores se presenta como una condición indispensable a la hora de poder aplicar algoritmos con un alto coste computacional a aplicaciones del mundo real. Dentro de los multiprocesadores y multicomputadores, se pueden distinguir dos tipos: 1) memoria compartida, donde todo el espacio de memoria es direccionable por todos los procesadores y 2) memoria distribuida donde cada procesador tiene su propio espacio de direcciones individual. Esta distinción es independiente del cómo esté organizada la memoria física. En un origen, las máquinas de memoria compartida eran considerablemente más caras que las de memoria distribuida puesto que éstas últimas son sencillamente implementables mediante una red, proporcionando buenas prestaciones. Sin embargo, el abaratamiento en la producción de los circuitos integrados así como el uso de nuevas tecnologías de fabricación están propiciando la aparición de nuevas máquinas bastante más asequibles y accesibles al mercado. Dentro de este contexto, el desarrollo de entornos, librerías y lenguajes de programación paralelos que se adapten a las máquinas paralelas se muestra como un imperativo [Kepner, 2004][Frederick P. Brooks, 1978].

Dentro del aprovechamiento del paralelismo a nivel de hebra/proceso, actualmente, hay dos grandes vertientes cimentadas en sus respectivos estándares: Multiprocesamiento Abierto (*Open Multi-Processing*, OpenMP) e Interfaz de Paso de Mensajes (*Message Passing Interface*, MPI). OpenMP está más orientado a las arquitecturas de memoria compartida mientras que MPI lo está a las de memoria distribuida, aunque es posible combinar ambas simultáneamente para explotar todas las posibilidades de un multicomputador de multiprocesadores.

OpenMP (`http://openmp.org/wp/`) es un conjunto de directivas, bibliotecas y variables de entorno diseñadas para ser usadas en máquinas de memoria compartida. Los lenguajes que pueden hacer uso de estos elementos son C/C++ y Fortran. OpenMP implementa multihebraje (*multithreading*) de modo que un proceso puede tener varias hebras en ejecución accediendo a recursos compartidos. Se está desarrollando una versión para *clusters*, es decir, máquinas de memoria distribuida conectadas mediante una red, denominada como Cluster OpenMP [Intel, 2008].

La Interfaz para Paso de Mensajes define un estándar par la comunicación entre procesos, siendo éste paradigma uno de los más utilizados a la hora de diseñar aplicaciones paralelas. El estándar es especialmente adecuado para máquinas de memoria distribuida aunque también puede utilizarse en máquinas de memoria compartida. Hay una gran cantidad de implementaciones tanto en el ámbito comercial como en iniciativas de código abierto: (Sun MPI[6], Intel MPI[7]) (OpenM-

[6] `http://www.sun.com/software/products/clustertools/`

[7] `http://www.intel.com/cd/software/products/asmo-na/eng/308295.htm`

PI[8], LAM[9], MPICH[10] y MPICH2[11]).

Matlab®[12] es una herramienta muy popular en los ámbitos científicos y técnicos debido a su amigable interfaz, su simpleza de programación y su amplia gama de *toolboxes* que proporcionan gran cantidad de aplicaciones para ingeniería, estadística, bioinformática, etc. lista para ser usadas. Existen algunas opciones comerciales para usar Matlab en la computación de altas prestaciones como la *Parallel Computing Toolbox* y *Matlab Distributed Computing Server* aunque existen varios proyectos alternativos en [Trefethen et al., 1996], [Almasi et al., 1999], [Norris, 2000], [Morrow and van de Geijn, 1998], [Quinn et al., 1998], [Rose et al., 1996], [Chauveau and Bodin, 1998] and Choy et al., 2005. Matlab permite la creación de aplicaciones independientes de modo que puedan ser ejecutadas sin Matlab, este punto es clave a la hora de poder realizar la integración con MPI. Hoy en día, las 3 principales alternativas a la hora de utilizar MPI en Matlab son: MatlabMPI [Kepner and Ahalt, 2004], MPITB [Fernandez et al., 2006] y MPIMEX [Guillén,][Guillen et al., 2008b]. MatlabMPI no utiliza una librería MPI sino que está programado en lenguaje Matlab. MPITB y MPIMEX están programadas en C/C++ para hacer de interfaz con las librerías de MPI. MPITB está más orientada al uso en la línea de comandos de Matlab mientras que MPIMEX está centrada en la ejecución de aplicaciones independientes de Matlab.

El Centro de Computacion Paralela de Edimburgo (*Edinburgh Parallel Computing Centre*, EPCC) en la Universidad de Edimburgo proporciona acceso a varias máquinas para computación de altas prestaciones. Las *toolboxes* de MPI más relevantes fueron instaladas en un cluster específico para poder desarrollar y ejecutar aplicaciones paralelas en Matlab. El estudio desarrollado en esta memoria consiste en el estudio, adaptación e instalación para una determinada máquina.

Matlab proporciona una herramienta denominada *Compiler* que permite generar aplicaciones totalmente independientes (*stand-alone*) del interprete de Matlab de modo que no es necesario tener instalado éste para poder ejecutar tales aplicaciones. Para que un *stand-alone* pueda ejecutarse es necesario disponer de unas bibliotecas que pueden ser distribuidas una vez generadas con Matlab. Estas bibliotecas contienen el *Matlab Component Runtime* (MCR) que interpreta los códigos escritos en los archivos .m tal y como el interprete de Matlab lo haría.

Un *Component Technology File* (CTF) es generado durante el proceso de compilación. Este archivo contiene todos los .m que componen la aplicación tras un proceso de compresión y encriptación, de modo que no es posible acceder al código fuente original. Cuando un aplicación es ejecutada por primera vez, el con-

[8] `http://www.open-mpi.org/`
[9] `http://www.lam-mpi.org/`
[10] `http://www-unix.mcs.anl.gov/mpi/mpich1/`
[11] `http://www.mcs.anl.gov/research/projects/mpich2/`
[12] Matlab es una marca registrada de The Mathworks, Inc

tenido de este archivo es descomprimido y un nuevo directorio es generado.

El proceso que Matlab sigue para generar una aplicación independiente es realizado de un modo automático y totalmente transparente a los usuarios de modo que éstos solo tienen que especificar los archivos .m que componen la aplicación y Matlab se encargará de llevar a cabo el proceso mostrado en la Tabla 6.1 e ilustrado en la Figura 6.9.

Tal y como está listado, hay un paso de generación de código donde se crea una interfaz para el MCR. Estos archivos de interfaz permiten a una arquitectura concreta ejecutar el código de Matlab compilado.

6.2.1. Plataforma de desarrollo y entorno software

El escenario concreto sobre el que se ha trabajado es el siguiente: dada una aplicación implementada en Matlab en un PC, deseamos paralelizarla utilizando MPI y ejecutar el ejecutable generado en una máquina HPC que no posee Matlab instalado. La mayor limitación en esta situación son que no poseemos derechos de administrador y que solo hay una licencia Matlab en la máquina HPC.

La plataforma de desarrollo (DP) es un PC con procesador 2.2 GHz AMD Turion y 1 GB de memoria, que tiene instalado el sistema operativo Linux (Kubuntu 7.10) y Matlab 7.4.0 r2007a (utilizando GCC 4.1.3 para compilar C/C++). La plataforma de ejecución (NESS) es una máquina paralela que es utilizada en proyectos del EPCC así como en el programa HPC-Europa que permite la mobilidad de investigadores europeos. NESS es un cluster producido por Sun con procesadores AMD Opteron ejecutando Linux. El cluster está compuesto por dos procesadores (front-end) que proporcionan acceso al cluster, donde los programas pueden ser compilados, y 32 procesadores como back-end con memoria compartida, donde los programas son ejecutados. El sistema operativo es Scientific Linux (`https://www.scientificlinux.org/`), GNU/Linux 2.6.18, con los compiladores Portland Group C/C++ (PGI, `http://www.pgroup.com/`) y GNU C/C++ (GCC, `http://www.gnu.org/software/gcc/`) y las bibliotecas MPICH2. Para enviar los trabajos al back-end, NESS utiliza el Sun Grid Engine (SGE) como sistema de colas `http://gridengine.sunsource.net/`.

La biblioteca MPICH2 implementa los estándares MPI-1 y MPI-2. En NESS, la biblioteca está disponible para los dos compiladores instalados (PGI y GCC) pero solo para el enlazado estático. Tanto el compilador como la biblioteca pueden ser elegidas por el programador mediante las opciones de configuración en la cuenta local. El SGE permite el envío de programas MPI y OpenMP para ejecución. Cada tarea está definida por un script con opciones para controlar el entorno de ejecución (exportar variables y compartir los directorios de ejecución entre los procesos), el tiempo de ejecución estimado, el entorno paralelo y el comando para ejecutar el programa.

El tipo de procesador, sistema operativo, compilador, biblioteca MPI y sistema

de colas son los elementos más importantes a la hora de adaptar la MPI *toolbox* a NESS dado que estos parámetros son los que limitan la compatibilidad con los ejecutables generados en Matlab en la DP. El sistema de colas debe ser especialmente tenido en cuenta ya que todos los trabajos deben ser enviados a través de él para ejecutarse. Esto podría ocasionar problemas ya que nuestros ejecutables no son ejecutable MPI habituales.

La compatibilidad binaria entre la DP y NESS está garantizada: ambos tienen instalados la versión de 64 bits de Linux con *kernels* y compiladores compatibles. Sin permisos de administración en NESS, la instalación del MCR, necesario para la ejecución del *stand-alone*, debe ser realizada en la cuenta local. Tras unas pequeñas modificaciones de las variables de entorno, el problema se resuelve. En el caso de los programas MPI, un problema adicional se presenta con respecto a los stand-alone: si p procesos se lanzan y el directorio de información (creado a partir del archivo CTF) no está presente, todos los procesos intentarán descomprimir simultáneamente este archivo generando errores en el sistema de archivos. Una solución es invocar el ejecutable con un único proceso que sea el encargado de realizar la descompresión.

6.2.2. *Toolboxes* para MPI

En esta sección se describirán las tres *toolboxes* de MPI que fueron instaladas en NESS. Cada una tiene elementos distintos a las otras y estas diferencias tienen repercusión en la compatibilidad de la biblioteca MPI y el sistema de colas de NESS, requiriendo especial atención a la hora de realizar la instalación.

6.2.3. MatlabMPI en NESS

MatlabMPI[13] fue desarrollada por el Dr. Jeremy Kepner del *Massachusetts Institute of Technology* [Kepner and Ahalt, 2004]. MatlabMPI consiste en un conjunto de funciones escritas en Matlab que simulan un subconjunto de las funciones MPI (ver Tabla 6.2) gracias a uso de un sistema de archivos compartido, por tanto, es independiente de la biblioteca MPI instalada en el sistema. Por otra parte, esta *toolbox* no se ciñe estrictamente al estándar MPI, de modo que, por ejemplo, el envío realizado con MPI_Bcast debe ser recibido con MPI_Recv, en vez de utilizar la misma función en el receptor.

Una de las mayores ventajas de esta propuesta es su simplicidad e independencia de las arquitecturas e implementaciones concretas de la biblioteca MPI. Al estar escrita en Matlab, permite trabajar con todos los tipos de datos disponibles en éste además de permitir la interactividad.

Dentro del contexto en el que nos encontramos, para poder utilizar esta *toolbox* en la maquina NESS es necesario realizar adaptaciones. Dado que NESS posee

[13] `http://www.ll.mit.edu/mission/isr/Matlabmpi/Matlabmpi.html`

un sistema de colas al que únicamente podemos enviar ejecutables, no es posible utilizar MatlabMPI en modo interactivo usando la línea de comandos de Matlab. Para poder crear un ejecutable, será necesario añadir unas cabeceras a los *scripts* de la *toolbox* de modo que el *Matlab Compiler* pueda generar la aplicación. Para poder establecer las comunicaciones será necesario inicializar un directorio al cual puedan acceder las distintas copias de los programas que se ejecutarán, esta tarea es llevada a cabo por el código mostrado a continuación:

```
function Launch_MPI_Matlab_app( n_proc )

if ischar(n_proc)
    n_proc = str2double(n_proc);
end

if exist('./MatMPI', 'dir') ~= 0
    unix('rm -fr ./MatMPI/*');
else
    mkdir('MatMPI');
end

MatMPI_Comm_init(n_proc,{'localhost'});
```

Código 1: Configuración de MatlabMPI para su ejecución

La última dificultad antes de poder ejecutar un programa en NESS que haga uso de MatlabMPI es la asignación de los identificadores de proceso (*rank*) en el comunicador MPI_COMM_WORLD, dado que este se proporciona como un argumento. La solución que se implementó consiste en la ejecución de una aplicación envoltorio que permita la inicialización del directorio donde tendrán lugar las comunicaciones, la descompresión del archivo CTF y, por último, la invocación al programa que utiliza MatlabMPI asignando los identificadores de procesos tal y como se muestra en el código siguiente:

6.2.4. MPIMEX en NESS

MPIMEX es un proyecto que está siendo desarrollado dentro del Departamento de Arquitectura y Tecnología de Computadores con el objetivo de poder utilizar las bibliotecas de paso de mensajes en Matlab [Guillén,][Guillen et al., 2008b]. Al contrario que MatlabMPI, MPIMEX sí que depende de la implementación que se esté utilizado en la arquitectura sobre la que se va a trabajar pero ya ha sido probada para una gran variedad de implementaciones (Sun MPI, MPICH, LAM/MPI, OpenMPI) obteniendo resultados satisfactorios. Esta *toolbox* está compuesta por

```
#include <stdio.h>
#include <mpi.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    char str[256];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    sprintf(str,"%s %d", argv[1], myid);
    system(str);

    MPI_Finalize();
    return 0;
}
```

Código 2: programa envoltorio (en C) para lanzar aplicaciones que usen MatlabMPI

```
#!/bin/bash
#$ -V
#$ -l h_rt=12::
#$ -cwd
#$ -pe mpi 2
mpiexec -np 2 ./wrapper ./my_function
```

Código 3: *Script* usado en el SGE para lanzar una función (my_function) en NESS usando MatlabMPI.

un archivo escrito en C que es enlazado estáticamente con una implementación de MPI concreta. Una vez se realizado este enlazado, se pueden escribir programas (funciones) en Matlab que, tras su posterior compilación, pueden ser ejectuadas en cualquier plataforma.

Un primer nivel de desarrollo consiste en el uso de vectores de tipo *double* o

char para las comunicaciones. A partir de estas primitivas, se ha desarrollado otro nivel que sí permite el uso de tipos de datos complejos de Matlab como matrices de dos dimensiones, estructuras y celdas. El primer nivel está implementado mediante un .ªrchivo mexz proporciona las primitivas mostradas en la Tabla 6.3. La rutinas del segundo nivel están escritas en código Matlab y se muestran en la Tabla 6.4. Las rutinas pueden usarse simultáneamente en un mismo código pero teniendo en cuenta que no se pueden mezclar, es decir, si realizamos un envío con una rutina de nivel 1, no podemos realizar su correspondiente recepción usando una rutina de nivel 2.

Para poder utilizar MPIMEX en NESS, el código original debía enlazarse con la implementación MPICH2 utilizada en NESS. Para ello, se instaló la misma versión de MPICH2 en la máquina DP y se realizó el enlazado satisfactoriamente. Una vez operativa MPIMEX en la máquina DP, la portabilidad a NESS fue inmediata salvo por la copia de distintas librerías dinámicas que no estaban disponibles en NESS.

6.2.5. MPITB en NESS

MPITB es otra *toolbox* disponible para poder utilizar el estándar de paso de mensajes en Matlab aunque su desarrollo ahora está centrado en su adaptación para Octave. Esta *toolbox* consiste en un complejo conjunto de archivos que combinan C y Matlab que permiten utilizar por completo el estándar MPI aprovechando toda la comodidad de Matlab y permitiendo el envío de datos complejos y matrices de éstos. MPITB puede usarse directamente en la línea de comandos o bien a través del compilador (aunque esta última opción no está documentada). El principal problema de MPITB es que solo está disponible para la implementación LAM. Dado que NESS utiliza la implementación MPICH2, fue necesario realizar un procedimiento similar al seguido con MPIMEX. MPITB fue recompilado y enlazado con MPICH2 tras la previa modificación del Makefile.

6.2.6. Resultados Experimentales

La medida de prestaciones (*benchmarking*) para una librería de paso de mensajes utilizada en Matlab es una tarea muy compleja dado que hay que considerar muchos factores tanto hardware como software. El criterio que se utiliza en esta memoria para determinar la bondad de las diferentes *toolboxes* es la latencia añadida en la comunicación en una máquina concreta, en este caso, NESS.

Los experimentos realizados calcularon la latencia introducida por cada una de las 3 *toolboxes* anteriores en las operaciones de comunicación más habituales en la programación paralela: Send/Receive y Bcast (*broadcast*). Dado que la operación de envío en MatlabMPI es no bloqueante, fue necesario añadir una llamada a una función disponible en esta *toolbox* que permite bloquear hasta la recepción del mensaje, igualando el comportamiento con las otras *toolboxes*. Los experimentos

se dividieron en dos grupos según el tipo de comunicación: *punto a punto* y *global*. Dentro de estas dos categorías se hicieron experimentos utilizando distintos tamaños en los mensajes a enviar: *pequeño* y *grande* dando resultado a un total de 12 programas (2 tipos de comunicaciones $\times$ 2 tamaños de mensaje $\times$ 3 *toolboxes*). Para tener una medida relativa con la cual comparar, también se codificaron estos programas usando C.

Dentro de la categorización de los mensajes, cada una de las categorías toma una serie de valores de modo que sea visible la progresión en el aumento de la latencia conforme va incrementando el tamaño del mensaje. Los mensajes considerados como *pequeños* consistieron en vectores de *doubles* con una longitud entre 100 y 5000 elementos, haciendo incrementos en 50 elementos. Los mensajes *grandes* van desde los 1KB hasta los 128 MB duplicando el tamaño en cada experimento. El motivo para estudiar el comportamiento de las *toolboxes* usando únicamente datos de tipo *double* es porque este tipo es el más utilizado en Matlab y otros tipos más complejos se sustentan en éste.

Los resultados correspondientes a las ejecuciones se muestran en las Tablas 6.5, 6.6, 6.7 y 6.8 donde se muestran las medias de las latencias en las distintas comunicaciones. Estos resultados se muestran gráficamente en las Figuras 6.10, 6.11, 6.12 y 6.13. Dado que los tiempos proporcionados por MatlabMPI son considerablemente mayores que los de las otras *toolboxes* se han representado de nuevo las figuras excluyendo estos resultados para poder apreciar diferencias entre las otras propuestas. Para el caso concreto de la comunicación entre 2 procesadores con mensajes *pequeños* en C, no se han incluido en la Tabla 6.5 los valores obtenidos debido a su pequeño valor, sin embargo sí se muestran en la Figura 6.10.

En las Figuras 6.10 y 6.11 se pueden apreciar unos saltos que están asociados a aspectos internos de implementación de la biblioteca MPICH2 tal y como se muestra en [Voorst and Seidel, 2000]. Estos patrones están originados por la saturación de los búferes internos que utiliza la biblioteca y la caché. En la Figura 6.11 aparece un salto considerable para mensajes de más de 1500 *doubles* (p.ej. 12KB) la razón es la misma que en los casos anteriores. Respecto a las prestaciones, los resultados muestran como MatlabMPI es la que tiene un peor comportamiento dado que realiza las comunicaciones a través del sistema de archivos. Para los mensajes *pequeños* la sobrecarga de la lectura/escritura en disco es especialmente grande proporcionando gráficas irregulares (Figuras 6.10 y 6.11) probablemente relacionadas con los búferes que, a nivel interno, posee el sistema operativo.

Las otras dos *toolboxes* tienen un comportamiento muy similar entre ellas, en el caso de los mensajes *pequeños* no se muestra ninguna influencia sobre la latencia en la comunicación. ésta es bastante similar aunque se puede apreciar cierta ventaja de MPIMEX sobre MPITB. En el caso de los mensajes *grandes*, la latencia introducida por MPIMEX sí que es notablemente menor que la que añade MPITB.

6.2.7. Recomendaciones para las toolboxes

Tras mostrar los resultados empíricos y haber comentado el proceso de instalación y adaptación a una arquitectura concreta, las recomendaciones que se pueden realizar son las siguientes:

Prestaciones De cara a obtener mejores prestaciones gracias a la insignificante adición de latencia en las comunicaciones, MPIMEX y MPITB respectivamente, son las dos mejores alternativas. MPIMEX tiene la restricción de no soportar tipos de datos complejos, al contrario que MPITB que sí que permite el envío de matrices de 1 y 2 dimensiones. En este aspecto MatlabMPI es la mejor dado que permite el envío de datos compuestos en Matlab. MPIMX y MPITB tienen una sintaxis lo más cercana posible al estándar MPI, haciendo trivial el cambio a Matlab a programadores con experiencia en programación con MPI en C.

Compatibilidad con la biblioteca MPI instalada De las tres alternativas, MPIMEX es la mejor si la biblioteca MPI instalada en el sistema permite el enlazado estático. MPITB es más sencilla de adaptar en el caso de enlazado dinámico si bien el soporte para las distintas implementaciones de MPI no es tan amplio como el proporcionado por MPIMEX. MatlabMPI no es dependiente de la biblioteca MPI instalada en el sistema, sin embargo requiere modificaciones importantes para poder funcionar en máquinas con la arquitectura descrita en las secciones anteriores.

Uso de sistema de colas Las dos mejores alternativas de cara a lanzar trabajos en una cola como, por ejemplo SGE, son MPIMEX y MPITB puesto que no requieren de modificaciones para ejecutar los programas.

Adaptación de la toolbox y documentación MPIMEX no se distribuye directamente, para poder instalarla es necesario contactar con los desarrolladores, sin embargo, éstos proporcionan soporte para la instalación, adaptándola para la arquitectura concreta donde se vaya a utilizar. Si se desea modificar o colaborar en el desarrollo, está implementada en pocas líneas de código haciendo su modificación bastante simple. Los archivos fuentes de MPITB están también disponibles así como documentación y ejemplos, sin embargo, la complejidad de la implementación es considerable y el proyecto está congelado (si bien el desarrollo está siendo continuado para Octave). MatlabMPI es la más asequible de modificar puesto que está enteramente escrita en Matlab, no necesita bibliotecas externas y el número de líneas de código es reducido.

Tabla 6.1: Operaciones realizadas por Matlab para compilar y generar una aplicación independiente (*stand-alone*).

Paso	**Operación**
1	Análisis de dependencias entre los archivos .m .
2	Generación de código: la interfaz C o C++ es generada en este paso.
3	Creación de archivo: una vez resueltas las dependencias, los archivos .m son comprimidos y encriptados en un archivo CTF.
4	Compilación: el código fuente de las interfaces es compilado.
5	Enlazado: el código objeto es enlazado con las bibliotecas exigidas por Matlab.

Tabla 6.2: Funciones MPI en MatlabMPI (ver [Kepner and Ahalt, 2004] para más detalles)

`MPI_Init`	Inicializa MPI.
`MPI_Comm`	Devuelve el número de procesos dentro de un comunicador.
`MPI_Comm`	Devuelve el identificador de un proceso dentro de un comunicador.
`MPI_Send`	Envía un mensaje a un proceso (no-bloqueante).
`MPI_Recv`	Recibe un mensaje de un proceso (bloqueante).
`MPI_Finalize`	Finaliza MPI.
`MPI_Abort`	Función para terminar todos los trabajos creados con MatlabMPI.
`MPI_Bcast`	Difunde un mensaje (bloqueante).
`MPI_Probe`	Devuelve una lista con todos los mensajes para recibir.

Tabla 6.3: Funciones MPIMEX de nivel 1

`MPI_Init`	Inicializa MPI.
`MPI_Finalize`	Finaliza MPI.
`MPI_Send`	Envía un vector fila a un proceso (bloqueante).
`MPI_Recv`	Recibe un vector fila desde un proceso (bloqueante).
`MPI_Bcast`	Difunde (*broadcast*) un vector fila en un grupo de procesos (bloqueante).
`MPI_Scatter`	Realiza una difusión dividiendo los datos a enviar dentro de un grupo de procesos (bloqueante).
`MPI_Gather`	Reune unificando en un proceso un conjunto de datos provenientes de un conjunto de procesos (bloqueante).
`MPI_Barrier`	Bloquea a los procesos hasta que todos lleguen a este punto (bloqueante).
`MPI_Comm_create`	Crea un nuevo comunicador (bloqueante).
`MPI_Comm_group`	Crea un grupo dentro del nuevo comunicador (bloqueante).
`MPI_Probe`	Devuelve una lista de todos los mensajes para recibir (bloqueante).
`MPI_ISend`	Envía un vector fila a un proceso (no-bloqueante).
`MPI_IRecv`	Recibe un vector fila de un proceso (no-bloqueante).
`MPI_Test`	Comprueba el estado de un envio/recepción no bloqueante (no-bloqueante).

Tabla 6.4: Funciones MPIMEX de nivel 2

`InitMPIMEX`	Inicializa MPI y descomprime el archivo CTF si se ejecuta un proceso.
`FinalizeMPIMEX`	Finaliza MPI.
`SendVariable`	Envía una variable a un proceso (bloqueante).
`ReceiveVariable`	Recibe una variable de un proceso (bloqueante).
`BcastVariable`	Difunde (*broadcast*) una variable en un grupo de procesos (bloqueante).

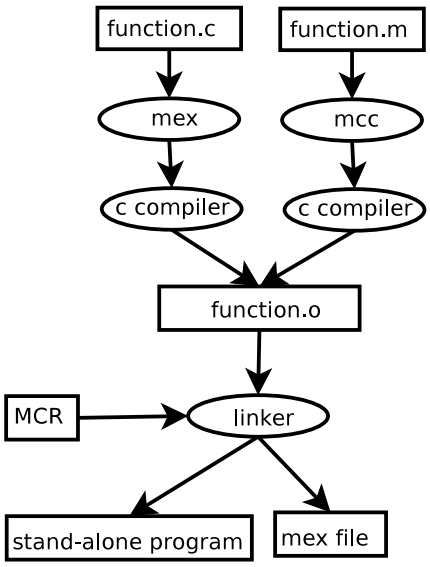
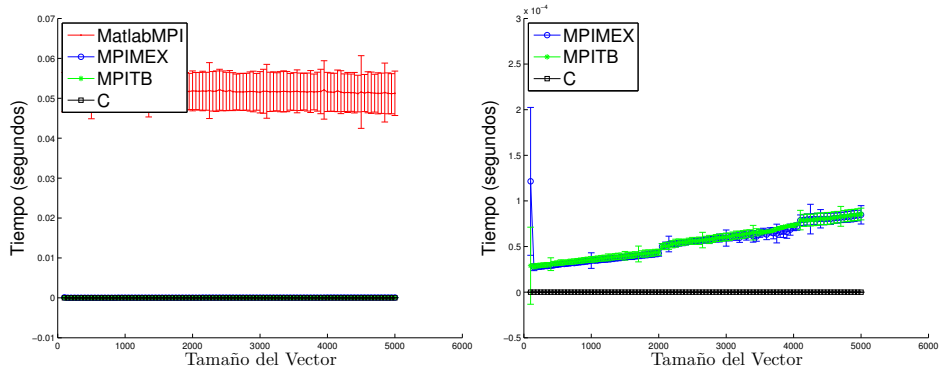
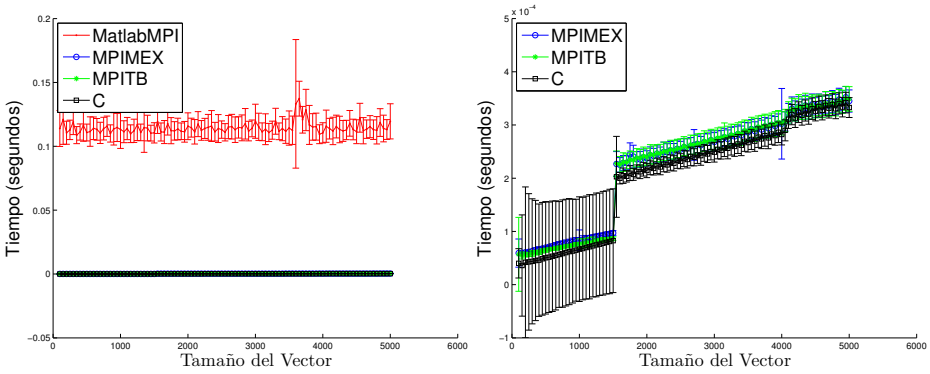


Fig. 6.9: Tiempo de generación de una aplicación y un archivo mex usando Matlab *Compiler*.



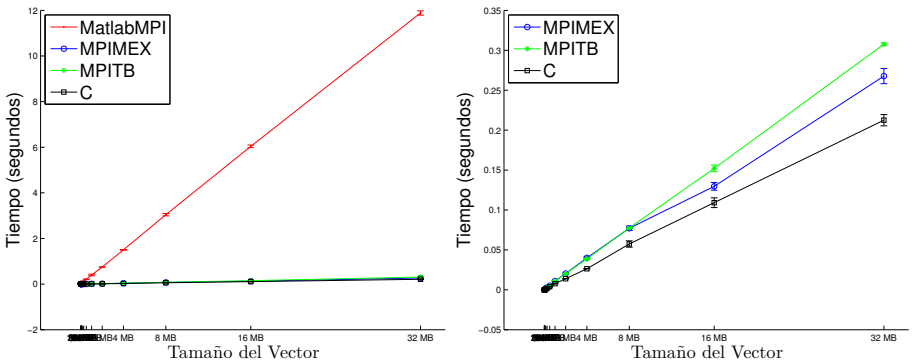
(a) MPI *toolboxes* y C (b) MPI *toolboxes* menos MatlabMPI

Fig. 6.10: Media de los tiempos de recepción de vectores de *double* usando *Send/Recv* entre 2 procesadores vs número de elementos en cada vector



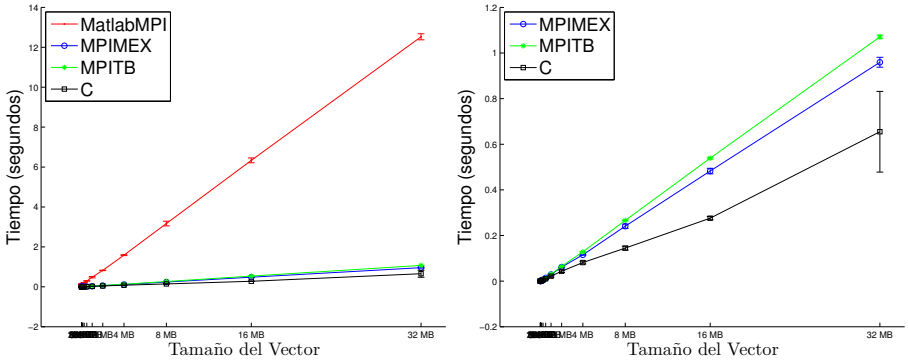
(a) MPI *toolboxes* y C (b) MPI *toolboxes* menos MatlabMPI y C

Fig. 6.11: Media de los tiempos de recepción de vectores de *double* usando *Bcast* entre 8 procesadores vs número de elementos en cada vector



(a) MPI *toolboxes* y C (b) MPI *toolboxes* menos MatlabMPI y C

Fig. 6.12: Media de los tiempos de recepción de vectores de *double* usando *Send/Recv* entre 2 procesadores vs tamaño del vector



(a) MPI *toolboxes* y C (b) MPI *toolboxes* menos MatlabMPI y C

Fig. 6.13: Media de los tiempos de recepción de vectores de *double* usando *Bcast* en 8 procesadores vs el tamaño del vector

Tabla 6.5: Media de los tiempos de recepción (en segundos) de vectores de *double* usando *Send/Recv* entre 2 procesos con las *toolboxes* (mejores resultados en negrita, *Size* es el número de elementos del vector)

Size	MatlabMPI	MPIMEX	MPITB
100	5.15e-02 ± 4.92e-03	1.21e-04 ± 8.10e-05	**2.90e-05 ± 4.22e-05**
250	5.16e-02 ± 5.16e-03	**2.78e-05 ± 2.35e-06**	2.94e-05 ± 2.41e-06
400	5.17e-02 ± 5.11e-03	**2.88e-05 ± 2.36e-06**	3.07e-05 ± 7.00e-06
550	5.16e-02 ± 4.43e-03	**3.06e-05 ± 2.48e-06**	3.24e-05 ± 2.98e-06
700	5.16e-02 ± 4.57e-03	**3.19e-05 ± 2.75e-06**	3.33e-05 ± 2.81e-06
850	5.34e-02 ± 6.36e-03	**3.31e-05 ± 2.89e-06**	3.46e-05 ± 2.80e-06
1000	5.17e-02 ± 5.03e-03	**3.46e-05 ± 8.56e-06**	3.59e-05 ± 3.00e-06
1150	5.17e-02 ± 4.67e-03	**3.56e-05 ± 2.91e-06**	3.71e-05 ± 3.16e-06
1300	5.17e-02 ± 4.88e-03	**3.66e-05 ± 3.02e-06**	3.83e-05 ± 3.18e-06
1450	5.18e-02 ± 4.98e-03	**3.80e-05 ± 3.09e-06**	3.95e-05 ± 3.31e-06
1600	5.16e-02 ± 4.55e-03	**3.89e-05 ± 3.10e-06**	4.05e-05 ± 3.37e-06
1750	5.17e-02 ± 4.28e-03	**4.05e-05 ± 3.15e-06**	4.19e-05 ± 3.52e-06
1900	5.18e-02 ± 4.69e-03	**4.16e-05 ± 3.34e-06**	4.28e-05 ± 3.46e-06
2050	5.18e-02 ± 4.65e-03	**5.01e-05 ± 3.09e-06**	5.02e-05 ± 3.12e-06
2200	5.18e-02 ± 4.76e-03	5.29e-05 ± 3.09e-06	**5.25e-05 ± 4.00e-06**
2350	5.18e-02 ± 5.07e-03	**5.41e-05 ± 3.18e-06**	5.46e-05 ± 3.32e-06
2500	5.17e-02 ± 4.82e-03	**5.57e-05 ± 2.59e-06**	5.62e-05 ± 3.85e-06
2650	5.16e-02 ± 4.48e-03	**5.68e-05 ± 2.67e-06**	5.71e-05 ± 8.49e-06
2800	5.16e-02 ± 4.62e-03	**5.82e-05 ± 3.25e-06**	5.83e-05 ± 3.76e-06
2950	5.16e-02 ± 4.61e-03	**5.94e-05 ± 3.10e-06**	6.01e-05 ± 4.07e-06
3100	5.18e-02 ± 6.75e-03	**6.10e-05 ± 3.48e-06**	6.19e-05 ± 4.01e-06
3250	5.16e-02 ± 4.77e-03	**6.29e-05 ± 3.72e-06**	6.33e-05 ± 4.21e-06
3400	5.17e-02 ± 5.03e-03	**6.11e-05 ± 6.32e-06**	6.52e-05 ± 8.00e-06
3550	5.17e-02 ± 5.38e-03	6.57e-05 ± 4.26e-06	**6.57e-05 ± 1.49e-06**
3700	5.16e-02 ± 4.76e-03	**6.47e-05 ± 6.20e-06**	6.78e-05 ± 1.37e-06
3850	5.17e-02 ± 4.88e-03	**6.62e-05 ± 6.36e-06**	7.12e-05 ± 1.57e-06
4000	5.16e-02 ± 4.90e-03	**7.12e-05 ± 3.81e-06**	7.32e-05 ± 2.19e-06
4150	5.19e-02 ± 5.21e-03	7.92e-05 ± 5.96e-06	**7.86e-05 ± 6.61e-06**
4300	5.15e-02 ± 5.02e-03	8.02e-05 ± 5.94e-06	**7.92e-05 ± 6.36e-06**
4450	5.12e-02 ± 4.59e-03	**8.01e-05 ± 6.28e-06**	8.04e-05 ± 6.44e-06
4600	5.15e-02 ± 5.14e-03	**8.11e-05 ± 5.76e-06**	8.18e-05 ± 6.31e-06
4750	5.12e-02 ± 5.14e-03	**8.21e-05 ± 5.70e-06**	8.33e-05 ± 6.26e-06
4900	5.12e-02 ± 5.12e-03	**8.34e-05 ± 6.04e-06**	8.46e-05 ± 6.16e-06

Tabla 6.6: Media de los tiempos de recepción (en segundos) de vectores de *double* usando *Bcast* entre 8 procesos con las *toolboxes* y un programa en C (mejores resultados en negrita, *Size* es el número de elementos del vector)

Size	MatlabMPI	MPIMEX	MPITB	C
100	1.14e-01 ± 1.41e-02	5.93e-05 ± 2.65e-05	**5.69e-05 ± 6.95e-05**	4.00e-05 ± 2.75e-05
250	1.13e-01 ± 7.61e-03	6.17e-05 ± 1.71e-06	**5.69e-05 ± 3.92e-06**	4.28e-05 ± 1.28e-04
400	1.11e-01 ± 6.33e-03	6.56e-05 ± 5.77e-06	**6.08e-05 ± 4.09e-06**	4.60e-05 ± 1.06e-04
550	1.12e-01 ± 1.15e-02	7.10e-05 ± 2.29e-06	**6.61e-05 ± 2.93e-06**	5.15e-05 ± 1.04e-04
700	1.10e-01 ± 8.33e-03	7.45e-05 ± 6.50e-06	**6.92e-05 ± 3.32e-06**	5.66e-05 ± 9.94e-05
850	1.10e-01 ± 6.74e-03	7.85e-05 ± 6.94e-06	**7.31e-05 ± 2.71e-06**	6.20e-05 ± 9.97e-05
1000	1.13e-01 ± 1.49e-02	8.36e-05 ± 1.92e-05	**7.67e-05 ± 9.33e-06**	6.67e-05 ± 9.83e-05
1150	1.10e-01 ± 6.96e-03	8.79e-05 ± 4.11e-06	**8.00e-05 ± 8.40e-06**	7.12e-05 ± 9.79e-05
1300	1.11e-01 ± 6.49e-03	9.20e-05 ± 4.12e-06	**8.32e-05 ± 3.85e-06**	7.59e-05 ± 9.61e-05
1450	1.10e-01 ± 6.16e-03	9.57e-05 ± 4.19e-06	**8.53e-05 ± 3.96e-06**	8.11e-05 ± 9.81e-05
1600	1.11e-01 ± 7.08e-03	**2.25e-04 ± 1.35e-05**	2.26e-04 ± 1.48e-05	2.00e-04 ± 1.08e-05
1750	1.12e-01 ± 1.02e-02	2.45e-04 ± 2.12e-05	**2.35e-04 ± 1.66e-05**	2.09e-04 ± 1.39e-05
1900	1.12e-01 ± 6.52e-03	**2.35e-04 ± 1.49e-05**	2.38e-04 ± 1.61e-05	2.12e-04 ± 1.29e-05
2050	1.15e-01 ± 7.38e-03	**2.43e-04 ± 1.57e-05**	2.45e-04 ± 1.81e-05	2.20e-04 ± 1.36e-05
2200	1.12e-01 ± 7.71e-03	**2.44e-04 ± 1.48e-05**	2.46e-04 ± 1.75e-05	2.21e-04 ± 1.46e-05
2350	1.16e-01 ± 1.21e-02	**2.52e-04 ± 1.68e-05**	2.54e-04 ± 1.74e-05	2.29e-04 ± 1.45e-05
2500	1.13e-01 ± 6.83e-03	**2.54e-04 ± 1.54e-05**	2.56e-04 ± 1.82e-05	2.32e-04 ± 1.42e-05
2650	1.15e-01 ± 9.64e-03	**2.61e-04 ± 1.58e-05**	2.63e-04 ± 1.80e-05	2.41e-04 ± 1.74e-05
2800	1.14e-01 ± 1.01e-02	**2.62e-04 ± 1.64e-05**	2.65e-04 ± 1.92e-05	2.42e-04 ± 1.80e-05
2950	1.14e-01 ± 6.57e-03	**2.71e-04 ± 1.79e-05**	2.73e-04 ± 1.91e-05	2.50e-04 ± 1.58e-05
3100	1.11e-01 ± 6.97e-03	**2.72e-04 ± 1.76e-05**	2.76e-04 ± 2.03e-05	2.52e-04 ± 1.63e-05
3250	1.13e-01 ± 7.06e-03	**2.80e-04 ± 1.60e-05**	2.82e-04 ± 1.99e-05	2.61e-04 ± 1.76e-05
3400	1.14e-01 ± 9.44e-03	**2.81e-04 ± 1.75e-05**	2.84e-04 ± 2.22e-05	2.62e-04 ± 1.97e-05
3550	1.13e-01 ± 1.11e-02	**2.92e-04 ± 1.79e-05**	2.93e-04 ± 2.11e-05	2.73e-04 ± 1.89e-05
3700	1.21e-01 ± 1.05e-02	**2.92e-04 ± 1.75e-05**	2.96e-04 ± 2.20e-05	2.74e-04 ± 1.83e-05
3850	1.15e-01 ± 1.04e-02	**3.00e-04 ± 1.71e-05**	3.02e-04 ± 2.07e-05	2.83e-04 ± 1.93e-05
4000	1.11e-01 ± 7.92e-03	**3.02e-04 ± 6.63e-05**	3.03e-04 ± 2.19e-05	2.82e-04 ± 1.95e-05
4150	1.12e-01 ± 1.08e-02	3.31e-04 ± 2.18e-05	**3.30e-04 ± 2.06e-05**	3.20e-04 ± 1.91e-05
4300	1.12e-01 ± 8.19e-03	**3.30e-04 ± 1.90e-05**	3.32e-04 ± 2.16e-05	3.18e-04 ± 1.85e-05
4450	1.13e-01 ± 8.07e-03	3.39e-04 ± 1.96e-05	**3.37e-04 ± 1.99e-05**	3.31e-04 ± 2.02e-05
4600	1.12e-01 ± 1.03e-02	**3.34e-04 ± 2.00e-05**	3.34e-04 ± 2.04e-05	3.28e-04 ± 1.80e-05
4750	1.13e-01 ± 7.63e-03	3.44e-04 ± 2.08e-05	**3.42e-04 ± 2.05e-05**	3.38e-04 ± 2.03e-05
4900	1.14e-01 ± 1.04e-02	**3.44e-04 ± 2.07e-05**	3.44e-04 ± 2.17e-05	3.34e-04 ± 1.84e-05

Tabla 6.7: Media de los tiempos de recepción (en segundos) de vectores de *double* usando *Send/Recv* entre 2 procesos con las *toolboxes* y un programa en C (mejores resultados en negrita)

Size	MatlabMPI	MPIMEX	MPITB	C
1 KB	5.24e-02 ± 4.61e-03	**3.37e-05 ± 5.19e-05**	5.52e-05 ± 2.96e-04	3.90e-06 ± 6.31e-06
2 KB	5.35e-02 ± 2.27e-03	2.91e-05 ± 2.44e-06	**2.55e-05 ± 4.47e-06**	7.82e-06 ± 2.68e-05
4 KB	5.44e-02 ± 7.00e-03	3.16e-05 ± 2.22e-06	**3.04e-05 ± 2.89e-06**	7.19e-06 ± 1.48e-05
8 KB	5.33e-02 ± 2.95e-03	**3.60e-05 ± 2.38e-06**	3.65e-05 ± 2.40e-06	1.62e-05 ± 4.67e-05
16 KB	5.36e-02 ± 3.45e-03	**4.93e-05 ± 3.98e-06**	5.17e-05 ± 2.21e-06	3.18e-05 ± 6.60e-05
32 KB	5.44e-02 ± 7.19e-03	**8.10e-05 ± 5.92e-06**	8.18e-05 ± 7.34e-06	6.56e-05 ± 9.30e-05
64 KB	5.37e-02 ± 1.94e-03	1.41e-04 ± 5.45e-06	**1.40e-04 ± 6.07e-06**	1.19e-04 ± 1.10e-04
128 KB	5.45e-02 ± 4.98e-03	**2.60e-04 ± 5.46e-06**	2.60e-04 ± 6.88e-06	2.35e-04 ± 1.29e-04
256 KB	5.32e-02 ± 8.57e-03	6.40e-04 ± 3.56e-04	**6.28e-04 ± 2.37e-04**	2.91e-04 ± 1.76e-04
512 KB	8.27e-02 ± 8.52e-03	1.49e-03 ± 2.46e-04	**1.17e-03 ± 4.06e-05**	9.99e-04 ± 8.65e-05
1 MB	1.24e-01 ± 3.71e-03	2.65e-03 ± 6.94e-05	**2.62e-03 ± 6.53e-05**	1.94e-03 ± 6.23e-05
2 MB	2.13e-01 ± 4.79e-03	5.18e-03 ± 1.37e-04	**5.15e-03 ± 1.40e-04**	3.78e-03 ± 1.42e-04
4 MB	4.01e-01 ± 3.43e-02	1.10e-02 ± 8.21e-04	**1.09e-02 ± 7.46e-04**	7.74e-03 ± 4.30e-04
8 MB	7.46e-01 ± 6.45e-03	2.04e-02 ± 6.69e-04	**2.02e-02 ± 8.87e-05**	1.38e-02 ± 1.45e-04
16 MB	1.50e+00 ± 1.42e-02	4.00e-02 ± 1.28e-03	**3.87e-02 ± 5.91e-04**	2.63e-02 ± 2.34e-04
32 MB	3.04e+00 ± 5.15e-02	**7.71e-02 ± 2.69e-03**	7.74e-02 ± 1.06e-04	5.74e-02 ± 3.96e-03
64 MB	6.04e+00 ± 5.76e-02	**1.30e-01 ± 4.85e-03**	1.52e-01 ± 4.23e-03	1.09e-01 ± 6.12e-03
128 MB	1.19e+01 ± 9.56e-02	**2.68e-01 ± 9.46e-03**	3.08e-01 ± 1.74e-03	2.12e-01 ± 7.06e-03

Tabla 6.8: Media de los tiempos de recepción (en segundos) de vectores de *double* usando *Bcast* entre 8 procesos con las *toolboxes* y un programa en C (mejores resultados en negrita)

Size	MatlabMPI	MPIMEX	MPITB	C
1 KB	1.15e-01 ± 1.12e-02	**7.76e-05 ± 2.82e-04**	1.29e-04 ± 7.05e-04	4.90e-05 ± 8.50e-04
2 KB	1.13e-01 ± 7.42e-03	**5.04e-05 ± 7.99e-05**	6.08e-05 ± 8.15e-06	1.00e-04 ± 9.95e-04
4 KB	1.20e-01 ± 1.55e-02	**5.95e-05 ± 1.09e-04**	6.88e-05 ± 7.16e-06	1.00e-04 ± 1.05e-03
8 KB	1.15e-01 ± 1.33e-02	**7.22e-05 ± 1.06e-04**	8.24e-05 ± 7.17e-06	1.40e-04 ± 1.20e-03
16 KB	1.12e-01 ± 8.67e-03	3.64e-04 ± 3.81e-04	**2.53e-04 ± 1.10e-04**	2.00e-04 ± 1.40e-03
32 KB	1.15e-01 ± 1.01e-02	4.04e-04 ± 5.01e-05	**3.23e-04 ± 2.02e-05**	2.00e-04 ± 1.40e-03
64 KB	1.14e-01 ± 7.79e-03	6.18e-04 ± 1.51e-04	**4.60e-04 ± 2.26e-05**	5.00e-04 ± 2.18e-03
128 KB	1.16e-01 ± 1.11e-02	1.10e-03 ± 1.46e-04	**7.69e-04 ± 4.94e-05**	6.00e-04 ± 2.37e-03
256 KB	1.35e-01 ± 7.44e-03	2.32e-03 ± 2.36e-04	**1.41e-03 ± 6.13e-05**	1.30e-03 ± 3.36e-03
512 KB	1.67e-01 ± 1.18e-02	3.52e-03 ± 3.54e-04	**3.30e-03 ± 8.09e-05**	2.50e-03 ± 4.33e-03
1 MB	2.08e-01 ± 9.12e-03	**6.39e-03 ± 8.81e-05**	6.66e-03 ± 8.86e-05	4.90e-03 ± 5.20e-03
2 MB	2.93e-01 ± 1.27e-02	**1.42e-02 ± 4.15e-04**	1.45e-02 ± 1.85e-04	1.01e-02 ± 3.32e-03
4 MB	4.80e-01 ± 3.43e-02	**2.92e-02 ± 5.28e-04**	3.08e-02 ± 3.11e-04	2.18e-02 ± 4.33e-03
8 MB	8.22e-01 ± 1.22e-02	**6.14e-02 ± 2.11e-03**	6.44e-02 ± 4.58e-04	4.32e-02 ± 5.27e-03
16 MB	1.59e+00 ± 2.20e-02	**1.15e-01 ± 2.16e-03**	1.28e-01 ± 8.32e-04	8.13e-02 ± 5.94e-03
32 MB	3.17e+00 ± 1.18e-01	**2.41e-01 ± 9.21e-03**	2.66e-01 ± 2.41e-03	1.45e-01 ± 8.30e-03
64 MB	6.33e+00 ± 1.22e-01	**4.83e-01 ± 1.21e-02**	5.39e-01 ± 4.04e-03	2.76e-01 ± 7.07e-03
128 MB	1.25e+01 ± 1.52e-01	**9.59e-01 ± 2.17e-02**	1.07e+00 ± 8.09e-03	6.55e-01 ± 1.77e-01

7. CONCLUSIONES Y PRINCIPALES APORTACIONES

A raíz del trabajo desarrollado en esta memoria, se pueden extraer una serie de conclusiones que serán comentadas y enumeradas en este último capítulo. Las primeras conclusiones surgen gracias al extenso análisis bibliográfico referente a los métodos *kernel* para regresión. Éstas muestran como estos métodos *kernels* han sido utilizados de manera equivalente a una RBFNN con tantos centros como puntos de entrada del conjunto de entrenamiento. Dicho uso obtiene buenos resultados de aproximación pero traslada la mayor parte de la dificultad del problema a la selección de características o, en el caso de aproximación de series temporales, de los regresores. Sin embargo, ésto también conlleva que normalmente no sean tratados con detalle en la bibliografía consultada dos elementos claves del entrenamiento de éstos modelos:

1. El algoritmo de optimización de parametros no lineales de los modelos (generalmente, los parámetros de la función de *kernel*).

2. La elección de su principal elemento diferenciador, es decir, la función *kernel* ya que ésta puede ser cualquiera que cumpla el teorema de Mercer.

Sin embargo, en esta memoria se ha demostrado cómo se pueden obtener mejores modelos con una adecuado método de optimización de los parámetros y con el uso de *kernels* específicos. Respecto al diseño de los *kernels*, se ha propuesto una metodología basada en el conocimiento proporcionado por el experto y también se ha propuesto un método automático basado en Programación Genética. Las conclusiones de este estudio han mostrado cómo la tarea del diseño del *kernel* con Programación Genética es demasiado costosa computacionalmente aún teniendo en cuenta que el método de evaluación de los *kernels* ha sido optimizado. Esta optimización de la función de evaluación es otra aportación del trabajo presentado en esta memoria puesto que implica la adaptación del test Delta al espacio de características definido por los *kernels*.

Otra gran novedad expuesta en esta memoria es una variante del algoritmo de los K vecinos más cercanos aplicado a problemas de regresión: el KWKNN,

que ha demostrado ser un algoritmo competitivo con LSSVM, y con menor coste computacional. Adicionalmente, se han implementado versiones de dicho algoritmo optimizadas para dos arquitecturas paralelas distintas: *clusters* de computadores y GPU (Graphic Processing Unit) de la empresa NVIDIA.

Finalmente, también se ha demostrado cómo se puede aprovechar arquitecturas anteriormente mencionadas desde Matlab con dos *toolboxes* que se han implementado con dicho fín: MPIMEX y CUBLASMEX.

A continuación se enumeran detalladamente las principales conclusiones y contribuciones científico-técnicas descritas en la presente memoria y desarrolladas durante la realización de esta Tesis Doctoral:

1. Se ha realizado un estudio sobre la elección de parámetros óptimos para modelos LSSVM a partir de un conjunto de ejemplos de entrenamiento para problemas de regresión. A partir de una intensa labor de revisión bibliográfica, se han propuesto dos heurísticas para inicialización de los parámetros de un modelo LSSVM para regresión:

   a) Para el parámetro γ (ver apartado 3.2), común a todos los modelos LSSVM independientemente del *kernel*, se presentó y justificó una heurística de inicialización basada en estimadores no paramétricos del ruido en los datos de un problema: los tests Delta y Gamma.

   b) Aún siendo conscientes de la imposibilidad de dar una heurística común para la inicialización de los parámetros concretos para cada posible *kernel*, dado que cada función requiere un conjunto de parámetros diferente, sí que se ha propuesto una para el *kernel* Gaussiano. Esta aportación es valiosa teniendo en cuenta que este tipo de *kernel* es el más utilizado en los modelos LSSVM para problemas de aproximación funcional.

2. Además de estas dos heurísticas, se describió cómo optimizar de manera más eficiente los modelos LSSVM para regresión con respecto al error de validación cruzada. Esto se consiguió calculando las derivadas parciales del método de evaluación de coste reducido de [An et al., 2007] (ver 3.2.1), lo que hace posible aplicar algoritmos de optimización como el Gradiente Conjugado. La relevancia de los resultados ha quedado avalada con la publicación de un artículo en el congreso ICANN 2009 [Rubio et al., 2009c] y otro en un número especial de la revista *International Journal of Forecasting* [Rubio et al., 2010b].

3. Se ha propuesto un nuevo método *kernel* basado en los K vecinos más cercanos ponderados, que denominamos KWKNN (ver Capítulo 4). Éste tiene la ventaja de tener un menor coste computacional que las alternativas clásicas (SVR,LSSVM,Procesos Gaussianos) tanto en el entrenamiento como en su

aplicación. Este nuevo método, eficiente de por sí, se ha implementado en dos plataformas de cómputo de alto rendimiento distintas:

a) *Cluster* de computadoras: en una implementación denominada PKWKNN específicamente creada para poder manejar conjuntos de datos cuyo tamaño impidan la aplicación de los demás métodos *kernel* (ver apartado 4.7).

b) Procesador gráfico: implementación que acelera espectacularmente el procesamiento con conjuntos de datos de tamaño medio (ver apartado 4.9).

El algoritmo se ha comparado con otros modelos en los diversos experimentos llevados a cabo en esta memoria (ver apartado 5.2.2), obteniendo resultados que demuestran su competitividad. El método KWKNN y PKWKNN fue presentado en el artículo del congreso ESTSP 2008 [Rubio et al., 2008], siendo premiado por la ENNS (*European Neural Network Society*) como mejor artículo del congreso, seleccionado para ampliación en número especial de Neurocomputing (editorial Elsevier) y cuyo trabajo resultante ganó la competición de series temporales del congreso. La versión ampliada del anterior trabajo ha sido publicada [Rubio et al., 2010a]. Aparte, una descripción de PKWKNN desde el punto de vista técnico fue presentada en el congreso HPCS 2009 [Rubio et al., 2009a].

4. Se abordó el, hasta la fecha no considerado, problema de creación de *kernels* específicos a los datos para modelos de regresión. Esta tarea se enfocó al importante caso particular de los problemas de predicción de series temporales. Se obtuvo una metodología de diseño de *kernels* basada en conocimiento experto (ver apartado 5.2) que se aplicó con éxito para predicción a largo plazo de series temporales, como puede verse en los ya citados artículos [Rubio et al., 2008] y [Rubio et al., 2010a]. También se diseñó un algoritmo de Programación Genética (ver apartado 5.3) para intentar obtener *kernels* adaptados a un problema en particular del cual no se tenga previo conocimiento, presentado en el congreso IWANN 2009 [Rubio et al., 2009b]. La mayor novedad del citado algoritmo con respecto a la literatura es la evaluación de los *kernels* de forma independiente de un modelo *kernel* concreto mediante una versión modificada del test Delta (ver apartado 5.3.2).

5. Se han desarrollado dos *toolboxes* de Matlab que permiten utilizar las plataformas de cómputo de alto rendimiento *Cluster* de computadoras y procesador gráficos de las tarjetas gráficas NVIDIA. La primera es MPIMEX, que da acceso a la librería MPI para programación paralela, y cuyo rendimiento con respecto a otras alternativas queda patente en el estudio presentado en el apartado 6.2. La segunda *toolbox* es CUBLASMEX, que da acceso desde Matlab a CUBLAS (una implementación de BLAS en GPU de NVIDIA).

7.1. Conclusions

There are several main conclusions that can be extracted the work developed and presented in this report. The first conclusion is based on literature review about kernel methods for regression. This shows as kernel methods have been used as RBFNN with a radial basis centered in every input vector of the sample set. Good results are achieved but the fact is that the main problem to be tackled is the well-known feature selection problem; more specifically, in the case of time series prediction, the selection of the most important regressors. Nevertheless, two fundamental issues associated to kernel methods are neglected:

1. The optimization algorithm for the non-linear parameters of the models (mainly, the parameters of the kernel functions).

2. The main differentiator of kernel methods: the possibility of using an arbitrary kernel function as long as it follows the Mercer's Theorem.

In this memory it has been demonstrated how better models can be obtained with an adequate method for the optimization of the parameters and the use of specific-to-problem kernels. Concerning the design of kernels, it is proposed a methodology based on the knowledge provided by the expert as well as an automatic method based on Genetic Programming. The studies have shown that the task designing a kernel with Genetic Programming is too expensive computationally even taking into account that the evaluation method for the kernels has been optimized. This optimization of the evaluation function is another contribution presented in this report since it involves the definition of the Delta test on the feature space defined by kernels.

Another main novelty presented in this memory is a variant of the K nearest neighbour for regression, KWKNN, which has proved to be competitive even comparing with LSSVM, and with lower computational costs. Moreover, it has been created implementations of KWKNN for two different parallel architectures: a compute cluster platform version of KWKNN and a NVIDIA GPU (Graphic Processing Unit) version, which makes use of the CUDA library.

Finalmente, también se ha demostrado cómo se puede aprovechar arquitecturas anteriormente mencionadas desde Matlab con dos *toolboxes* que se han implementado con dicho fín: MPIMEX y CUBLASMEX.

Finally, it has also shown how take advantage from Matlab of the aforementioned two HPC architectures with two toolboxes (MPIMEX and CUBLASMEX) that have been implemented for this purpose.

The main conclusions as well as scientific and technical contributions developed during the course of this thesis, and finally presented in this report are described schematically below in more detail:

1. The choice of optimal parameters for LSSVM models from a training set of examples in regression problems has been studied. After an intensive review of the literature on the subject, two heuristics for initialization of the parameters of a LSSVM model for regression were presented:

   a) For the parameter γ (see section 3.2), common to all LSSVM models regardless of kernel, a heuristic for its initialization based on the Delta and Gamma tests (which are non parametric noise estimators [Lendasse et al., 2006]) is presented and justified.

   b) Even being aware of the impossibility of giving a common heuristic for the initialization of the specific parameters for each possible kernel, since each function requires a different set of parameters, it has been proposed one for the kernel Gaussian. This contribution is valuable given that this kernel is the most widely used in LSSVM models for functional approximation problems.

2. Additionally from these two heuristics, it was described how to optimize more efficiently LSSVM models for regression with respect to cross-validation error. This was achieved computing the partial derivatives of the low cost evaluation method of cross-validation error of [An et al., 2007] (see Section 3.2.1), which makes possible to apply optimization algorithms such as Conjugate Gradient. The relevance of the results has been endorsed by the publication of a paper in the ICANN 2009 conference [Rubio et al., 2009c] and other in a special issue of the *International Journal of Forecasting* [Rubio et al., 2010b].

3. A new method kernel, KWKNN, based on the weighted K nearest neighbors, has been proposed (see Chapter 4). This method has the advantage of having lower computational costs in training and implementation than classical approaches, such as SVR, LSSVM or Gaussian processes. This efficient method has been additionally implemented on two high performance computing platforms:

   a) Compute cluster: an implementation called PKWKNN was specifically created to handle data sets whose size would prevent the use of usual kernel methods (see section 4.7).

   b) Graphics Processing Unit: the implementation using CUDA accelerates dramatically the processing of data sets of medium size (see section 4.9).

   The algorithm has been compared with other models in the various experiments carried out in this report (see section 5.2.2), with results that demonstrate its competitiveness. KWKNN and PKWKNN were presented in a paper at the

ESTSP 2008 conference [Rubio et al., 2008], which was awarded by the ENNS (textit (European) Neural Network Society) as best paper of the conference, selected for publication in the special issue of Neurocomputing (Elsevier) and this results won the competition of time series prediction of the conference. The extended version of the previous work has been published [Rubio et al., 2010a]. In addition, a description from the technical point of view of PKWKNN was presented at the HPCS 2009 conference [Rubio et al., 2009a].

4. The creation of specific-to-problem kernels to the data for regression models, so far not considered, was addressed. This task was focused to the important case of time series forecasting. A design methodology kernels based on expert knowledge was obtained (see section 5.2), and successfully applied to time series long-term prediction, as shown in the aforementioned paper [Rubio et al., 2008] and [Rubio et al., 2010a]. A Genetic Programming algorithm was also created to attempt to obtain kernels adapted to a particular problem without prior knowledge (see section 5.3). It was presented at the IWANN 2009 conference [Rubio et al., 2009b]. The main novelty of the algorithm with respect to the literature is the evaluation of the kernels independently of a particular model using a modified version of the Delta test (see section 5.3.2).

5. We have worked in the implementation of two Matlab toolboxes that allow the use of HPC platforms compute cluster and NVIDIA graphics processor units. The first is MPIMEX, which gives access to the MPI library for parallel programming, and which performances over other alternatives are made evident in the study presented in Section 6.2. The second is CUBLASMEX, which gives access from Matlab to CUBLAS, an implementation of BLAS on NVIDIA GPUs.

8. PUBLICACIONES

- Revistas:

  - G. Rubio, H. Pomares, I. Rojas, L.J. Herrera; Heuristic for selection of parameters in LS-SVM for time series prediction; International Journal of Forecasting http://dx.doi.org/10.1016/j.ijforecast.2010.02.007 (Factor de Impacto: 1.685)
  - G. Rubio, L.J. Herrera, H. Pomares, I. Rojas, A. Guillén; Design of Specific-to-Problem Kernels and Use of Kernel Weighted K-Nearest Neighbours for Time Series Modelling; Neurocomputing; http://dx.doi.org/10.1016/j.neucom.2009.11.029 (Factor de Impacto: 1.234)
  - A. Guillén, G. Rubio, I. Toda, A. Rivera, H. Pomares, I. Rojas; Applying Multiobjective RBFNNs Optimization and Feature Selection to a Mineral Reduction Problem; Expert Systems with Applications (Factor de Impacto: 2.596)
  - A. Guillén, L.J. Herrera, G. Rubio, H. Pomares, A. Lendasse, I. Rojas; New Method for Instance or Prototype Selection using Mutual Information in Time Series Prediction; Neurocomputing (Factor de Impacto: 1.234)
  - A. Guillen, F. Garcia del Moral, G. Rubio, L.J. Herrera, I. Rojas, O. Valenzuela, H. Pomares; Using Near Infrared Spectroscopy in the Classification of White and Iberian Pork with Neural Networks; Neural Computing and Applications (Factor de Impacto: 0.767)

- Congresos Internacionales:

  - G. Rubio, H. Pomares, I. Rojas, L.J. Herrera, A. Guillén: Efficient Optimization of the Parameters of LS-SVM for Regression versus Cross-Validation Error. ICANN (2); pp. 406-415; 2009:
  - G. Rubio, H. Pomares, I. Rojas, A. Guillén; Creation of specific-to-problem kernel functions for function approximation ; IWANN 2009: Proceedings of the 10th International Work-Conference on Artificial Neural Networks, LNCS, Vol. 5517; pp. 335-342; 2009

- G. Rubio, A. Guillén, H. Pomares, I. Rojas; Parallelization of the Nearest Neighbour Search and the Cross-Validation Error Evaluation for the Kernel Weighted k-nn Algorithm Applied to Large Data Dets in Matlab; HPCS 2009: Proceedings of The 2009 High Performance Computing & Simulation (HPCS'09) Conference; 2009
- A. Guillén, Antti Sorjamaa, G. Rubio, Amaury Lendasse, I. Rojas: Mutual Information Based Initialization of Forward-Backward Search for Feature Selection in Regression Problems. ICANN (1) 2009: 1-9
- A. Guillén, L.J. Herrera, G. Rubio, H. Pomares, A. Lendasse, I. Rojas; Applying Mutual Information for Prototype or Instance Selection in Regression Problems; ESANN 2009 `http://www.dice.ucl.ac.be/Proceedings/esann/esannpdf/es2009-126.pdf`
- A. Guillén, A. Martinez-Trujillo, G. Rubio, I. Rojas, H. Pomares, L.J. Herrera; Risk Factor Identification and Classification of Macrosomic Newborns by Neural Networks; ISDA 2009
- A. Guillén, G. Rubio, S. Romero, L.J. Herrera, H. Pomares, O. Valenzuela; MPI Interface for Matlab Applied to Teaching Parallel Programming; International Conference on Education (IADAT 09)
- L.J. Herrera, G. Rubio, H. Pomares, B. Paechter, A. Guillén, I. Rojas: Strengthening the Forward Variable Selection Stopping Criterion. ICANN (2); pp. 215-224; 2009
- L.J. Herrera, H. Pomares, I. Rojas, A. Guillén, G. Rubio, J. M. Urquiza: Global and Local Modelling in Radial Basis Functions Networks. IWANN (1); pp. 49-56; 2009
- J. M. Urquiza, I. Rojas, H. Pomares, J. P. Florido, G. Rubio, L.J. Herrera, J. C. Calvo, Julio Ortega: Method for Prediction of Protein-Protein Interactions in Yeast Using Genomics/Proteomics Information and Feature Selection. IWANN (1); pp. 853-860; 2009
- G. Rubio, A. Guillen, L. J. Herrera, H. Pomares, I. Rojas; Use of specific-to-problem kernel functions for time series modeling; ESTSP'08: Proceedings of the European Symposium on Time Series Prediction, ISBN 978-951-22-9544-9; pp. 177-186; 2008
  - *Premio especial ENNS mejor artículo del congreso*
  - *Seleccionado para ampliación en número especial de Neurocomputing (editorial Elsevier)*
  - *El trabajo resultante ganó la competición de series temporales del congreso*

- Alberto Guillén, L.J. Herrera, G. Rubio, A. Lendasse, H. Pomares, I. Rojas; Instance or Prototype Selection for Function Approximation Using Mutual Information; ESTSP'08: Proceedings of the European Symposium on Time Series Prediction, ISBN 978-951-22-9544-9; pp. 1-6; 2008
- H. Pomares, I. Rojas, L.J. Herrera, A. Guillén, M. Damas, G. Rubio, J. González; Platform-Independent Didactic Environment for Digital Systems Design; Proceedings of the IASK International Conference Teaching and Learning; pp. 932-936; 2008
- A. Guillén, I. Rojas, G. Rubio, H. Pomares, L.J. Herrera, J. González; A New Interface for MPI in Matlab and its Application over a Genetic Algorithm; ESTSP'08: Proceedings of the European Symposium on Time Series Prediction, ISBN 978-951-22-9544-9; pp. 1-6; 2008
- G. Rubio, H. Pomares, L. J. Herrera, I. Rojas; Kernel Methods Applied to Time Series Forecasting; LNCS, ISSN 0302-9743, Vol. 4507; pp. 782-789; 2007
- G. Rubio, L. J. Herrera, H. Pomares; Gaussian Process regression applied to time series prediction: a particular case study in ESTSP 2007 time series prediction competition; ESTSP'07: Proceedings of the European Symposium on Time Series Prediction; pp. 21-27; isbn 978-951-22-9601-0; 2007
  - *El trabajo resultante quedó en tercer puesto en la competición de series temporales del congreso*
- L.J. Herrera, H. Pomares, I. Rojas, A. Guillén, G. Rubio: On Incorporating Seasonal Information on Recursive Time Series Predictors. ICANN (2) 2007: 506-515
- L.J. Herrera, H. Pomares, I. Rojas, G. Rubio, A. Guillén; Removing Seasonality on Time Series, a Practical Case; ESTSP'07: Proceedings of the European Symposium on Time Series Prediction; pp. 279-286; 2007
- G. Rubio, H. Pomares; A Basic Approach to Reduce the Complexity of a Self-generated Fuzzy Rule-Table for Function Approximation by Use of Symbolic Interpolation; Lecture Notes on Computer Science (LNCS), Vol 3512, pp.34-41, June 2005
- G. Rubio, H. Pomares, I. Rojas, A. Guillén; A Basic Approach to Reduce the Complexity of a Self-generated Fuzzy Rule-Table for Function Approximation by Use of Symbolic Regression in 1D and 2D Cases; LNCS Vol. 3562, pp 143-152; 2005

- Congresos Nacionales:

- A. Guillén, G. Rubio, S. Romero, O. Valenzuela; Interfaz para MPI en MATLAB (MPIMEX) aplicada a la docencia en programación paralela; Congreso de Docencia Universitaria (Vigo) 2009
- A. Guillén, F. G. del Moral, G. Rubio, L. J. Herrera, I. Rojas; Clasificación del Cerdo Ibérico Utilizando Espectroscopía Infrarroja y Redes Neuronales, ISBN: 978-84-691-5807-4, ESTYLF08, Cuencas Mineras (Mieres - Langreo), 17 - 19 de Septiembre de 2008, pág. 695-699
- L. J. Herrera, H. Pomares, I. Rojas, A. Guillén, G. Rubio; Modelado Recursivo de Series Temporales Utilizando Modelos TSK Avanzados, ISBN: 978-84-691-5807-4, ESTYLF08, Cuencas Mineras (Mieres - Langreo), 17 - 19 de Septiembre de 2008, pág. 701-706
- G. Rubio, H. Pomares, I. Rojas; Aplicación de la programación genética a la obtención de modelos analíticos no lineales para predicción de series temporales; Actas del Simposio de Inteligencia Computacional, SICO2005 (IEEE Computacional Inteligence society, SC), pp. 3-10, Thomson, ISBN84-9732-444-7, I Congreso Español de Informática Informática (CEDI2005) Granada, 13-16 Septiembre 2005

BIBLIOGRAFÍA

(2007). *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*.

Agirre-Basurko, E., Ibarra-Berastegi, G., and Madariaga, I. (2006). Regression and multilayer perceptron-based models to forecast hourly o3 and no2 levels in the bilbao area. *Environ. Model. Softw.*, 21(4):430–446.

Ahdesmäki, M., Lähdesmäki, H., Pearson, R., Huttunen, H., and Yli-Harja, O. (2005). Robust detection of periodic time series measured from biological systems. *BMC Bioinformatics*, 6:117.

Akaike, H. (1974). A New Look at the Statistical Model Identification. *IEEE Trans. Automat. Contr.*, 19:716–723.

Almasi, G., Cascaval, C., and Padua, D. A. (1999). Mat marks: A shared memory environment for matlab programming. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 21, Washington, DC, USA. IEEE Computer Society.

Alves da Silva, A. P., Ferreira, V. H., and Velasquez, R. M. (2008). Input space to neural network based load forecasters. *International Journal of Forecasting*, 24(4):616–629.

Amdahl, G. (1967). Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Conference*, pages 483–485.

An, S., Liu, W., and Venkatesh, S. (2007). Fast cross-validation algorithms for least squares support vector machine and kernel ridge regression. *Pattern Recogn.*, 40(8):2154–2162.

Aparício, G., Blanquer, I., and Hernández, V. (2006). A parallel implementation of the k nearest neighbours classifier in three levels: Threads, mpi processes and the grid. In *VECPAR*, pages 225–235.

A.Sorjamaa, J.Hao, N.Reyhani, Y.Ji, and A.Lendasse (2007). Methodology for long-term prediction of time series. *Neurocomputing*, 70:2861–2869.

Bäck, T. (1995). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York.

Baker, M. and Carpenter, B. (2000). Mpj: A proposed java message passing api and environment for high performance computing. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 552–559, London, UK. Springer-Verlag.

Baker, M. A., Grove, M., and Shafi, A. (2006). Parallel and distributed computing with java. In *ISPDC '06: Proceedings of the Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, pages 3–10, Washington, DC, USA. IEEE Computer Society.

Balkin, S. D. and Ord, J. K. (2000). Automatic neural network modeling for univariate time series. *International Journal of Forecasting*, 16(4):509 – 515.

B.B., C. (September 2000). Efficient training and improved performance of multilayer perceptron in pattern classification. *Neurocomputing*, 34:11–27(17).

Belleman, R. G., Bédorf, J., and Zwart, S. F. P. (2008). High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy*, 13(2):103 – 112.

Bellman, R. (1966). Dynamic programming, system identification, and suboptimization. *SIAM Journal on Control and Optimization*, 4:1–5.

Bello, M. (1992). Enhanced training algorithms, and integrated training/architecture selection for multilayer perceptron networks. *Neural Networks, IEEE Transactions on*, 3(6):864 –875.

Bennet, K., Ferris, M. C., and Ioannidis, Y. E. (1991). A Genetic Algorithm for Database Query Optimization. In Belew, R. and Booker, L. B., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 400–407, San Mateo, CA. Morgan Kaufmann.

Benoudjit, N. and Verleysen, M. (2003). On the kernel widths in radial basis function networks. *Neural Processing Letters*, 18(2):139–154.

Berardi, V. and Zhang, G. (May 2003). An empirical investigation of bias and variance in time series forecasting: modeling considerations and error evaluation. *Neural Networks, IEEE Transactions on*, 14(3):668–679.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA.

Bo, L., Wang, L., and Jiao, L. (2006). Sparse gaussian processes using backward elimination. In *ISNN (1)*, pages 1083–1088.

Booker, L. B. (1982). *Intelligent Behavior as an Adaption to the Task Environment*. PhD thesis, University of Michigan.

Bors, A. G. (2001). Introduction of the Radial Basis Function (RBF) networks. *OnLine Symposium for Electronics Engineers*, 1:1–7.

Bosch, A. v. d. and Sloot, K. v. d. (2007). Superlinear parallelization of k-nearest neighbor retrieval.

Box, G. E. P. and Jenkins, G. M. (1976). *Time series analysis, forecasting and control*. Holden Day: San Francisco, CA.

Brahim-Belhouari, S. and Bermak, A. (2004). Gaussian process for nonstationary time series prediction. *Computational Statistics & Data Analysis*, 47(4):705–712. available at http://ideas.repec.org/a/eee/csdana/v47y2004i4p705-712.html.

Branke, J. (1995). Evolutionary Algorithms in Neural Network Design and Training – A Review. In Alander, J. T., editor, *Proc. of the First Nordic Workshop on Genetic Algorithms and their Applications*, number 95-1, pages 145–163, Vaasa, Finnland.

Broomhead, D. S. and Lowe, D. (1988). Multivariate Functional Interpolation and Adaptive Networks. *Complex Systems*, 2:321–355.

Broyden, C. G. (1970). The convergence of a class of double-rank minimization algorithms, II: The new algorithm. 6(??):222–231.

Bruck, J., Dolev, D., Ho, C.-T., Roşu, M.-C., and Strong, R. (1995). Efficient message passing interface (mpi) for parallel computing on clusters of workstations. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 64–73, New York, NY, USA. ACM.

B.V. Bonnlander and A.S. Weigend (2004). Selecting input variables using mutual information and nonparametric density estimation. In *Proc. of the ISANN*, Taiwan.

Catanzaro, B., Sundaram, N., and Keutzer, K. (2008). Fast support vector machine training and classification on graphics processors. Technical Report 11, EECS Department, University of California, Berkeley.

Cawley, G. C. and Talbot, N. L. C. (2002). Efficient formation of a basis in a kernel induced feature space. In *ESANN*, pages 1–6.

Chattopadhyay, S. (2007). Prediction of mean monthly total ozone time series-application of radial basis function network. *Int. J. Remote Sens.*, 28(18):4037–4046.

Chauveau, S. and Bodin, F. (1998). Menhir: An environment for high performance matlab. In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 27–40, London, UK. Springer-Verlag.

Chen, S., Cowan, C. F., and Grant, P. M. (1991). Orthogonal Least Squares Learning Algorithm for Radial Basis Function Networks. *IEEE Trans. Neural Networks*, 2:302–309.

Chen, S., Grant, P. M., and Cowan, C. F. (1992). Orthogonal Least Squares Learning Algorithm for Training Multi-Output Radial Basis Function Networks. In *Proceedings of the Institution of Electrical Engineers*, volume 139, pages 378–384.

Cherkassky, V. and Ma, Y. (2004). Practical selection of svm parameters and noise estimation for svm regression. *Neural Netw.*, 17(1):113–126.

Choy, R., Edelman, A., and Of, C. M. (2005). Parallel matlab: Doing it right. In *Proceedings of the IEEE*, pages 331–341.

Coulibaly, P., Anctil, F., and Bobée, B. (2000). Daily reservoir inflow forecasting using artificial neural networks with stopped training approach. *Journal of Hydrology*, 230(3-4):244 – 257.

Coulibaly, P., Anctil, F., and Bobee, B. (2001). Multivariate reservoir inflow forecasting using temporal neural networks. *Journal of Hydrologic Engineering*, 6:367–376.

Crammer, K., Keshet, J., and Singer, Y. (2002). Kernel design using boosting. In Becker, S., Thrun, S., and Obermayer, K., editors, *NIPS*, pages 537–544. MIT Press.

Cristianini, N., Shawe-Taylor, J., Elisseeff, A., and Kandola, J. (2002). On kernel-target alignment. In Dietterich, T. G., Becker, S., and Ghahramani, Z., editors, *Advances in Neural Information Processing Systems 14*, Cambridge, MA. MIT Press.

Dalcin, L., Paz, R., and Storti, M. (2005). Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115.

Dalcín, L., Paz, R., Storti, M., and D'Elía, J. (2008). Mpi for python: Performance improvements and mpi-2 extensions. *Journal Parallel Distributed Computing*, 68(5):655–662.

Darwin, C. (1859). *The Origin of Species by means of Natural Selection*. The Modern Library, London.

Dasarathy, B. V. (1990). *Nearest neighbor (NN) norms: NN pattern classification techniques*. Los Alamitos: IEEE Computer Society Press, 1990.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan.

De Jong, K. A. (1985). Genetic Algorithms: A 10 Year Perspective. In Grefenstette, J. J., editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 169–177, Hillsdale, NJ. Lawrence Erlbaum Associates.

de Kruif, B. and de Vries, T. (May 2003). Pruning error minimization in least squares support vector machines. *Neural Networks, IEEE Transactions on*, 14(3):696–702.

D.E. Rumelhart, G.E. Hinton, R. W. (1986). Learning representations by back-propagation errors. *Nature*, 323:533–536.

Dehghan, M. and Shokri, A. (2008). A numerical method for solution of the two-dimensional sine-gordon equation using the radial basis functions. *Math. Comput. Simul.*, 79(3):700–715.

Diosan, L., Rogozan, A., and Pecuchet, J. P. (2007). Evolving kernel functions for svms by genetic programming. In *ICMLA '07: Proceedings of the Sixth International Conference on Machine Learning and Applications*, pages 19–24, Washington, DC, USA. IEEE Computer Society.

Drepper, U. and Molnar, I. (2005). Redhat: The native posix thread library for linux. Technical report.

Dutilleul, P. (2001). Multi-frequential periodogram analysis and the detection of periodic components in time series. *Communications in Statistics - Theory and Methods*, 30:1063 – 1098.

Evans, D. (2002). The gamma test: Data derived estimates of noise for unknown smooth models using near neighbour asymptotics.

Fagg, G. E., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., and Dongarra, J. (2005). Process fault tolerance: Semantics, design and applications for high performance computing. *IJHPCA*, 19(4):465–477.

Farmer, J. D., Packard, N. H., and Perelson, A. S. (1986). The immune system, adaptation, and machine learning. *Phys. D*, 2(1-3):187–204.

Fernandez, J., Anguita, M., Ros, E., and Bernier, J. (2006). SCE Toolboxes for the development of high-level parallel applications. *Lecture Notes in Computer Science*, 3992:518–525.

Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948+.

Fogel, D. B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, NJ.

Forrest, S. (1993). Genetic Algorithms - Principles of Natural Selection applied to Computation. *Science*, 261(5123):872–878.

Foster, I., Geisler, J., Gropp, W., Karonis, N., Lusk, E., Thiruvathukal, G., and Tuecke, S. (1998). Wide-area implementation of the message passing interface. *Parallel Comput.*, 24:1735–1749.

Francq, C., Roy, R., and Zakoian, J.-M. (2005). Diagnostic checking in arma models with uncorrelated errors. *Journal of the American Statistical Association*, 100:532–544.

Frederick P. Brooks, J. (1978). *The Mythical Man-Month: Essays on Softw*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gagné, C., Schoenauer, M., Sebag, M., and Tomassini, M. (2006). Genetic programming for kernel-based learning with co-evolving subsets selection. *DANS PPSN*, 06:1008.

Garcia, V., Debreuve, E., and Barlaud, M. (2008). Fast k nearest neighbor search using gpu.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. (2008). Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27.

Gaur, V., Giloni, A., and Seshadri, S. (2005). Information Sharing in a Supply Chain Under ARMA Demand. *MANAGEMENT SCIENCE*, 51(6):961–969.

Goldberg, D. E. (1989a). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley.

Goldberg, D. E. (1989b). *Genetic Algorithms in Search, Optimization, and Machine Learning* . Addison-Wesley: New York.

Goldberg, D. E. (1994). Genetic and Evolutionary Algorithms come of age. *Communications of the ACM.*, 37(3):113–119.

Gomm, J. and Yu, D. (2000). Selecting radial basis function network centers with recursive orthogonal least squares training. *Neural Networks, IEEE Transactions on*, 11(2):306 –314.

González, J., Rojas, I., Ortega, J., Pomares, H., Fernández, F., and Díaz, A. (2003). Multiobjective evolutionary optimization of the size, shape, and position parameters of radial basis function networks for function approximation. *IEEE Transactions on Neural Networks*, 14(6):1478–1495.

González, J., Rojas, I., Pomares, H., Ortega, J., and Prieto, A. (2002). A new Clustering Technique for Function Aproximation. *IEEE Transactions on Neural Networks*, 13(1):132–142.

Graf, H. P., Cosatto, E., Bottou, L., Durdanovic, I., and Vapnik, V. (2005). Parallel support vector machines: The cascade svm. In *In Advances in Neural Information Processing Systems*, pages 521–528. MIT Press.

Gregor, D. and Lumsdaine, A. (2008). "design and implementation of a high-performance mpi for c# and the common language infrastructure". In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 133–142, New York, NY, USA. ACM.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828.

Guillén, A. MATLAB Interface for Sun MPI functions. *http://atc.ugr.es/ãguillen/MPI/MPI_MATLAB.htm*.

Guillén, A. (2005). *Writing programs in MATLAB using any implementation of MPI*. http://atc.ugr.es/ aguillen.

Guillén, A., González, J., Rojas, I., Pomares, H., Herrera, L., Valenzuela, O., and Prieto, A. (2007). Improving Clustering Technique for Functional Approximation Problem Using Fuzzy Logic: ICFA algorithm. *Neurocomputing*, DOI: 10.1016/j.neucom.2006.06.017.

Guillén, A., Rojas, I., González, J., Pomares, H., and Herrera, L. J. (2005). RBF centers initialization using fuzzy clustering technique for function approximation problems. *Fuzzy Economic Review*, 10(2):27–45.

Guillen, A., Rojas, I., Rubio, G., Pomares, H., Herrera, L., and Gonzalez, J. (2008a). A new interface for mpi in matlab and its application over a genetic algorithm. In *Proceedings of the European Symposium on Time Series Prediction*.

Guillen, A., Rojas, I., Rubio, G., Pomares, H., Herrera, L., and Gonzalez, J. (2008b). A new interface for mpi in matlab and its application over a genetic algorithm. *ESTSP'08: Proceedings of the European Symposium on Time Series Prediction*, pages 37–46.

Harpham, C. and Dawson, C. (2006). The effect of different basis functions on a radial basis function network for time series prediction: A comparative study. *Neurocomputing*, 69(16-18):2161 – 2170. Brain Inspired Cognitive Systems - Selected papers from the 1st International Conference on Brain Inspired Cognitive Systems (BICS 2004).

Hatanaka, T., Kondo, N., and Uosaki, K. (2003). Multi-objective structure selection for radial basis function networks based on genetic algorithm. In *The 2003 Congress on Evolutionary Computation*, pages 1095–1100.

Haykin, S. (1994). *Neural networks: a comprehensive foundation*. Prentice Hall.

Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Hénon, M. (1976). A two-dimensional mapping with a strange attractor. *Communications in Mathematical Physics*, 50:69–77.

Herrera, L., Pomares, H., Rojas, I., Guillén, A., Prieto, A., and Valenzuela, O. (2007a). Recursive prediction for long term time series forecasting using advanced models. *Neurocomputing*, 70(16-18):2870–2880.

Herrera, L., Pomares, H., Rojas, I., Guillén, A., and Rubio, G. (2007b). On incorporating seasonal information on recursive time series predictors. In *proceedings of ICANN (2)*, pages 506–515.

Herrera, L., Pomares, H., Rojas, I., Valenzuela, O., and Prieto, A. (2005). TaSe, a Taylor Series-based fuzzy system model that combines interpretability and accuracy. *Fuzzy Sets and Systems*, 153:403–427.

Hinton, G. E.; Sejnowski, T. J. (1983). Analyzing cooperative computation.

Holland, J. J. (1975). *Adaption in Natural and Artificial Systems*. University of Michigan Press.

Hontoria, L., Aguilera, J., and Zufiria, P. (2005). An application of the multilayer perceptron: Solar radiation maps in spain. *Solar Energy*, 79(5):523 – 530.

Hopfield, J. J. (1988). Neural networks and physical systems with emergent collective computational abilities. pages 457–464.

Howley, T. and Madden, M. (November 2005). The genetic kernel support vector machine: Description and evaluation. *Artificial Intelligence Review*, 24:379–395(17).

Hsieh, W. W. and Tang, B. (1998). Applying neural network models to prediction and data analysis in meteorology and oceanography. 89(9):1855–1870.

Hunt, J. E. and Cooke, D. E. (1996). Learning using an artificial immune system. *Journal of Network and Computer Applications*, 19(2):189 – 212.

Intel (2008). Cluster openmp*, user's guide. `http://cache-www.intel.com/cd/00/00/29/78/297875_297875.pdf`.

Jang, J. S. R. (1993). ANFIS: Adaptive Network-based Fuzzy Inference System. *IEEE Trans. Syst., Man, Cybern.*, 23:665–685.

Ji, Y., Hao, J., Reyhani, N., and Lendasse, A. (2005). Direct and recursive prediction of time series using mutual information selection. In *IWANN*, pages 1010–1017.

Jones, A. J., Evans, D., and Kemp, S. E. (2007). A note on the Gamma test analysis of noisy input/output data and noisy time series. *Physica D Nonlinear Phenomena*, 229:1–8.

Jursa, R. and Rohrig, K. (2008). Short-term wind power forecasting using evolutionary algorithms for the automated specification of artificial intelligence models. *International Journal of Forecasting*, 24(4):694–709.

Kanjilal, P. P. and Banerjee, D. (1995). On the Application of Orthogonal Transformation for the Design and Analysis of Feedforward Networks. *IEEE Trans. Neural Networks*, 6(5):1061–1070.

Karayiannis, N. B. and Mi, G. W. (1997a). Growing Radial Basis Neural Networks: Merging Supervised and Unsupervised Learning with Network Growth Techniques. *IEEE Trans. Neural Networks*, 8(6):1492–1506.

Karayiannis, N. B. and Mi, G. W. (1997b). Growing radial basis neural networks: Merging supervised and unsupervised learning with network growth techniques. *IEEE Transactions on Neural Networks*, 8:1492–1506.

Keane, A. J., Choudhury, A., Choudhury, A., Nair, P. B., Nair, P. B., Choudhury, A. J. K. F., and Nair, P. B. (2002). A data parallel approach for large-scale gaussian process modeling. In *in Proc. the Second SIAM International Conference on Data Mining*.

Kepner, J. (2004). High performance computing productivity model synthesis. *Int. J. High Perform. Comput. Appl.*, 18(4):505–516.

Kepner, J. and Ahalt, S. (2004). Matlabmpi. *Journal of Parallel and Distributed Computing*, 64(8):997 – 1005.

K.I. Funahashi (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192.

Kohonen, T., editor (1997). *Self-organizing maps*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Kourentzes, N. and Crone, S. (2008). Automatic modelling of neural networks for time series prediction - in search of a uniform methodology across varying time frequencies. In *ESTSP'08: Proceedings of the European Symposium on Time Series Prediction*, pages 117–127.

Koza, J. (1992). *Genetic Programming*. MIT Press.

Kraskov, A., Stögbauer, H., and Grassberger, P. (2004). Estimating mutual information. *Phys. Rev. E*, 69(6):066138.

Leithead, W. and Zhang, Y. (2007). $O(N^2)$-operation approximation of covariance matrix inverse in Gaussian process regression based on quasi-Newton BFGS method. *Commun. Stat., Simulation Comput.*, 36(2):367–380.

Lendasse, A., Corona, F., Hao, J., Reyhani, N., and Verleysen, M. (2006). Determination of the mahalanobis matrix using nonparametric noise estimations. In *ESANN*, pages 227–232.

Lendasse, A., Ji, Y., Reyhani, N., and Verleysen, M. (2005). Ls-svm hyperparameter selection with a nonparametric noise estimator. In *ICANN (2)*, pages 625–630.

Liitiäinen, E., Lendasse, A., and Corona, F. (2007). Non-parametric residual variance estimation in supervised learning. In *IWANN*, pages 63–71.

Lin, G. and Chen, L. (2005). Time series forecasting by combining the radial basis function network and the self-organizing map. *Hydrological Processes*, 19:1925–1937.

Liu, W., Schmidt, B., Voss, G., and Müller-Wittig, W. (2008). Accelerating molecular dynamics simulations using graphics processing units with cuda. *Computer Physics Communications*, 179(9):634 – 641.

Lora, A. and Santos, J. (2002). A comparison of two techniques for next-day electricity price forecasting. In *Third International Conference on Intelligent Data Engineering and Automated Learning, LNCS*, volume 2412, pages 384–390.

MacKay, D. J. C. (1998). Introduction to Gaussian processes. In Bishop, C. M., editor, *Neural Networks and Machine Learning*, NATO ASI Series, pages 133–166. Kluwer.

Mackey, M. C. and Glass, L. (1977). Oscillation and Chaos in Physiological Control Systems. *Science*, 197(4300):287–289.

Manavski, S. and Valle, G. (2008). Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10.

May, R. M. (1976). Simple Mathematical Models with Very Complicated Dynamics. *Nature*, 261:459–467.

Methasate, I. and Theeramunkong, T. (2007). Kernel Trees for Support Vector Machines. *IEICE Trans Inf Syst*, E90-D(10):1550–1556.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures=Evolution Programs*. Springer-Verlag, 3rd edition.

Michalewicz, Z., Krawczyk, J., Kazemi, M., and Janikow, C. (1990). Genetic Algorithms and Optimal Control Problems. In *Proceedings of the 29th IEEE Conference on Decision and Control*, pages 1664–1666, Honolulu. IEEE Computer Society Press.

Misra, D., Oommen, T., Agarwal, A., Mishrad, S. K., and Thompsone, A. M. (2009). Application and analysis of support vector machine based simulation for runoff and sediment yield. *Biosystems Engineering*, 103:527–535.

Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Comps. in Opns. Res.*, 24:1097–1100.

Moody, J. E. and Darken, C. J. (1989). Fast Learning in Networks of Locally-Tuned Processing Units. *Neural Computation*, 1:281–294.

Morrow, G. and van de Geijn, R. (1998). A parallel linear algebra server for matlab-like environments. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, Washington, DC, USA. IEEE Computer Society.

Müller, K.-R., Mika, S., Rätsch, G., Tsuda, K., and Schölkopf, B. (2001). An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12:181–201.

Müller, K.-R., Smola, A., Rätsch, G., Schölkopf, B., Kohlmorgen, J., and Vapnik, V. (2000). Using support vector machines for time series prediction.

Murray, A. and Edwards, P. (1993). Synaptic weight noise during multilayer perceptron training: fault tolerance and training improvements. *Neural Networks, IEEE Transactions on*, 4(4):722 –725.

Musavi, M. T., Ahmed, W., Chan, K. H., Faris, K. B., and Hummels, D. M. (1992). On the Training of Radial Basis Function Classifiers. *Neural Networks*, 5(4):595–603.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with cuda. *Queue*, 6(2):40–53.

Norris, B. R. (2000). *An environment for interactive parallel numerical computing*. PhD thesis, Champaign, IL, USA. Adviser-Michael T. Heath.

Ong, C. S., Smola, A. J., and Williamson, R. C. (2005). Learning the kernel with hyperkernels. *Journal of Machine Learning Research*, 6:1043–1071.

Pai, P.-F. and Hong, W.-C. (2005). Support vector machines with simulated annealing algorithms in electricity load forecasting. *Energy Conversion and Management*, 46(17):2669 – 2688.

Park, J. and Sandberg, J. W. (1991). Universal approximation using radial basis functions network. *Neural Computation*, 3:246–257.

Pi, H. and Peterson, C. (1994). Finding the embedding dimension and variable dependencies in time series. *Neural Computation*, 6(3):509–520.

Poggio, T. and Girosi, F. (1990). Networks for approximation and learning. In *Proceedings of the IEEE*, volume 78, pages 1481–1497.

Pomares, H., Rojas, I., Ortega, J., González, J., and Prieto, A. (2000). A Systematic Approach to a Self-Generating Fuzzy Rule-Table for Function Approximation. *IEEE Trans. Syst., Man, Cyber. Part B*, 30(3):431–447.

Quinn, M. J., Malishevsky, A., and Seelam, N. (1998). Otter: Bridging the gap between matlab and scalapack. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 114, Washington, DC, USA. IEEE Computer Society.

Rasmussen, C. E. and Williams, C. K. I. (2005). *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14:445–471.

Rojas, I., Anguita, M., Prieto, A., and Valenzuela, O. (1998). Analysis of the operators involved in the definition of the implication functions and in the fuzzy inference proccess. *International Journal of Approximate Reasoning*, 19:367–389.

Rojas, I., González, J., Cañas, A., Díaz, A. F., Rojas, F. J., and Rodriguez, M. (2000a). Short-Term Prediction of Chaotic Time Series by Using RBF Network with Regression Weights. *Int. Journal of Neural Systems*, 10(5):353–364.

Rojas, I., González, J., Pomares, H., Rojas, F. J., Fernández, F. J., and Prieto, A. (2001). Multidimensional and Multideme Genetic Algorithms for the Construction of Fuzzy Systems. *International Journal of Approximate Reasoning*, 26:179–210.

Rojas, I., Pomares, H., Gonzáles, J., Bernier, J., Ros, E., Pelayo, F., and Prieto, A. (2000b). Analysis of the functional block involved in the design of radial basis function networks. *Neural Processing Letters*, 12:1–17.

Rojas, I., Pomares, H., González, J., Bernier, J. L., Ros, E., Pelayo, F., and Prieto, A. (2000c). Analysis of the functional block involved in the design of radial basis function networks. *Neural Processing Letters*, 12(1):1–17.

Rojas, I., Pomares, H., González, J., Ros, E., Salmerón, M., Ortega, J., and Prieto, A. (2000d). A New Radial Basis Function Networks Structure: Application to Time Series Prediction. In Amari, S. I., Giles, C. L., Gori, M., and Piuri, V., editors, *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, volume IV, pages 449–454, Como, Italy. IEEE Computer Society.

Rojas, I., Valenzuela, O., Rojas, F., Guillen, A., Herrera, L., Pomares, H., Marquez, L., and Pasadas, M. (2008). Soft-computing techniques and arma model for time series prediction. *Neurocomputing*, 71(4-6):519 – 537. Neural Networks: Algorithms and Applications, 4th International Symposium on Neural Networks; 50 Years of Artificial Intelligence: a Neuronal Approach, Campus Multidisciplinary in Perception and Intelligence.

Rose, L. D., Gallivan, K., Gallopoulos, E., Marsolf, B. A., and Padua, D. A. (1996). Falcon: A matlab interactive restructuring compiler. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 269–288, London, UK. Springer-Verlag.

Rubio, G., Guillen, A., Herrera, L. J., Pomares, H., and Rojas, I. (2008). Use of specific-to-problem kernel functions for time series modeling. In *ESTSP'08: Proceedings of the European Symposium on Time Series Prediction*, pages 177–186.

Rubio, G., Guillen, A., Pomares, H., Rojas, I., Paechter, B., Glosekotter, P., and Torres-Ceballos, C. (2009a). Parallelization of the nearest-neighbour search and the cross-validation error evaluation for the kernel weighted k-nn algorithm applied to large data dets in matlab. In *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, pages 145 –152.

Rubio, G., Herrera, L. J., and Pomares, H. (2007). Gaussian Process regression applied to time series prediction: a particular case study in ESTSP 2007 time series prediction competition. In *ESTSP'07: Proceedings of the European Symposium on Time Series Prediction*, pages 21–27.

Rubio, G., Herrera, L. J., Pomares, H., Rojas, I., and Guillén, A. (2010a). Design of specific-to-problem kernels and use of kernel weighted k-nearest neighbours for time series modelling. *Neurocomputing*, 73(10-12):1965 – 1975.

Rubio, G., Pomares, H., Rojas, I., and Guillén, A. (2009b). Creation of specific-to-problem kernel functions for function approximation. In *IWANN '09: Proceedings of the 10th International Work-Conference on Artificial Neural Networks*, pages 335–342, Berlin, Heidelberg. Springer-Verlag.

Rubio, G., Pomares, H., Rojas, I., and Herrera, L. J. (2010b). A heuristic method for parameter selection in ls-svm: Application to time series prediction. *International Journal of Forecasting*.

Rubio, G., Pomares, H., Rojas, I., Herrera, L. J., and Guillén, A. (2009c). Efficient optimization of the parameters of ls-svm for regression versus cross-validation error. In *ICANN (2)*, pages 406–415.

Sapankevych, N. I. and Sankar, R. (2009). Time series prediction using support vector machines: A survey. *Computational Intelligence Magazine, IEEE*, 4(2):24–38.

Schölkopf, B., Mika, S., Burges, C. J. C., Knirsch, P., Müller, K. R., Rätsch, G., and Smola, A. J. (1999). Input space versus feature space in kernel-based methods. *IEEE Transactions on Neural Networks*, 10(5):1000–1017.

Scholkopf, B. and Smola, A. J. (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA.

Seidel, P., Seidel, A., and Herbarth, O. (2007). Multilayer perceptron tumour diagnosis based on chromatography analysis of urinary nucleosides. *Neural Netw.*, 20(5):646–651.

Shiblee, M., Kalra, P. K., and Chandra, B. (2009). Time series prediction with multilayer perceptron (mlp): A new generalized error based approach. *Advances in Neuro-Information Processing*, pages 37–44.

Smith, M. (1993). *Neural Networks for Statistical Modeling*. John Wiley & Sons, Inc., New York, NY, USA.

Sonnenburg, S., Rätsch, G., and Schäfer, C. (2005). A general and efficient multiple kernel learning algorithm. In *NIPS*.

Sorjamaa, A., Hao, J., and Lendasse, A. (2005). Mutual Information and k-Nearest Neighbors Approximator for Time Series Prediction. *Lecture Notes in Computer Science*, 3697:553–558.

Sullivan, K. M. and Luke, S. (2007). Evolving kernels for support vector machine classification. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1702–1707, New York, NY, USA. ACM.

Suykens, J., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J. (2002). *Least Squares Support Vector Machines*. World Scientific Publishing, Singapore.

Suykens, J. and Vandewalle, J. (1999). Training multilayer perceptron classifiers based on a modified support vector method. *Neural Networks, IEEE Transactions on*, 10(4):907 –911.

Tay, F.E.H., L. C. (2001). Application of support vector machines in financial time series forecasting. *Omega: The International Journal of Management Science*, 29(4):309–317.

Teräsvirta, T., Medeiros, M. C., and Rech, G. (2006). Building neural network models for time series: a statistical approach. *Journal of Forecasting*, 25(1):49–75.

Thiessen, U. and van Brakel, R. (2003). Using support vector machines for time series prediction. *Chemometrics and Intelligent Laboratory Systems*, 69:35–49.

Torres, J., García, A., Blas, M. D., and Francisco, A. D. (2005). Forecast of hourly average wind speed with arma models in navarre (spain). *Solar Energy*, 79(1):65 – 77.

Trefethen, A. E., Menon, V. S., Chang, C., Czajkowski, G., Myers, C., and Trefethen, L.Ñ. (1996). Multimatlab: Matlab on multiple processors. Technical report, Ithaca, NY, USA.

Uykan, Z. and Güzelis, C. (1997). Input-output clustering for determining the centers of radial basis function network. In *in Proc. ECCTD 1997*, pages 435–439, Budapest, Hungary.

Valenzuela, O., Rojas, I., Rojas, F., Pomares, H., Herrera, L., Guillen, A., Marquez, L., and Pasadas, M. (2008). Hybridization of intelligent techniques and arima models for time series prediction. *Fuzzy Sets and Systems*, 159(7):821 – 845. Theme: Fuzzy Models and Approximation Methods.

Van Gestel, T., Suykens, J., Baestaens, D.-E., Lambrechts, A., Lanckriet, G., Vandaele, B., De Moor, B., and Vandewalle, J. (2001). Financial time series prediction using least squares support vector machines within the evidence framework. *Neural Networks, IEEE Transactions on*, 12(4):809–821.

Vignaux, G. A. and Michalewicz, Z. (1991). A Genetic Algorithm for the Linear Transportation Problem. *IEEE Trans. Syst. Man and Cybern.*, 21(2):445–452.

Vlachos, M., Yu, P., and Castelli, V. (2005). On periodicity detection and structural periodic similarity. In *Proceedings of SDM*.

Volkov, V. and Demmel, J. (2008). Lu, qr and cholesky factorizations using vector capabilities of gpus. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley.

Voorst, B. V. and Seidel, S. (2000). Comparison of mpi implementations on a shared memory machine. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 847–854, London, UK. Springer-Verlag.

Walker, D. W. (1994). The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Comput.*, 20:657–673.

Weigend, A. S. and Gershenfeld, N. A., editors (1994). *Time series prediction: Forecasting the future and understanding the past*.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, Cambridge, MA.

Willcock, J., Lumsdaine, A., and Robison, A. (2002). robison,a, using mpi with c# and the common language infrastructure". Technical report.

Wu, W., Massart, D. L., and de Jong, S. (1997). The kernel pca algorithms for wide data. part i: Theory and algorithms. *Chemometrics and Intelligent Laboratory Systems*, 36(2):165 – 172.

Xu, R.R., G. B. (2005). Discussion about nonlinear time series prediction using least squares support vector machine. *Communications in Theoretical Physics*, 43:1056–1060.

Ying, Z. and Keong, K. C. (23-26 Aug. 2004). Fast leave-one-out evaluation and improvement on inference for ls-svms. *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, 3:494–497 Vol.3.

Zeng, X. and wen Chen, X. (Nov. 2005). Smo-based pruning methods for sparse least squares support vector machines. *Neural Networks, IEEE Transactions on*, 16(6):1541–1546.

Zhang, D.-Q. and Chen, S.-C. (2003). Clustering incomplete data using kernel-based fuzzy c-means algorithm. *Neural Process. Lett.*, 18(3):155–162.

Zhang, G. P. (2003). Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175.

Zhang, Y. and Leithead, W. (2007). Approximate implementation of the logarithm of the matrix determinant in Gaussian process regression. *J. Stat. Comput. Simulation*, 77(4):329–348.

Zhou, J., Bai, T., Zhang, A., and Tian, J. (2008). Forecasting Share Price using Wavelet Transform and LS-SVM based on Chaos Theory. In *IEEE International Conference on Cybernetic Intelligent Systems (CIS 2008), SEP 21-24, 2008 Chengdu, PEOPLES R CHINA*, pages 300–304.