Universidad de Valladolid

Facultad de Ciencias

Trabajo Fin de Grado

Grado en Estadística

Análisis y mejoras en algoritmos de cálculo de estimadores bajo restricciones Up-Down-Up y su aplicación en cronobiología

Autor: Adrián Lamela Pérez

Tutor: Miguel Alejandro Fernández Temprano

Agradecimientos

Me gustaría dedicar unas palabras de agradecimiento a aquellas personas y entidades que me han brindado su apoyo a lo largo de la realización de este Trabajo de Fin de Grado.

En primer lugar, al Grupo de Investigación Reconocido "Inferencia con Restricciones" de la Universidad de Valladolid, y especialmente a Miguel Alejandro Fernández Temprano y Cristina Rueda Sabater, por acogerme bajo su seno y enseñarme los fundamentos de la labor investigadora en nuestra universidad. Gracias a que me han transmitido la enorme pasión y dedicación por esta disciplina he podido desarrollar este trabajo (casi) siempre con una sonrisa en la cara.

A mi familia, con especial dedicatoria a mi madre, mi tía, y mi hermana. Mi madre, Beatriz, por apoyarme en esta etapa académica que llega a su fin y por reforzar mi libertad tanto en mis elecciones académicas como personales. A mi hermana, Patricia, por llenar de alegría los días que no tenían tanta; me esforzaré por seguir siendo el referente por el que me tomas. A mi tía, Virginia, por ofrecerse a arreglar conmigo cualquier problema que se presentara, por enseñarme la mejor manera de buscar soluciones, y por ayudarme a comprender que mis límites estaban mucho más allá de lo que podía imaginar. Gracias a todos por ese toque de cariño que todo el mundo debería tener.

A mis compañeros de clase y grandes amigos, Raúl e Irene. Habéis conseguido que el paso por la Universidad haya sido realmente divertido. Su cercanía, sincera amistad e inolvidables momentos han hecho que tanto el desarrollo de este TFG como toda la carrera haya sido de lo más llevadera, incluso en las épocas de exámenes. A Irene, por estar siempre dispuesta a prestar ayuda y compartir todo el material. A Raúl, por hacerme reír hasta en las clases más duras y enseñarme ese lado despreocupado tan necesario hoy en día. Os agradezco de corazón vuestro apoyo, y sabed que yo también estaré cuando me necesitéis.

Sobre todo, a Manuel Jiménez. Gracias por tu paciencia, tu comprensión, tu apoyo incondicional y tu ayuda sea cual sea el momento en el que la pida. Te agradezco que hayas sido una fuente de inspiración tan valiosa y toda la energía que me has transmitido. No olvidaré todas aquellas ocasiones en las que te has preocupado más por mi bienestar que por el tuyo propio.

Por último, me gustaría agradecer a Miguel Alejandro Fernández Temprano por acogerme bajo su tutela en este TFG, por todo el tiempo y apoyo que me ha brindado desde ese momento (y que no ha sido precisamente poco), por estar siempre dispuesto a ayudar, y por darme la oportunidad de seguir aprendiendo cada día.

Índice general

Abstract **7**

Introducción **9**

1. Funciones de R para la regresión isotónica **11**

1.1. Introducción a la regresión isotónica 12

1.2. Algoritmos para la regresión isotónica 13

1.2.1. Algoritmo basado en el máximo minorante convexo 13

1.2.2. Pool-Adjacent-Violators Algorithm 17

1.3. Funciones de R para el cálculo de la regresión isotónica 20

1.4. Modelo para las simulaciones 21

1.5. Análisis preliminar 23

1.5.1. Orden de crecimiento de pavaC 28

1.5.2. Orden de crecimiento de monoreg 30

1.5.3. Orden de crecimiento de intcox.pavaC 33

1.5.4. Orden de crecimiento de isoreg 34

1.5.5. Orden de crecimiento de pava 38

1.5.6. Orden de crecimiento de gpava 39

1.5.7. Comentarios generales 42

1.6. Comportamiento empírico de las funciones de R 43

1.6.1. Comportamiento de isoreg 44

1.6.2. Diferencias entre monoreg, pavaC e intcox.pavaC 46

1.7. Conclusiones 52

2. Cálculo del mejor estimador up-down-up **55**
2.1. Modelo de señal up-down-up 58
2.1.1. Notación y definiciones 58
2.1.2. Regresión isotónica para las señales up-down-up 59
2.1.3. Rendimiento del algoritmo original 61
2.2. Mejoras y depuración del código 62
2.2.1. Influencia de los algoritmos para calcular el PAVA 62
2.2.2. Comprobación temprana de las condiciones de validez 63
2.2.3. Modificación de la búsqueda de mínimos y máximos locales . . . 66
2.2.4. Modificación de los accesos a memoria 69
2.2.5. Mejora con respecto al original 71
2.3. Mejoras derivadas de resultados teóricos 73
2.3.1. Reducción del espacio de búsqueda 73
2.3.2. Comentarios generales y mejora obtenida 77
2.4. Reescritura de la función en C 81
2.4.1. Cambios y consideraciones necesarias 82
2.4.2. Impacto de la reescritura en C 83
2.5. Paralelización 84
2.5.1. Paralelización de una ejecución 85
2.5.2. Paralelización de múltiples ejecuciones 87
2.5.3. Comentarios generales acerca de la paralelización y mejora global 91

Conclusiones **95**

Referencias **97**

Abstract

The proposal of Restricted Inference is to develop more useful and efficient procedures than classical methods, knowing certain information about the parameter of interest. With this new information, we can obtain estimations of this parameter which are more useful, because these restrictions are supposed to be true.

The study of restricted models has many uses, in particular for mixed, semiparametric, discrimination and circular data models [11], [12]. Medical diagnosis and gene expression analysis are examples of applications of these models [19], [20], [21]. In medical diagnosis, knowing that certain biological parameters are decreasing or increasing as a result of a disease, allows us to develop a more efficient methodology to discriminate sick and healthy individuals. Considering gene expression data (the most direct application discussed in this work), the knowledge of a circular order associated with some genes can make a difference in estimating phases of expression. This estimation can be used, among other things, to explain changes occuring in some carcinogenic cells.

For all these reasons, many algorithms that compute this restricted estimation have been developed. However, there are no studies about their efficiency in the literature, which is particularly important nowadays, given the growing complexity of the problems, motivated in great measure by the influence of Big Data.

In Chapter One, we discuss the theoretical basis of isotonic regression, in which several number of procedures are based. Additionally, most used algorithms and dedicated functions in R will be presented. We are specially interested in finding the fastest and most efficient function, because it will be necessary in the next chapter. To do that, we will present a theoretical analysis of two algorithms, as well as an experiment monitoring some R functions.

Chapter Two is dedicated to a methodology developed recently by GIR "Inferencia con Restricciones" (University of Valladolid), which is designed to estimate circular-ordered gene expression. Theoretical basis and procedures of this methodology will be presented. Moreover, we will propose some improvements for these procedures that can be divided in four groups: programming improvements, theoretical improvements of reduction of the search space, programming in C, and parallel computation.

Introducción

La inferencia estadística con restricciones tiene como objetivo elaborar procedimientos que resulten más eficientes y útiles que los métodos clásicos, aprovechando una información a priori que a menudo se conoce sobre los parámetros de interés, para obtener estimaciones en situaciones prácticas donde las restricciones impuestas se verifican.

El estudio de modelos con restricciones tiene numerosos usos, en particular para modelos mixtos y semiparamétricos, modelos de discriminación, y modelos de datos circulares [11], [12]. El problema de diagnóstico médico y análisis de expresiones de genes son algunas de las aplicaciones más utilizadas de estos modelos [19], [20], [21]. En el diagnóstico médico, el conocimiento de que determinados parámetros biológicos disminuyen o aumentan como consecuencia de los síntomas de una enfermedad permite desarrollar una metodología más eficiente para discriminar entre enfermos y sanos. Si trabajamos con expresiones de genes (que por otro lado es la aplicación más directa de la metodología discutida en este trabajo), el conocimiento de un orden circular en que se expresan los genes asociados a ciclos celulares, se utiliza para estimar de mejor forma las fases de expresión. Esta estimación sirve, entre otras cosas, para explicar los cambios observados en células cancerígenas.

Por estos motivos, se han desarrollado varios algoritmos que permiten el cálculo de estos estimadores bajo restricciones. Sin embargo, no existe en la literatura un estudio sobre la eficiencia relativa de estos algoritmos, lo que tiene especial importancia en los momentos actuales dado el incremento en la complejidad de los problemas, motivado en gran parte por el desarrollo del Big Data.

En el Capítulo 1 de esta memoria se exponen los fundamentos teóricos de la regresión isotónica, un pilar fundamental en la estimación bajo restricciones. Además de comentar los algoritmos más utilizados en su cálculo, se realizará un análisis teórico de los mismos y un análisis empírico de algunas funciones de R que se dedican a esta estimación. Interesa especialmente encontrar la más rápida, pues será requerido en el Capítulo 2.

En el Capítulo 2 se expone una metodología desarrollada por el GIR “Inferencia con Restricciones” de la Universidad de Valladolid para la estimación de expresiones de genes con un orden circular. Además de presentar los fundamentos necesarios para la comprensión de la metodología y los procedimientos que se utilizan, se proponen mejoras de diversos tipos para maximizar la eficiencia en la estimación. En concreto, las mejoras introducidas se dividen entre las asociadas a la depuración de la programación, mejoras teóricas de reducción del espacio de búsqueda, impacto de la reescritura de los procedimientos en C, y paralelización del código.

Capítulo 1

Comparación de funciones de R para el cálculo de la regresión isotónica

Una enorme variedad de técnicas estadísticas han resultado útiles en numerosos entornos y aplicaciones, desde campos como la biología y la epidemiología hasta otros tan diferentes como la economía, procesado de señales y termodinámica. Frecuentemente, estas técnicas se pueden mejorar añadiendo cierta información que se conoce a priori, por ejemplo, restricciones de orden de los parámetros de un modelo. Gracias al conocimiento adicional, los modelos pueden resultar más fiables, coherentes, y con un potencial mayor.

La regresión isotónica es una de las técnicas más conocidas de esta rama de estudio. Dado un conjunto de observaciones unidimensionales, el problema es encontrar una secuencia de valores ajustados no decreciente (o no creciente, dependiendo de la formulación del problema) minimizando el error cuadrático medio. Generalmente, las restricciones de orden que se imponen se escriben mediante inecuaciones.

En este capítulo se aborda el estudio de la regresión isotónica dado un orden simple o total. Se presentarán los fundamentos teóricos de la misma, además de mostrar ciertas estrategias para su cálculo. El objetivo fundamental en el primer capítulo es proporcionar una visión acerca de las posibilidades de su cálculo en el entorno de programación *R*. Es especialmente interesante encontrar aquella función que proporcione un cálculo eficiente y rápido, pues este procedimiento será una subrutina en la metodología descrita en el siguiente capítulo.

Para la realización del análisis de las funciones se propone un experimento donde se considerarán distintas situaciones de cálculo de la regresión isotónica, anotando los tiempos de ejecución de varias de las funciones, y elaborando modelos que permitan distinguir unas funciones de otras. Veremos que algunas funciones se comportan mejor en algunos casos pero resultan peores en otros. Tan interesante es encontrar una función rápida, como una que ofrezca versatilidad y se comporte de forma similar ante diferentes entradas.

1.1. Introducción a la regresión isotónica

Consideremos un conjunto finito de puntos $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Supongamos que queremos explicar la variable Y a partir de X. Si es posible suponer que la variable respuesta procede de una distribución normal, es ampliamente conocido el uso de una regresión lineal para modelar la media de Y en función de X.

Sin embargo, en muchas ocasiones es interesante restringir el tipo de regresión que se realiza en base a un conocimiento del dominio del problema. Habitualmente se emplean restricciones basadas en el orden o en la forma de estas funciones de regresión, y son expresadas mediante desigualdades. Este problema se resuelve con las técnicas que proporciona la **regresión isotónica**.

Sea $\boldsymbol{X} = \{x_1, x_2, \ldots, x_n\}$ el conjunto que contiene los n valores de una variable explicativa, w una función de pesos definida sobre $\boldsymbol{X}$ y F una familia de funciones que cumplen alguna restricción. Para una función g cualquiera definida sobre $\boldsymbol{X}$, sea g^* la proyección de mínimos cuadrados de g sobre F. Formalmente, g^* es la solución de

$$\text{Minimizar} \sum_{x \in \boldsymbol{X}} w(x) \left(g(x) - f(x)\right)^2 \text{ sujeto a } f \in F. \tag{1.1}$$

Si la familia de funciones F satisface ciertas condiciones, entonces g^* proporciona una solución que puede diferir significativamente de la estimación de mínimos cuadrados. En caso de que se conozca el conjunto de funciones F para un problema determinado, es conveniente restringir las posibles estimaciones para conseguir la mejor atendiendo a las restricciones reales del problema.

El caso que se abordará en este capítulo es el cálculo de la regresión isotónica dado un **orden simple**. La familia de funciones F se refiere a funciones isotónicas.

Definición 1. Sea el conjunto $\boldsymbol{X} = \{x_1, x_2, \ldots, x_n\}$ sobre el que se define un orden simple $x_1 \lesssim x_2 \lesssim \ldots \lesssim x_n$. Una función f sobre $\boldsymbol{X}$ es isotónica con respecto a este orden si $f(x_1) \leq f(x_2) \leq \ldots \leq f(x_n)$.

Definición 2. Sea g una función sobre $\boldsymbol{X}$. Una función g^* se dice que es la *regresión isotónica de* g con función de pesos w si y solo si g^* es isotónica y minimiza

$$\sum_{x \in \boldsymbol{X}} w(x) \left(g(x) - f(x)\right)^2 \tag{1.2}$$

donde f es cualquier función isotónica definida sobre $\boldsymbol{X}$.

Es posible comprobar que cualquier punto $P = (f(x_1), f(x_2), \ldots, f(x_n))$, donde f es una función isotónica, está dentro de una región del espacio de $\mathbb{R}^n$ definida por un cono convexo, que habitualmente es conocido como el **cono de orden** [2].

Definición 3. Sea V un espacio vectorial y C un subconjunto de V. Se dice que C es un cono si y solo si $\forall x \in C$ y cualquier $\alpha > 0$, $\alpha x \in C$.

Definición 4. Un cono C es un cono convexo si, $\forall x, y \in C$ y cualesquiera escalares $\alpha, \beta > 0$, $\alpha x + \beta y \in C$.

Definición 5. Sea el conjunto $\boldsymbol{X} = \{x_1, x_2, \ldots, x_n\}$ sobre el que se define un orden simple $x_1 \lesssim x_2 \lesssim \ldots \lesssim x_n$, y w una función de pesos definida sobre $\boldsymbol{X}$. Se define el *cono de orden para la regresión isotónica* (*weighted monotone cone*) como el conjunto

$$K^w = \left\{ \boldsymbol{v} \in \mathbb{R}^n : \frac{v^1}{\sqrt{w(x_1)}} \leq \frac{v^2}{\sqrt{w(x_2)}} \leq \ldots \leq \frac{v^n}{\sqrt{w(x_n)}} \right\}$$

Es fácil comprobar que el cono de orden de la definición 5 es realmente un cono y que cumple la propiedad de convexidad. La regresión isotónica de g con respecto al orden simple establecido es la proyección de g sobre este cono de orden.

1.2. Algoritmos para el cálculo de una regresión isotónica dado un orden simple

Tradicionalmente existen dos familias de algoritmos para el cálculo de la regresión isotónica dado un orden simple. En esta sección se presentan los dos tipos que propone *Robertson et al.* (1998) [1], así como un análisis teórico sobre la eficiencia de cada uno de ellos.

1.2.1. Algoritmo basado en el máximo minorante convexo

El primer método que se presenta es también el menos utilizado. Definiremos en primer lugar el máximo minorante convexo, para luego presentar el fundamento teórico del algoritmo, su pseudocódigo y el análisis de la eficiencia del mismo.

Definición 6. El **minorante convexo** de una función real f definida sobre $U \subseteq \mathbb{R}$ es una función convexa cualquiera $f_M(x)$ que verifica $f_M(x) \leq f(x)\ \forall x \in U$.

Definición 7. Sea F_M el conjunto de todos los minorantes convexos. El **máximo minorante convexo** (en adelante, GCM) es la función $f_M^* \in F_M$ que verifica $\forall x \in U$ que $f_M^*(x) \geq f_M(x),\ \forall f_M \in F_M$.

La figura 1.1 muestra un ejemplo de una función que representa un camino aleatorio y el cálculo del correspondiente GCM. Puede encontrar más información y aplicaciones interesantes en las referencias [3].

El uso del GCM para el cálculo de la regresión isotónica se describe a continuación. Consideremos los valores $W_j = \sum_{i=1}^{j} w(x_i)$ y $G_j = \sum_{i=1}^{j} w(x_i)g(x_i)$ para $j = 1, \ldots, n$, donde w es una función de pesos positiva y g una función cualquiera. Sean los puntos $P_0 = (0, 0)$ y $P_j = (W_j, G_j),\ j = 1, \ldots, n$. El gráfico de estos puntos se denomina **diagrama de sumas acumuladas** (CSM, Cumulative Sum Diagram). Sea G^* el GCM del CSM en el intervalo $[0, W_n]$. Debido a la convexidad de G^*, dicha función es derivable por la izquierda en todos los puntos $W_1, \ldots, W_n$. Sea $g^*(x_i)$ la derivada por la izquierda de G^* en el punto W_i. Nótese que para cualquier i, si se verifica que $G^*(W_i) < G_i$ (esto es, el máximo minorante convexo está estrictamente por debajo del diagrama de sumas

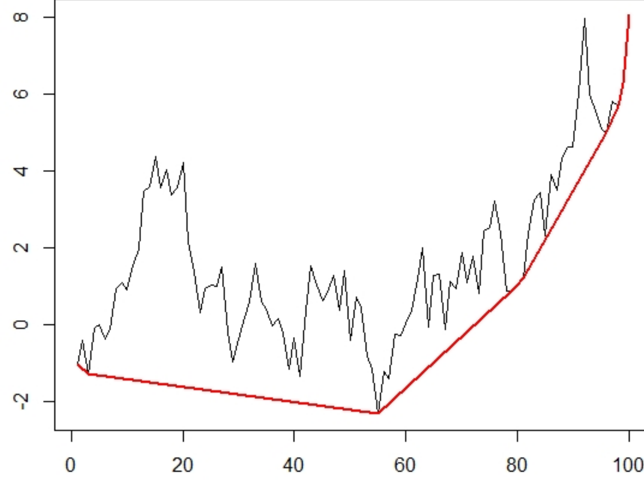


Figura 1.1: Máximo minorante convexo de un camino aleatorio.

acumuladas), la derivada de G^* por la izquierda y por la derecha en W_i es la misma. Formalmente:

$$G^*(W_i) < G_i \implies g^*(x_{i+1}) = g^*(x_i) \tag{1.3}$$

Esta condición es fácilmente verificable en la figura 1.1.

Teorema 1. Si $\boldsymbol{X}$ está definido sobre un orden simple, entonces la derivada por la izquierda g^* del máximo minorante convexo del diagrama de sumas acumuladas es la regresión isotónica de g. Además, si f es isotónica sobre $\boldsymbol{X}$, entonces

$$\sum_{x \in \boldsymbol{X}} w(x)\,(g(x) - f(x))^2 \geq \sum_{x \in \boldsymbol{X}} w(x)\,(g(x) - g^*(x))^2 + \sum_{x \in \boldsymbol{X}} w(x)\,(g^*(x) - f(x))^2 \tag{1.4}$$

y la regresión isotónica es única.

La prueba de este teorema puede encontrarse en [1] (capítulo 1, página 8).

La conclusión inmediata de los resultados anteriores es la creación de un algoritmo para el cálculo de la regresión isotónica, que se reduce al cálculo del GMC de una función. La pendiente del GCM en los distintos puntos constituye la solución buscada. Una vez calculado el diagrama de sumas acumuladas, y partiendo del punto $P_0 = (0, 0)$, es necesario encontrar el siguiente punto $P_k = (W_k, G_k)$ para el cuál la recta que une a estos dos puntos tiene la pendiente más pequeña. Una vez que se conoce el valor de k, es necesario encontrar el siguiente punto, de nuevo, con la pendiente más pequeña. El procedimiento se repite hasta que se ha conseguido el GCM de todos los puntos, de manera que la pendiente por la izquierda en cada punto constituye la regresión isotónica. El pseudocódigo del algoritmo se puede encontrar en la figura 1.2.

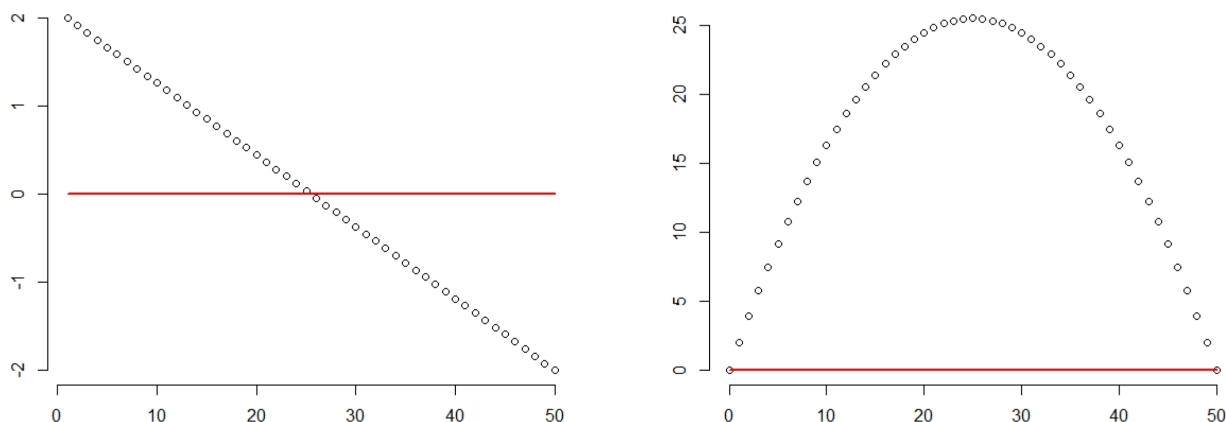
Entrada: $\boldsymbol{X} = \{x_1, \ldots, x_n\}$, función de pesos w y función g
Salida: g^*, regresión isotónica de g

$W_0 \leftarrow 0$
$G_0 \leftarrow 0$
$W_j \leftarrow \sum_{i=1}^{j} w(x_i)$, $j = 1, \ldots, n$
$G_j \leftarrow \sum_{i=1}^{j} w(x_i)g(x_i)$, $j = 1, \ldots, n$
$k \leftarrow 0$
$k_{aux} \leftarrow 0$
do
    $slope \leftarrow inf$
    for $i \in \{k+1, \ldots, n\}$ **do**
        $tmp \leftarrow$ pendiente de la recta que une (W_k, G_k) y (W_i, G_i)
        if $tmp < slope$ **then**
            $slope \leftarrow tmp$
            $k_{aux} \leftarrow i$
        end-if
    end-do
    for $i \in \{k+1, \ldots, k_{aux}\}$ **do**
        $g^*(x_i) \leftarrow slope$
    end-do
    $k \leftarrow k_{aux}$
while $k < n$

Figura 1.2: Pseudocódigo de la función para el cálculo de la regresión isotónica basado en el máximo minorante convexo

El tiempo que este algoritmo deba invertir en el cálculo del GCM depende fuertemente de los datos. Concretamente, depende del número de cambios que se producen en la pendiente del máximo minorante convexo. Consideremos dos ejemplos, correspondientes al mejor caso y al peor caso en cuanto a número de operaciones realizadas.

En el mejor caso no se realiza ningún cambio en la pendiente del máximo minorante convexo y la regresión isotónica es constante. Esto ocurre, por ejemplo, si la secuencia de datos es decreciente cuando se quiere calcular la regresión isotónica restringida a una función creciente. Los datos para una secuencia decreciente del 2 al -2 se representan en la figura 1.3 (a). Si realizamos el diagrama de sumas acumuladas siguiendo los pasos anteriores se obtiene lo que se muestra en la figura 1.3 (b). El cálculo del GCM es evidente a simple vista, y muy rápida con el pseudocódigo presentado. En la primera pasada del bucle se busca unir el primer punto con aquel que proporcione la mínima pendiente, que obviamente resulta ser el último. Con esto concluye el cálculo del GCM, siendo una función constante en el 0, por lo que la regresión isotónica es también constante en el 0. La complejidad temporal de este algoritmo en el mejor caso es $O(n)$.



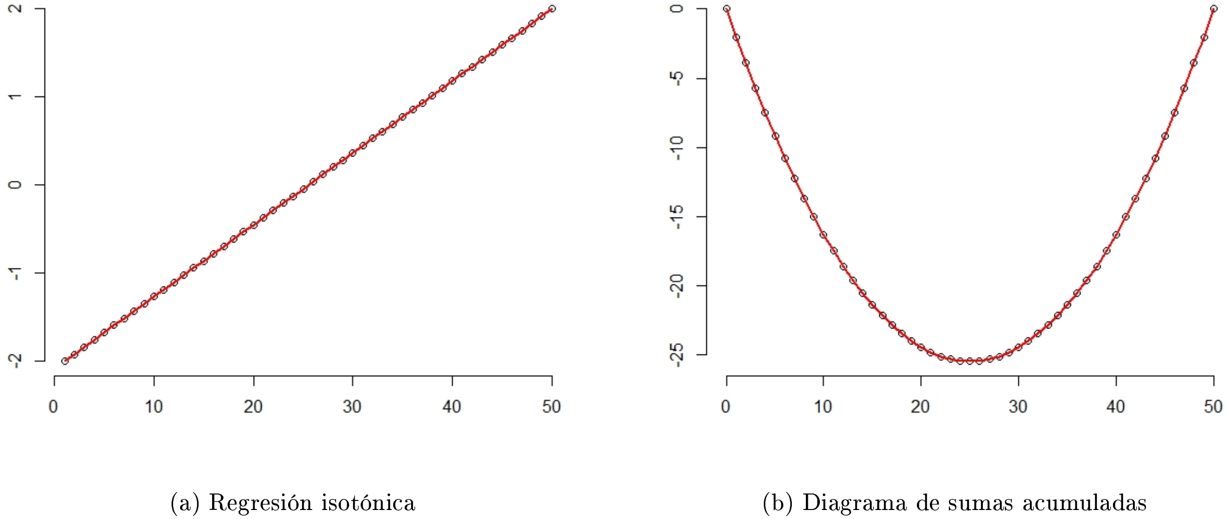
(a) Regresión isotónica

(b) Diagrama de sumas acumuladas

Figura 1.3: Mejor caso en el cálculo del máximo minorante convexo

En el peor caso (figura 1.4), se tiene una secuencia ordenada creciente (g ya es isotónica) y hay tantos cambios de pendiente como puntos. El algoritmo en cada paso tiene que comprobar la pendiente del punto actual con todos los siguientes, aunque siempre resulta ser el inmediato posterior, y el máximo minorante convexo se corresponde con la propia función representada. La regresión isotónica coincide con la propia secuencia creciente, y la complejidad temporal del algoritmo en el peor caso es $O(n^2)$ (la notación $f = O(g)$ quiere decir que f crece no más rápido que g, aunque habitualmente se utiliza para indicar que el crecimiento es del mismo orden).

De este modo, se puede concluir que cualquier cálculo de regresión isotónica basado



(a) Regresión isotónica (b) Diagrama de sumas acumuladas

Figura 1.4: Peor caso en el cálculo del máximo minorante convexo

en el máximo minorante convexo funciona mejor si los datos no presentan un claro patrón creciente (suponiendo un orden simple creciente como restricción) y se asemejan más a un ruido blanco. Si denotamos por p a los distintos valores que puede tomar la regresión isotónica, también llamados conjuntos de nivel, la complejidad temporal para el cálculo del GCM es, en general, $O(pn)$. En el mejor caso $p = 1$; en el peor $p = n$.

1.2.2. Pool-Adjacent-Violators Algorithm

El algoritmo que se presenta en esta sección es el más conocido para el cálculo de la regresión isotónica, denominado comúnmente como PAVA por sus siglas en inglés. De igual forma que antes, presentaremos el fundamento teórico y el pseudocódigo del algoritmo antes de realizar un pequeño análisis sobre la eficiencia del mismo.

Es claro que, a partir del algoritmo basado en el GCM, si para algún i se cumple que $g(x_{i-1}) > g(x_i)$, el cálculo del GCM entre los puntos $[W_{i-2}, W_i]$ mostrará una línea recta. Podemos pensar en modificar el cálculo del GCM por una versión más eficiente donde vayamos detectando las "violaciones" en las restricciones de orden, y llegar a lo que se conoce como PAVA. En cada paso se supone que el siguiente punto constituye la mínima pendiente. Si no se cumple, entonces se vuelve hacia atrás hasta determinar dónde se ha violado la suposición, y se corrige en el mínimo punto detectado.

La regresión isotónica, g^*, provoca una partición de los elementos de X en conjuntos de elementos consecutivos para los cuales g^* es constante dentro de cada conjunto. A estos conjuntos se les llama *conjuntos de nivel* o *level sets*. En cada conjunto de nivel, se puede comprobar que el valor de g^* es la media ponderada de los valores de g aplicados a los elementos del conjunto [1].

El algoritmo PAVA empieza trabajando sobre la función g. Si g es isotónica, entonces es claro que $g^* = g$. En otro caso, debe existir una "violación" donde $g(x_i) > g(x_{i+1})$ para algún i. Cuando se encuentre esto, se reemplazan los valores de g en esos casos por la media ponderada, de forma que $g'(x_i) = g'(x_{i+1}) = \frac{w_i g(x_i) + w_{i+1} g(x_{i+1})}{w_i + w_{i+1}}$. También se actualizan los pesos de las observaciones implicadas, siendo ahora $w'_i = w'_{i+1} = w_i + w_{i+1}$. Con esto se ha reducido en uno el número de cojuntos de nivel, puesto que ahora se tiene $g(x_1) \leq g(x_2) \leq \ldots \leq g(x_{i-1}) \leq g'(x_i) = g'(x_{i+1}) \leq g(x_{i+2}) \leq \ldots \leq g(x_n)$. Si este nuevo conjunto de valores es isotónico entonces el proceso termina, pero en caso contrario se repite encontrando un nuevo par de valores que no cumplen la restricción de orden hasta que sí lo sea.

El PAVA se basa entonces en el siguiente teorema que facilita el cálculo [4].

Teorema 2. Para una solución óptima, si $g(x_i) > g(x_{i+1})$, entonces $g^*(x_i) = g^*(x_{i+1})$.

Demostración. Supongamos por el contrario que $g^*(x_i) < g^*(x_{i+1})$. Escogemos un ϵ lo suficientemente pequeño, de manera que

$g^*_{new}(x_i) = g^*(x_i) + w_{i+1}\epsilon$

$g^*_{new}(x_{i+1}) = g^*(x_{i+1}) - w_i\epsilon$

Esto hace decrecer la suma $\sum_{i=1}^{n} w_i(g(x_i) - g^*(x_i))^2$ sin violar las restricciones de orden. La solución inicial no es óptima y llegamos a una contradicción. □

Como $g^*(x_i) = g^*(x_{i+1})$, la solución pasa por combinar los puntos $(w_i, g(x_i))$ y $(w_{i+1}, g(x_{i+1}))$ en un nuevo punto $(w'_i, g'(x_i)) = (w_i + w_{i+1}, \frac{w_i g(x_i) + w_{i+1} g(x_{i+1})}{w_i + w_{i+1}})$. Sin embargo, es posible que después de combinar estos dos puntos, se viole la restricción $g(x_{i-1}) \leq g'(x_i)$. En este caso, el punto $(w'_i, g'(x_i))$ debe combinarse de nuevo con el punto $(w_{i-1}, g(x_{i-1}))$, y así sucesivamente hasta que no se viole ninguna restricción.

El PAVA es entonces un algoritmo iterativo en el que en cada paso se considera un punto más allá. La función termina cuando se han considerado todos los puntos. En cada iteración, el nuevo punto se añade sin más si no viola la restricción de orden, o se ejecuta el procedimiento hacia atrás descrito en el paso anterior hasta que las restricciones de orden quedan garantizadas. Al final de la ejecución, existirán distintos conjuntos de nivel $a_1, a_2, ..., a_j$, que indican los j valores distintos que forman el cálculo de la regresión isotónica. El pseudocódigo del PAVA se encuentra en la figura 1.5.

El significado de las variables que forman el pseudocódigo de la figura 1.5 es:

- j es el número de conjuntos de nivel.
- $\boldsymbol{a}$ es el vector que contiene los conjuntos de nivel.
- $\boldsymbol{w'}$ es el vector con los pesos de cada conjunto de nivel
- $\boldsymbol{S}$ contiene la información relativa a qué conjunto de nivel pertenece cada punto. El conjunto de nivel k empieza en el punto $S_{k-1} + 1$ y termina en el punto S_k.
- g^* es la función buscada.

Entrada: $\boldsymbol{X} = \{x_1, \ldots, x_n\}$, función de pesos w y función g
Salida: g^*, regresión isotónica de g

$a_1 \leftarrow g(x_1)$
$w'_1 \leftarrow w_1$
$j \leftarrow 1$
$S_0 \leftarrow 0$
$S_1 \leftarrow 1$
for $i = 2, 3, ..., n$ **do**
  $j \leftarrow j + 1$
  $a_j \leftarrow g(x_i)$
  $w'_j \leftarrow w_i$
  while $j > 1$ and $a_j < a_{j-1}$ **do**
    $a_{j-1} \leftarrow \frac{w'_j a_j + w'_{j-1} a_{j-1}}{w'_j + w'_{j-1}}$
    $w'_{j-1} \leftarrow w'_j + w'_{j-1}$
    $j \leftarrow j - 1$
  end-do
  $S_j \leftarrow i$
end-do
for $k = 1, 2, ..., j$ **do**
  for $l = S_{k-1} + 1, ..., S_k$ **do**
    $g^*(x_l) = a_k$
  end-do
end-do

Figura 1.5: Pseudocódigo del algoritmo PAVA, extraído de [4]

El análisis del mejor y el peor caso es más sencillo en el PAVA. En todos los casos en los que se ejecuta se hace un recorrido por todos los valores comprobando las restricciones de orden, por lo que sea cual sea el vector de datos de entrada, el algoritmo tiene una complejidad temporal $\Omega(n)$ (la notación $f = \Omega(g)$ significa que f crece no más despacio que g). Adicionalmente se realizarán más operaciones que son proporcionales al número de violaciones en las restricciones de orden. Recordemos que en cada paso en el que se produce una violación, dos valores se "fusionan" en uno solo. El peor caso corresponde a la regresión isotónica en la que sólo hay un conjunto de nivel y se han producido $n - 1$ "fusiones", el máximo número posible.

El mejor caso del PAVA se corresponde con aquella situación en la que g ya es isotónica y los valores están ordenados, sin ninguna violación en las restricciones de orden. El algoritmo en este caso tiene una complejidad $O(n)$, en contraposición a la complejidad cuadrática que maneja el GCM en la misma situación. El peor caso corresponde al máximo número de violaciones en las restricciones de orden, por ejemplo, con una secuencia decreciente, donde ahora se maneja una complejidad temporal de $O(2n)$.

Es especialmente llamativo que el mejor caso de uno de los algoritmos sea el peor del otro y viceversa. Sin embargo, el basado en el máximo minorante convexo tiene una complejidad cuadrática en su peor caso, cuando esto no ocurre en el peor caso del PAVA. Podría estudiarse la rentabilidad del algoritmo basado en GCM en el caso en

que los datos no presenten una tendencia creciente, pues es previsible que el cálculo sea un poco más rápido que con el PAVA.

1.3. Funciones de R para el cálculo de la regresión isotónica

En este apartado se comentarán algunas de las funciones más conocidas que permiten el cálculo de la regresión isotónica en R [5]. El objetivo final de este capítulo es la realización de un análisis empírico que permita concluir si alguna de estas funciones, que son las que pasarán a formar parte del análisis, es significativamente mejor que otra en cuanto al tiempo empleado en el cálculo.

Aquellas que se tienen en cuenta se describen a continuación.

1. Función *isoreg*, que forma parte del paquete base de R. Aunque es una función de R, el contenido de la misma no es más que un preparatorio de los argumentos de entrada para el verdadero núcleo de la función, que está escrito en C. La función original no admitía regresión isotónica con pesos positivos, por lo que la misma se ha extendido para añadir dicha posibilidad. El algoritmo en el que se basa es el del máximo minorante convexo original.

2. Función *pava*, que forma parte del paquete *Iso* [6]. De la misma manera que antes, es una función de R, pero utiliza la interfaz de acceso a FORTRAN, donde se realizan los cálculos principales. Según la ayuda original del autor, el algoritmo en el que se basa es el clásico PAVA.

3. Función *intcox.pavaC*, que forma parte del paquete *intcox* [7]. De nuevo es una función que ha sido escrita en C. Debido a su enorme simpleza tiene ciertas limitaciones, pues sólo ofrece la posibilidad del cálculo con restricciones de orden creciente y la especificación de pesos. Según el manual, está basada en el algoritmo ICM (*Iterative Convex Minorant Algorithm*). Este es el nombre que recibe el procedimiento que calcula el máximo minorante convexo de la misma manera que PAVA, determinando las violaciones sobre la marcha, y evitando así la comprobación repetida y el crecimiento cuadrático. La complejidad temporal del algoritmo ICM es la misma que para PAVA.

4. Función *monoreg*, que forma parte del paquete *fdrtool* [8]. Es una extensión alternativa para la utilización de pesos que la que se utiliza en *isoreg*. También está implementada en C y según el autor realiza el cálculo del máximo minorante convexo, aunque no especifica qué variante.

5. Función *gpava*, que forma parte del paquete *isotone* [9]. Es la más completa de las tres y, entre otras posibilidades, ofrece la utilización de métricas no euclídeas, el cálculo para problemas de regresión isotónica general, o la minimización de otras funciones diferentes a las aquí consideradas. Está escrita íntegramente en R y basada en el PAVA original.

6. Función *pavaC*, que no forma parte de ningún paquete, sino que ha sido diseñada desde cero por el autor de este trabajo como alternativa a las anteriores, basándose

en el pseudocódigo mostrado en la figura 1.5. Tal y como su nombre indica, implementa el algoritmo PAVA pero a través de una llamada a un programa escrito en C.

1.4. Modelo para las simulaciones

Consideramos un modelo de señal más ruido al que poder aplicar el algoritmo correspondiente del cálculo de la regresión isotónica. Este modelo se describe como

$$\mathbf{Y} = \boldsymbol{\mu} + \boldsymbol{\epsilon}$$

donde $\boldsymbol{\mu}$ es un parámetro que pertenece al cono de orden, es decir, que verifica las restricciones $\mu_1 \leq \mu_2 \leq ... \leq \mu_n$, y $\boldsymbol{\epsilon}$ es el ruido.

El experimento que se propone en esta sección tiene como objetivo la comparación de los tiempos de ejecución en diferentes situaciones para cada uno de las funciones de R especificadas. Los factores y variables que se consideran en este experimento son los siguientes.

- Valor de la media real $\boldsymbol{\mu}$.
- La existencia o no de pesos.
- La cantidad de ruido que se incluye en los datos.
- El tamaño del conjunto $\boldsymbol{X} = \{x_1, x_2, \ldots, x_n\}$ sobre el que se define el orden simple.
- La función de R empleada en el cálculo.

Adicionalmente, se considera un número de repeticiones de cada una de las situaciones base donde se hayan especificado los valores de los parámetros mencionados.

Para la generación del parámetro $\boldsymbol{\mu}$ se consideran dos posibles situaciones: que el valor de $\boldsymbol{\mu}$ sea siempre $\mathbf{0}$, o se genere aleatoriamente pero verificando $\mu_1 < \mu_2 < ... < \mu_n$ (estrictamente creciente). El caso en el que $\boldsymbol{\mu} = \mathbf{0}$ es de especial interés para estudiar la influencia del extremo del cono de orden en el cálculo de la regresión isotónica. Por otro lado, cuando la media real se considera creciente, se genera una muestra aleatoria $X_1, X_2, ..., X_n$ que proviene de una distribución normal con media 0 y desviación estándar 1. Se toma $\boldsymbol{\mu} = (X_{(1)}, X_{(2)}, ..., X_{(n)})$, el estadístico ordenado de la muestra generada. De esta manera se garantiza que $\boldsymbol{\mu}$ cumpla las restricciones de orden.

Por otro lado, el experimento se repite tanto para la situación donde se consideran pesos como en la que no. Siempre que se haga referencia a simulaciones con pesos, éstos son generados de forma aleatoria siguiendo el procedimiento descrito a continuación. Sea $W_1, W_2, ..., W_n$ una muestra aleatoria que proviene de una distribución $\mathcal{N}(0, 100)$. El vector de pesos se construye como $\mathbf{W} = (|W_1|, |W_2|, ..., |W_n|)$. Cuando no hay pesos, es equivalente a realizar el cálculo tomando $\mathbf{W} = \mathbf{1}_n$.

El vector $\boldsymbol{\epsilon}$ que contiene el ruido se genera con una distribución normal de media 0 y varianza σ, de manera que ϵ_i sigue una distribución $\mathcal{N}(0, \sigma^2)$. Cada ϵ_i es independiente de ϵ_j cuando $j \neq i$. Se consideran tres posibles niveles para el factor ruido:

- Si se considera "poco" ruido, se toma $\sigma = 0.05$.
- Si se considera una cantidad "intermedia" de ruido, $\sigma = 1$.
- Si hay "mucho" ruido, $\sigma = 100$. Se espera que esta situación se asemeje al caso $\boldsymbol{\mu} = \mathbf{0}$, independientemente del valor real de $\boldsymbol{\mu}$.

El vector $\mathbf{Y}$ así generado verifica

$$Y_i \sim \mathcal{N}(\mu_i, \sigma^2)$$

para $i = 1, 2, ..., n$

El tamaño de este vector, n, también entra como variable numérica. En las siguientes secciones, donde se explican cada uno de los dos experimentos desarrollados, se acotará el tamaño considerado. Las funciones de R que se consideran han sido descritas en la sección 1.3. El número de repeticiones de cada experimento base también será variable. La variable respuesta será el tiempo dedicado a la ejecución, que se pretende explicar utilizando las anteriores.

En la figura 1.6 puede verse un ejemplo en el que se ha generado una nube de puntos con las instrucciones explicadas en esta sección, con $n = 500$ y cantidad intermedia de ruido. El par (μ_i, y_i) representa el punto i-ésimo de la nube de puntos. Se muestra también la regresión isotónica obtenida. A efectos de la notación utilizada anteriormente, puede tomarse como $\boldsymbol{X} = \{1, 2, \ldots, n\}$ y $g(x_i) = y_i$

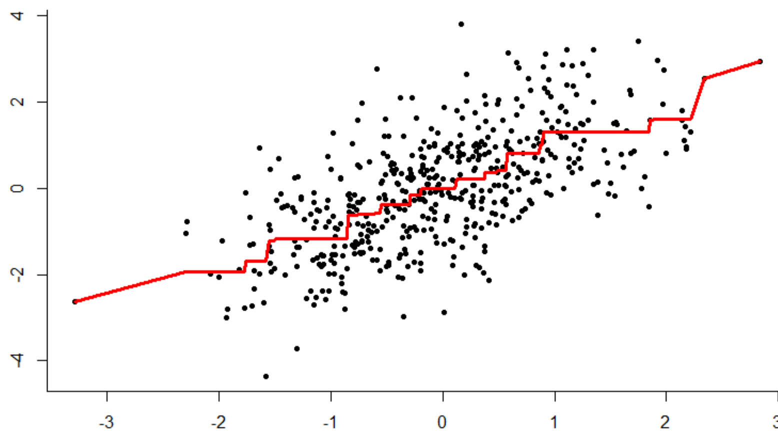


Figura 1.6: Ejemplo de nube de puntos generada con $n = 500$ y $\sigma = 1$.

Durante la ejecución de las funciones de R para determinar el tiempo de ejecución (concretamente, *sys.time()*), se han detectado ciertas anomalías que producen perturbaciones en las medidas. Debido a la ejecución de ciertas tareas del sistema y de los procesos que crea R, de vez en cuando se alteran las medidas reales introduciendo una perturbación que es muy difícil de detectar y eliminar. Por ejemplo, R invoca de vez en cuando a un procedimiento de *recolección de basura* que elimina el espacio asignado que no va a ser utilizado más. Esto consume, evidentemente, ciertos recursos y puede llevar algún tiempo. Habitualmente se realiza en paralelo mientras se llevan a cabo los cálculos principales, por lo que si una medida está distorsionada es probable que otra que se haya tomado en un instante temporal cercano también lo esté.

Puesto que no es posible detectar cuáles de estas medidas tienen este añadido temporal, el experimento se aleatoriza lo máximo posible para evitar que este problema afecte sólo a una combinación de factores concreta. Inicialmente se crea un conjunto de datos, reservando espacio para introducir los valores de todos los experimentos base. A continuación, se escoge aleatoriamente una entrada del conjunto de resultados, se mide el tiempo para la combinación de factores de dicha entrada, y se anota. De entre todas las entradas que aún estén vacías se repite el resultado hasta que el experimento se haya completado. Una descripción detallada del procedimiento puede verse en la figura 1.7.

Todos los experimentos han sido realizados en un ordenador cuyas características se listan a continuación.

- 2 procesadores idénticos (Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, Intel Corp.), cada uno de ellos con 16 núcleos físicos y 32 núcleos lógicos, lo que da lugar a la posibilidad de ejecutar hasta 64 tareas simultáneamente (64 bits).
- Caché de nivel 1 de 1 MiB.
- Caché de nivel 2 de 16 MiB.
- Caché de nivel 3 de 22 MiB.
- 256 GiB de memoria principal DDR4.
- Sistema operativo CentOS.

1.5. Análisis preliminar

En un primer acercamiento nos queremos preguntar si existen algunas funciones cuyos tiempos de ejecución se distancien tanto de las demás, que no merezcan ser tenidas en cuenta en un análisis más minucioso. A este nivel nos preguntamos de qué orden es el crecimiento temporal en cuanto al tamaño, sin preocuparnos demasiado por los demás factores.

Los parámetros que se tienen en cuenta en este experimento son:

- Primeramente, tamaños desde 1e+4 hasta 1e+5, con un paso de 1e+4. Además, también se considera una secuencia desde 1e+5 hasta 1e+6 con un paso de 1e+5.

Para cada repetición **hacer**
  Para cada tamaño considerado **hacer**
    Para cada valor de $\boldsymbol{\mu}$ **hacer**
      Para cada valor del factor "ruido", **hacer**
        Para cada posibilidad de pesos **hacer**
          Para cada función de R **hacer**
            Crear una entrada en el conjunto de resultados
          Fin-para
        Fin-para
      Fin-para
    Fin-para
  Fin-para
Fin-para
Para cada entrada del conjunto de resultados, elegidas con un orden aleatorio, **hacer**
  Obtener los parámetros $\boldsymbol{\mu}$, tamaño, ruido, pesos y algoritmo de la entrada actual
  Si $\boldsymbol{\mu}$ "creciente", **entonces**
    $\boldsymbol{\mu} \leftarrow (X_{(1)}, ..., X_{(n)}), \quad X_i \rightsquigarrow \mathcal{N}(0,1)$
  si no
    $\boldsymbol{\mu} \leftarrow \mathbf{0}$
  Fin-si
  Si poco ruido, **entonces**
    $\epsilon_i \rightsquigarrow \mathcal{N}(0, 0.05^2), \quad \forall i \in \{1, \ldots, n\}$
  Si ruido normal, **entonces**
    $\epsilon_i \rightsquigarrow \mathcal{N}(0, 1), \quad \forall i \in \{1, \ldots, n\}$
  Si mucho ruido, **entonces**
    $\epsilon_i \rightsquigarrow \mathcal{N}(0, 100^2), \quad \forall i \in \{1, \ldots, n\}$
  Fin-si
  $\boldsymbol{y} \leftarrow \boldsymbol{\mu} + \boldsymbol{\epsilon}$
  Si no hay pesos, **entonces**
    $\boldsymbol{w} \leftarrow \mathbf{1}$
  Si hay pesos **entonces**
    $W_i \rightsquigarrow \left|\mathcal{N}(0, 100^2)\right|, \quad \forall i \in \{1, \ldots, n\}$
  Fin-si
  Realizar el cálculo de la regresión isotónica con la función actual
  Anotar el tiempo empleado
Fin-para

Figura 1.7: Pasos que se realizan en todos los experimentos para explicar el tiempo de ejecución.

- Todas las funciones de R descritas en la sección 1.3.
- El valor de $\boldsymbol{\mu}$ siempre es $\mathbf{0}$.
- No se consideran pesos.
- La cantidad de ruido es intermedia.

- Se repite el experimento 100 veces para las funciones *pavaC*, *intcox.pavaC*, *isoreg* y *monoreg*. Sólo se realiza una repetición en el caso de *pava* y *gpava*, ya que el cálculo requiere mucho tiempo, tal y como se verá a continuación.

En la tabla 1.1 se encuentra la media del tiempo invertido (en segundos) en el cálculo de la regresión isotónica para cada algoritmo y tamaño considerado. En el caso de los algoritmos *pava* y *gpava*, dado su tiempo de ejecución notablemente más elevado, la media se corresponde con la única realización del experimento. Además, para el caso de *gpava*, el experimento no está completo; falta el tiempo invertido para algunos tamaños.

Tamaño	*gpava**	*intcox.pavaC*	*isoreg*	*monoreg*	*pava**	*pavaC*
10000	2.364	0.001	0.000	0.001	0.056	0.001
20000	9.083	0.001	0.001	0.001	0.220	0.001
30000	20.920	0.001	0.001	0.002	0.493	0.002
40000	37.265	0.001	0.002	0.002	0.874	0.002
50000	58.828	0.002	0.002	0.003	1.368	0.003
60000	85.913	0.002	0.002	0.004	1.967	0.003
70000	118.644	0.003	0.003	0.005	2.672	0.004
80000	158.665	0.003	0.003	0.006	3.492	0.004
90000	204.762	0.003	0.004	0.006	4.415	0.005
1e+5	257.508	0.004	0.004	0.007	5.456	0.005
2e+5	1154.712	0.008	0.008	0.016	21.799	0.011
3e+5	2955.243	0.011	0.012	0.026	50.484	0.016
4e+5	6989.690	0.015	0.016	0.034	87.554	0.021
5e+5	9905.457	0.018	0.021	0.043	136.648	0.026
6e+5	12625.648	0.022	0.026	0.052	196.564	0.032
7e+5	19069.481	0.026	0.029	0.063	267.599	0.037
8e+5	NA	0.030	0.035	0.071	349.467	0.042
9e+5	NA	0.033	0.040	0.082	442.304	0.048
1e+6	NA	0.037	0.042	0.088	546.132	0.053

Tabla 1.1: Tiempo medio (en segundos) invertido en el cálculo de la regresión isotónica en función del tamaño para los parámetros fijados

Adicionalmente, en la tabla 1.2 se muestra la desviación estándar del tiempo invertido en aquellos casos donde sí hay repeticiones.

A la vista de estos datos se puede concluir que los tiempos más razonables se tienen en las funciones *pavaC*, *intcox.pavaC*, *isoreg* y *monoreg*, sin aparentes grandes diferencias entre sí. La función *pava* sí se distancia de las demás a medida que los tamaños aumentan, además de que parece tener un crecimiento no lineal. Se estima que es unas 6200 veces más lenta que la que presenta un mayor tiempo en el primer grupo de 4 para el máximo tamaño considerado, pues el tiempo invertido por *monoreg* es de apenas 90 milisegundos cuando *pava* tarda algo más de 9 minutos.

Peor aún es el caso de *gpava*, pues en un tamaño de vector de 5e+5 el tiempo invertido es de aproximadamente 5 horas y media, mientras que *pava* invierte 4 minutos

Tamaño	*intcox.pavaC*	*isoreg*	*monoreg*	*pavaC*
10000	0.0005	0.0005	0.00046	0.0005
20000	0.00034	0.00048	0.00044	0.00035
30000	0.0004	0.00047	0.00027	0.00046
40000	0.0005	0.00051	0.00049	0.00039
50000	0.00031	0.0005	0.00014	0.00045
60000	0.00043	0.00057	0.00047	0.00044
70000	0.00046	0.00065	0.00046	0.0004
80000	0.00026	0.00076	0.0005	0.00045
90000	0.00045	0.0008	0.00034	0.00037
1e+5	0.00046	0.0009	0.00022	0.00047
2e+5	0.00052	0.00168	0.00036	0.00049
3e+5	0.00029	0.00272	0.0004	0.00044
4e+5	0.00041	0.00336	0.00029	0.00046
5e+5	0.00046	0.00423	0.00053	0.00058
6e+5	0.00034	0.00518	0.00046	0.00065
7e+5	0.0004	0.00619	0.00196	0.00109
8e+5	0.00064	0.00718	0.00224	0.00093
9e+5	0.00033	0.00826	0.00109	0.00094
1e+6	0.00042	0.00885	0.00131	0.00098

Tabla 1.2: Desviación típica del tiempo invertido en el cálculo de la regresión isotónica en función del tamaño para los parámetros fijados

y medio y *monoreg* 63 milisegundos. Esto nos llevará a descartar las funciones *pava* y *gpava* en secciones posteriores.

En cuanto a la tabla que muestra las desviaciones típicas, es especialmente llamativo el caso de *isoreg*, donde se comprueba que aumentan bastante a medida que se consideran tamaños mayores. Algo parecido ocurre con *monoreg*, aunque no es tan significativo, y ligeramente con *pavaC*.

En la figura 1.8 se encuentran los tiempos de ejecución de las seis funciones en términos del tamaño. Debido a las enormes diferencias entre ellas, se representa el tiempo en escala logarítmica. Es claramente apreciable como las funciones *gpava* y *pava* no merecen una discusión mayor sobre su eficiencia.

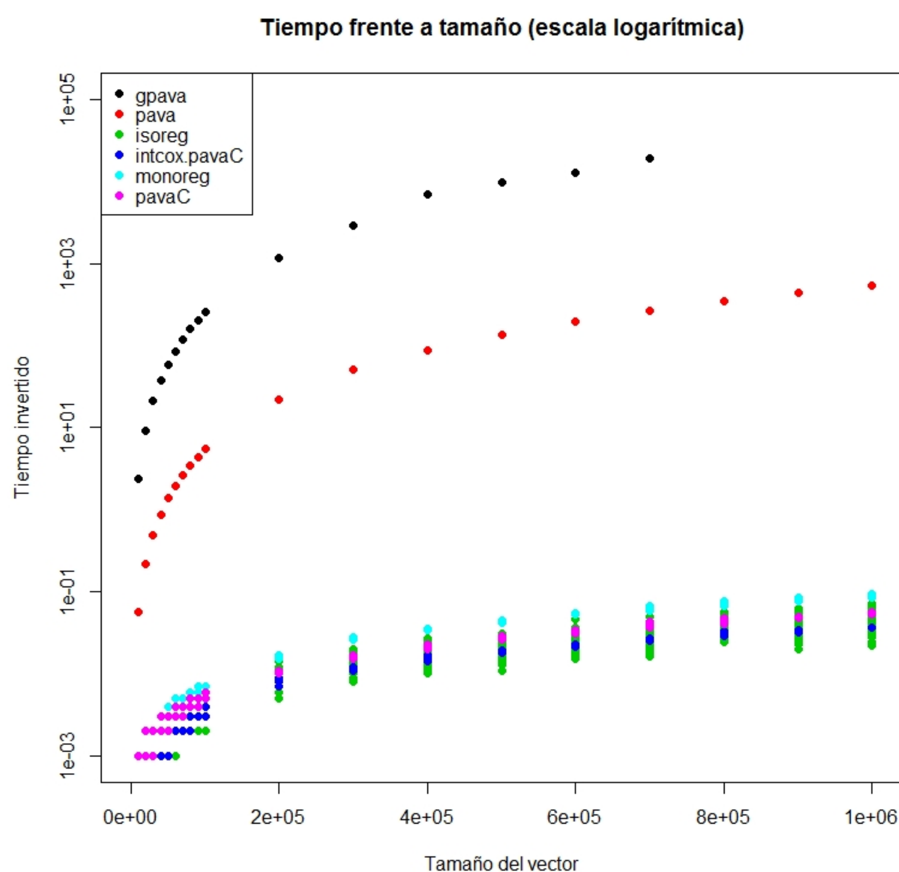


Figura 1.8: Tiempo de ejecución de las seis funciones en términos del tamaño del vector de entrada (escala logarítmica)

Con el objetivo de determinar empíricamente el orden de crecimiento del tiempo en función del tamaño, en las siguientes subsecciones se ajustan algunos modelos por separado para las seis funciones. Aunque no se indique explícitamente, en algunos casos se han detectado ciertas anomalías en las medidas por los motivos ya expuestos (procesamiento de tareas adicionales del sistema). Dichas medidas han sido reemplazadas por

otras nuevas o eliminadas si ya había suficientes datos.

1.5.1. Orden de crecimiento de pavaC

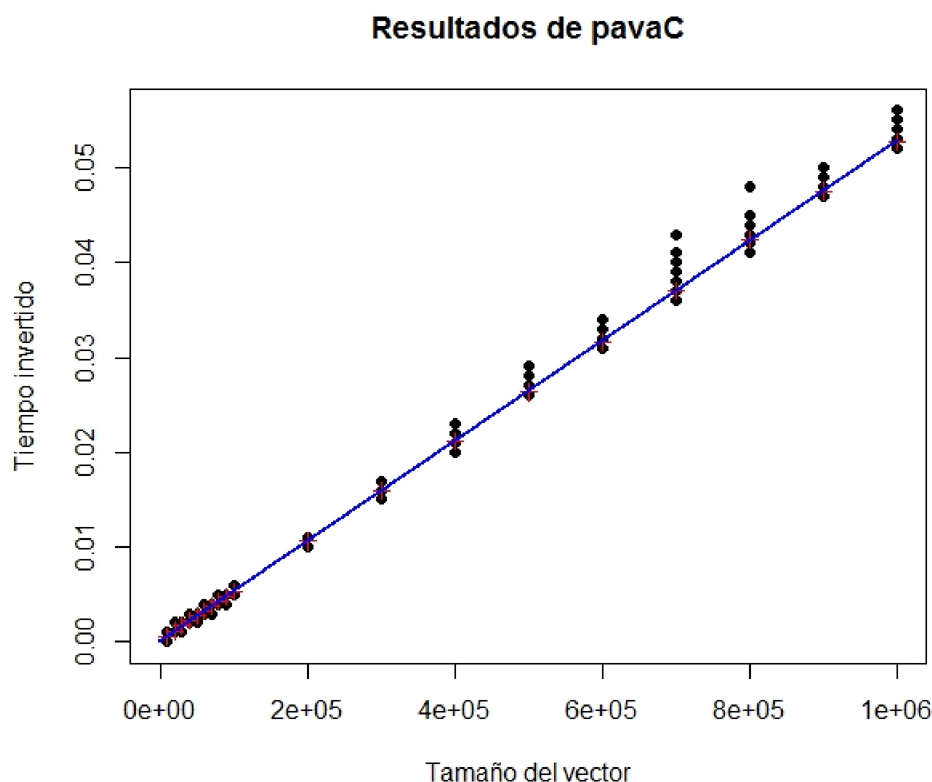


Figura 1.9: Tiempos de ejecución de pavaC en función del tamaño para el análisis preliminar, junto con la recta de regresión

En la figura 1.9 se muestran los tiempos de ejecución obtenidos para la función *pavaC* junto con una recta de regresión simple para un modelo normal. También se representa con una cruz el valor de la media.

Puede comprobarse que, bajo las suposiciones de un modelo normal, es perfectamente asumible que el tiempo presenta un orden lineal, y que corresponde con una buena implementación del algoritmo PAVA. La recta de regresión está muy próxima a los valores de las medias reales, y evidentemente todos los coeficientes son significativos en este modelo.

La aplicación de un modelo en el que se introduce un término cuadrático para el tamaño no resulta rentable en términos del AIC, así como tampoco es significativamente distinto de cero el coeficiente que lo acompaña. Además, la realización de un test de falta de ajuste apoya de nuevo el modelo lineal. No obstante, no es posible tomar como ciertas las hipótesis de un modelo lineal (figura 1.10).

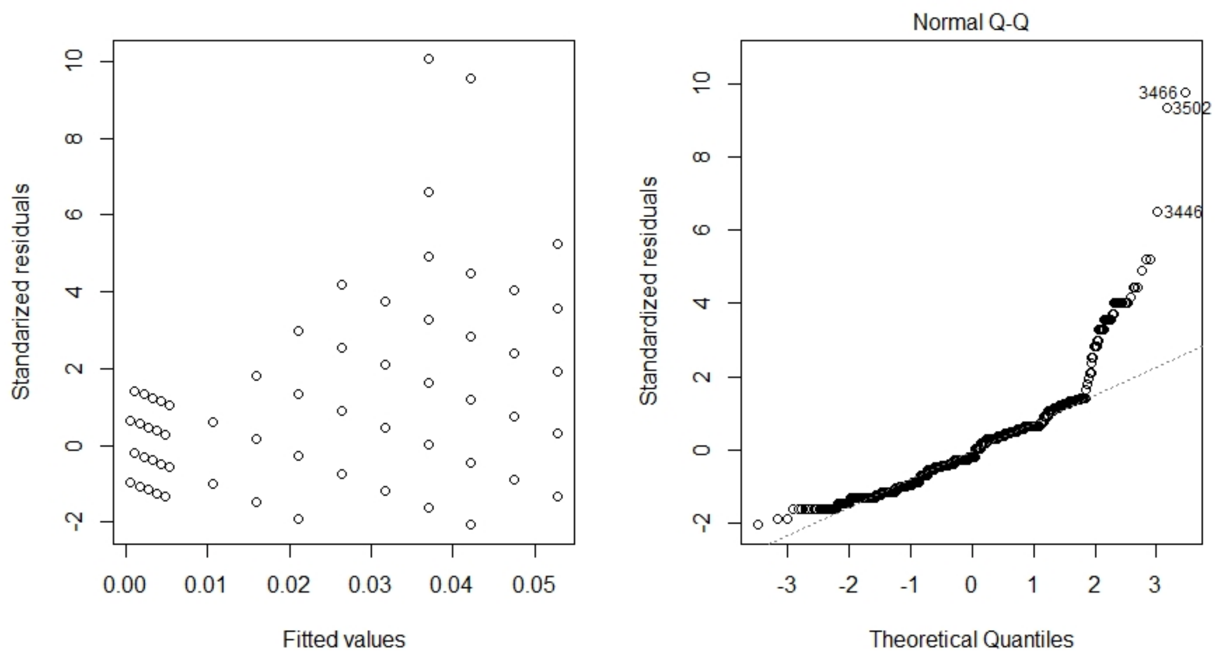


Figura 1.10: A la izquierda, gráfico de residuales frente a los valores predichos para el análisis preliminar del pavaC. A la derecha, el qqplot de normalidad.

Tras un análisis de la verosimilitud perfil para determinar la mejor transformación de *boxcox* se revela que el intervalo de confianza contiene el valor $\lambda = 1$ y no es necesario hacer ninguna transformación. Algunos modelos robustos o lineales generalizados han sido ajustados, como una eliminación de *outliers*, un modelo con respuesta Gamma o alguno basado en la cuasiverosimilitud, pero ninguno de ellos resulta mejor que el normal en términos de residuos o del AIC.

Por todos estos motivos, el modelo final elegido para explicar el tiempo es el modelo lineal normal. Si bien es cierto que las violaciones de las hipótesis del modelo normal pueden afectar a las estimaciones y, sobre todo, a las varianzas estimadas de los parámetros del modelo, el mayor interés en este análisis preliminar es comprobar el orden de crecimiento del tiempo. El p-valor que indica que el coeficiente lineal es significativo es menor que 2e−16, así que aún en estas condiciones se concluye lo mismo. Parece lo mejor asumir que el tiempo depende linealmente del tamaño y que la introducción de un término cuadrático no es necesario.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.00007220	0.00001954	3.69543621	0.00022573
tamanyo	0.00000005	0.00000000	1219.70819842	0.00000000

Tabla 1.3: Resumen del modelo ajustado para *pavaC*

El modelo lineal normal proporciona la siguiente estimación del tiempo empleado en función del tamaño:

$$t_1(n) = 5.274\text{e}{-8} \cdot n + 7.22\text{e}{-5}$$

1.5.2. Orden de crecimiento de monoreg

En cuanto a la función *monoreg*, su crecimiento también es aparentemente lineal. Igual que antes, varios modelos han sido ajustados para valorar el efecto del tamaño en la explicación del tiempo.

A pesar de que se han probado ajustes polinómicos de hasta grado 3 y todos los coeficientes son significativos, una representación gráfica revela que apenas hay diferencias en el tramo representado entre los diferentes modelos. De hecho, si el ajuste se realiza únicamente con las medias, el único parámetro significativo es el lineal.

El gráfico de residuales y el qqplot muestran algo parecido a lo que ocurría en el caso *pavaC* (figura 1.12). No se puede suponer que la varianza es constante en todos los experimentos, ni tampoco normalidad. No obstante, en este caso, sí hay falta de ajuste. El ajuste de otros modelos lineales generalizados no aporta ninguna luz nueva a lo que ya se conocía.

Este modelo explica peor los datos al compararlo con el modelo de *pavaC*, pero como de nuevo lo único que nos interesa a este nivel es determinar el orden de crecimiento, parece razonable suponer linealidad por el momento. Recordemos que este algoritmo está basado en el GCM y que la ayuda no esclarece el tipo de algoritmo utilizado, por

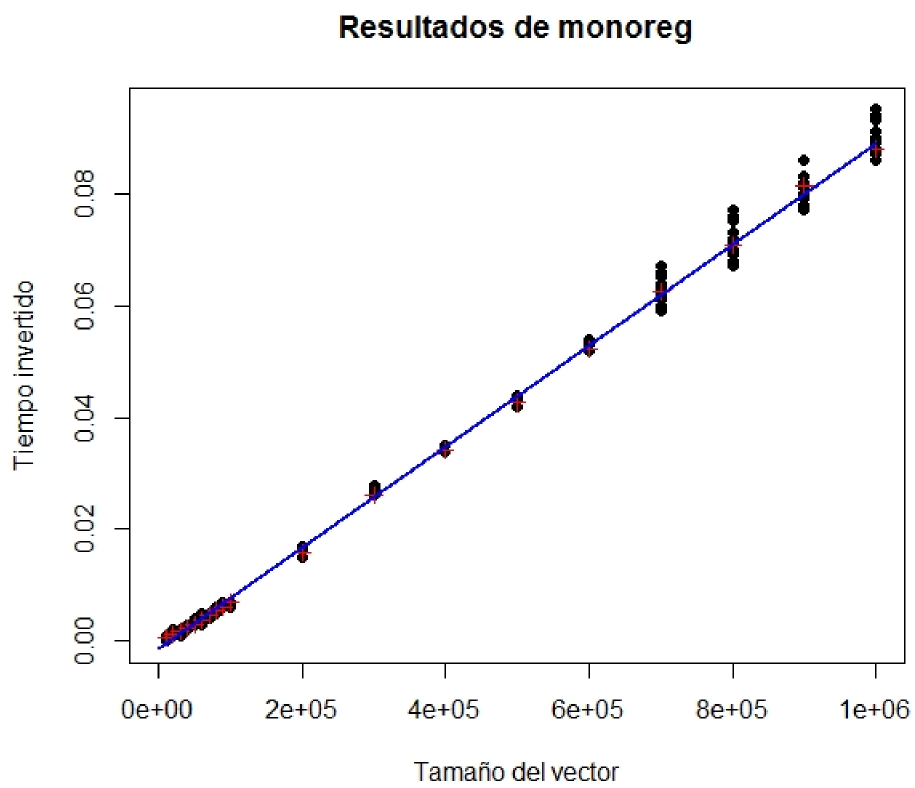


Figura 1.11: Tiempos de ejecución de monoreg en función del tamaño para el análisis preliminar, junto con la recta de regresión

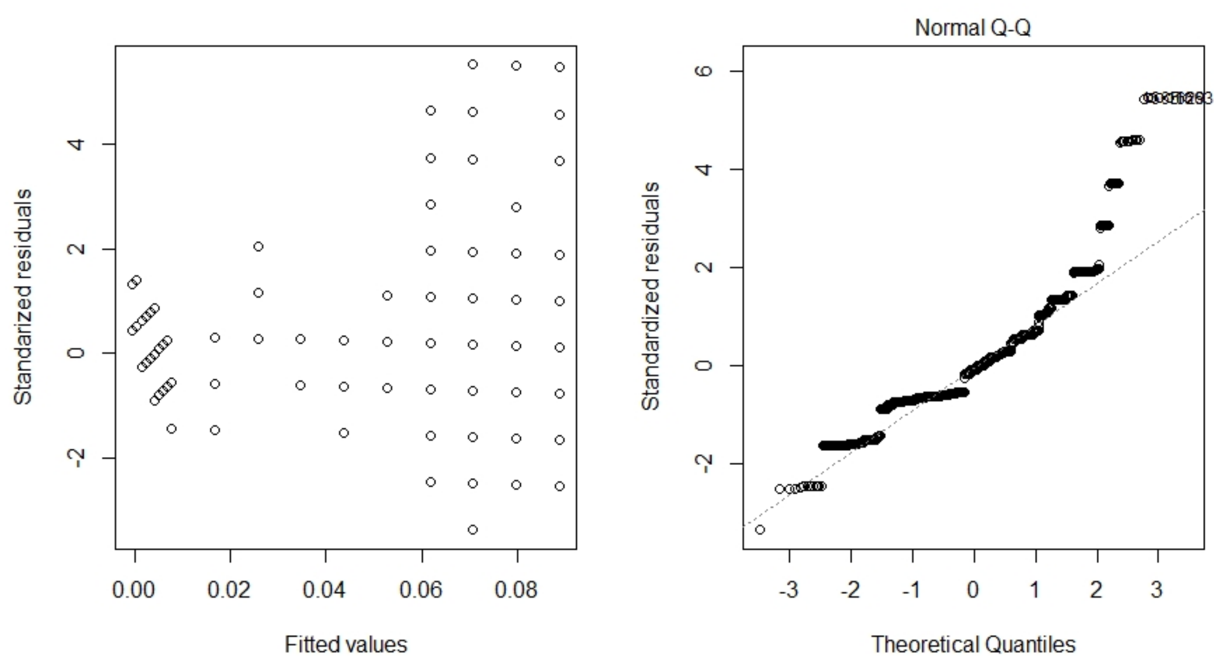


Figura 1.12: A la izquierda, gráfico de residuales frente a los valores predichos para el análisis preliminar del monoreg. A la derecha, el qqplot de normalidad.

lo que podría darse el caso de que, para vectores ordenados, se necesitara un término cuadrático del tamaño.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.00140723	0.00003612	-38.96441337	0.00000000
tamanyo	0.00000009	0.00000000	1129.24046123	0.00000000

Tabla 1.4: Resumen del modelo ajustado para *monoreg*

De momento, supondremos que la respuesta es lineal y viene dada por

$$t_2(n) = 9.027\text{e}{-8} \cdot n - 1.407\text{e}{-3}$$

Nótese que la inclusión de un *intercept* negativo no tiene sentido en un modelo de estas características. Tomamos con pinzas las conclusiones extraídas en este análisis.

1.5.3. Orden de crecimiento de intcox.pavaC

La figura 1.13 muestra los datos y la recta de regresión para el caso de *intcox.pavaC*.

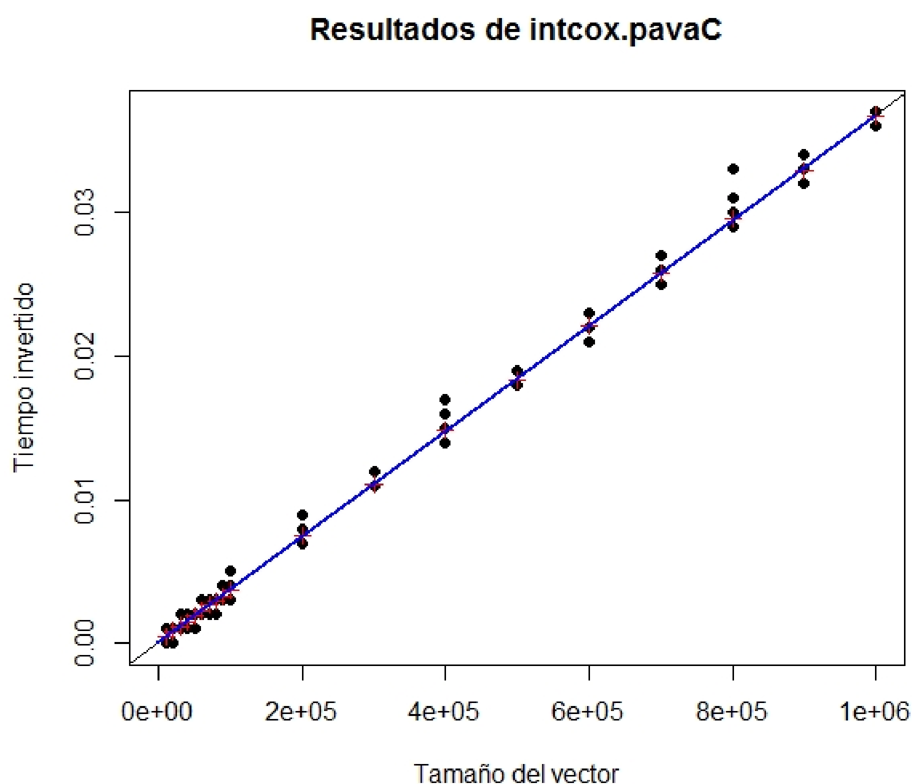


Figura 1.13: Tiempos de ejecución de intcox.pavaC en función del tamaño para el análisis preliminar, junto con la recta de regresión

El ajuste de un modelo lineal con un término cuadrático no resulta significativo, aunque con un término lineal sí hay falta de ajuste. De nuevo, las hipótesis de un modelo normal no se pueden suponer ciertas (figura 1.14).

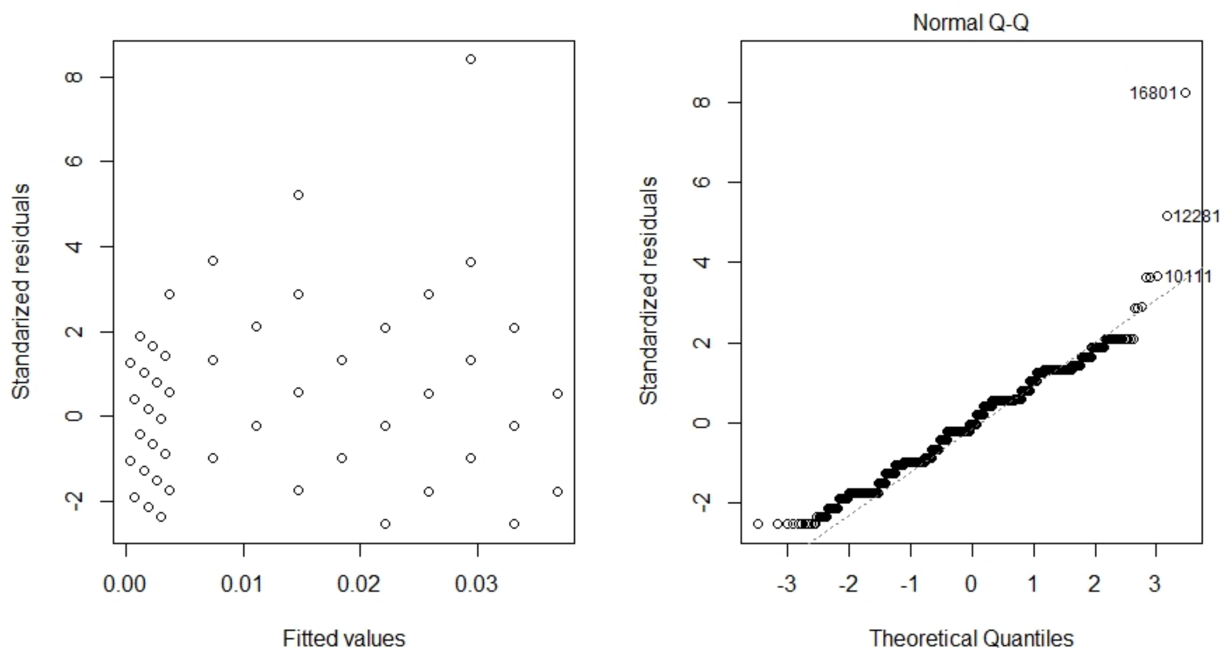


Figura 1.14: A la izquierda, gráfico de residuales frente a los valores predichos para el análisis preliminar del intcox.pavaC. A la derecha, el qqplot de normalidad.

	Estimate	Std. Error	t value	$Pr(>\lvert t \rvert)$
(Intercept)	0.00008428	0.00001380	6.10885558	0.00000000
tamanyo	0.00000004	0.00000000	1201.22893599	0.00000000

Tabla 1.5: Resumen del modelo ajustado para *intcox.pavaC*

La función *intcox.pavaC* está basada en el algoritmo ICM, la versión mejorada del clásico GCM que tiene un orden de complejidad lineal. Parece razonable suponer que el orden de crecimiento es lineal y, aunque las estimaciones pueden ser erróneas, se asume por el momento que aproximadamente

$$t_3(n) = 3.668\text{e}{-8} \cdot n + 3.428\text{e}{-5}$$

1.5.4. Orden de crecimiento de isoreg

El caso de isoreg es un poco diferente. En la figura 1.15 se encuentran los datos, aunque en este caso no se ha ajustado una recta de regresión normal.

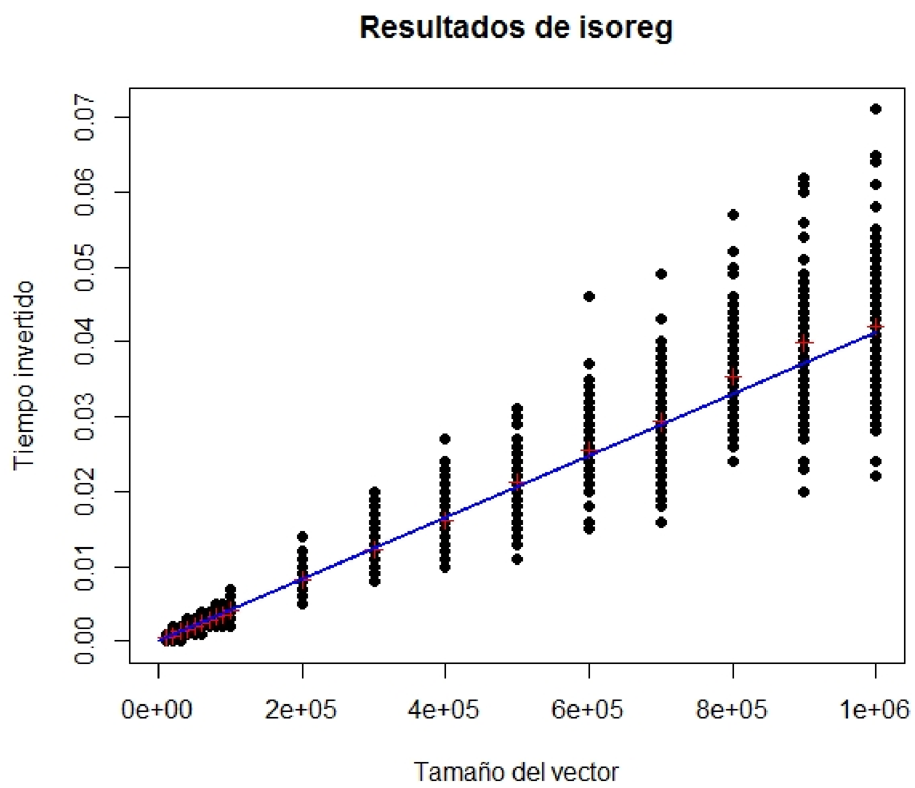


Figura 1.15: Tiempos de ejecución de isoreg en función del tamaño para el análisis preliminar, junto con la recta de regresión.

Aunque inicialmente se pensó en un modelo lineal normal, a la vista de los datos, es imposible tomar la varianza como constante para los diferentes tamaños. La normalidad tampoco es una suposición válida. La solución adoptada en este caso es un modelo lineal generalizado basado en la cuasiverosimilitud.

Al especificar un MLG (Modelo Lineal Generalizado) es necesario tener en cuenta los siguientes aspectos [10].

- La distribución subyacente, que debe ser parte de la familia exponencial, o bien especificar una función para la varianza sobre la media.
- En muchos casos se modela linealmente una función de la media. A esta función se la llama función de enlace.
- El predictor lineal y las variables que intervienen.
- Opcionalmente, unos pesos a las observaciones.

En el caso de *isoreg* se ajusta un MLG sin especificar una distribución concreta, sino que nos basaremos en una cuasiverosimilitud. Como es posible comprobar empíricamente que la desviación estándar depende linealmente de la media, se ajusta un modelo con esta relación. Como función de enlace se elige la identidad.

Si $\boldsymbol{Y}$ representa la variable respuesta tiempo y n el tamaño, queremos establecer un modelo donde

$$E(Y_i) = \beta_0 + \beta_1 n_i \qquad Var(Y_i) = (E(Y_i))^2$$

El gráfico de residuales de este MLG se muestra en la figura 1.16.

La elección de la varianza en función de la media resulta todo un acierto. Aunque se han ajustado modelos de órdenes superiores, dado que la función *isoreg* está basada en el GCM original, no se ha encontrado un término cuadrático significativo. Como todas las simulaciones se han realizado sobre un ruido blanco, no es de esperar que se presente el caso en el que un vector de datos esté ordenado para que *isoreg* tarde más. La existencia de heterocedasticidad puede deberse, fundamentalmente, a la inestabilidad y enorme dependencia de los datos en el cálculo del GCM.

En cuanto a la estimación de los parámetros, el *intercept* no resulta significativo, pero no se tiene ningún motivo para prescindir de él.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.00002125	0.00001557	1.36505032	0.17239901
tamanyo	0.00000004	0.00000000	95.34622725	0.00000000

Tabla 1.6: Resumen del modelo ajustado para *isoreg*

Para este modelo,

$$t_4(n) = 4.12\text{e}{-8} \cdot n + 2.125\text{e}{-5}$$

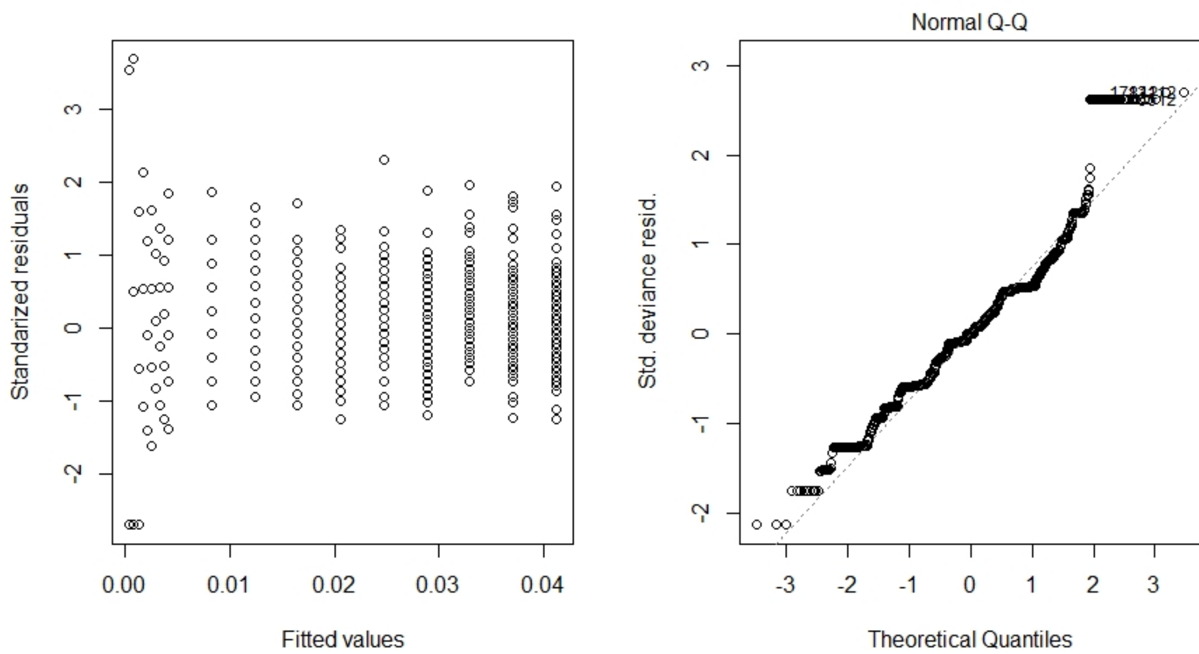


Figura 1.16: A la izquierda, gráfico de residuales frente a los valores predichos para el análisis preliminar de isoreg. A la derecha, el qqplot de normalidad.

1.5.5. Orden de crecimiento de pava

En la figura 1.17 se comprueba que el orden de crecimiento de *pava* no es lineal. Tras probar con un modelo lineal normal con un término cuadrático, éste parece el más razonable. La inclusión de un término cúbico no resulta significativo.

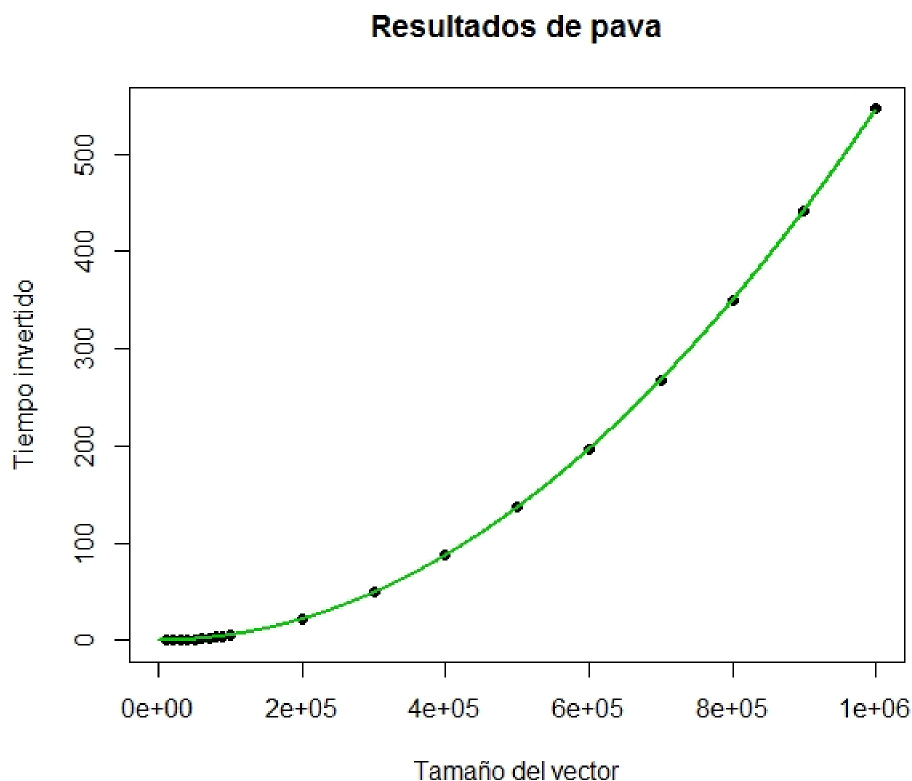


Figura 1.17: Tiempos de ejecución de pava en función del tamaño para el análisis preliminar, junto con la recta de regresión

Sin embargo, parece que existen *outliers* que empañan el ajuste. En la figura 1.18 se puede observar que en algunos casos los residuales se distancian demasiado de lo que se consideraría normal. Aún así, si los eliminamos y repetimos el ajuste sin ellos, la historia se repite con otros puntos que antes no destacaban y la estimación de los coeficientes apenas cambia.

También se ha probado un ajuste de un modelo Gamma y, en términos del AIC, resulta algo mejor. No obstante, para no empañar los resultados y dificultar el modelo, no se asume una distribución Gamma. Además, los coeficientes estimados eran muy similares en ambos casos. Incluso con una única observación por tamaño, el gráfico muestra un buen ajuste para el propósito que aquí se busca.

Es llamativo que esta función escrita en FORTRAN tenga un orden de crecimiento mayor que lineal. Anteriormente hemos comprobado que, incluso en el peor caso, el

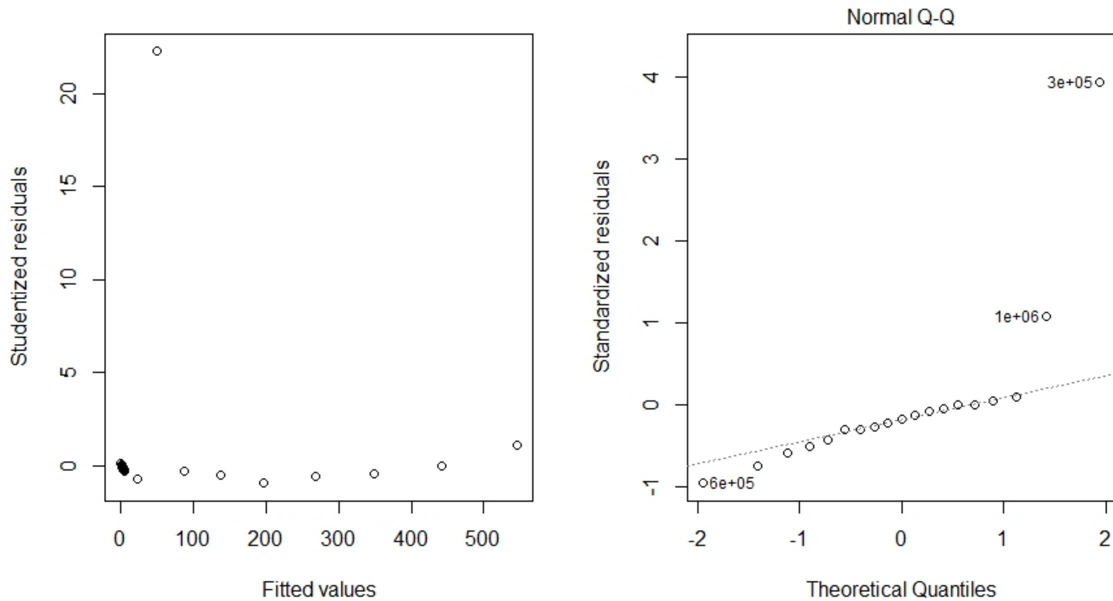


Figura 1.18: A la izquierda, gráfico de residuales frente a los valores predichos para el análisis preliminar del pava. A la derecha, el qqplot de normalidad.

PAVA no tiene un orden cuadrático. Esto nos lleva a preguntarnos por la verdadera naturaleza de la implementación de esta función concreta.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.03538341152	0.11943215244	-0.29626370118	0.77083961439
tamanyo	0.00000132864	0.00000084371	1.57476435925	0.13487479268
I(tamanyo^2)	0.00000000054	0.00000000000	600.11620282028	0.00000000000

Tabla 1.7: Resumen del modelo ajustado para *pava*

La estimación de los parámetros permite concluir que

$$t_5(n) = 5.446\text{e}{-10} \cdot n^2 + 1.329\text{e}{-6} \cdot n - 3.538\text{e}{-2}$$

Aunque los resultados muestran que los coeficientes para el término lineal y el *intercept* no son significativos.

1.5.6. Orden de crecimiento de gpava

En el ajuste del modelo para *gpava* se han eliminado las observaciones para tamaños pequeños y nos hemos quedado con puntos equidistantes. Los resultados se muestran en la figura 1.19 junto con un ajuste polinómico de orden 2.

Si se intenta hacer un ajuste con un polinomio de orden 3, el coeficiente asociado es no significativo, por lo que nos quedaremos con el de orden 2. El gráfico de residuales

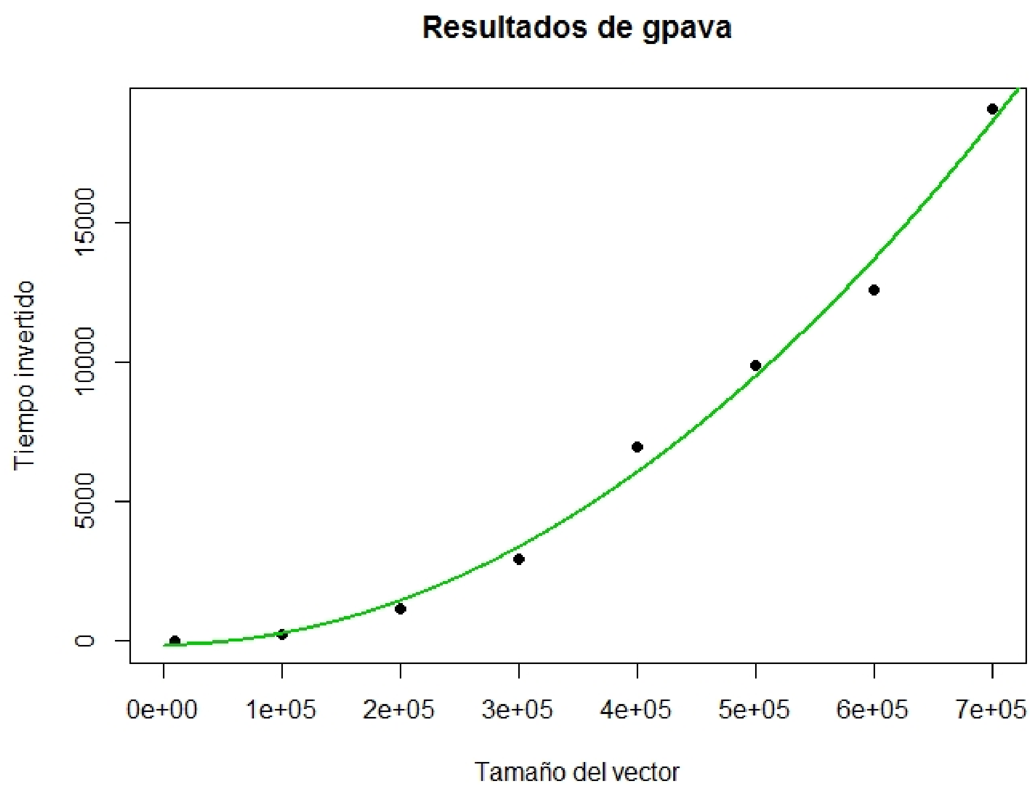


Figura 1.19: Tiempos de ejecución de pava en función del tamaño para el análisis preliminar, junto con la recta de regresión

y el qqplot se muestran en la figura 1.20. Hay pocas observaciones para poder concluir que el ajuste es satisfactorio.

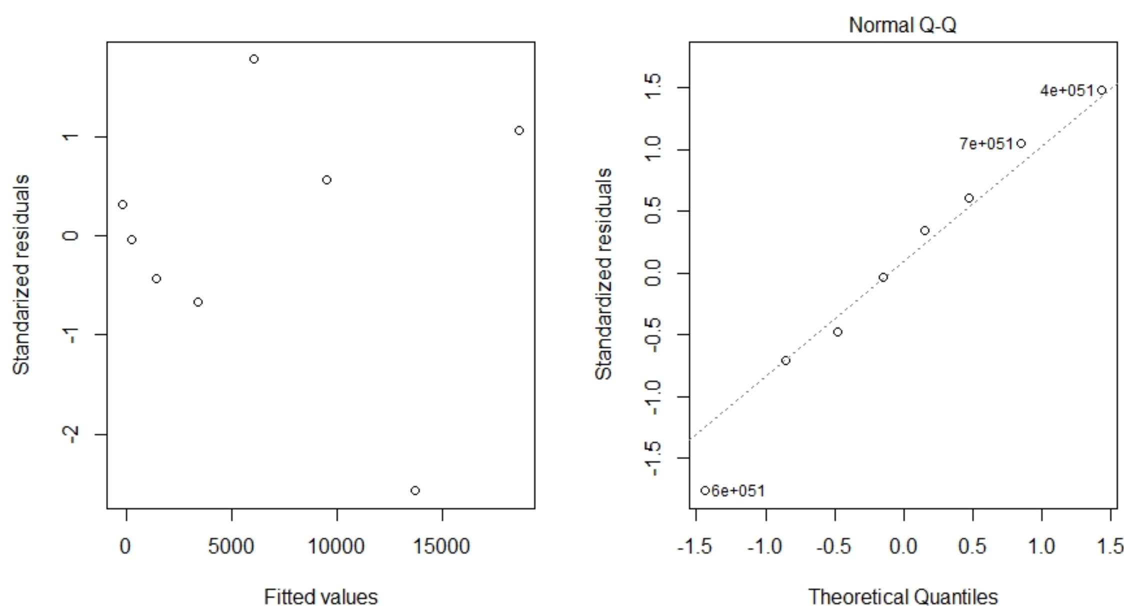


Figura 1.20: A la izquierda, gráfico de residuales frente a los valores predichos para el análisis preliminar del gpava. A la derecha, el qqplot de normalidad.

Esta función también está basada en el PAVA original, pero está escrita íntegramente en R. Es bien sabido que R es un lenguaje de alto nivel, fácil de programar, pero a costa de una ejecución más lenta. Además está el añadido de todas las posibilidades que ofrece esta función, puesto que a mayor complejidad, mayor tiempo esperado en la ejecución.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-146.10557541718	634.21202902616	-0.23037339049	0.82693057049
tamanyo	0.00048219804	0.00422979602	0.11400030707	0.91367370550
I(tamanyo^2)	0.00000003769	0.00000000576	6.54106879990	0.00125032418

Tabla 1.8: Resumen del modelo ajustado para *gpava*

Si asumimos el modelo cuadrático, los coeficientes son

$$t_6(n) = 3.769\text{e}{-8} \cdot n^2 + 4.822\text{e}{-4} \cdot n - 0.0015$$

aunque sólo el término cuadrático es significativo.

1.5.7. Comentarios generales

Como hemos podido comprobar, las funciones *pavaC*, *inctox.pavaC*, *isoreg* y *monoreg* tienen un comportamiento aparentemente lineal del tiempo en función del tamaño del vector. En la figura 1.21 se muestran los tiempos estimados. La función *monoreg* tiende a separarse del otro grupo de 3, mientras que la que pertenece al paquete *intcox* es la que mejor funciona hasta el momento.

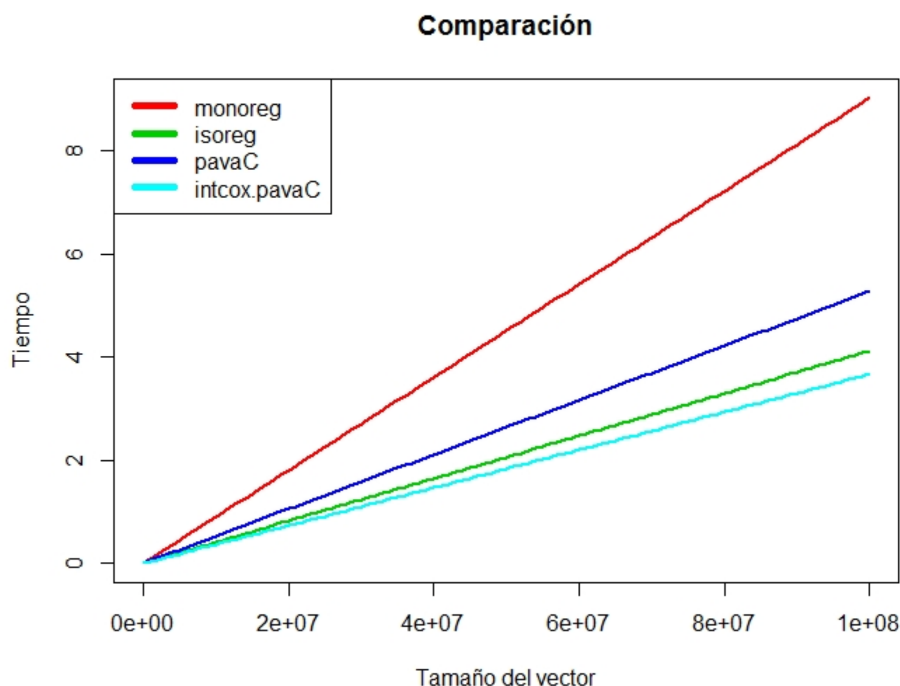


Figura 1.21: Tiempos de ejecución estimados según los modelos anteriores para todos los algoritmos que presentan un comportamiento lineal.

Si comparamos la función *monoreg*, la más lenta de este grupo de 4, con las que tienen un orden cuadrático, los resultados son contundentes (figura 1.22). La función *pava* tiene un crecimiento mucho más rápido, y aún más *gpava*.

Cabe destacar que todas estas conclusiones son válidas únicamente para el caso $\boldsymbol{\mu} = \mathbf{0}$ y sin pesos en las observaciones. Sin embargo, dada la enorme diferencia que existe entre *pava* y *gpava* con los demás algoritmos, éstos serán descartados en el siguiente análisis más exhaustivo, donde nos preguntaremos sobre la influencia del ruido, pesos, y media real.

Es especialmente llamativo que se hayan descartado dos funciones basadas en el PAVA, pues es el algoritmo más eficiente para el cálculo de la regresión isotónica con un orden simple. Las funciones que permanecen están todas escritas en *C*, un lenguaje más rápido en comparación con R y FORTRAN.

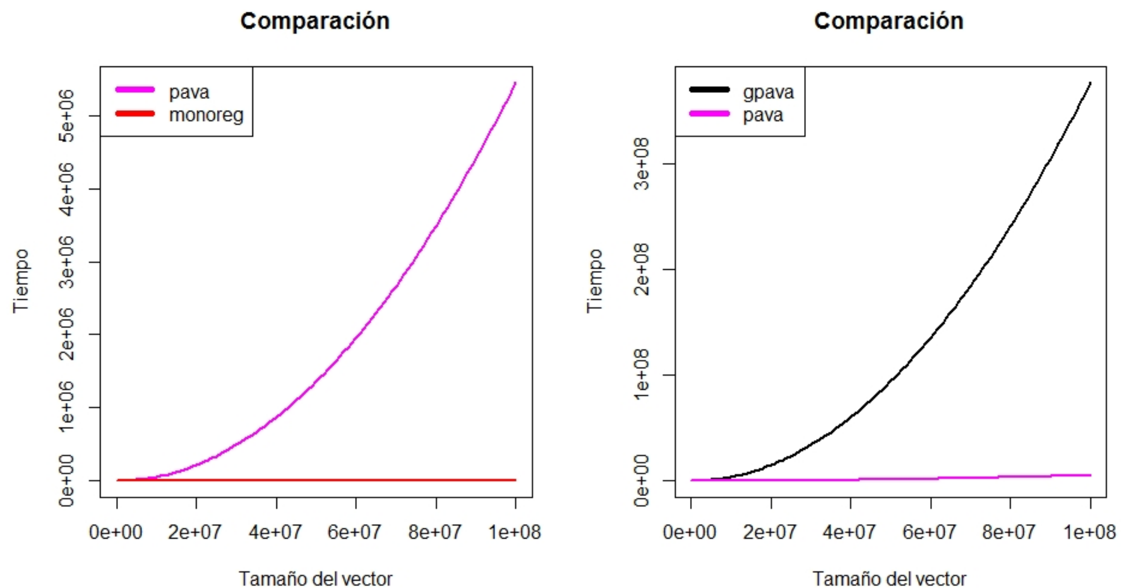


Figura 1.22: Tiempos de ejecución estimados según los modelos anteriores para todos los algoritmos que presentan un comportamiento cuadrático en comparación con monoreg.

1.6. Comportamiento empírico de las funciones de R

Para completar el análisis de los algoritmos se propone el experimento completo donde se quiere estudiar la influencia de los pesos, media real y ruido en los datos, además del orden de crecimiento en función del tamaño. También interesa conocer si existe interacción entre alguno de los factores. El objetivo último es detectar diferencias significativas entre las funciones R para acortar el máximo posible los tiempos de ejecución.

Los parámetros que se consideran en este experimento son:

- Tamaños desde 2e+6 hasta 1e+8 con un paso de 2e+6 (50 valores diferentes).
- Funciones *pavaC*, *isoreg*, *intcox.pavaC* y *monoreg*.
- El valor de $\boldsymbol{\mu}$ se asigna tanto a $\mathbf{0}$ como a un valor creciente generado con el proceso descrito en la sección 1.4.
- Se considera el escenario con y sin pesos.
- El factor de ruido toma los tres valores posibles: poco, intermedio y mucho.

Para cada posible cruce de todos los factores que intervienen en el experimento se consideran 5 repeticiones.

Tras la ejecución de las simulaciones se observó un comportamiento anómalo de la función *isoreg*. Aunque en en análisis preliminar no fue posible discriminar a esta

función, se ha observado que los tiempos de ejecución se ven fuertemente influidos por la existencia de pesos, ruido, y sobre todo la forma del parámetro. Esto motiva la realización de un análisis por separado de *isoreg* con respecto a las demás funciones. En el primer apartado de esta sección examinaremos el comportamiento de *isoreg* en los distintos escenarios planteados, mientras que en el segundo se comentan las diferencias del grupo de funciones restante.

1.6.1. Comportamiento de isoreg

La función *isoreg* está basada en el algoritmo GCM (versión original). En la sección 1.2 se elaboró una discusión sobre el mejor y el peor caso en cuanto a tiempos de ejecución para esta función. Se llegó a la conclusión de que el algoritmo presenta un comportamiento cuadrático en aquellos casos donde los datos estén ordenados y haya pocas violaciones en las restricciones de orden, pero es muy rápido en las situaciones contrarias.

En lo que respecta al experimento planteado, es de esperar que el algoritmo tarde significativamente más en aquellos casos donde se define $\boldsymbol{\mu}$ como creciente y hay poco ruido. En el caso donde el ruido sea abundante la influencia del parámetro será menor, dado que se esperan más violaciones en las restricciones de orden. Por otro lado, la existencia de pesos complica ligeramente los cálculos, teniendo que efectuar más sumas en punto flotante en lugar de con números enteros, por lo que también se espera que el tiempo aumente en este escenario.

Tras el ajuste de varios modelos, se ha comprobado la existencia de interacción de orden 4 entre los factores que conforman el experimento. Esto significa que para cada combinación del valor de $\boldsymbol{\mu}$, nivel de ruido, pesos y tamaño, la función se comporta de una manera diferente. En la figura 1.23 se encuentra una representación gráfica del tiempo de ejecución en función del tamaño para los 12 posibles escenarios que se producen al combinar los tres niveles de ruido, dos valores del parámetro, y la existencia o no de pesos. Los gráficos de la misma fila tienen el mismo nivel de ruido, mientras que los que se encuentran en la misma columna comparten la forma del parámetro. El color rojo indica una ejecución con pesos en las observaciones; en verde se indica una ejecución sin pesos.

Es posible justificar, a la vista de estos gráficos, esta interacción de orden 4. Por un lado, todos los gráficos de la segunda columna muestran un comportamiento similar. Esto es debido a que el valor de $\boldsymbol{\mu}$ es nulo, lo que hace que *isoreg* esté en su mejor caso, independientemente del ruido que haya. Sí que se observa que la existencia de pesos complica los cálculos, así como un aumento de la dispersión en función del tamaño. Esto último es lógico si se tiene en cuenta lo sensible que es *isoreg* a as violaciones de las restricciones de orden. En todos los casos, el aspecto del crecimiento es lineal.

El escenario cambia cuando se considera la forma del parámetro creciente. A medida que se reduce el ruido se disminuye también el número de veces que hay violaciones en las restricciones de orden, y con esto aumentamos la complejidad temporal del algoritmo. Cuando el ruido todavía es abundante, se aprecia una forma similar a los casos con media cero, aunque comienza a aparecer un término cuadrático especialmente visible

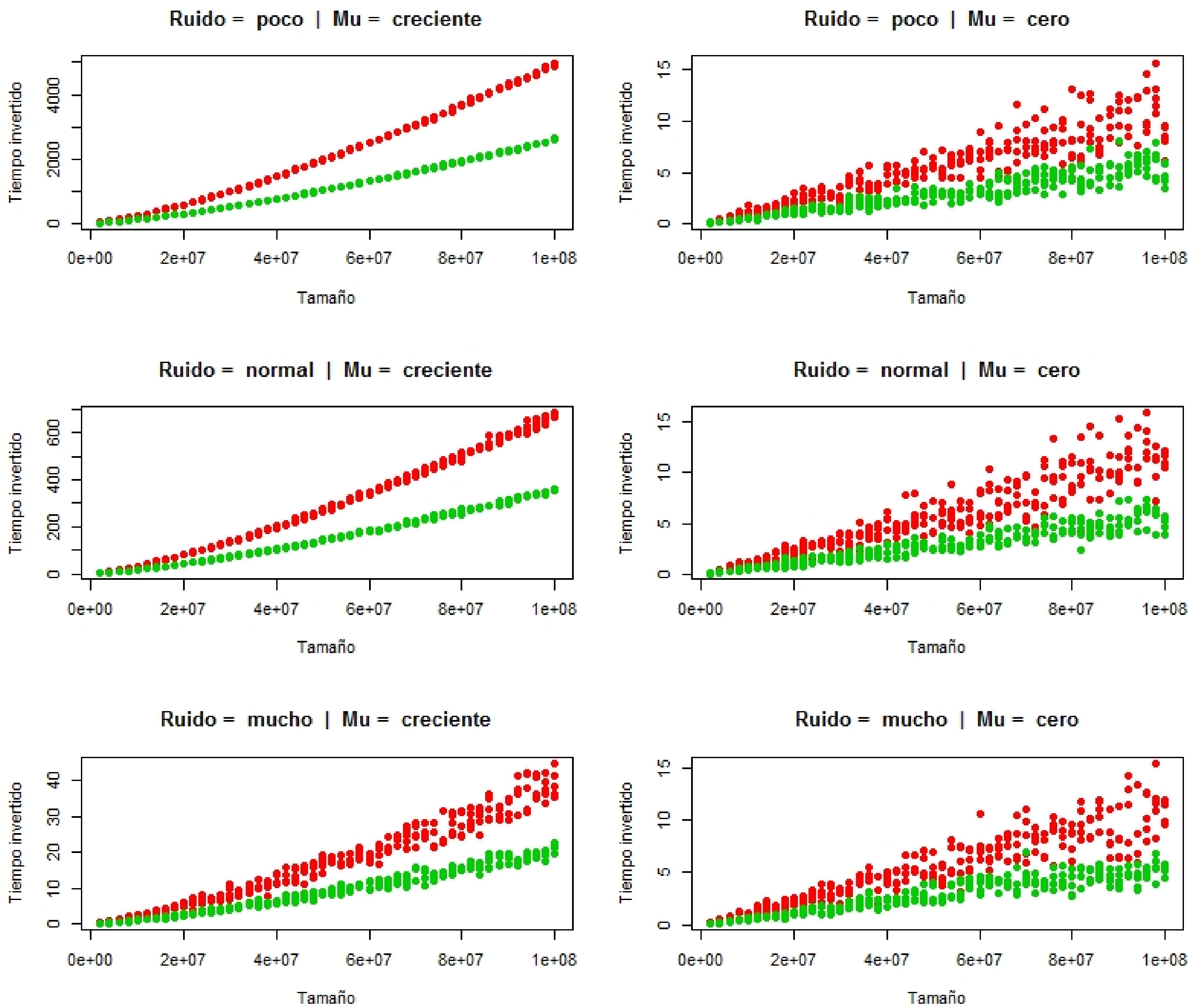


Figura 1.23: Tiempo de ejecución de *isoreg* en todos los posibles escenarios en función del tamaño. En color rojo se muestra el tiempo con pesos, mientras que el verde indica una ejecución sin pesos.

en el caso con pesos. Comprobando la escala de tiempo es fácil darse cuenta de que el aumento es muy grande en el caso con parámetro creciente, y más si hay poco ruido. La dispersión también disminuye al aproximarse al peor caso.

La interacción de orden 4 se puede reconocer en las diferencias que existen con poco ruido y parámetro creciente en comparación con las demás. Por un lado, la interacción entre estos dos factores es clara, ya que el ruido solo afecta al algoritmo si el parámetro es distinto de cero. Además, la curva de tiempo entre las observaciones con y sin pesos parece distanciarse en los peores casos, creciendo de manera cada vez más rápida cuando sí hay pesos. La última variable numérica, el tamaño, interacciona con la combinación de los tres factores, de manera que se produce un ajuste por separado para cada escenario.

Puesto que el propósito de este capítulo es determinar el mejor algoritmo para el cálculo de la regresión isotónica en una amplia variedad de situaciones en las que, previsiblemente, se encontrará el caso creciente y con poco ruido, no podemos considerar *isoreg* como un buen candidato. En la bolsa quedan tres algoritmos cuyas diferencias se explican a continuación.

1.6.2. Diferencias entre monoreg, pavaC e intcox.pavaC

En este último modelo buscamos examinar las diferencias que existen entre las tres funciones de R más eficientes de las consideradas para el cálculo de la regresión isotónica. La función *pavaC* está basada en el clásico algoritmo PAVA, mientras que *intcox.pavaC* y *monoreg* calculan el GCM. Puesto que el comportamiento de esta última no se parece al de *isoreg*, supondremos que el cálculo se realiza en base a la optimización propuesta para evitar un orden cuadrático (el manual no especificaba qué versión).

Tras un análisis preliminar, el modelo ajustado es un modelo lineal normal de regresión que incluye interacción de hasta orden tres, obtenido a partir de un procedimiento de selección *bakward* basado en el AIC. El modelo de partida consideraba todas las posibles interacciones de orden tres, aunque finalmente las entradas que forman parte del modelo seleccionado son:

- El factor “algoritmo”, que toma tres valores, uno para cada posible función; el factor “ruido”, que indica la cantidad de ruido; “pesos”, que distingue entre el cálculo con y sin pesos en las observaciones; y el factor “mu” que puede ser creciente si la media real es creciente, o cero si ésta es nula.
- La variable numérica que indica el tamaño y también su cuadrado, pues se ha comprobado la existencia de una débil tendencia cuadrática y un aumento significativo de la bondad del ajuste al incluirla.
- Interacciones de orden dos entre la variable “tamaño” y los factores “mu” y “pesos”.
- Interacciones de orden dos entre algoritmo y las siguientes variables: “tamaño”, “tamaño^2”, “ruido”, “pesos” y “mu”.
- Interacción de orden tres entre “algoritmo”,“tamaño” y “pesos”; también entre “algoritmo”,“tamaño” y “mu”.

Los resultados que se muestran sobre este modelo son:

- La tabla ANOVA, que puede encontrarse en la tabla 1.9.
- El resumen de los coeficientes del modelo y cuán importantes son para explicar la respuesta, en la tabla 1.10.
- Gráficos que muestran las observaciones junto con la predicción generada a partir del modelo en función del tamaño (figuras 1.24 y 1.25). En el primer caso se muestran los gráficos para una media real creciente, mientras que en el segundo la media es cero.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
algoritmo	2	35711.97	17855.99	2558150.93	0.0000*
tamanyo	1	28778.75	28778.75	4123008.85	0.0000*
I(tamanyo^2)	1	14.53	14.53	2081.02	0.0000*
ruido	2	0.01	0.01	1.05	0.3510
pesos	1	5.35	5.35	767.01	0.0000*
mu	1	0.16	0.16	23.63	0.0000*
tamanyo:pesos	1	1.04	1.04	148.37	0.0000*
tamanyo:mu	1	0.10	0.10	14.37	0.0002*
algoritmo:tamanyo	2	7102.71	3551.35	508787.17	0.0000*
algoritmo:I(tamanyo^2)	2	18.03	9.02	1291.64	0.0000*
algoritmo:ruido	4	0.08	0.02	2.76	0.0261*
algoritmo:pesos	2	16.25	8.12	1163.71	0.0000*
algoritmo:mu	2	0.16	0.08	11.69	0.0000*
algoritmo:tamanyo:pesos	2	2.67	1.33	191.07	0.0000*
algoritmo:tamanyo:mu	2	0.10	0.05	7.33	0.0007*
Residuals	7352	51.32	0.01		

Tabla 1.9: Tabla ANOVA para el modelo de comparación de *isoreg*, *monoreg* y *intcox.pavaC*

Las representaciones gráficas muestran un comportamiento similar de todos los algoritmos en los doce escenarios posibles. La función *monoreg* es la más lenta de los tres y sobre la que se aprecia una ligera tendencia cuadrática. Las otras dos funciones tienen un comportamiento similar, siendo la función del paquete *intcox* la más rápida.

El ajuste que proporciona el modelo a la vista de los gráficos es bastante bueno. Se obtiene un valor de $R^2 = 0.9993$ y una estimación de la desviación estándar del error de $\hat{\sigma} = 0.08355$. Para valorar las diferencias que existen entre las funciones es necesario analizar los coeficientes estimados, pues gráficamente no se aprecian grandes diferencias.

La tabla ANOVA muestra que todas las variables incluidas en el análisis son significativas. El factor "ruido" parece que tiene poca influencia por sí sólo, pero interviene en su interacción con el algoritmo.

En cuanto a los coeficientes estimados, proporcionar una interpretación no es sencillo debido al gran número de parámetros del modelo, así como a las interacciones de

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	7.3790e-03	1.3179e-02	0.56	0.5756
algoritmointcox.pavaC	4.5801e-03	1.8636e-02	0.25	0.8059
algoritmopavaC	-6.2796e-03	1.8636e-02	-0.34	0.7362
tamanyo	1.1627e-07	4.2239e-10	275.28	0.0000*
I(tamanyo^2)	2.2819e-16	3.3667e-18	67.78	0.0000*
ruidonormal	1.0311e-02	4.1273e-03	2.50	0.0125*
ruidopoco	1.1927e-03	4.1261e-03	0.29	0.7725
pesossi	1.0523e-02	9.1821e-03	1.15	0.2518
mucreciente	2.2412e-02	9.1821e-03	2.44	0.0147*
tamanyo:pesossi	-3.2118e-09	1.4237e-10	-22.56	0.0000*
tamanyo:mucreciente	-7.5069e-10	1.4237e-10	-5.27	0.0000*
algoritmointcox.pavaC:tamanyo	-7.0572e-08	5.9730e-10	-118.15	0.0000*
algoritmopavaC:tamanyo	-5.7826e-08	5.9730e-10	-96.81	0.0000*
algoritmointcox.pavaC:I(tamanyo^2)	-2.0410e-16	4.7607e-18	-42.87	0.0000*
algoritmopavaC:I(tamanyo^2)	-2.1474e-16	4.7607e-18	-45.11	0.0000*
algoritmointcox.pavaC:ruidonormal	-1.7412e-02	5.8360e-03	-2.98	0.0029*
algoritmopavaC:ruidonormal	-5.5816e-03	5.8360e-03	-0.96	0.3389
algoritmointcox.pavaC:ruidopoco	-8.7085e-03	5.8351e-03	-1.49	0.1356
algoritmopavaC:ruidopoco	2.8451e-03	5.8351e-03	0.49	0.6259
algoritmointcox.pavaC:pesossi	-8.5298e-03	1.2986e-02	-0.66	0.5113
algoritmopavaC:pesossi	-4.4094e-03	1.2986e-02	-0.34	0.7342
algoritmointcox.pavaC:mucreciente	-2.2110e-02	1.2986e-02	-1.70	0.0887
algoritmopavaC:mucreciente	-1.7375e-02	1.2986e-02	-1.34	0.1809
algoritmointcox.pavaC:tamanyo:pesossi	2.8666e-09	2.0133e-10	14.24	0.0000*
algoritmopavaC:tamanyo:pesossi	3.7689e-09	2.0133e-10	18.72	0.0000*
algoritmointcox.pavaC:tamanyo:mucreciente	7.1644e-10	2.0133e-10	3.56	0.0004*
algoritmopavaC:tamanyo:mucreciente	6.0458e-10	2.0133e-10	3.00	0.0027*

Tabla 1.10: Resumen del modelo ajustado en la comparación de *isoreg*, *intcox.pavaC* y *monoreg*

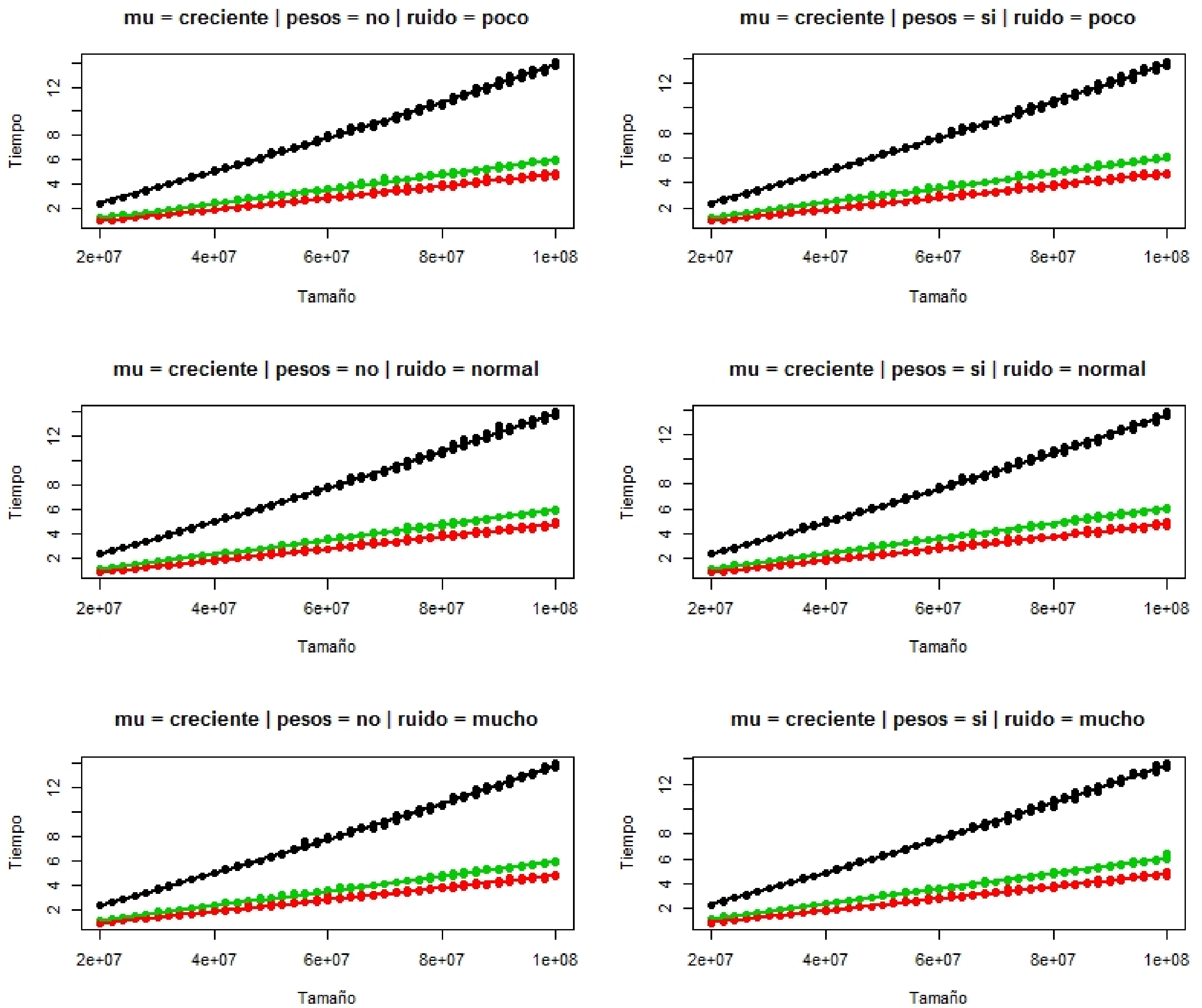


Figura 1.24: Tiempo de ejecución de *monoreg* (en negro), *pavaC* (en verde) y *int-cox.pavaC* (en rojo) en todos los posibles escenarios en función del tamaño y cuando la media real es creciente.

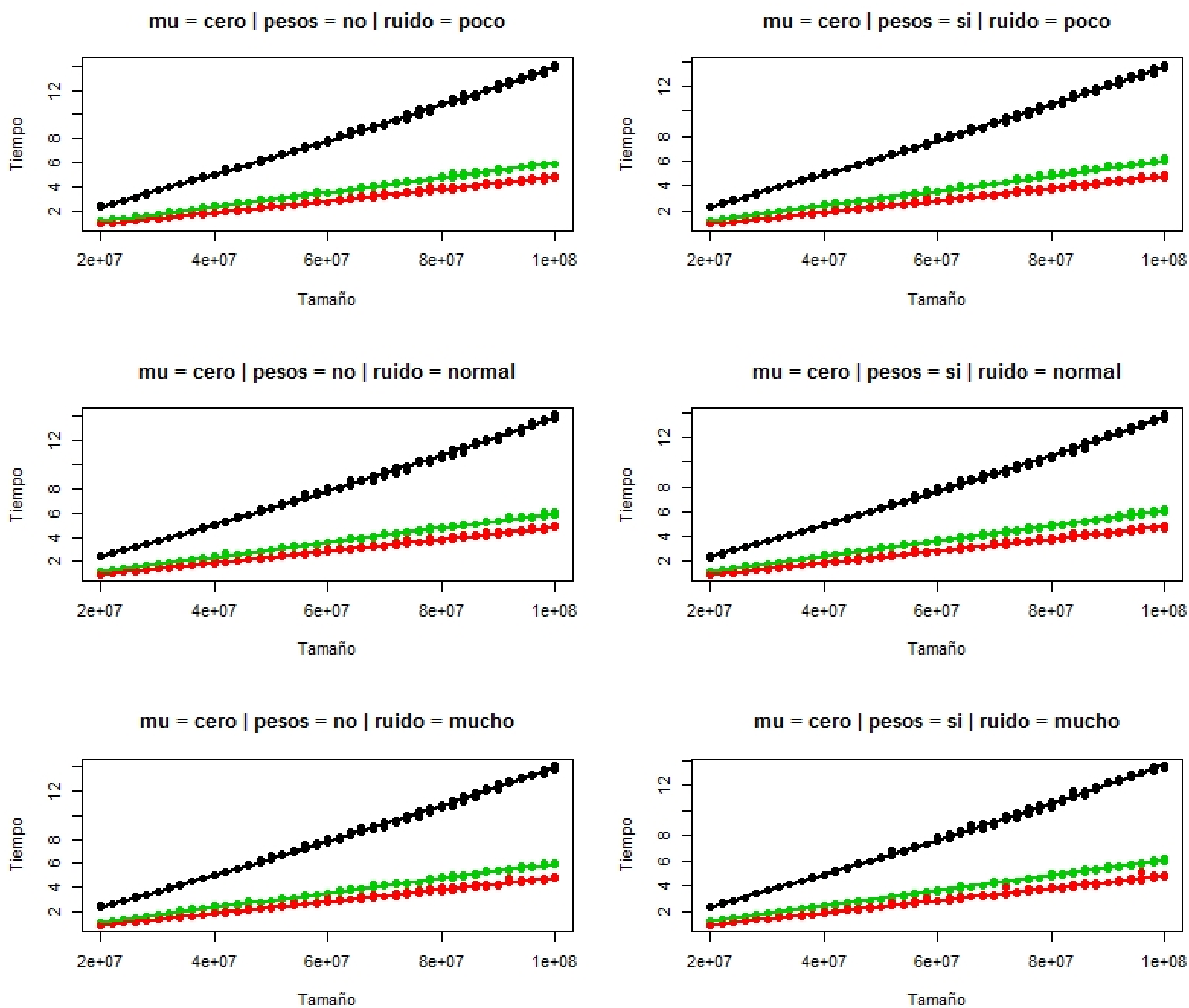


Figura 1.25: Tiempo de ejecución de *monoreg* (en negro), *pavaC* (en verde) y *int-cox.pavaC* (en rojo) en todos los posibles escenarios en función del tamaño y cuando la media real es cero.

orden tres. No podemos fijarnos en los coeficientes de los factores individuales puesto que, por ejemplo, se asigna a *pavaC* un coeficiente menor que a *intcox.pavaC* cuando sabemos que éste último está por debajo en los gráficos. En consecuencia, la interpretación se reducirá a los coeficientes más grandes y a las diferencias que hay en la interacción de los factores. Especialmente interesante es la interacción de "algoritmo" con el resto de factores.

Dado que el conjunto de datos generado es muy grande, es muy fácil detectar diferencias entre distintos niveles de los factores, y muchos tienen un p-valor asociado muy pequeño aunque el estadístico t no sea muy grande. Por este motivo, además del p-valor, se toma el valor absoluto de este estadístico como una medida del impacto del coeficiente asociado. Aquellos casos con un p-valor pequeño pero con un estadístico t no muy grande no se consideran como muy influyentes. Nótese que en esta columna hay algunos casos donde el valor de t es muy grande, y es precisamente estos casos donde radica nuestro interés.

Por un lado, es evidente que el tamaño y el tamaño al cuadrado influyen en la regresión. Para el caso del término cuadrático, se puede concluir que es especialmente significativo para *monoreg*. En el caso de las otras dos funciones, la interacción entre "algoritmo" y el tamaño contrarresta el coeficiente estimado. El término cuadrático para *monoreg* es 2.281947e-16, mientras que para *pavaC* es 1.345796e-17 y para *intcox.pavaC* es 2.409496e-17. Esto es, el coeficiente cuadrático para *monoreg* es de unos 10 veces el orden de magnitud de los otros dos.

La interacción entre "tamaño" y "algoritmo" también tiene un comportamiento similar. En general, al aumentar el tamaño también se incrementa el tiempo requerido para el cálculo. Sin embargo, el crecimiento es bastante más notorio para *monoreg* que para *pavaC*, y el de *pavaC* ligeramente mayor que el de *intcox.pavaC*.

El coeficiente negativo de la interacción entre tamaño y el caso con pesos parece indicar que el tiempo es menor en cuando se tienen pesos en las observaciones. La interacción triple de "algoritmo", "pesos" y "tamaño" contrarresta el coeficiente negativo para *pavaC* e *intcox.pavaC*. Parece que *pavaC* es el que se ve más influido cuando hay pesos al aumentar el tamaño, incrementando su tiempo de ejecución en más medida que *intcox.pavaC*. Por otro lado, *monoreg* no tiene esta penalización.

En menor medida se pueden extraer conclusiones similares de la interacción de "tamaño" con "mu". Si el parámetro es creciente, la interacción con el tamaño tiene un coeficiente negativo. Sin embargo, si consideramos *pavaC* hay que sumarle una penalización, y más en el caso de *intcox.pavaC*. Aún así, estas penalizaciones siguen proporcionando un valor negativo en el coeficiente de tamaño cuando el parámetro es creciente, lo que parece indicar que cuanto menos se ajusten los datos a un patrón creciente, más tiempo tardará la función. Esto es lógico si pensamos en la naturaleza del PAVA, donde se hacen menos operaciones cuantas menos violaciones en las restricciones haya.

Tras el análisis de este modelo, en resumidas cuentas, se pueden extraer las siguientes conclusiones.

- Los tres algoritmos presentan un débil comportamiento cuadrático con el tamaño del vector de entrada. No obstante, es especialmente notorio en el caso de *monoreg*,

y es posible que en los otros dos casos resulte significativo por algún problema de sobreajuste.

- El término lineal del tamaño es bastante mayor en el caso de *monoreg*. En general, el comportamiento de esta función es el peor de las tres, mientras que *intcox.pavaC* es la que mejor se comporta.

- La función *monoreg* se ve menos influida por la existencia de pesos que las otras dos funciones. La que más influida se ve es *pavaC*.

- Todas las funciones mejoran sus tiempos de ejecución en el caso en que la media es creciente, especialmente *monoreg*.

1.7. Conclusiones

A lo largo de este capítulo hemos podido comprobar el fundamento teórico que subyace en el cálculo de la regresión isotónica dado un orden simple, y se han explorado algunas de las posibilidades para su cálculo en R.

En primer lugar se hizo una pequeña discusión sobre los mejores y peores casos de cada una de las dos formas para el cálculo de la regresión isotónica: el máximo minorante convexo y el PAVA. Aunque el mejor caso de uno sea el peor caso del otro y viceversa, y pueda pensarse en aprovechar esta posibilidad para elegir la mejor opción de cálculo dinámicamente, hemos podido comprobar empíricamente que las diferencias no son demasiado grandes.

De entre todas las funciones de R, hay una que llama la atención especialmente por el enorme gasto computacional que conlleva su uso. La función *gpava*, del paquete *Isotone* [9], ofrece muchas posibilidades, pero a costa de un mayor tiempo de ejecución, muy por encima de sus competidores. Es recomendable su uso sólo en caso de necesitarse sus ventajas, pues es la más completa de todas y ofrece muchas opciones adicionales.

La función *pava* también tiene un orden de crecimiento sustancialmente mayor que las demás, aunque las diferencias no son tan grandes. La función *isoreg* parece ser rentable sólo en ciertos casos al estar basada en el GCM original, además de tener una dispersión mayor por su dependencia de los datos.

Con respecto a las otras tres funciones, las diferencias entre ellas son menores que con las anteriores. La que tiene un mejor comportamiento es la del paquete *intcox*, seguida de la que se ha creado desde cero llamada *pavaC*. El propósito último de este capítulo es el de elegir una función de estas seis, puesto que será requerida en el siguiente capítulo e interesa que el tiempo invertido sea el menor posible. La opción más sensata sería utilizar *intcox.pavaC* siempre que sea posible, aunque finalmente no se escogerá esta por las razones que se exponen a continuación.

En primer lugar, el paquete *intcox* [7] ha sido eliminado de forma oficial del CRAN de R, y sólo pueden descargarse versiones antiguas del mismo que ya no tienen soporte oficial por parte del autor. Además, la función sólo ofrece el cálculo de un PAVA creciente. Es bien sabido que el cálculo del PAVA decreciente puede hacerse a partir del

PAVA creciente con una ligera modificación: en primer lugar se revierte el orden del vector original, para luego aplicar el PAVA, y posteriormente multiplicar revertir el orden del resultado. Sin embargo, las operaciones añadidas provocan un gasto computacional extra que se puede ahorrar si se adapta un PAVA desde el principio.

Por este motivo, finalmente se utilizará la función *pavaC*. Tras un análisis del código original de la función del paquete *intcox*, se han encontrado ciertas mejoras que pueden ser añadidas en *pavaC* para disminuir su tiempo de cálculo. Una vez que estas modificaciones se hagan permanentes, es de esperar que *pavaC* resulte una muy buena opción para el cálculo del PAVA, tanto en el caso creciente como en el caso decreciente.

Adicionalmente, también se ha podido comprobar que la utilización de C mejora en gran medida el tiempo invertido en cualquier algoritmo. No es casualidad que la función más lenta esté escrita en R, puesto que R no se caracteriza por su eficiencia. Aunque FORTRAN es más rápido que R, aún así parece distanciarse lo suficiente de C como para no considerar su implementación del PAVA.

En todos los modelos ajustados se han encontrado ciertas observaciones que no se corresponden con un comportamiento esperado debido a la influencia de las tareas del sistema durante las simulaciones. Es por este motivo que, en muchos casos, algunas observaciones que claramente presentaban anomalías se han eliminado (o sustituido en caso de que fuera sencilla su repetición). Además, no parece interesante considerar tamaños mayores a 1e+8, puesto que no se prevé que se necesite el cálculo del PAVA con tantos puntos.

Capítulo 2

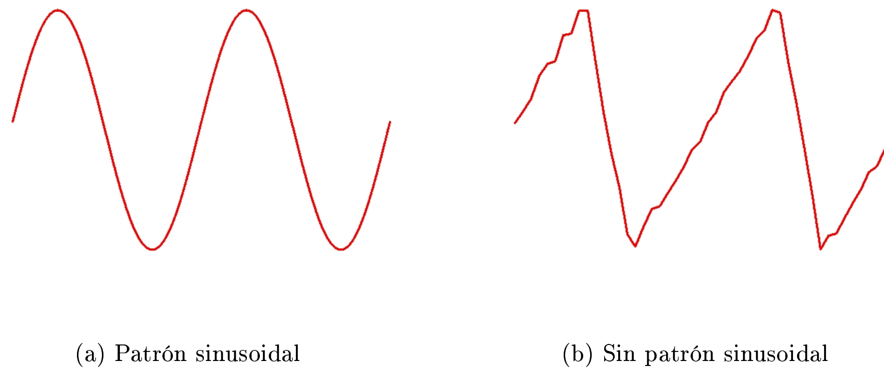
Mejoras de eficiencia para el algoritmo de cálculo del estimador up-down-up

Numerosos procesos biológicos en la naturaleza presentan un comportamiento oscilatorio a lo largo del tiempo, pues son gobernados por ciertos componentes que exhiben patrones rítmicos [11], [12]. Algunos de los ejemplos más comunes abarcan desde el ciclo celular o el ciclo menstrual, hasta la detección de genes asociados con trastornos neurodegenerativos o depresiones [13], [14]. El estudio de dichos componentes con patrones rítmicos y los cambios que se producen en su comportamiento bajo ciertas condiciones son una de las labores de la cronobiología [15], [16], [17] y [18]. La cronobiología es una ciencia que está cada vez más presente y activa, y en la que se han hecho importantes descubrimientos recientemente. Sin ir más lejos, *E. Hauss* [19] probó que los tratamientos de radioterapia en pacientes con cáncer puede tener un mejor o peor rendimiento en función del momento del día en el que se administre.

Los investigadores y biólogos comparten un creciente interés en la identificación y estudio de genes que participan en el ciclo circadiano, uno de los procesos biológicos en el que se exhiben patrones rítmicos [20]. El problema de la identificación de estos genes no es trivial debido a la variabilidad de los datos sobre expresiones de genes a lo largo del tiempo, así como a la ausencia de modelos paramétricos lo suficientemente flexibles como para acomodarse a una razonable colección de genes rítmicos. Recientemente, [11] introdujo una metodología bautizada como *ORIOS* (Order Restricted Inference for Oscillatory Systems) que permite no sólo la clasificación de los genes en rítmicos y no rítmicos, sino también en distintos tipos de patrones rítmicos.

En términos estadísticos, la estimación de un modelo es complicada puesto que la densidad de puntos con la que se trabaja es muy baja, y el número de periodos es pequeño [21], [22], [23]. Esto imposibilita un análisis adecuado utilizando métodos de series temporales, por lo que es necesario buscar métodos alternativos. Así mismo, algunos modelos paramétricos y ajuste de funciones matemáticas han sido probados sin demasiado éxito. El motivo es que la rigidez de los modelos paramétricos es excesiva cuando se intenta ajustar a patrones irregulares y no sinusoidales. En la figura 2.1 puede

verse un ejemplo de un patrón sinusoidal y otro no sinusoidal. Mientras que en genes cuya expresión tenga un patrón similar al de la figura 2.1a, donde se espera que un ajuste de una función coseno funcione bien, no es posible afirmar lo mismo de la figura 2.1b. En este caso un modelo no paramétrico es más conveniente, pues admite una mayor flexibilidad y robustez.



(a) Patrón sinusoidal (b) Sin patrón sinusoidal

Figura 2.1: Ejemplo de patrón sinusoidal y no sinusoidal

El objetivo principal de la metodología desarrollada en [11] es la estimación de las expresiones de los genes, que tiene su posterior aplicación en la clasificación de genes en rítmicos y no rítmicos. Dichos procedimientos están basados en inferencia con restricciones, con los que, utilizando un algoritmo basado en regresión isotónica [1], se obtiene una estimación del patrón que posteriormente se utiliza para efectuar el test de ritmicidad.

En este trabajo se pretende proporcionar una mejora en la eficiencia de dicho algoritmo, implementado en R [5], depurando y mejorando la implementación práctica, siempre con el objetivo de reducir los tiempos de ejecución al máximo posible. La función no muestra unos tiempos elevados para el ajuste individual, pero lo normal es que las bases de datos de genes estén compuestas por alrededor de 40000 genes. Esta ejecución múltiple sí provoca que el algoritmo tarde en calcular los estimadores para todos los genes, lo que genera la motivación de la mejora en la eficiencia del mismo.

Las mejoras que se efectúan sobre el código original han sido divididas en cuatro grandes grupos.

- En primer lugar, se considera una *depuración de la programación* y mejoras generales que no tienen que ver con el *qué*, sino con el *cómo*. Las mejoras incluidas en este apartado no son específicas de este código, sino más generales: optimización de los accesos a memoria, sustitución de funciones nativas por otras más eficientes, etc.

- *Mejoras derivadas de resultados teóricos*, que permiten reducir el espacio de búsqueda y con ello el tiempo invertido. En concreto, se demuestran ciertos resultados

teóricos que permiten simplificar el procedimiento de estimación de los parámetros. Evidentemente y a diferencia de las mejoras del apartado anterior, estas sí son específicas del problema que se quiere resolver.

- Se considerarán los beneficios de la *implementación en C* de parte del código del algoritmo. C es conocido por ser un lenguaje estructurado de programación en bajo nivel, y su mayor punto fuerte es la eficiencia de los programas escritos en este lenguaje [24].

- Por último, se tendrán en cuenta las mejoras producidas por la *paralelización de procesos*, permitiendo la ejecución múltiple y en distintos hilos de tareas independientes, que luego son combinadas. La ejecución simultánea de procesos conlleva un aumento del uso de la CPU, pero se espera que las tareas se lleven a cabo en un tiempo menor.

Las mejoras implementadas pertenecientes al primer grupo son tratadas en la sección 2.2. Las mejoras de los tres siguientes grupos se tratan en las secciones 2.3, 2.4 y 2.5 respectivamente. En el caso del primer grupo, se evaluará el impacto que tienen los arreglos de depuración con respecto al código original. En los sucesivos apartados, se evaluará el rendimiento con respecto a las mejoras inmediatas anteriores; es decir, en el caso de las mejoras derivadas de los resultados teóricos, su rendimiento se evaluará con respecto a las mejoras del primer grupo, y así con las demás. Cuando todos los grupos hayan sido cubiertos, se efectuará una comparación global con respecto al código original.

Todos los experimentos han sido realizados en un ordenador cuyas características se listan a continuación.

- 2 procesadores idénticos (Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, Intel Corp.), cada uno de ellos con 16 núcleos físicos y 32 núcleos lógicos, lo que da lugar a la posibilidad de ejecutar hasta 64 tareas simultáneamente (64 bits).

- Caché de nivel 1 de 1 MiB.

- Caché de nivel 2 de 16 MiB.

- Caché de nivel 3 de 22 MiB.

- 256 GiB de memoria principal DDR4.

- Sistema operativo CentOS.

2.1. Modelo de señal up-down-up

En esta sección se introduce el modelo de señal *up-down-up* que es utilizado por el algoritmo que será mejorado. A lo largo de todo el trabajo se asume, por simplicidad, que los datos obtenidos presentan períodos completos, aunque la metodología es fácilmente extensible al caso contrario.

2.1.1. Notación y definiciones

En lo sucesivo se utilizará el término "gen" para referirse a la variable respuesta, y "expresión del gen" los valores de dicha variable.

Para cada gen, suponemos que conocemos su expresión en los instantes de tiempo $t_i, i = 1, ..., n$ en cada uno de los p períodos, siendo T la longitud de cada período (usualmente, $p = 2$ y $T = 24$). Sea X_{ij} una variable continua que indica el valor de la expresión del gen en el instante temporal i del período j, con $i = 1, ..., n$ y $j = 1, ..., p$; $\mathbf{X}_j = (X_{1j}, ..., X_{nj})$ el vector que contiene los datos relativos al período j-ésimo e $\mathbf{Y} = \left(\overline{X}_1, ..., \overline{X}_n\right)$ la información resumida de la expresión del gen a lo largo de todos sus períodos, donde $\overline{X}_i = \frac{1}{p}\sum_{j=1}^{p} X_{ij}$.

Se asume un modelo de señal más ruido de la forma:

$$\mathbf{X}_j = \boldsymbol{\mu} + \boldsymbol{\epsilon_j} \tag{2.1}$$

donde el parámetro $\boldsymbol{\mu}$ es un patrón *up-down-up*, que se definirá más adelante (véase la definición 8).

En este modelo se asume que los datos obtenidos a lo largo de los p períodos tienen el mismo valor medio esperado, es decir, $E(\boldsymbol{X_j}) = \boldsymbol{\mu}$ para $j = 1, ..., p$; también que la varianza del error es constante y que son independientes, o expresado de otra manera, $Var(\boldsymbol{X_j}) = \sigma^2 \boldsymbol{I_n}$, donde $\boldsymbol{I}_n$ es la matriz identidad n-dimensional.

En principio, no se hace ninguna suposición distribucional adicional en el modelo. Sólo cuando se indique de forma explícita que se supone normalidad, el modelo quedaría descrito por:

$$\mathbf{X_j} \sim \mathcal{N}_{\mathbf{n}}(\boldsymbol{\mu}, \sigma^{\mathbf{2}}\mathbf{I_n}) \tag{2.2}$$

Se denotará por L y U los índices en los que la señal $\boldsymbol{\mu}$ alcanza su mínimo y su máximo respectivamente. Formalmente:

$$L = \underset{i=1,...,n}{\arg\min}\, \mu_i \qquad U = \underset{i=1,...n}{\arg\max}\, \mu_i$$

El valor de L y U es único, de acuerdo a la definición de señal *up-down-up* que se presenta a continuación. Esta señal crece hasta el valor μ_U, después decrece hasta el valor μ_L, y por último vuelve a crecer hasta su valor inicial. Dos ejemplos de patrones up-down-up pueden verse en la figura 2.1. Puesto que este tipo de patrones pueden expresarse con desigualdades, es posible aplicar metodología de inferencia con restricciones.

Definición 8 (Señal up-down-up). Una señal $\boldsymbol{\mu}$ en el espacio euclídeo se dice que es una señal up-down-up si y sólo si $\boldsymbol{\mu} \in C = \bigcup_{LU} C_{LU}$, donde $L, U \in \{1, ..., n\}$, $C_{LU} = \{\boldsymbol{\mu} \in \mathbb{R}^n : \mu_1 \leq \ldots \leq \mu_U \geq \ldots \geq \mu_L \leq \ldots \leq \mu_n \leq \mu_1\}$ si $L > U$ y $C_{LU} = \{\boldsymbol{\mu} \in \mathbb{R}^n : \mu_1 \geq \ldots \leq \mu_L \leq \ldots \leq \mu_U \geq \ldots \geq \mu_n \geq \mu_1\}$ si $U > L$.

Sin pérdida de gerenalidad y a menos que se indique lo contrario, de ahora en adelante siempre supondremos que $L > U$.

2.1.2. Regresión isotónica para las señales up-down-up

El objetivo del algoritmo que se mejora en este trabajo es la estimación de la señal up-down-up. Sea $\boldsymbol{\mu}$ una señal up-down-up y sea $\boldsymbol{Y} = \boldsymbol{\mu} + \boldsymbol{\epsilon}$ un modelo de señal más ruido. Denotemos por $\boldsymbol{Y}^*$ el estimador basado en regresión isotónica para $\boldsymbol{\mu}$. El problema de encontrar $\boldsymbol{Y}^*$ se puede formular como un problema de mínimos cuadrados de la siguiente forma:

$$\boldsymbol{Y}^* = \arg\min_{\boldsymbol{Z} \in C} \sum_{i=1}^{n} \omega_i (Y_i - Z_i)^2 \tag{2.3}$$

con $\boldsymbol{\omega} = (\omega_1, ..., \omega_n)'$ un vector de pesos positivos. Estos pesos no serán homogéneos en el caso de que la varianza no sea constante a lo largo del tiempo.

Como el estimador está basado en regresión isotónica, $\boldsymbol{Y}^*$ será una función a trozos (véase Robertson et al. [1]). Cada conjunto de puntos donde el estimador toma los mismos valores es llamado *conjunto de nivel*. El valor de la ordenada de los puntos de un mismo conjunto de nivel es la media ponderada de los valores observados en dichos puntos, tal y como fue discutido en el capítulo 1. Evidentemente, un estimador up-down-up puede tener entre 1 y n conjuntos de nivel.

En la figura 2.2 puede verse un ejemplo de la expresión de un gen (simulado mediante un computador, con el modelo descrito de señal más ruido), junto con su estimador up-down-up que minimiza el criterio de mínimos cuadrados.

A continuación se presentan ciertos resultados teóricos probados en [12] que se utilizan en el algoritmo de cálculo del estimador.

Teorema 3. Sea $I_M = \{j : j = \arg\max_{i=1,\ldots,n} Y_i^*\}$. Entonces, I_M es un conjunto formado por exactamente un elemento. Además, el conjunto $I_L = \{j : j = \arg\min_{i=1,\ldots,n} Y_i^*\}$ también tiene exactamente un elemento.

Corolario 1. Sea $U^* = \arg\max_{i=1,\ldots,n} Y_i^*$. Entonces $U^* \in loc_M(\boldsymbol{Y}) = \{i : Y_i$ es un máximo local de $\boldsymbol{Y}\}$. De manera análoga, si $L^* = \arg\min_{i=1,\ldots,n} Y_i^*$, entonces $L^* \in loc_m(\boldsymbol{Y}) = \{i : Y_i$ es un mínimo local de $\boldsymbol{Y}\}$.

Finalmente, se mostrará el pseudocódigo (figura 2.3) del algoritmo original sobre el que se trabajará en el resto del trabajo, suponiendo $L^* > U^*$. En su mayor parte está basado en el conocido *Pool Adjacent Violators Algorithm* [1]. Como se ha comprobado que los valores U^* y L^* sólo pueden ser mínimos o máximos locales, se hace el ajuste para todas estas posibles combinaciones. En cada cálculo individual es necesario ajustar una parte utilizando un PAVA creciente, mientras que en otra el sentido es decreciente. De entre todas ellas, se elige aquella que tenga un MSE más pequeño.

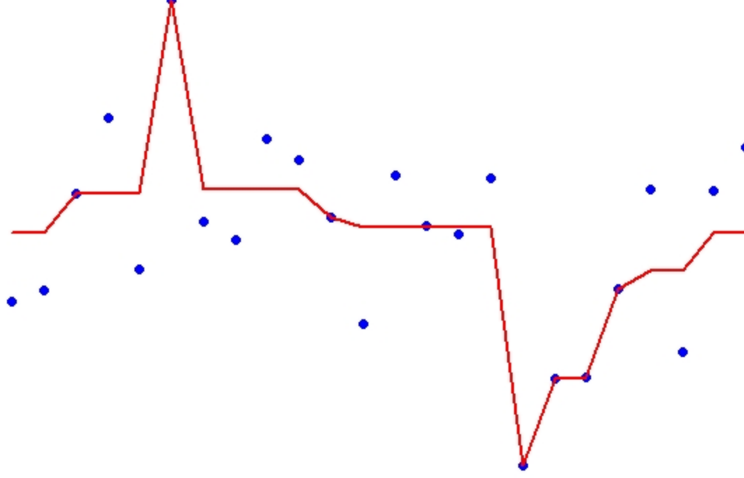


Figura 2.2: Ejemplo de una señal más ruido y cálculo del estimador up-down-up

Entrada: $\boldsymbol{Y}, loc_M(\boldsymbol{Y}), loc_m(\boldsymbol{Y})$
Salida: $\boldsymbol{Y}^*, mse$
$mse \leftarrow Inf$
for each $(m, M) \in A = \{(m_i, M_j) : i \neq j, m_i \in loc_m(\boldsymbol{Y}) \text{ y } M_j \in loc_M(\boldsymbol{Y}))\}$ **do**
Aislar Y_m, Y_M
$\boldsymbol{Y}^*_{up} \leftarrow$ PAVA creciente de $(Y_{m+1}, ..., Y_n, Y_1, ..., Y_{M-1})$
$\boldsymbol{Y}^*_{down} \leftarrow$ PAVA decreciente de $(Y_{M+1}, ..., Y_{m-1})$
$\hat{\boldsymbol{Y}}^* \leftarrow (\boldsymbol{Y}^*_{up}[1 : M-1], Y_M, \boldsymbol{Y}^*_{down}, Y_m, \boldsymbol{Y}^*_{up}[m+1 : n])$
$\hat{mse}^* \leftarrow MSE(\boldsymbol{Y}, \hat{\boldsymbol{Y}}^*) = \sum_{k=1}^n \omega_k (Y_k - \hat{Y}^*_k)^2$
if $\hat{\boldsymbol{Y}}^* \in C_{mM}$ **and** $\hat{mse}^* < mse$ **then**
$\boldsymbol{Y}^* \leftarrow \hat{\boldsymbol{Y}}^*$
$mse^* \leftarrow \hat{mse}^*$
$mse \leftarrow \hat{mse}^*$
end-if
end-do

Se supone que el mínimo local ocurre antes que el máximo, aunque se puede extender fácilmente al caso contrario. La notación $\boldsymbol{Y}^*_{up}[1 : M-1]$ se refiere a tomar las posiciones del PAVA creciente asociadas a los valores $(Y_1, \dots, Y_{M-1})$.

Figura 2.3: Pseudocódigo de la función original que calcula el mejor estimador up-down-up en el sentido de mínimos cuadrados

2.1.3. Rendimiento del algoritmo original

En esta sección se pretende evaluar el rendimiento del algoritmo original, en términos de tiempos de ejecución, con el objetivo de establecer comparaciones con las mejoras futuras.

Para la ejecución de pruebas se utilizarán 6 bases de datos que se describen a continuación.

- En primer lugar se utilizarán cuatro bases de datos de genes bien conocidas en la literatura, disponibles online en NCBI GEO (*http:www.ncbi.nlm.nih.gob/geo/*). En tres de ellas se tienen datos sobre $N = 45101$ genes de hígado de ratón (48RMA), glándula pituitaria (Pituitary) y datos de líneas celulares NIH3T3, respectivamente. En la última, se tienen datos sobre $N = 32321$ genes sobre líneas celulares U2OS. Cada gen de estas bases de datos tiene información sobre 48 instantes de tiempo y representan dos períodos de datos (es decir, $p = 2$ y $T = 24$).
- La quinta base de datos también representa datos de genes reales. Corresponde a datos normalizados de células madre de ratón [29]. Está formada por $N = 307$ genes a lo largo de 238 instantes de tiempo, que representan un único período
- La última base de datos son datos simulados. Se utiliza como señal up-down-up el patrón sinusoidal mostrado en la figura 2.1a. A ésta se le añade un ruido procedente de una distribución normal, de forma que cada fila de la base de datos procede de $\mathbf{Y} \sim \mathcal{N}_{\mathbf{2500}}(\boldsymbol{\mu}, \mathbf{I_{2500}})$, donde $\boldsymbol{\mu}$ es la señal sinusoidal, con $N = 10$.

Para cada una de estas 6 bases de datos se calcula el tiempo de ejecución necesario para determinar el mejor estimador up-down-up de todos los genes que pertenecen a dicha base de datos. Utilizando esta información, se mostrará el tiempo total invertido, el tiempo medio de cada uno de los cálculos individuales, y un intervalo de confianza para este tiempo medio. Los resultados pueden verse en la tabla 2.1. Los intervalos de confianza se han calculado con una aproximación normal, salvo en el caso de los datos simulados, que debido al pequeño número de repeticiones, se utiliza un intervalo de confianza t-student.

Base datos	Nº genes	T	Tiempo tot.	Tiempo med.	D. típica	I. Confianza 95 %
48RMA	45101	24	77.142	1.71e−3	9.675e−4	(1.701e−3, 1.719e−3)
NIH3T3	45101	24	87.628	1.943e−3	6.057e−4	(1.937e−3, 1.949e−3)
Pituitary	45101	24	86.148	1.91e−3	6.062e−4	(1.905e−3, 1.916e−3)
U2OS	32321	24	63.134	1.953e−3	5.968e−4	(1.947e−3, 1.96e−3)
Genes	307	238	239.876	0.781	0.621	(0.712, 0.851)
Simulados	10	2500	17977.510	1797.751	65.227	(1757.323, 1838.179)

Tabla 2.1: Rendimiento del algoritmo original sobre las 6 bases de genes

Los resultados numéricos muestran que los tiempos de ejecución aumentan significativamente cuando T crece, es decir, es más costoso el cálculo del estimador up-down-up cuanto más largo es el período. Esto es razonable, debido a que el algoritmo original

evalúa el estimador calculado con todas las combinaciones de mínimos y máximos locales. El número de posibilidades crece aproximadamente de forma cuadrática conforme el período se hace más grande.

De la misma forma, también puede observarse que la variabilidad en el tiempo de cálculo es sustancialmente mayor con T grande. Para el caso $T = 2500$ la desviación típica es de unos 65 segundos, y se espera que sea mayor cuanto más crezca T. Sin embargo, ya se ha comentado que T suele ser 24, por lo que no será un problema en las aplicaciones prácticas. Especialmente llamativa es la desviación típica de la base "Genes", de un orden de magnitud similar a su media. Esto indica una fuerte dispersión en la distribución de los tiempos de ejecución. De hecho, el ajuste de algunos genes de esta base de datos requiere un tiempo mucho mayor, debido a que el número de candidatos a L^* y a U^* es muy variable.

Por último, también se puede afirmar que los tiempos de ejecución dependen de la base de datos elegida. En los cuatro primeros casos se tiene la misma longitud en el período, pero los intervalos de confianza para el tiempo medio muestran que hay diferencias significativas. En concreto, los genes de la base 48RMA son los que menos tardan en ajustarse, con un intervalo de confianza que muestra unos tiempos de al menos 0.2 milisegundos menos que para el resto. Aunque esta diferencia no es apreciable para el caso de $T = 24$, es interesante conocer que la eficiencia del ajuste depende de la base de genes.

2.2. Mejoras y depuración del código

En esta sección se comenzará a hablar sobre las mejoras introducidas en el código original, y más concretamente aquellas relacionadas con la depuración del código, que tiene que ver más con el estilo de programación que con el contenido de los programas. Se enumerarán las distintas modificaciones[1] y, para concluir la sección, se determinará el impacto de estas mejoras con respecto al algoritmo original.

2.2.1. Influencia de los algoritmos para calcular el PAVA

Siguiendo el pseudocódigo del algoritmo original que puede verse en la figura 2.3, el algoritmo que efectúa el PAVA es llamado un número significativo de veces, especialmente cuando el número de combinaciones entre mínimos y máximos locales es alto. Por este motivo es uno de los puntos más críticos del programa: debe ejecutarse múltiples veces y no es un paso trivial.

En el capítulo 1, seis funciones que permiten el cálculo de la regresión isotónica fueron analizadas. Las conclusiones permitieron clasificar a las funciones en varios grupos en función del tiempo invertido en el cálculo. De mayor a menor tiempo invertido se tiene:

[1] Existen algunas mejoras menores que se han implementado en relación con el estilo de programación que no merecen ser explicadas con detalle. Algunas de ellas incluye el cambio de orden de ciertas instrucciones, la adición de evaluación perezosa, la modificación de estructuras de control if-else, y la inclusión de comentarios.

- La función *gpava* es muy completa, pero muy lenta en comparación con las demás.
- Aunque *pava* es rápida para tamaños pequeños, se distancia mucho de las demás si el tamaño empieza a crecer.
- La función *isoreg* es más rápida que las anteriores, pero se me influida en gran medida por los pesos y el ruido, pudiendo llegar a ser bastante lenta en algunas situaciones.
- Las funciones *monoreg*, *pavaC* e *intcox.pavaC* son las más rápidas.

Las diferencias existentes en los tiempos de cálculo se hacen visibles a medida que el tamaño del vector de entrada aumenta. Para tamaños pequeños no se esperan grandes diferencias entre las seis funciones. Si $T > 1000$, *gpava* empieza a ser mucho más lenta. Por otro lado, *pava* comienza a comportarse peor si $T > 10000$.

En el algoritmo original, la función utilizada es *pava*. La primera mejora introducida es evidente siguiendo los puntos explicados anteriormente: se sustituye la función *pava*, por la función *pavaC*. Se espera que la eficiencia asociada a estos cambios sea mayor cuando la longitud del período aumenta (es decir, para un valor más elevado de T). Es en este caso cuando se ejecutará el cálculo del PAVA con un mayor número de puntos y, por tanto, cuando más se notará la mejoría.

2.2.2. Comprobación temprana de las condiciones de validez

Siguiendo el pseudocódigo original, para cada combinación de mínimos y máximos locales se efectúan los siguientes cálculos:

1. Calcular el PAVA de la parte creciente, desde el mínimo hasta el máximo.
2. Calcular el PAVA de la parte decreciente, desde el máximo hasta el mínimo.
3. Combinar los dos para crear el estimador up-down-up.
4. Calcular el MSE del estimador.
5. Comprobar si dicho estimador es válido y, en caso afirmativo, elegirlo como mejor actual si el MSE es menor que el mejor MSE conocido.

Sin embargo, a continuación veremos que es posible efectuar la comprobación de validez antes y, en consecuencia, continuar con los cálculos sólamente en el caso de que sea válido.

Sea $\boldsymbol{Y}$ el conjunto de puntos para el cual se va a calcular el mejor estimador up-down-up. Como sabemos, el algoritmo efectúa múltiples iteraciones, una para cada posibilidad de combinar un mínimo local con un máximo local. Sea m el mínimo local elegido en cierta iteración, y M el máximo local. Según el pseudocódigo original, se aislan los puntos Y_m y Y_M. Sea $\boldsymbol{Y}^*_{up}(m+1, M-1)$ el resultado de aplicar el PAVA creciente a $\boldsymbol{Y}$ entre los puntos $m+1$ y $M-1$ de forma cíclica, es decir, desde el siguiente al

mínimo local hasta el anterior al máximo local (en adelante se abreviará como $\boldsymbol{Y}^*_{up}$); sea $\boldsymbol{Y}^*_{down}(M+1, m-1)$ el análogo decreciente desde $M+1$ hasta $m-1$ (abreviado como $\boldsymbol{Y}^*_{down}$). No necesitamos preocuparnos de si $m < M$ o $m > M$ en la notación, pues al ser una señal periódica puede replicarse hasta que sea posible su cálculo. Se denota por $\hat{\boldsymbol{Y}}^*$ a la combinación de $\boldsymbol{Y}^*_{up}$ y $\boldsymbol{Y}^*_{down}$, junto con los puntos aislados Y_m y Y_M, para formar el estimador up-down-up.

La condición que ha de comprobarse es que $\hat{\boldsymbol{Y}}^* \in C_{mM}$ (véase la definicion 8). Comprobaremos en detalle cómo puede comprobarse esta condición para el caso $m > M$; la extrapolación hacia el caso contrario es más que evidente.

Por un lado tenemos el PAVA creciente $\boldsymbol{Y}^*_{up}$ desde $m+1$ hasta $M-1$. Por construcción, al utilizar $\boldsymbol{Y}^*_{up}$ como la parte creciente de $\hat{\boldsymbol{Y}}^*$ se tienen garantizadas las restricciones $\hat{Y}^*_{m+1} \leq \ldots \leq \hat{Y}^*_n \leq \hat{Y}^*_1 \leq \ldots \leq \hat{Y}^*_{M-1}$. De forma análoga y teniendo en cuenta el PAVA decreciente $\boldsymbol{Y}^*_{down}$ desde $M+1$ hasta $m-1$, se tienen gratantizadas las restricciones $\hat{Y}^*_{M+1} \geq \ldots \geq \hat{Y}^*_{m-1}$. Por tanto, las únicas condiciones que realmente deben comprobarse son:

$$Y_m < Y^*_{up,m+1} \tag{2.4}$$
$$Y_M > Y^*_{up,M-1} \tag{2.5}$$
$$Y_m < Y^*_{down,m-1} \tag{2.6}$$
$$Y_M > Y^*_{down,M+1} \tag{2.7}$$

Nótese que las desigualdades anteriores son estrictas, a diferencia de lo comentado en la definición 8. Esto es así porque sabemos que el máximo y el mínimo del ajuste son únicos c.s. (véase el teorema 3 y corolario 1).

Las condiciones (2.4) y (2.5) se pueden comprobar únicamente a partir de $\boldsymbol{Y}^*_{up}$. Si alguna de estas no se satisface, puede concluirse que $\hat{\boldsymbol{Y}}^* \notin C_{mM}$. Puesto que lo primero que se hace es ajustar la parte creciente, no tiene sentido seguir con el ajuste una vez que ha sido comprobado que no será válido. Sólo en el caso de que (2.4) y (2.5) sean ciertas, se procede al cálculo de $\boldsymbol{Y}^*_{down}$, que se puede utilizar a su vez para comprobar (2.6) y (2.7). Si estas dos condiciones también se cumplen, entonces se concluye que $\hat{\boldsymbol{Y}}^* \in C_{mM}$.

El pseudocódigo puede ser modificado para incluir este avance. El resultado puede verse en la figura 2.4 y compararse con el pseudocódigo de la figura 2.3. Puede apreciarse que los cambios realizados son, en líneas generales, el desglose de la verificación de $\hat{\boldsymbol{Y}}^* \in C_{mM}$ y la comprobación de las condiciones lo antes posible.

Gracias a este cambio, conseguimos mejorar los tiempos de ejecución, especialmente en aquellos casos donde el cómputo del PAVA es más costoso. En el capítulo 1 se muestra que el tiempo de cálculo de la regresión isotónica con el paquete *Iso* crece de forma cuadrática con el tamaño del vector de puntos, por lo que se puede ahorrar tiempo al no calcular un PAVA decreciente cuando no es realmente necesario y será más notorio si T es grande. Si tras comprobar las condiciones de $\boldsymbol{Y}^*_{up}$ aún es necesario seguir con el ajuste, pero se comprueba con $\boldsymbol{Y}^*_{down}$ que $\hat{\boldsymbol{Y}}^* \notin C_{mM}$, aún podemos ganar tiempo evitando la combinación de las dos regresiones isotónicas en $\hat{\boldsymbol{Y}}^*$ y el cálculo del MSE.

Entrada: $\boldsymbol{Y}, loc_M(\boldsymbol{Y}), loc_m(\boldsymbol{Y})$
Salida: $\boldsymbol{Y}^*, mse$
    $mse \leftarrow Inf$
    for each $(m, M) \in A = \big\{(m_i, M_j) : i \neq j, m_i \in loc_m(\boldsymbol{Y})$ y $M_j \in loc_M(\boldsymbol{Y}))\big\}$ **do**
        Aislar Y_m, Y_M
        $\boldsymbol{Y}^*_{up} \leftarrow$ PAVA creciente de $(Y_{m+1}, ..., Y_n, Y_1, ..., Y_{M-1})$
        if $Y_m \geq Y^*_{up,m+1}$ **or** $Y_M \leq Y^*_{up,M-1}$ **then**
            Terminar iteración
        end-if
        $\boldsymbol{Y}^*_{down} \leftarrow$ PAVA decreciente de $(Y_{M+1}, ..., Y_{m-1})$
        if $Y_m \geq Y^*_{down,m-1}$ **or** $Y_M \leq Y^*_{down,M+1}$ **then**
            Terminar iteración
        end-if
        $\hat{\boldsymbol{Y}}^* \leftarrow (\boldsymbol{Y}^*_{up}[1 : M-1], Y_M, \boldsymbol{Y}^*_{down}, Y_m, \boldsymbol{Y}^*_{up}[m+1 : n])$
        $\hat{mse}^* \leftarrow MSE(\boldsymbol{Y}, \hat{\boldsymbol{Y}}^*) = \sum_{k=1}^{n} \omega_k (Y_k - \hat{Y}^*_k)^2$
        if $\hat{mse}^* < mse$ **then**
            $\boldsymbol{Y}^* \leftarrow \hat{\boldsymbol{Y}}^*$
            $mse^* \leftarrow \hat{mse}^*$
            $mse \leftarrow \hat{mse}^*$
        end-if
    end-do

Figura 2.4: Pseudocódigo de la función mejorada para comprobar las condiciones de validez lo antes posible

2.2.3. Modificación de la búsqueda de mínimos y máximos locales

R es un lenguaje muy amigable y ofrece un entorno con numerosas funciones que pueden ser muy útiles para cualquier programador. Una de las ventajas de este tipo de lenguajes de alto nivel es que cuando el programador necesita un procedimiento para un cálculo genérico u operación no específica de un cierto contexto, es muy probable que dicha función ya haya sido implementada y puesta a disposición de los usuarios. Muchas veces dicho procedimiento estará proporcionado en las librerías base de R, pero otras sólo estará a unas pocas órdenes (o pocos clicks) de distancia, ya que la la distribución de paquetes y bibliotecas de funciones en R es abierta y cualquiera puede crear una librería.

Un ejemplo de estas funciones es la orden *cut* (R base). Aunque tiene funciones extra, básicamente toma como entrada un vector de valores numéricos y divide el rango del mismo en intervalos (que pueden ser de la misma longitud o no). A cada uno de estos intervalos se le asigna un código, y *cut* devuelve un factor donde cada elemento es el código asociado al valor numérico correspondiente. Otra función es *lm*, que permite el ajuste de modelos lineales, con un catálogo de opciones muy completo.

Un último ejemplo es la función *unique*, que recibe un array y devuelve el mismo array pero eliminando los elementos duplicados. En la implementación original (véase la figura 2.5) se utiliza esta función para garantizar que no existen duplicados en las listas de mínimos y máximos locales.

```
v2<-c(v,v)
i2<-c(1:length(v),1:length(v))
candL<-c()
candU<-c()
for(i in 2:(length(v)+1)){
  #maximos locales
  if(v2[i-1]<=v2[i] & v2[i]>=v2[i+1]){
    candiU<-i2[i]
    candU<-unique(c(candU,candiU))
  }
  #minimos locales
  if(v2[i-1]>=v2[i] & v2[i]<=v2[i+1]){
    candiL<-i2[i]
    candL<-unique(c(candL,candiL))
  }
}
```

Figura 2.5: Búsqueda de mínimos y máximos locales (código original)

El código supone que los datos originales se encuentran en el vector *v*. A grandes rasgos, el procedimiento consiste en lo siguiente:

1. Se crea *v2*, el resultado de concatenar el vector original consigo mismo.

2. Se crea *i2*, el vector que contiene el índice original al que corresponde cada posición en *v2*.

3. Se recorre desde la segunda posición de *v2* hasta la posición $length(v) + 1$. Esto equivale a recorrer el vector original empezando por la segunda posición y terminando en la primera. Para cada elemento:

    a) Se comprueba si es un máximo local, es decir, si es mayor que sus dos adyacentes. En caso afirmativo, se añade a la lista de candidatos, asegurándose primero de que dicho elemento no se añade si ya estaba incluido antes.

    b) Se comprueba, de la misma forma, si es un mínimo local.

De estas líneas de código surgen ciertos problemas:

- El uso de funciones es adecuado cuando se necesita repetir código o para la creación de un procedimiento general con un fin determinado. Es muy útil y permite un mayor nivel de abstracción, pero requiere un cambio de contexto y debe usarse cuando sea necesario, especialmente si la función va a ser llamada muchas veces.

- La función *unique* se utiliza en cada iteración del bucle para comprobar si el extremo local ya se había incluido en la lista de extremos locales. Sin embargo, la naturaleza de esta función es la de eliminar elementos duplicados. A fin de ejecutar menos código y evitar llamadas innecesarias a la función, sería mucho más rentable añadir los extremos locales a la lista directamente y, cuando se haya terminado, ejecutar la función *unique* para eliminar todos los duplicados de un sólo golpe.

- Es fácil ver (y es posible que el lector ya se haya percatado) que el uso de la función *unique* no es necesario. La lista de extremos locales comienza vacía y se van añadiendo a medida que se van encontrando. Sin embargo, los candidatos que se examinan son los índices del vector *v2* empezando en la posición 2 y hasta $length(v) + 1$. Gracias a *i2* podemos recuperar los índices originales que, en el orden en el que son examinados, serán $2, 3, ..., length(v), 1$. Es imposible que existan candidatos repetidos, por lo que el uso de la función *unique* no está justificado y puede omitirse

El resultado es que se puede reescribir la búsqueda de extremos locales para ejecutar menos instrucciones, efectuar menos cambios de contexto, y ahorrar memoria. La función modificada se puede ver en la figura 2.6.

Para la búsqueda de extremos locales se siguen los pasos que se detallan a continuación.

1. Se recorrerá el vector original v desde la primera posición hasta la última (a diferencia de antes, que se comenzaba por la segunda).

2. Si la posición que se considera es la primera o la última, hay que tener en cuenta que las posiciones adyacentes no son adyacentes realmente, sino que hay que dar un salto desde el principio hasta el final.

```
extremosLocales <- function(v){
  candL<-c()
  candU<-c()
  for(i in 1:length(v)){
    if(i == length(v)){
      iant <- i-1
      ipos <- 1
    } else if (i == 1){
      iant <- length(v)
      ipos <- i+1
    } else {
      iant <- i-1
      ipos <- i+1
    }
    #minimos locales
    if(v[iant]<=v[i] & v[i]>=v[ipos]){
      candU<-c(candU,i)
    }
    #maximos locales
    if(v[iant]>=v[i] & v[i]<=v[ipos]){
      candL<-c(candL,i)
    }
  }
  return(list(candL=candL,candU=candU))
}
```

Figura 2.6: Búsqueda de mínimos y máximos locales (código modificado)

3. Se comprueba si es un mínimo o un máximo local y en caso afirmativo, se añade a la lista de candidatos.

Como ya no se llama a la función *unique*, no hay problemas de cambios de contexto. Además, no se utiliza un segundo vector *v2*, por lo que se ahorra tanto tiempo como memoria del computador.

2.2.4. Modificación de los accesos a memoria

La última mejora de esta sección tiene que ver con los accesos a memoria. Cuando un programa cualquiera se ejecuta, primero ha de cargarse en memoria principal (memoria RAM). La memoria RAM se utiliza como memoria de trabajo de computadoras y distintos dispositivos, y almacena el Sistema Operativo, los programas, y casi todo el software que se ejecuta en una máquina [25]. Suele ser el mayor cuello de botella en cuanto al rendimiento de un computador.

El nombre de la memoria RAM proviene de *Random Access Memory*, o *Memoria de Acceso Aleatorio*. La característica de este tipo de almacenamiento es que se puede leer o escribir en una posición de memoria con un tiempo de espera idéntico (aproximadamente), independientemente de la posición elegida.

En la memoria RAM se cargan todas las instrucciones que se ejecutan de la CPU. La CPU únicamente puede acceder de forma directa a esta memoria principal. Si la CPU quiere acceder a disco, es necesario primero copiar el contenido en la memoria RAM. Es por esto que en ordenadores con poca memoria RAM, el rendimiento se ve muy perjudicado. Si un programa o unos datos no caben perfectamente en la memoria RAM, la CPU utilizará un mecanismo de memoria virtual añadida utilizando el disco. Los tiempos de acceso a disco son muchísimo mayores (del orden de milisegundos) con respecto a los accesos a memoria RAM (del orden de nanosegundos), por lo que el rendimiento del programa se ve seriamente afectado.

La memoria RAM no es el primer nivel de memoria al que accede la CPU. Cada procesador incluye una serie de registros con los que la CPU trabaja directamente y accede sin ningún retardo, más que los ciclos de lectura y escritura correspondientes. Después de los registros, se encuentra la memoria caché (puede ser una memoria caché de varios niveles). Esta memoria es extremadamente rápida, pero su capacidad es muy limitada (1MB aproximadamente). En ella se guardan los datos e instrucciones más populares. Por tanto, cuando una CPU quiere leer algo de memoria principal, tiene que ascender a través de las memorias caché hasta el procesador. Aunque los tiempos de acceso a la RAM son pequeños, conviene evitar accesos innecesarios, pues si son muchos, el cambio puede llegar a ser de segundos.

Se denomina *tiempo de acceso* al tiempo que toma a un programa o dispositivo localizar una determinada porción de memoria y hacer que esté disponible para su procesamiento [26]. Las memorias *DRAM* (*Dynamic Random Access Memory*) tienen unos tiempos de acceso de entre 50 y 150 nanosegundos. Los tiempos de acceso a disco varían entre 9 y 15 milisegundos aproximadamente, lo que supone unos 60000 accesos convencionales a una DRAM en el mejor de los casos.

El código original presenta ciertos aspectos en los que se podría optimizar el número de accesos a memoria; en concreto, los accesos que tienen que ver con la lectura de mínimos locales y máximos locales. Recordemos que se tienen en cuenta todas las combinaciones posibles, explorando en cada iteración el ajuste con un candidato a L^* y otro a U^* (para comprobar de nuevo su definición, véase el corolario 1 en la página 59). En el código original, cada vez que se produce un acceso a uno de estos candidatos, es de la forma que se puede apreciar en la figura 2.7.

```
for(i in 1:length(candL)) {
  for(j in 1:length(candU)){
    ...
    #Acceso al candidato a L*
    candL[i]
    ...
    #Acceso al candidato a U*
    candU[j]
    ...
  }
}
```

Figura 2.7: Acceso a candidatos a mínimo y máximo

El acceso a cada candidato se hace mediante *candL[i]* o *candU[j]*. Cada vez que se accede a una posición de un array, se realizan los siguientes pasos:

1. Se accede a memoria para obtener la dirección de memoria donde empieza en vector *candL* (o *candU*).

2. Se suma, a la dirección de memoria obtenida, la cantidad correspondiente (que será *i* x *tamaño de cada celda*), para obtener la dirección de memoria donde se encuentra *candL[i]* (análogamente para *candU[j]*)

3. Se accede a la posición de memoria deseada

En total, se producen dos accesos a memoria y una suma cada vez que se quiere leer el candidato.

Por el contrario, si se utilizara una aproximación como la de la figura 2.8 para la lectura, el número de accesos se reduciría hasta prácticamente la mitad. La lectura de la información relevante ya no se hace mediante *candL[i]*, sino con *indL*, lo que supone un único acceso a memoria. Adicionalmente, dentro del iterador del bucle *for* se produce otra lectura, que corresponde al cálculo de la posición de memoria donde está la información, y su escritura dentro de la variable *indL*. Esta aproximación se puede utilizar cuando el índice del elemento accedido es prescindible.

En cada una de las iteraciones del algoritmo básico se consultan aproximadamente 15 veces el valor del candidato a mínimo y máximo. Esto lleva, al utilizar el código original, a unos 15 accesos más a memoria que la versión otpimizada. En una base de

```
for(indL in candL) {
  for(indU in candU){
    ...
    #Acceso al candidato a L*
    indL
    ...
    #Acceso al candidato a U*
    indU
    ...
  }
}
```

Figura 2.8: Acceso a candidatos a mínimo y máximo (modificado)

datos que contiene genes con $T = 24$, es habitual que haya alrededor de 8 candidatos a L^* y otros 8 candidatos a U^*. Para este cálculo aproximado, supongamos que se realizan todos los cálculos, aunque ya hemos comprobado que es posible ahorrarse el cálculo de algún PAVA comprobando las condiciones de validez lo antes posible. Esto nos deja con 64 iteraciones del algoritmo, lo que hacen un total de 960 accesos innecesarios a memoria para el ajuste de cada gen. Si tenemos en cuenta que las bases de datos de genes con las que se trabaja tienen muchísimos datos, este número aumenta. En concreto, si consideramos cualquiera de las 3 primeras que se proponen en los cálculos de rendimiento, tenemos bases de datos con 45101 genes, lo que provocan un total de... ¡43 296 960 accesos innecesarios a memoria! Si consideramos 100 nanosegundos como el tiempo de acceso y que la lectura es directamente desde la memoria RAM y no la caché, entonces tenemos 4.3297 segundos de cálculo desperdiciados. No es mucho, pero varias repeticiones del algoritmo pueden hacer que esta cantidad sea significativa.

2.2.5. Mejora con respecto al original

Antes de concluir la sección, se pretende mostrar el impacto que las mejoras comentadas tienen, de forma práctica, en la implementación.

El experimento mostrado en la sección anterior con el código original se ha replicado, pero utilizando esta vez el código modificado con toda la depuración mencionada. Los resultados se encuentran en la tabla 2.2. Todos los tiempos están, de nuevo, medidos en segundos.

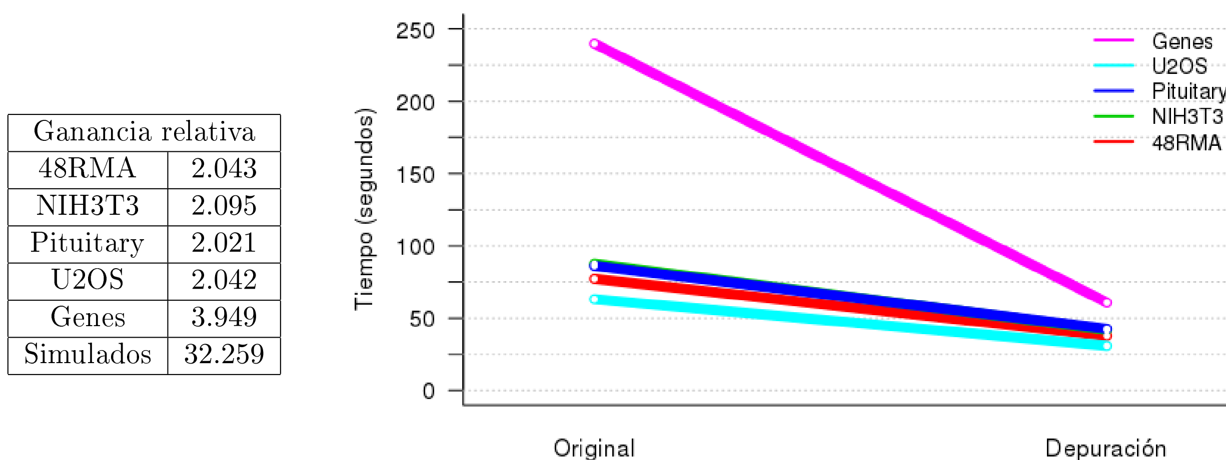
Es posible apreciar una reducción drástica en los tiempos de cálculo, simplemente mediante la depuración de la programación. En las bases de genes con $T = 24$, los tiempos de ejecución se reducen hasta la mitad. En las otras dos, donde T va siendo cada vez mayor, el aumento es más significativo. En la base de datos “Genes” se cuadruplica el rendimiento, mientras que en la base simulada con $T = 2500$, la ganancia de tiempo llega a ser de hasta 32, con respecto al original. La desviación estándar de los tiempos también es cada vez menor, lo que permite unos intervalos de confianza más estrechos y la certeza de que los tiempos serán parecidos en todas las ocasiones. En concreto,

Base datos	Nº genes	T	Tiempo tot.	Tiempo med.	D. típica	I. Confianza 95 %
48RMA	45101	24	37.766	8.374e−4	6.862e−4	(8.310e−4, 8.437e−4)
NIH3T3	45101	24	41.830	9.275e−4	4.087e−4	(9.237e−4, 9.312e−4)
Pituitary	45101	24	42.630	9.452e−4	4.021e−4	(9.415e−4, 9.489e−4)
U2OS	32321	24	30.915	9.565e−4	3.971e−4	(9.522e−4, 9.608e−4)
Genes	307	238	60.750	0.198	0.103	(0.186, 0.209)
Simulados	10	2500	557.288	55.729	1.794	(54.617, 56.841)

Tabla 2.2: Rendimiento del algoritmo depurado sobre las 6 bases de genes

para los datos simulados, la desviación típica del tiempo pasa a ser de 65.227 a tan sólo 1.794.

En la figura 2.9 se muestra una tabla con las ganancias exactas de tiempo, así como un gráfico que permite visualizar de forma rápida la evolución de los tiempos en todos los casos salvo en los datos simulados. Dado que la ganancia en esta última base de datos es tan grande y los tiempos de ejecución son mayores que en el resto, añadirla al gráfico no haría sino impedir la correcta visualización del resto.



Ganancia relativa	
48RMA	2.043
NIH3T3	2.095
Pituitary	2.021
U2OS	2.042
Genes	3.949
Simulados	32.259

Figura 2.9: A la izquierda, ganancia relativa del tiempo medio de la versión depurada con respecto a la original. A la derecha, evolución de los tiempos de ejecución (excepto para la base de datos simulada).

2.3. Mejoras derivadas de resultados teóricos

En esta sección se tratan temas relacionados con resultados teóricos que permiten mejorar la eficiencia del algoritmo y reducir el espacio de búsqueda. Estos resultados están relacionados con el algoritmo PAVA, por lo que resulta conveniente conocer con precisión la aplicación y el pseudocódigo del algoritmo descritos en el capítulo 1. Seguidamente, se mostrarán ciertos resultados de utilidad y se discutirá la forma de aplicación de los mismos en la implementación práctica que estamos tratando, para producir un nuevo pseudocódigo mejorado reduciendo el espacio de búsqueda. Para concluir la sección se mostrará la mejora obtenida en comparación con la implementación mejorada de la depuración de la programación y con el original.

2.3.1. Reducción del espacio de búsqueda

Antes de presentar los resultados teóricos, repasemos la notación que se utilizará en esta sección.

Sea $\boldsymbol{Y}$ la información resumida de la expresión de un gen a lo largo de todos sus períodos, y n la longitud del vector $\boldsymbol{Y}$. Se denota por $\boldsymbol{Y}^*$ el mejor estimador Up-Down-Up para $\boldsymbol{Y}$ en el sentido de mínimos cuadrados, con $L^* = \arg\min_{i=1,\dots,n} Y_i^*$ y $U^* = \arg\max_{i=1,\dots,n} Y_i^*$. En cada una de las iteraciones básicas del algoritmo para el cálculo de $\boldsymbol{Y}^*$, sea m el candidato actual a L^* y M el candidato a U^*. El conjunto de mínimos locales de $\boldsymbol{Y}$ se escribe como $loc_m(\boldsymbol{Y}) = \{i : Y_i \text{ es un mínimo local de } \boldsymbol{Y}\}$, mientras que $loc_M(\boldsymbol{Y}) = \{i : Y_i \text{ es un máximo local de } \boldsymbol{Y}\}$ se refiere a los máximos locales. Por último, se considera $\boldsymbol{Y}_{up}^*(m+1, M-1)$, el vector $(M-m-1)$-dimensional (suponiendo $M > m$) resultado de aplicar el PAVA creciente a $\boldsymbol{Y}$ entre los puntos $m+1$ y $M-1$(abreviado como $\boldsymbol{Y}_{up}^*$ siempre que m y M estén perfectamente claros); y $\boldsymbol{Y}_{down}^*(M+1, m-1)$ el análogo decreciente desde $M+1$ hasta $m-1$ (abreviado como $\boldsymbol{Y}_{down}^*$). El subíndice asociado a estos vectores, si aparece, denota no la posición real del elemento, sino el término al que está asociado. Por ejemplo, $Y_{up,m+1}^*$ se refiere a la primera posición de $\boldsymbol{Y}_{up}^*$, que es la que está asociada a Y_{m+1}.

El objetivo es mostrar que, tras ciertas condiciones, no es necesario proseguir en la búsqueda de L^* y U^*. Los resultados que se muestran a continuación permiten reducir el número de PAVA calculados, a partir de las condiciones de validez que hay que comprobar.

Proposición 1. Si $Y_{up,m+1}^*(m+1, M-1) \leq \delta$, con m fijo, entonces para cualquier otro M' tal que $(M'-m) \bmod n \geq (M-m) \bmod n$, con $1 \leq M' \leq n$, se tiene que $Y_{up,m+1}^*(m+1, M'-1) \leq \delta$.

Nótese que la condición sobre M' no es más que añadir más puntos al cálculo del PAVA creciente fijado m. M' está más a la derecha que M, por lo que el PAVA hasta M' involucra más puntos que teniendo el corte en M. El operador módulo es necesario en el caso en que $M' < m$ o $M < m$. De esta manera, $(M-m) \bmod n-1$ es la longitud del PAVA creciente $\boldsymbol{Y}_{up}^*(m+1, M-1)$. $(M'-m) \bmod n-1$ indica la longitud de $\boldsymbol{Y}_{up}^*(m+1, M'-1)$. M' se elige de forma que la longitud del nuevo PAVA creciente sea mayor que antes.

Demostración. La prueba se realizará por inducción sobre la diferencia de longitudes $A = (M' - m) \text{ mód } n - (M - m) \text{ mód } n$.

Paso base. $A = 0$. En este caso $(M' - m) \text{ mód } n = (M - m) \text{ mód } n$ y la longitud de los dos PAVA es la misma. De $A = 0$ se deduce que $M' = M + i \cdot n, i = 0, 1, ...,$ pero como $1 \leq M' \leq n$, necesariamente $M' = M$. En consecuencia, trivialmente $Y^*_{up,m+1}(m + 1, M' - 1) = Y^*_{up,m+1}(m + 1, M - 1) \leq \delta$.

Paso inductivo. Supongamos que es cierto para algún A, y veamos que es cierto para $A + 1$. Si consideramos $A = (M' - m) \text{ mód } n - (M - m) \text{ mód } n$, entonces la hipótesis inductiva es $Y^*_{up,m+1}(m+1, M'-1) \leq \delta$. En el caso de $A+1$, la longitud del nuevo PAVA es una unidad mayor que en el caso anterior. Por tanto, lo que se quiere demostrar en el paso inductivo es $Y^*_{up,m+1}(m + 1, M') \leq \delta$. El nuevo punto añadido en el PAVA es $Y_{M'}$.

Sean $a'_1, ..., a'_j$ los j conjuntos de nivel de $\boldsymbol{Y}^*_{up}(m + 1, M' - 1)$. Nótese que necesariamente $Y^*_{up,m+1}(m + 1, M' - 1) = a'_1$ y $Y^*_{up,M'-1}(m + 1, M' - 1) = a'_j$. Según el pseudocódigo del PAVA (véase la figura 1.5), al añadir un punto por la derecha nada cambia hasta llegar a la última iteración, considerando el punto $Y_{M'}$. Por tanto, el cómputo de $\boldsymbol{Y}^*_{up}(m+1, M')$ es exactamente igual al de $\boldsymbol{Y}^*_{up}(m+1, M'-1)$, salvo que el primero tiene una iteración a mayores. En esta última iteración, pueden ocurrir dos cosas:

- Si $Y_{M'} \geq a'_j$, la última iteración del PAVA no viola las restricciones de orden. El nuevo punto $Y_{M'}$ simplemente se añade a $\boldsymbol{Y}^*_{up}(m + 1, M' - 1)$ para formar $\boldsymbol{Y}^*_{up}(m + 1, M')$. Como el único cambio ha sido en el último punto, $Y^*_{up,m+1}(m + 1, M') = Y^*_{up,m+1}(m + 1, M' - 1) \leq \delta$ (H. ind.), y el teorema se cumple.
- Por otro lado, si $Y_{M'} < a'_j$, se viola la restricción de orden. Siguiendo el pseudocódigo del PAVA, se calcula la media ponderada de $Y_{M'}$ y a'_j para formar un nuevo conjunto de nivel a''_j. Necesariamente $a''_j < a'_j$. Es posible que se viole de nuevo una restricción de orden. En caso de que eso ocurra, habrá que combinar los conjuntos de nivel a'_{j-1} y a''_j para formar $a''_{j-1} < a'_{j-1}$. La única manera de que se modifique el valor del punto $Y^*_{up,m+1}(m + 1, M')$ es que las restricciones de orden se violen hasta modificar a'_1. Esto ocurre si $a''_2 < a'_1$, en cuyo caso se combinarían para formar $a''_1 < a'_1$. El nuevo punto $Y^*_{up,m+1}(m + 1, M') = a''_1 < a'_1 = Y^*_{up,m+1}(m + 1, M' - 1)$. De nuevo, con la hipótesis inductiva $Y^*_{up,m+1}(m+1, M'-1) \leq \delta$, se concluye que $Y^*_{up,m+1}(m+1, M') < Y_m$.

Con esto se consigue probar que para cualquier M' por la derecha de M, $Y^*_{up,m+1}(m + 1, M') \leq Y^*_{up,m+1}(m + 1, M' - 1)$. □

A partir de la proposición 1, podemos concluir el siguiente teorema que permite reducir el espacio de búsqueda.

Teorema 4. Si $m = L^*$ y $Y^*_{up,m+1}(m+1, M-1) \leq Y_m$, entonces $U^* \in loc_M(\boldsymbol{Y}) \cap \{m+1, ..., M-1\}$.

Demostración. En todo caso se supone que $m = L^*$, es decir, m es el mínimo que provoca el mejor estimador $\boldsymbol{Y}^*$. Sabemos que debe existir un U^* que verifique las condiciones necesarias para que $\boldsymbol{Y}^* \in C_{mU^*}$. Ya vimos que, por construcción, las únicas condiciones que necesitaban ser validadas para comprobar esto eran (2.4), (2.5), (2.6) y (2.7) (página 64). Si no se cumple (2.4), como en la hipótesis de este teorema, se tiene que $Y^*_{up,m+1}(m+1, M-1) \leq Y_m$, el estimador resultante no será válido y, por tanto, $M \neq U^*$. Aplicamos la proposición 1 con $\delta = Y_m$, con lo que se concluye que para cualquier M' por la derecha de M se tiene que $Y^*_{up,m+1}(m+1, M'-1) \leq Y_m$ y tampoco cumple (2.4). Para todo M' así escogido $\hat{\boldsymbol{Y}}^* \notin C_{mM'}$. El punto U^* debe encontrarse estrictamente antes de M, puesto que ni éste ni los siguientes serán válidos. □

La consecuencia directa de este teorema es evidente. Supongamos que en un momento dado se explora la posibilidad de m como mínimo Up-Down-Up. Hay que ir probando con diferentes candidatos a máximo M, de manera que se realiza el ajuste completo para comprobar si el resultado es válido. Sin embargo, ahora sabemos que si para algún M se viola la restricción de m por la derecha, entonces no es necesario seguir con el cálculo probando con valores de M por la derecha.

Para completar la reducción de la búsqueda, es evidente preguntarse si esta mejora también puede aplicarse al cálculo del PAVA decreciente. La respuesta es que sí, como se puede ver a continuación.

El cálculo del PAVA decreciente es inmediato a partir del cálculo del PAVA creciente. Puede considerarse como un cálculo de un PAVA creciente en el sentido contrario, como sugiere el pseudocódigo de la figura 2.10

Entrada: Valores $a_1, ..., a_n$ y pesos $w_1, ..., w_n$
Salida: Valores no crecientes $y_1, ..., y_n$ minimizando $\sum_{i=1}^{n} w_i(y_i - a_i)^2$
$\boldsymbol{a}' \leftarrow reverse(\boldsymbol{a})$
$\boldsymbol{w}' \leftarrow reverse(\boldsymbol{w})$
$\boldsymbol{y}' \leftarrow$ PAVA creciente de $\boldsymbol{a}'$ y $\boldsymbol{w}'$
$\boldsymbol{y} \leftarrow reverse(\boldsymbol{y}')$

Figura 2.10: Pseudocódigo del algoritmo PAVA decreciente

La siguiente proposición es análoga a la proposición 1 en el caso decreciente.

Proposición 2. Si $Y^*_{down,m-1}(M+1, m-1)$ denota el último elemento del vector $\boldsymbol{Y}^*_{down}$, el asociado a Y_{m-1}, y $Y^*_{down,m-1}(M+1, m-1) \leq \delta$, con m fijo, entonces para cualquier otro M' tal que $(m - M') \bmod n \geq (m - M) \bmod n$ (M' por la izquierda de M), con $1 \leq M' \leq n$, se tiene que $Y^*_{down,m-1}(M'+1, m-1) \leq \delta$.

Demostración. Consideremos los puntos $Y_M, Y_{M+1}, ..., Y_{m-1}, Y_m$. Queremos calcular el PAVA decreciente $\boldsymbol{Y}^*_{down}(M+1, m-1)$. Con este objetivo, vamos a tomar los puntos considerados en sentido inverso, es decir, sea $Y'_m = Y_m, Y'_{m+1} = Y_{m-1}, ..., Y'_{M^*-1} = Y_{M+1}, Y'_{M^*} = Y_M$. La figura 2.11 muestra de forma gráfica la reversión de los puntos. Nótese que la distancia entre los puntos Y_m y Y_M es la misma que entre Y'_m y Y'_{M^*}.

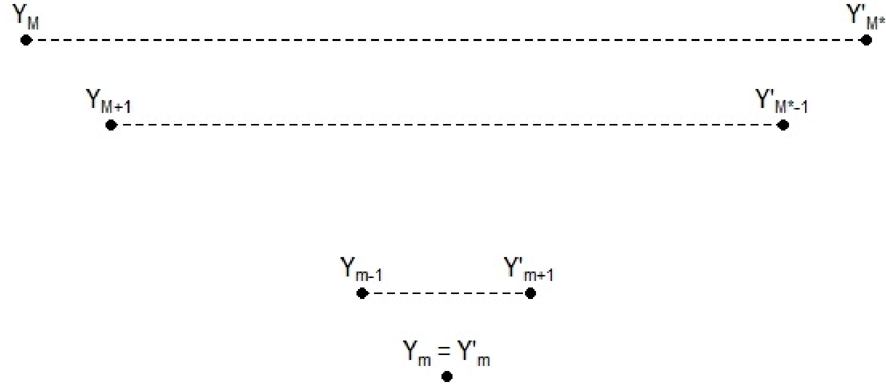


Figura 2.11: Revertir los puntos en el cálculo del PAVA decreciente (ayuda para la demostración de la proposicion 2)

Sea $\boldsymbol{Y}'^{*}_{up}(m+1, M^*-1)$ el PAVA creciente de $Y'_{m+1}, ..., Y'_{M^*-1}$. Según el pseudocódigo de la figura 2.10, podemos conseguir el PAVA decreciente buscado invirtiendo el orden de $\boldsymbol{Y}'^{*}_{up}(m+1, M^*-1)$. De esta manera,

$$Y^*_{down,M+1}(M+1, m-1) = Y'^{*}_{up,M^*-1}(m+1, M^*-1) \tag{2.8}$$

$$\vdots$$

$$Y^*_{down,m-1}(M+1, m-1) = Y'^{*}_{up,m+1}(m+1, M^*-1) \tag{2.9}$$

Evidentemente, las longitudes de $\boldsymbol{Y}'^{*}_{up}(m+1, M^*-1)$ y $\boldsymbol{Y}^*_{down}(M+1, m-1)$ deben coincidir. La longitud del vector $\boldsymbol{Y}'^{*}_{up}(m+1, M^*-1)$ es (M^*-m) mód $n-1$, mientras que la longitud de $\boldsymbol{Y}^*_{down}(M+1, m-1)$ será $(m-M)$ mód $n-1$ (de nuevo, los operadores módulo se utilizan porque los datos son circulares). Fijo m, a cada valor de M le corresponde un valor de M^* de manera que

$$(M^*-m) \text{ mód } n = (m-M) \text{ mód } n \tag{2.10}$$

La proposición 1 aplicada a $\boldsymbol{Y}'^{*}_{up}(m+1, M^*-1)$ reza de la siguiente manera: Si $Y'^{*}_{up,m+1}(m+1, M^*-1) \leq \delta$, con m fijo, entonces para cualquier otro $M^{*'}$ tal que $(M^{*'}-m)$ mód $n \geq (M^*-m)$ mód n, con $1 \leq M^{*'} \leq n$, se tiene que $Y'^{*}_{up,m+1}(m+1, M^{*'}-1) \leq \delta$. Utilizando las relaciones (2.9) y (2.10), podemos traducir esta afirmación a la proposición que queríamos demostrar. □

Una vez que tenemos la proposición anterior clara, el resultado que le sigue es evidente.

Teorema 5. Si $m = L^*$ y $Y^*_{down,m-1}(M+1, m-1) \leq Y_m$, entonces $U^* \in loc_M(\boldsymbol{Y}) \cap \{M+1, ..., m-1\}$.

Demostración. En todo caso se supone que $m = L^*$, es decir, m es el mínimo que provoca el mejor estimador $\boldsymbol{Y}^*$. Sabemos que debe existir un U^* que verifique las condiciones necesarias para que $\boldsymbol{Y}^* \in C_{mU^*}$. Ya vimos que, por construcción, las únicas

condiciones que necesitaban ser validadas para comprobar esto eran (2.4), (2.5), (2.6) y (2.7) (página 64). Si no se cumple (2.6), como en la hipótesis de este teorema, se tiene que $Y^*_{down,m-1}(M+1, m-1) \leq Y_m$, el estimador resultante no será válido y, por tanto, $M \neq U^*$. Aplicamos la proposición 2 con $\delta = Y_m$, con lo que se concluye que para cualquier M' por la izquierda de M se tiene que $Y^*_{down,m-1}(M'+1, m-1) \leq Y_m$ y tampoco cumple (2.6). Para todo M' así escogido $\hat{\boldsymbol{Y}}^* \notin C_{L^*M'}$. El punto U^* debe encontrarse estrictamente después de M, puesto que ni éste ni los anteriores serán válidos. □

Gracias a estos resultados conseguimos reducir la búsqueda de máximos locales en ambos sentidos. El pseudocódigo puede ser mejorado para no realizar búsquedas innecesarias, cuando se ha comprobado que tras ciertas condiciones no hace falta seguir probando con máximos locales. El resultado final puede verse en la figura 2.12.

El algoritmo ahora es más complejo que en el caso anterior, pero se consigue reducir el espacio de búsqueda en ciertos casos. En primer lugar, se elige un cierto m como candidato a L^* y, a diferencia de antes, se exploran todas las posibilidades de PAVA creciente antes de hacer el PAVA decreciente. Se han añadido dos conjuntos adicionales:

MValidos es un conjunto que, para un m fijo, contendrá todos aquellos candidatos a U^* que verifican las condiciones relacionadas con $\boldsymbol{Y}^*_{up}$ (condiciones 2.4 y 2.5).

PAVASCrecientes es un conjunto que, fijado m, contiene todos los PAVAS válidos de los anteriores candidatos.

Si en la exploración de un cierto candidato M no se cumple la condición $Y_m < Y^*_{up,m+1}$, por el teorema 4 sabemos que ningún candidato M' que se encuentre por la derecha de M será válido, y podemos terminar la búsqueda de candidatos a U^* (se corresponde con el primer *break*). Si por el contrario esta condición sí que se cumple, aún falta por comprobar si $Y_M > Y^*_{up,M-1}$. Cuando esta condición se cumpla, entonces se verifica lo necesario por la parte de $\boldsymbol{Y}^*_{up}$ y se guarda M como un posible candidato, junto con el PAVA creciente asociado.

Una vez explorados todos los candidatos válidos en relación con las condiciones de $\boldsymbol{Y}^*_{up}$, es necesario realizar el ajuste del PAVA decreciente. Según el teorema 5, si para algún cierto M no se cumple $Y^*_{down,m-1} > Y_m$, sabemos que para cualquier otro M' situado a la izquierda de M tampoco se cumplirá. Por ese motivo, comenzamos la búsqueda desde el primer M válido a la izquierda de m (*reverse*). Si en algún momento se detecta que $Y^*_{down,m-1} \leq Y_m$, la búsqueda se detiene. En caso de que esta condición sí sea válida, se debe comprobar la última desigualdad: $Y_M > Y^*_{down,M+1}$. Cuando se verifique esta última condición, se pueden combinar $Y_m, \boldsymbol{Y}^*_{up}, Y_M$ y $\boldsymbol{Y}^*_{down}$ para formar $\hat{\boldsymbol{Y}}^*$, cumpliéndose que $\hat{\boldsymbol{Y}}^* \in C_{mM}$. Con los datos del nuevo estimador generado, se guarda como mejor opción si su MSE es más bajo que el mejor MSE hasta el momento.

2.3.2. Comentarios generales y mejora obtenida

Es importante destacar, además del ahorro en las búsquedas, que el nuevo algoritmo es ligeramente más complejo que el anterior. A diferencia del que se tiene en la figura

Entrada: $\boldsymbol{Y}, loc_M(\boldsymbol{Y}), loc_m(\boldsymbol{Y})$
Salida: $\boldsymbol{Y}^*, mse$
    $mse \leftarrow Inf$
    for each $m \in loc_m(\boldsymbol{Y})$ **do**
        $MValidos \leftarrow \{\}$
        $PAVASCrecientes \leftarrow \{\}$
        for each $M \in \{m+1, ..., n, 1, ..., m\} \cap loc_M(\boldsymbol{Y})$ **do**
            $\boldsymbol{Y}^*_{up} \leftarrow$ PAVA creciente de $(Y_{m+1}, ..., Y_n, Y_1, ..., Y_{M-1})$
            if $Y_m \geq Y^*_{up,m+1}$ **then**
                break
            end-if
            if $Y_M > Y^*_{up,M-1}$ **then**
                $MValidos \leftarrow MValidos \cup \{M\}$
                $PAVASCrecientes \leftarrow PAVASCrecientes \cup \{\boldsymbol{Y}^*_{up}\}$
            end-if
        end-do
        for each $M \in reverse(MValidos)$ **do**
            $\boldsymbol{Y}^*_{down} \leftarrow$ PAVA decreciente de $(Y_{M+1}, ..., Y_{m-1})$
            if $Y_m \geq Y^*_{down,m-1}$ **then**
                break
            end-if
            if $Y_M > Y^*_{down,M+1}$ **then**
                Aislar Y_m y Y_M
                $\boldsymbol{Y}^*_{up} \leftarrow$ Recuperar PAVA de la lista $PAVASCrecientes$
                $\hat{\boldsymbol{Y}}^* \leftarrow (\boldsymbol{Y}^*_{up}[1 : M-1], Y_M, \boldsymbol{Y}^*_{down}, Y_m, \boldsymbol{Y}^*_{up}[m+1 : n])$
                $\hat{mse}^* \leftarrow MSE(\boldsymbol{Y}, \hat{\boldsymbol{Y}}^*) = \sum_{k=1}^{n} \omega_k (Y_k - \hat{Y}^*_k)^2$
                if $\hat{mse}^* < mse$ **then**
                    $\boldsymbol{Y}^* \leftarrow \hat{\boldsymbol{Y}}^*$
                    $mse^* \leftarrow \hat{mse}^*$
                    $mse \leftarrow \hat{mse}^*$
                end-if
            end-if
        end-do
    end-do

Figura 2.12: Pseudocódigo de la función mejorada para comprobar las condiciones de validez lo antes posible

2.4, en el nuevo pseudocódigo se tienen más accesos a memoria y más estructuras de datos que mantener. En concreto, los cambios más importantes son:

- Se añaden las listas *MValidos* y *PAVASCrecientes*. Estas listas deben inicializarse múltiples veces, y en ocasiones puede ser necesario almacenar muchos datos sobre PAVAS crecientes (especialmente si el período es grande). En definitiva, la complejidad espacial y la ocupación de memoria es significativamente superior al algoritmo base, así como el no despreciable coste de su inicialización.
- El impacto que tienen estos avances será mayor cuanto mayor sea el período de los datos. Para bases de datos con genes con T pequeño, el número de candidatos a mínimo y máximo que se ha de explorar por gen no es muy elevado, además de que es más probable que no se cumplan las condiciones que permiten reducir el espacio de búsqueda. Sin embargo, en casos con T más alto, la reducción de la búsqueda puede ser mucho más significativa.
- Se añaden más instrucciones en general, como por ejemplo las necesarias para cambiar el orden de búsqueda de los M válidos (recordemos que se empieza por la derecha de m), las necesarias para guardar los candidatos válidos y el PAVA creciente, instrucciones de control de bucles, etc. Todo ello contribuye a aumentar el tiempo de ejecución del sistema. De hecho, si en algún gen no se puede reducir la búsqueda y se exploran todas las posibilidades, el número de instrucciones ejecutadas en el nuevo pseudocódigo será mayor que antes y, en consecuencia, también su tiempo de ejecución. Por lo tanto, el aumento de la complejidad será rentable siempre que se pueda reducir de alguna manera el espacio de búsqueda.

El experimento para comprobar los tiempos de cálculo se extiende ahora incluyendo las mejoras relativas a resultados teóricos. Los resultados se encuentran en la tabla 2.3.

Base datos	Nº genes	T	Tiempo tot.	Tiempo med.	D. típica	I. Confianza 95 %
48RMA	45101	24	36.946	8.192e−4	7.436e−4	(8.123e−4, 8.26e−4)
NIH3T3	45101	24	40.474	8.974e−4	4.24e−4	(8.935e−4, 9.013e−4)
Pituitary	45101	24	42.246	9.367e−4	4.1e−4	(9.329e−4, 9.405e−4)
U2OS	32321	24	30.447	9.42e−4	4.065e−4	(9.376e−4, 9.464e−4)
Genes	307	238	38.425	0.125	0.025	(0.122, 0.128)
Simulados	10	2500	389.784	38.978	1.357	(38.138, 39.819)

Tabla 2.3: Rendimiento del algoritmo con mejoras de resultados teóricos sobre las 6 bases de genes

Los resultados numéricos no difieren tanto del código que tiene únicamente la depuración, lo que significa que el impacto de las mejoras añadidas en este apartado es menor. En las cuatro primeras bases de datos, que se refieren a los genes reales con $T = 24$ los tiempos de ejecución se reducen muy poco, aunque los intervalos de confianza para el tiempo medio no se solapan (salvo en el caso de Pituitary), lo que indica que realmente sí es una mejora significativa.

Es llamativo que en todos los casos salvo en la base “Genes”, la desviación típica en los tiempos de cálculo aumenta. El espacio de búsqueda sólo se puede reducir en

algunos de los casos, por lo que es lógico que la dispersión en los tiempos sea mayor. Para “Genes”, la disminución en la desviación típica es importante, pues antes era del orden de la media.

Para las bases de datos con T mayor la mejora es más notoria, reduciéndose hasta casi un 40 % el tiempo en la base de datos “Genes”, y alrededor de un 30 % en los datos simulados. La ganancia relativa de tiempo de la versión actual del código en comparación con el original y con la mejora de la depuración puede encontrarse en la tabla 2.4. Adicionalmente, la evolución de los tiempos de ejecución hasta este punto puede encontrarse en la figura 2.13.

Ganancia rel. respecto...	Original	Depuración
48RMA	2.088	1.022
NIH3T3	2.165	1.034
Pituitary	2.039	1.009
U2OS	2.074	1.015
Genes	6.243	1.581
Simulados	46.121	1.43

Tabla 2.4: Ganancia relativa de la versión con mejoras de resultados teóricos con respecto a la función original y a la que contiene sólo la depuración

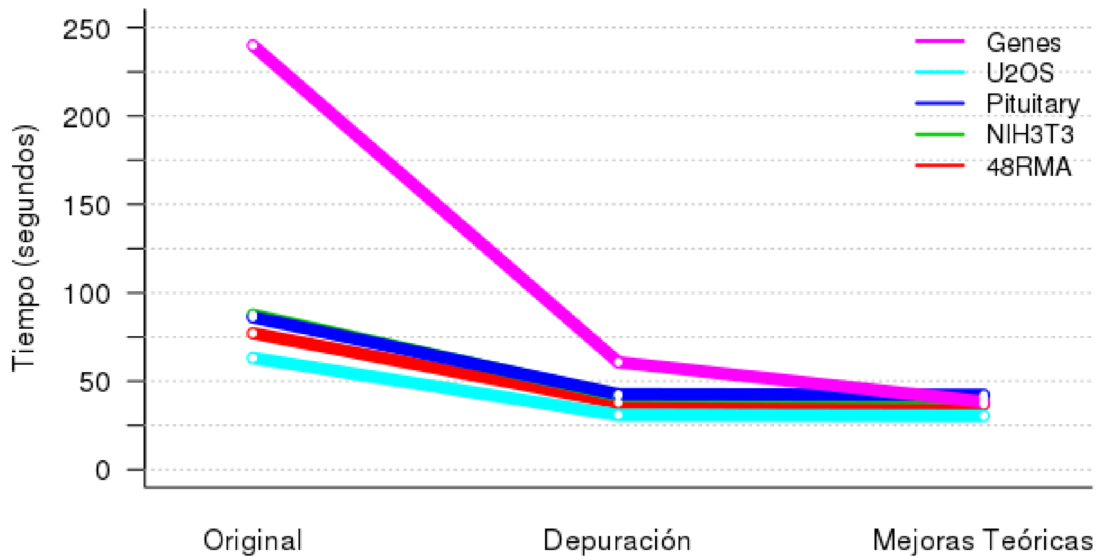


Figura 2.13: Evolución en los tiempos de ejecución hasta las mejoras teóricas (excluyendo la base de datos simulada).

Una última mejora que puede considerarse es la modificación de la función del cálculo del PAVA para que no sea necesario volver a iniciar el cálculo desde el principio cuando sea posible. Cuando hemos hablado del pseudocódigo del PAVA, hemos visto que es un algoritmo iterativo incremental, y el cálculo de dos PAVA’s que involucren los

mismos puntos al comienzo es idéntico al principio. En consecuencia, podría modificarse la función para permitir este incremento en el conjunto de puntos y no repetir cálculos que ya habían sido realizados.

Desafortunadamente, esta modificación no resulta rentable de forma práctica. La función *pavaC* que se utiliza no está escrita en R, por lo que existe una complicación adicional para guardar los datos de distintas ejecuciones. Se ha comprobado de forma empírica que este guardado tiene un coste adicional que supera el reinicio del cálculo desde el principio. La función *pavaC* es una función simple con unos tiempos de ejecución realmente bajos, y añadir complejidad a la función no sale rentable.

2.4. Reescritura de la función en C

Los lenguajes de programación pueden dividirse y clasificarse de acuerdo a muchos factores. Uno de ellos es el nivel de abstracción que permiten, y distingue los lenguajes de programación en dos grupos.

Los **lenguajes de alto nivel** se asemejan a nuestra forma de razonar, permitiendo que el programador evada detalles técnicos innecesarios a costa de una pérdida en la eficiencia.

Los **lenguajes de bajo nivel** controlan directamente la circuitería del ordenador. Programar en ellos es muchísimo más complicado, pero son mucho más eficientes en compensación.

Las ventajas fundamentales de los lenguajes de alto nivel con respecto a los de bajo nivel son, fundamentalmente, la sencillez, uniformidad y portabilidad de los programas [27]. No obstante, aunque el programador no trate detalles técnicos, todo programa debe traducirse a lenguaje máquina para que el ordenador pueda ejecutar instrucciones. Para ello, existen dos posibilidades:

1. Por un lado, la **compilación** traduce el programa completo a código máquina antes de su ejecución.
2. Por otro lado, la **interpretación** traduce el código línea a línea a medida que se va ejecutando.

El lenguaje C es un lenguaje intermedio en cuanto al nivel de abstracción. Es un lenguaje de propósito general y muy portable. Muchísimos programas se continúan escribiendo en C hoy en día, a pesar de ser un lenguaje bastante antiguo (*Dennis Ritchie*, 1972), pues permite la ejecución eficiente de código sin tener que llegar a un nivel de abstracción más próximo al código máquina o ensamblador. Es compilado, por lo que no es necesario invertir tiempo convirtiendo las instrucciones a lenguaje máquina durante su ejecución.

R es un entorno y lenguaje de programación [28] con un enfoque al análisis estadístico. Apareció en 1993 (*Ross Ihaka y Robert Gentleman*) y adopta un enfoque de alto

nivel. Se trata de un software libre diseñado para la comunidad estadística con enfoques de minería de datos, investigación biomédica, bioinformática y matemáticas financieras. Permite una gran flexibilidad, como cualquier usuario de R conoce, pero a costa de una menor eficiencia en la ejecución donde, la interpretación es, en gran medida, su causa.

En esta sección se evaluará el impacto que tiene pasar a C la función del cálculo del estimador Up-Down-Up. La función original que estamos considerando toma como entradas el vector de datos y sus extremos locales. El cálculo de los extremos locales se mantendrá en R, pero la búsqueda del mejor estimador se realizará en C.

2.4.1. Cambios y consideraciones necesarias

La reescritura de una función a C no es una tarea sencilla. La función original en R ocupa 396 líneas de código, pero el paso a C hacen necesarias 777 líneas. Además, las diferencias en el estilo de programación, las órdenes, funciones, reserva de memoria, etc., imposibilitan que la tarea sea una simple traducción. Los cambios más importantes que se producen en este proceso son los siguientes.

- Todas las variables y estructuras de datos que se utilizan a lo largo del algoritmo deben declararse en el preámbulo de la función.
- Es necesario definir el tipo de cada variable y estructura de datos.
- Debe tenerse en cuenta la llamada a la función de C desde R. En este caso se utiliza la interfaz *.Call*.
- Se debe reservar memoria de forma manual para todas las estructuras de datos no básicas que requieran más de una posición de memoria, utilizando *malloc*.
- Se utilizan punteros a direcciones de memoria para manejar los vectores.
- Todas las variables y estructuras de datos que se utilizan tanto en C como en R son las mismas y están en el formato por defecto de R. Para poder acceder a ellas desde C, se debe declarar el tipo de datos como *SEXP* y utilizar las funciones *REAL*, *INTEGER*, etc. proporcionadas por el wrapper de C.
- El resultado que devuelve la función C se almacena en una estructura de datos que ha de ser reservada utilizando la función *mkNamed*.
- Con el fin de que el recolector de basura de R no elimine las estructuras de datos mientras se está trabajando en C, se utiliza la orden *PROTECT* para proteger las posiciones de memoria involucradas. Antes de que la función C retorne, se ejecuta la orden *UNPROTECT*, que elimina dicha protección.
- La memoria utilizada por las estructuras de datos de C debe ser liberada manualmente con la orden *free*.
- Ciertas órdenes y aspectos de R no pueden ser traducidos directamente a C y debe programarse manualmente. Es el caso, por ejemplo, de la recuperación de elementos de una lista por nombre en lugar de por índice, así como la selección de elementos de un vector que satisfacen alguna condición.

- R puede manejar de forma abstracta un número muy grande bajo el nombre de "Inf", pero en C es necesario proporcionar de forma exacta el número concreto.

2.4.2. Impacto de la reescritura en C

Como se mostrará a continuación, el beneficio de ejecutar el algoritmo en C es muy significativo. En la tabla 2.5 se encuentran los tiempos en segundos del experimento que se lleva realizando en todo el escrito. Estos tiempos han disminuido drásticamente en todos los casos, así como la desviación típica. La tabla 2.6 muestra la ganancia de tiempo relativa de esta versión con respecto a todas las anteriores. Para las cuatro primeras bases de datos, los tiempos de ejecución son aproximadamente 12 veces menores que con respecto a la original. Para la base "genes", la ganancia es de 86 unidades, mientras que la base de datos con $T = 2500$ tarda 215 veces menos. Este último es un gran avance, si pensamos que al principio el ajuste individual llevaba 30 minutos (en media), y ahora se logra en 8 segundos.

Base datos	Nº genes	T	Tiempo tot.	Tiempo med.	D. típica	I. Confianza 95 %
48RMA	45101	24	6.594	1.462e−4	3.581e−4	(1.429e−4, 1.495e−4)
NIH3T3	45101	24	6.434	1.427e−4	3.497e−4	(1.394e−4, 1.459e−4)
Pituitary	45101	24	6.745	1.496e−4	3.566e−4	(1.463e−4, 1.528e−4)
U2OS	32321	24	4.697	1.453e−4	3.524e−4	(1.415e−4, 1.492e−4)
Genes	307	238	2.775	0.009	0.002	(0.00887, 0.00921)
Simulados	10	2500	83.324	8.332	0.279	(8.159, 8.505)

Tabla 2.5: Rendimiento del algoritmo en C sobre las 6 bases de genes

Ganancia rel. respecto...	Original	Depuración	Result. teóricos
48RMA	11.699	5.727	5.603
NIH3T3	13.620	6.501	6.291
Pituitary	12.772	6.320	6.263
U2OS	13.441	6.582	6.482
Genes	86.442	21.892	13.847
Simulados	215.754	6.688	4.678

Tabla 2.6: Ganancia relativa de la versión en C con respecto a la función original, la depurada, y la mejorada con resultados teóricos

Adicionalmente, la evolución de los tiempos de ejecución pueden comprobarse en la figura 2.14. Nótese que aunque la base de datos "genes" es la que menos tarda, no quiere decir que el ajuste sea más rápido cuando $T = 238$ que con $T = 24$. El tiempo que se muestra en esta figura es el necesario para ajustar todos los genes de la base de datos, pero la base "genes" contiene 307 genes mientras que las demás tienen más de 30000.

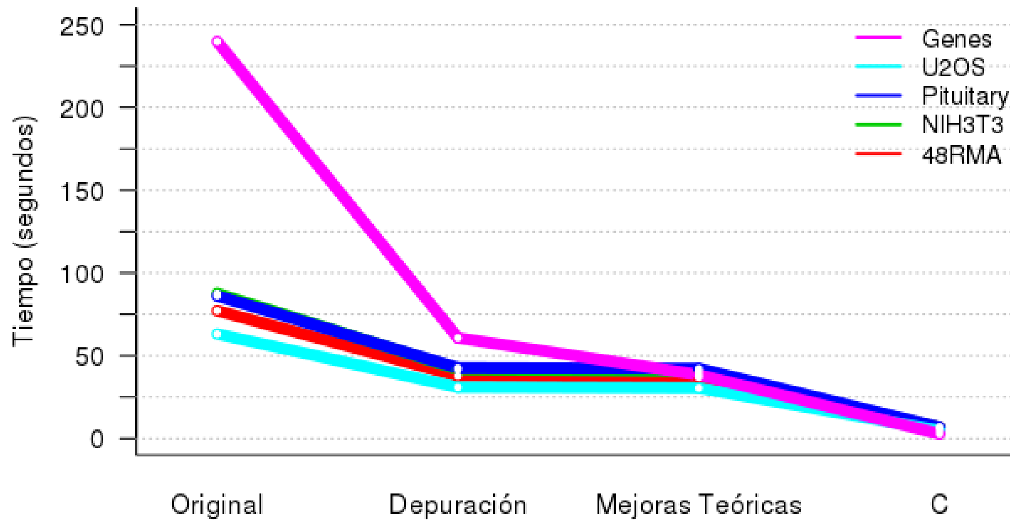


Figura 2.14: Evolución en los tiempos de ejecución hasta la implementación en C (excluyendo la base de datos simulada).

2.5. Paralelización

La paralelización será la última mejora introducida en este trabajo. La computación paralela es una forma de cómputo en la que numerosas instrucciones se ejecutan a la vez, reduciendo un problema grande en varios trozos independientes más pequeños que se resuelven de forma concurrente [30]. Dentro de la computación paralela, existen diferentes niveles:

- **Paralelismo a nivel de bit**: se aumenta el tamaño de la palabra en un computador, con el objetivo de reducir el número de instrucciones que se deben ejecutar en operaciones entre variables cuyos tamaños son mayores que la longitud de la palabra por defecto [31].

- **Paralelismo a nivel de instrucción**: las distintas órdenes que se ejecutan en un programa dado lo hacen en paralelo en procesadores diferentes para acelerar la ejecución, siempre que las instrucciones sean independientes [32].

- **Paralelismo de datos**: el conjunto de datos de entrada a un programa se divide, de manera que a cada procesador le corresponde un conjunto de datos, pero en todos ellos se efectúa la misma operación [33].

- **Paralelismo de tareas**: el paradigma de la programación concurrente más complicado de todos, donde se asignan distintas tareas a cada uno de los procesadores de un sistema de cómputo. Cada procesador tiene su propia secuencia de operaciones [34].

En nuestro caso, adoptaremos el enfoque de paralelismo de datos.

Hay que tener en cuenta que no siempre es posible realizar la paralelización: aquello que se ejecuta simultáneamente debe ser independiente. Considérese el siguiente ejemplo de paralelismo a nivel de instrucción.

1. $e = a + b$
2. $f = c + d$
3. $g = e * f$

La tercera operación depende de las dos anteriores. Sin embargo, las dos primeras pueden ejecutarse simultáneamente sin ningún problema. Podrían realizarse a la vez en un ciclo, y en el siguiente ciclo se ejecutaría la tercera operación.

En R, la paralelización se lleva a cabo utilizando los paquetes *doParallel* y *forEach*. Es un poco más compleja que en otros sistemas, y quizá menos eficiente. No obstante, estos dos paquetes proporcionan un entorno completo para la mayoría de operaciones, aunque no están orientados al paradigma de paralelización de tareas, sino más bien al paralelismo de datos.

Para la ejecución paralela, es necesario registrar el número de procesadores lógicos que se van a utilizar. A continuación, hay que introducir el código a paralelizar dentro de un bucle *foreach*, y posteriormente los resultados son combinados. No es posible utilizar, por ejemplo, el enfoque de *Java*, donde en la ejecución de un programa se crea un hilo que realiza una tarea específica mientras el programa principal continúa. Unos buenos tutoriales introductorios a la paralelización de R pueden encontrarse en [35] y [36] (Steve Weston).

Existen dos formas de llevar a cabo el paralelismo de datos a nuestra función de cálculo del estimador Up-Down-Up, cada una de las cuales se explora en la subsección correspondiente.

2.5.1. Paralelización de una ejecución

La paralelización que aquí se describe se realizará a nivel de cada ejecución, es decir, se paraleliza el cálculo del estimador Up-Down-Up. Anteriormente hemos comprobado que los tiempos de cálculo eran muy reducidos para ejemplos con un período pequeño (ni siquiera un microsegundo si $T = 24$, unos nueve microsegundos si $T = 238$), pero que este tiempo empieza a aumentar a medida que T lo hace (ocho segundos si $T = 2500$). La paralelización únicamente es rentable si la tarea a la que hace referencia es pesada. De lo contrario, si se trata de una tarea ligera, la paralelización no aportará gran cosa, o incluso puede empeorar los tiempos de ejecución debido al coste de inicialización y combinación de los resultados. Por este motivo, se espera que la rentabilidad se alcance a partir de un T grande. Más adelante se examinarán más detalles acerca de la rentabilidad.

La búqueda del mejor estimador pasa por la exploración de todos los mínimos y máximos locales hasta encontrar aquel que tiene un menor MSE. En concreto, se empieza por los mínimos locales y, para cada uno de ellos se exploran todos los posibles máximos locales. Una forma de paralelizar la búsqueda es dividir el conjunto de mínimos locales

y hacer que cada procesador se encargue de buscar el mejor estimador restringiéndose a ese conjunto.

Entrada: $\boldsymbol{Y}$
Salida: $\boldsymbol{Y}^*, mse$

Calcular $loc_M(\boldsymbol{Y}), loc_m(\boldsymbol{Y})$
$nCores \leftarrow min(detectCores() - 1, length(loc_m(\boldsymbol{Y})))$
Dividir $loc_m(\boldsymbol{Y})$ en $nCores$ grupos de igual tamaño: $loc_{m,1}(\boldsymbol{Y}), ..., loc_{m,nCores}(\boldsymbol{Y})$
Programar la paralelización con $nCores$
for each $i \in \{1, 2, ..., nCores\}$ **do Parallel**
    $\hat{\boldsymbol{Y}}_i^* \leftarrow busquedaMejor(\boldsymbol{Y}, loc_{m,i}(\boldsymbol{Y}), loc_M(\boldsymbol{Y}))$ (figura 2.12)
end do Parallel
$k \leftarrow \arg\min_{i=1,...,nCores} MSE(\boldsymbol{Y}, \hat{\boldsymbol{Y}}_i^*)$
$\boldsymbol{Y}^* \leftarrow \hat{\boldsymbol{Y}}_k^*$
$mse \leftarrow MSE(\boldsymbol{Y}, \hat{\boldsymbol{Y}}_k^*)$

Figura 2.15: Pseudocódigo de la función paralelizada

La figura 2.15 contiene el pseudocódigo de la versión paralelizada de la función. Más adelante se presenta otra variante de paralelización. Para distinguir ambas, haremos referencia a esta como la **versión paralela**.

Lo primero que hay que calcular es el conjunto de mínimos y máximos locales. El número de cores a utilizar será el mínimo entre el número de procesadores disponibles (siempre dejando uno libre para no saturar el ordenador) y el número de candidatos a L^*. El conjunto de mínimos locales se divide en tantos grupos de igual tamaño como procesadores a utilizar. La ejecución paralela comienza con la búsqueda del mejor candidato, de manera que cada uno de los procesadores trabaja con un subconjunto de los mínimos locales. La búsqueda del mejor se hace según el pseudocódigo de la figura 2.12. Cuando todos los procesadores han terminado su tarea, hay que elegir el mejor ajuste de entre todos los que devuelven los distintos procesadores.

El rendimiento de la paralelización depende directamente de las especificaciones del ordenador. En ordenadores más lentos y con menos procesadores se espera que el rendimiento ganado sea menor que en otros ordenadores más potentes y con más núcleos. Sin embargo, si el número de procesadores disponible es muy elevado, también es posible que al utilizar todos el rendimiento sea peor, puesto que los distintos procesadores deben sincronizarse entre sí en el cálculo.

La ganancia producida al utilizar esta versión de la función es impredecible sin un estudio previo, por lo que se ha creado un procedimiento en R que puede ayudar a determinar a partir de qué período es rentable su utilización. Esta función comienza examinando los tiempos de ejecución de la versión secuencial y paralela con $T = 24$, y se va incrementando el valor de T de 24 en 24 hasta que el tiempo de ejecución de la versión paralela sea menor que el de la versión secuencial. Se espera que, a partir de este valor de T, compense utilizar la versión paralela de la función. En el caso del ordenador con el que se están realizando todos los cálculos, este procedimiento ha determinado que la versión paralela es rentable a partir de $T_u = 1904$.

Una modificación natural que puede hacerse a la implementación del cálculo del estimador Up-Down-Up incluye una optimización y elección automática de la mejor versión. En primer lugar, se ejecuta el procedimiento para determinar el período T rentable en el ordenador en cuestión. A continuación, en cada llamada posterior se examina el valor del período del vector de datos $\boldsymbol{Y}$. Si el tamaño de $\boldsymbol{Y}$ es superior al umbral T_u, se utilizará por defecto la versión paralela, mientras que si es menor se optará por la secuencial.

Para comprobar el funcionamiento de la versión paralela de la función en genes con distintos períodos, vamos a utilizar esta modificación sobre las 6 bases de datos que hemos considerado hasta ahora. Las modificaciones en los tiempos medios de ejecución (en segundos) se encuentran en la tabla 2.7. Los resultados son muy sorprendentes:

- Los tiempos aumentan de forma exagerada para las 5 primeras bases de datos, que tienen un valor de T “pequeño”. Concretamente, para las bases de datos con $T = 24$, los tiempos pasan a ser del orden de microsegundos a casi segundos. En el caso de 48RMA se llega a tardar hasta 2000 veces más en el cálculo, pero para U2OS es 7000 veces más lento.
- Como habíamos comprobado en el estudio de rentabilidad, sólamente es en el caso de los datos simulados con $T = 2500 > T_u$ cuando la función reduce el tiempo medio de ejecución de $8.3s$ a $5.05s$.

Base de datos		Secuencial		Paralela	(ganancia)
48RMA	($T = 24$)	0.000146	→	0.317	(0.0005)
NIH3T3	($T = 24$)	0.000143	→	0.5798	(0.0002)
Pituitary	($T = 24$)	0.00015	→	0.8255	(0.0002)
U2OS	($T = 24$)	0.000145	→	1.0551	(0.0001)
Genes	($T = 238$)	0.00904	→	4.3767	(0.002)
Simulados	($T = 2500$)	8.3324	→	5.0486	(1.65)

Tabla 2.7: Relaciones entre los tiempos medios de ejecución de la versión secuencial en C y la paralela, en segundos

Aunque estos resultados puedan parecer desesperanzadores, existe otra forma de aprovechar los recursos de los ordenadores en la tarea que nos concierne, como veremos a continuación.

2.5.2. Paralelización de múltiples ejecuciones

Una mejor manera de aprovechar los procesadores de la máquina es la paralelización a nivel de base de datos, en lugar de a nivel de ejecución. Supongamos que tenemos un conjunto grande de datos, y se quiere obtener el mejor estimador Up-Down-Up para todos los genes de dicha base de datos. Podríamos dividir el conjunto de datos en varios grupos, y hacer que cada procesador se encargue de calcular los estimadores para los genes del grupo que se le ha asignado. En todos los casos se utilizaría la versión secuencial, puesto que los procesadores ya se han asignado a las tareas en el reparto

de los genes en grupos. A esta versión de la función la llamaremos, a partir de ahora, **versión múltiple**, en contraposición a la versión paralela de la sección anterior. Las principales ventajas de utilizar el enfoque múltiple en lugar del paralelo son:

1. No se necesitarán unos valores de T exagerados para que la función sea rentable. Por contra, se necesitarán bases de datos grandes, que es lo que suele darse en la realidad.

2. La inicialización de los procesadores y de la paralelización sólo se realiza una vez, al contrario que ejecutar muchas veces la versión paralela del apartado anterior, donde hay que inicializar la paralelización una vez por cada gen.

3. Nos aseguramos de que se aprovecha al máximo posible el rendimiento de los distintos procesadores. En la versión paralela de la sección anterior hay una parte secuencial y una parte en paralelo. La ejecución de este procedimiento para todos los genes de una base de datos tendrá, globalmente, una parte no despreciable de cómputo secuencial. Sin embargo, al distribuir los genes en distintos grupos, la paralelización se aprovecha muchísimo más, y el porcentaje de uso de las distintas CPU's aumenta considerablemente. Las figuras 2.16 y 2.17 muestran los diagramas de flujo para la ejecución y pueden ayudar a comprender la idea que aquí se propone. Nótese que la versión paralela de la función debe preparar la paralelización muchas más veces que la versión múltiple, y el porcentaje de tiempo que se trabaja de forma secuencial es mucho mayor.

La desventaja inmediata de esta función múltiple es que no es válida para el ajuste individual de genes, y se necesita ajustar bastantes para que comience a ser rentable.

En la figura 2.18 se encuentra el pseudocódigo de la versión múltiple. En ella se muestra cómo se divide el conjunto de datos en varios grupos, de manera que cada procesador se encarga de uno de estos grupos. Después, el ajuste individual se hace de forma secuencial.

Igual que hemos hecho antes, es necesario un estudio acerca de la rentabilidad de la versión múltiple de la función. Esta vez, éste será un poco más complejo, puesto que no hay que tener en cuenta únicamente el período T de los genes, sino también el número N de ellos que serán ajustados. Se espera que, a medida que aumenten T y N, la rentabilidad sea cada vez mayor.

Para encontrar la rentabilidad, vamos a comenzar examinando un valor de $N = 100$. Se efectuarán cálculos empezando por $T = 24$, y se aumentará T de 24 en 24 hasta que el tiempo de la versión múltiple sea menor que la versión secuencial. Cuando se haya determinado el valor de T para el cual es rentable el cálculo de $N = 100$, se tomará el nuevo $N' = 1.5N$ y se repetirá el procedimiento empezando por $T = 24$. Se continúa multiplicando N por 1.5 hasta que, para algún cierto N, sea rentable la aplicación de la versión múltiple con $T = 24$.

Evidentemente, los resultados vuelven a depender del ordenador donde se hayan realizado estas comprobaciones. En concreto, para el ordenador descrito en la introducción, los resultados pueden verse en la tabla 2.8. Con un tamaño $N = 100$ se necesitan

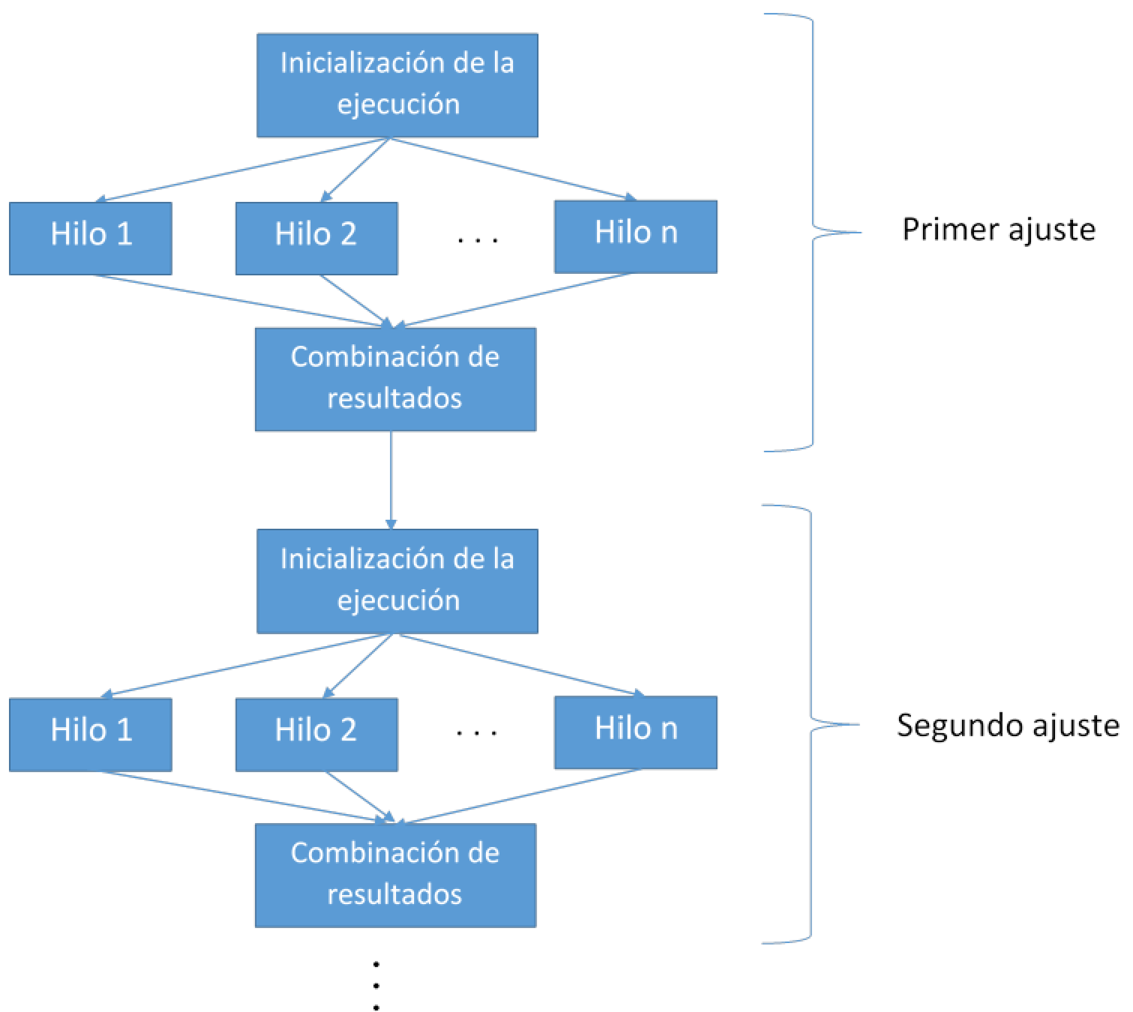


Figura 2.16: Diagrama de flujo de la versión paralela

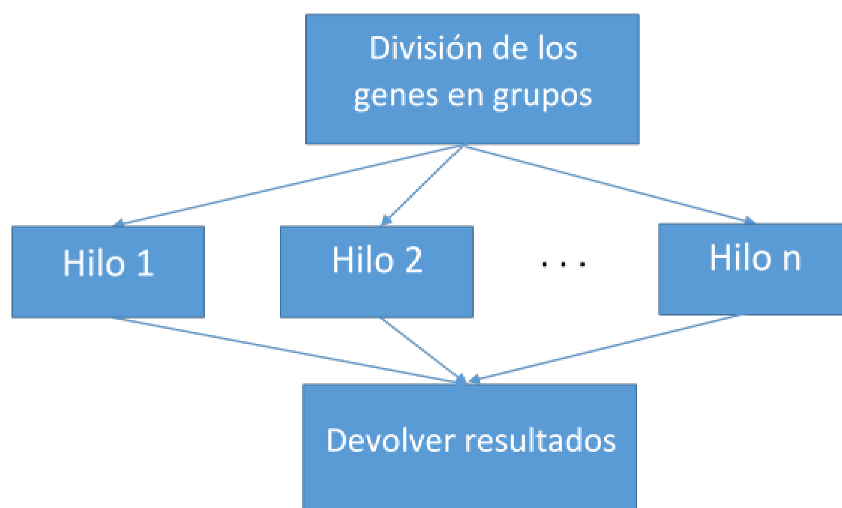


Figura 2.17: Diagrama de flujo de la versión múltiple

Entrada: $\boldsymbol{Y}_1, ..., \boldsymbol{Y}_N$
Salida: $(\boldsymbol{Y}_1^*, mse_1), ..., (\boldsymbol{Y}_N^*, mse_N)$
  $nCores \leftarrow min(detectCores() - 1, N)$
  Dividir $\boldsymbol{Y}_1, ..., \boldsymbol{Y}_N$ en $nCores$ grupos de igual tamaño: $G_1, ..., G_{nCores}$
  Programar la paralelización con $nCores$
  for each $i \in \{1, 2, ..., nCores\}$ **do Parallel**
    for each $\boldsymbol{Y} \in G_i$ **do**
      Calcular $loc_M(\boldsymbol{Y}), loc_m(\boldsymbol{Y})$
      $\hat{\boldsymbol{Y}}^* \leftarrow busquedaMejor(\boldsymbol{Y}, loc_m(\boldsymbol{Y}), loc_M(\boldsymbol{Y}))$ (figura 2.12)
      $mse \leftarrow MSE(\boldsymbol{Y}, \hat{\boldsymbol{Y}}_k{}^*)$
      Guardar el par $(\boldsymbol{Y}^*, mse)$
    end do
  end do Parallel

Figura 2.18: Pseudocódigo de la función múltiple

datos con período igual o superior a $T = 336$ para que se produzca una ganancia. Por otro lado, a partir de $N = 43830$ se espera que siempre sea rentable. La función mostrará un *warning* cuando se crea posible que la versión múltiple no sea rentable. Nótese que no se necesitan unos períodos exageradamente grandes para que la paralelización produzca beneficios, en contraposición con la versión paralela de la función.

N	100	150	225	338	507	760	1140	1710	2565	3848	5772	8658
T	336	228	264	240	216	168	144	120	120	96	96	72

N	12987	19480	29220	43830
T	72	48	48	24

Tabla 2.8: Rentabilidad de la versión múltiple de la función

El procedimiento para calcular la rentabilidad tanto de la versión múltiple como la paralela es automático, y ha sido diseñado de cara a una posible creación de un paquete de R, para optimizar los recursos de cada ordenador.

En la tabla 2.9 se encuentran las evoluciones de los tiempos de cálculo utilizando la versión múltiple. Al contrario que antes, las ganancias ahora son positivas y significativas. Nótese que al utilizar la versión múltiple de la función, no tiene sentido hablar de tiempo medio de cálculo en el ajuste de cada gen individual, puesto que se están ajustando muchos modelos simultáneamente. Por este motivo, los tiempos de ejecución que se muestran son el necesario para el ajuste de cada base de datos completa, y no son comparables directamente entre sí (cada base de datos puede tener distinto número de genes) ni con los resultados de la tabla 2.7 (donde se muerstran tiempos medios). Las conclusiones que podemos extraer son:

- En todos los casos se mejoran los tiempos de ejecución. Podemos llegar a tardar más de 3 veces menos en las bases de datos con $T = 24$, y para los datos simulados el ajuste se hace 8 veces más rápido.
- La mejora en la base de datos simulada es mucho mayor utilizando la versión múltiple que la versión paralela. Para que la versión paralela sea rentable, el período T tiene que verificar $T > T_u$. Si el ajuste es individual, evidentemente la versión múltiple no produce ninguna ganancia, pero aún con un número de datos pequeño ($N = 10$), la ganancia de la versión múltiple es mucho mayor.

2.5.3. Comentarios generales acerca de la paralelización y mejora global

Hasta ahora, las versiones existentes de la función son:

La **secuencial**, donde se realiza el ajuste individual.

La **paralela**, donde existe paralelización a nivel de ejecución individual.

La **multiple**, donde se paraleliza a nivel de base de datos.

Base de datos		Secuencial		Multiple (ganancia)	
48RMA	($T = 24$)	6.594	$\rightarrow$	1.899	(3.472)
NIH3T3	($T = 24$)	6.434	$\rightarrow$	1.859	(3.461)
Pituitary	($T = 24$)	6.745	$\rightarrow$	2.059	(3.276)
U2OS	($T = 24$)	4.697	$\rightarrow$	1.942	(2.419)
Genes	($T = 238$)	2.775	$\rightarrow$	1.601	(1.733)
Simulados	($T = 2500$)	83.324	$\rightarrow$	10.287	(8.1)

Tabla 2.9: Relaciones entre los tiempos totales de ejecución de la versión secuencial en C y la múltiple

Las recomendaciones para la elección de la mejor versión son las siguientes:

- Se debe utilizar la versión secuencial para el ajuste de genes individuales o cuando los procedimientos descritos determinen que no es rentable utilizar ninguna de las otras dos versiones.
- La versión paralela funciona bien para el ajuste individual o de pocos datos, siempre que $T > T_u$.
- La versión múltiple es adecuada para el ajuste de bases de datos completas de genes.

Si no existe una certeza sobre qué versión utilizar, el procedimiento que se debería seguir es el siguiente:

1. Ejecutar los procedimientos de determinación de la rentabilidad paralela y múltiple, que servirán para optimizar todas las llamadas futuras.
2. Si el ajuste es de varios genes, lanzar la versión múltiple. Si el número de genes o el período no es lo suficientemente grande, la función mostrará un *warning* que indica que es mejor el ajuste individual.
3. Si el ajuste es individual o de pocos genes, o el procedimiento anterior ha determinado que la versión múltiple no es rentable, se puede ejecutar la versión individual con optimización automática. En este caso se examina el período de cada gen y, se utilizará la versión secuencial o paralela en función de si el período es menor o mayor que el umbral T_u.

Para concluir las secciones de optimización y mejora, vamos a comparar cuánto se ha ganado partiendo desde el código original hasta la versión múltiple, que ha resultado ser la más eficiente para el ajuste de todas las bases de datos. En la tabla 2.10 están los tiempos, en segundos, necesarios para el ajuste completo de la base de genes.

Los resultados son completamente satisfactorios:

- Para el caso de las tres primeras bases de datos, el ajuste se realiza aproximadamente 40 veces más rápido.

Base de datos		Original		Multiple (ganancia)	
48RMA	($T = 24$)	77.142	$\rightarrow$	1.899	(40.622)
NIH3T3	($T = 24$)	87.628	$\rightarrow$	1.859	(47.137)
Pituitary	($T = 24$)	86.148	$\rightarrow$	2.059	(41.84)
U2OS	($T = 24$)	63.134	$\rightarrow$	1.942	(32.51)
Genes	($T = 238$)	239.876	$\rightarrow$	1.601	(149.829)
Simulados	($T = 2500$)	17977.51	$\rightarrow$	10.287	(1747.595)

Tabla 2.10: Relaciones entre los tiempos totales de ejecución de la versión secuencial en C y la múltiple

- Para la base de datos U2OS, se pasa de $63.134s$ a $1.942s$, tardando más de 30 veces menos.
- Para la base "Genes", que ya tiene un período mayor, la mejora se nota aún más: de 4 minutos a menos de 2 segundos.
- Con los datos simulados ($T = 2500$), el tiempo se reduce de 5 horas a... ¡10 segundos! Es una mejora increíble, y se espera que sea aún mayor si T crece.

Conclusiones

A lo largo de este Trabajo de Fin de Grado hemos explorado algunas de las posibilidades de la Inferencia con Restricciones y ciertas cuestiones sobre su implementación. Es evidente que, con todos los ejemplos propuestos, tanto la regresión isotónica como la estimación up-down-up son dos ejemplos muy potentes que ayudan en una estimación sobre el parámetro de interés en situaciones prácticas donde este tipo de restricciones se verifican.

La aplicación más directa que hemos examinado aquí ha sido la estimación de expresiones de genes, íntimamente relacionada con el ciclo circadiano, y que sirve de pie a la clasificación de genes en rítmicos y no rítmicos. Este paso no ha sido discutido en este trabajo, pero su interés es mayúsculo; entre sus aplicaciones más destacadas está conocer si un determinado tratamiento contra el cáncer tiene un mejor o peor impacto en función del momento del día en que se administra [19].

En el primer capítulo se ha elaborado una extensa discusión sobre las distintas formas de cálculo de la regresión isotónica. Por un lado, un análisis teórico revelaba que las dos formas más conocidas (el PAVA y el algoritmo basado en el máximo minorante convexo) prefieren situaciones totalmente contradictorias: PAVA tarda menos cuando hay pocas violaciones en la restricciones de orden, mientras que el máximo minorante convexo se beneficia de la situación contraria.

También se ha diseñado un experimento con el que poder realizar un análisis empírico de los tiempos de ejecución de varias funciones de R, dedicadas al cálculo de la regresión isotónica, examinando múltiples situaciones diferentes atendiendo al ruido en los datos, forma del parámetro, existencia de pesos, etc. Entre las principales conclusiones de este análisis se encuentran que las funciones más rápidas están escritas en C y basadas en PAVA.

Lo aprendido en el primer capítulo ha sido de gran ayuda en los procedimientos del segundo. En éste se ha presentado la metodología de estimación de señales up-down-up, y cómo depende del cálculo de la regresión isotónica tradicional. Tras un examen de los fundamentos necesarios, hemos pasado a realizar ciertas modificaciones en los algoritmos originales con el objetivo de hacer más eficiente el cálculo. Conocer qué función para el cálculo de la regresión isotónica era la más rápida ha sido muy relevante, en tanto que se requiere un número muy elevado de veces.

A través de resultados teóricos, se ha conseguido reducir el espacio de búsqueda, de forma que, aunque la reducción en los tiempos no fuera tan marcada, permite ahorrarnos ciertas operaciones. Estas modificaciones han sido, precisamente, las más complejas,

pues han conllevado un cambio sustancial en la manera de organizar el código del procedimiento. Motivado por la rapidez de C, tal y como se comprobó en el capítulo 1, la reescritura en este lenguaje del procedimiento agiliza en gran medida el cálculo. Por último, hemos comprobado que la paralelización es totalmente dependiente del ordenador, para lo cual se ha diseñado un procedimiento automático simple para escoger la mejor opción en cada momento.

El trabajo aquí desarrollado permite por tanto diseñar procedimientos que van a ser más eficientes en el cálculo de estimadores bajo restricciones. Lo aprendido no solo puede aplicarse al caso del estimador Up-Down-Up, sino que también permite establecer unas líneas generales que pueden ser útiles a la hora de diseñar algoritmos para otras situaciones.

Como trabajo futuro a desarrollar a continuación de éste, cabe señalar la posibilidad de publicar los resultados aquí obtenidos sobre la mejora de los algoritmos en revistas como *Journal of Statistical Software* o *The R Journal*. Así mismo, destaca la posibilidad de integrar estos procedimientos con el resto de ellos, también asociados a la estructura Up-Down-Up, que está desarrollando el Grupo de Investigación. Esto podría derivar en la creación de un paquete de R que permita, de forma eficiente, el uso de todos estos procedimientos por parte de usuarios externos en la web.

Bibliografía

[1] T. Robertson, F. T. Wright and R. L. Dykstra. (1998). *Order Restricted Statistical Inference (Wiley series in probability and mathematical statistics).* Chichester, England. John Wiley & Sons.

[2] A. B. Németh and S. Z. Németh. 2 de junio de 2015. *Isotonic regression and isotonic projection.* Babes Bolyai University and University of Birmingham.

[3] Jim Pitman and Nathan Ross. *The Greatest Convex Minorant of Brownian motion, meander and bridge.* 25 de octubre de 2018. Recuperado de arXiv. Enlace web: https://arxiv.org/pdf/1011.3073.pdf.

[4] Isotonic Regression, (s.f.), recuperado de http://stat.wikia.com/wiki/Isotonic_Regression.

[5] R Core Team (2015). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

[6] Rolf Turner (2015). Iso: Functions to Perform Isotonic Regression. R package version 0.0-17. https://CRAN.R-project.org/package=Iso

[7] Volkmar Henschel and Ulrich Mansmann (2013). intcox: Iterated Convex Minorant Algorithm for interval censored event data. R package version 0.9.3. https://CRAN.R-project.org/package=intcox

[8] Bernd Klaus and Korbinian Strimmer. (2015). fdrtool: Estimation of (Local) False Discovery Rates and Higher Criticism. R package version 1.2.15. https://CRAN.R-project.org/package=fdrtool

[9] Jan de Leeuw, Kurt Hornik, Patrick Mair (2009). Isotone Optimization in R: Pool-Adjacent-Violators Algorithm (PAVA) and Active Set Methods. Journal of Statistical Software, 32(5), 1-24. URL http://www.jstatsoft.org/v32/i05/.

[10] Dobson, Annette J. (2002). An introduction to generalized linear models. Boca Raton : Chapman & Hall/CRC

[11] Y. Larriba, C. Rueda, M.A. Fernández and S.D. Peddada,"Order restricted inference for oscillatory systems for detecting rythmic signals", *Nucleic Acid Research*, vol.44, no.22, 2016, doi:10.1093/nar/gkw771.

[12] Yolanda Larriba, Cristina Rueda, Miguel A. Fernández and Shaymal D. Peddada, "Order Restricted Inference in Chronobiology" (PREPRINT) .

[13] J.Z. Li, B.G. Bunney, F. Meng, M.H. Hagenauer, D.M. Walsh, M.P. Vawter, S.J. Evans, P.V. Choudary, P. Cartagena, J.D. Barchas, A.F. Schatzberg, E.G. Jones, R.M. Myers, S.J. Watson Jr., H. Akil and W.E. Bunney, "Circadian patterns of gene expression in the human brain and disruption in major depressive disorder," *Proceedings of the National Academy of Sciences of the United States of America*, vol.110, no.24, 2013, pp. 9950-9955.

[14] R. Chauhan, K.F. Chen, B.A. Kent and D.C. Crowther, "Central and peripheral circadian clocks and their role in Alzheimer's disease," *DMM Disease Models and Mechanisms*, vol.10, no.10, 2017, pp. 1187- 1199.

[15] F. Halberg, "Chronobiology," *Annual Review of Physiology*, vol.31, no.1, 1969, pp. 675-726.

[16] R. Refinetti, G. Cornelissen and F. Halberg, "Procedures for numerical analysis of circadian rhythms," *Biological Rhythm Research*, vol.38, no.4, 2007, pp. 275-325.

[17] G. Cornelissen, "Cosinor-based rhythmometry," *Theoretical Biology and Medical Modelling*, vol.11, no.1, 2014, pp. 16, doi:10.1186/1742- 4682-11-16.

[18] G. Cornelissen and K. Otsuka, "Chronobiology of Aging: A Mini- Review," *Gerontology*, vol.63, no.2, 2017, pp. 118-128.

[19] E. Hauss, "Chronobiology in Oncology," International Journal of *Radiation Oncology Biology Physics*, vol.73, no.1, 2009, pp. 3-5.

[20] Zhang,R., Lahens,N.F., Ballance,H.I., Hughes,M.E. and Hogenesch,J.B. (2014) "A circadian gene expression atlas in mammals: Implications for biology and medicine", *PNAS*, 111, 16219–16224.

[21] M.E. Hughes, L. DiTacchio, K.R. Hayes, C. Vollmers, S. Pulivarthy, J.E. Baggs, S. Panda and J.B. Hogenesch, "Harmonics of circadian gene transcription in mammals," *PLoS Genetics*, vol.5, no.4, 2009, doi:10.1371/journal.pgen.1000442.

[22] R. Yang and Z. Su, "Analyzing circadian expression data by harmonic regression based on autoregressive spectral estimation," *Bioinformatics*, vol.26, no.12, 2010, pp. i168-i174.

[23] S. Panda, M.P. Antoch, B.H. Miller, A.I. Su, A.B. Schook, M. Straume, P.G. Schultz, S.A. Kay, J.S. Takahashi and J.B. Hogenesch, "Coordinated transcription of key pathways in the mouse by the circadian clock," *Cell*, vol.109, no.3, 2002, pp. 307- 320.

[24] Ismael, s.f., "Características Lenguaje C", Recuperado de http://www.programandoenc.16mb.com/index.php/lenguaje-cc.

[25] Memoria de acceso aleatorio. (s.f.). En Wikipedia. Recuperado el 22 de agosto de 2018 de https://es.wikipedia.org/wiki/Memoria_de_acceso_aleatorio

[26] Vangie Beal. Access Time. (s.f.). Recuperado de https://www.webopedia.com/TERM/A/access_time.html

[27] Joaquin Fdez-Valdivia. (s.f). Introducción al lenguaje C. Universidad de Granada. Recuperado de http://decsai.ugr.es/ jfv/ed1/c/cdrom/cap1/f_cap12.htm

[28] R (lenguaje de programación). (s.f). En Wikipedia. Recuperado el 29 de agosto de 2018 de https://es.wikipedia.org/wiki/R_(lenguaje_de_programación)

[29] Haghverdi, L., Buttner, M., Wolf, F. A., Buettner, F. & Theis, F. J. Diffusion pseudotime robustly reconstructs lineage branching. Nat. Methods 13, 845–848 (2016).

[30] Computación Paralela. (s.f.). En Wikipedia. Recuperado el 3 de septiembre de 2018 de https://es.wikipedia.org/wiki/Computacion_paralela

[31] Bit-level parallelism. (s.f.). En Wikipedia. Recuperado el 3 de septiembre de 2018 de https://en.wikipedia.org/wiki/Bit-level_parallelism

[32] Instruction level parallelism. (s.f.). En Wikipedia. Recuperado el 3 de septiembre de 2018 de https://simple.wikipedia.org/wiki/Instruction_level_parallelism

[33] Paralelismo de datos. (s.f.). En Wikipedia. Recuperado el 3 de septiembre de 2018 de https://es.wikipedia.org/wiki/Paralelismo_de_datos

[34] Paralelismo de tareas. (s.f.). En Wikipedia. Recuperado el 3 de septiembre de 2018 de https://es.wikipedia.org/wiki/Paralelismo_de_tareas

[35] Microsoft and Steve Weston (2017). foreach: Provides Foreach Looping Construct for R. R package version 1.4.4. https://CRAN.R-project.org/package=foreach

[36] Microsoft Corporation and Steve Weston (2017). doParallel: Foreach Parallel Adaptor for the 'parallel' Package. R package version 1.0.11. https://CRAN.R-project.org/package=doParallel

Índice de tablas

1.1. Tiempo medio del análisis preliminar 25
1.2. Desviaciones típicas del análisis preliminar 26
1.3. Resumen del modelo ajustado para *pavaC* 30
1.4. Resumen del modelo ajustado para *monoreg* 33
1.5. Resumen del modelo ajustado para *intcox.pavaC* 34
1.6. Resumen del modelo ajustado para *isoreg* 36
1.7. Resumen del modelo ajustado para *pava* 39
1.8. Resumen del modelo ajustado para *gpava* 41
1.9. Tabla ANOVA para el modelo de comparación de las funciones de R . . 47
1.10. Resumen del modelo ajustado para comparar las funciones R 48

2.1. Rendimiento del algoritmo original sobre las 6 bases de genes 61
2.2. Rendimiento del algoritmo depurado sobre las 6 bases de genes 72
2.3. Rendimiento del algoritmo con mejoras de resultados teóricos sobre las 6 bases de genes 79
2.4. Ganancia relativa de la versión con mejoras de resultados teóricos con respecto a la función original y a la que contiene sólo la depuración . . . 80
2.5. Rendimiento del algoritmo en C sobre las 6 bases de genes 83
2.6. Ganancia relativa de la versión en C con respecto a la función original, la depurada, y la mejorada con resultados teóricos 83
2.7. Relaciones entre los tiempos medios de ejecución de la versión secuencial en C y la paralela, en segundos 87
2.8. Rentabilidad de la versión múltiple de la función 91
2.9. Relaciones entre los tiempos totales de ejecución de la versión secuencial en C y la múltiple 92
2.10. Relaciones entre los tiempos totales de ejecución de la versión secuencial en C y la múltiple 93

Índice de figuras

1.1. GMC de un camino aleatorio . . . 14
1.2. Código GCM para regresión istónica . . . 15
1.3. Máximo minorante convexo - mejor caso . . . 16
1.4. Máximo minorante convexo - peor caso . . . 17
1.5. Pseudocódigo del algoritmo PAVA, extraído de [4] . . . 19
1.6. Ejemplo de nube de puntos generada con $n = 500$ y $\sigma = 1$. . . . 22
1.7. Pseudocódigo del experimento para regresión isotónica . . . 24
1.8. Tiempo de ejecución de las seis funciones en términos del tamaño del vector de entrada (escala logarítmica) . . . 27
1.9. Análisis preliminar de pavaC . . . 28
1.10. Ajuste del modelo pavaC preliminar . . . 29
1.11. Análisis preliminar de monoreg . . . 31
1.12. Ajuste del modelo monoreg preliminar . . . 32
1.13. Análisis preliminar de intcox.pavaC . . . 33
1.14. Ajuste del modelo intcox.pavaC preliminar . . . 34
1.15. Análisis preliminar de isoreg . . . 35
1.16. Ajuste del modelo isoreg preliminar . . . 37
1.17. Análisis preliminar de pava . . . 38
1.18. Ajuste del modelo pava preliminar . . . 39
1.19. Análisis preliminar de isoreg . . . 40
1.20. Ajuste del modelo pava preliminar . . . 41
1.21. Comparación preliminar entre las funciones lineales . . . 42
1.22. Comparación preliminar con las funciones cuadráticas . . . 43
1.23. Comportamiento de isoreg . . . 45
1.24. Comparación de las tres mejores funciones R cuando el parámetro de forma es creciente . . . 49

1.25. Comparación de las tres mejores funciones R cuando el parámetro de forma es cero . . . 50

2.1. Ejemplo de patrón sinusoidal y no sinusoidal . . . 56
2.2. Ejemplo de una señal más ruido y cálculo del estimador up-down-up . . . 60
2.3. Pseudocódigo de la función original que calcula el mejor estimador up-down-up en el sentido de mínimos cuadrados . . . 60
2.4. Pseudocódigo de la función mejorada para comprobar las condiciones de validez lo antes posible . . . 65
2.5. Búsqueda de mínimos y máximos locales (código original) . . . 66
2.6. Búsqueda de mínimos y máximos locales (código modificado) . . . 68
2.7. Acceso a candidatos a mínimo y máximo . . . 70
2.8. Acceso a candidatos a mínimo y máximo (modificado) . . . 71
2.9. Ganancia de la depuración de la programación . . . 72
2.10. Pseudocódigo del algoritmo PAVA decreciente . . . 75
2.11. Revertir los puntos en el cálculo del PAVA decreciente (ayuda para la demostración de la proposicion 2) . . . 76
2.12. Pseudocódigo de la función mejorada para comprobar las condiciones de validez lo antes posible . . . 78
2.13. Evolución en los tiempos de ejecución hasta las mejoras teóricas (excluyendo la base de datos simulada). . . . 80
2.14. Evolución en los tiempos de ejecución hasta la implementación en C (excluyendo la base de datos simulada). . . . 84
2.15. Pseudocódigo de la función paralelizada . . . 86
2.16. Diagrama de flujo de la versión paralela . . . 89
2.17. Diagrama de flujo de la versión múltiple . . . 90
2.18. Pseudocódigo de la función múltiple . . . 90