Trabajo Fin de Máster

Análisis de datos de los consumos de computación y sus resultados utilizando plataformas de Big Data

Elena Pernas Goy

Máster en Técnicas Estadísticas

Curso 2016-2017

Propuesta de Trabajo Fin de Máster

Título en galego: Análise de datos dos consumos de computación e os seus resultados empregando plataformas de Big Data

Título en español: Análisis de datos de los consumos de computación y sus resultados utilizando plataformas de Big Data

English title: Analysis of the computer consumption data and their results using Big Data platforms

Modalidad: Modalidad B

Autor/a: Elena Pernas Goy, Universidade de Santiago de Compostela

Director/a: Manuel Febrero Bande, Universidade de Santiago de Compostela

Tutor/a: Andrés Gómez Tato, Centro de Supercomutación de Galicia (CESGA)

Breve resumen del trabajo:

El CESGA tiene acumulada mucha información sobre el uso de sus recursos de computación con una historia de más de 10 años. Entre la información existente está la capacidad instalada, las peticiones de ejecución de trabajos, los tiempos de espera, etc. Sin embargo, la explotación de esta información no es suficiente.

Se propone en este trabajo la realización de una exploración de los datos con el fin de definir mejor las métricas de uso y obtener más información útil para la gestión del centro. Para el análisis de datos se utilizará la plataforma de Big Data del CESGA, en principio utilizando Spark como entorno de gestión de los datos y Python como entorno de programación.

Don Manuel Febrero Bande, Catedrático de Universidad de la Universidade de Santiago de Compostela, don Andrés Gómez Tato, Administrador de Aplicaciones y Proyectos de Centro de Supercomutación de Galicia (CESGA), informan que el Trabajo Fin de Máster titulado

Análisis de datos de los consumos de computación y sus resultados utilizando plataformas de Big Data

fue realizado bajo su dirección por doña Elena Pernas Goy para el Máster en Técnicas Estadísticas. Estimando que el trabajo está terminado, dan su conformidad para su presentación y defensa ante un tribunal.

En Santiago de Compostela, a 6 de Julio de 2017.

El director:

Don Manuel Febrero Bande

El tutor:

Don Andrés Gómez Tato

La autora:

Doña Elena Pernas Goy

Agradecimientos

A mis tutores Manuel Febrero y Andrés Gómez por ayudarme, guiarme y aconsejarme durante todos estos meses. A mis compañeros en el Centro de Supercomputación de Galicia por hacer que el ambiente de trabajo fuera excepcional, consiguiendo que las prácticas fueran no sólo una gran experiencia profesional si no también personal. Y por último, a mi familia y amigos por un apoyo constante que me ha permitido llegar a donde estoy ahora.

Índice general

Resumen

Resumen en español

Esta memoria recoge las tareas llevadas a cabo durante las prácticas en el CESGA. Las bases de datos que se han empleado en las mismas pertenecen al sistema Finisterrae y son consideradas Big Data, por lo que se han empleado herramientas adecuadas, en este caso Apache Spark. Spark es una plataforma que permite implementar código en distintos lenguajes de programación de los cuales se ha utilizado Python.

En un primer momento se toma contacto con el programa realizando tareas sencillas de filtrado y cálculo de proporciones y medidas estadísticas para, posteriormente, pasar al grueso de las prácticas que consiste en el análisis del sistema de colas.

El análisis del sistema de colas se divide en dos partes. En una de ellas se trata de predecir los tiempos de espera de las colas mediante series de tiempo. En la otra se trata de realizar una simulación del propio sistema para poder obtener más información de él y del comportamiento de las colas.

Para finalizar, se proponen diferentes caminos por los que se podría continuar trabajando.

English abstract

This report contains the tasks carried out during the internship in CESGA. The data bases that have been used in them belong to the Finisterrae system and they are considered Big Data, so that appropiate tools have been used, in this case Apache Spark. Spark is a platform that allows you to implement code in different programming languages from which Python has been used.

Initially, the program is contacted by performing simple filtering and proportioning tasks and statistical measures and then moving on to the most important thing of the practice of queuing system analysis.

The analysis of the queuing system is divided into two parts. In one of them it is a question of predicting queue waiting times by time series. On the other it is to make a simulation of the own system to be able to obtain more information concerning the behavior of the queues. Finally, some ways are proposed to continue the work.

Prefacio

Este proyecto se ha realizado bajo la modalidad tipo B de prácticas en empresa, en este caso en el CESGA (Centro de Supercomputación de Galicia). Por tanto, lo que aquí se recoge es una memoria del trabajo realizado en dicha empresa, incluyendo dificultades que fueron apareciendo durante su realización. Debido a que las prácticas consistieron en aplicar los conocimientos adquiridos en el Máster de Técnicas Estadísticas a un trabajo real no se desarrollan en esta memoria contenidos teóricos vistos en el máster (al menos no en profundidad), aunque sí se incluye la bibliografía correspondiente donde se pueden consultar.

Durante la realización de las prácticas se ha tenido acceso a dos tipos de bases de datos. Los ficheros acct, que contienen información sobre todo lo que se ha ejecutado en los ordenadores del CESGA, y los datos del sistema de colas.

Los primeros se tratarán en la Sección 2 del Capítulo 1 de forma muy básica como una toma de contacto con Spark y el lenguaje de programación Python.

En cuanto a los datos del sistema de colas han sido empleados en el grueso de las prácticas. Se presentarán en la Sección 3 del Capítulo 1 y se utilizarán en los Capítulos 2 y 3 que constituyen la parte central del trabajo. En dichos capítulos se tratará de realizar predicciones del tiempo de espera de un trabajo una vez que llega al sistema de colas mediante el uso de series temporales. Así mismo en el Capítulo 3 se procederá a realizar una simulación de los sistemas de colas del CESGA.

Ambas bases de datos contienen información sensible por lo que en ocasiones, para explicar el proceso empleado, se utilizarán datos simulados. No obstante, se intentarán mostrar los resultados reales en los casos en los que no existan problemas con la confidencialidad de los mismos.

Los datos incluyen registros diarios, desde 2008 hasta 2016, de todo lo que se ha ejecutado tanto en los ordenadores como en el sistema de colas. Es decir, se dispone de millones de datos por lo que no se podrán emplear las herramientas habituales para su tratamiento ya que el coste computacional sería enorme. De hecho, se trabajará sobre una plataforma de Big Data denominada Apache Spark. Esta plataforma básicamente lo que permite es la ejecución en paralelo tal y como se explica en el Capítulo 1. A grandes rasgos la ejecución en paralelo se diferencia de la ejecución en serie en la posibilidad de fragmentar los datos en varios trozos y aplicar las funciones deseadas sobre cada uno de ellos a la vez para, posteriormente, volver a juntarlos. La principal consecuencia de esto es que el tiempo requerido para la ejecución de un programa es mucho menor, solucionando de esta forma el problema del coste computacional.

El trabajo concluye con un Capítulo resumen en el que se ofrecen unas indicaciones de como se podrían continuar explotando ambas bases de datos. Por último, se incluye en los Apéndices tanto el código completo de todas las funciones que se mencionan en esta memoria (comentados brevemente) y que se han desarrollado con PySpark durante la realización de las prácticas, como el código de la Simulación que se desarrolló en lenguaje R.

Capítulo 1

Apache Spark

Como ya se ha dicho los ficheros acct contienen un registro de todo lo que se ha ejecutado en los ordenadores, incluyendo información del tipo qué comando se ha ejecutado, cuándo, cuánto tiempo le llevó ejecutarse, quién lo ejecutó, cuánta memoria ocupó, etc.

En cuanto a los datos del sistema de colas, han sido empleados en el grueso de las prácticas. Imagínense que hay varias personas ejecutando trabajos en un mismo ordenador, lógicamente si el ordenador sólo puede ejecutar un número determinado de trabajos al mismo tiempo, llegará un momento en el que no pueda aceptar más, de forma que los trabajos sobrantes formarán una cola. Este tipo de información es la que recogen los ficheros de colas, solo que con una pequeña diferencia, en el caso del CESGA, la estructura es mucho más compleja ya que existen varias colas y nodos en los que se pueden ejecutar los trabajos. Así pues, se tiene información sobre a qué cola se envió el trabajo, quién la envió, en qué instante temporal, cuando comenzó a ejecutarse, en qué momento terminó, etc.

Debido al elevado número de entradas que presentan ambas bases de datos son consideradas Big Data. Para el tratamiento de dichos datos, se ha empleado Apache Spark (que permite la ejecución en paralelo). Todo esto se explicará a lo largo del capítulo actual, así como la presentación de los ficheros de datos y un primer contacto con el programa.

Se comenzará por definir formalmente lo que se entiende por Big Data.

Definición 1.1. Big Data, también denominados datos masivos o macrodatos, es un concepto que hace referencia al almacenamiento y tratamiento de grandes cantidades de datos y a los procedimientos para encontrar patrones dentro de ellos.

En [4] se presenta la siguiente clasificación para los distintos tipos de Big Data:

- El número de variables (p) es grande pero más pequeño que el tamaño muestral (n). Es decir, $n >> p$.
- El número de variables es grande y además mayor que el tamaño muestral.
- Los datos son funcionales.

Las bases de datos empleadas en las prácticas a las que se hace referencia en este trabajo son del primer tipo. Además el número de variables a estudiar, de todas las que se disponen, es bastante bajo, de hecho, la única herramienta para el análisis de Big Data que se empleó es la ejecución en paralelo de Apache Spark. De ser mayor el número de variables a considerar deberían emplearse a mayores técnicas estadísticas, como por ejemplo la reducción de la dimensionalidad aplicando el análisis de componentes principales, tal y como se puede ver en el ya citado [4].

La plataforma usada para la resolución de los casos con datos reales ha sido la de Big Data del CESGA [12] que permite el procesado de grandes volúmenes de información en paralelo. En concreto, de los dos entornos disponibles, se ha utilizado Hadoop ya que es el que tiene disponible Apache Spark.

A continuación se tratará de explicar con la mayor brevedad posible en qué consiste Spark, su combinación con el lenguaje de programación Python y unos ejemplos que sirven como toma de contacto con ambos.

1.1. De Hadoop a Spark

Tal y como se ha dicho la principal ventaja de usar Apache Spark es que permite la ejecución en paralelo. No obstante, no es el primer entorno que se ha desarrollado con esa misma característica.

Definición 1.2. Hadoop es un sistema de código abierto que soporta aplicaciones distribuidas bajo una licencia libre. Permite trabajar con miles de nodos y petabytes.

A la hora de hablar de Hadoop existen dos conceptos básicos que hay que definir: MapReduce y HDFS.

Definición 1.3. MapReduce es un entorno de desarrollo que permite trabajar en paralelo con grandes cantidades de datos en sistemas de memoria distribuida (clusters, sistemas Grid y entornos cloud).

Básicamente MapReduce [1] permite procesar y generar grandes conjuntos de datos. Consta de dos tipos básicos de funciones:

- Map: procesa pares clave/valor para crear un conjunto de pares intermedios del mismo tipo.
- Reduce: permite combinar valores asociados con la misma clave.

El problema es que MapReduce no es compatile con algunas de las aplicaciones con las que Spark sí que será compatible.

Definición 1.4. HDFS (Hadoop Distributed File System) es un sistema de ficheros distribuidos diseñado para trabajar con un hardware económico y almacenar grandes cantidades de datos.

Además Spark es de diez a cien veces más rápido que Hadoop dependiendo del tipo de algoritmo que se esté ejecutando tal y como se demuestra en [9]. Y aunque en esta memoria no se exlica pormenorizadamente, en ése mismo artículo se incluye la información de cómo surge Spark.

Definición 1.5. Spark es una plataforma de computación en clúster[1] diseñada para ser rápida y de uso general que puede ejecutar cálculos en memoria.

Debido a esa generalidad Spark extiende el modelo MapReduce de forma que es compatible con más tipos de operaciones (más allá del Map y Reduce), de hecho incluye colas interactivas y procesos en streaming.

Otra de las características de Spark es su compatibilidad con Hadoop. Aunque no sea necesaria la instalación del mismo para su correcto funcionamiento, cabe decir que Spark puede ser ejecutado en los clusters de Hadoop y acceder a los ficheros HDFS.

En cuanto a las mejoras que introduce Spark ya se ha hablado de la velocidad y de que amplía el abanico de funciones que se pueden emplear. Otra de las novedades que incluye con respecto a Hadoop es que introduce una forma de definir los conjuntos de datos denominada RDD. Manteniendo eso sí, las formas habituales de almacenamiento de datos como los `data frame SQL`, `arrays` y `Pandas data frame`.

Definición 1.6. Un `array` es un tipo de dato estructurado que permite almacenar un conjunto de datos homogéneo, es decir, todos ellos del mismo tipo y relacionados.

Definición 1.7. El `data frame` se basa en la idea de que los conjuntos de datos en estadística se pueden organizar de forma rectangular con las filas representando unidades muestrales y las columnas representando variables.

[1] Conjunto de ordenadores unidos entre sí y que se comportan como si fuesen una única computadora.

El `data frame` es semejante al `array` sólo que el primero admite que los datos sean de distinto tipo. Se denominarán `data frame SQL` o `Pandas data frame` en función del módulo de Python con el que son generados y tratados.

Definición 1.8. Un RDD (Resilient Distributed Datasets) en Spark es simplemente una colección de objetos distribuida e inmutable. Cada RDD es separado en múltiples particiones que pueden ser computadas en diferentes nodos del clúster.

A la hora de implementar código en Spark se ha intentado almacenar los datos en formato RDD mientras fuese posible, pasándolos posteriormente a formatos `data frame` en función de las necesidades del usuario (primero a uno de tipo `SQL` y finalmente, y como último recurso, a uno tipo `Pandas`). Esto se ha hecho así puesto que los dos primeros permiten la ejecución en paralelo y el último no, por tanto, se evita su uso hasta que los datos hayan sido reducidos al mínimo posible (mediante filtrados, eliminación de variables, etc.). Se ha optado por esta metodología de trabajo debido a que cuando en un primer momento se intentó trabajar con `Pandas` directamente el tiempo requerido para la ejecución era muy elevado.

Por último cabe decir que Spark está implementado en Scala, aunque es compatible con otros lenguajes de programación como Python, R y Java. En este trabajo se ha empleado el lenguaje de programación Python, utilizando lo que se conoce como PySpark, por requerimiento de la empresa.

1.1.1. PySpark

En este apartado se realiza una breve descripción de los principales módulos y funciones definidas en Python que se han empleado en este trabajo. Para la instalación y configuración de PySpark consultar [11].

Definición 1.9. Un módulo es un archivo que contiene definiciones y declaraciones de Python. El nombre del archivo es el nombre del módulo con el sufijo `.py` agregado.

Para poder emplear un módulo y las funciones que contiene en primer lugar hay que importarlo con la siguiente sentencia:

```
import <nombre_del_modulo_sin_el_.py>
```

o bien

```
import <nombre_del_modulo> as <nombre_que_le_vamos_a_dar>
```

Por ejemplo

```
import pandas as pd
```

Para cargar una función en concreto en lugar de todo el módulo se tiene la sentencia:

```
from <nombre_modulo> import <nombre_funcion>
```

Los módulos más empleados a lo largo de las prácticas son:

- `Matplotlib` [13]: permite realizar gráficos.
- `Numpy` [14]: fundamental para el cálculo científico con Python. Entre otras cosas permite emplear arrays N-dimensionales, implementar funciones, integrar código C/C++ y Fortran y además incluye las herramientas para trabajar con álgebra lineal, la transformada de Fourier y aleatoriedad.
- `Pandas` [15]: facilita herramientas para el análisis de datos y es fácil de usar sobre datos estructurados.

Para definir los conjuntos de datos se emplean las siguientes funciones:

- `sc.paralellize`: genera un RDD.

  ```
  rdd=sc.paralellize(datos)
  ```

- `sql.Context.createDataFrame`: estructura los datos en un `data frame` de tipo `SQL`, donde cada una de las variables coincide con las columnas especificadas.

  ```
  df=sqlContext.createDataFrame([datos],('columna1','columna2',...))
  ```

- `pandas.DataFrame`: genera un `data frame` de tipo `Pandas`.

  ```
  pdf=pandas.DataFrame([datos],columns=['columna1','columna2',...])
  ```

A partir de ahora y para mostrar los ejemplos correspondientes al resto de funciones que aquí se presentan: se denotarán los tipos de estructuras de datos como se acaba de ver.

Ya se ha indicado como generar las distintas estructuras de datos, pero en ocasiones será necesario pasar de unas a otras para lo que se emplean las siguientes funciones:

- `.rdd` para pasar de `data frame SQL` a RDD.

  ```
  rdd=df.rdd
  ```

- `.toDF()` para pasar de RDD a `data frame SQL`.

  ```
  df=rdd.toDF(('columna1','columna2',...))
  ```

- `.toPandas()` para pasar de `data frame SQL` a uno tipo `Pandas`. No se indica en este caso el nombre de las columnas puesto que ya toma el mismo que tenían en el `data frame`.

  ```
  pdf=df.toPandas()
  ```

Ahora sí, las funciones que más se han empleado a lo largo de este trabajo son:

- `.count()`. Función que cuenta los elementos de un conjunto de datos.
- `.filter()`. Filtra datos en función de una condición de igualdad, desigualdad, pertenencia...
- `.groupBy()`. Agrupa los datos en función de las distintas categorías de una variable.
- `.mean()`. Función del cálculo de media.
- `.select()`. Selección de una o más variables dentro de un conjunto de datos.

Evidentemente, como con cualquier lenguaje de programación se pueden definir funciones propias. Para definir una función se comienza con `def` acompañado del nombre de la función y los datos que va a recibir (entre paréntesis) seguido de dos puntos. A continuación, en las siguientes líneas, se introducen las sentencias que se quieren ejecutar sobre esos datos y, para finalizar la función, se escribe una última línea con el comando `return` y la salida de la función.

Por ejemplo, una función que se llame `f`, que reciba dos números `x` e `y`, que obtenga como salida `x, y` y la suma de ambos sería

```
def f(x,y):
    a = x + y
return (x, y, a)
```

De esto se verán ejemplos de mayor complejidad a lo largo de la memoria, puesto que se incluye el código desarrollado. También se pueden realizar bucles de repetición y de condicionales.

```
for i in ... :
   hacer algo
if (condición lógica):
   se hace esto
else:
   hacer esta otra cosa
```

Lo que sí que hay que tener en cuenta es que Python es muy exigente con los indexados y, por tanto, debe de emplearse el tabulador en cada línea dentro de un bucle o función para que no se produzcan errores.

A la hora de crear un programa largo, o bien para consultar posteriormente, se puede necesitar comentar algo dentro del propio código. Para escribir comentarios en Python simplemente hay que poner `#` y escribir a continuación las aclaraciones que se desee.

1.2. Un primer contacto con PySpark

Como ya se han mencionado las características de Spark y el lenguaje de programación que se utilizan sólo queda por explicar como iniciar sesión.

Para ello, en primer lugar, tienes que situarte en el entorno donde se encuentre el programa. Abriendo la terminal y tecleando:

```
ssh -X login.hdp.cesga.es
```

Una vez hecho esto debes introducir tu clave de usuario CESGA y ya se está en condiciones de comenzar. Hay principalmente dos formas de trabajar con Pyspark: o bien se elaboran scripts que deben ser guardados con la terminación `.py` y se ejecutan con la siguiente línea.

```
spark-submit --master yarn --deploy-mode cluster --name <nombreusuario>
 --num-executors 10  --executor-cores 2 --driver-cores 2 --executor-memory 4G
   trabajo.py input output
```

donde los números se pueden cambiar en función de los recursos que se necesiten, o bien se pone en método interactivo

```
pyspark
```

Apareciendo así la pantalla, que se puede ver en la Figura 1.1.

Durante las prácticas se ha utilizado la segunda por ser más cómoda a la hora de corregir el código. Sin embargo, la sesión interactiva no es recomendable para tareas que requieran una mayor cantidad de tiempo.

1.2.1. Datos del accounting

Para la toma de contacto con PySpark se utilizaron los datos acct. Como ya se ha dicho este tipo de archivos contienen información de todo lo que se ha ejecutado en un ordenador llegando a acumular hasta 19 variables sobre cada ejecución. Dichas variables son `flag`, `version`, `tty`, `exitcode`, `uid`, `gid`, `pid`, `ppid`, `btime`, `etime`, `utime`, `stime`, `mem`, `io`, `rw`, `minflt`, `majflt`, `swaps` y `command`. Se definen brevemente y a continuación las que se emplean en este trabajo.

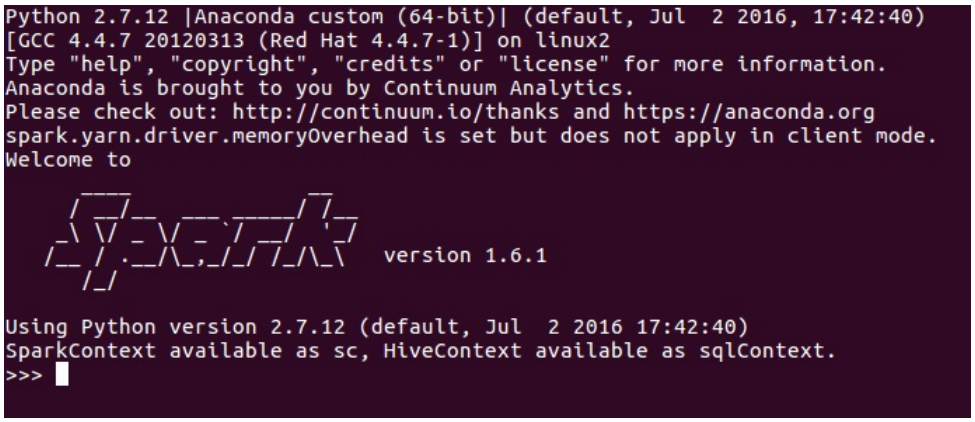


Figura 1.1: Pantalla del modo interactivo PySpark

- `uid`: cada usuario tiene asignado un id. El `uid` hace referencia a dicho número de identificación.
- `gid`: cada usuario pertenece a un grupo (ya sea de investigación o un departamento dentro de una empresa), a este grupo se le asigna un id, el `gid`.
- `etime`: es el elapsed time o tiempo transcurrido.
- `utime`: tiempo de uso de la CPU.
- `stime`: tiempo de CPU (del sistema).
- `command`: contiene el comando que se ha ejecutado.

Este tipo de datos se encuentran almacenados en el sistema de ficheros HDFS, cada línea del fichero original está compuesta por 64 bits, y su contenido está en binario. Por tanto, un primer paso es crear un programa que lea los acct y, en los casos que sea necesario, transforme las variables para que se pueda trabajar con ellas. Por ejemplo, algunas variables temporales están medidas en tiempo GMT Unix[2] y para saber al instante temporal al que se refieren se deben convertir a formato fecha y hora, para lo que se emplea la siguiente función del módulo `time`:

```
time.asctime(time.localtime(values[i]))
```

donde `values[i]` denota la posición de la tupla donde se encuentra el dato temporal que hay que modificar.

Otras variables que deben ser transformadas son `utime, stime` y `mem` que están guardadas en forma de potencia: 3 bits con un exponente de base 8 y 13 bits de mantisa. La transformación a número usual se realiza mediante la siguiente función según se recoge en el manual del CESGA.

```
def expand(dato):
   return (dato & 0x1ffff) << (((dato >> 13) & 0x7) * 3);
```

Por último hay que transformar la variable `etime` puesto que se encuentra almacenada en centésimas de segundo y debe pasarse a segundos para estar en la misma unidad que todas las demás. Para ello simplemente se divide entre 100 mediante la definición de una constante que se denominará `AZH` y que tendrá dicho valor.

Una vez definidas las transformaciones necesarias sólo queda leer y estructurar los datos almacenándolos de alguna de las formas que se ha visto. En este caso, en un RDD. Para dar dicha estructura se cargan los módulos necesarios y se emplea el siguiente código:

[2]Número de segundos que han pasado desde la medianoche del 1 de Enero de 1970

```
AcctRecord = namedtuple('AcctRecord',
'flag version tty exitcode uid gid pid ppid '
'btime etime utime stime mem io rw minflt majflt swaps '
'command')

def read_record(data):
   AZH=100;
   values = struct.unpack("2BH6If8H16s", data)
   print(type(values))
   # Remove null characters from command
   command = values[-1].replace('\x00', '')
   # Replace command value with the new version that has no nulls
   t1 = time.asctime(time.localtime(values[8]))
   t2 = expand(values[10])
   t3 = values[9]/AZH
   t4 = expand(values[11])
   t5 = expand(values[12])
   values =values[:8] + (t1, ) + (t3, ) + (t2, ) + (t4, ) + (t5, ) + values[13:-1]
 + (command, )
return AcctRecord(*values)
```

En primer lugar, se definen los nombres de cada una de las variables de la tupla `AcctRecord` mediante la función `namedtuple`. En cuanto a la función `read_record` aplica las transformaciones sobre los datos de accounting (acct). Se analiza a continuación dicha función pormenorizadamente.

Se define la constante `AZH` y se le da la estructura a los datos línea por línea. Los comandos que se encuentran en el entrecomillado indican si los datos son caracteres, números enteros, etc. La correspondencia se puede ver en [10].

Debido a como están almacenados los datos la última variable (`command`) tiene bits de relleno hasta completar los 64 bits que conforman cada línea. Estos bits de relleno se traducen mediante la línea de código previa en un número de repeticiones de `\x00`. Mediante el comando `replace` se reestablece la variable `command` con los valores que ya tenía pero eliminados los bits de relleno. El resto de transformaciones aplicadas ya se han visto y la penúltima línea del fragmento de código anterior simplemente sobreescribe las variables por sí mismas pero en su versión transformada.

Para emplear la función anterior se utilizarían las siguientes líneas de código, donde la primera carga los datos y la segunda los estructura con la función previa.

```
>>> rdd = sc.binaryRecords('NOMBRE DEL FICHERO', recordLength=64)
>>> records = rdd.map(read_record)
```

Una entrada de dicho `RDD` tendría el siguiente aspecto:

```
[AcctRecord(flag=2, version=3, tty=0, exitcode=0, uid=0, gid=0, pid=32719, ppid=27182,
btime='Tue Dec 31 10:15:22 2013', etime=5591.02, utime=16, stime=27, mem=76144, io=0,
rw=0, minflt=2688, majflt=0, swaps=0, command='sge_shepherd')
```

Una vez que se han leído los datos y se tienen estructurados correctamente en un `RDD` ya se está en condiciones de comenzar a trabajar con ellos.

1.2.2. Tablas y gráficos con los ficheros acct

En primer lugar se quiso realizar un estudio sobre el consumo de los ordenadores del CESGA, entendiéndose consumo como tiempo consumido en la ejecución de los trabajos. Como en los registros se encuentran datos de todo lo que se ha ejecutado, tal y como se ha dicho varias veces, en primer

lugar hay que distinguir en qué casos no interesa conocer el consumo que genera la ejecución de un trabajo.

Por ejemplo, no es relevante el estudio del tiempo que consumen comandos que se utilizan constantemente y por defecto como el `ls`, que en Linux permite ver un listado de los archivos que se encuentran en un directorio. En consecuencia, este tipo de comandos fueron filtrados a partir de una lista proporcionada por la empresa. Para ello se introdujeron todos estos comandos en una lista de Python y se realizó un filtrado

```
>>> lista=['comando1','comando2',...]
>>> datos=rdd.filter(lambda x: x[-1] not in lista)
```

donde `-1` denota la posición en la que se encuentra la variable `command` y `rdd` es el `RDD` donde se habían cargado los datos.

La función `lambda` que se puede ver en la línea de código anterior no se ha explicado con anterioridad por ser más comprensible aplicada. Esta función lo que hace es recorrer fila por fila el `RDD` y aplicarle a cada fila la instrucción que aparece a la derecha de ":". En este caso se encuentra dentro de un filtrado por lo que se almacenan en el `RDD datos` todas las filas cuya última columna no coincide con ningún elemento de la lista proporcionada.

En cuanto al cálculo del consumo de las CPUs se querían sacar esos datos en función del usuario (`uid`), grupo al que pertenece (`gid`), empresa y comando (`command`). Por tanto se eliminarán todas las variables que no se vayan a utilizar. Para ello se emplea la función `select` y se sobreescriben los datos (ya que los originales no se van a necesitar).

```
>>> datos=datos.select(lambda x: (x[4],x[5],x[10]+x[11],x[-1]))
```

donde `x[i]`, $i \in \{4, 5, 10, 11\}$ se refieren a la posición de `uid, gid, utime, stime` dentro del `RDD`, respectivamente, y `x[-1]` es, nuevamente, la posición de la variable `command`.

Como se puede observar al mismo tiempo que se realiza la selección de variables se aprovecha para realizar una operación. Dicha suma es el cálculo del tiempo consumido por el comando ejecutado y se calcula como la adición de las variables `stime` y `utime`.

Por tanto, llegados a este punto, se tiene un `RDD` denominado `datos` que contiene la información que se puede ver en la Tabla 1.1.

Usuario (uid)	Grupo (gid)	Consumo (segundos)	Comando
u1	A	0	c1
u2	B	2	c2
u3	A	6	c3
u2	B	3	c4
u4	C	1	c5
u3	A	0	c1

Tabla 1.1: Encabezado de un data frame de datos (no reales) de una CPU en un período de tiempo de un año.

Si se presta atención se verá que en la Tabla 1.1 no aparece la variable empresa por ninguna parte. Esto se debe a que dicha variable no se encuentra almacenada en los ficheros acct por lo que debe ser añadida. Durante el proceso se tuvo acceso a un archivo `.csv` que contenía a qué empresa pertenecía cada `uid` (y más variables que no se emplean aquí). Una vez cargados estos datos a la sesión interactiva la creación de una nueva variable resulta bastante sencilla puesto que Python tiene una función denominada `join` que se emplea para unir conjuntos de datos a partir de una variable común. Eso sí, entre dos conjuntos de datos del mismo tipo, por eso, en primer lugar, se convierte `datos` a un `data frame SQL`. Después se leen los datos del `.csv` como un `Pandas data frame` (ya que es la forma de leer los `.csv` que se eligió).

Se transforma dicho `data frame` a uno `SQL`. Y, en la última línea de código, se indica qué `data frames` se van a unir, por medio de qué variable (`df.uid` con `empresas.uid`) y se hace una selección de las variables que se van a necesitar evitando almacenar las que no sean de utilidad.

```
df=datos.toDF(('uid','gid','cons','command'))
empresas=pandas.read_csv(r'dump.csv')
empresas=sqlContext.createDataFrame(empresas,('login','uid','institucion',
'centro','departamento'))
tabla=df.join(empresas, df.uid == empresas.uid,'inner').select(df.uid,df.gid,
df.cons,df.command,empresas.institucion)
```

Se obtiene así la Tabla 1.2, donde ya se pueden observar todas las variables involucradas en esta primera tarea.

Usuario (uid)	Grupo (gid)	Consumo (segundos)	Comando	Empresa
u1	A	0	c1	1
u2	B	2	c2	2
u3	A	6	c3	1
u2	B	3	c4	2
u4	C	1	c5	1
u3	A	0	c1	1

Tabla 1.2: Data frame de datos (no reales) de una CPU en un período de tiempo de un año.

Se comienza por tanto a definir las funciones que calculen la proporción de tiempo usado por `uid, gid,` etc. El código completo de dichas funciones se puede ver en el Apéndice A, se incluyen aquí las líneas centrales del mismo para su explicación, en este caso se seleccionaron los correspondientes al consumo por empresa siendo el resto análogas.

Por tanto, se definen las siguientes funciones[3]:

```
def tablas(d):
   total=d.groupBy().sum('Consumo').collect()
```

[3]La función se ha visto reducida puesto que sólo se muestra el caso de cálculo por empresa. La función original calcula las proporciones con respecto a las otras variables al mismo tiempo y se incluye en el Apéndice A con el nombre `CPU_rdd.py`

```
   emp=d.select(d.Empresa,d.Consumo).groupBy('Empresa').sum('Consumo')
.lit(total[0][0]).alias('total'))
   c=emp.toPandas()
   porp=c.suma_cons/c.total
   c['prop']=prop*100
   return c

def grafica(d,C,b):
   plt.pie(d.prop,labels=d.empresa)
   plt.title('uso CPU %s en el %d' %(C,b))
   return
```

Antes de explicar qué es lo que hace cada una de las funciones anteriores cabe decir que faltaría una tercera función que recibe como argumentos el año y de qué CPU se quieren obtener los datos. Lo que hace es leer los datos día a día para la CPU y año indicados, con la función `read_record`, filtrarlos y prepararlos como se acaba de explicar. Posteriormente, llama a las dos funciones que se acaban de indicar y devuelve tres variables: los datos filtrados, los datos incluyendo la empresa (por si se quieren emplear para algo), las tablas calculadas. Además imprime por pantalla las cuatro gráficas.

En cuanto a lo que hacen las funciones se comenzará por la primera:

- `tablas(d)`. Recibe los datos ya preparados y realiza los siguientes cálculos (línea por línea):

  Suma todos los elementos de la columna `Consumo` (esto es posible porque no se le indica en base a qué tiene que agrupar los elementos). Guarda el resultado en una lista (`collect()`) en la que el dato se encuentra en la posición `[0][0]`.

  Selecciona las columnas del `data frame` con las que se desea trabajar, en este caso `Empresa` y `Consumo`. Agrupa los datos por empresa y realiza la suma del consumo.

  Cambia los nombres de las columnas, selecciona las dos columnas a tratar y aumenta una columna que repita el valor del consumo total para posteriormente operar con ella.

  Transforma los datos a un `Pandas data frame`.

  Divide la columna que contiene la suma de los consumos por empresa entre el consumo total para calcular la proporción (`prop`).

  Crea una nueva columna en el `Pandas data frame` que contiene los datos de la proporción en tanto por cien.

- `grafica(d)`. Recibe el resultado de la función anterior y realiza el gráfico de tartas correspondiente.

Aplicando estas funciones a los datos de la Tabla 1.2 se obtienen los resultados recogidos en la Tabla 1.3 y la Figura 1.2.

En cuanto a un ejemplo con datos reales se obtendrían los siguientes resultados para una CPU concreta para el 2016 (nótese que se han modificado los números de identificación para respetar al confidencialidad de los datos).

```
Uso de la CPU *** por usuario:
uid  suma_cons      total          prop
0     a    88369951  504555594  1.751441e+01
1     b           2  504555594  3.963884e-07
2     c          22  504555594  4.360273e-06
```

Empresa	Consumo
1	0.5833
2	0.4167

Tabla 1.3: Ejemplo de tabla del uso de una CPU en función de la empresa a partir de los datos simulados.

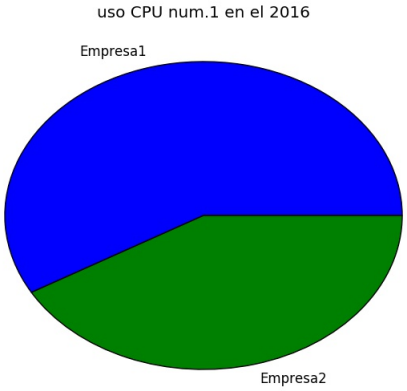


Figura 1.2: Ejemplo de gráfico del uso de una CPU en función de la empresa a partir de los datos simulados.

```
3     d   272245283  504555594  5.395744e+01
4     e          42  504555594  8.324157e-06
5     f   143905937  504555594  2.852132e+01
6     g       23909  504555594  4.738625e-03
7     h       10448  504555594  2.070733e-03
Uso de la CPU *** por grupo:
gid  suma_cons      total       prop
0     a  272245283  504555594  53.957440
1     b      10448  504555594   0.002071
2     c  232275912  504555594  46.035742
3     d         42  504555594   0.000008
4     e      23909  504555594   0.004739
Uso de la CPU *** por comando:
command  suma_cons      total          prop
0             qhost         49  504555594  9.711517e-06
1                su         13  504555594  2.576525e-06
2             mkdir          0  504555594  0.000000e+00
3        version.sh          0  504555594  0.000000e+00
4               cat          4  504555594  7.927769e-07
5      qrsh_starter          2  504555594  3.963884e-07
6                mv          0  504555594  0.000000e+00
7              nrpe        975  504555594  1.932394e-04
```

8	sge_shepherd	18	504555594	3.567496e-06
9	chown	0	504555594	0.000000e+00
10	stty	0	504555594	0.000000e+00
11	pdflush	12449	504555594	2.467320e-03
12	alaska.exe	29892169	504555594	5.924455e+00
13	suse.de-backup-	0	504555594	0.000000e+00
14	ptree_only_pids	3492	504555594	6.920942e-04
15	syslog	1	504555594	1.981942e-07
16	seward.exe	4617280	504555594	9.151182e-01
17	suse.de-check-b	0	504555594	0.000000e+00
18	mktemp	0	504555594	0.000000e+00
19	wrf.exe	0	504555594	0.000000e+00
20	suse.de-cron-lo	0	504555594	0.000000e+00
21	killproc	0	504555594	0.000000e+00
22	bash	90	504555594	1.783748e-05
23	rassi.exe	482695	504555594	9.566736e-02
24	sshd	30	504555594	5.945826e-06
25	mpid	27	504555594	5.351244e-06
26	date	0	504555594	0.000000e+00
27	cron	0	504555594	0.000000e+00
28	sed	2	504555594	3.963884e-07
29	bzip2	11273	504555594	2.234243e-03
..	...	...	...	...
44	acct	0	504555594	0.000000e+00
45	vasp.5.2.12-p	33704472	504555594	6.680031e+00
46	muestra_proceso	7	504555594	1.387360e-06
47	suse-do_mandb	0	504555594	0.000000e+00
48	suse-clean_catm	0	504555594	0.000000e+00
49	job_starter.sh	25	504555594	4.954855e-06
50	molcas.exe	552	504555594	1.094032e-04
51	cesga-alfeijoo.	13	504555594	2.576525e-06
52	parnell.exe	2110	504555594	4.181898e-04
53	qstat	23	504555594	4.558467e-06
54	tar	12	504555594	2.378331e-06
55	python	24	504555594	4.756661e-06
56	grep	5	504555594	9.909711e-07
57	check_procesos	39	504555594	7.729574e-06
58	check_ldap	15	504555594	2.972913e-06
59	vasp.4.6.28-p	143905920	504555594	2.852132e+01
60	mclr.exe	34287769	504555594	6.795637e+00
61	pidof	21	504555594	4.162079e-06
62	rm	1150	504555594	2.279233e-04
63	transform.exe	0	504555594	0.000000e+00
64	uname	0	504555594	0.000000e+00
65	perl	5972	504555594	1.183616e-03
66	ln	0	504555594	0.000000e+00
67	suse.de-clean-t	0	504555594	0.000000e+00
68	logrotate	1	504555594	1.981942e-07
69	ls	30	504555594	5.945826e-06
70	udevd	0	504555594	0.000000e+00
71	show_lost_mpi_p	2	504555594	3.963884e-07
72	check_disk	213	504555594	4.221537e-05

```
73              sh         58  504555594  1.149526e-05

[74 rows x 4 columns]
Uso de la CPU *** por institucion:
institucion  suma_cons      total       prop
0         A            42  504555594   0.000008
1         B     34357  504555594   0.006809
2         C   272245283  504555594  53.957440
3         D 232275912  504555594  46.035742
```

1.3. Breve análisis de los ficheros de colas

Al igual que se acaba de hacer con los ficheros acct se realiza también un primer contacto con PySpark a través de los ficheros de colas. Aprovechando para comenzar a manejar estos ficheros que, como ya se ha dicho en el Prefacio, se emplearán en los próximos capítulos y conformarán el grueso del trabajo.

Cuando se trata de ejecutar un trabajo en un ordenador compuesto de varios nodos, cada nodo con varios procesadores, puede ser que los recursos necesarios para su ejecución no estén disponibles en ese instante. Si esto se produce el trabajo entra en un sistema de colas y permanece en esta cola hasta que los recursos que necesita queden libres. Si mientras esto sucede llega otro trabajo que sí tiene disponibles los recursos que necesita, éste no accede al sistema de colas si no que comienza a ejecutarse. En caso contrario pasa a la cola detrás del trabajo enviado previamente. Las colas del CESGA se distribuyen como sigue:

Existen cinco colas principales y varias secundarias. Se denominan principales aquellas que han estado activas durante toda la vida del Finisterrae (sistema que se analiza en este trabajo) y se consideran secundarias las colas que se crearon para una tarea concreta y sólo estuvieron en uso durante la realización de la misma. En esta memoria sólo se tendrán en cuenta las del primer tipo. Cada una de las colas tienen unas características específicas en función del número de nodos que las componen:

- interactive (cola i). Consta sólo de 2 nodos. Esta cola como su propio nombre indica está reservada para trabajos interactivos.
- large_queue_PdE (cola lp). Es la cola de puertos y consta de 9 nodos.
- small_queue (cola s). La cola pequeña, que consta de 16 nodos.
- medium_queue (cola m). Consta de 40 nodos.
- large_queue (cola l). Es la cola más grande, es decir, con mayor capacidad y está formada por 75 nodos.

Los datos de colas proporcionados para estas prácticas incluyen datos de cada trabajo que se ha ejecutado, del tiempo en el que se ha enviado el trabajo, en el que empezó a ejecutarse y el tiempo en que terminó de ejecutarse. Además del número de referencia del trabajo, el usuario, la memoria que ocupa, en qué cola se ha ejecutado, etc.

1.3.1. Lectura de los datos de colas

Los datos de colas se leen en dos etapas[4]. La primera carga el fichero y la segunda estructura los datos originales, donde cada variable viene separada por dos puntos de la siguiente, en un `data frame`

[4]El código de lectura se encuentra en el Apéndice A bajo el nombre de `lectura_colas.py`.

SQL. Es en la segunda parte donde se tiene que especificar el esquema de los datos, así se tendrán caracteres de palabras, números enteros, etc. Una vez que se han leído todos los datos se genera un RDD. Una entrada de dicho RDD tendría el siguiente aspecto:

```
[Row(qname=u'interactive', hostname=u'cn002.null', groupname=u'cesga',
 owner=u'agomez', jobname=u'QLOGIN', jobnumber=2785459, account=u'sge', priority=0,
 qsub_time=u'2011-01-05 16:31:42', start_time=u'2011-01-05 16:31:45', end_time=
 u'2011-01-05 16:33:05', failed=0, exit_status=127, ru_wallclock=80, ru_utime=0.804,
 ru_stime=0.204, ru_ixrss=0, ru_ismrss=0, ru_idrss=0, ru_isrss=0, ru_minflt=13849,
 ru_majflt=18, ru_nswap=0, ru_oublock=0, ru_msgsnd=0, ru_msgrcv=0, ru_nsignals=0,
 ru_nvcsw=1303, ru_nivcsw=58, project=u'NONE', department=u'defaultdepartment',
 granted_pe=u'NONE', slots=1, na1=u'0', cpu=1.008, mem=0.028028, na2=u'0.000000',
 command_line_arguments=u'-U software -u agomez -l h_fsize=20G,h_rt=36300,
 h_stack=256M, num_proc=1,s_rt=36000,s_vmem=4G -I y', na3=u'0.000000', na4=u'NONE',
 start=1294241505,  send=1294241502)]
```

Se podría pensar en cargar los datos directamente en un RDD pero la forma más cómoda (esto es una elección completamente subjetiva) de definir la estructura de los datos es la que se ha empleado: generar el data frame y después convertirlo a un RDD.

Como se ha dicho hay muchas variables en este tipo de ficheros, sin embargo, se centrará la atención en tres de ellas:

- El nombre de la cola en la que se ha ejecutado el trabajo.
- El segundo en el que se envió el trabajo (en tiempo GMT Unix).
- El segundo en el que comenzó a ejecutarse (en tiempo GMT Unix).

Se puede definir así una nueva variable:

Definición 1.10. Se define la variable tiempo de espera como la diferencia, en segundos, entre el instante en que empieza la ejecución de un trabajo y el instante en el que se envía.

Una vez calculados los tiempos de espera y generada la nueva variable se procede al análisis de los datos de colas.

1.3.2. Problemas con los tiempos de espera calculados

El primer problema que surgió al analizar los tiempos de espera calculados fue la existencia de tiempos de espera negativos. Evidentemente esto implicaba que había algún error, ya fuera en los datos o en los cálculos de dichos tiempos.

Para comprobar que el código funcionaba correctamente se tomaron las diez primeras entradas de un fichero y se hicieron los cálculos, con PySpark y a mano, sacando los mismos datos pero, en vez de tomados del RDD, de los ficheros de datos originales por si el problema estaba en la lectura de los mismos (aunque el programa de lectura ya se había comprobado con anterioridad que funcionaba correctamente). Como esto demostró que el cálculo de los tiempos era el correcto, se procedió al estudio de los datos que provocaban esas singularidades. Para ello se filtraron los mismos.

```
>>> tiempos_negativos = datos.filter(lambda x: x[-1] < 0)
```

donde la función lambda permite ir recorriendo fila a fila el fichero RDD y x[-1] indica la posición en que se encuentran los tiempos de espera calculados, visualizándolo como un data frame sería la posición correspondiente a la última columna.

Una vez generado este nuevo RDD, tiempos_negativos, se analiza pormenorizadamente. Mostrando algunas filas del mismo se pudo observar que este efecto se producía principalmente en dos

situaciones: o bien el trabajo formaba parte de toda la programación de instalación del propio sistema, o bien existían desfases de unos segundos provocados por reajustes de la máquina (provocando tiempos de espera de -1).

Vistos los motivos por los que se producían esos tiempos negativos se optó por filtrar los datos antes de trabajar con ellos. Así el conjunto de datos empleado en el resto de esta memoria se corresponde a los trabajos cuyos tiempos de espera son positivos[5].

```
>>> datos = datos.filter(lambda x: x[-1] > 0)
```

1.3.3. Medidas estadísticas de los tiempos de espera

En un primer momento se pidieron distintas medidas estadísticas de los tiempos de espera en colas con el objetivo de ver cuál era mejor para proporcionarle al usuario a modo de información[6] sobre el tiempo de espera.

Para ello se calculan la media, moda y mediana globales como sigue:

Nota 1.11. Se considera que los datos ya han sido leídos y filtrados por los tiempos de espera (eliminados los no positivos) aunque en las funciones definidas para realizar esta práctica la lectura y el filtrado se incluyen dentro de cada una sólo se analiza a continuación la línea de código correspondiente al propio cálculo de las medidas estadísticas encontrándose el código completo en el Apéndice A (con el nombre de `media_moda.py`).

- Media. La función `mean()` de Python calcula la media directamente sin necesidad de programar uno mismo la función. Simplemente le hay que indicar sobre qué variable se quiere calcular dicha media.

  ```
  >>> media = df.map(lambda x: x[-1]).mean()
  ```

  donde `df` es el nombre que recibe el `RDD` donde se tienen los datos filtrados y `-1` denota la posición de la variable tiempos de espera sobre la cual se calculará la media.

- Moda. No existe una función definida en Python que calcule la moda por lo que se programó una aplicando directamente la definición de moda.

  Nota 1.12. A pesar de que el tiempo es una variable continua se tratará como discreta para el cálculo de la moda puesto que no tiene más resolución que el segundo.

  Definición 1.13. La moda es el valor que tiene mayor frecuencia absoluta.

  Es decir, se elaboró un código que cuenta las veces que se repite un dato y que de entre todos ellos devuelva el valor que más veces se repitió. Para ello se emplearon funciones de Python definidas para los `data frame` de tipo `SQL` por lo que, en primer lugar, se transforma el `RDD` a ese tipo de `data frame`.

  ```
  >>> a = df.toDF(('queue','wait'))
  >>> moda=a.groupBy(a.wait).count().orderBy('count').collect()[-1]
  ```

  donde `df` es el `RDD` que contiene los datos ya filtrados por tiempos de espera y en el que se han eliminado ya todas las variables salvo el nombre de la cola y el propio tiempo de espera que se ha calculado (la eliminación de variables se realiza mediante la función `map` en combinación con

[5] Se ignoraron también los de tiempo de espera iguales a cero a petición de la empresa.

[6] Actualmente el tiempo de espera que se le da al usuario se calcula en función de los tiempos solicitados por los demás usuarios cuando envían un trabajo al sistema de colas. Esto provoca que sea superior al tiempo real de espera puesto que la tendencia es pedir más tiempo del que en verdad se necesita.

`lambda` como ya se ha visto). La segunda línea de código agrupa los datos por tiempos de espera (`.groupBy(a.wait)`) y cuenta cuántos datos caen en cada grupo (`.count()`), luego ordena dichos datos en función de la variable `count` (`.orderBy('count')`) y transforma lo anterior en una lista (`.collect()`) de la que selecciona el último valor (`[-1]`) que es el que contiene el valor que se ha repetido más veces.

- Mediana. Tampoco existe una función en Python que calcule la mediana por lo que se programará nuevamente a partir de la definición:

  Definición 1.14. La mediana es el valor central de un grupo de números ordenados crecientemente. Si la cantidad de términos es par, la mediana es el promedio de los dos números centrales. Si es impar es el valor central.

  Para establecer esta función se emplearán condiciones lógicas. El código que se incluye aquí fue una primera forma de cálculo de la mediana, posteriormente se decidió estudiar los cuantiles (ya se explicará por qué) y se integró el cálculo de la misma en dicha función, `cuantiles.py`, que se incluye en el Apéndice A.

```
>>> ordenados = sorted(a.collect())
>>> n = len(ordenados)
>>> middle = n/2
>>> if n%2==0:
mediana = (ordenados[(n/2)][0]+ordenados[(n/2)+1][0])/2
else:
mediana = ordenados[(n/2)][o]*,
```

  donde `a` es el `data frame` creado a partir del `RDD` donde se cargan y se filtran los datos, en el cual se han eliminado todas las variables menos el tiempo de espera. Desglosando el código se tiene lo siguiente:

  Se crea una lista denominada `ordenados` que contiene los datos de `a` ordenados (valga la redundancia) por orden creciente (`sorted`). `n` es el número de datos que hay en dicha variable (`len(ordenados)`) y `middle` es simplemente la mitad de `n`. Dentro del bucle se tiene lo siguiente, si `n` es par (`n%2==0`, es decir, el resto de esa división es cero) entonces la mediana es el punto medio de los dos valores centrales de la lista `ordenados`, en caso contrario se tiene que la mediana es el valor central (tal y como se decía en la definición).

Los resultados obtenidos empleando el código anterior se recogen en la Tabla 1.4. Sin embargo, como ya se ha dicho, cada cola tiene una característica diferente, es decir, existen colas interactivas donde el tiempo de espera es mínimo y otras donde se lanzan trabajos más pesados, computacionalmente hablando, y en consecuencia los tiempos de espera serán mayores. Por tanto se calculan también la media, la moda y la mediana para cada una de las colas principales, simplemente añadiendo un filtrado a las funciones anteriores, y se incluyen los resultados también en la Tabla 1.4.

A la vista de los resultados obtenidos se puede apreciar lo que se mencionaba antes de que cada cola tiene características diferentes. La media de las colas 4 y 5 distan mucho de la global, esto se debe al tipo de trabajos que se envían a esas colas (por ejemplo, la cola 5 es interactiva). También se aprecia una gran diferencia entre la media y las otras dos medidas, mediana y moda, que son más robustas.

Esto puede deberse a que, debido al número de usuarios que tienen acceso a los sistemas de colas, se ha puesto una limitación de trabajos que cada uno de ellos puede ejecutar de forma simultánea. Es decir, si x es el número de trabajos que se pueden ejecutar simultáneamente y el usuario lanza y trabajos con $y > x$ entonces se ejecutan x trabajos e $y - x$ pasan automáticamente al sistema de colas (aunque haya recursos disponibles).

	Media	Moda	Mediana
Globales	224350.58	1	12
Cola s	465963.57	1	128628
Cola l	218958.10	1	6
Cola m	112868.66	1	1
Cola lp	471.94	4	211
Cola i	31.56	1	5

Tabla 1.4: Datos de colas del año 2011: media, moda y mediana, globales y por cola. Unidad temporal empleada el segundo. Datos redondeados a dos decimales.

Otra posible explicación es que aunque la mayor parte de los trabajos entren en ejecución rápidamente, algún trabajo "muy pesado", es decir, que tarde mucho tiempo en acabar de ejecutarse, bloquee una de las colas provocando que el tiempo de espera aumente. Pero ¿qué proporción de trabajos es la que sufre esos tiempos de espera tan elevados?.

Para entender mejor esto se hizo un estudio de los cuantiles, del 90 al 99 por ciento, para ver dónde se produce ese punto de inflexión en los tiempos de espera.

Para ello se modificó la función que se acaba de emplear para cacular la mediana obteniendo el siguiente código:

```
def cuantil(q, lineas):
   df=datos_colas(lineas).rdd
   df=df.filter(lambda x: x[-2]-x[-1] > 0) # elimina tiempos de espera no positivos
   df=df.filter(lambda x: x[-1]!=0)
   a=df.map(lambda x: (x[-2]-x[-1])).collect()
   n=len(a)
   ordenados=sorted(a)
   b=100/float(q)
   if n%b == 0:
      c=ordenados[int(n/b)]+ordenados[int(n/b)+1])/2
   else:
      c=ordenados[int(n/b)]*1
   return c
```

donde `datos_colas` es la función de lectura y estructuración de los datos de la que ya se ha hablado. En este caso se muestra como se realiza el filtrado antes de calcular los tiempos de espera, `-2` es la posición del tiempo de inicio de la ejecución y `-1` del tiempo de envío del trabajo a la cola. Se realiza un segundo filtrado de los trabajos que nunca llegaron a enviarse (es decir, los que forman parte del proceso de instalación). Se genera la variable tiempos de espera eliminando todas las demás y se crea una lista con ella. El resto es igual que en la función de la mediana sólo que en lugar de tomarse el dato `n/2` toma el correspondiente a `n/100*q`. Indicando como `q` el cuantil que se quiere obtener se calcularon los cuantiles del 90 al 99 por ciento obteniendo los datos recogidos en la Tabla 1.5.

Si se analizan dichos resultados se puede ver que sólo un 9 % de los trabajos tardan cuatro días o

Cuantiles	91 %	92 %	93 %	94 %	95 %	96 %	97 %	98 %	99 %
Globales	332212	337776	347065	523206	1191520	1961652	2658312	3802800	5387832
Cola s	1828958	1960229	2080113	2169221	2177085	2254907	2593994	3123659	5251118
Cola l	198447	246563	278908	299930	320865	604178	3887335	5020209	5856535
Cola m	136799	152437	172074	197045	238423	288482	357211	2821401	3724986
Cola lp	945	1521	1599	1601	1608	1616	1622	1631	1635
Cola i	32	34	36	38	41	45	52	60	70

Tabla 1.5: Datos de colas del año 2011: cuantiles. Unidad temporal empleada el segundo.

más en comenzar a ejecutarse. De los cuales el 5 % están mínimo 14 días esperando en el sistema de colas. Además un 1 % tarda más de 2 meses en empezar su ejecución.

Esto quiere decir que, en general, un 91 % de las veces los trabajos entran en ejecución rápidamente. Sin embargo, hay un porcentaje mínimo de trabajos que están tanto tiempo en espera que provocan que la media de los tiempos de espera se dispare como se veía antes.

Se tienen por tanto analizados los tiempos de espera en función de algunas medidas estadísticas globales y por cola. Sin embargo, podría pensarse si hay épocas en que los tiempos de espera sean mayores o menores, es decir, no habrá el mismo número de trabajos enviados a colas en vacaciones que el resto del año. Con el fin de estudiar este efecto se realizó el cálculo de la media, mediana y moda por fecha en lugar de por cola.

Realizando dichos cálculos para los datos pertenecientes al año 2011 se obtienen los resultados recogidos en la Tabla 1.6.

El dato que más destaca a simple vista es el correspondiente al mes de Octubre, donde se puede apreciar que casi se duplica el valor medio de los tiempos de espera. ¿A qué es debido esto? Yendo a los datos originales se localizaron los trabajos que provocan ese incremento de los tiempos de espera. Se encontró que había un usuario que envió 100000 trabajos con tiempo de espera de 50 días, lo cual explicaría perfectamente los resultados obtenidos.

En cuanto al resto de los tiempos de espera son todos como cabía esperar. Destacan los datos de Agosto y Septiembre pero estos quedan completamente explicados por el hecho de que los usuarios suelen aprovechar el período vacacional para lanzar trabajos en el sistema de colas y así, a la vuelta, disponer ya de los resultados.

Finalmente se hizo lo mismo por empresa para poder conocer la relación de cuánto esperan de media cada una, lo cual es interesante para los propios usuarios. Esos datos no se incluyen aquí debido a la confidencialidad de los mismos. Cabe decir que a pesar de que los tiempos de espera globales sean completamente diferentes, todas las empresas aquí registradas han tenido una mediana de 5 segundos, lo que volvería a conducir al razonamiento de los cuantiles.Es decir, hay pocos trabajos que bloquean la cola provocando tiempos de espera muy grandes.

A la vista de las tablas que se acaban de presentar se concluye que aunque más del 50 % de los trabajos tienen un tiempo de espera pequeño, los que tienen un tiempo de espera grandes tardan tanto en comenzar a ejecutarse que aumentan en gran medida la media. Por tanto, desde mi punto de vista, la medida estadística de los tiempos de espera que habría que darle al usuario para que se haga una idea de cuánto va a tener que esperar para que su trabajo se ejecute es la mediana acompañada de los cuantiles, para no crear falsas expectativas de que el trabajo se va a ejecutar rápidamente.

Mes	Media	Moda	Mediana
Enero	23504.01	1	6.53
Febrero	13313.90	1	3.70
Marzo	87101.61	1	24.19
Abril	169182.44	1	47.00
Mayo	171822.96	1	47.73
Junio	64311.08	1	17.86
Julio	118721.37	1	32.98
Agosto	222102.24	1	61.70
Septiembre	821564.24	1	228.21
Octubre	1349783.38	1	374.94
Noviembre	9531.12	1	2.65
Diciembre	15204.19	1	4.22

Tabla 1.6: Datos de colas del año 2011: media, mediana y moda por fecha. Unidad temporal empleada el segundo. Datos redondeados a dos decimales.

Capítulo 2

Predicción de los tiempos de espera

La dificultad para dar buenas aproximaciones de cuál va a ser el tiempo de espera de un trabajo enviado al sistema de colas conduce a intentar predecir dichos tiempos. Como ya se ha dicho, en la actualidad, cuando un usuario quiere saber cuánto tardará en entrar en ejecución el trabajo que va a enviar al sistema de colas, la información que se le ofrece es el tiempo estimado en función del solicitado por el resto de usuarios. El problema de esto es que, generalmente, dichos usuarios tienden a pedir más tiempo del que realmente necesitan y, en consecuencia, esta aproximación del tiempo de espera es siempre superior al real. Por tanto, se tratará de realizar la predicción de los tiempos de espera empleando series de tiempo. Los contenidos teóricos de este Capítulo se pueden encontrar en [6].

Definición 2.1. Una serie de tiempo es una colección de observaciones de una variable X tomadas secuencialmente a lo largo del tiempo. Dichas observaciones serán tomadas a intervalos regulares.

Para poder predecir a partir de una serie temporal se supone que la serie de tiempo ha sido generada por un modelo estocástico, y hay que averiguar cuál es dicho modelo. Las etapas para identificar el modelo de la serie de tiempo y poder realizar predicciones son:

- Identificación del modelo estocástico.
- Estimación del modelo.
- Diagnosis del modelo.
- Predicción.

Se pretende construir un proceso estocástico sencillo que pueda generar una serie de tiempo con características similares a la que se quiere analizar. Se realiza ahora el análisis de una de las colas con la correspondiente explicación de cada una de las etapas.

2.1. Creación y lectura de la serie temporal

Empleando varios bucles y el programa de lectura de colas del que ya se ha hablado se consiguieron hacer tres lecturas por mes para cada cola, de forma que para cada diez días se calcula el tiempo medio de espera, la variable X en este caso. Estos datos se guardaron posteriormente en un fichero `.txt` obteniendo así una serie temporal de los tiempos medios de espera.

Debido a que las colas principales de este sistema son cinco se repitió esto para los datos pertenecientes a cada una de ellas. Se obtienen así cinco ficheros `.txt` con datos sobre la evolución de los tiempos de espera.

Para este capítulo se emplearon el software estadístico R[8] y el paquete `TSA`[5] para el análisis de series temporales. El procedimiento seguido es el siguiente:

La serie total se divide en dos partes, la primera formada por todos los datos menos los 12 últimos y otra serie formada por los datos que se eliminaron de la primera. La serie mayor se emplea para aproximar el modelo y realizar las predicciones y la menor se emplea para comprobar el error cometido al predecir. Se denominarán a lo largo del análisis `serie` y `serie2` respectivamente.

```
serie.total<-ts(scan("datos.txt"))
serie<-window(serie.total, end=end(serie.total)-c(0,12))
serie2<-window(serie.total, start=end(serie)+c(0,1))
```

Para evitar la falta de homogeneidad de las primeras mediciones se optó por eliminarlas directamente, ya que sólo darían lugar a atípicos y a una complicación del ajuste del modelo. Esta heterogeneidad es debida a que los datos se empezaron a almacenar desde el instante en el que se arrancó la máquina, incluyendo así el periodo de instalación (en el que no funcionaba con normalidad). El código anterior quedaría entonces como sigue:

```
serie<-window(serie.total, start=start(serie.total)+c(0,30), end=end(serie.total)
-c(0,12))
```

A veces es necesario realizar una transformación previa sobre la serie antes de identificar su modelo. Para saber si es necesario aplicar una transformación, ya sea logarítmica o Box-Cox, se agruparon los datos de la serie original en períodos de 24 y se representaron las respectivas medias frente a las desviaciones típicas correspondientes con las siguientes líneas de código.

```
serie2<-ts(serie.total[-c(0:30)],frequency=24)
mu=aggregate(serie2,nfrequency=1,mean)
sig=aggregate(serie2,nfrequency=1,sd)
plot(mu,sigma,xlab="media",ylab="desviación típica")
```

Se obtiene de esta forma la Figura 2.1 donde se aprecia que se puede aplicar una transformación logarítmica, se realiza dicha transformación (`serie_log<-log(serie)`) y se trabaja con la nueva serie a partir de ahora.

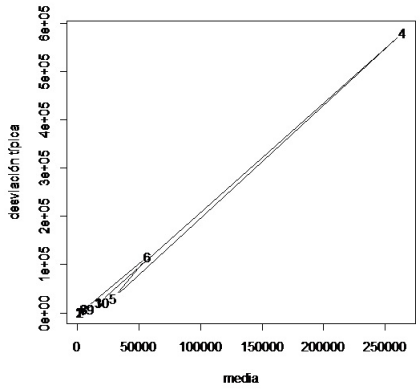


Figura 2.1: Media frente a desviación típica de los datos agregados de 24 en 24.

2.2. Identificación del modelo estocástico

El análisis de una serie temporal se apoya en recursos gráficos y numéricos. Se comienza por el análisis gráfico que incluye los gráficos secuencial, de autocorrelaciones simples y parciales para, posteriormente, proponer un modelo para la serie temporal. Como ya se explicará más adelante, se supone que se ha producido una intervención, como existen diferentes tipos de intervención, para considerar varias posibilidades, se han probado distintos modelos de tipo Box Jenkins.

Definición 2.2. Se define el gráfico secuencial como la representación gráfica de cada observación frente al instante en que se observa y luego unir cada punto con el del instante siguiente.

En dicho gráfico se puede apreciar la posible presencia de tendencia (comportamiento a largo plazo), estacionalidad (comportamiento periódico de la serie) y heterocedasticidad (variabilidad de la serie no es constante).

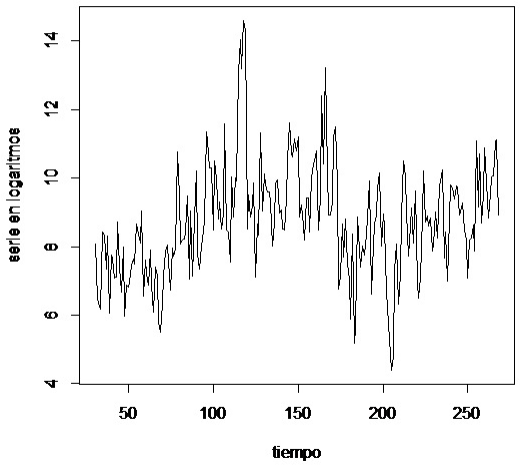


Figura 2.2: Gráfico secuencial de la serie en logaritmos.

Se tiene el gráfico secuencial de la serie en la Figura 2.2 en el que se puede apreciar que probablemente se haya producido una intervención en la serie, es decir, la serie sufre una modificación de su comportamiento debido a un suceso externo. Existen dos tipos de efectos provocados por una intervención: los permanentes y los transitorios. Los primeros se refieren a cambios de nivel producidos a partir de un instante conocido y los segundos afectan sólo a algunos valores.

La posible intervención se produciría en torno al valor 170, más concretamente en el 173. Yendo a los datos originales se comprobó que ese cambio sucede a finales del 2011 coincidiendo con la instalación de nuevas máquinas en el SVG que entraron en competencia con el Finisterrae (sistema que estamos analizando) provocando que usuarios migraran al otro sistema.

Se establece el modelo ARIMA de la serie preintervención, es decir, se corta la serie en ese instante y se trabaja sobre ella. Se procede a continuación a establecer los valores de los parámetros del modelo para lo que se obtiene en primer lugar el gráfico secuencial de la serie preintervención, Figura 2.3 (superior). Gráfico que suele ir acompañado del de autocorrelaciones simples.

Definición 2.3. La función de autocorrelaciones simples (fas) es una medida del grado de dependencia lineal existente entre los valores que toma la serie en dos instantes temporales distintos.

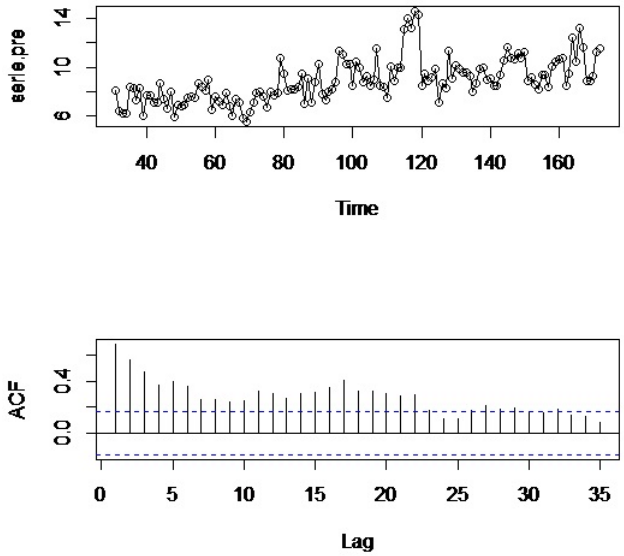


Figura 2.3: Gráfico secuencial (superior) y gráfico de autocorrelaciones simples (inferior) del logaritmo de la serie preintervención.

En este último gráfico, Figura 2.3 (inferior) se puede apreciar que las autocorrelaciones simples van disminuyendo hasta quedar entre las bandas de confianza. Esto lleva a pensar que la serie no presenta tendencia, ya que en caso de haberla, el decaimiento de las autocorrelaciones simples sería muy lento. Para corroborar lo que se deduce gráficamente se emplearon dos contrastes de la librería `urca`[7], el test de raíces unitarias de Dickey-Fuller y el KPSS (`ur.df` y `ur.kpss` respectivamente), mediante los que se concluyó que la serie no tenía raíces unitarias y, por tanto, no era necesaria la diferenciación regular para eliminar la presencia de tendencia.

Se tiene por tanto que la serie no presenta tendencia, su variabilidad es homogénea y además no presenta estacionariedad, por tanto, se está en condiciones de iniciar la búsqueda de los parámetros p y q del modelo ARMA(p, q) correspondiente, que no es otra cosa que un ARIMA(p, d, q) con $d = 0$. Para ello se analizarán los gráficos de autocorrelaciones simples y parciales (Figura 2.4).

Definición 2.4. La función de autocorrelaciones parciales es una medida del grado de dependencia lineal existente entre los valores que toma la serie temporal en dos instantes de tiempo distintos una vez que se les ha sustraído el efecto lineal que sobre ellas ejercen las variables medidas en los periodos comprendidos entre dichos instantes.

Para proponer el modelo hay que fijarse en el número de retardos (líneas verticales) que sobresalen de las líneas discontinuas de la Figura 2.4. Se intentará modelizar la serie con un modelo ARMA como ya se ha dicho. Sin embargo, se intentará fijar a cero p o q puesto que esto genera modelos más sencillos. En caso de que no sea posible se intentará modelizar un ARMA con $p \neq 0 \neq q$. Si el modelo propuesto fija $p = 0$ el número de retardos que sobresalen de las bandas horizontales en la gráfica superior dará el valor de q, en el caso análogo para $q = 0$ el valor de p dependerá de los retardos que sobresalgan en el gráfico inferior.

Se observa que la estructura es más simple en el gráfico inferior por lo que se fija a cero el parámetro p, y se propone en consecuencia un modelo AR(q). Se probaron varios valores para dicho parámetro:

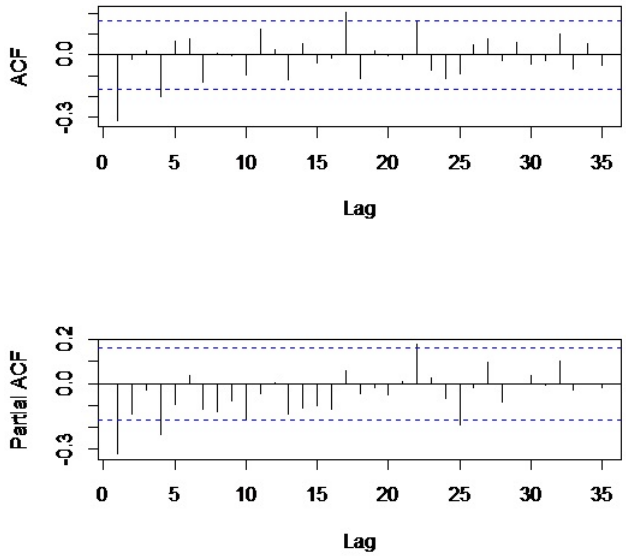


Figura 2.4: Gráfico de autocorrelaciones simples (superior) y parciales (inferior) del logaritmo de la serie preintervención.

1, 2 y 5. Las dos primeras opciones no pasaban la diagnosis por lo que no se incluyen en esta memoria, se procede a ajustar un AR(5) para la serie preintervención.

```
> ajuste3.pre <- arimax(serie.pre, order=c(5,0,0))
> ajuste3.pre

Call:
arimax(x = serie.pre, order = c(5, 0, 0))

Coefficients:
         ar1     ar2     ar3      ar4     ar5  intercept
      0.5522  0.1562  0.0497  -0.1365  0.1863     8.9310
s.e.  0.0825  0.0945  0.0952   0.0943  0.0837     0.5067

sigma^2 estimated as 1.482:  log likelihood = -229.9,  aic = 471.79
```

Sin embargo, puede que alguno de los coeficientes no sea significativo, es decir, significativamente distinto de cero. Serán significativos al 95 % de confianza los coeficientes que cumplan $\frac{|\text{valor del coeficiente}|}{1{,}96*\text{s.e.}} \geq 1$. Es decir,

```
> abs(ajuste3.pre$coef)/(1.96*sqrt(diag(ajuste3.pre$var.coef)))
      ar1       ar2       ar3       ar4       ar5 intercept
3.4155569 0.8435960 0.2664635 0.7385839 1.1363617 8.9931521
```

Se eliminan los coeficientes no significativos fijándolos a cero. El procedimiento para hacer esto es eliminar primero el que tiene un cociente menor, ajustar nuevamente el modelo sin ese parámetro y calcular de nuevo ese cociente. Así hasta que todos los coeficientes son significativos, obteniendo el siguiente modelo:

```
> ajuste3.pre <- arimax(serie.pre, order=c(5,0,0),fixed=c(NA,0,0,0,NA,NA))
> ajuste3.pre

Call:
arimax(x = serie.pre, order = c(5, 0, 0), fixed = c(NA, 0, 0, 0, NA, NA))

Coefficients:
         ar1  ar2  ar3  ar4     ar5  intercept
      0.6267    0    0    0  0.1714     8.9135
s.e.  0.0640    0    0    0  0.0653     0.4902

sigma^2 estimated as 1.534:  log likelihood = -232.31,  aic = 470.63
> abs(ajuste3.pre$coef)[-c(2:4)]/(1.96*sqrt(diag(ajuste3.pre$var.coef)))
     ar1       ar5 intercept
4.995290  1.337901  9.277542
```

Al tener que todos los coeficientes son significativos se procede a identificar la función de transferencia, que aporta la parte de la intervención. Al igual que a la hora de proponer el modelo de la serie preintervención se tuvieron en cuenta varias opciones.

Como primera opción se propuso un único cambio de nivel que se produce en el instante de la intervención. Esto se hizo mediante una función de transferencia $w(B) = w_0$ aplicada al instante 173 de la serie, que teniendo en cuenta que se han quitado los 30 primeros datos se transforma en un 143 en el modelo.

```
# Opción1
> x.143 <- 1*(seq(serie_log)>=143)
> ajuste3 <- arimax(serie_log, order=c(5,0,0), xtransf=data.frame(S143=x.143),
 transfer=list(c(0,0)),fixed=c(NA,0,0,0,NA,NA,NA))
...
Coefficients:
         ar1  ar2  ar3  ar4     ar5  intercept  S143-MA0
      0.6133    0    0    0  0.1263     8.9582   -0.5970
s.e.  0.0510    0    0    0  0.0525     0.3721    0.5507

sigma^2 estimated as 1.482:  log likelihood = -384.85,  aic = 777.71
> abs(ajuste3$coef)[-c(2:4)]/(1.96*sqrt(diag(ajuste3$var.coef)))
      ar1        ar5  intercept   S143-MA0
6.1302478  1.2264419 12.2813296  0.5531229
```

El cociente anterior muestra que el coeficiente correspondiente a la intervención no es significativamente distinto de cero. Por lo que el modelo queda como el modelo AR(5) de la serie preintervención. La segunda opción incluye una función que modeliza el cambio de nivel en dos etapas, el instante de la intervención y el instante inmediatamente posterior.

```
> #Opción 2
> x.144 <- 1*(seq(serie_log)>=144)
> ajuste2 <- arimax(serie_log, order=c(5,0,0),fixed=c(NA,0,0,0,NA,NA,NA,NA),
+ xreg=data.frame(S143=x.143, S144=x.144))
....
Coefficients:
         ar1  ar2  ar3  ar4     ar5  intercept   S143    S144
      0.6158    0    0    0  0.1268     9.0194  0.352  -1.111
s.e.  0.0514    0    0    0  0.0529     0.3830  1.071   1.069
sigma^2 estimated as 1.475:  log likelihood = -384.31,  aic = 778.63
```

En este caso también se obtiene que los coeficientes de la intervención no son significativos. Por último, y por tener en cuenta otro tipo de posibilidades se considera que la intervención tenga un efecto transitorio, esto quiere decir que la serie sufre un cambio y se va recuperando gradualmente. Se considera una función de transferencia del tipo

$$w(B) = u(B) + v(B), \tag{2.1}$$

con $u(B) = w_0$ y $v(B) = \frac{w_1}{1-\delta B}$. Es decir, en el instante de la intervención se produce un efecto w_0+w_1 y j instantes después: $w_1\delta^j$ cuya notación en R es `c(0,0)` y `c(1,0)` respectivamente. Ajustando el modelo:

```
> x.143 <- 1*(seq(serie_log)>=143)
> ajuste1 <- arimax(serie_log, order=c(5,0,0),fixed=c(NA,0,0,0,NA,NA,NA,NA,NA)
+ , xtransf=data.frame(I143=x.143, I143=x.143), transfer=list(c(0,0), c(1,0)))
...
Coefficients:
         ar1  ar2  ar3  ar4     ar5  intercept  I143-MA0  I143.1-AR1
      0.6138    0    0    0  0.1265     8.9859    1.4138      0.2152
s.e.  0.0512    0    0    0  0.0528     0.3797    6.6130      0.4133
      I143.1-MA0
         -1.6441
s.e.      5.8287

sigma^2 estimated as 1.479:  log likelihood = -384.64,  aic = 781.29

> abs(ajuste1$coef)[-c(2:4)] / (1.96*sqrt(diag(ajuste1$var.coef)))
      ar1        ar5  intercept   I143-MA0 I143.1-AR1 I143.1-MA0
6.1130618  1.2232641 12.0743549  0.1090751  0.2657071  0.1439087
```

Como se puede observar los coeficientes referidos a la intervención no son significativos, siguiendo el proceso de eliminarlos uno a uno se llega a que ninguno es significativo. Por tanto, el modelo que se propone para realizar las predicciones es el modelo autorregresivo de orden 5, AR(5), que quedaría como se puede ver a continuación:

$$X_t = 8{,}7144 + 0{,}6155X_{t-1} + 0{,}1269X_{t-5} + a_t,$$

donde a_t representa un proceso de ruido blanco.

Nota 2.5. Si se presta atención a los valores de los coeficientes en la serie preintervención y se comparan con los distintos valores obtenidos en los modelos con intervención se puede apreciar que no son muy diferentes por lo que antes de mirar si son significativos ya cabía esperar que no lo fuesen.

2.3. Diagnosis de los modelos

Llegado el momento de efectuar la diagnosis existen métodos gráficos y contrastes numéricos. Hay que comprobar que el modelo no tenga datos atípicos además de que las innovaciones del modelo, que se aproximarán por los residuos del ajuste, sean ruido blanco. Es decir, tengan media cero, varianza constante y estén incorreladas. Se presenta en primer lugar la diagnosis del modelo que se nombró como Opción 1 siendo análogas las del resto de modelos. Además, para finalizar, se presentan las diagnosis de los modelos AR(1) y AR(2) que fueron descartados para la serie preintervención, para que se pueda ver que, efectivamente, no pasaron la diagnosis.

Modelo AR(5) Opción1

Se comprueba en primer lugar que la media es cero. Como se puede ver en la siguiente salida de R se tiene que el p-valor es grande, mayor que los niveles de significación usuales, por lo que no existen evidencias significativas para rechazar la hipótesis nula de que la media sea cero.

```
>t.test(residuals(ajuste3), mu=0)

One Sample t-test

data:  residuals(ajuste3)
t = 0.098984, df = 237, p-value = 0.9212
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
-0.1483214  0.1640147
sample estimates:
mean of x
0.00784666
```

La varianza constante se comprueba gráficamente representando los residuos del ajuste frente al tiempo. Tal y como se puede ver en la Figura 2.5 se cumple esta condición.

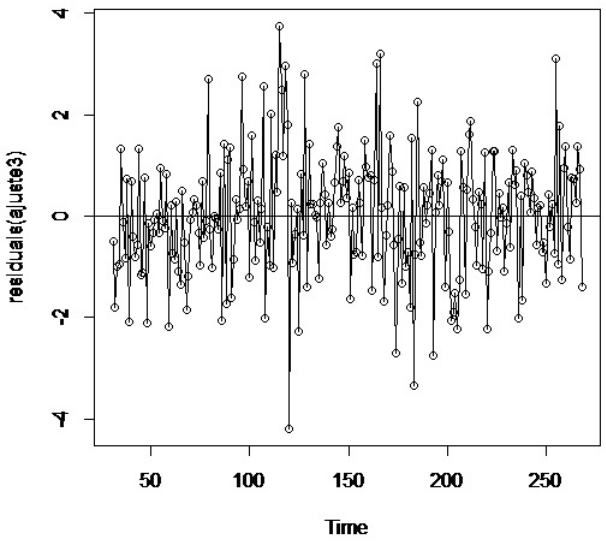


Figura 2.5: Residuos del ajuste frente al tiempo

Para la independencia se tiene el comando `tsdiag` que aplicado al ajuste muestra tres gráficos por pantalla, Figura 2.6. En el superior se muestran los atípicos, valores que tienen poca probabilidad de ocurrir dada la estructura habitual de evolución de la serie. Como se puede observar, no hay ninguno ya que de existir aparecerían unas líneas discontinuas rojas marcando el límite entre los puntos que se consideran normales y los atípicos. Además si se aplican sobre el ajuste los comandos propios para detectar atípicos aditivos e innovativos, `detectAO` y `detectIO` respectivamente, se confirma que no presenta atípicos de ninguno de los dos tipos. En el gráfico del medio se muestran las autocorrelaciones

de los residuos, éstas están todas entre las bandas rojas salvo una, esto no implica que no pase la diagnosis puesto que 1 de cada 20 retardos pueden sobresalir de entre las bandas debido a la aleatoriedad del proceso, por tanto se considera que están incorrelacionados. El último gráfico confirma que los p-valores son altos por lo que se cumple la hipótesis de incorrelación.

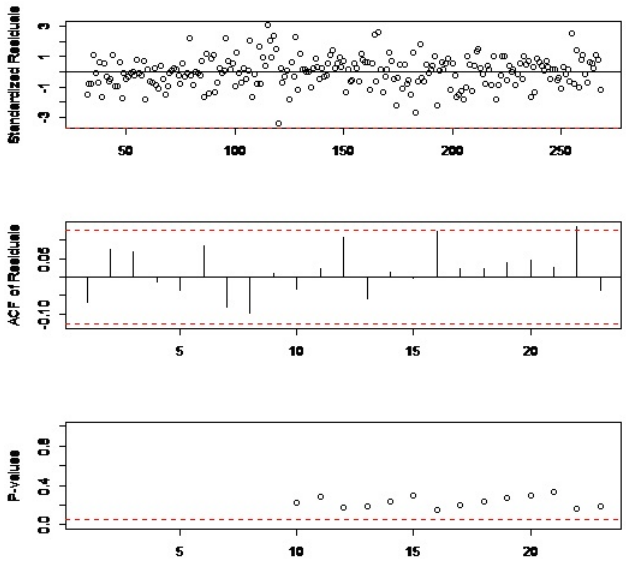


Figura 2.6: Diagnosis del modelo AR(5)

También se realizan contrastes de normalidad de los residuos porque en el caso de que las innovaciones sean gaussianas se tendrá automáticamente la independencia, además bajo normalidad los estimadores son asintóticamente eficientes y, para terminar, en este caso, se tiene que al realizar las predicciones (próxima sección) se puede garantizar el nivel de confianza. Gráficamente se tiene el QQ-plot, como se puede ver en la Figura 2.7 en los extremos los puntos se alejan de la línea recta por lo que podría presentarse una ausencia de normalidad, no obstante se comprobará mediante contrastes numéricos como el de Shapiro-Wilk o el de Jarque Bera. Los p-valores obtenidos con ambos contrastes se presentan en la Tabla 2.1. A la vista de los resultados se puede considerar que los residuos, y por tanto las innovaciones, son normales.

Contraste	P-valor	Estadístico
Jarque Bera	0.1676	3.5726
Shapiro-Wilk	0.2248	0.9920

Tabla 2.1: Resultados de los contrastes de normalidad.

Se tiene que las innovaciones del modelo son ruido blanco y gaussianas, además no hay presencia de atípicos por lo que el ajuste se da por bueno.

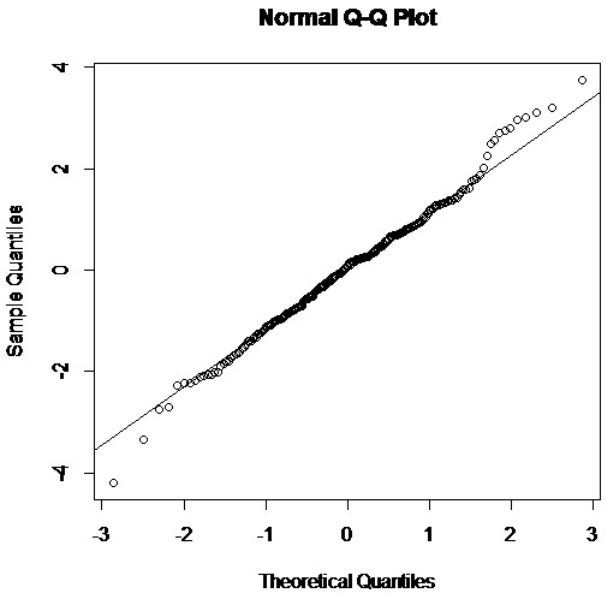
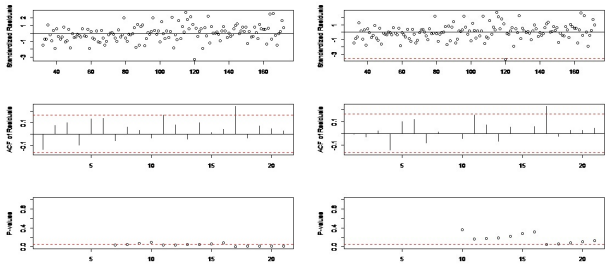


Figura 2.7: QQ-plot de los residuos del ajuste

AR(1) y AR(2)

En este caso se mostrará simplemente con el comando `tsdiag` que ambos modelos no pasaron la diagnosis, tal y como se dijo en un principio. Como se puede ver en la Figura 2.8, en ambos casos, existen p-valores que quedan por debajo de la banda roja. Por tanto, no se cumple la hipótesis de incorrelación y se descarta el modelo.



(a) Modelo AR(1) (b) Modelo AR(2)

Figura 2.8: Diagnosis de los modelos AR(1) y AR(2)

2.4. Predicción

Ahora que ya se tiene identificado el modelo, AR(5) con constante, fijando los parámetros no significativos a cero, se realizan las predicciones. Se realizan 12 predicciones y se comparan con los valores reales que se han apartado en la primera sección de este capítulo, para ello se emplearán las funciones `plot` y `predict`. Además, para realizar las predicciones de forma correcta, hay que deshacer la transformación logarítmica que se aplicó a la serie en la sección antes mencionada.

Gráficamente se tiene la Figura 2.9, sin embargo, interesa más la salida numérica. El valor de las predicciones se incluye en la Tabla 2.2 junto con el valor real y su diferencia en valor absoluto. Como se puede observar en dicha tabla, aunque algunos errores podrían ser admisibles en este caso (ya que lo que se quiere es dar un tiempo de espera aproximado al usuario) hay otros, como el que se produce en el instante $t = 6$, que son demasiado grandes.

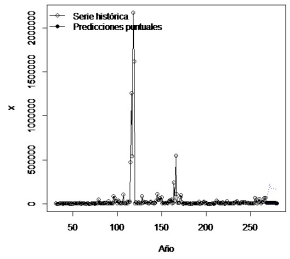
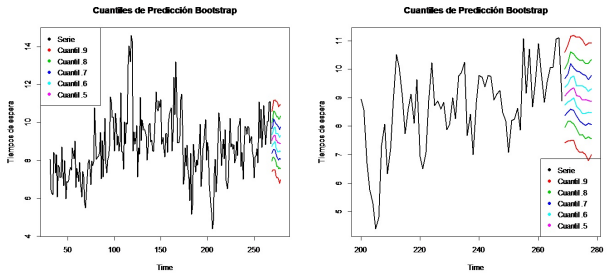


Figura 2.9: Predicción con innovaciones gaussianas.

Probando otro tipo de algoritmo de predicción, cuyo código se incluye en el Apéndice B (en la sección Predicción cuantiles bootstrap) y fue desarrollado por el director de este proyecto, se obtienen los cuantiles bootstrap de predicción. Se obtiene así la Figura 2.10 serie completa, donde se pueden observar gráficamente dichos resultados. Ahora bien, en esa gráfica se ven los logartimos de la serie y de las predicciones. Para verlo más claramente se hace un zoom sobre la parte final, para ello se emplea el comando `window` y se obtiene la Figura 2.10, final de la serie.



(a) Serie completa (b) Final de la serie

Figura 2.10: Cuantiles de predicción bootstrap

Instante	Valor Original	Predicción calculada	Error (diferencia en valor absoluto)
t=1	14420.2423	8017.4970	6402.7450
t=2	5583.8563	8561.1830	2977.3270
t=3	3307.8974	10122.7150	6814.8170
t=4	40136.9171	11278.6930	28858.2240
t=5	13285.3571	9095.8410	4189.5160
t=6	44233.8509	8072.4250	36161.4260
t=7	11517.7069	7563.2890	3954.4180
t=8	10019.7846	7422.0860	2597.6980
t=9	510.2351	7437.8180	6927.5830
t=10	10769.4503	7247.0470	3522.4030
t=11	160.3024	7024.8830	6864.5800
t=12	1.5000	6834.8140	6833.3140

Tabla 2.2: Valores reales, predicciones y errores cometidos en los 12 instantes siguientes a la finalización de la serie. Unidad de medida el segundo.

Capítulo 3

Simulación del sistema de colas

Para poder simular el sistema de colas del CESGA hay que conocer en profundidad las características del mismo. El sistema de colas está compuesto, como ya se ha dicho, por cinco colas principales y otras que son creadas para la realización de una tarea concreta y luego se eliminan. Como cada cola tiene unas características especiales se simularán individualmente.

La Teoría de Colas engloba distintos tipos o clases de colas, desde las más sencillas, las clásicas colas M/M/1, a otras más complejas como las redes de Jackson abiertas. Sin embargo, ninguna de estas colas, cuyo marco teórico se puede consultar en [2], cubre este caso particular. No obstante, como las colas M/M/1 son las más simples se intentará modelizar el sistema de colas como combinación de varias de ellas.

Este modelo consiste en modelar cada cola del sistema como combinación lineal de las colas M/M/1 como sigue:

$$q_i = c_1 q_{i1} + ... + c_j q_{ij} + ... + c_k 1_{ik}, \quad i \in \{1, ..., 5\}$$

donde q_{ij} es una cola M/M/1, $j \in \{1, ..., k\}$ con una tasa de servicio μ_{ij} y una tasa de llegada λ_{ij}, y donde c_j es la proporción de trabajos de cada clase. Esto se hará para una de las colas ya que para el resto simplemente habría que cambiar los parámetros.

3.1. Planteamiento de la simulación: fijando las clases

Cuando se tiene una cola M/M/1 se supone que la llegada de los clientes se produce siguiendo una distribución de Poisson de parámetro λ, y que el tiempo de servicio es exponencial con parámetro μ.

Antes se mencionaba que se emplearía una combinación lineal de colas para simular cada una de las colas principales, para lo que se fuerza la existencia de clases de trabajos en cada una de las colas. Estas clases se establecen en función de los tiempos de espera de los trabajos.

Al realizar un histograma de los tiempos de espera, aplicándole logaritmos a los mismos, se obtiene la Figura 3.1 en la que se puede apreciar que existen dos modas bien diferenciadas. En consecuencia se establecen dos clases de trabajos dentro de cada cola. Es decir, cada cola está formada por la combinación lineal de dos colas M/M/1, una con las características de los trabajos con tiempos de espera hasta 4.5 segundos, que una vez deshecho el logaritmo queda como 90.02 segundos, y la otra con el resto de trabajos. Puede darse la situación de que alguna de las colas presente sólo una de las clases, en este caso no se realiza combinación lineal de colas si no que se modeliza directamente como una cola M/M/1 con los parámetros obtenidos de todos los trabajos.

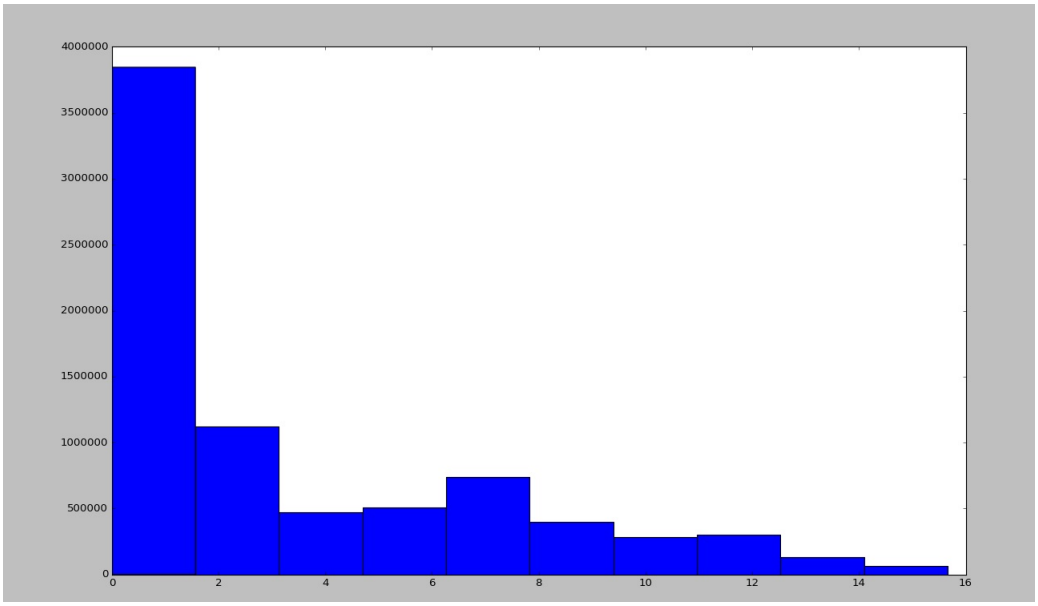


Figura 3.1: Histograma de los logaritmos de los tiempos de espera del sistema de colas.

3.2. Obtención de los datos de colas

Para explicar como se obtuvieron los datos necesarios para la simulación se parte de un `RDD` que contiene los datos de colas ya filtrados como se ha explicado anteriormente. A este `RDD` se le denominará `df`. Se incluye aquí paso por paso cómo se obtienen los datos para una cola, siendo análogos para el resto (simplemente cambiando el nombre de la cola en el filtrado).

Se sacan los datos en función de las dos categorías que hemos establecido en la sección anterior. Para ello se toman los datos originales, `df`, y se filtran en dos `RDD`, `tiempos1` y `tiempos2`, el primero contiene los datos de tiempos de espera menores y el segundo mayores. Además se añade una letra antes para hacer referencia a la cola, por ejemplo, la `s` para la cola pequeña. Quedando así:

```
s_tiempos1=df.filter(lambda x: x[0] == "small_queue").filter(lambda x: x[-2]-x[-1]
<90)
s_tiempos2=df.filter(lambda x: x[0] == "small_queue").filter(lambda x: x[-2]-x[-1]
>=90)
```

A continuación se cuenta el número de trabajos que hay en cada uno de los `RDD` para poder calcular la proporción de trabajos de esa cola que pertenecen a cada categoría. Como en otras ocasiones se utiliza la función `.count()`.

```
s_tiempos1.count()
s_tiempos2.count()
```

Se calcula la proporción de trabajos dividiendo el número de trabajos de `s_tiempos1` entre la suma de los trabajos de los dos `RDD`s. También se podría emplear la función `.count()` escribiendo `df.filter(lambda x: x[0] == "small_queue").count()`, pero es más rápido calcular la suma. Los resultados se recogen en la Tabla 3.1.

La tasa de servicio se calcula como la medida de la variable `ru_walk` que es el tiempo que tardó en ejecutarse el trabajo, con lo que encaja perfectamente en la definición de tasa de servicio. En este caso se utiliza la función `.mean()`.

Cola	Proporción (c_i)	Tasa de llegada (λ_{ij})	Tasa de servicio (μ_{ij})
Cola s < 90	0.5655	0.0023	10060.3336
Cola s ≥ 90	0.4345	0.0018	50603.2309
Cola l < 90	0.6757	0.0081	3172.2476
Cola l ≥ 90	0.3243	0.0039	5738.3219
Cola m < 90	0.7240	0.0089	4088.5281
Cola m ≥ 90	0.2760	0.0034	12835.5356
Cola i < 90	0.9876	0.0003	8219.4403
Cola i ≥ 90	0.0124	0.0000*	8030.3085
Cola lp < 90	0.9662	0.0017	253.2114
Cola lp ≥ 90	0.0338	0.0006	570.9745

Tabla 3.1: Datos para la simulación. Proporción de trabajos de cada cola según su categoría. Tasa de llegada de los trabajos por categoría. Tasa de servicio de los trabajos por categoría. (Tiempos en segundos). *En realidad el valor es 4.365717e-06 aunque en la tabla sólo se muestran cuatro decimales.

```
s_tiempos1.map(lambda x: x[13]).mean()
s_tiempos2.map(lambda x: x[13]).mean()
```

donde `13` es la posición en la que se encuentra almacenada la variable dentro del `RDD`.

El número de trabajos que ya se tiene se emplea también para calcular la tasa de llegada. Esta tasa se calcula como el número de trabajos por el tiempo total. Este tiempo se toma como la diferencia entre el mínimo y el máximo del instante en el que se envió el primer trabajo y el último empleando las funciones `.min()` y `.max()` respectivamente sobre el `RDD df`. Se calcula la división y se obtiene la tasa de interés. Estos datos y los análogos para las otras colas se pueden ver en la ya citada Tabla 3.1.

A la hora de realizar una simulación completa se necesita también la proporción de trabajos que llegan a cada una de las colas. El número de trabajos total viene dado por `df.count()`, en cuanto al número de trabajos por cola es la suma del correspondiente `s_tiempos1, s_tiempos2`. Se divide esta suma por el total y se obtiene la proporción de trabajos de cada cola como se puede ver en la Tabla 3.2.

Nota 3.1. Si se suman las proporciones de la Tabla 3.2 se aprecia que no suman uno. Esto se debe a que el 1.71 % de los trabajos se ejecutaron en colas temporales o no principales.

3.3. Simulando una cola

En este apartado se realizará la simulación de una de las colas, para la simulación del resto simplemente hay que cambiar el valor de los parámetros. La tasa de llegada de los trabajos de la clase 1 y 2 así como de su tasa de servicio. Además se introduce `n` que es el dato que incluye cuántos trabajos

	Proporción	Número de procesadores
Cola s	0.1283	16
Cola l	0.3893	75
Cola m	0.3791	40
Cola i	0.0112	2
Cola lp	0.0750	9

Tabla 3.2: Datos para la simulación. Proporción de trabajos que llegan a cada cola en función del número total de trabajos y número de procesadores de cada cola.

se pueden ejecutar a la vez. En este caso `n=16` ya que la cola tiene 16 nodos cada uno con 16 slots. Aunque existen trabajos que ocupan sólo unos procesadores en un nodo y otros que ocupan varios nodos, para esta simulación se considera que cada trabajo ocupa un nodo completo y sólo uno.

Nota 3.2. A partir de este momento se utilizará el término procesador para referirse a los nodos.

Por tanto se genera un vector de ceros de tamaño `n=16` donde se marcará si el procesador está ocupado o no. A este vector se le denominará `servicio`, para poder saber cuando queda libre se le irán introduciendo los tiempos que tardan en ejecutar dichos trabajos y, cada vez que el tiempo avance, se le restará un instante temporal hasta llegar a cero y quedar nuevamente vacío. El número de procesadores que tienen las distintas colas se pueden observar también en la ya citada Tabla 3.2.

Todos los datos se guardarán en una matriz de dimensiones `Bx6` donde se almacena el instante temporal, número de trabajos que llegan en cada instante, el tiempo que estará en ejecución cada trabajo, el estado en que se encuentra (0=en espera, 1= en ejecución, 2= finalizado), el tiempo final que no es otra cosa que el instante en el que se terminará de ejecutar el trabajo, y el tiempo de espera que es como un contador que cada instante que el trabajo no se está ejecutando añade un `+1` en esta columna. Cuando un servidor está libre el trabajo comienza a ejecutarse y, cuando no, pasa a esperar.

En primer lugar se generan `B` poissones para cada una de las clases. Así, según como se han presentado las colas se hacen el número de trabajos que llegan de cada clase. Además mediante un bucle se le asigna el instante de tiempo el número de trabajos que llegan y el tiempo que está en ejecución dicho trabajo. Este último tiempo se genera a partir de la exponencial correspondiente (en función de si el trabajo pertenece a una clase o a otra).

El código con el que se genera dicha matriz se puede ver a continuación:

```
B=1440
clase1=rpois(B,lambda11)
clase2=rpois(B,lambda12)
d1=clase1
d2=clase2

matriz=cbind(0,0,0,0,0,0)

for (i in 1:B){
   if((clase1[i]+clase2[i]) == 0){
      matriz=rbind(matriz,c(i,0,0,0,0,0))
   }else{while(clase1[i]!=0){
```

```
            a=c(i,1,rexp(1,mu11),0,0,0)
            matriz=rbind(matriz,a)
            clase1[i]=clase1[i]-1}
         while(clase2[i]!=0){b=c(i,1,rexp(1,mu12),0,0,0)
            matriz=rbind(matriz,b)
            clase2[i]=clase2[i]-1}
   }
}

colnames(matriz)=c("tiempo","num_trab","tservicio","estatus","tfinal",
"tespera")

matriz
```

De esta forma si llegan varios trabajos a la vez existirían varias filas que harían referencia al mismo instante temporal. Se muestran a continuación las 10 primeras filas de la matriz en la que se puede apreciar que el primer trabajo entra en el instante 9 y que este trabajo tardará en ejecutarse 201.096 minutos.

```
> matriz[1:10,]
tiempo num_trab tservicio estatus tfinal tespera
0        0     0.000       0      0       0
1        0     0.000       0      0       0
2        0     0.000       0      0       0
3        0     0.000       0      0       0
4        0     0.000       0      0       0
5        0     0.000       0      0       0
6        0     0.000       0      0       0
7        0     0.000       0      0       0
8        0     0.000       0      0       0
9        1   201.096       0      0       0
```

El siguiente paso es hacer un bucle para asignar a cada trabajo el tiempo en que acabará de ejecutarse, suponiendo que comience su ejecución en el instante en el que llega al sistema. Es decir, el tiempo de servicio más el instante temporal en el que llega al sistema (suma de las columnas 1 y 3). En caso de que el trabajo entre en espera se le sumará también la cantidad de tiempo que está esperando, sin embargo, esto se hará más adelante (en función de lo que se vaya obteniendo en la columna 6). Además se aprovecha este mismo bucle para asignar un estatus de finalizado a todos los instantes en los que no llegan trabajos nuevos.

```
servicio=numeric(n)
m=dim(matriz)[1]

# Acordarse de ignorar la primera fila de tiempo = 0
copia=matriz
matriz=copia
for (j in 1:m){
   if(sum(as.numeric(matriz[1:j,2]==0))==0){
      matriz[1:j,4]==2
   }else{
      for (i in 1:j){
         if(matriz[i,3]==0){matriz[i,5]=0
         }else{matriz[i,5]=matriz[i,3]+matriz[i,1]
```

```
         }
      }
   }
}
```

En los siguientes párrafos se presenta un resumen del funcionamiento del bucle que imita todo el proceso de llegada y ejecución de los trabajos. Este bucle es de tamaño `B=1440`, ya que son los minutos que hay en un día:

Para cada instante anterior al actual se comprueba en primer lugar si la tercera columna es menor que 1, es decir, si el tiempo que le queda al trabajo en el sistema es menor que uno. Recuérdese que en esta columna lo que se tenía almacenado era el tiempo total de servicio por lo que habrá que ir descontando un minuto por cada vuelta del bucle. Si este tiempo es menor que uno entonces se considera que el trabajo ha terminado su ejecución y se pasa a estatus 2 (finalizado).

Además se comprueba si han llegado trabajos en ese instante o en los anteriores lo cual implica que cuando un trabajo empieza a ejecutarse también hay que descontarlo de la columna de llegada, la columna 2. Si han llegado trabajos entonces se empieza otro bucle:

Este nuevo bucle recorre todos los instantes previos. Si la segunda columna es no nula entonces hay trabajos que meter en el sistema. Llegados a este punto se tienen dos opciones: la primera que haya sitio en el servidor para meter el trabajo. En este caso se almacena el número de trabajos de ese instante en una variable `b`.

Se establece que como ha entrado en ejecución el número de trabajos en espera ahora es cero (columna 2), el nuevo estatus de ese instante se cambia por 1 ,en ejecución, (columna 4) y el tiempo en el que acabará de ejecutarse es el tiempo final que ya se había calculado más el tiempo de espera (es decir, la nueva columna 5 es la antigua columna 5 más la 6). Este trabajo ocupa un sitio en el servidor, para ello se hace un bucle y se introduce el valor del tiempo de servicio en el primer valor nulo del vector servicio.

En caso contrario no hay sitio y se pasa a espera. Es decir, el estatus del trabajo es en espera (0) y en la columna del tiempo de espera hay que sumarle un instante más.

Al final de cada vuelta se hace una actualización de los tiempos. Esto significa que se restará un instante (un minuto) a los trabajos que se están ejecutando (es decir, columna 4 =1), sumar uno a los que están esperando (columna 4=0 y columna 2 no nula por lo que hay trabajos en ejecución). También se actualiza el tiempo de servicio. Si el servicio es menor que 1 se asume que el trabajo se acabó de ejecutar. En caso contrario se le resta un instante de tiempo y vuelve a empezar todo el proceso.

```
servicio=numeric(n)
for (j in 1:m){
   for(a in 1:j){if(matriz[a,3]<1){matriz[a,3]=0
      matriz[a,4]=2}}
      if(sum(as.numeric(matriz[1:j,2]==0))==0){
         matriz[1:j,4]==2
      }else{
         for (i in 1:j){
            if(matriz[i,2]!=0){
               # hay trabajos que meter
               if(sum(as.numeric(servicio>=1))!=n){
                  # hay sitio para meter el trabajo
                  b=matriz[i,2]
                  matriz[i,2]=0
                  matriz[i,4]=1
                  matriz[i,5]=matriz[i,5]+matriz[i,6]
                  for (s in 1:n){
```

```
                    if(servicio[s]<1 & b!=0){
                       servicio[s]=matriz[i,3]
                       b=b-1
                    }
                 }
              }else{# no hay sitio y se pasa a espera
                 matriz[i,6]=matriz[i,6]+1
                 matriz[i,4]=0
              }
           }
        }
     }
# Al final de cada vuelta hay que sumar uno a los que están esperando y
# restar uno a los que se están ejecutando.
     for(t in 1:j){
        if(matriz[t,4]==1){matriz[t,3]=matriz[t,3]-1}
        if(matriz[t,4]==0 & matriz[t,2]!=0){matriz[t,6]=matriz[t,6]+1}
     }
# Se actualiza el tiempo de servicio
     for (u in 1:n){if(servicio[u]<1){servicio[u]=0}else{servicio[u]=servicio[u]-1}}
}
```

Puede llamar la atención que no se emplea en ninguna parte del código la proporción de trabajos que existen de cada clase. En un primer momento se pensaba emplear una Bernouilli para determinar si el trabajo entrante pertenecía a la clase 1 o a la clase 2, y así asignarle las tasas correspondientes. Sin embargo, esta idea se descartó en favor de la que se puede observar en el código contenido en esta sección debido al hecho de que la llegada de un trabajo de clase 1 no implica que no pueda llegar también un trabajo de clase 2 en ese mismo instante. Por tanto, se generó la llegada de trabajos de las dos clases sin emplear la proporción de forma directa, aunque sí indirectamente debido a la forma de obtener los parámetros de las distribuciones empleadas, para simular la llegada y el tiempo de ejecución, en cada una de las clases.

3.4. Resultados obtenidos

Por como está diseñada la matriz antes de hacer ningún estudio sobre ella deben eliminarse las filas correspondientes a los instantes en los que no entraron trabajos. Como se han ido modificando los tiempos de servicio y la columna de los números de trabajos que llegan se seleccionan las tres primeras columnas de la copia de seguridad, que se elaboró al comienzo cuando se indican los trabajos que llegan (en qué instante y cuánto tardan en ejecutarse), y las tres últimas de la matriz obtenida al final del bucle. Posteriormente se eliminan las filas que no tienen trabajos.

```
a=cbind(copia[,1:3],matriz[,4:6])
c=c(0,0,0,0,0,0)
m=dim(a)[1]
for (i in 1:m){
   if(a[i,5]!=0){c=rbind(c,a[i,])}
}
```

Repitiendo `R` veces todo lo anterior se puede obtener un vector de dicho tamaño donde se almacenan diferentes características:

- El número de trabajos que han empezado a ejecutarse en un día.
- El número de trabajos que terminaron su ejecución en un día.
- El tiempo de espera medio por día.
- El tiempo medio de servicio por día.
- El tiempo medio que cada trabajo ha estado en el sistema.

Los vectores referidos a los dos primeros puntos se calculan mediante contadores y de los tres últimos mediante medias de columnas de la matriz. El código completo se encuentra en la sección Código completo simulación del Apéndice B.

Se puede obtener así la probabilidad de que empiecen a ejecutarse más de `x` trabajos, Tabla 3.3. La probabilidad de que se ejecuten más de `y` trabajos, es decir, que comiencen y finalicen su ejecución en el propio día, Tabla 3.4.

	Cola s	Cola l	Cola m	Cola i	Cola lp
$\mathbb{P}(X \geq 70)$	0.58	1.00	1.00	0.00	1.00
$\mathbb{P}(X \geq 80)$	0.25	1.00	1.00	0.00	1.00
$\mathbb{P}(X \geq 90)$	0.08	1.00	1.00	0.00	1.00
$\mathbb{P}(X \geq 100)$	0.00	1.00	1.00	0.00	1.00

Tabla 3.3: Número de trabajos que empiezan a ejecutarse en un día.

	Cola s	Cola l	Cola m	Cola i	Cola lp
$\mathbb{P}(Y \geq 30)$	1.00	1.00	1.00	0.02	1.00
$\mathbb{P}(Y \geq 40)$	0.99	1.00	1.00	0.00	1.00
$\mathbb{P}(Y \geq 60)$	0.37	1.00	1.00	0.00	1.00
$\mathbb{P}(Y \geq 80)$	0.02	1.00	1.00	0.00	1.00

Tabla 3.4: Número de trabajos que terminan su ejecución en el propio día.

Por otro lado, para obtener un promedio de los tiempos de espera se almacena en dicho vector la media de la sexta columna. A calcular la media de este vector se obtiene que el tiempo de espera medio es de 1099.869 minutos en un día.

Realizando de forma similar repeticiones se tiene que el tiempo medio de servicio es 470.548 minutos. Y el promedio de tiempo que están un trabajo en el sistema es de 2286.32. Todo esto para la cola s que fue la que se simuló en este capítulo.

Se pueden aprovechar los datos de la simulación, de la última matriz, para representar el número de trabajos esperando en función del tiempo, es decir, el número de trabajos que están esperando en cada instante, Figura 3.2. Para ello se crea un vector de tamaño el número de instantes (1440), para cada trabajo con tiempo de espera no nulo se suma uno para cada vector en ese instante y en los `r` siguientes, siendo `r` el tiempo en minutos que espera el trabajo para comenzar su ejecución.

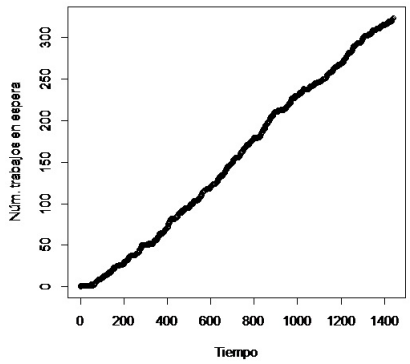


Figura 3.2: Tiempo en minutos que está el trabajo en espera frente al instante temporal

Capítulo 4

Resumen y continuación

La primera parte del trabajo refleja el camino del aprendizaje de PySpark para cualquier persona que quiera estudiar en profundidad este tema. Aunque no es una parte muy atractiva inicialmente, es completamente vital para entender el problema, sacarle partido a los datos y progresar en el resto de tareas. Suele consumir mucha parte del tiempo de un iniciado ya que no es una herramienta popular entre estudiantes o investigadores ajenos a sistemas de supercomputación.

En cuanto a la predicción con series de tiempo se han probado multitud de modelos de tipo Box Jenkins para modelizar los tiempos de espera de las colas. Sin embargo, aunque el modelo ajusta la serie, el error de predicción en algunos casos es elevado, por lo que podrían emplearse otro tipo de modelos de series de tiempo para tratar de realizar las predicciones.

Por último se trató de realizar una simulación del sistema de colas, es en esta parte donde sí se podría tratar de hacer una futura ampliación (si alguien quisiese). Se han incluído las principales características de las colas en su simulación de forma que se han sacado nuevos datos de ellas. Sin embargo, aún queda mucho por explicitar.

Por ejemplo, cuando un usuario solicita ejecutar un trabajo en el sistema de colas selecciona el número de nodos que va a utilizar y el número de procesadores dentro de cada nodo. Para la simulación se ha supuesto que todos los trabajos ocupaban lo mismo y, en cierto modo, los que ocupan muchos nodos se compensan con los que ocupan pocos. La introducción de esta nueva información llevaría a una simulación más compleja, aunque habría que ver como de diferentes son los resultados obtenidos.

En conclusión, el trabajo realizado ha servido para:

- Ver que la media de los tiempos de espera por sí sola no es la forma idónea de indicar a los clientes cuánto tendría que esperar un trabajo que enviasen para su ejecución en el sistema de colas. Si no que, proporcionando los datos dados por los cuantiles, la información se aproxima más a la realidad.

- Realizando las predicciones de los tiempos de espera mediante modelos Box Jenkins se tiene que, en el peor de los casos, el error es de unas 10 horas (según la Tabla 2.2 del Capítulo 2). Empleándose otros modelos de series de tiempo podrían obtenerse mejores predicciones.

- Se ha aportado la simulación de las colas principales de tal forma que se puede modificar y readaptar a otras colas simplemente variando los parámetros. Esta simulación ayuda a una mejor comprensión de cómo funciona el sistema de colas, pudiendo ser mejorada como se indicó anteriormente

Apéndice A

Código Python

A.1. leer_acct.py

```
# lee el fichero acct y devuelve un rdd llamado records (ya filtrado)
# se cargan los módulos que se van a usar
import struct
from collections import namedtuple
import resource
import time
import pandas as pd

# se define la función expand explicada en el Capítulo 1
def expand(dato):
   return (dato & 0x1ffff) << (((dato >> 13) & 0x7) * 3);

# se da nombre a las variables en forma de tupla
AcctRecord = namedtuple('AcctRecord',
'flag version tty exitcode uid gid pid ppid '
'btime etime utime stime mem io rw minflt majflt swaps '
'command')


def read_record(data):
   AZH=100;
   values = struct.unpack("2BH6If8H16s", data)
   print(type(values))
   # Se eliminan los bits de relleno de la variable command
   command = values[-1].replace('\x00', '')
   # Se reemplazan las variables por sus transformadas
   t1 = time.asctime(time.localtime(values[8]))
   t2 = expand(values[10])
   t3 = values[9]/AZH
   t4 = expand(values[11])
   t5 = expand(values[12])
   values =values[:8] + (t1, ) + (t3, ) + (t2, ) + (t4, ) + (t5, ) + values[13:-1]
    + (command, )
   return AcctRecord(*values)
```

```
# lista

lista=['acpi_available','ip','parted','agetty',
.... # se omite parte de la lista debido a su extensión
,'mshortname','zlib-flate']


#lectura y filtrado

rdd = sc.binaryRecords('nombre_del_fichero', recordLength=64)
records = rdd.map(read_record).filter(lambda x: x[-1] not in lista)
```

A.2. CPU_rdd.py

```
# Primero cargar leer_acct.py
# Se cargan los módulos que se van a necesitar
from pyspark.sql.types import *
import pandas as pd
from pyspark.sql.functions import lit
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np


# función del cálculo de consumo por usuario, grupo, comando y empresa
def usoCpu_tablas(d):
   total=d.groupBy().sum('cons').collect()
   f=d.filter(d.cons != 0) # para evitar que la division de null
   f=f.select(f.uid,f.gid,f.command,np.divide(f.cons,f.etime).alias('frac'),
   f.institucion)
   f_total=f.groupBy().sum('frac').collect()
   user=d.select(d.uid,d.cons).groupBy('uid').sum('cons')
   user=user.toDF('uid','suma_cons').select('uid','suma_cons',lit(total[0][0])
   .alias('total'))
   b1=user.toPandas()
   prop=b1.suma_cons/b1.total
   b1['prop']=prop*100
   f_user=f.select(f.uid,f.frac).groupBy('uid').sum('frac')
   f_user=f_user.toDF('uid','suma_frac').select('uid','suma_frac',lit(f_total[0][0])
   .alias('total'))
   c1=f_user.toPandas()
   f_prop=c1.suma_frac/c1.total
   c1['f_prop']=f_prop*100
   group=d.select(d.gid,d.cons).groupBy('gid').sum('cons')
   group=group.toDF('gid','suma_cons').select('gid','suma_cons',lit(total[0][0])
   .alias('total'))
   b2=group.toPandas()
   prop=b2.suma_cons/b2.total
   b2['prop']=prop*100
   f_group=f.select(f.gid,f.frac).groupBy('gid').sum('frac')
```

```
   f_group=f_group.toDF('gid','suma_frac').select ('gid','suma_frac',lit(f_total
   [0][0]).alias('total'))
   c2=f_group.toPandas()
   f_prop=c2.suma_frac/c2.total
   c2['f_prop']=f_prop*100
   comm=d.select(d.command,d.cons).groupBy('command').sum('cons')
   comm=comm.toDF('command','suma_cons').select('command','suma_cons',lit(total
   [0][0]).alias('total'))
   b3=comm.toPandas()
   prop=b3.suma_cons/b3.total
   b3['prop']=prop*100
   f_comm=f.select(f.command,f.frac).groupBy('command').sum('frac')
   f_comm=f_comm.toDF('command','suma_frac').select('command','suma_frac',lit(
   f_total[0][0]).alias('total'))
   c3=f_comm.toPandas()
   f_prop=c3.suma_frac/c3.total
   c3['f_prop']=f_prop*100
   inst=d.select(d.institucion,d.cons).groupBy('institucion').sum('cons')
   inst=inst.toDF('institucion','suma_cons').select('institucion','suma_cons',lit(
   total[0][0]).alias('total'))
   b4=inst.toPandas()
   prop=b4.suma_cons/b4.total
   b4['prop']=prop*100
   f_inst=f.select(f.institucion,f.frac).groupBy('institucion').sum('frac')
   f_inst=f_inst.toDF('institucion','suma_frac').select('institucion','suma_frac',
   lit(f_total[0][0]).alias('total'))
   c4=f_inst.toPandas()
   f_prop=c4.suma_frac/c4.total
   c4['f_prop']=f_prop*100
   return(b1,b2,b3,b4,c1,c2,c3,c4)

# función de elaboración de gráficos
def graf(b1,b2,b3,b4):
   a1=b1[b1.prop != 0]
   del a1['total']
   a2=b2[b2.prop != 0]
   del a2['total']
   a3=b3[b3.prop != 0]
   del a3['total']
   a4=b4[b4.prop != 0]
   del a4['total']
   plt.ion()
   a1.set_index('uid').plot(kind = 'bar')
   plt.legend(bbox_to_anchor=(1, 1),
   bbox_transform=plt.gcf().transFigure)
   plt.title('por usuario')
   a2.set_index('gid').plot(kind = 'bar')
   plt.legend(bbox_to_anchor=(1, 1),
   bbox_transform=plt.gcf().transFigure)
   plt.title('por grupo')
   a3.set_index('command').plot(kind = 'bar')
   plt.legend(bbox_to_anchor=(1, 1),
```

```
   bbox_transform=plt.gcf().transFigure)
   plt.title('por comando')
   a4.set_index('institucion').plot(kind = 'bar')
   plt.legend(bbox_to_anchor=(1, 1),
   bbox_transform=plt.gcf().transFigure)
   plt.title('por institucion')
   plt.show()
   return


# función principal
def usoCPU(C,i):
   rdd=sc.binaryRecords('/accounting/pacct-uncompress/FT/20140101/acct-%s-20140101' %C ,
    recordLength=64)
   records=rdd.map(read_record).filter(lambda x: x[-1] not in lista)
   d=records.map(lambda x: (x[4],x[5],x[9],x[10]+x[11],x[-1]))
   for j in range(1,3+1):
      for k in range(1,31+1):
         if (j == 3) & (k>26): break
         if (j == 2) & (k>28):
            pass
         elif (j == 1) & (k == 1):
            pass
         elif (j == 4) & (k>30):
            pass
         elif (j == 6) & (k>30):
            pass
         elif (j == 9) & (k>30):
            pass
         elif (j == 11) & (k>30):
            pass
         else:
            try:
               rdd=sc.binaryRecords('/accounting/pacct-uncompress/FT/%d%02d%02d/acct-%s-*'
                % (i,j,k,C), recordLength=64)
               records = rdd.map(read_record).filter(lambda x: x[-1] not in lista)
               dat=records.map(lambda x: (x[4],x[5],x[9],x[10]+x[11],x[-1]))
               d=d.union(dat)
            except:
               print("No existe el archivo para %d%02d%02d" % (i,j,k))
   empresas=pd.read_csv(r'dump.csv')
   empresas=sqlContext.createDataFrame(empresas,('login','uid','institucion','centro',
   'departamento'))
   df=d.toDF(('uid','gid','etime','cons','command'))
   tabla=df.join(empresas,df.uid == empresas.uid,'inner').select(df.uid,df.gid,df.etime,
   df.cons,df.command,empresas.institucion)
   s=usoCpu_tablas(tabla)
   print('Uso de la CPU %s por usuario:' %C)
   print s[0]
   print('Uso de la CPU %s por grupo:' %C)
   print s[1]
   print('Uso de la CPU %s por comando:' %C)
```

```
   print s[2]
   print('Uso de la CPU %s por institucion:' %C)
   print s[3]
   graf(s[0],s[1],s[2],s[3])
   return (df,tabla,s)


def graf_tarta(C,a,b):
   plt.pie(a.prop, labels = a.institucion)
   plt.title('uso CPU %s en el %d' % (C,b))
   return


sol=usoCPU('cn100',2014)

# gráfico de tartas por institución
graf_tarta('cn100',sol[2][3],2014)
```

A.3. lectura_colas.py

```
# Lectura de los datos de colas.

# Se cargan los módulos que sean necesarios:
from pyspark.sql.types import *
import struct


# Transforma el tiempo GMT Unix a formato fecha
def fecha(a):
   import datetime
   b=datetime.datetime.fromtimestamp(int(a)).strftime('%Y-%m-%d %H:%M:%S')
   return b


# Función que estructura las "lineas" que se le pasen en forma de data frame SQL
def datos_colas(lineas):
   colas=lineas.map(lambda x: x.split(':'))
   schema = StructType([
   StructField("qname", StringType(), True),
   StructField("hostname", StringType(), True),
   StructField("groupname", StringType(), True),
   StructField("owner", StringType(), True),
   StructField("jobname", StringType(), True),
   StructField("jobnumber", LongType(), True),
   StructField("account", StringType(), True),
   StructField("priority", IntegerType(), True),
   StructField("qsub_time", StringType(), True),
   StructField("start_time", StringType(), True),
   StructField("end_time", StringType(), True),
```

```
   StructField("failed", IntegerType(), True),
   StructField("exit_status", IntegerType(), True),
   StructField("ru_wallclock",LongType(), True),
   StructField("ru_utime", DoubleType(), True),
   StructField("ru_stime", DoubleType(), True),
   StructField("ru_ixrss", LongType(), True),
   StructField("ru_ismrss", LongType(), True),
   StructField("ru_idrss", LongType(), True),
   StructField("ru_isrss", LongType(), True),
   StructField("ru_minflt", LongType(), True),
   StructField("ru_majflt", LongType(), True),
   StructField("ru_nswap", LongType(), True),
   StructField("ru_oublock", LongType(), True),
   StructField("ru_msgsnd", LongType(), True),
   StructField("ru_msgrcv", LongType(), True),
   StructField("ru_nsignals", LongType(), True),
   StructField("ru_nvcsw", LongType(), True),
   StructField("ru_nivcsw", LongType(), True),
   StructField("project", StringType(), True),
   StructField("department", StringType(), True),
   StructField("granted_pe", StringType(), True),
   StructField("slots", IntegerType(), True),
   StructField("na1", StringType(), True),
   StructField("cpu", DoubleType(), True),
   StructField("mem", DoubleType(), True),
   StructField("na2", StringType(), True),
   StructField("command_line_arguments", StringType(), True),
   StructField("na3", StringType(), True),
   StructField("na4", StringType(), True),
   StructField("start", LongType(), True),
   StructField("send", LongType(), True)
   ])
   b=sqlContext.createDataFrame(colas.map(lambda x: (x[0], x[1], x[2], x[3], x[4],
   long(x[5]), x[6], int(x[7]), fecha(x[8]), fecha(x[9]), fecha(x[10]), int(x[11]),
   int(x[12]), long(x[13]), float(x[14]), float(x[15]), long(x[17]),  long(x[18]),
   long(x[19]), long(x[20]), long(x[21]), long(x[22]), long(x[23]), long(x[25]),
   long(x[26]), long(x[27]), long(x[28]), long(x[29]), long(x[30]), x[31], x[32],
   x[33], int(x[34]), x[35], float(x[36]), float(x[37]), x[38], x[39], x[40], x[41],
   long(x[9]),long(x[8]))), schema)
   # las variables start y send time se almacenan en dos formatos distintos para
   tener la fecha y al mismo tiempo mantener el tiempo en segundos (para el cálculo
   de la variable tiempo de espera)
   return b

# ejemplo de lectura de colas, lectura año 2011:
lineas = sc.textFile("/accounting/sge/FT/2011*")
df=datos_colas(lineas).rdd
```

A.4. media_moda.py

```
# Antes cargar lectura_colas.py

lineas = sc.textFile("/accounting/sge/FT/2011*")

# Media y moda por cola:

def m(lineas,cola):
   df=datos_colas(lineas).rdd
   # filtramos tiempos de espera negativos
   df=df.filter(lambda x: x[-2]-x[-1] > 0)
   # filtramos datos que no se envían (los de instalación de colas)
   df=df.filter(lambda x: x[-1] != 0)
   df=df.map(lambda x: (x[0],x[-2]-x[-1]))
   # cálculo de la media global y por cola
   media=df.map(lambda x: x[-1]).mean()
   media_cola=df.filter(lambda x: x[0] == cola).map(lambda x: x[-1]).mean()
   # cálculo de la moda global y por cola
   a=df.toDF(('queue','wait'))
   moda=a.groupBy(a.wait).count().orderBy('count').collect()[-1]
   moda_cola=a.filter(a.queue == cola).groupBy(a.wait).count().orderBy('count').
   collect()[-1]
   return (media,media_cola,moda,moda_cola)

# Ejemplo:
lineas = sc.textFile("/accounting/sge/FT/2011*")
a=m(lineas,'nombre_cola')



# Media y moda por fecha:

# indicar la fecha en formato 201510*

def mf(fecha):
   lineas = sc.textFile("/accounting/sge/FT/%d*" %(fecha))
   df=datos_colas(lineas).rdd
   # filtramos tiempos de espera negativos
   df=df.filter(lambda x: x[-2]-x[-1] > 0)
   # filtramos datos que no se envían (los de instalación de colas)
   df=df.filter(lambda x: x[-1] != 0)
   df=df.map(lambda x: (x[0],x[-2]-x[-1]))
   # cálculo de la media por fecha
   media=df.map(lambda x: x[-1]).mean()
   # cálculo de la moda por cola
   a=df.toDF(('queue','wait'))
   moda=a.groupBy(a.wait).count().orderBy('count').collect()[-1]
   return (media,moda)

# Ejemplo:
```

```
mf(201110)


# Media y moda por empresa:


def mE(lineas,empresa):
   import pandas as pd
   df=datos_colas(lineas).rdd
   # filtramos tiempos de espera negativos
   df=df.filter(lambda x: x[-2]-x[-1] > 0)
   # filtramos datos que no se envían (los de instalación de colas)
   df=df.filter(lambda x: x[-1] != 0)
   pdf=df.map(lambda x: (x[3],x[-10],x[-5],x[-2]-x[-1]))
   pdf=pdf.toDF(('owner','slots','command','wait'))
   empresas=pd.read_csv(r'dump.csv')
   empresas=sqlContext.createDataFrame(empresas,('login','uid','institucion','centro',
   'departamento'))
   tabla=pdf.join(empresas,pdf.owner == empresas.login,'inner').select(pdf.owner,
   pdf.slots,pdf.command,pdf.wait,empresas.institucion)
   # cálculo de la media por institución
   media=tabla.map(lambda x: x[-2]).mean()
   # cálculo de la moda por empresa
   a=tabla.filter(tabla.institucion == '%s' %empresa)
   moda=a.groupBy(a.wait).count().orderBy('count').collect()[-1]
   return (media,moda)

# Ejemplo:
lineas = sc.textFile("/accounting/sge/FT/2011*")
mE(lineas,'nombre_empresa')
```

A.5. cuantiles.py

```
# Cargar primero lectura_colas.py
# q= cuantil que se quiere calcular 95% etc. sin el %
# lineas= las lineas donde se carga el fichero correspondiente


def cuantil(q,lineas):
   df=datos_colas(lineas).rdd
   # filtramos tiempos de espera negativos
   df=df.filter(lambda x: x[-2]-x[-1] > 0)
   # filtramos datos que no se envían (los de instalación de colas)
   df=df.filter(lambda x: x[-1] != 0)
   a=df.map(lambda x: (x[-2]-x[-1])).collect()
   n=len(a)
   ordenados=sorted(a)
   b=100/float(q)
   if n%b==0:
       c=(ordenados[int(n/b)] + ordenados[int(n/b)+1]) / 2
```

```
   else:
       c=ordenados[int(n/b)]*1
   return c


def cuantil_cola(q,lineas,cola):
   df=datos_colas(lineas).rdd
   # filtramos tiempos de espera negativos
   df=df.filter(lambda x: x[-2]-x[-1] > 0)
   # filtramos datos que no se envían (los de instalaicón de colas)
   df=df.filter(lambda x: x[-1] != 0)
   df=df.filter(lambda x: x[0] == '%s' % cola)
   a=df.map(lambda x: (x[-2]-x[-1])).collect()
   n=len(a)
   ordenados=sorted(a)
   b=100/float(q)
   if n%b==0:
      c=(ordenados[int(n/b)] + ordenados[int(n/b)+1]) / 2
   else:
      c=ordenados[int(n/b)]*1
   return c

# Por ejemplo

lineas = sc.textFile("/accounting/sge/FT/2011*")
cuantil(95,lineas)
cuantil_cola(95,lineas,"nombre_cola")
```

Apéndice B

Código R

B.1. Predicción cuantiles bootstrap

```
arima.simforecast=function (arma, n.ahead=1, series=NULL,rand.gen = rnorm,
n.start = NA, std=TRUE,nrep=100,...)
{
if (is.null(series)) series=get(arma$series)

season=arma$arma[5]
d=arma$arma[6]
D=arma$arma[7]
P=arma$arma[3]
Q=arma$arma[4]
p=arma$arma[1]
q=arma$arma[2]
# print(paste("Modelo (",p,d,q,")x(",P,D,Q,")"))
intercept=ifelse((d==0 & D==0),coef(arma)[["intercept"]],0) # Mean, not intercept

nlar=max(season*P,p)
par.ar=numeric(nlar)
nlma=max(season*Q,q)
par.ma=numeric(nlma)
nombres=names(arma$coef)

if (P>0) {nP=grep("^sar",nombres)
par.ar[season*(1:P)]=arma$coef[nP]
nombres=nombres[-nP]}

if (Q>0) { nQ=grep("^sma",nombres)
par.ma[season*(1:Q)]=arma$coef[nQ]
nombres=nombres[-nQ]}

if (p>0) par.ar[1:p]=arma$coef[grep("^ar",names(arma$coef))]
if (q>0) par.ma[1:q]=arma$coef[grep("^ma",names(arma$coef))]
n.start=max(nlar,nlma)
dseries=series
```

```
if (d>0) {xid=tail(series,d);dseries=diff(dseries,lag=1,differences=d)}
if (D>0) {xiD=tail(dseries,D*season);dseries=diff(dseries,lag=season,differences=D)}
start.series=tail(dseries,n.start)
start.resid=tail(arma$residuals,n.start)
x=matrix(NA,nrow=n.start+n.ahead,ncol=nrep)
for (j in 1:nrep){
x[,j]=c(start.series,rep(0,n.ahead))
innov=rand.gen(n.ahead,...)
if (std) a=c(start.resid,innov[1L:n.ahead]*sqrt(arma$sigma2)) else a=c(start.resid,
innov[1L:n.ahead])
for (i in 1:n.ahead){
x[n.start+i,j]=intercept+sum(c(1,par.ma)*a[n.start+i-(0:length(par.ma))])+
sum(c(0,par.ar)*(x[n.start+i-(0:length(par.ar)),j]-intercept))
#% if (p>0) x[n.start+i]=x[n.start+i]+sum(par.ar*x[n.start+i-(1:length(par.ar))])
}
}
if (n.start > 0)
x <- x[-(seq_len(n.start)),]
if (d>0 | D>0) {
x2=rbind(matrix(tail(series,d+D*season),ncol=nrep,nrow=d+D*season,byrow=FALSE),x)
uni=c(1,-1)
unideg=c(0,1)
p=1
pdeg=0

if (d>0) {uni=c(1,-1)
unideg=c(0,1)
for (l in 1:d) {p=as.vector(outer(p,uni));pdeg=as.vector(outer(pdeg,unideg,"+"))}
}
if (D>0) {uni=c(1,-1)
unideg=c(0,season)
for (l in 1:D) {p=as.vector(outer(p,uni));pdeg=as.vector(outer(pdeg,unideg,"+"))}
}
ss=aggregate(p,list(lag=pdeg),sum)[-1,]
for (j in 1:nrep){
for (i in (d+D*season+1):nrow(x2)){
x2[i,j]=x[i-(d+D*season),j]
for (k in 1:nrow(ss)){x2[i,j]=x2[i,j]-x2[i-ss$lag[k],j]*ss$x[k]}
}
}
x=x2[-(1:(d+D*season)),]
}
x=ts(x,start=tsp(series)[2]+deltat(series),frequency=frequency(series))
return(x)
}

#myboot=function(n,vec,prob=NULL){sample(vec,size=n,replace=TRUE,prob=prob)}
#Replace Residuals
#data(AirPassengers)
ap=serie
ap.res=arima(log(ap),order=c(5,0,0),fixed=c(NA,0,0,0,NA,NA))#,seasonal=list(
order=c(0,1,0),frequency=12)
```

```
xsim=arima.simforecast(ap.res,10,series=log(ap),nrep=500)
quan.ts=ts(t(apply(xsim,1,quantile,prob=seq(.1,.9,by=.1))),start=start(xsim),
frequency=frequency(xsim))


plot(cbind(log(ap),quan.ts),plot.type="single",lwd=2,
col=c(1,2,3,4,5,6,5,4,3,2),ylab="Tiempos de espera",
main="Cuantiles de Predicción Bootstrap")
legend("topleft",c("Serie","Cuantil .9", "Cuantil .8", "Cuantil .7", "Cuantil .6",
"Cuantil .5"), pch = 16,col=c(1,2,3,4,5,6))


plot(cbind(window(log(ap),200,268),quan.ts),plot.type="single",lwd=2,
col=c(1,2,3,4,5,6,5,4,3,2),ylab="Tiempos de espera",
main="Cuantiles de Predicción Bootstrap")
legend("bottomright",c("Serie","Cuantil .9", "Cuantil .8", "Cuantil .7", "Cuantil .6",
"Cuantil .5"), pch = 16,col=c(1,2,3,4,5,6))
```

B.2. Código completo simulación

```
# Tasa de llegada de los trabajos de la clase 1
lambda11=0.0023*60
# Tasa de llegada de los trabajos de la clase 2
lambda12=0.0018*60
# Tasa de servicio de los trabajos de la clase 1
mu11=1/(10060.3336/60)
# Tasa de servicio de los trabajos de la clase 2
mu12=1/(50603.2309/60)
# Número de servidores
n=16

##############################

# Vamos a generar B poissones para cada una de las clases. Número de trabajos
# que llegan.
B=1440
R=100
est1=numeric(R)
est2=est1
est3=est1
est4=est1
est5=est1
for (r in 1:R){
clase1=rpois(B,lambda11)
clase2=rpois(B,lambda12)
d1=clase1
d2=clase2
```

```
matriz=cbind(0,0,0,0,0,0)



for (i in 1:B){
if((clase1[i]+clase2[i]) == 0){
matriz=rbind(matriz,c(i,0,0,0,0,0))
}else{while(clase1[i]!=0){
a=c(i,1,rexp(1,mu11),0,0,0)
matriz=rbind(matriz,a)
clase1[i]=clase1[i]-1}
while(clase2[i]!=0){b=c(i,1,rexp(1,mu12),0,0,0)
matriz=rbind(matriz,b)
clase2[i]=clase2[i]-1}
}
}

colnames(matriz)=c("tiempo","num_trab","tservicio","estatus","tfinal",
"tespera")

matriz

# Acordarse de ignorar la primera fila de tiempo = 0.
servicio=numeric(n)
m=dim(matriz)[1]

# Acordarse de ignorar la primera fila de tiempo = 0
copia=matriz
matriz=copia
for (j in 1:m){
if(sum(as.numeric(matriz[1:j,2]==0))==0){
matriz[1:j,4]==2
}else{
for (i in 1:j){
if(matriz[i,3]==0){matriz[i,5]=0
}else{matriz[i,5]=matriz[i,3]+matriz[i,1]
}
}
}
}
# Ahora mismo se tiene la matriz sin añadirle el hecho de los tiempos de espera
# Se mete la ejecución
copia1=matriz
matriz=copia1
servicio=numeric(n)
for (j in 1:m){
for(a in 1:j){if(matriz[a,3]<1){matriz[a,3]=0
matriz[a,4]=2}}
#if(sum(as.numeric(matriz[1:j,2]==0))==0){
if(sum(as.numeric(matriz[1:j,3]==0))==0){
```

```
matriz[1:j,4]==2
}else{
for (i in 1:j){
if(matriz[i,2]!=0){
# hay trabajos que meter
if(sum(as.numeric(servicio>=1))!=n){
# hay sitio para meter el trabajo
b=matriz[i,2]
matriz[i,2]=0
matriz[i,4]=1
matriz[i,5]=matriz[i,5]+matriz[i,6]
for (s in 1:n){
if(servicio[s]<1 & b!=0){
servicio[s]=matriz[i,3]
b=b-1
}
}
}else{# no hay sitio y se pasa a espera
matriz[i,6]=matriz[i,6]+1
matriz[i,4]=0
}
}
}
}
# Al final de cada vuelta hay que sumar uno a los que están esperando y
# restar uno a los que se están ejecutando.
for(t in 1:j){
if(matriz[t,4]==1){matriz[t,3]=matriz[t,3]-1}
if(matriz[t,4]==0 & matriz[t,2]!=0){matriz[t,6]=matriz[t,6]+1}
}
# Se actualiza el tiempo de servicio
for (u in 1:n){if(servicio[u]<1){servicio[u]=0}else{servicio[u]=servicio[u]-1}}
}


##############

a=cbind(copia[,1:3],matriz[,4:6])
c=c(0,0,0,0,0,0)
m=dim(a)[1]
c1=0
c2=0
for (i in 1:m){
if(a[i,2]!=0 & a[i,4]!=0){c1=c1+1}
if(a[i,2]!=0 & a[i,4]==2){c2=c2+1}
if(a[i,5]!=0){c=rbind(c,a[i,])}
}

est1[r]=c1 # Número de trabajos que han empezado a ejecutarse
est2[r]=c2 # Número de trabajos que terminaron de ejecutarse
est3[r]=mean(c[,6]) # media del tiempo de espera
est4[r]=mean(c[,3]) # tiempo medio de servicio
```

```
est5[r]=mean(c[,3]+c[,1]+c[,6]) # tiempo que ha estado en el sistema
}


sum(as.numeric(est1>=70))/R
sum(as.numeric(est1>=80))/R
sum(as.numeric(est1>=90))/R
sum(as.numeric(est1>=100))/R

sum(as.numeric(est2>=10))/R
sum(as.numeric(est2>=20))/R
sum(as.numeric(est2>=30))/R
sum(as.numeric(est2>=40))/R
sum(as.numeric(est2>=60))/R
sum(as.numeric(est2>=80))/R


mean(est3)
mean(est4)
mean(est5)




#### c resumen

# Eliminando los instantes en los que no entran trabajos se tienen que el
# tiempo medio de espera es de

mean(c[,6]) # media del tiempo de espera
# tiempo medio de servicio
mean(c[,3])
# tiempo que ha estado en el sistema:
s=c[,3]+c[,1]+c[,6]
mean(s)

# número de trabajos que han entrado en el sistema
r=dim(c)[1]
# número de trabajos que se han terminado de ejecutar en el propio día
t=0
for (i in 1:r){if(c[i,4]!=0){t=t+1}}
t
# proporción de trabajos que se han ejecutado:
t/r



x=numeric(dim(matriz)[1])
l=length(x)
y=x
r=matriz[2,6]
for (i in 2:l){
```

```
r=matriz[i,6]
if ((i+r)<=l){y[i:(i+r)]=y[i:(i+r)]+1}else{y[i:l]=y[i:l]+1}
x[i]=matriz[i,1]
}


plot(x,y,xlab='Tiempo',ylab='Núm. trabajos en espera')
```

Bibliografía

[1] Dean J, Ghemawat S, (2008) MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107-113.

[2] Gross D, (2008) Fundamentals of queueing theory. John Wiley & Sons.

[3] Karau H, Konwinski A, Wendell P, Zaharia M (2015) Learning spark: lightning-fast big data analysis. O'Reilly Media, Inc.

[4] Koch I (2013) Analysis of multivariate and high-dimensional data. Cambridge University Press.

[5] Kung-Sik Chan and Brian Ripley (2012). TSA: Time Series Analysis. R package version 1.01. https://CRAN.R-project.org/package=TSA. Accedido 27 de Junio de 2017.

[6] Peña D. (2005) Analisis de Series Temporales. Alianza Editorial.

[7] Pfaff, B. (2008) Analysis of Integrated and Cointegrated Time Series with R. Second Edition. Springer, New York. ISBN 0-387-27960-1. http://www.pfaffikus.de. Accedido 27 de Junio de 2017.

[8] R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. [fecha de consulta 02/11/2016]. Disponible en Web: http://www.R-project.org/.

[9] Zaharia M, Chowdhury M, Franklin M J, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. HotCloud, 10(10-10):95.

[10] http://docs.python.org/2/library/struct.html

[11] http://spark.apache.org/docs/0.9.0/python-programming-guide.html

[12] http://www.cesga.es/en/infraestructuras/computacion/InfraBigDataEn

[13] http://www.matplotlib.org

[14] http://www.numpy.org

[15] http://www.pandas.pydata.org/pandas-docs/stable