UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Trabajo Fin de Master en Ingeniería y Ciencias de Datos

PREDICCIÓN DE CONSUMO DE RECURSOS PARA LA EJECUCIÓN DE PROCESOS

ANTONIO GARCÍA HERNÁNDEZ

Dirigido por: AGUSTÍN CARLOS CAMINERO HERRÁEZ
Codirigido por: JOSÉ MANUEL CUADRA TRONCOSO

Curso: 2021-2022. Convocatoria de septiembre

PREDICCIÓN DE CONSUMO DE RECURSOS PARA LA EJECUCIÓN DE PROCESOS

Trabajo Fin de Master en Ingeniería y Ciencias de Datos
Modalidad específica

Realizado por: ANTONIO GARCÍA HERNÁNDEZ

Dirigido por: AGUSTÍN CARLOS CAMINERO HERRÁEZ
Codirigido por: JOSÉ MANUEL CUADRA TRONCOSO

Fecha de lectura y defensa: 10 de octubre de 2022

1. Resumen

La creación y consumo de datos a través de internet ha experimentado un incremento en los últimos años que implica que cada vez sea más necesario disponer de aplicaciones basadas en tecnologías Big Data que puedan tratar con esa información y obtener un valor de esos datos. Estas tecnologías generalmente funcionan de modo distribuido sobre plataformas empresariales que pueden alcanzar en algunos casos miles de máquinas de procesamiento.

Dado que estas plataformas van a gestionar cada vez más volumen de datos, se hace necesario optimizar los recursos existentes en la infraestructura de modo que se continue dando servicio a la ejecución de esas aplicaciones de un modo más eficiente.

Una opción podría ser planificar la ejecución de aplicaciones conociendo cuando se van a producir los mayores consumos de recursos por parte de estas. De modo que se pudiese adelantar o retrasar algunos trabajos planificados para que sus picos de consumo no coincidan en tiempo y finalicen incorrectamente por falta de recursos en el sistema.

En este trabajo se analizará un conjunto de datos real extraído de una plataforma de procesamiento distribuido donde multitud de aplicaciones ejecutan sus tareas de modo paralelizado y concurrente. Posteriormente, los datos serán utilizados para crear modelos de aprendizaje automático mediante series temporales con la idea de predecir, para la próxima ejecución de una determinada aplicación, cuando se producirá su pico de consumo máximo y cuál será el valor de este.

2. Palabras clave

Predicción de consumo de recursos

Series temporales

Google Borg Cluster

Traces v3

Aprendizaje automático

Aprendizaje automático profundo

Jupyter Notebook

Python3

Scikit-Learn

Keras

TensorFlow

3. Abstract

Over the last years, creation and consumption of internet data has been increased. This makes necessary using Big Data technologies for managing these data and getting a value from them. Usually, these technologies work in distributed business environments that can amount to thousands of machines.

As these platforms are expected to manage an increasing data volume, finding new ways for optimizing the available resources are needed, so they can continue providing support to those applications that must be executed in distributed environments.

One possible solution could be scheduling the applications executions based on their peak resource consumptions. This way, several jobs could be delayed or forwarded so they cannot reach their consumption peaks at the same time and avoid evicting them due to the lack of available resources.

This work will analyze a real distributed environment dataset where lots of applications are executed in a parallel way. Next, the dataset will be used for building time series machine learning models that will be able to forecast when will be the consumption peak and what is its value for the next execution of a certain application.

4. Keywords

Resources consumption forecasting

Time series

Google Borg Cluster

Traces v3

Machine Learning

Deep Learning

Jupyter Notebook

Python3

Scikit-Learn

Keras

TensorFlow

Índice de contenido

Índice de figuras

Índice de tablas

1 Introducción, motivación y objetivos

1.1 Introducción y motivación

Durante los últimos años el incremento en la creación y consumo de datos a través de internet ha experimentado un rápido crecimiento. Sin ir más lejos, el último informe Global DataSphere publicado por la empresa IDC en mayo de 2022 estima que, en los próximos 5 años, se duplicará el volumen de datos actualmente existente [1]. Con un escenario como este, es obvio que seguirá surgiendo la necesidad de disponer de aplicaciones que puedan tratar con esa información y obtener un valor de esos datos.

Se trata de aplicaciones basadas en tecnologías Big Data que generalmente funcionan de modo distribuido sobre plataformas empresariales y que pueden alcanzar en algunos casos miles de máquinas de procesamiento. Dado que estas plataformas van a gestionar cada vez más volumen de datos, se hace necesario optimizar los recursos existentes en la infraestructura de modo que se continue dando servicio a la ejecución de esas aplicaciones de un modo más eficiente.

Una opción podría ser planificar la ejecución de aplicaciones conociendo cuando se van a producir los mayores consumos de recursos por parte de estas. De modo que se pudiese adelantar o retrasar algunos trabajos planificados para que sus picos de consumo no coincidan en tiempo y finalicen incorrectamente por falta de recursos en el sistema.

En este trabajo se analizará un conjunto de datos real extraído de una plataforma de procesamiento distribuido donde multitud de aplicaciones ejecutan sus tareas de modo paralelizado y concurrente. Posteriormente, los datos serán utilizados para crear modelos de aprendizaje automático mediante series temporales con la idea de predecir, para la próxima ejecución de una determinada aplicación, cuando se producirá su pico de consumo máximo y cuál será el valor de este.

1.2 Objetivos

El objetivo principal del trabajo es crear un modelo predictivo para el consumo de recursos y el tiempo de ejecución de procesos.

Para llevarlo a cabo, se creará un modelo de predicción de recursos para la ejecución de procesos a partir de un conjunto de datos público que contiene información relacionada con las ejecuciones de aplicaciones y consumos de recursos en un entorno de procesamiento de datos distribuido. De modo que, al recibir una nueva petición de ejecución de una aplicación, el modelo sea capaz de predecir tanto el momento de la ejecución donde se producirá el mayor consumo de recursos y el valor de este.

Para ello, se realizará un estudio detallado del conjunto de datos y se seleccionará la información más relevante con la idea de adaptarla a un nuevo conjunto de datos que pueda ser utilizado por parte de una serie de modelos de aprendizaje automático para la predicción de estos consumos. Los modelos serán creados y entrenados siguiendo el paradigma de las series temporales. Los resultados obtenidos serán finalmente analizados para verificar la precisión de los modelos creados.

1.3 Planificación del trabajo

El plazo para la ejecución del trabajo se ha fijado en 13 semanas, con una estimación total de 300 horas aproximadamente. El plazo estará distribuido en 4 fases principales y una transversal tal y como se representa en figura 1-1:

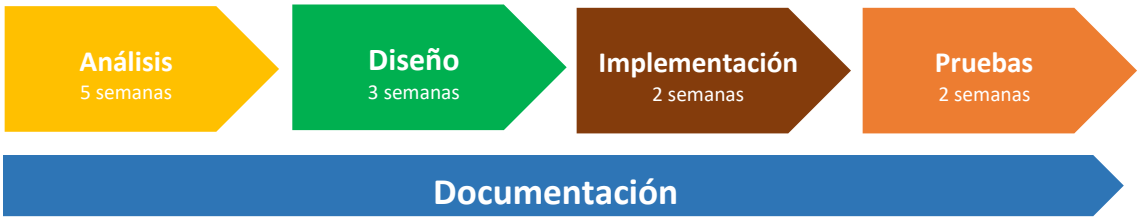

Figura 1-1. Detalle de las fases del trabajo

Las fases principales se han realizado durante las primeras 12 semanas compartiendo parte de su tiempo con la fase transversal de documentación. Aunque esta última es una tarea transversal, se prolongó de modo exclusivo una semana más

para revisión de documentación y finalización del trabajo. De este modo queda cubierto el plazo total comentado anteriormente.

La tabla 1-1 muestra la distribución semanal por cada fase. En la misma se observa que se han realizado dos ciclos completos de iteraciones por cada una de ellas. El primer ciclo corresponde a los trabajos realizados con el conjunto de datos en la nube, desde el análisis del mismo hasta su descarga a local. El segundo ciclo corresponde a los trabajos realizados con el conjunto de datos en local. Por otro lado, el estudio de las series temporales está repartido entre los dos ciclos de análisis.

Finalmente, añadir que en la fase de documentación se observa que el color de la tarea en las últimas semanas es más intenso, se ha hecho así para reflejar de algún modo que el tiempo compartido de esta tarea durante ese periodo de tiempo ha sido superior con respecto a semanas anteriores.

Semana	1	2	3	4	5	6	7	8	9	10	11	12	13
Análisis													
Diseño													
Implementación													
Pruebas													
Documentación													

Tabla 1-1. Distribución semanal de las fases del trabajo

1.4 Coste económico

La tabla 1-2 muestra el coste económico del trabajo desglosado por fases, en ella se detalla la distribución de horas que tiene cada una así como su coste parcial. El total de horas se distribuyen a lo largo de 13 semanas tal y como se indicó en el apartado de planificación. Los importes incluyen las cotizaciones a la Seguridad Social e IRPF.

Fase	Horas	Precio €/hora	Total
Análisis	115	37,5	4.312,50 €
Diseño	70	35	2.450,00 €
Implementación	45	31,25	1.406,25 €
Pruebas	45	27,5	1.237,50 €
Documentación	25	21,25	531,25 €
Total	**300**	**152,5**	**9.937,50 €**

Tabla 1-2. Coste económico de las fases del trabajo

1.5 Organización de la memoria

La memoria está organizada en 6 capítulos principales que hacen referencia a todas las fases del trabajo y que han sido expuestas en el apartado anterior. En ellas se detalla todas las acciones llevadas a cabo para la consecución de las mismas. Junto a estos capítulos se incluye uno introductorio que trata brevemente de las tecnologías a utilizar durante el trabajo, un capítulo de revisión de trabajos similares ya realizados y un capítulo final dirigido a las conclusiones y a las líneas de trabajo que surgen a partir del mismo. En total 9 capítulos que se comentan con más detalle a continuación.

En el capítulo dedicado al **estado del arte**, se comentarán algunos trabajos realizados con la predicción de consumo de recursos existentes en la literatura. Se revisarán sus resultados y se comentarán cual será el enfoque de este trabajo y las particularidades que lo diferencian de los trabajos revisados.

En el capítulo dedicado a las **tecnologías utilizadas**, se revisarán aquellas tecnologías que serán empleadas durante el trabajo para el estudio y transformación del conjunto de datos, la creación y entrenamiento de los modelos de aprendizaje automático y los elementos visuales que servirán de apoyo a la revisión de los resultados obtenidos.

En el capítulo dedicado a **el conjunto de datos Traces,** se describirán los pasos que tendrá lugar en la nube. Se analizará este conjunto de datos, describiendo su contenido así como sus tablas y campos más relevantes. También se comentará el diseño e implementación de la solución que permitirá reducir su volumen de información inicial para continuar con su tratamiento y estudio en un entorno local.

En el capítulo dedicado a la fase de **análisis**, se analizará el contenido del conjunto de datos a nivel local así como algunos de sus valores estadísticos. Se establecerán los criterios para seleccionar los registros más relevantes a nivel de aplicación y se detallará la relación de atributos que han de incluirse en el conjunto de datos para enriquecer a este. Por otro lado, se fijará la relación de modelos de aprendizaje automático a evaluar así como sus requisitos para el entrenamiento y pruebas.

En el capítulo dedicado a la fase de **diseño**, se establecerá el diseño de los componentes que permitirán filtrar, transformar, importar, exportar y dividir el conjunto de datos inicial en otros subconjuntos que puedan ser utilizados por los modelos de aprendizaje. Se definirán las estructuras comunes a todos los modelos así como las específicas de cada uno de ellos. También se definirán los componentes que realizarán la comparativa de los resultados obtenidos tanto cualitativa como cuantitativamente.

En el capítulo dedicado a la fase de **implementación**, se detallará la implementación de todos componentes, estructuras y modelos especificados en capítulos anteriores. También se mostrarán los bloques de código más relevantes tanto comunes como específicos de cada modelo.

En el capítulo dedicado a la fase de **pruebas**, se revisarán los resultados de las pruebas obtenidas en el entrenamiento y predicción de los modelos. Se mostrarán las combinaciones de hiperparámetros utilizadas y comentarán las diferencias observadas durante la fase de predicciones a partir del conjunto de datos de validación y pruebas.

En el capítulo dedicado a la **evaluación de los resultados obtenidos**, se comprobarán los resultados obtenidos por los modelos de aprendizaje en sus predicciones sobre el conjunto de pruebas y se seleccionará al mejor modelo candidato para predecir el consumo de recursos sobre el conjunto de datos seleccionado.

En el capítulo dedicado a las **conclusiones y líneas futuras**, se revisarán las etapas seguidas a lo largo del trabajo para la consecución del objetivo planteado. Se comentarán algunas conclusiones y dificultades encontradas se describirán posibles líneas futuras de trabajo que darían continuidad a este trabajo.

1.6 Resumen

En este capítulo introductorio se han planteado aquellas motivaciones que llevan a la ejecución de este trabajo, como el incremento en el volumen de datos creados y la necesidad de disponer de mecanismos que permitan un uso más eficiente de los recursos disponibles. Lo que lleva a la propuesta planteada por este trabajo: crear un modelo predictivo para el consumo de recursos y el tiempo de ejecución de procesos.

La planificación del trabajo se divide en 4 fases principales de análisis, diseño, implementación y pruebas junto a una fase transversal de documentación cuyo plazo de ejecución está estimado en 13 semanas.

Finalmente, se ha realizado una pequeña introducción acerca de cómo se encuentra estructurada esta memoria, que está compuesta por 6 capítulos que tratan sobre todas las fases anteriormente comentadas y un capítulo adicional que trata sobre las conclusiones y líneas futuras del trabajo. Para cada capítulo se ha hecho una breve introducción acerca de su contenido.

2 Estado del arte

En este capítulo, se revisarán algunos trabajos realizados y relacionados con el estudio en curso. El primer apartado incluye aquellos trabajos orientados a la predicción de consumo de recursos y el segundo apartado los que han explorado el conjunto de datos que se utilizará a lo largo de este trabajo. El último apartado recoge algunas de las particularidades que diferencia el presente estudio de los trabajos revisados en los apartados anteriores.

2.1 Trabajos relacionados con la predicción de recursos

La predicción de consumo de recursos es un campo de interés que ha sido estudiado en números trabajos, utilizándose diferentes enfoques para abordar el problema. En [2], los autores utilizan modelo de regresión lineal predecir la carga de trabajo en un entorno simulado de servicios cloud donde dos aplicaciones generan esta carga. Aunque el estudio está más centrado en el algoritmo de auto escalado, se basan en el modelo de regresión lineal para hacer la predicción de demanda de recursos.

Por otro lado, en [3] los autores sí utilizan un conjunto de datos real para realizar una comparativa de predicción de consumo de CPU mediante tres métodos de predicción diferentes: ARIMA, Holt Winters y LSTM. Resultando ser este último, basado en aprendizaje profundo, el que mejor se predicciones realizaba. Sin embargo, el conjunto de datos se limitaba a una sola máquina, donde dos clientes realizaban peticiones de consumos cada cierto tiempo.

El modelo de predicción de LSTM también aparece reflejado en [4], donde se realiza un estudio de predicción de consumo de CPU sobre un conjunto de datos real que contiene detalles de consumos pertenecientes a una red 50 clústeres. Se realizaban dos tipos de predicciones, una a corto plazo donde se predecía el consumo para cada clúster para la próxima hora, y otro a largo plazo donde se predecía el consumo dentro de 3 días. Las predicciones se realizaron utilizando dos modelos de predicción, uno estadístico basado en el modelo SARIMA y otro basado en el modelo en una red neuronal LSTM. Donde cada modelo se entrenó de modo independiente 50 veces, una

por cada clúster reportado. Los resultados obtenidos fueron que las predicciones del modelo LSTM era más precisas a corto plazo y las del modelo SARIMA las mejores a largo plazo.

De la revisión de los trabajos anteriores se desprende que los modelos de aprendizaje automático son válidos para la predicción de recursos utilizando series temporales y que en algunos escenarios ofrecen mejores resultados que los modelos estadísticos más clásicos como los basados en ARIMA.

Por otro lado, estos trabajos centran la predicción de consumo en las máquinas o clúster de máquinas que contienen los recursos, normalmente es un grupo cerrado donde el modelo se entrena por separado para cada componente de la misma. Además, en algunos casos los conjuntos de datos son simulados o limitados a un único entorno, sin quedar claro si los resultados obtenidos son extrapolables a sistemas de procesamiento de datos distribuidos.

2.2 Trabajos relacionados con el conjunto de datos Traces

El conjunto de datos que se utilizará a lo largo de este trabajo [5] ha sido estudiado desde varias perspectivas desde que fue publicado en 2019. En [6] por ejemplo, los autores realizan una comparativa del conjunto de datos con respecto a otro conjunto que ya se obtuvo del mismo sistema en 2011 [7]. El artículo revisa la nueva información incorporada en la nueva versión y compara el incremento que ha experimentado la demanda de recursos a lo largo de los años. Además, analizan como se ha redistribuido los porcentajes de tipos de procesos según su prioridad, la heterogeneidad de estos con respecto al consumo de recursos y las mejoras observadas en cuanto a la asignación de los mismos.

Por otro lado, en [8] se exploran los posibles patrones que subyacen en la planificación de trabajos en entornos de procesamiento distribuido. Para ello, el autor aplica varias técnicas de aprendizaje no supervisado para obtener subconjuntos de elementos con características similares. Los resultados confirmaron que existe una agrupación natural de procesos en función de la prioridad que tienen asignada.

Además, utilizó métodos de regresión para determinar la necesidad de aislar las cargas de trabajo para una mejor predicción de recursos y métodos de ingeniería inversa para identificar los puntos clave del funcionamiento de *Autopilot*, el sistema interno utilizado para el escalado automático de recursos. Los resultados confirmaron que el consumo de recursos es heterogéneo entre procesos, donde un gran número de estos consume un pequeño porcentaje de recursos y unos pocos acaparan la mayoría de los mismos. Respecto a *Autopilot*, concluyeron que las asignaciones de recursos que realiza son más eficientes que las proporcionadas por parte de los usuarios.

2.3 Aportaciones del trabajo en curso

A diferencia de los trabajos comentados en el apartado 2.1, en este trabajo se pretende cambiar el enfoque de la predicción, centrándola en las aplicaciones que realizan el consumo y no en la máquina que lo proporciona. La predicción no será cerrada a un conjunto determinado de aplicaciones, sino que abarcará a todas aquellas aplicaciones que se hayan ejecutado un número suficiente de veces como para poder predecir su próximo consumo mediante una serie temporal. Además, las predicciones las realizará un único modelo de aprendizaje automático, que se espera tenga el conocimiento suficiente para discriminar entre aplicaciones. Todo el estudio se realizará utilizando un conjunto de datos extraído de un sistema de procesamiento distribuido real.

Respecto a los trabajos comentados en el apartado 2.2 realizados con el conjunto *Traces*, esto se centran en su estructura interna y en el comportamiento en general de Borg, pero no se utilizan para realizar predicciones de consumos. En este trabajo la utilidad que se dará al conjunto de datos será precisamente esta, utilizar su información para realizar predicciones de consumos de recursos en entornos de procesamiento distribuido.

2.4 Resumen

En este capítulo se han comentado algunos trabajos realizados con la predicción de consumo de recursos existentes en la literatura. Se han destacado sus resultados y revisado algunas de las limitaciones que presentaban. También se han revisado algunos trabajos relacionados con el conjunto de datos que se utilizará a lo largo de este trabajo y que estaban centrados en la estructura y comportamiento a nivel general. Por otro lado, también se ha comentado cual será el enfoque de este trabajo con respecto al conjunto de datos y las particularidades que lo diferencian de los trabajos revisados.

3 Tecnologías utilizadas

A lo largo de las distintas etapas del trabajo se irán utilizando un conjunto de plataformas y herramientas tecnológicas que permitirán ir modelando el conjunto de datos y la relación de modelos de aprendizaje automático que se pretende implementar. A continuación se comentarán las más relevantes.

3.1 BigQuery

BigQuery [9] es una plataforma de almacenamiento de datos en la nube propiedad de Google. Presenta como ventajas que es un entorno totalmente administrado que permite almacenar y extraer grandes volúmenes de datos del orden de petabytes. La información está almacenada en tablas y puede ser accedida a través de consultas en lenguaje SQL. El conjunto de datos a analizar en este trabajo se encuentra alojado en esta plataforma, así que su uso simplificará el tratamiento inicial que se realice de este.

Por otro lado, el uso de la plataforma sin costes implica algunas limitaciones que se tendrán en cuenta para no incurrir en gastos adicionales. Por ejemplo, el primer terabyte de movimiento de datos al mes está libre de coste y la creación y almacenamiento de datos en tablas también es gratuito por debajo de 10GB. Además, como también permite la exportación de datos a local, cabe la posibilidad de extraer la parte más importante del conjunto de datos para seguir trabajando en este entorno.

3.2 Jupyter Notebook

Es una aplicación Web que permite la creación y uso de documentos desde una interfaz gráfica sencilla de utilizar. Su código es *open source* y está desarrollada por *Project Jupyter* [10], una organización sin ánimo de lucro que promueve el uso interactivo de la computación científica y la ciencia de datos para todos los lenguajes de programación.

Con los documentos de Jupyter es posible interactuar desde su interfaz web con el sistema operativo donde Jupyter Notebook se encuentra instalado. Permite programar aplicaciones y ejecutarlas desde la misma web, donde también se muestran los resultados de la ejecución.

Los documentos pueden guardarse en local y ser compartidos para ser ejecutados en otros sistemas. Además, admite multitud de lenguajes de programación distintos y por defecto viene preparado para Python.

Las prestaciones que Jupyter ofrece son perfectas para poder realizar en local tanto el estudio del conjunto de datos como la creación de los modelos de aprendizaje automático.

3.3 Python v3

Se trata de un lenguaje de programación con licencia *open source* administrado por la *Python Software Foundation*[11]. Es ampliamente utilizado en análisis de datos y destaca por la legibilidad de su código y por la variedad de librerías que contiene. Algunas de las que serán utilizadas en este trabajo serán *numpy* y *pandas* para gestión de las estructuras del conjunto de datos y su contenido, *matplotlib* para la creación de figuras y elementos gráficos, *scikit-Learn* para la creación de modelos de aprendizaje automático y *keras* y *tensorFlow* para la creación de modelos de aprendizaje automático basados en redes neuronales.

3.4 Resumen

En este capítulo se ha realizado un breve repaso de aquellas tecnologías que serán empleadas durante el trabajo para el estudio y transformación del conjunto de datos, la creación y entrenamiento de los modelos de aprendizaje automático y los elementos visuales que servirán de apoyo a la revisión de los resultados obtenidos.

4 El conjunto de datos Traces

Borg es el sistema interno de gestión de clúster creado y utilizado por Google [12] para la administración de los recursos y procesos que se ejecutan en las miles de máquinas que lo componen. Su arquitectura basada en *Borg Cells* permite la planificación y ejecución distribuida de aplicaciones así como alta disponibilidad y tolerancia a fallos. La figura 4-1 muestra la arquitectura simplificada de una de las celdas de Borg, que puede contener de media unas 10.000 máquinas.

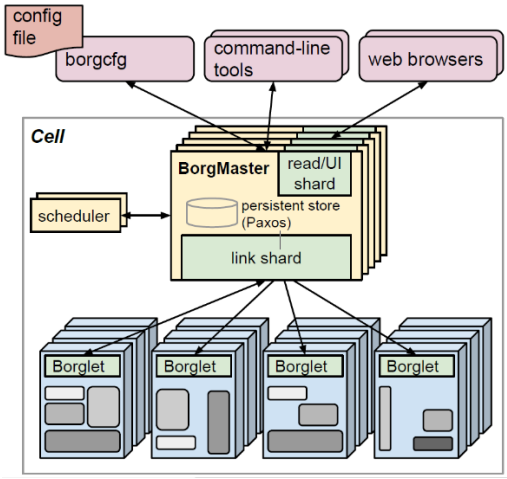

Figura 4-2. Arquitectura simplificada de una celda Borg

Desde que un proceso se planifica para su ejecución hasta que finaliza, este pasa por una serie de estados donde su demanda y consumo de recursos es variable.

Durante el mes de mayo de 2019, esta interacción entre máquinas y procesos fue monitorizada sobre 8 celdas Borg. Donde toda la actividad quedó registrada dando lugar al conjunto de datos *Traces v3*. Este conjunto será el origen de estudio de este trabajo y su especificación se encuentra recogida en el documento *Google cluster-usage traces v3* [5].

La figura 4-2 muestra la subestructura de objetos y eventos que son de interés para el estudio en curso. Corresponden a una simplificación del modelo entidad-relación que, de modo completo, se incluye en dicho documento:

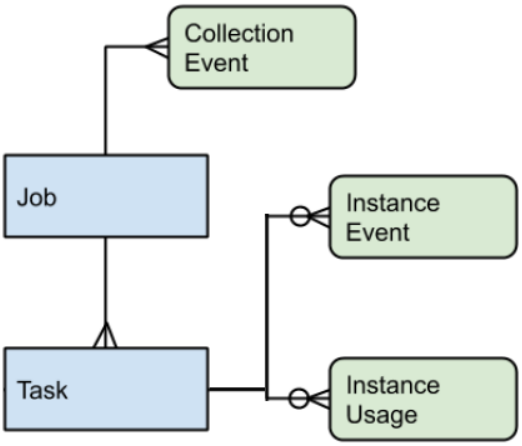

Figura 4-2. Modelo entidad-relación de Borg simplificado

Los recuadros azules son los objetos que se planifican y ejecutan dentro de un clúster de Borg y los recuadros verdes corresponden a los eventos que dichos objetos generan durante su ejecución.

Conceptualmente, el documento engloba todo lo anterior dentro de una entidad genérica llamada *thing*. A partir de esta surgen las subcategorías mostradas en la figura 4-3:

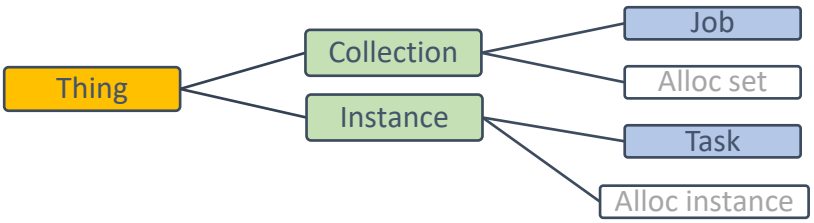

Figura 4-3. Jerarquía de objetos utilizados en Borg

Donde:

- **Thing**: Es cualquier objeto o evento que genera información que puede almacenarse en las tablas de *traces*.
- **Collection**: Hace referencia a los procesos (Job) y reservas de recursos (Alloc Set) que se planifican y ejecutan dentro de Borg.
- **Instance**: Hace referencia a las tareas asociadas a Jobs (Task) e Instancias de recursos (Alloc Instance) que se planifican y ejecutan dentro de Borg.

Con lo anterior, el estudio debería centrarse en las ejecuciones de Jobs y sus tareas asociadas, que son las que van a generar los consumos de recursos. La parte relacionada con la reserva de recursos (Alloc Set y Alloc Instance) están más enfocadas a contenedores de recursos, que tendrán asignados las tareas a ejecutar. Son asignaciones que no varían a lo largo de la ejecución de los procesos, de ahí que sea más interesante centrar el estudio en los objetos que realizarán dicho consumo.

A modo de resumen, una colección corresponde a la ejecución de una aplicación que puede repetirse a lo largo del tiempo. Cada ejecución tendrá un identificador único que servirá para trazar dicha ejecución. Una instancia corresponde a la tarea o tareas asociadas a una colección, son las monitorizadas en el documento principal a intervalos irregulares de tiempo y las que informan del consumo de recursos. Pueden encontrarse más detalles acerca de la ejecución de una aplicación en el Anexo A de este documento.

4.1 Análisis del conjunto de datos en BigQuery

El contenido de *Traces* se encuentra accesible por dos vías. Por un lado es descargable desde *Google Cloud* [13] en formato JSON. Donde un conjunto de ficheros almacenan toda la información. Por otro lado, también se encuentra almacenado en un conjunto de tablas en BigQuery bajo el trabajo: *Google Cluster Data (id= google.com:google-cluster-data)*. Donde la información de cada una de las 8 celdas Borg forman un conjunto de datos independiente.

Debido a que en BigQuery la información se encuentra almacenada en tablas, debidamente indexada y que el acceso a su contenido es más simple a través de consultas SQL, se elige este segundo formato como vía de acceso a los datos. Para el estudio en curso y sin pérdida de generalidad, el acceso se realizará solo al contenido relacionado con la celda Borg *clusterdata_2019_a,* ya que el contenido entre celdas es similar y los resultados obtenidos en una de ellas podrían ser extrapolables al resto. Además, acceder solo a una partición de los datos reduce los costes económicos que genera el acceso y manipulación de datos en BigQuery. Hay 3 tablas que son las que contienen la información de colecciones e instancias que es conveniente revisar. Los campos clave de cada tabla se muestran en cursiva.

4.1.1 CollectionEvents

Esta tabla contiene la información relacionada con los Jobs que se planifican en Borg. Los campos más interesantes son los siguientes:

- ***time***: Timestamp en el que se realiza el evento.
- **type**: Tipo de evento registrado (encolado, en ejecución, finalizado…)
- ***collection_id***: Identificador único para una determinada ejecución de una aplicación.
- **collection_type**: Tipo de colección, para el estudio en curso interesa 0 (job).
- **priority**: Prioridad asignada a la ejecución, a mayor valor mayor prioridad.
- **collection_name**: Nombre ofuscado de la aplicación a ejecutar, puede variar entre lanzamientos.
- **collection_logical_name**: Nombre ofuscado de la aplicación a ejecutar, parecido al anterior pero en este caso no varía entre lanzamientos.

4.1.2 InstanceEvents

Esta tabla contiene la información relacionada con los tareas que se planifican en Borg como consecuencia de la planificación del Job al que están asociadas. Los campos más interesantes son los siguientes:

- ***time***: Timestamp en el que se realiza el evento.
- **type**: Tipo de evento registrado (encolado, en ejecución, finalizado…)
- ***collection_id***: Identificación de la colección (Job) al que pertenecen.
- **priority**: Prioridad asignada a la ejecución, a mayor valor mayor prioridad.
- ***instance_index***: Posición de la tarea dentro del Job. Sería como un id de tarea interno.
- **resource_request**: CPU y memoria solicitada por la tarea.

4.1.3 InstanceUsage

Esta tabla contiene información acerca de los consumos que se van generando durante la ejecución de las tareas. La información corresponde a ventanas de tiempo dentro de la ejecución global de la tarea, donde el dato que se registra es un agregado de todo el periodo de monitorización observado dentro de una determinada ventana y que no tiene por qué ser el mismo en todos los casos. Los campos más interesantes son los siguientes:

- ***start_time***: Hora de comienzo de la ventana de monitorización.
- ***end_time***: Hora de finalización de la ventana de monitorización.
- **collection_id**: Identificador del Job al que pertenece el evento registrado.
- ***instance_index***: Identificador de la tarea asociada al evento registrado.
- **collection_type**: Tipo de colección, para el estudio en curso interesa 0 (job).
- **average_usage**: Valores medios de CPU y memoria observados durante la ejecución.
- **maximum_usage**: Valores máximos de CPU y memoria observados durante la ejecución.
- **random_sampled_usage**: Valor de CPU seleccionado aleatoriamente dentro de la ventana de monitorización.
- **assigned_memory**: Límite de memoria asignado para la ejecución.
- **cpu_usage_distribution**: 11 valores de grano grueso correspondientes al consumo de CPU según los percentiles 0, 10, 20,…,100. El valor se interpreta como el valor de CPU consumido por los procesos según el percentil N.
- **tail_cpu_usage_distribution**: 9 valores de grano fino correspondientes al consumo de CPU según los percentiles 91, 92, …, 99. El valor se interpreta como el valor de CPU consumido por los procesos según el percentil N.

Hay que comentar que Google utiliza una métrica propia para informar el valor de CPU consumido llamado GCU (*Google Compute Unit*), y que equivale al trabajo de cómputo realizado por el núcleo de una máquina base nominal. A su vez, este valor está normalizado conforme al máximo valor de GCU informado por alguna de las máquinas que componen el clúster de Borg. Esta nueva unidad se llama NCU (*Normalized Compute Units*) y es la que se informa en *Traces*, se encuentra en el rango de valores [0, 1].

Respecto a los campos de tiempo *time*, *start_time* y *end_time*, su valor corresponde al número de nanosegundos transcurridos desde que comenzó la captura de actividad en Traces. Por tanto, cualquier valor de tiempo será relativo a este inicio.

Tras analizar los valores recogidos en las tablas, el estudio sobre el consumo de CPU podría realizarse sobre los campos *maximum_usage*, *random_sampled_usage* o *cpu_usage_distribution*. En los tres casos el valor que se aporta es una estimación sobre la ventana observada. Los valores de máximo usado y percentil 100 normalmente coinciden. Para el estudio en curso se utilizará el valor máximo por ser el menos restrictivo, pero podría adaptarse al valor de los otros campos.

Por otro lado, sería necesario filtrar y unificar la información contenida en esas 3 tablas en un nuevo conjunto de datos. De modo que solo se dispongan de ejecuciones que hayan finalizado correctamente y donde una sola fila del conjunto contuviese la información relativa a la instancia que se ejecuta, el momento en el tiempo en el que la realiza, la aplicación a la que pertenece y los consumos máximos que se han producido a lo largo de su ejecución. De este modo, el volumen de datos a manipular se reduciría considerablemente y cabría la posibilidad de trabajar con el conjunto de datos en local.

4.2 Diseño de la exportación de datos

La solución debe contemplar un modo sencillo y optimizado de descargar los datos que se encuentran en BigQuery al entorno local. Como ya se comentó anteriormente, lo mejor es reunir todo el contenido de interés en una sola tabla que será la que finalmente se descargue a local.

Se utilizarán los campos claves de las 3 tablas comentadas para relacionar el contenido de las mismas. Se realizará mediante consultas SQL que irán filtrando la información de interés para el estudio y volcándola a tablas intermedias. Esto permitirá ir reduciendo gradualmente el volumen de datos a consultar, facilitar la revisión de la integridad de la información filtrada y simplificar la creación de nuevas tablas intermedias. La tabla 4-1 muestra las claves primarias usadas para relacionar el contenido.

CollectionEvents	InstanceEvents	InstanceUsage
collection_id	collection_id	collection_id
	instance_index	instance_index

Tabla 4-1. Relación de claves primarias a tener en cuenta

Donde se utilizará el tipo de relación *inner join* de modo que solo se tengan en cuenta eventos que estén informados íntegramente en todas las tablas. Además, a la hora de filtrar datos hay que conservar los registros relacionados con la ejecución de aplicaciones (*jobs y tasks, collection_type=0*). Por otro lado, solo se tendrán en cuenta ejecuciones que hayan finalizado correctamente (*type=6, FINISH*).

Una vez cruzado los datos de todas las tablas, los campos a seleccionar de cada una de ellas para la tabla final serán los indicados en la tabla 4-2.

CollectionEvents	InstanceEvents	InstanceUsage
collection_logical_name	collection_id	start_time
	instance_index	end_time
		maximum_usage.cpu

Tabla 4-2. Relación de campos a seleccionar para la tabla final

Finalmente, la información relacionada con esos campos será volcada a una tabla final, cuyo nombre será *aplication_usage_action_node_a_reduced* y que estará compuesta de los siguientes campos:

- **collection_logical_name**: Nombre ofuscado de la aplicación a ejecutar, no varía entre lanzamientos.

- **collection_id**: Identificador único para una determinada ejecución.
- **instance_index**: Posición de la tarea dentro del Job. Es un id de tarea interno.
- **start_time**: Hora de comienzo de la ventana de monitorización.
- **end_time**: Hora de finalización de la ventana de monitorización.
- **cpus**: Valor máximo de CPU observado durante la ejecución.

Una vez se tenga la tabla disponible en BigQuery, se creará el mecanismo de descarga a local. Deberá realizarse desde un documento de Jupyter Notebooks, a través de una API de conexión. Quedando el contenido de la tabla almacenado dentro de un objeto *DataFrame* de *Pandas* [14].

4.3 Implementación de la exportación de datos

La implementación del diseño comentado anteriormente se realiza directamente desde la consola de BigQuery. La dinámica seguida ha sido ir lanzado consultas SQL sobre el conjunto de datos e ir almacenando resultados intermedios, no solo para obtener la tabla final sino también para comprender cómo se encuentran distribuidos a nivel global los datos. La figura 4-4 muestra el detalle de las tablas intermedias usadas en GCP.

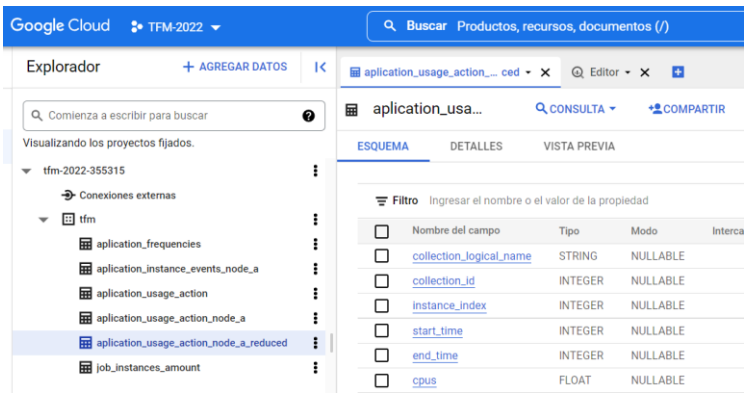

Figura 4-4. Detalle de las tablas intermedias usadas en GCP

Algunas de las tablas intermedias creadas son las siguientes:

- **Tabla *job_instances_amount***: Muestra el total de instancias por aplicación, solo saca lo relacionado con Jobs. Contabiliza el número de tareas (tasks) que una determinada aplicación ha tenido en una determinada ejecución(collection_id).

- **Tabla *aplication_frequencies***: Frecuencia de ejecuciones por aplicación. Necesario para identificar la frecuencia con la que se ejecutan las aplicaciones.

- **Tabla *aplication_usage_action_node_a***: Contiene la información de las 3 tablas principales de Traces relacionada a través de sus claves primarias. Permite disponer de toda la información que es necesaria para el estudio sin tener que acceder al conjunto original de datos. Cada consulta movía 1.4TB de información. Con esta tabla el movimiento se redujo a 6.24 GB para obtener los mismos resultados.

- **Tabla *aplication_usage_action_node_a_reduced***: Se crea a partir de la anterior y contiene únicamente los 6 campos que conformarán el conjunto final de datos. De este modo la descarga a local se limita a 1.62 GB sin pérdida de información.

El fichero *extraccion_desde_BigQuery\Consultas_SQL_utilizadas.sql* incluido en la documentación contiene el detalle de todas las consultas SQL utilizadas para la creación de estas tablas.

Respecto al mecanismo de descarga a local, se realiza mediante un documento de Jupyter Notebooks utilizando código Python v3. Previamente a la conexión hay que crear un fichero *token* de credenciales desde la consola de BigQuery que es el que permite el acceso desde una conexión externa.

La conexión desde local se realiza utilizando la API de BigQuery [15], utilizando un objeto de la clase *storage* perteneciente a esta biblioteca y el token comentado anteriormente. Una vez abierta la conexión, se utiliza un objeto de la clase *Client* que

es al que se le proporcionan por parámetro el detalle de la tabla a descargar. La figura 4-5 muestra el detalle de la implentación.

```
bqclient = bigquery.Client()

# Download a table.
table = bigquery.TableReference.from_string(
    "tfm-2022-355315.tfm.aplication_usage_action_node_a_reduced"
)
rows = bqclient.list_rows(
    table,
    selected_fields=[
        bigquery.SchemaField("collection_logical_name","STRING"),
        bigquery.SchemaField("collection_id","INTEGER"),
        bigquery.SchemaField("instance_index","INTEGER"),
        bigquery.SchemaField("start_time","INTEGER"),
        bigquery.SchemaField("end_time","INTEGER"),
        bigquery.SchemaField("cpus","FLOAT"),
    ],
)
df_usage_action_node = rows.to_dataframe(
    create_bqstorage_client=True,
)
```

Figura 4-5. Detalle de la implementación para la descarga de la tabla final

Finalmente, el contenido descargado puede cargarse en un objeto *pandas.DataFrame* que será el encargado de guardar el contenido de la tabla en un fichero con formato csv. De este modo, el contenido del conjunto de datos que se encontraba en la nube queda almacenado en local y disponible para los procesos de la siguiente fase.

El fichero *extraccion_desde_BigQuery\extraccion_relacion_usage_action_node_a_reduced.ipynb* incluido en la documentación contiene el detalle de todas las acciones realizadas para acceder al conjunto de datos y almacenarlos en local.

4.4 Resumen

En este capítulo se han descrito los pasos realizados en la primera fase del trabajo, que ha tenido lugar en la nube. Se ha analizado el conjunto de datos *Traces*, origen del conjunto de datos cuya información será utilizada en el estudio en curso. Se ha descrito su contenido así como sus tablas y campos más relevantes. Se ha diseñado e implementado una solución que ha permitido reducir el volumen de información inicial que lo compone a una cantidad que permite continuar su tratamiento y estudio en un entorno local. El siguiente capitulo da comienzo a la segunda fase del trabajo, donde se continuará trabajando con el conjunto de datos ya en dicho entorno local.

5 Análisis

El capítulo anterior realizó un tratamiento sobre el conjunto de datos en la nube que ahora posibilita su carga en un entorno local, permitiendo profundizar más acerca del contenido del mismo. El análisis que se describirá en este capítulo está centrado en el contenido de este conjunto de datos en local y en las modificaciones que deberían aplicarse para poder entrenar los modelos y realizar predicciones a partir de los mismos.

5.1 Estudio estadístico del conjunto de datos

El análisis comienza con la carga de datos en el entorno local, a partir del fichero *csv* resultado del capítulo anterior. Un objeto *pandas.DataFrame* almacena su contenido y se utilizarán sus algunas de sus funciones para revisar el conjunto de datos, que se compone de 20.219.223 registros. La figura 5-1 muestra un detalle del contenido.

	collection_logical_name	collection_id	instance_index	start_time	end_time	cpus
20219219	oSsTDVhDJAZDf4brFtPHYJo22CPFBOycY6p7FLALXCA=	382560900034	0	936826.0	936827.0	0.000005
20219220	oSsTDVhDJAZDf4brFtPHYJo22CPFBOycY6p7FLALXCA=	382560900038	0	936789.0	936807.0	0.007004
20219221	oSsTDVhDJAZDf4brFtPHYJo22CPFBOycY6p7FLALXCA=	382560900038	0	936807.0	936809.0	0.000240
20219222	oSsTDVhDJAZDf4brFtPHYJo22CPFBOycY6p7FLALXCA=	382560900038	0	936809.0	936810.0	0.000000
20219223	oSsTDVhDJAZDf4brFtPHYJo22CPFBOycY6p7FLALXCA=	382560918545	0	936801.0	936802.0	0.000000

Figura 5-1. Detalle del contenido del dataset descargado a local

Dado que el tamaño del conjunto de datos aún sigue siendo elevado para su tratamiento en local, se seleccionará un subconjunto de eventos que sean representativos del original y a su vez facilite el estudio del mismo. La idea sería seleccionar un conjunto de aplicaciones que tenga más de una tarea de ejecución en paralelo y que se haya ejecutado un número suficiente de veces. La figura 5-2 muestra algunos de los valores estadísticos del conjunto revisados.

	count	mean	std	min	50%	70%	95%	max
count	30459.0	42.641781	1548.669504	1.0	1.0	2.0	46.0	145249.0

Figura 5-2. Valores estadísticos del conjunto de datos inicial

A partir de los resultados anteriores, se tienen en total 30.459 aplicaciones con una media de 42 ejecuciones por aplicación. La elevada desviación estándar hace que el valor medio no sea muy representativo, ya que hay mucha variabilidad en el número de ejecuciones de una aplicación.

Observando los percentiles, prácticamente el 70% de las aplicaciones solo se han ejecutado una vez, y no es hasta el percentil 95 cuando se supera el valor medio de referencia. Es decir, solo aproximadamente el 5% de las aplicaciones se ejecutan más veces que ese valor medio.

Por otro lado, se realiza un agrupado de datos para estudiar el paralelizado de tareas por aplicación. Para cada una se obtiene el valor máximo de instancia (*instance_index*) que ha llegado a ejecutar en paralelo, de modo que pueda conocerse el paralelizado que cada una aplica. A partir de estos valores se obtienen valores estadísticos mostrados en la figura 5-3:

	count	mean	std	min	50%	90%	93%	94%	95%	96%	97%	98%	99%	max
instance_index	30459.0	9.578253	112.553538	0.0	0.0	0.0	1.0	1.0	7.0	10.0	14.0	45.84	195.0	8949.0

Figura 5-3. Valores estadísticos relacionados con la paralelización de tareas

De los valores anteriores se desprende que solo un 7% de las aplicaciones lanzan tareas en paralelo ya que, no es hasta el percentil 93 cuando se identifican las primeras aplicaciones con tareas paralelizadas. Otro valor a destacar es que al menos en el 99% de los casos, el paralelizado está por debajo de 200 instancias.

Con lo anterior, la selección del conjunto representativo debería basarse solo en el número de repeticiones con independencia que la aplicación paralelice tareas o no. Eligiendo un grupo que se hayan ejecutado un número considerable de veces como para disponer de una serie temporal con suficiente información, entre 30 y 200 repeticiones por ejemplo. Por otro lado, dado que en el 99% de los casos el número de paralelizado de tareas está por debajo de 200, se fijará este valor como máximo para descartar los casos con paralelizado más extremo, principalmente por cuestiones de rendimiento.

Al aplicar el filtrado comentado al conjunto de datos inicial, se obtiene un subconjunto de 1517 aplicaciones, con un numero de repetición de ejecuciones irregular y que en algunos casos aplican paralelizado y en otros no.

5.2 Transformación del conjunto de datos

Para el entrenamiento y predicción con los modelos de aprendizaje automático será preciso adaptar la estructura y contenido del conjunto de datos inicial para tal fin. Además, también será necesario añadir más características al conjunto de datos que aporten más información a la hora de entrenar los modelos. Las transformaciones necesarias se describen a continuación.

5.2.1 Filtrado de registros

Para el estudio de consumos máximos, en realidad solo interesa aquellos eventos donde se produce el pico máximo de consumo en cada ejecución. Por lo que se podría seleccionar solamente estos registros y prescindir del resto, reduciendo aún más el volumen de información a manipular. De modo que, para una sola ejecución (*collection_id*) e instancia de paralelizado (*instance_index*), se debe seleccionar solamente el registro que informa el valor máximo de consumo de CPU y el momento a lo largo de la ejecución donde este se produce.

Para la selección de los máximos consumos de CPU, basta con agrupar el dataset por aplicación (*collection_logical_name*), ejecución (*collection_id*) y tarea (*instance_index*). De cada agrupado resultante, se selecciona el registro que informe el mayor consumo de CPU (*cpus*). Estos eventos serán los incluidos en el conjunto final, descartando al resto.

Respecto al momento de la ejecución en segundos donde se produce el máximo es un valor que hay que calcular ya que, al ser relativo a un punto de inicio común, no aparece como tal en el conjunto de datos inicial. Para su cálculo, hay que identificar por un lado el tiempo de inicio de la ejecución, que será el registro con la menor fecha de inicio (*start_time*). Por otro lado, habrá que restar este valor a cada uno de los registros que componen la ejecución, de modo que se obtiene el desplazamiento en segundos de cada registro con respecto al inicio de la ejecución. Surge así un nuevo

campo para el conjunto que se llamará *offset*. Este valor representa, para un determinado registro, el número de segundos que han trascurrido desde que se inició la ejecución de su aplicación hasta que este registro se produce. Por ejemplo, cuando arranca la ejecución de una aplicación el valor del *offset* es cero. Si para esta aplicación hay un registro informado con un valor de offset igual a 20, quiere decir que este evento se produjo 20 segundos después de que la aplicación arrancase. De este modo, pueden identificarse los picos de consumo mediante el valor del *offset*.

5.2.2 Ampliación de características

Otro aspecto a tener en cuenta es el formato del campo *collection_logical_name*, su valor alfanumérico no es válido para el entrenamiento de los modelos ya que es un valor que la mayoría de modelos no admite. Para modificar su valor y conservar su unicidad dentro del conjunto, se aplicará una función hash sobre la cadena de texto inicial que lo convertirá en un número entero de longitud variable y que sí será válido para el entrenamiento de los modelos. El nuevo campo se llamará *app_id* y sustituirá al comentado anteriormente. En este punto hay que verificar que no se producen colisiones en las transformaciones de valores hash. Es decir, no se generan valores de *app_id* duplicados.

Además, dado que las predicciones se realizarán mediante series temporales, es preciso enriquecer cada evento de tiempo con información relativa a eventos anteriores, de modo que los modelos dispongan de más información acerca de estados de ejecución anteriores. Se añadirán 6 nuevos campos al conjunto de datos que serán obtenidos a partir de los registros ya incluidos en el mismo:

- **Consumo anterior** (*cpus-1*): Cada registro tendrá informado el consumo máximo de CPU que se produjo en la ejecución anterior.

- **Offset anterior** (*offset-1*): Cada registro tendrá informado el momento en segundos de la ejecución anterior donde se produjo el consumo máximo de CPU.

- **Media de consumo** (*cpus_mean*): Cada registro incluirá la media centrada de consumo máximo generada durante las 3 últimas ejecuciones.

- **Media de offset** (*offset_mean*): Cada registro incluirá la media centrada del offset donde se produjo el consumo máximo de CPU en las 3 últimas ejecuciones.
- **Desviación estándar de consumo** (*cpus_std*): Cada registro incluirá la desviación estándar de consumo máximo generada durante las 3 últimas ejecuciones.
- **Desviación estándar de offset** (*offset_std*): Cada registro incluirá la desviación estándar del offset donde se produjo el consumo máximo de CPU en las 3 últimas ejecuciones.

Comentar que los valores de media y desviación estándar solo contemplan las 3 últimas ejecuciones para no penalizar aquellas aplicaciones con menor número de repeticiones. Hay casos en el conjunto de datos donde alguna tarea paralelizada solo se ejecuta 5 veces por lo que, si se aumenta el histórico para calcular el agregado, se perdería información relativa a este tipo de ejecuciones. Los valores anteriores han de conseguirse agrupando los registros de diversas formas y aplicando las funciones estadísticas sobre estas agrupaciones.

5.2.3 Distribución de los conjuntos de entrenamiento, validación y pruebas

Dado que el entrenamiento se realizará utilizando series temporales, la distribución de registros ha de realizarse respectando la cronología de tiempo en la que estos se producen. De modo que el entrenamiento no se realice conociendo registros futuros antes que registros pasados.

El reparto se realizará en base a cada grupo de aplicación e instancia (par *app_id* – *instance_index*). Se reservará el último registro informado para el conjunto de pruebas, el penúltimo para el conjunto de validación y el resto de eventos anteriores a estos para entrenamiento.

De este modo, el conjunto de entrenamiento contendrá la mayoría de registros que, en orden cronológico, serán entregados a los modelos para su entrenamiento. El

conjunto de validación contendrá el registro inmediatamente siguiente a los del conjunto anterior, sirviendo para realizar predicciones intermedias necesarias para el refinamiento de los modelos. A su vez, el conjunto de pruebas contendrá el registro inmediatamente siguiente a los del conjunto de validación, siendo el último del conjunto de datos inicial y dedicado a la predicción final del modelo una vez refinado.

Añadir que, una vez ajustado el modelo, su entrenamiento con la parametrización óptima ha de realizarse con la unión de los conjuntos de entrenamiento y validación de cara a la predicción con el conjunto de pruebas. De este modo, se asegura que la predicción se realiza para el evento inmediatamente siguiente.

Los campos que compondrán el entrenamiento serán : *app_id, instance_index, cpus-1, offset-1, cpus_mean, offset_mean, cpus_std y offset_std*. Una vez entrenado, los conjuntos de validación y pruebas estarán formado por los campos: *cups* y *offset*.

Con lo anterior, se tiene una serie de tiempo multivariada con la que se intentarán predecir dos valores objetivos (*CPU* y *offset*) mediante un conjunto de modelos de aprendizaje automático.

Por otro lado, dado que algunos modelos necesitan que los datos de entrada se encuentren estandarizados, el conjunto de datos inicial deberá tener normalizados sus campos dentro del rango [0,1]. Para los valores relacionados con la CPU no será necesario ya que los valores informados ya se encuentran en ese rango. Además, esta conversión ha de ser reversible, de modo que existan mecanismos que permitan obtener los valores originales a partir de los ya estandarizados.

Finalmente, cada conjunto de datos habrá de almacenarse en un fichero csv independiente y cada conjunto tendrá por separado los atributos de entrada por un lado y los atributos objetivos por otro. Por ejemplo, para el conjunto de entrenamiento existirá un fichero *csv* que contendrá los atributos objetivos *cpus* y *offset* y otro fichero *csv* que contendrá el resto de atributos utilizados en el entrenamiento.

5.2.4 Selección de los modelos de aprendizaje automático

La selección de los modelos de aprendizaje automático está determinada por el contenido de los registros del conjunto de datos a utilizar. Normalmente, los conjuntos de datos utilizados en la predicción mediante series temporales utilizan registros reportados a intervalos regulares por una sola fuente de información (*univariate*) o por varias fuentes para un mismo punto en el tiempo (*multivariate*).

Para el caso concreto del conjunto de datos de *traces*, se tienen un total de 20.219.223 registros reportados por 1571 aplicaciones distintas a intervalos irregulares. Esto aporta complejidad a la predicción ya que, de modo secuencial, un modelo va a ir recibiendo registros de una determinada aplicación en un momento de tiempo *t*, que no tienen por qué provenir de la misma aplicación que del registro reportado en el momento de tiempo *t-1*.

Debido a esta particularidad, se descartan los modelos de predicción de series temporales basados en métodos estadísticos tales como Modelos Auto Regresivos, Media Móvil, ARIMA y derivados [16]. Estos modelos se basan en los valores informados en puntos de tiempo anteriores y, al provenir en este caso de aplicaciones diferentes, los cálculos estadísticos a utilizar pueden no ser fiables. Es decir, para que estos modelos fuesen viables habría que crear 1517 conjuntos de datos distintos, cada uno asociado a una determinada aplicación y a partir de hay hacer los estudios de estacionalidad, existencia de patrones y evolución de tendencias para entrenar 1517 modelos distintos.

En su lugar, se recurrirán a un conjunto de modelos de aprendizaje automático de los que se espera puedan capturar las particularidades que contiene el conjunto de datos de *traces* y ser utilizadas para realizar predicciones acerca del consumo máximo que tendrá una aplicación y el punto del tiempo en el que se producirá.

Tras realizar un análisis de los modelos de aprendizaje que han resultado ser válidos para la predicción de series temporales [16, 17, 18] se seleccionan para el conjunto dos modelos de aprendizaje automático y otros dos de aprendizaje automático profundo. Son los siguientes:

- Modelo de bosque aleatorio.
- Modelo de intensificación de gradiente.
- Red Neuronal Recurrente (RNN).
- Red Neuronal Convolucional (CNN)

Los bosques aleatorios se crean a partir de un conjunto de árboles de decisión, donde la predicción del bosque surge de promediar la predicción de todos los árboles que lo componen. Tienen como ventaja que son sencillos de entrenar al tener que informar pocos parámetros de configuración y que no tienden a sobre entrenar el conjunto de datos.

El modelo de intensificación de gradiente es parecido al modelo anterior en el sentido de que también se basa en la construcción de árboles pero de un modo distinto. En este caso las construcción de los árboles es incremental, de modo que los errores de los árboles iniciales son corregidos por los árboles creados con posterioridad. En este caso es posible que se produzca un sobre entrenamiento del modelo, por lo que habrá que utilizar mecanismos para limitar este efecto.

Respecto a los modelos de aprendizaje profundo, para la RNN se utilizarán celdas LSTM (*Long Short-Term Memory*) como mecanismo de aprendizaje del estado del conjunto de datos que se va recibiendo. Aunque esta tipología de red se utiliza en tareas de aprendizaje secuencial y reconocimiento del lenguaje. También se ha probado con éxito en series temporales [16].

Por otro lado, aunque las CNN se utilizan principalmente para el reconocimiento de patrones en imágenes y audio, se adaptará el conjunto de datos para entrenar una red neuronal de este tipo para comprobar su precisión en la predicción de series temporales. Para forzar la secuencialidad en el aprendizaje habrá que adaptar los registros del conjunto de datos de modo que formen una matriz de elementos donde aplicar la convolución. El modelo ira recibiendo grupos de registros solapados que hará

las veces de matriz bidimensional y donde los atributos de cada registro hará las veces de capas asociadas a esa matriz [19].

Además de los modelos comentados, se creará un modelo simple de predicción que servirá de línea base para comparar la eficacia del resto de modelos. Por ejemplo, podría ser un modelo de referencia que muestre como predicción los valores del registro de datos recibido anteriormente.

Lo que se espera de estos 4 modelos es que sean capaces de capturar las particularidades del conjunto de datos *traces* respecto a su heterogeneidad de valores informados para conseguir una predicción de consumo de CPU y *offset* aceptable.

5.2.5 Acciones a realizar sobre los modelos de Aprendizaje Automático

5.2.5.1 Entrenamiento

Aunque la configuración de cada modelo de aprendizaje es diferente, para que los resultados que se obtengan de ellos sean comparables deben entrenarse en un entorno que sea común y han de utilizar las mismas métricas de precisión.

Por ello, todos tendrán que entrenarse a partir del mismo conjunto inicial de datos, siendo los comandos de carga y los objetos que almacenen las información exactamente los mismos. Posteriormente, podrá realizarse alguna modificación sobre la estructura del conjunto de datos si así lo requiere el entrenamiento del modelo pero de base, todos han de partir de los mismos ficheros *csv*. Además, deberá existir alguna comprobación que verifique que la carga del fichero ha sido correcta antes de que los datos sean utilizados por el modelo de aprendizaje.

5.2.5.2 Pruebas

La métrica a utilizar para evaluar la precisión en la predicción de los modelos será la raíz cuadrada del error medio al cuadrado (RMSE, *Root Mean Squared Error*). Se decide utilizar esta métrica debido a que es rápida de calcular con respecto al error absoluto medio (MAE, *Mean Absolute Error*) y que está incluida en las clases a utilizar para entrenar los modelos como función de pérdida [20]. Por otro lado, aunque puede ser más sensible a valores atípicos (*outliers*), se considera que el filtrado inicial que se

realizará en los datos, donde solo se obtienen ejecuciones que finalizan correctamente, actuará como filtro de estos casos atípicos.

Las pruebas sobre los modelos consistirán en una serie de entrenamientos con distintas configuraciones de sus hiperparámetros, el objetivo es ir haciendo pequeños ajusten que produzcan distintos valores de RMSE. Las validaciones de los entrenamientos se realizará haciendo predicciones sobre el conjunto de validación.

Una vez finalizada la fase de pruebas, se volverá a entrenar el modelo con la configuración que mejores resultados haya obtenido en global tanto para *CPU* como para *offset*. Tras el entrenamiento, se realizará una nueva predicción utilizando en esta ocasión el conjunto de pruebas. El resultado que se obtenga debe quedar reflejado tanto cuantitativa como cualitativamente, mediante una tabla de valores y unas gráficas que muestren el error producido, por ejemplo. Será este resultado el que se utilice para la comparativa entre los distintos modelos.

5.2.6 Resumen

En este capítulo se ha analizado el contenido del conjunto de datos a nivel local así como algunos de sus valores estadísticos. Por cuestiones de rendimiento, este se ha acotado a un grupo más reducido de aplicaciones pero de modo representativo, con un número suficiente de ejecuciones que permita el entrenamiento mediante series temporales. Además, se han establecido los criterios para seleccionar los registros más relevantes a nivel de aplicación y se han detallado la relación de atributos que han de incluirse en el conjunto de datos para enriquecer a este.

Por otro lado, se han establecido la distribución de los registros en los distintos subconjuntos de entrenamiento, validación y pruebas. Finalmente, se ha fijado la relación de modelos de aprendizaje automático a evaluar así como sus requisitos para el entrenamiento y pruebas.

6 Diseño

En la fase de análisis se revisó el conjunto de datos inicial y se establecieron algunos requisitos para la creación y prueba de los modelos de aprendizaje automático. En esta fase se profundizará más en estos aspectos, planteando algunas decisiones de diseño y definiendo el modo en que estas especificaciones han de ser abordadas.

6.1 Transformación del conjunto de datos

Para aplicar las transformaciones requeridas se continuará en la línea de capítulos anteriores, utilizando el leguaje *Python.v3* como código de programación sobre documentos de formato *Jupyter Notebooks*. La carga de datos se realizará sobre un objeto de la clase *pandas.DataFrame* y se utilizarán las funcionalidades que esta clase ofrece para ir realizando las transformaciones requeridas.

Una vez se tenga cargado el objeto con el conjunto de datos, hay que recorrerlo a nivel de aplicación. De modo que se vayan cargando en un objeto intermedio todos los registros relativos a una determinada aplicación y poder realizar en bloque el filtrado de registros y la ampliación de características.

Para la modificación del contenido del campo *collection_logical_name* se diseñará una función que transforme una cadena de texto a un valor hash con formato entero. La función ha de aplicarse en bloque sobre el objeto intermedio y su resultado almacenado en un nuevo atributo con nombre *app_id*.

Para el cálculo del offset habrá que realizar un segundo agrupado sobre el objeto intermedio para conseguir que sus registros pertenezcan a un determinado *app_id-instance_index*. Para cada subgrupo, se calcula el offset a partir de la diferencia entre su fecha de ejecución y la del registro con menor fecha del grupo. El valor obtenido será el *offset* del registro y el valor almacenado en un nuevo atributo del objeto intermedio.

Para la selección de los consumos máximos, se aprovecha el agrupado anterior y se selecciona aquel registro que tenga informado el mayor consumo de CPU. El resto de registro se descartan del objeto intermedio. En este punto, se tiene el consumo

máximo producido en todas las ejecuciones de una determinada aplicación así como el momento de la ejecución donde se produjo.

Con lo anterior, se puede proceder al cálculo de los valores que ampliarán las características del conjunto. Con las funciones *rolling*, *shift*, *mean* y *std* incluidas en el objeto *DataFrame* intermedio debería ser suficiente para calcular los valores anterior, medio y desviación estándar de cada uno de los registros del mismo. Los valores resultantes serán añadidos al objeto como los 6 nuevos atributos que se especificaron con los siguientes nombres: *cpus-1, cpus_mean, cpus_std, offset-1, offset _mean y offset _std.*

Una vez aplicada todas las transformaciones sobre el objeto intermedio, se añade el contenido ordenado por *app_id, instance_index y timestamp* de este a un nuevo objeto *DataFrame* que será el que componga el conjunto de datos final. El proceso anterior se repetirá sobre todas las aplicaciones que componen el conjunto de datos inicial.

La normalización de atributos se aplicará una vez se obtenido el conjunto de datos comentado en los párrafos anteriores. Dado que la conversión ha de ser reversible, los valores originales de máximo serán almacenados en un *DataFrame* intermedio con la idea de exportarlo a un fichero *csv*. De este modo, los valores serán recuperables en cualquier momento del entrenamiento de los modelos.

Respecto a la distribución de registros en los ficheros de entrenamiento, validación y pruebas, se realizará una iteración parecida a la indicada anteriormente. Se irán conformando grupos a nivel de par *app_id-instance_index* e insertándolos en un objeto *DataFrame* intermedio. Los registros estarán ordenados por *timestamp*, de modo que puedan ir repartiéndose a tres variables tipo *array* distintas en función de la posición en la que se encuentren: el último a un *array*, el penúltimo a otro y el resto de registros a otro *array* más. Este reparto es la equivalencia a los conjuntos pruebas, validación y entrenamiento respectivamente.

Una vez completado el reparto previo sobre todas las aplicaciones se exportará el contenido de los *arrays* a ficheros con formato *csv*. De modo que cada uno debe

generar dos ficheros, uno que contendrá los campos a utilizar en los entrenamientos y otro fichero que contendrá las variables objetivo. Es decir, el conjunto quedará repartido en 6 ficheros tal y como se describe en la figura 6-1.

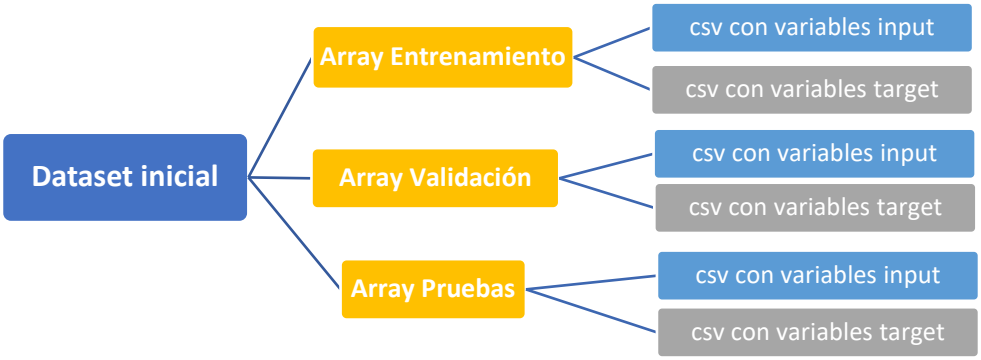

Figura 6-1. Distribución de contenido en los ficheros para entrenamiento, validación y pruebas

La funcionalidad del reparto anterior se consigue utilizando *DataFrames* intermedios y exportando desde estos objetos a los ficheros *csv*. Del mismo modo, ha de implementarse una estructura inversa que permita cargar el contenido de estos ficheros a objetos tipo *DataFrame.* El diseño de la funcionalidad es el inverso al descrito anteriormente.

6.2 Modelos de aprendizaje automático

Aunque cada modelo de aprendizaje automático tiene unas particularidades de diseño, hay que diseñar un conjunto de funcionalidades comunes que permitan la entrega de los conjuntos de datos a los modelos y la posterior validación de los resultados obtenidos. Los siguientes apartados muestran con más detalle estos aspectos.

6.2.1 Estructuras comunes

La carga del contenido de los ficheros *csv* ha de hacerse en objetos *DataFrame.* De modo que el contenido del conjunto de datos inicial se encuentre repartido en 6 objetos que los modelos de datos puedan interpretar. Una vez se realice la carga, se comprobará el tamaño de todos los conjuntos para verificar que la carga ha sido correcta, se utilizarán funciones básicas de *Python* para esta comprobación.

Del mismo modo, se cargará en un *Dataframe* los datos relativos a la normalización de atributos comentada en el apartado anterior. Se diseñará una función que hará la reversión de la normalización y que estará a disposición de los modelos para cuando se precise.

Por otro lado, los valores del error RMSE resultantes de la predicción de los modelos deberán ser calculados del mismo modo. Tanto predicciones como valores reales se adaptarán a objetos *pandas.Series.* Ambos objetos serán utilizados por una función común que devolverá el resultado de los errores RMSE.

Los valores resultantes en cada lanzamiento se irán añadiendo a una tabla que estará incluida dentro del mismo documento de *Jupyter Notebooks*. De este modo se tendrán representados todos los valores de modo cuantitativo.

Para la representación cualitativa se utilizarán los objetos *Series* que contenían el error RMSE y los objetos *Series* de los valores originales. Creándose así un nuevo objeto *DataFrame* que será utilizado para crear unas gráficas comparativas de predicción y valor real. El mecanismo de creación de la gráfica será el mismo y su representación varía en función del *DataFrame* que utilice como entrada. Básicamente, se mostrará sobre una línea de tiempo el solape existente entre cada uno de los registros incluidos en el conjunto de pruebas. Tanto valor real como la predicción tendrán un color asignado de modo que las diferencias sean visualmente apreciables.

6.2.2 Modelo básico de predicción (baseline)

Como ya se comentó en el capítulo anterior, este modelo servirá de línea base para comparar la eficacia del resto de modelos. No tiene entrenamiento y simplemente utiliza como predicción el valor informado en el campo *cpus-1.* Tampoco hay hiperparámetros que configurar, por lo que sólo habría que pasar las columnas de valor real (*cpus*) y predicción (*cpus-1*) a la función de cálculo del error RMSE.

6.2.3 Modelo de Bosque aleatorio

Los modelos de bosque aleatorio se caracterizan por necesitar pocos hiperparámetros en su entrenamiento. El valor más destacable es el número de árboles a crear. Para su construcción se utilizará la clase *RandomForestRegressor*[21] de *sklearn*.

De los hiperparámetros configurables que ofrece esta clase se utilizarán el número de árboles que compondrán el modelo y el uso de *Bootstrap* (árboles con repetición de registros) en el entrenamiento. Ambos hiperparámetros serán configurables de modo que puedan realizarse varios entrenamientos con distintos valores.

Las ordenes de entrenamiento y predicción serán independientes y recibirán por parámetro el conjunto de datos con el que han de trabajar. El modelo admite la predicción de más de una variable objetivo, así que el entrenamiento se hará en conjunto para estimación de *CPU* y *offset*.

6.2.4 Modelo de árbol de intensificación de gradiente

Para la creación de este modelo de aprendizaje automático se utilizará la clase *HistGradientBoostingRegressor* [22] de *sklearn*. Su uso está recomendado en conjuntos de datos con mas de 10.000 registros. Además, es más rápida de entrenar la clase original de la que deriva *GradientBoostingRegressor*[23].

De los hiperparámetros configurables que ofrece esta clase se utilizarán el factor de aprendizaje (*learning_rate*) y la profundidad de los árboles a crear. Dado que este modelo tiende al sobre entrenamiento, se hará uso del hiperparámetro *early_stopping* para detener el entrenamiento en curso y limitar así este efecto. Por lo que no será necesario aportar valores de configuración para el número máximo de árboles a crear. Los dos primeros hiperparámetros sí serán configurables de modo que puedan realizarse varios entrenamientos con distintos valores.

Las ordenes de entrenamiento y predicción serán independientes y recibirán por parámetro el conjunto de datos con el que han de trabajar. Dado que este modelo no admite la predicción de más de una variable objetivo, se realizarán dos

entrenamientos para cada configuración, uno para la estimación de *CPU* y otro para el *offset*.

6.2.5 Modelo de Red Neuronal Recurrente LSTM

El modelo se creará a partir de la clase *Sequential*[24] de *Keras*, donde se le irán añadiendo una serie de capas ocultas y una de salida que compondrán el modelo final. Todas las capas son pertenecientes al API *Keras Layers* [25].

La capa principal será la de tipo *LSTM*. Su número deberá ser configurable, de modo que pueda entrenarse el modelo con una o más capas de este tipo. Dentro de esta capa, se utilizará la función de regularización *relu* debido a que los valores que recibirá la red son positivos. También es necesario utilizar el hiperparámetro *return_sequences* para que los valores de una capa oculta pasen al la entrada de la capa siguiente. Además, también será configurable su hiperparámetro de número de celdas, de modo que pueda ajustarse para distintos entrenamientos.

Respecto a la capa de salida, será de tipo *Dense* y se configurará con dos neuronas de salida, una para cada variable objetivo. Además, se utilizará la función de activación sigmoide para acotar los valores de salida al rango [0, 1].

El modelo se compilará utilizando como función de pérdida el error medio cuadrado MSE, se selecciona esta función por ser la base de la que se utilizará para el cálculo del error de predicción RMSE.

En la compilación también se utilizará como algoritmo de optimización incluido en la clase *Adam*[26] de *Keras*. Mediante esta clase será posible poder utilizar distintos valores de *learning_rate* en cada entrenamiento.

Dado que puede existir sobre entrenamiento con este modelo, se utilizará la clase *EarlyStopping* de la API *Keras Callbacks*[27], de modo que el modelo finalice su entrenamiento cuando no existan mejoras en la métricas de pérdida durante un determinado número de ciclos.

Los conjuntos de validación se utilizarán durante los entrenamientos para apoyar el aprendizaje del modelo. También se utilizará como conjunto de predicción para los

distintos modelos intermedios que se entrenen. Finalmente, habrá que desnormalizar los valores de offset obtenidos durante la predicción para estimar el error RMSE de la misma.

Como la entrada de este modelo espera un array de 3 dimensiones donde una de ellas son los grupos de registros (*timesteps*), y dado que los registros del conjunto de entrenamiento se ubican en un solo grupo global, será preciso redimensionar el conjunto de datos inicial de dos dimensiones [muestras, [características]] a uno de 3 dimensiones que contemple la dimensión del *timesteps* [muestras, timesteps, [características]]. De este modo, se le indica al modelo que se le pasa un solo grupo de registros (*timesteps*) con un determinado número de muestras y cada una con un determinado conjunto de características. Esta funcionalidad puede implementarse a partir de las funciones *shape* y *reshape* de *pandas.DataFrame*.

6.2.6 Modelo de Red Neuronal Convolucional

El modelo se creará a partir de la clase *Sequential* de *Keras*, donde se le irán añadiendo una serie de capas ocultas y una de salida que compondrán el modelo final. Todas las capas son pertenecientes al API *Keras Layers*.

La estructura en este caso será fija en lo que respecta al número de capas, siendo los hiperparámetros de las mismas los elementos configurables que permitirán el entrenamiento de varios diseños. La tabla 6-1 muestra la relación de capas apiladas que conformarán el modelo.

Capa	Detalles e hiperparámetros configurables
Sequential	Capa de entrada al modelo
Conv1D	Capa de convolución, número de filtros configurable
MaxPooling1D	Capa de agrupado, selección del máximo valor
Dropout	Capa de pérdida. Ratio de pérdida configurable
Flatten	Capa de aplanamiento de la matriz de datos recibida
Dense	Capa completamente conectada.
Dropout	Capa de pérdida. Ratio de pérdida configurable
Dense	Capa completamente conectada de salida con 2 neuronas.

Tabla 6-1. Detalle de las capas que componen el modelo convolucional

Tal y como se observa en la tabla, el número de filtros de la capa de convolución ha de ser configurable así como el ratio de pérdida de las capas de *dropout*. Existirá un tercer parámetro configurable que será el factor de aprendizaje (*learning_rate*). El diseño será el mismo que el comentado en el modelo anterior, a través de la clase *Adam* de *Keras*. Las funciones de activación y el control de sobre entrenamiento también aplicaran del mismo modo ya comentado.

Los conjuntos de validación se utilizarán durante los entrenamientos para apoyar el aprendizaje del modelo. También se utilizará como conjunto de predicción para los distintos modelos intermedios que se entrenen.

Finalmente, habrá que desnormalizar los valores de *offset* obtenidos durante la predicción para estimar el error RMSE de la misma.

Al igual que el modelo anterior, la CNN también espera un array de 3 dimensiones donde una de ellas son los grupos de registros (*timesteps*). Sin embargo, en este caso sí es preciso que se adapte el conjunto de datos para que reciba lotes de registros. Esto se debe a que el modelo espera recibir una matriz de más de una dimensión sobre la que aplicar la convolución. Además, de este modo también se fuerza al modelo a que haga una lectura en la línea temporal en la que se producen los registros.

Se diseñará una función que haga esta división de grupos (*timesteps*), de modo que cada grupo esté formado por un registro y su dos registros inmediatamente anteriores. La tabla 6-2 muestra un ejemplo de cómo se conformarían los grupos.

Registros X de entrada
Registro 1
Registro 2
Registro 3
Registro 4
Registro 5
Registro 6
Registro 7
Registro 8

Nº Grupo	Registros X que lo componen	Salida de Y que se asocia
1	Registros 1, 2 y 3	Registro 3
2	Registros 2, 3 y 4	Registro 4
3	Registros 3, 4 y 5	Registro 5
4	Registros 4, 5 y 6	Registro 6
5	Registros 5, 6 y 7	Registro 7
6	Registros 6, 7 y 8	Registro 8
7		

Tabla 6-2. Detalle del agrupado de registros para el modelo convolucional

Con lo anterior, el modelo recibirá grupos de 3 elementos (X) siendo los valores objetivo del último registro recibido (Y) los que se utilicen para la predicción. Esta agrupamiento habrá que realizarlo en todos los grupos para que el modelo funcione correctamente.

6.3 Resumen

En este capítulo se ha establecido el diseño de los componentes que permitirán filtrar, transformar, importar, exportar y dividir el conjunto de datos inicial en otros subconjuntos que puedan ser utilizados por los modelos de aprendizaje.

Se han definido estructuras de entrada y salida comunes a todos los modelos así como estructuras específicas para la normalización y agrupamiento de datos requeridas por algunos modelos. También se han definido los componentes que realizarán la comparativa de los resultados obtenidos tanto cualitativa como cuantitativamente.

Finalmente, se ha establecido el diseño que han de tener cada uno de los modelos de aprendizaje, principalmente las estructuras a construir y los hiperparámetros utilizables y parametrizables.

7 Implementación

En la fase de diseño se profundizó en los aspectos requeridos para la creación y prueba de los modelos de aprendizaje automático, estableciendo el modo en el que se debían acometer determinadas funcionalidades y las estructuras que albergarían los datos.

Finalmente, en esta fase se comentarán las implementaciones realizadas y que dan lugar a los documentos que contienen toda la funcionalidad necesaria para el entrenamiento y pruebas de los modelos de aprendizaje automático mediante series temporales.

A continuación se comentarán las estructuras más relevantes, pudiéndose encontrar el contenido integro de las implementaciones en los ficheros referenciados en cada apartado.

7.1 Transformación del conjunto de datos

El fichero *creacion_del_dataset_principal_v4.ipynb* incluido en la documentación contiene el detalle de todas las implementaciones para el filtrado de registros y la ampliación de características. También incluye el cálculo del *offset* y la inclusión de los campos calculados basados en la *CPU* y el *offset* anterior. La figura 7-1 muestra el detalle del cálculo comentado para los campos de CPU.

```
#Desplaza los valores de cpu y offset a nivel de instance_index
#cpu
for instance_index in df_app_to_treat['instance_index'].unique():
    #valor anterior
    df_app_to_treat.loc[df_app_to_treat['instance_index'] == instance_index,'cpus-1']=\
    df_app_to_treat.loc[df_app_to_treat['instance_index'] == instance_index,'cpus'  ].shift(1)

    #media movil centrada
    df_app_to_treat.loc[df_app_to_treat['instance_index'] == instance_index,'cpus_mean']=\
    df_app_to_treat.loc[df_app_to_treat['instance_index'] == instance_index,'cpus'  ].\
    rolling(window=3,center=True).mean()

    #Desviación estándar
    df_app_to_treat.loc[df_app_to_treat['instance_index'] == instance_index,'cpus_std']=\
    df_app_to_treat.loc[df_app_to_treat['instance_index'] == instance_index,'cpus'  ].\
    rolling(window=3,center=True).std()
```

Figura 7-1. Detalle de la implementación para la ampliación de características de CPU

La normalización de atributos se realiza mediante una función que recibe por parámetro el objeto Series a normalizar y devuelve el mismo objeto ya normalizado y

otro objeto Series con la relación de valores utilizados en la normalización. De este modo el proceso se puede revertir en acciones posteriores. La figura 7-2 muestra el detalle de la implementación.

```
def norm_Series(serie):
    max_value=serie.max()
    min_value=serie.min()

    ini_len=len(serie)

    #Normaliza los datos
    serie= (serie - min_value) / (max_value - min_value)


    #Crea otra serie con los valores utilizados para la normalizacion
    norm_values=pd.Series([min_value,max_value],name=serie.name, index=[['min','max']])

    print("Normalizada serie: "+ serie.name + " i:"+str(ini_len)+" o:" + str(len(serie)) + " elementos.")

    return serie, norm_values
```

Figura 7-2. Detalle de la implementación de la función de normalización

Respecto a la distribución de registros en los ficheros de entrenamiento, validación y pruebas, la figura 7-3 muestra el detalle del reparto de registros en función de su posición una vez han sido agrupados a nivel del par *app_id-instance_index*. Las últimas líneas implementan el reparto de contenidos a nivel de atributos de entrada y objetivo.

```
    #Para cada bloque del par collection_id-instance_id, hace el reparto de registros

    for instance_index in df_app_to_treat['instance_index'].unique():

        #si hay registros suficientes para el reparto entre los 3 conjuntos, se reparte
        if len(df_app_to_treat[df_app_to_treat['instance_index']  == instance_index]) > 2:

            #control de lo seleccionado
            #print(str(coll_log_name)+" -> " + str(instance_index)+" -> " + str(len(df_app_to_treat[df_app_to_treat['instance

            #El ultimo registro se asigna al los conjuntos de test
            cluster_usage_A_test.append(df_app_to_treat[df_app_to_treat['instance_index']  == instance_index].iloc[-1:])

            #El penultimo registro se asigna a los conjuntos de validacion
            cluster_usage_A_val.append(df_app_to_treat[df_app_to_treat['instance_index']   == instance_index].iloc[-2:-1])

            #El resto de registros se asignan a los conjuntos de pruebas
            cluster_usage_A_train.append(df_app_to_treat[df_app_to_treat['instance_index'] == instance_index].iloc[:-2])

#Finalizados los repartos se costruyen los correspondientes dataset X Y
#repartiendo los campos entre variables y target
cluster_usage_A_train_X= pd.concat(cluster_usage_A_train)[cols_train].sort_index()
cluster_usage_A_train_Y= pd.concat(cluster_usage_A_train)[cols_target].sort_index()
cluster_usage_A_test_X = pd.concat(cluster_usage_A_test)[cols_train].sort_index()
cluster_usage_A_test_Y = pd.concat(cluster_usage_A_test)[cols_target].sort_index()
cluster_usage_A_val_X  = pd.concat(cluster_usage_A_val)[cols_train].sort_index()
cluster_usage_A_val_Y  = pd.concat(cluster_usage_A_val)[cols_target].sort_index()
```

Figura 7-3. Detalle de la creación de los grupos de entrenamiento validación y pruebas

7.2 Modelos de aprendizaje automático

Para la implementación de los modelos de aprendizaje automático se han creado 5 documentos de *Jupyter Notebooks*, uno por modelo. Cada fichero contiene implementadas las particularidades que se definieron durante la fase de diseño. Además, también implementan tanto las estructuras comunes para adaptar los datos a la entrada de los modelos como las de salida para la validación de resultados.

7.2.1 Estructuras comunes

Una de las estructuras diseñadas fue la de la carga de los conjuntos de entrenamiento, validación y pruebas a la entrada de cada modelo. La figura 7-4 muestra la implementación de la carga. En ella se observa que como cada fichero es asignado a un *DataFrame* que será el que posteriormente utilicen los modelos. Las ultimas líneas hacen referencia a la comprobación de la carga mediante la longitud de los distintos conjuntos de datos.

```
#Carga del fichero con los eventos
cluster_usage_A_train_X=pd.read_csv('../csv_datasets/cluster_A_train_test_datasets/cluster_A_train_X.csv.gz',
                                    sep=',', index_col='timestamp', compression='gzip',float_precision='legacy')

cluster_usage_A_train_Y=pd.read_csv('../csv_datasets/cluster_A_train_test_datasets/cluster_A_train_Y.csv.gz',
                                    sep=',', index_col='timestamp', compression='gzip',float_precision='legacy')

cluster_usage_A_test_X=pd.read_csv(  '../csv_datasets/cluster_A_train_test_datasets/cluster_A_test_X.csv.gz',
                                    sep=',', index_col='timestamp', compression='gzip',float_precision='legacy')

cluster_usage_A_test_Y=pd.read_csv(  '../csv_datasets/cluster_A_train_test_datasets/cluster_A_test_Y.csv.gz',
                                    sep=',', index_col='timestamp', compression='gzip',float_precision='legacy')

cluster_usage_A_val_X=pd.read_csv(  '../csv_datasets/cluster_A_train_test_datasets/cluster_A_val_X.csv.gz',
                                    sep=',', index_col='timestamp', compression='gzip',float_precision='legacy')

cluster_usage_A_val_Y=pd.read_csv(  '../csv_datasets/cluster_A_train_test_datasets/cluster_A_val_Y.csv.gz',
                                    sep=',', index_col='timestamp', compression='gzip',float_precision='legacy')

scaler_values=pd.read_csv(  '../csv_datasets/cluster_A_train_test_datasets/scaler_values.csv.gz',
                                    sep=',', index_col=0, compression='gzip',float_precision='legacy')

#Comprobación de tamaños
([len(cluster_usage_A_train_X),len(cluster_usage_A_test_X),len(cluster_usage_A_val_X)],
 [len(cluster_usage_A_train_Y),len(cluster_usage_A_test_Y),len(cluster_usage_A_val_Y)])
```

Figura 7-4. Carga de los conjuntos de entrenamiento validación y pruebas

En la figura anterior también se observa como una de las entradas está dedicada a los datos de escalado para poder revertir la normalización del conjunto. La funcionalidad se ha implementado mediante una función que recibe un objeto *Series* y el *DataFrame* que contiene los datos originales antes de normalizar. A partir de este aplica la reversión sobre el objeto *Series* y lo devuelve con sus valores originales. La figura 7-5 muestra el detalle de la implementación.

```
#Se crea una función que desnormaliza un objeto Series
#a partir de sus valores min/max originales

def denorm_Series(serie,scaler_values):
    max_value=scaler_values.loc['max'][serie.name]
    min_value=scaler_values.loc['min'][serie.name]

    #desnormaliza los datos
    serie= (serie + min_value) * (max_value + min_value)

    return serie
```

Figura 7-5. Función para revertir una normalización aplicada

Como ya se comentó, era necesario revertir la normalización una vez entrenado el modelo para poder calcular el error RMSE de la predicción a partir de los valores originales. Este cálculo era otra de las estructuras de salida que se comentaron en el diseño. La figura 7-6 muestra el detalle de cómo se reasignan los valores obtenidos en una predicción para que pueda calcularse el error RMSE de forma estandarizada. El detalle también muestra la llamada a la función comentada anteriormente para hacer el cálculo a partir de los valores originales.

```
#Valores de la prediccion
p_cpu=p[:,0]
p_offset=denorm_Series(pd.Series(p[:,1],name='offset',index=cluster_usage_A_val_Y['offset'].index),scaler_values)

#Calculo del RMSE
[
mean_squared_error(cluster_usage_A_val_Y['cpus'], p_cpu, squared=False),
mean_squared_error(cluster_usage_A_val_Y['offset'], p_offset, squared=False)
]
```

Figura 7-6. Adaptación de datos para el cálculo de error RMSE

La funcionalidad anterior entrega el valor cuantitativo del error, para la parte cuantitativa se han implementado dos tipos de gráficas que muestran sobre una línea de tiempo el solape existente entre valor real y valor predicho. Una está enfocada en resaltar la distancia de aquellas predicciones que más discrepan respecto al valor real. La otra se centra en las pequeñas distancias de aquellos casos donde la predicción es más acertada. La figura 7-7 muestra los detalles de la implementación, en ella se observa la creación del objeto que contiene todos los valores y las 4 órdenes que generaran las 2 gráficas para *CPU* y *offset*.

```
#Graficas comparativa valor real vs valor de predicción

diff=pd.DataFrame([cluster_usage_A_test_Y['cpus'].values,
                   cluster_usage_A_test_Y['offset'].values,
                   p_cpu, p_offset], index=['cpus','offset','p_cpus','p_offset']).transpose()

diff[['cpus','p_cpus']].plot.line(figsize=(w,h),alpha=0.5,linestyle="-", marker="o", markersize=10 ,color=['red','blue'])
diff[['offset','p_offset']].plot.line(figsize=(w,h),alpha=0.5,linestyle="-", marker="o", markersize=10 ,color=['red','blue'])

diff[['cpus','p_cpus']].plot.line(figsize=(w,h),alpha=0.5,linewidth=2,color=['red','blue'])
diff[['offset','p_offset']].plot.line(figsize=(w,h),alpha=0.5,linewidth=2,color=['red','blue'])
```

Figura 7-7. Implementación de las gráficas de valor real vs predicción

7.2.2 Modelo básico de predicción (baseline)

Como ya se comentó en el capítulo anterior, este modelo servirá de línea base para comparar la eficacia del resto de modelos. No tiene entrenamiento y simplemente utiliza como predicción el valor informado en el campo *cpus-1*. La figura 7-8 muestra cómo se calcula directamente el error RMSE a partir del campo mencionado.

```
#Valores de la prediccion

#La prediccion de cpu corresponde al campo cpu-1
p_cpu=cluster_usage_A_test_X['cpus-1']

#La prediccion de offset corresponde al campo offset-1
p_offset=denorm_Series(cluster_usage_A_test_X['offset-1'],scaler_values)


#Calculo del RMSE
[
mean_squared_error(cluster_usage_A_test_Y['cpus'], p_cpu, squared=False),
mean_squared_error(cluster_usage_A_test_Y['offset'], p_offset, squared=False)
]
```

Figura 7-8. Implementación de la predicción del modelo base

El fichero *models/00_Baseline_model.ipynb* incluido en la documentación contiene el detalle de toda la implementación para este modelo de aprendizaje.

7.2.3 Modelo de Bosque aleatorio

La figura 7-9 muestra la implementación del modelo en base a la especificación indicada en el capítulo anterior. En ella se observan los parámetros configurables para el entrenamiento de varias versiones del modelo.

```
from sklearn.ensemble import RandomForestRegressor

mdl = RandomForestRegressor(n_estimators=200, n_jobs=-1, random_state=0, oob_score=True,bootstrap=True)
mdl.fit(cluster_usage_A_train_X, cluster_usage_A_train_Y)
```

Figura 7-9. Implementación del modelo de bosque aleatorio

El modelo se entrena para las dos variables objetivo tal y como se especificó. La figura 7-10 muestra el detalle de la función de predicción, donde se observa que se le pasa como parámetro el conjunto completo de los datos de validación.

```
#Predice con el conjunto de test
p = mdl.predict(cluster_usage_A_test_X)

#Valores de la prediccion
p_cpu=p[:,0]
p_offset=denorm_Series(pd.Series(p[:,1],name='offset',index=cluster_usage_A_test_Y['offset'].index),scaler_values)

#Calculo del RMSE
[
mean_squared_error(cluster_usage_A_test_Y['cpus'], p_cpu, squared=False),
mean_squared_error(cluster_usage_A_test_Y['offset'], p_offset, squared=False)
]
```

Figura 7-10. Detalle de la predicción con el modelo de bosque aleatorio

El fichero *models/01_random_forrest_model.ipynb* incluido en la documentación contiene el detalle de toda la implementación para este modelo de aprendizaje.

7.2.4 Modelo de árbol de intensificación de gradiente

La figura 7-11 muestra la implementación del modelo en base a la especificación indicada en el capítulo anterior. En ella se observan los parámetros configurables para el entrenamiento de varias versiones del modelo.

```
from sklearn.ensemble import HistGradientBoostingRegressor

#Entrenamiento para la CPU
mdl_GB_cpus = HistGradientBoostingRegressor(early_stopping=True,  max_iter=10000, learning_rate=0.01, max_depth=6)
mdl_GB_cpus.fit(cluster_usage_A_train_X, cluster_usage_A_train_Y['cpus'])


#Entrenamiento para el offset
mdl_GB_offset = HistGradientBoostingRegressor(early_stopping=True, max_iter=10000, learning_rate=0.01, max_depth=6)
mdl_GB_offset.fit(cluster_usage_A_train_X, cluster_usage_A_train_Y['offset'])
```

Figura 7-11. Implementación del modelo de árbol de intensificación de gradiente

En la figura también se observa como en realidad se entrenan dos modelos, uno para cada variable objetivo. Como ya se comentó, este modelo no admite la predicción de más de una variable objetivo. De ahí que se haya optado por dos entrenamientos independientes.

Lo mismo ocurre a la hora de realizar la predicción, cada conjunto objetivo deberá ser pasado al modelo correspondiente. La figura 7-12 muestra el detalle de la implementación.

```
#Prediccion utilizando el conjunto de validación
p_GB=pd.DataFrame()
p_cpu    = mdl_GB_cpus.predict(cluster_usage_A_val_X)
p_offset = mdl_GB_offset.predict(cluster_usage_A_val_X)

#Valores de la prediccion
p_offset=denorm_Series(pd.Series(p_offset,name='offset',index=cluster_usage_A_val_Y['offset'].index),scaler_values)
```

Figura 7-12. Detalle de la predicción con el modelo de árbol de intensificación de gradiente

El fichero *models/02_Gradient Boosted_Trees_model.ipynb* incluido en la documentación contiene el detalle de toda la implementación para este modelo de aprendizaje.

7.2.5 Modelo de Red Neuronal Recurrente LSTM

La figura 7-13 muestra la implementación del modelo en base a la especificación indicada en el capítulo anterior. En ella se observan los parámetros configurables para el entrenamiento así como el optimizador y el objeto *callback* que controlará el sobre entrenamiento del modelo.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam


# Estructura de la red
mdl = Sequential()
mdl.add(LSTM(40, return_sequences=True, input_shape=(train_X.shape[1], train_X.shape[2]), activation="relu"))
mdl.add(LSTM(40, return_sequences=True, activation="relu"))
mdl.add(Dense(2,activation="sigmoid"))

#Crea una callback para controlar la parada por sobreajuste
earlyStop=EarlyStopping(monitor="val_loss",verbose=2,mode='min',patience=5)

optimizer = Adam(learning_rate=0.001)

#Compila el modelo
mdl.compile(loss="mse", optimizer=optimizer)

# Entrenamiento
history = mdl.fit(train_X, train_y, epochs=100, batch_size=32, validation_data=(val_X, val_y), \
                  shuffle=False, callbacks=[earlyStop])
```

Figura 7-13. Implementación del modelo de Red Neuronal Recurrente LSTM

La función para desnormalizar los valores de *offset* obtenidos durante la predicción para estimar el error RMSE de la misma es la misma que la ya comentada en el modelo anterior.

Respecto a la redimensión del conjunto inicial de 2 a 3 dimensiones para que el modelo lo acepte como matriz de entrada, la implementación se ha realizado a partir de las funciones *shape* y *reshape* de *pandas.DataFrame* tal y como se especificó. La figura 7-14 muestra los pasos seguidos para la restructuración comentada.

```
#Asignacion de valores input
train_X=cluster_usage_A_train_X.values
test_X=cluster_usage_A_test_X.values
val_X=cluster_usage_A_val_X.values

#Asignacion de valores target
train_y=cluster_usage_A_train_Y.values
test_y=cluster_usage_A_test_Y.values
val_y=cluster_usage_A_val_Y.values

#Acciones de redimensionado
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))
val_X = val_X.reshape((val_X.shape[0], 1, val_X.shape[1]))
print(train_X.shape, train_y.shape, test_X.shape, test_y.shape, val_X.shape, val_y.shape)
```

Figura 7-14. Redimensión de los conjuntos de entrada para el modelo LSTM

El fichero *models/03_LSTM_model.ipynb* incluido en la documentación contiene el detalle de toda la implementación para este modelo de aprendizaje.

7.2.6 Modelo de Red Neuronal Convolucional

La figura 7-15 muestra la implementación del modelo en base al diseño indicado en el capítulo anterior. En ella se observan el apilamiento de las capas incluyendo los parámetros configurables para el entrenamiento.

```
from keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Conv1D, MaxPooling1D, Flatten, Dropout, Dense


# Estructura de la red
mdl = Sequential()
mdl.add(Conv1D(128, kernel_size=2, activation='relu', input_shape=(n_steps, train_X.shape[2])))
mdl.add(MaxPooling1D(pool_size=2))
mdl.add(Dropout(0.50))

mdl.add(Flatten())
mdl.add(Dense(32,activation='relu'))
mdl.add(Dropout(0.50))
mdl.add(Dense(2, activation="sigmoid"))
```

Figura 7-15. Implementación del modelo de Red Neuronal Convolucional

La implementación del optimizador y del objeto *callback* que controlará el sobre entrenamiento del modelo es la misma que la ya comentada en el modelo anterior.

Respecto a la redimensión del conjunto inicial de 2 a 3 dimensiones para que el modelo pueda realizar la convolución sobre los grupos de registros, se ha implementado una función que recibe por parámetro los conjuntos de variables *input, target* y el número de registros que han de conformar cada grupo. La función recorrerá

secuencialmente los conjuntos e irá haciendo el agrupado correspondiente. Finalmente, devuelve un objeto tipo *array* con el redimensionado correcto para cada conjunto recibido. La figura 7-16 muestra los detalles de la implementación comentados.

```
# Funcion para agrupar un dataset en lotes de secuencias de registros
# Necesario para que la convolucion se realice correctamente

from numpy import array

def split_sequences(dataset_X, dataset_Y, n_steps):
    X, y = list(), list()
    for i in range(len(dataset_X)):
        # Busca el final del patron
        end_ix = i + n_steps
        # Comprueba si se sobrepasa el dataset
        if end_ix > len(dataset_X):
            break
        #Reune las partes de entrada y salida del patron
        seq_x, seq_y = dataset_X[i:end_ix,:], dataset_Y[end_ix-1, :]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)
```

Figura 7-16. Redimensión de los conjuntos de entrada para el modelo CNN

El fichero *models/04_CNN_model.ipynb* incluido en la documentación contiene el detalle de toda la implementación para este modelo de aprendizaje.

7.3 Resumen

En este capítulo se ha detallado la implementación de todos componentes, estructuras y modelos que se especificaron y diseñaron en capítulos anteriores. Cada apartado ha incluido los bloques de código más relevantes tanto comunes como específicos de cada modelo. Además, se han insertado las referencias a los ficheros de formato *ipynb* incluidos en la documentación donde pueden encontrarse el resto de detalles implementados.

8 Pruebas

En este capítulo se revisarán las pruebas realizadas sobre los modelos de predicción implementados en el capítulo anterior así como los resultados de las mismas.

Para todos los modelos la dinámica de las pruebas es la misma: los modelos serán entrenados con distintas configuraciones de hiperparámetros y probados realizando predicciones sobre el conjunto de validación. Los resultados se irán anotando en una tabla y, una vez finalizados todos los lanzamientos, se seleccionará la configuración que mejores resultados haya obtenido.

El modelo se entrenará de nuevo con esta configuración más exitosa y se realizará una única predicción sobre el conjunto de pruebas. El resultado será utilizado para montar las gráficas comparativas de predicción vs valor real. Además, los valores de RMSE obtenidos en este entrenamiento serán los utilizados para realizar la comparativa entre todos los modelos.

Añadir que, previamente a la selección de los valores de los hiperparámetros que se informarán en la tabla, se realizan algunos entrenamientos con otros valores que permitan acotar los rangos finales a utilizar en cada modelo. Es decir, los valores asignados no son fruto del azar, sino de una revisión previa de cuáles pueden ser los valores más idóneos para refinar el entrenamiento de los modelos.

Añadir que el RMSE representa la raíz cuadrada de la varianza de los residuos. Su valor indica el error que, de media, se comete a la hora de realizar una predicción. Cuanto más bajo sea, más acertadas son las predicciones que el modelo realiza. Por otro lado, la métrica es sensible a valores atípicos (*outliers*) pero, como ya se comentó en el capítulo de análisis, se considera que el filtrado inicial realizado en los datos para obtener solo ejecuciones que finalizan correctamente actuará como filtro de estos casos atípicos. Es decir, que todos los registros reportados son correctos y que no se trata de errores en la captura de los mismos.

Respecto a las gráficas que representarán el error en la predicción de modo cualitativo, el código de colores utilizado es el mismo para todas las líneas representadas. El color azul corresponde a la predicción realizada por los modelos, en

la leyenda aparecerán referenciados como p_*cups* y p_*offset*. El color rojo hace referencia al valor real informado para una determinada ejecución, en la leyenda aparecerán referenciados como *cups* y *offset*.

Por otro lado, todas las gráficas comparten la misma escala de valores en el eje *X*. El valor que se informa es el de la posición de orden que cada registro ocupa en el conjunto de datos objetivo. Es decir, cada punto del eje X corresponde a uno de los 3206 registros que lo componen. De modo que la equivalencia de posición de la gráfica y el conjunto de datos sea más evidente a la hora de cotejar ambos elementos. Además, dado que estos se encuentran ordenados temporalmente por *timestamp*, el eje X también puede considerarse una línea temporal.

Cada punto informa el momento en que se produce el pico de consumo para una determinada aplicación e instancia (par *app_id-instance_index*). Estará informado con al menos dos valores: el real y el de la predicción. Cuanto más próximos se encuentren estos puntos mejor es la predicción realizada, lo ideal es que estuviesen solapados. Por el contrario, la predicción no será buena si los puntos se encuentran verticalmente muy distanciados. Respecto al eje *Y*, su escala depende de la variable objetivo que se predice: *NCU* si es *CPU* o segundos si es el *offset*.

8.1 Modelo básico de predicción (baseline)

Como ya se comentó en el capítulo anterior, este modelo servirá de línea base para comparar la eficacia del resto de modelos. No tiene entrenamiento y simplemente utiliza como predicción el valor informado en el campo *cpus-1*, por lo que se calculará directamente el error RMSE a partir de los valores *cpus* y *cpus_1*. Son los mostrados en la tabla 8-1.

CPU RMSE	OFFSET RMSE
0.0185703639	3360.89590

Tabla 8-1. Error RMSE del modelo baseline obtenido en pruebas

Dado que solo ha habido un lanzamiento, estos valores de RMSE son los mejores que este modelo puede alcanzar y los que representan la precisión del mismo.

Las figuras 8-1 y 8-2 representan el error cometido en la predicción de consumo de CPU para todas las aplicaciones. La figura 8-1 muestra que existen muchos casos en los que la discrepancia entre predicción y valor real es elevada, sobre todo en torno al rango [1200, 2600]. Estos valores influyen negativamente en el valor RMSE obtenido.

Además, no se observa que las discrepancias sean siempre positivas o negativas con respecto a la predicción. Dado que esta se basa en el valor de CPU anterior, podría decirse que las discrepancias corresponden a aplicaciones donde su consumo máximo fue muy diferente entre ejecuciones. Por otro lado, en la figura 8-2 también se observa que existe el mismo comportamiento en los consumos cercanos a cero. En este caso las distancias son menores, por lo que se trata de aplicaciones con menos variación de consumos máximos entre ejecuciones.

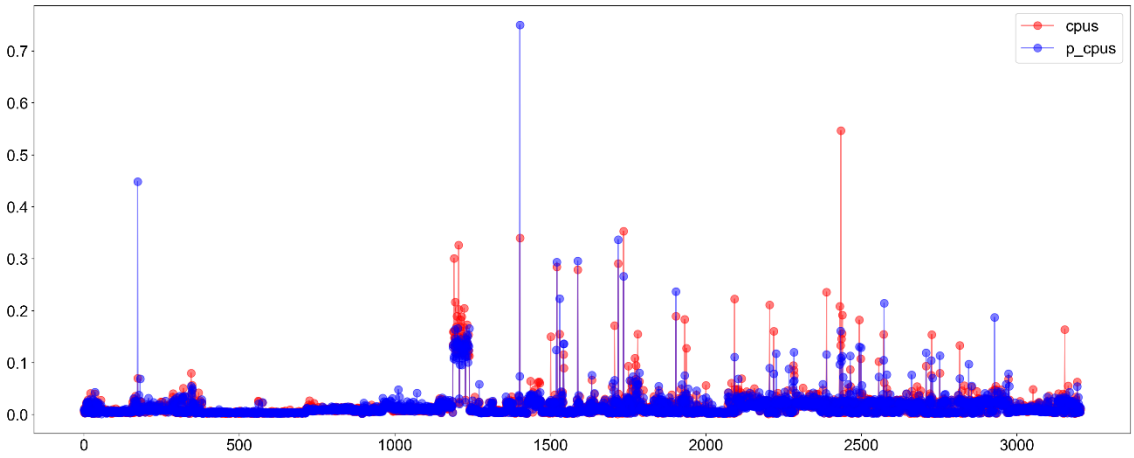

Figura 8-1. Error RMSE de CPU del modelo baseline, grano grueso

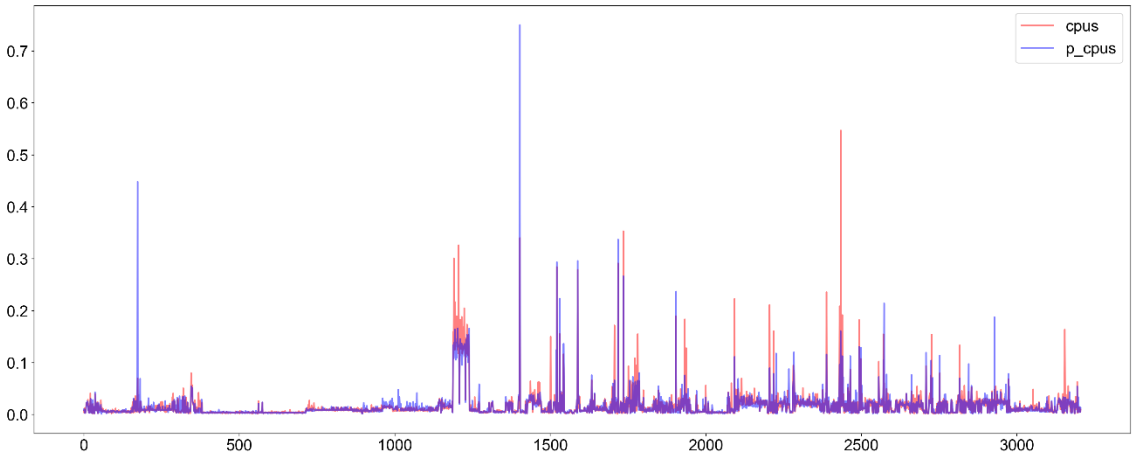

Figura 8-2. Error RMSE de CPU del modelo baseline, grano fino

Respecto a las predicciones del *offset*, las figuras 8-3 y 8-4 representan el error cometido en la predicción para todas las aplicaciones. Observando la figura 8-3, llama la atención que las mayores discrepancias entre predicción y valor real también se encuentren en torno al rango [1200, 2600]. Parece que en este tramo existen varias aplicaciones cuya duración de ejecución fue muy distinta entre lanzamientos. Sin embargo, hay un caso en el que se ve que de media su duración es elevada. Se trata de la aplicación que ocupa la posición 1402, en la gráfica se identifica fácilmente porque es el valor más elevado de la misma, tanto para *offset* como para CPU. Podría existir una relación entre la duración de una ejecución y el consumo de la misma en el sentido de que, en ejecuciones muy prolongadas, existe más probabilidad de disponer de recursos libres en la máquina donde se encuentre y poder disponer de cuotas más elevadas de CPU. Revisando la figura 8-4 también se observa lo ya descrito anteriormente con las duraciones cercanas a cero.

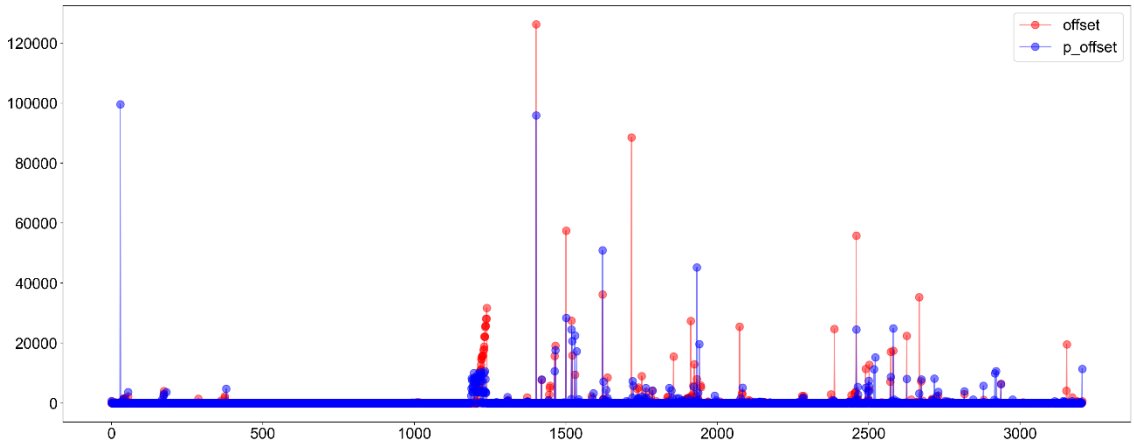

Figura 8-3. Error RMSE de offset del modelo baseline, grano grueso

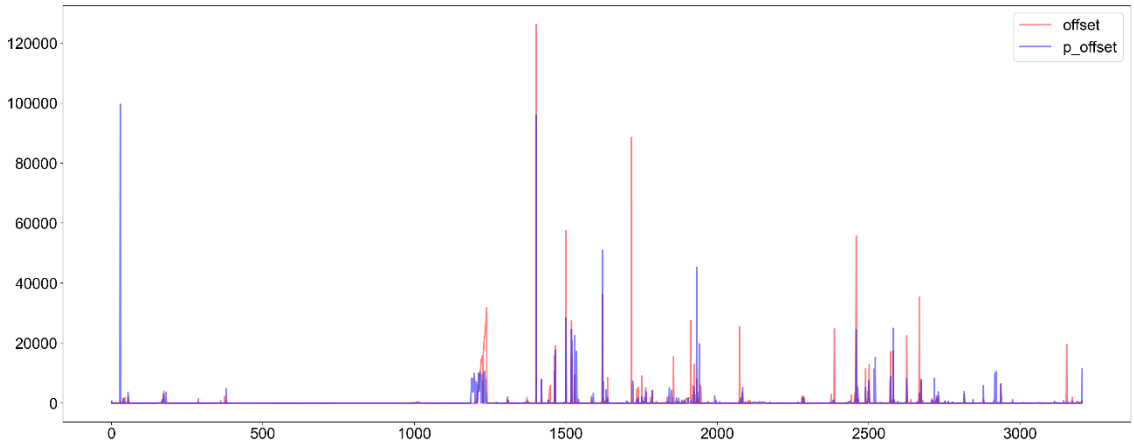

Figura 8-4. Error RMSE de offset del modelo baseline, grano fino

Una vez revisados los resultados para este modelo base, se concluye que existe cierta irregularidad tanto en duración como en consumo de CPU entre las ejecuciones de una determinada aplicación, por lo que no se espera que siempre sean las mismas. Intentar hacer una predicción del próximo lanzamiento tiene sentido a nivel de planificación de recursos ya que se espera que los valores no sean los mismos. Esta situación se da tanto en las partes altas como en las bajas de las gráficas, por lo que se trata de un comportamiento generalizado.

Por otro lado, se han observado algunos puntos que pueden ser interesantes de revisar en los modelos siguientes, que son el rango [1200, 2600] que se referenciará como rango A y el punto 1402, que es el de mayor consumo en CPU y *offset*.

8.2 Modelo de Bosque aleatorio

Las pruebas con este modelo se han realizado utilizando distintos valores para los hiperparámetros que controlan el número de árboles que componen el bosque y si se pueden utilizar repeticiones de registros en los mismos (*bootstrap*). Para el número de árboles se han establecido 4 valores diferentes: 50, 100, 200 y 400. Los lanzamientos se han realizado con y sin *bootstrap*. La tabla 8-2 muestra los valores obtenidos.

Prueba	Árboles	Bootstrap	CPU RMSE	OFFSET RMSE
1	50	NO	0.0114761937	1936.33149
2	100	NO	0.0114760944	1935.58641
3	200	NO	0.0114711621	1950.91212
4	400	NO	0.0114886727	1946.45250
5	50	SI	0.0097604949	1850.85751
6	100	SI	0.0097156184	1863.91982
7	200	SI	0.0094153135	1805.01883
8	400	SI	0.0094857640	1855.11785

Tabla 8-2. Error RMSE del modelo de bosque aleatorio obtenido en validación

De la tabla anterior se desprende que los entrenamientos con repetición de registros en la construcción de los árboles ofrecen mejores resultados que sin repetición. Por otro lado, también se observa que la reducción del error no es lineal con respecto al número de árboles utilizados. Por ejemplo, la prueba 8 tiene más árboles en su construcción que las pruebas 5, 6 y 7 y sin embargo ofrece peores resultados.

En todos los casos se mejora el resultado del modelo base pero es la configuración de la prueba 7 la que ofrece los mejores resultados: modelo con 200 árboles y *bootstrap* activo. Con la configuración anterior se entrena de nuevo el modelo y se realiza la predicción sobre el conjunto de pruebas, la tabla 8-3 muestra los resultados obtenidos.

Conf. Modelo	Árboles	Bootstrap	CPU RMSE	OFFSET RMSE
7	200	SI	0.0096490600	2613.73148

Tabla 8-3. Error RMSE del modelo de bosque aleatorio obtenido en pruebas

A nivel de CPU, el error obtenido en la predicción es similar al anterior pero a nivel de *offset* este se ha incrementado unos 800 segundos. Dado que este tipo de modelos no tienden al sobre entrenamiento y para la CPU el ajuste es bueno, es posible que existan algunos valores en el conjunto de pruebas para el *offset* que provoquen el incremento de este valor.

Las figuras 8-5 y 8-6 representan el error cometido en la predicción de consumo de CPU para todas las aplicaciones. Observando la figura 8-5 se aprecia que en esta ocasión las distancias de las aplicaciones incluidas en el rango A ([1200, 2600]), son menores que en el caso base. Por lo que se interpreta que el modelo está haciendo algo más que predecir con el último valor. Recordar que en este rango A era donde se veían las diferencias más acusadas. Respecto a la aplicación 1402, el modelo no ha sabido predecir este caso extremo, estimando un valor de casi el doble del real.

Por otro lado, la figura 8-6 muestra que existe cierta linealidad en las predicciones ya que, a nivel global, ambas líneas describen las mismas subidas y bajadas. Parece que el modelo "sabe" aproximar el consumo en función de la aplicación, sobre todo en los consumos reales por debajo de 0.05 *ncus*.

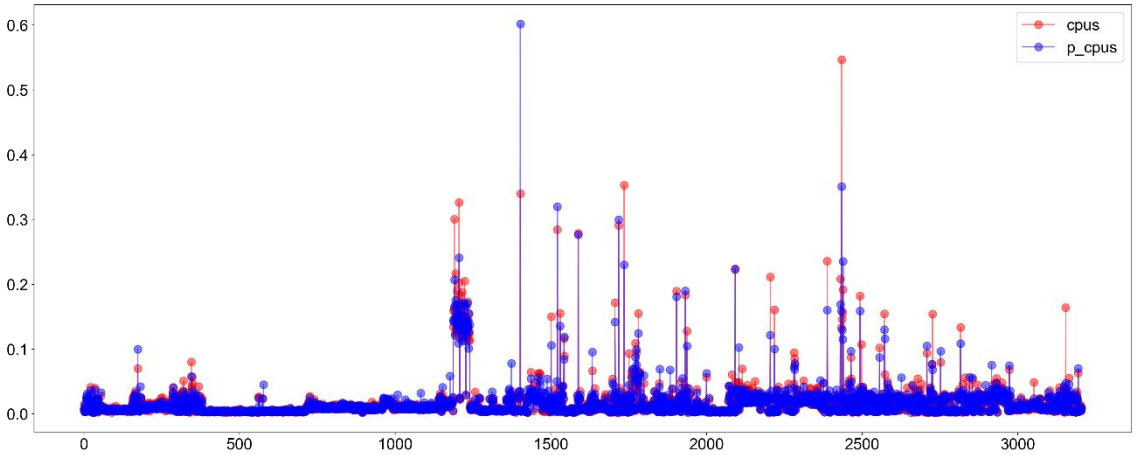

Figura 8-5. Error RMSE de CPU del modelo de bosque aleatorio, grano grueso

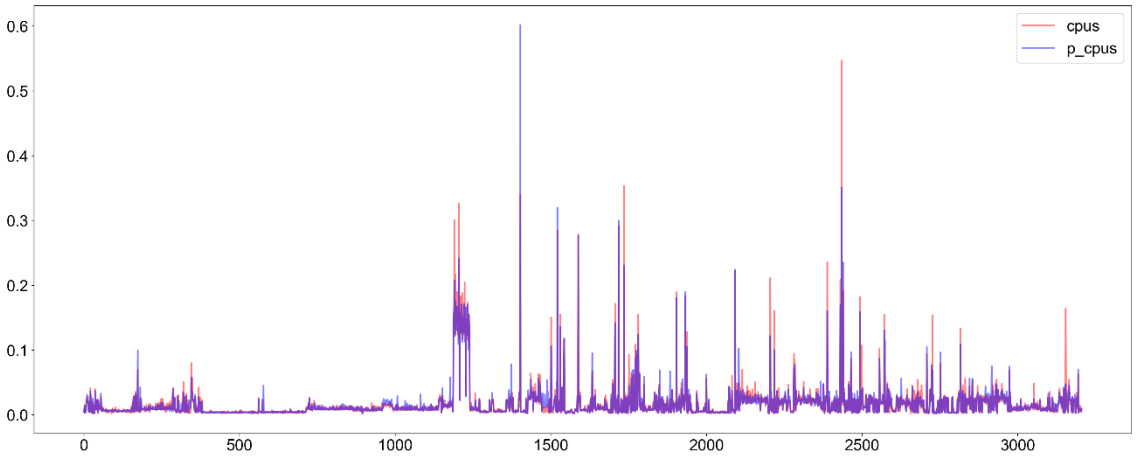

Figura 8-6. Error RMSE de CPU del modelo de bosque aleatorio, grano fino

Respecto a las predicciones del *offset*, las figuras 8-7 y 8-8 representan el error cometido en la predicción para todas las aplicaciones. La figura 8-7 también confirma que el modelo mejora en general en las ejecuciones de mayor duración, se aprecia que las distancias son más cortas. La predicción realizada sobre la aplicación 1402 tampoco es buena para el *offset*. También llama la atención una predicción realizada casi al principio del eje *X,* el valor real es muy pequeño pero la predicción es en comparación extremadamente elevada. Estos casos aportan mucha parte del error en el RMSE final, por lo que este podría ser algo menor que lo obtenido.

Respecto a la comparativa de grano fino que ofrece la figura 8-8, también se observa cierta proporción entre los valores reales y predicciones para el *offset*. La

forma de las líneas guardan similitud y parece que el modelo también intenta estimar en función de la aplicación.

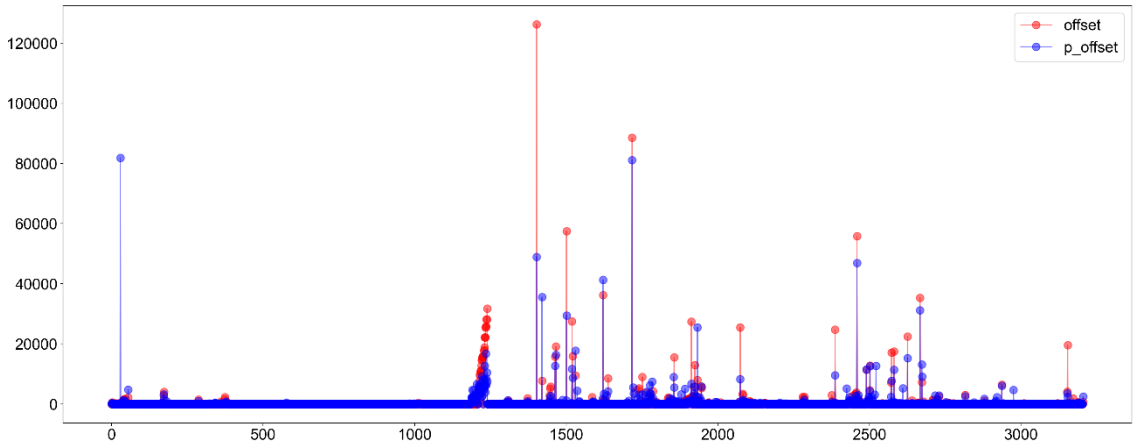

Figura 8-7. Error RMSE de offset del modelo de bosque aleatorio, grano grueso

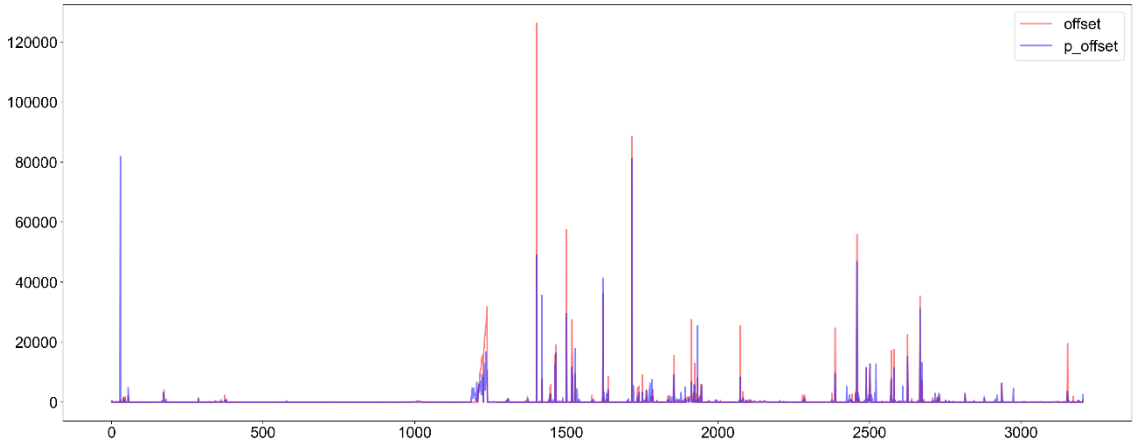

Figura 8-8. Error RMSE de offset del modelo de bosque aleatorio, grano fino

Revisados los resultados, este modelo predice mejor que el caso base comentado anteriormente. Tanto los valores de RMSE como las representaciones gráficas observadas muestran que las predicciones que realiza son más acertadas y las hace con más criterio. Parece que está teniendo en cuenta la discriminación de la aplicación a la hora de construir el bosque. Este aspecto es muy interesante ya que se está tratando un conjunto de series temporales pertenecientes a 1517 aplicaciones y que se encuentran mezcladas a lo largo de la línea temporal.

8.3 Modelo de árbol de intensificación de gradiente

Las pruebas con este modelo se han realizado utilizando distintos valores para los hiperparámetros que controlan el factor de aprendizaje (*learning_rate*) y la

profundidad de los árboles a crear. Dado que este modelo tiende al sobre entrenamiento, se hará uso del hiperparámetro *early_stopping* para detener el entrenamiento en curso y limitar así este efecto. El valor utilizado para el número de árboles a crear es de 10.000, se utiliza este valor para asegurar que el modelo no se detiene antes de tiempo y asegurar que se aplica la función de *early_stopping*.

Para el factor de aprendizaje se han establecido 3 valores diferentes: 0.1, 0.01 y 0.001. Respecto a la profundidad de los árboles, se fijan también 3 valores: 3, 6 y 12.

El hecho de fijar valores pequeños para la profundidad del árbol se debe a que este tipo de modelos se basa en estructuras de árboles muy sencillas que realizan una predicción simple y cuyo error se intenta reducir en el siguiente árbol a crear. No tiene sentido crear árboles con estructuras complejas, de ahí que los valores sean pequeños.

La tabla 8-4 muestra los valores obtenidos para las configuraciones anteriores.

Prueba	F. Aprendiz.	Profundidad	CPU RMSE	OFFSET RMSE
1	0.1	3	0.0126767288	3123.54132
2	0.1	6	0.0130640729	3531.04235
3	0.1	12	0.0134629386	3582.13377
4	0.01	3	0.0137718265	3025.25841
5	0.01	6	0.0133091667	3440.61000
6	0.01	12	0.0133948367	3040.82074
7	0.001	3	0.0138595164	2983.09001
8	0.001	6	0.0133542233	2811.04061
9	0.001	12	0.0136810667	2899.31655

Tabla 8-4. Error RMSE del modelo de árbol de intensificación de gradiente obtenido en validación

Los datos de la tabla muestran que tanto el factor de aprendizaje como la profundidad del árbol influyen en el entrenamiento del modelo. También llama la atención que en las pruebas 2, 3, y 5 los valores de *offset* son peores que los del modelo base. En el caso de la CPU parece no haber este problema ya que todas las pruebas mejoran el caso básico.

Por otro lado, no hay una única configuración que sea la mejor tanto para CPU como para *offset.* Es decir, la prueba 1 es la que mejor resultados ofrece para la CPU y la prueba 8 para el *offset*. Dado que en las dos pruebas la diferencia en la predicción de

la CPU es del 5% y del 10% para el *offset*, se seleccionará el modelo 8 como el que mejores resultados ofrece: Factor de aprendizaje 0.001 y profundidad de árbol 6.

Con la configuración anterior se entrena de nuevo el modelo y se realiza la predicción sobre el conjunto de pruebas, la tabla 8-5 muestra los resultados obtenidos.

Conf. Modelo	F. Aprendiz.	Profundidad	CPU RMSE	OFFSET RMSE
8	0.001	6	0.0105990513	3095.25206

Tabla 8-5. Error RMSE del modelo de árbol de intensificación de gradiente obtenido en pruebas

Lo primero que llama la atención es que el error en la CPU es inferior al obtenido durante la fase de pruebas, normalmente el error suele ser superior pero no siempre tiene por qué ser así. Por otro lado, el valor obtenido para el *offset* sí que es superior al obtenido en la fase de pruebas, habiendo una diferencia relativamente pequeña de unos 300 segundos. Ambos errores indican que el entrenamiento del modelo es correcto y que no existe sobre ajuste en el mismo. Parece que el hiperparámetro de *early_stopping* ha funcionado correctamente.

Las figuras 8-9 y 8-10 representan el error cometido en la predicción de consumo de CPU para todas las aplicaciones. Observando la figura 8-9 se aprecia que de modo general la predicción se queda por debajo del valor real en los casos donde los consumos son más elevados como por ejemplo en el rango A ([1200, 2600]). En este tramo se observa que los valores reales quedan mayoritariamente por encima de las predicciones, parece que el modelo tiende a predecir en base a valores medios antes que a extremos. Este hecho se confirma observando la figura 8-10, donde la predicción ahora está por encima de los valores reales. Es decir, de algún modo el modelo tiende a predecir un valor intermedio entre valores extremos, tanto por arriba como por abajo. Por último, comentar que este modelo sí ha estimado bastante bien la predicción de la aplicación 1402 que se estaba siguiendo de referencia.

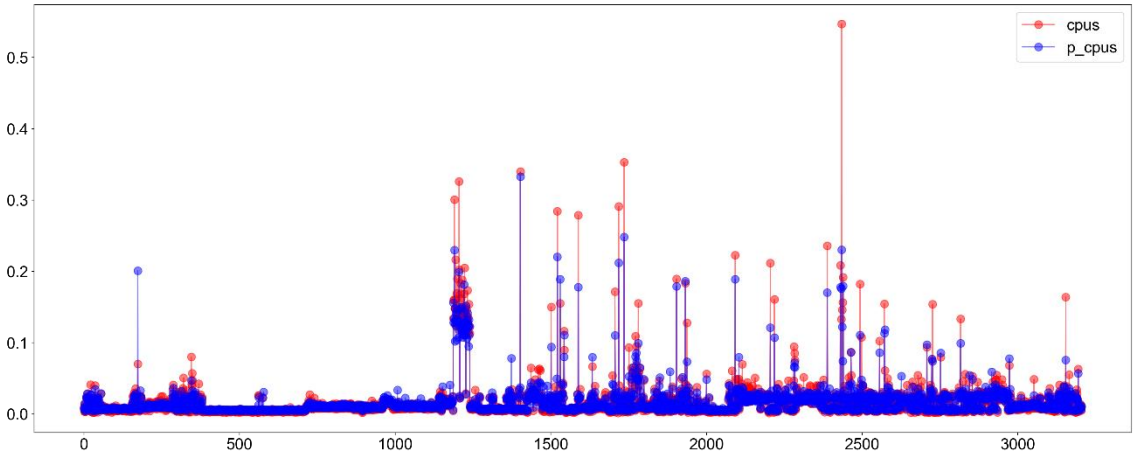

Figura 8-9. Error RMSE de CPU del modelo de árbol de intensificación de gradiente, grano grueso

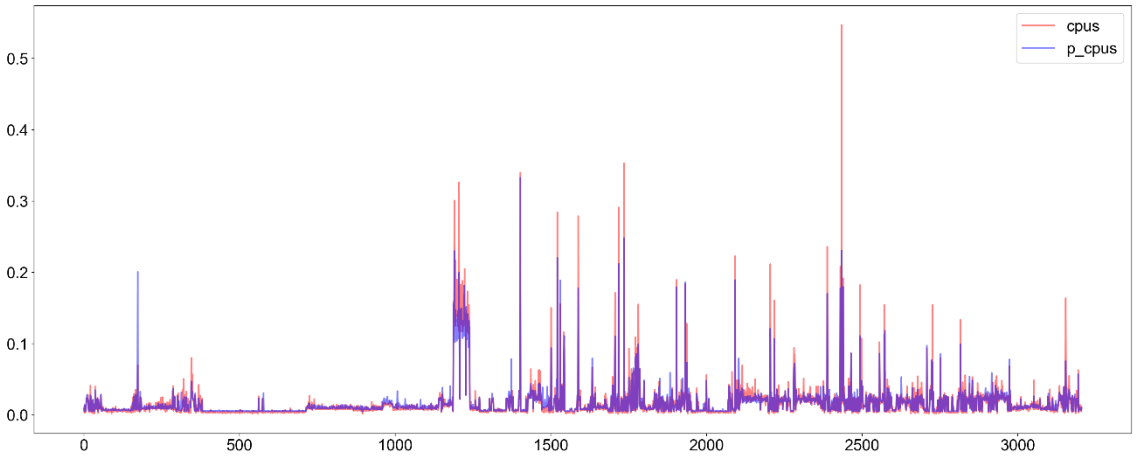

Figura 8-10. Error RMSE de CPU del modelo de árbol de intensificación de gradiente, grano fino

Respecto a las predicciones del *offset*, las figuras 8-11 y 8-12 representan el error cometido en la predicción para todas las aplicaciones. La figura 8-11 también confirma que la predicción se queda por debajo del valor real en los casos donde los consumos son más elevados como por ejemplo en el rango A ([1200, 2600]). Sin embargo, se aprecia que de forma general las distancias son más pronunciadas que en el caso de la CPU. Para el *offset* el modelo predice peor que para la CPU en las ejecuciones de más duración. También se observa que la predicción de la aplicación 1402 no es acertada, lo que indica que existe independencia entre los dos modelos entrenados a la hora de construir su estructura de predicción interna a pesar de utilizar el mismo conjunto de datos como input.

Observando la figura 8-12, también se confirma lo ya comentado respecto a predecir por encima de valores reales más pequeños. Parece que para el *offset* el modelo también predice buscado un punto medio.

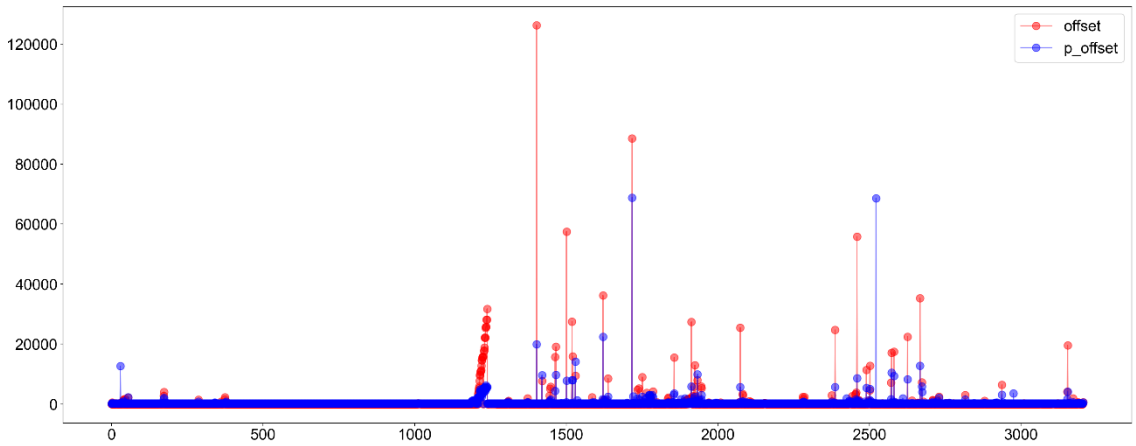

Figura 8-11. Error RMSE de offset del modelo de árbol de intensificación de gradiente, grano grueso

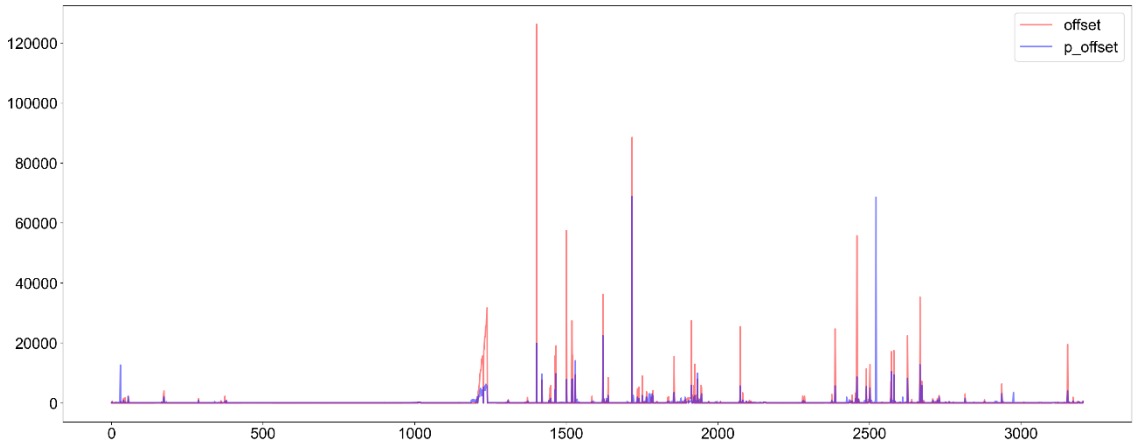

Figura 8-12. Error RMSE de offset del modelo de árbol de intensificación de gradiente, grano fino

Revisados los resultados, este modelo predice mejor que el caso base. Tanto los valores de RMSE como las representaciones gráficas observadas muestran que las predicciones que realiza son más acertadas y las hace con criterio. Observando las gráficas de grano fino, también se aprecia que ambas líneas describen las mismas subidas y bajadas.

El modelo es conservador a la hora de predecir valores extremos y podría utilizarse en aplicaciones que presenten una variabilidad más ajustada en su duración y consumo de CPU.

8.4 Modelo de Red Neuronal Recurrente LSTM

Las pruebas con este modelo se han realizado utilizando distintos valores para los hiperparámetros que controlan el factor de aprendizaje, el número de capas y el número de celdas que estas contienen. Dado que este modelo tiende al sobre entrenamiento, se hará uso de la clase *EarlyStopping* para detener el entrenamiento en curso y limitar así este efecto.

Se establecen 2 opciones para todos los hiperparámetros configurables:

- Factor de aprendizaje: 0.01 y 0.001
- Modelo con 1 y 2 capas LSTM ocultas.
- Número de celdas LSTM por capa 20 y 40.

La tabla 8-6 muestra los valores obtenidos para las configuraciones anteriores.

Prueba	Núm. Capas	Núm. celdas	F. Aprendiz.	CPU RMSE	OFFSET RMSE
1	1	20	0.01	0.0249379019	3125.81873
2	1	20	0.001	0.0239245455	3132.28305
3	1	40	0.01	0.0249864629	3129.80260
4	1	40	0.001	0.0244971301	3045.40554
5	2	20	0.01	0.0245948422	3122.45139
6	2	20	0.001	0.0242547868	3128.27294
7	2	40	0.01	0.0241789909	3115.19728
8	2	40	0.001	0.0238236686	3122.25748

Tabla 8-6. Error RMSE del modelo de Red Neuronal Recurrente LSTM obtenido en validación

Lo primero que se observa de estos resultados es que todos predicen el consumo de CPU peor que el modelo base. Para el *offset* las estimaciones son muy parecidas y todas algo mejor que el modelo básico. Con estos valores ya se observa que el modelo no está aprendiendo correctamente la lógica interna que pueda tener el conjunto de datos entrenados y de ahí que los errores de predicción sean elevados.

De todas las pruebas, la versión que parece hacerlo mejor es la número 8. Su error de CPU es, por poco, el más pequeño que el resto y su error de *offset* es el tercero mejor. Dado que los que tienen mejor error en *offset* que este no lo tienen en CPU, se

decide elegir el modelo 8 como el que mejores resultados ofrece: 2 capas LSTM, 40 celdas por capa y factor de aprendizaje 0.001.

Observando los valores de error, da la sensación de que cuanto más capas, celdas y factor de aprendizaje se tenga menor es el error cometido. Sin embargo, en las pruebas previas para la selección de valores se probaron configuraciones con valores más elevados y los resultados no eran mejor que los que ofrece la prueba 8. Con la configuración de esta versión se entrena de nuevo el modelo y se realiza la predicción sobre el conjunto de pruebas, la tabla 8-7 muestra los resultados obtenidos.

Conf. Modelo	Núm. Capas	Núm. celdas	F. Aprendiz.	CPU RMSE	OFFSET RMSE
8	2	40	0.001	0.0259791460	3860.26842

Tabla 8-7. Error RMSE del modelo de Red Neuronal Recurrente LSTM obtenido en pruebas

Con los resultados obtenidos se entiende que este modelo no sirve para predecir ni consumo de CPU y *offset*. Los errores son superiores al modelo básico y no tiene sentido utilizar este cuando el básico ofrece errores menores.

En las figuras 8-13 y 8-14 se observa perfectamente lo que ocurre con la CPU. El modelo se limita a aplicar siempre una estimación muy acotada en un rango muy pequeño. En la figura 8-13 se observa como el rango A ([1200, 2600]) prácticamente es plano y llega casi al final del eje X. Si se observa la figura 8-14 se aprecia que la predicción se está acotando a unos valores máximos y mínimos.

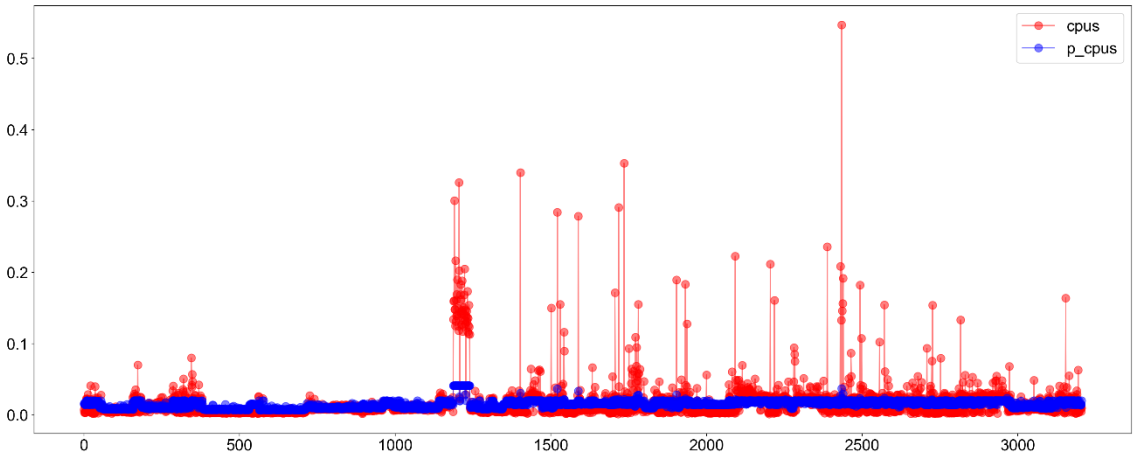

Figura 8-13. Error RMSE de CPU del modelo de Red Neuronal Recurrente LSTM, grano grueso

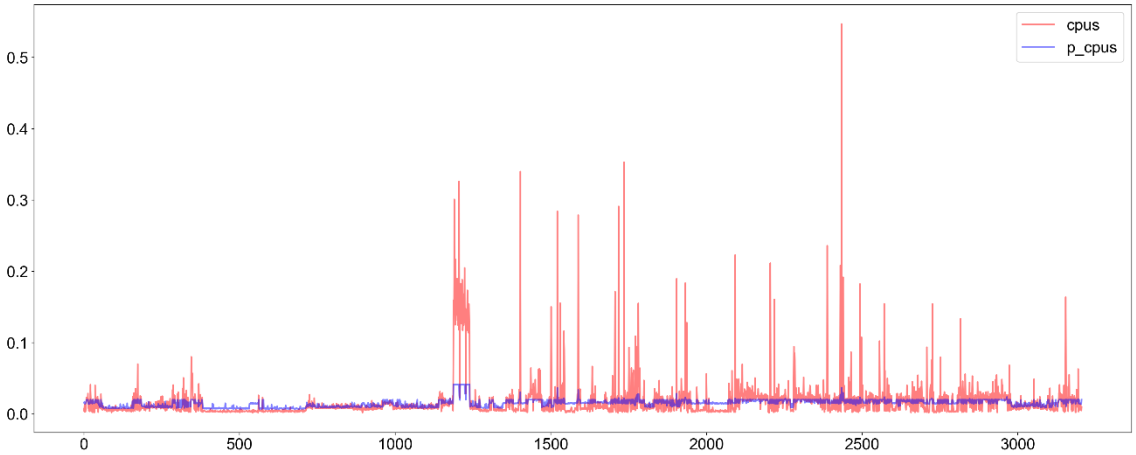

Figura 8-14. Error RMSE de CPU del modelo de Red Neuronal Recurrente LSTM, grano fino

Respecto al offset ocurre algo similar, las figuras 8-15 y 8-16 muestran que las estimaciones también son bastante planas. En la figura 8-15 se observa que no hay aproximaciones a las duraciones más elevadas y la figura 8-16 muestra que también existen cotas máximas y mínimas muy ajustadas.

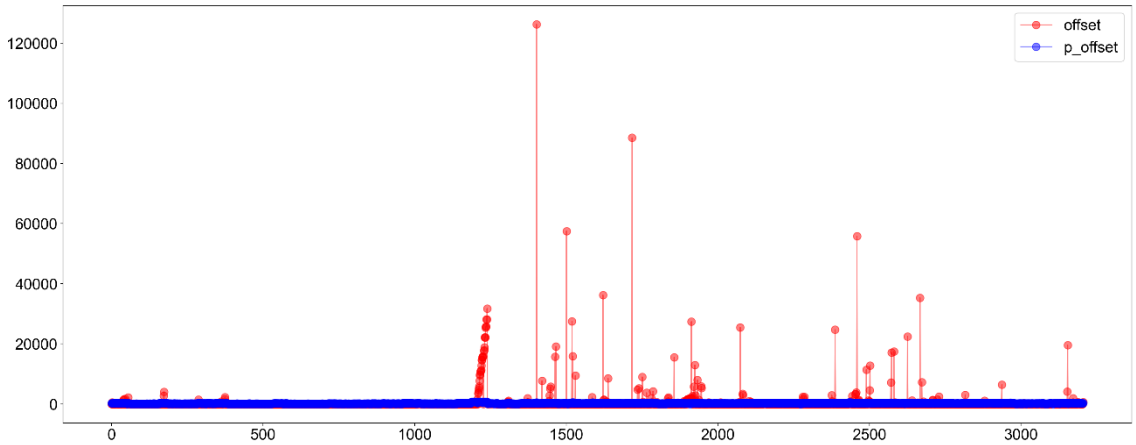

Figura 8-15. Error RMSE de offset del modelo de Red Neuronal Recurrente LSTM, grano grueso

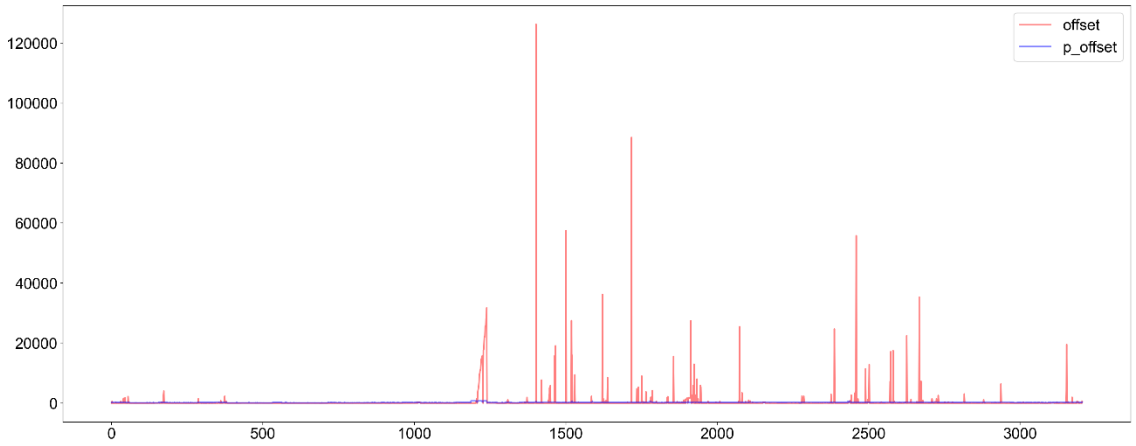

Figura 8-16. Error RMSE de offset del modelo de Red Neuronal Recurrente LSTM, grano fino

Revisados los resultados, este modelo no es adecuado para la estimación de recursos que se pretende realizar. Las predicciones no se ajustan bien a las fluctuaciones que puedan producirse por parte de las aplicaciones. La tendencia observada es siempre la misma, un rango de valores acotados que fluctuan a lo largo de todas las predicciones.

Entrando en la estructura del modelo, se probaron entrenamiento con otras regularizaciones además de *relu* y los resultados no eran mejores. También se probaron diferentes números de capas y neuronas LSTM sin éxito. Parece que el estado interno que el modelo va "memorizando" a medida que se produce el entrenamiento no tiene el alcance suficiente como para amoldarse a la variedad de aplicaciones que están informadas en el conjunto de datos.

8.5 Modelo de Red Neuronal Convolucional

Las pruebas con este modelo se han realizado utilizando distintos valores para los hiperparámetros que controlan el factor de aprendizaje, el ratio de pérdida de las capas de *dropout* y el número de filtros de la capa de convolución. Dado que este modelo tiende al sobre entrenamiento, se hará uso de la clase *EarlyStopping* para detener el entrenamiento en curso y limitar así este efecto.

Se establecen 2 opciones para todos los hiperparámetros configurables:

- Factor de aprendizaje: 0.01 y 0.001
- Ratio de pérdida (dropout): 0.25 y 0.50.
- Número filtros de la capa de convolución: 64 y 128.

La tabla 8-8 muestra los valores obtenidos para las configuraciones anteriores.

Prueba	Núm. Filtros	Dropout	F. Aprendiz.	CPU RMSE	OFFSET RMSE
1	64	0.25	0.01	0.0122242921	2975.16973
2	64	0.25	0.001	0.0113421130	2382.29694
3	64	0.50	0.01	0.0178495712	3078.96804
4	64	0.50	0.001	0.0123459832	2968.65338
5	128	0.25	0.01	0.0322292511	3165.72202
6	128	0.25	0.001	0.0110264929	2458.26004
7	128	0.50	0.01	0.0187553233	3074.70251
8	128	0.50	0.001	0.0127206731	3069.87245

Tabla 8-8. Error RMSE del modelo de Red Neuronal Convolucional obtenido en validación

Los resultados de las pruebas realizadas mejoran en general los obtenidos con el caso base. El valor de CPU de la prueba 7 es el único que ligeramente supera al caso básico. Para el *offset* todos se encuentran por debajo del valor básico de referencia.

Por otro lado, los datos de la tabla muestran que el factor de aprendizaje de 0.001 mejora el rendimiento del modelo a igualdad de valores en el resto de hiperparámetros. También se observa que, de media, los resultados obtenidos con 64 filtros son mejores que con 128. Además, parece que la regularización de 0.25 también ayuda a reducir más el error ya que, a igualdad de valores en el resto de hiperparámetros, el valor de los errores son ligeramente inferiores.

Además, no hay una única configuración que sea la mejor tanto para CPU como para *offset.* Es decir, la prueba 6 es la que mejor resultados ofrece para la CPU y la prueba 2 para el *offset*. Dado la prueba 2 contiene los valores que mejoran la reducción del error comentados en el párrafo anterior, se seleccionará el modelo 8 como el que mejores resultados ofrece: Factor de aprendizaje 0.001, 64 filtros en la capa de convolución y ratio de pérdida de 0.25. Con la configuración anterior se entrena de nuevo el modelo y se realiza la predicción sobre el conjunto de pruebas, la tabla 8-9 muestra los resultados obtenidos.

Conf. Modelo	Núm. Filtros	Dropout	F. Aprendiz.	CPU RMSE	OFFSET RMSE
2	64	0.25	0.001	0.0131250819	3118.18975

Tabla 8-9. Error RMSE del modelo de Red Neuronal Convolucional obtenido en pruebas

Las predicción sobre el conjunto de pruebas ha elevado unas milésimas el error de CPU, margen que puede considerarse aceptable. Para el *offset* la diferencia es bastante mayor, en torno a 800 segundos. Sigue siendo mejor que el caso base pero es posible que existan casos donde la predicción es extremadamente errónea.

Las figuras 8-17 y 8-18 representan el error cometido en la predicción de consumo de CPU para todas las aplicaciones. Observando la figura 8-17 se aprecia que de modo general la predicción se queda por debajo del valor real en los casos donde los consumos son más elevados como por ejemplo en el rango A ([1200, 2600]). Además, las distancias observadas en este rango no son ajustadas, por lo que mucha parte del error de la predicción proviene de este tramo. Por otro lado, comentar que este

modelo no ha estimado bien la predicción de la aplicación 1402 que se estaba siguiendo de referencia.

Observando la figura 8-18, también se aprecia que la predicción queda de modo general por debajo del valor real. Se tiene por tanto que, en conjunto, este modelo tiende a predecir por debajo del valor real pero siguiendo las variaciones que puedan experimentar las aplicaciones. Esta apreciación se debe a la forma que describen las dos líneas a lo largo del eje temporal, ambas experimentan de modo similar las subidas y bajadas.

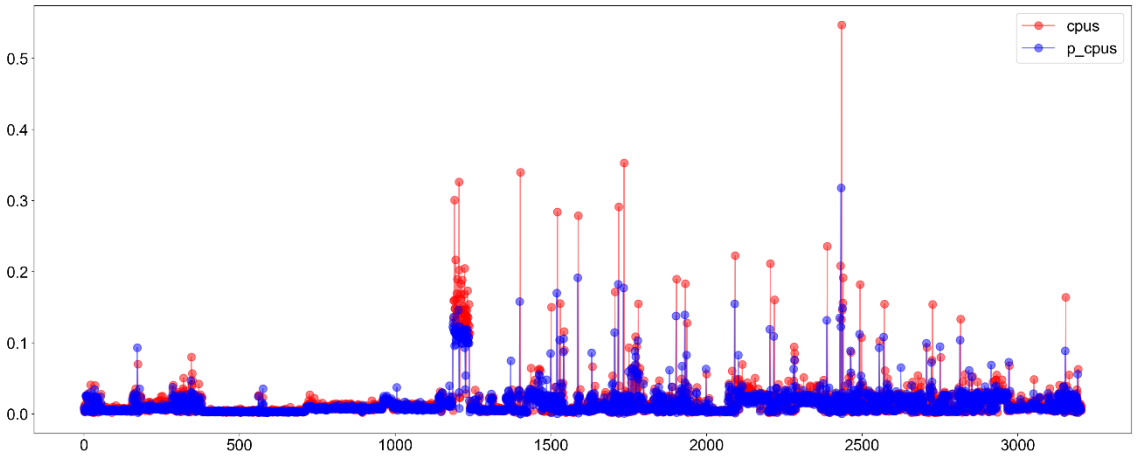

Figura 8-17. Error RMSE de CPU del modelo de árbol de Red Neuronal Convolucional, grano grueso

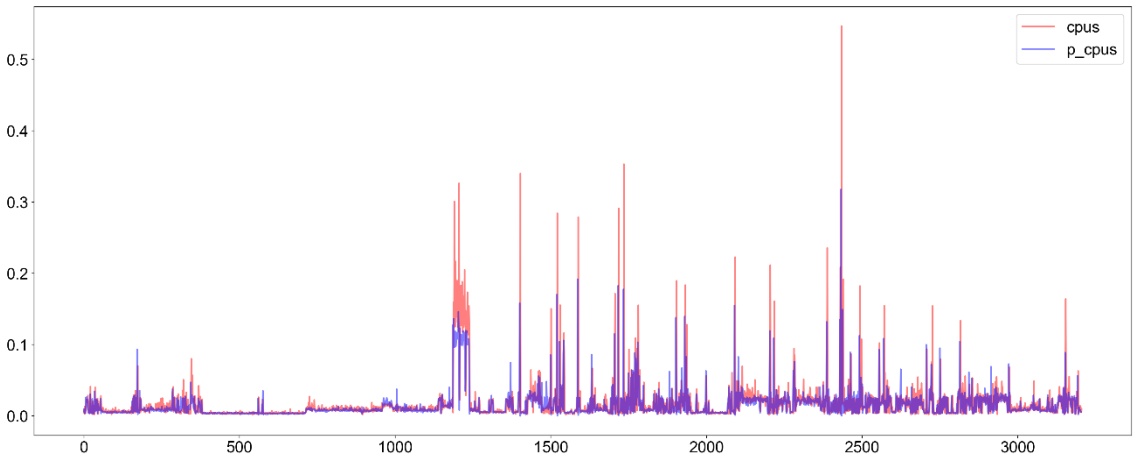

Figura 8-18. Error RMSE de CPU del modelo de Red Neuronal Convolucional, grano fino

Respecto a las predicciones del *offset*, las figuras 8-19 y 8-20 representan el error cometido en la predicción para todas las aplicaciones. En este caso se observa que las aproximaciones entre valores son algo más ajustadas que con la CPU en el rango A

([1200, 2600]). Sin embargo, hay algunas predicciones que son extremadamente erróneas, como la del grupo situado en torno al punto 1200 o la del punto 1402, en estos casos de las predicciones están muy separadas del valor real. En el resto de la gráfica se aprecia que la tendencia es a predecir valores pequeños. Sin embargo, al revisar la gráfica 20 se observa que las predicciones de estos valores pequeños son acertadas. Podría decirse que las diferencias en la predicción de los valores mayores están penalizando mucho en el incremento del error para el *offset*.

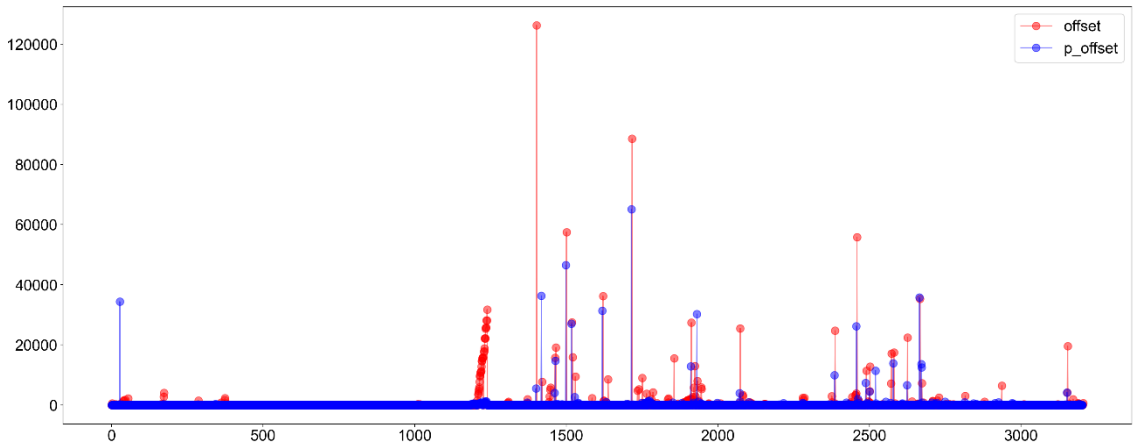

Figura 8-19. Error RMSE de offset del modelo de Red Neuronal Convolucional, grano grueso

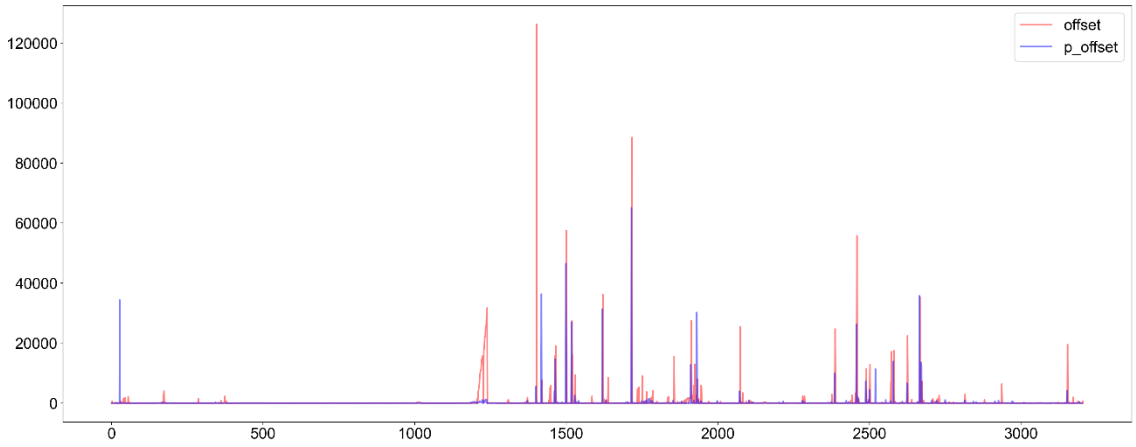

Figura 8-20. Error RMSE de offset del modelo de Red Neuronal Convolucional, grano fino

Revisados los resultados, este modelo obtiene mejores valores que el caso base y en teoría debería predecir mejor que este. La parte de la CPU ha dado mejores resultados que la del *offset*, donde parece que el modelo no termina de predecir bien para aquellos valores que son más elevados. Sin embargo, los resultados tampoco son tan buenos si se tiene en cuenta todas las adaptaciones que hay que realizar sobre el

conjunto de datos para que el modelo pueda interpretarlo y el tiempo que toma entrenar una red de este tipo con respecto al resto de modelos.

Quizás si se hubiesen creado secuencias con más registros los resultados habrían mejorado pero, dada la naturaleza del conjunto de datos, donde pueden existir paralelizados de aplicaciones con pocos registros, se corre el riesgo de perder información relacionada con esos casos.

8.6 Resumen

En este capítulo se han revisado los resultados de las pruebas obtenidas en el entrenamiento y predicción de los modelos. Para cada uno se ha detallado las combinaciones de hiperparámetros utilizadas y comentado las diferencias observadas durante la fase de predicciones a partir del conjunto de datos de validación. Las mejor combinación se ha utilizado para entrenar de nuevo al modelo y predecir los valores del conjunto de pruebas. A partir de estos resultados se han generado las gráficas comparativas que han aportado información adicional acerca de la precisión y comportamiento de cada modelo. Si bien las valoraciones han sido individuales, el siguiente capitulo realizará la comparativa entre modelos.

9 Evaluación de los resultados obtenidos

En este capítulo se van a comparar los resultados obtenidos en las predicciones realizadas sobre el conjunto de pruebas por parte de los modelos de aprendizaje automático. Como ya se describió en el capítulo anterior, cada modelo realiza esta predicción entrenado con la mejor configuración obtenida durante una fase previa de pruebas, por lo que se supone que los valores de error RMSE obtenidos son los mejores que ese modelo puede aportar para el conjunto de datos comentado.

Las predicciones se realizan sobre dos parámetros, por lo que cada modelo entrega dos valores. Uno el consumo máximo de CPU que se espera durante la ejecución de una determinada aplicación y otro el momento de la ejecución en segundos donde se va a producir.

9.1 Resultados a nivel de CPU

La figura 9-1 muestra los errores RMSE obtenidos por cada uno de los modelos. En color más claro y siendo el primero por la izquierda, se encuentra el valor del modelo base, que ha sido la referencia utilizada para valorar el rendimiento del resto de modelos.

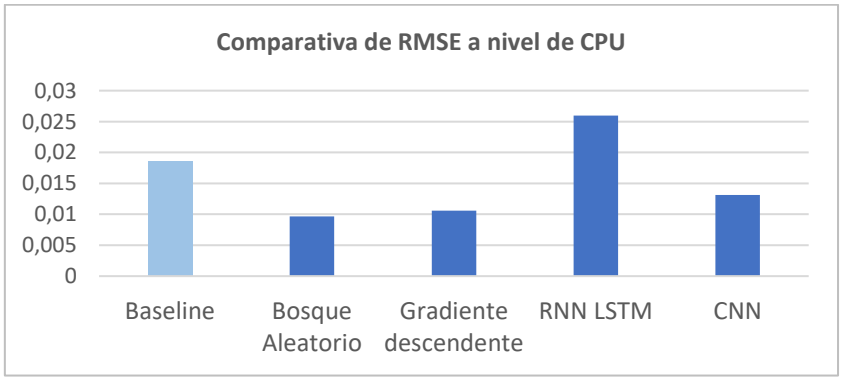

Figura 9-1. Error RMSE para la CPU obtenido en pruebas por todos los modelos

En la figura se observa que todos los modelos mejoran el resultado del caso base a excepción de la RNN-LSTM. Durante la fase de pruebas se observó que el rendimiento de este modelo era bastante deficiente y no llegaba a interpretar la estructura interna que pudiese tener el conjunto de datos. Se probaron varias configuraciones sin éxito y puede decirse que el modelo no es válido para la predicción buscada.

Respecto a la CNN, esta mejora los resultados del caso base en un 29,32% pero en las pruebas se vio que el modelo no era muy preciso en la predicción de consumos más elevados y que tenía una tendencia generalizada a predecir por debajo del valor real. Además de que requiere una adaptación extra a la entrada del conjunto de datos y de que sus tiempos de entrenamiento son de media más elevados que en el resto de casos, hay dos modelos que en principio son más simples y que han ofrecido mejores resultados que este, por lo que la CNN tampoco sería la opción a elegir para el objetivo que se pretende.

Los modelos de bosque aleatorio y gradiente descendente son los que mejores resultados han obtenido, mejorando al caso base en un 48,04% y un 42,92% respectivamente. Aunque la construcción es diferente, los dos se basan en la creación de árboles de decisión intermedios que puede ser la clave del rendimiento conseguido. El conjunto de datos es una mezcla de series temporales pertenecientes a distintas aplicaciones y parece que, de algún modo, los árboles son capaces de filtrar estas series. Ambos han ajustado bien los puntos del rango A que se tomó como referencia e incluso el modelo de gradiente descendente fue el único que hizo una precisión acertada del punto 1402, que también se tomó como referencia por ser un valor extremo. A nivel de error RMSE, el bosque aleatorio es el que mejores resultados obtiene pero a nivel de CPU ambos modelos han ajustado las predicciones de modo parecido.

La tabla 9-1 muestra los valores de RMSE para la CPU obtenidos por los modelos así como los porcentajes de rendimiento relativo comentados.

Modelo	CPU RMSE	% de mejora
Baseline	0,01857036	0,00
Bosque Aleatorio	0,00964906	48,04
Gradiente descendente	0,01059905	42,92
RNN LSTM	0,02597915	-39,90
CNN	0,01312508	29,32

Tabla 9-1. Error RMSE para la CPU obtenido en pruebas por todos los modelos

9.2 Resultados a nivel de OFFSET

La figura 9-2 muestra los errores RMSE obtenidos por cada uno de los modelos. En color más claro y siendo el primero por la izquierda, se encuentra el valor del modelo base, que ha sido la referencia utilizada para valorar el rendimiento del resto de modelos.

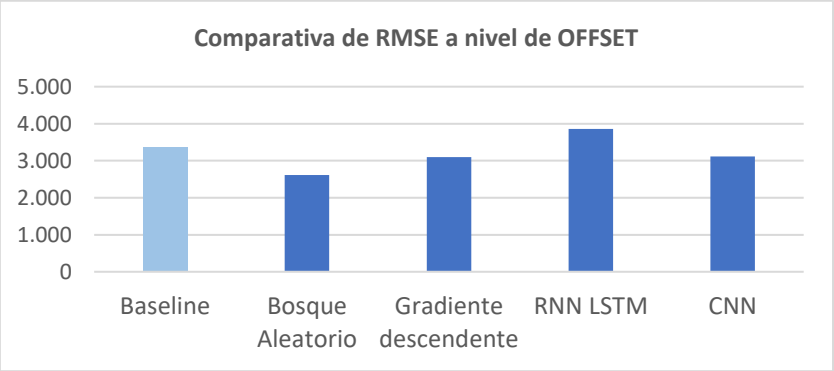

Figura 9-2. Error RMSE para el offset obtenido en pruebas por todos los modelos

La clasificación de rendimiento es la misma que la vista anteriormente pero, en general, las diferencias de mejora con respecto al modelo base no son tan evidentes como ocurría con la CPU. El error cometido por el modelo base es considerable y el hecho que el resto de modelos no mejoren a este de forma sustancial es indicativo de que las predicciones para el *offset* han sido más complejas de obtener que para la CPU.

El modelo RNN-LSTM es el único que no consigue mejorar el caso base y aplica lo ya comentado en el apartado anterior, durante la fase de pruebas se observó que el rendimiento de este modelo era bastante deficiente y no llegaba a interpretar la estructura interna que pudiese tener el conjunto de datos.

El modelo CNN ha reducido el error del caso base en un 7,22%, la mejora no es relevante si se tiene en cuenta que ha sido uno de los modelos más complejos de construir y que hay opciones más simples que lo superan.

Los modelos de bosque aleatorio y gradiente descendente son los que mejores resultados han obtenido también para el *offset*. Pero en este caso los rendimientos no son tan sustanciales, ya que solo mejoran al caso base en un 22,23% y un 7,90% respectivamente.

La tabla 9-2 muestra los valores de RMSE para el *offset* obtenidos por los modelos así como los porcentajes de rendimiento relativo comentados.

Modelo	OFFSET RMSE	% de mejora
Baseline	3.361	0,00
Bosque Aleatorio	2.614	22,23
Gradiente descendente	3.095	7,90
RNN LSTM	3.860	-14,86
CNN	3.118	7,22

Tabla 9-2. Error RMSE para el offset obtenido en pruebas por todos los modelos

Con objeto de identificar el motivo de estas deficiencias en la predicción, en un primer momento se pensó que el hecho de que se estuviesen entrenando los modelos para predecir dos variables objetivos podría estar provocando la mejora de una variable en detrimento de la otra. Sin embargo, para el modelo de gradiente descendente los entrenamientos obligatoriamente había que hacerlos por separado y sus resultados no han sido mejores que los del árbol aleatorio, que sí fue entrenado en conjunto. Además, se hicieron pruebas con el resto de modelos entrenando un solo conjunto y los valores obtenidos eran similares, no apreciándose mejoras.

Por otro lado, el motivo puede venir de la distribución en sí que tiene el conjunto de datos para esta variable objetivo. La tabla 9-3 muestra algunos valores estadísticos pertenecientes al conjunto objetivo utilizado para la validación durante las pruebas.

Media	Desv. est.	Min	0,25	0,5	0,75	0,9	0,95	Max
346,167498	3146,25907	0	1	2	9	44,5	434,25	99616

Tabla 9-3. Valores estadísticos del conjunto objetivo de validación

Los datos de la tabla muestran que el conjunto tiene una desviación estándar muy elevada con respecto al valor medio, por lo que existe una variabilidad considerable en este conjunto que podría dificultar el aprendizaje de los modelos. Por otro lado, se observa que el 90% de los registros informan un *offset* por debajo de los 45 segundos. Lo que podría explicar por qué algunos modelos daban unas predicciones más ajustadas en los valores reales más pequeños que en los más elevados. Es decir, se habían especializado sobre un conjunto de datos donde aproximadamente el 90% de los registros recibidos en su entrenamiento eran valores por debajo de 45 segundos.

Sin embargo, el 10% restante son registros que también se han dado en su entorno real y prescindir de ellos para mejorar los resultados de RMSE sería incorrecto.

Finalmente, dado que el modelo de bosque aleatorio es el que mejores resultados ha obtenido tanto en la predicción de la *CPU* como del *offset*, se considera que este modelo sería el más idóneo para las predecir el consumo de recursos sobre este conjunto de datos.

9.3 Resumen

En este capítulo se han comparado los resultados obtenidos por los modelos de aprendizaje en sus predicciones sobre el conjunto de pruebas. Además, se han comparado los valores de modo relativo a los del caso base y se han clasificado los modelos por orden de rendimiento. Por otro lado, se ha revisado las causas que pueden estar detrás de la deficiencia de precisión encontrada por el *offset* y finalmente se ha seleccionado al modelo de bosque aleatorio como el mejor candidato para predecir el consumo de recursos sobre el conjunto de datos seleccionado.

10 Conclusiones y líneas futuras

10.1 Conclusiones

Las fases llevadas a cabo para la consecución de este trabajo han tenido como objetivo principal obtener un modelo de aprendizaje automático que, entrenado bajo el paradigma de las series temporales, tenga la capacidad de predecir los consumos futuros de un conjunto de aplicaciones.

Para llegar a este objetivo se han tenido que seguir una serie de etapas que han comenzado con el análisis de un conjunto de datos complejo, basado en datos reales y con el suficiente nivel de detalle como para que las predicciones conseguidas por parte del modelo puedan aplicarse a un entorno real.

Este conjunto de datos se encontraba alojado en *BigQuery*, una plataforma de almacenamiento de datos en la nube. Siendo necesario estudiar primero la gestión de esta plataforma para poder realizar el primer análisis del conjunto de datos y determinar cómo amoldarlo para el objetivo del trabajo. Se trataba de un conjunto con un tamaño inicial de 2.4TB de información. Tras estudiar su estructura completa de tablas y el contenido de las mismas se consiguió reducir el conjunto a 6GB de tamaño, que contenía los datos realmente importantes para el trabajo. Esta reducción permitió poder seguir trabajando en local y evitar los costes asociados que implica utilizar *BigQuery*.

Una vez en local hubo que analizar de nuevo el conjunto de datos para adaptarlo a un formato que permitiese entrenar a los modelos mediante series temporales. Hubo que añadir más campos al conjunto, ya que el único valor real del que se disponía era el consumo de CPU. El resto fueron valores calculados a partir de este valor y del *timestamp*.

Con el modelo adaptado fue el turno de los modelos de aprendizaje. A partir de la bibliografía y recursos de Internet se identificaron los modelos candidatos que podrían dar mejores resultados utilizando series temporales. Hubo que realizar numerosas pruebas hasta determinar cuáles serían los rangos de valores idóneos para configurar los hiperparámetros de cada modelo y realizar las pruebas.

Entender cómo estaba funcionando cada modelo y comprobar si los entrenamientos que estaban realizando eran correctos fue una tarea que tomó bastante tiempo. Tras comprobar que los resultados que se obtenían eran correctos se pudo entonces evaluar el rendimiento de cada uno.

De todos los modelos propuestos, el que mejor resultados ofreció en general fue el de bosque aleatorio. Sorprende que el modelo más básico de configurar y rápido de entrenar sea el que mejor rendimiento ofreció. Este hecho corrobora lo que normalmente aparece en la bibliografía relacionada, que no se debe descartar a un modelo de aprendizaje por su simplicidad.

De las dos variables que se plantearon como predicción, se obtenían mejores resultados para la CPU que para el *offset*. Tras revisar las posibles causas de esta falta de ajuste, se determinó que podía deberse al elevado porcentaje de registros con un valor pequeño de *offset* incluido en el conjunto de datos. El modelo se había especializado en este gran porcentaje de registros.

Con lo anterior, puede afirmarse que el objetivo del trabajo se ha cumplido. Ya que el producto final obtenido es capaz de predecir con cierta precisión cual será el consumo máximo de CPU de la próxima ejecución así como del momento en el que puede producirse.

10.2 Líneas futuras

Para el trabajo solo se ha tenido en cuenta el consumo de CPU reportado por las aplicaciones en ejecuciones anteriores. Una línea de revisión interesante sería incorporar más conocimiento del dominio al evento como por ejemplo la disponibilidad de recursos de la maquina donde se ejecuta la aplicación en el momento de su lanzamiento.

También hay un porcentaje de aplicaciones que se han quedado fuera de la predicción por tener un único lanzamiento, sería interesante utilizar un modelo de aprendizaje no supervisado o semi supervisado para categorizar este tipo de aplicaciones en clústeres de aplicaciones y predecir su consumo en base a aplicaciones ya conocidas que se encuentren categorizadas en ese mismo clúster.

10.3 Resumen

En este capítulo se han revisado las etapas seguidas a lo largo del trabajo para la consecución del objetivo planteado. A lo largo de las mismas se han comentado algunas conclusiones y dificultades encontradas. Finalmente, se han descrito dos líneas futuras de trabajo que darían continuidad al trabajo realizado.

11 Bibliografía

[1]. IDC's Global DataSphere Forecast. Disponible en: https://www.idc.com/getdoc.jsp?containerId=US49018922. Fecha del último acceso: 15 de septiembre de 2022

[2]. Yang, J., Liu, C., Shang, Y., Mao, Z., & Chen, J. (2013, June). Workload predicting-based automatic scaling in service clouds. In 2013 IEEE Sixth International Conference on Cloud Computing (pp. 810-815). IEEE.

[3]. Rao, S. N., Shobha, G., Prabhu, S., & Deepamala, N. (2019, December). Time Series Forecasting methods suitable for prediction of CPU usage. In 2019 4th International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS) (Vol. 4, pp. 1-5). IEEE.

[4]. Nashold, L., & Krishnan, R. (2020). Using lstm and sarima models to forecast cluster cpu usage. arXiv preprint arXiv:2007.08092.

[5]. John Wilkes et al (2020). Google cluster-usage traces v3. Disponible en: https://github.com/google/cluster-data/blob/master/ClusterData2019.md . Fecha del último acceso: 19 de septiembre de 2022.

[6]. Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., ... & Wilkes, J. (2020, April). Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems* (pp. 1-14).

[7]. John Wilkes et al (2020). ClusterData 2011 traces. Disponible en: https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md. Fecha del último acceso: 26 de septiembre de 2022.

[8]. Kang, Z. (2022). *An Analysis of Workload Patterns In Borg Cloud Cluster Traces* (Doctoral dissertation).

[9]. BigQuery. Disponible en: https://cloud.google.com/bigquery. Fecha del último acceso: 26 de septiembre de 2022.

[10]. Project Jupyter. Disponible en: https://jupyter.org/ . Fecha del último acceso: 26 de septiembre de 2022.

[11]. Python Software Foundation. Disponible en: https://www.python.org/psf/. Fecha del último acceso: 26 de septiembre de 2022.

[12]. Abhishek Verma et al (2015). Large-scale cluster management at Google with Borg. Disponible en: https://dl.acm.org/doi/10.1145/2741948.2741964 . Fecha del último acceso: 19 de septiembre de 2022.

[13]. Google Cloud. Disponible en https://console.cloud.google.com/. Fecha del último acceso: 19 de septiembre de 2022.

[14]. Pandas DataFrame. Disponible en: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html. Fecha del último acceso: 19 de septiembre de 2022.

[15]. API Google BigQuery. Disponible en https://cloud.google.com/bigquery/docs/reference/libraries?hl=es_419. Fecha del último acceso: 19 de septiembre de 2022.

[16]. Nielsen, A. (2019). Practical time series analysis: Prediction with statistics and machine learning. O'Reilly Media.

[17]. Brownlee, J. (2017). Introduction to time series forecasting with python: how to prepare data and develop models to predict the future. Machine Learning Mastery.

[18]. Brownlee, J. (2018). Deep learning for time series forecasting: predict the future with MLPs, CNNs and LSTMs in Python. Machine Learning Mastery.

[19]. Using CNN for financial time series prediction. Disponible en: https://machinelearningmastery.com/using-cnn-for-financial-time-series-prediction/. Fecha del último acceso: 20 de septiembre de 2022.

[20]. Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc.".

[21]. sklearn.ensemble.RandomForestRegressor. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html. Fecha del último acceso: 20 de septiembre de 2022.

[22]. sklearn.ensemble.HistGradientBoostingRegressor. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingRegressor.html. Fecha del último acceso: 20 de septiembre de 2022.

[23]. sklearn.ensemble.GradientBoostingRegressor. Disponible en: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ensemble.GradientBoostingRegressor. Fecha del último acceso: 20 de septiembre de 2022.

[24]. The Keras Sequential model. Disponible en: https://keras.io/guides/sequential_model/. Fecha del último acceso: 20 de septiembre de 2022.

[25]. API Keras Layers. Disponible en: https://keras.io/api/layers/. Fecha del último acceso: 20 de septiembre de 2022.

[26]. Keras Adam Optimizer. Disponible en: https://keras.io/api/optimizers/adam/. Fecha del último acceso: 20 de septiembre de 2022.

[27]. Keras Callbacks API. Disponible en: https://keras.io/api/callbacks/. Fecha del último acceso: 20 de septiembre de 2022.

12 Siglas

API: Application Programming Interface

ARIMA: AutoRegressive Integrated Moving Average

CNN: Convolutional Neural Network

CPU: Central Processing Unit

GB: GigaByte

GCP: Google Cloud Platform

GCU: Google Compute Unit

JSON: JavaScript Object Notation

LSTM: Long Short-Term Memory

MAE: Mean Absolute Error

NCU: Normalized Compute Units

RMSE: Root Mean Squared Error

RNN: Recurrent Neural Network

SARIMA: Seasonal Autoregressive Integrated Moving Average

SQL: Structured Query Language

TB: TeraByte

13 Anexo A: Detalle de traza de una aplicación de Borg

Con objeto de entender mejor la línea de tiempo en la que se ejecutan las aplicaciones de Borg, se muestra a continuación un ejemplo simplificado de la aplicación: 'XDTtaVIz3RXS8HhJPZI7LRKUe1aTtT2z6NelbOke17Y='

Por claridad solo se tendrán en cuenta 2 de sus ejecuciones, en las que se trazará el consumo de CPU de sus 2 primeras tareas, 4 en total.

La tabla A-1 muestra el detalle de las dos ejecuciones comentadas, donde collection_id es el identificador de ejecución, task es el identificador de tarea dentro de la ejecución, duration es el punto en el tiempo donde se informa el consumo de CPU durante la ejecución y cpus es el valor informado en ese punto.

collection_id	task	duration	cpus
384907706209	0	14	0,01834
384907706209	0	53	0,00172
384907706209	0	62	0,00041
384907706209	0	65	0,00124
384907706209	1	11	0,02524
384907706209	1	52	0,00057
384907706209	1	54	0,00000
384907706209	1	56	0,00141

collection_id	task	Duracion	cpus
384911844450	0	8	0,01855
384911844450	0	22	0,02036
384911844450	0	36	0,00040
384911844450	0	40	0,00099
384911844450	1	5	0,02634
384911844450	1	24	0,02316
384911844450	1	92	0,00041
384911844450	1	96	0,00104

Tabla A-1. Relación simplificada de eventos informados durante las ejecuciones de una aplicación

De los valores registrados en la tabla se desprende que existe solapamiento de tareas dentro de una misma ejecución. Para esta aplicación en concreto, incluso la tarea 1 comienza unos segundos antes que la tarea 0 en ambas ejecuciones. Este hecho muestra que la ejecución de las tareas está paralelizada, lo cual tiene sentido para aumentar el rendimiento en la ejecución de la aplicación.

La figura A-1 muestra las dos ejecuciones de la aplicación distanciadas en el tiempo. Se observa que en ambos casos sus tareas se ejecutan de modo paralelo como se comentaba anteriormente.

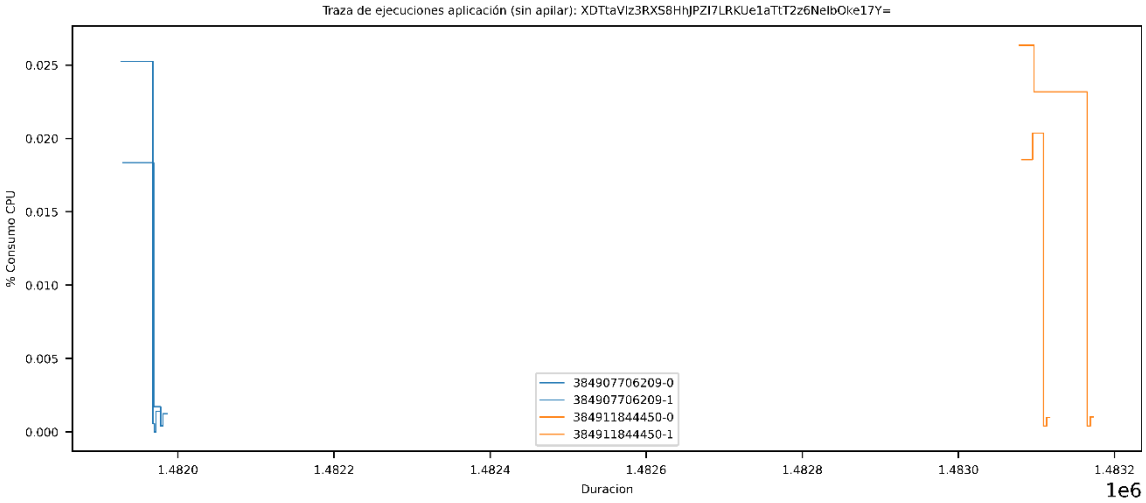

Figura A-1. Gráfica de ejecuciones de un aplicación sin solapar

La gráfica muestra la línea de tiempo natural de ejecución de la aplicación pero, para revisar su comportamiento en general, es más conveniente solapar en una gráfica todas sus ejecuciones. De modo que todas tengan un punto de inicio común y pueda mostrarse gráficamente los posibles picos de consumo que se producen a lo largo de la ventana global de ejecución y ver si hay patrones que se repitan.

La figura A-2 muestra de modo solapado las ejecuciones correspondientes a la aplicación comentada. Cada línea hace referencia a cada una de las tareas.

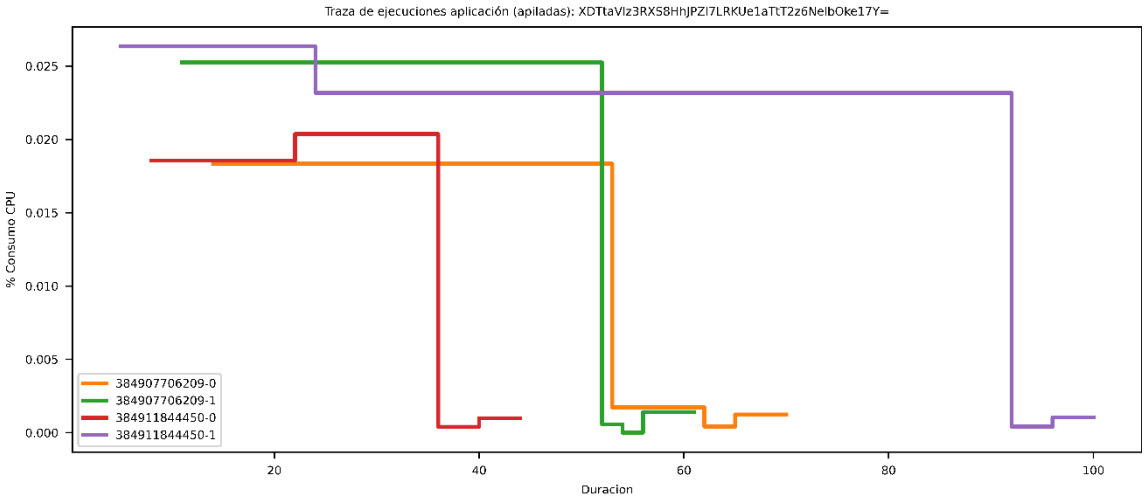

Figura A-2. Gráfica de ejecuciones de un aplicación con solapamiento

Las variaciones de consumo a lo largo del tiempo corresponden a cada uno de los valores informados en la tabla anterior. Por ejemplo, para la tarea *384907706209-0* (línea naranja), comienza su ejecución a los 14 segundos (*cpu*=0,01834), a los 53 segundos el consumo máximo disminuye (*cpu*=0,00172), a los 62 segundos vuelve a

disminuir (*cpu*=0,00041) y a los 65 segundos aumenta ligeramente (*cpu*=0,00124), 5 segundos más tarde la tarea finaliza.

El método anterior puede aplicarse al resto de aplicaciones con independencia del número de ejecuciones y tareas que contengan. Si el número es elevado, se pierde visibilidad en la gráfica al discriminar las tareas por colores. Pero generalizando a un solo color el gráfico sigue conservando el comportamiento global de la aplicación.

Por ejemplo, la siguiente gráfica muestra la 61 ejecuciones solapadas de la aplicación 'bC1hSPDOqSOtU9TOaVU54/r7dIznXJZGJgcC+dHxpkw=', con un total de 60472 tareas.

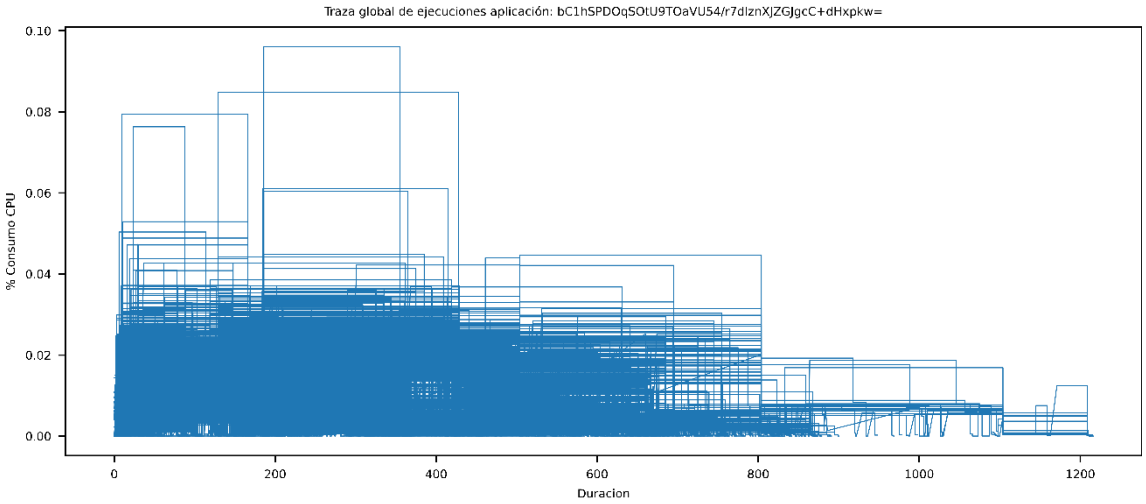

Figura A-3. Gráfica global de ejecuciones de un aplicación con solapamiento

Observando el gráfico, podría decirse que para esta aplicación el consumo máximo normalmente no sobrepasa 0.03 (aprox) durante los primeros 600 segundos o que no se esperan consumos superiores a este valor más allá de los 800 segundos.