eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO DE FIN DE GRADO

MODELO BASADO EN REDES NEURONALES PARA LA PREDICCIÓN DE DISPONIBILIDAD DE UN SISTEMA URBANO DE PRÉSTAMO DE BICICLETAS

Alumno de Martín, Gil, Javier
Director Saratxaga, Couceiro, Ibon

Curso: 2017-2018

Fecha: 24 de julio de 2018

Resumen Laburpena Abstract

Los servicios de préstamo de bicicletas permiten a los usuarios acercarse a una estación y depositar o retirar bicicletas. Uno de sus inconvenientes es que la disponibilidad no está garantizada como puede ser en otros medios como autobús o metro. Se analizará el comportamiento del sistema propio de Bilbao (Bilbon Bizi) empleando técnicas de Machine Learning para reconocer patrones de uso. Generando una red neuronal que, en base a datos anteriores, prediga la disponibilidad para el día siguiente. El entrenamiento de la red neuronal es posible gracias al acceso a catálogos de datos abiertos que contienen datos de disponibilidad esenciales para este proyecto. Éstos son públicos y la mayoría de sus usos son la creación de proyectos de analítica de datos como este. Finalmente, intentando mejorar la experiencia del usuario, se implementará este algoritmo de predicción en una aplicación de consulta que utiliza otros métodos menos precisos.

Palabras Clave: machine learning, bicicletas, app, datos abiertos.

Bizikleta mailegu-zerbitzuei esker, erabiltzaileak geltoki batera hurbildu daitezke eta bizikleta hauek hartu edo utzi. Beste garraiotan (autobusa, metroa...) erabilgarritasuna bermatuta egon daiteke, baina zerbitzu honetan ez da hori gertatzen, hau izanik haren desabantailetako bat. Bilboko berezko bizikleta mailegu-sistemaren (BilbonBizi) portaera aztertuko da, Machine Learning tekniketaz baliatuz erabilera-patroiak ezagutzeko. Datu historikoak erabiliz, hurrengo eguneko erabilgarritasuna iragarriko duen sare neuronala sortuko da. Sare neuronalaren entrenamendua aurrera eramateko ezinbestekoa da datu irekien katalogoak erabiltzea, hauek baitituzte erabilgarritasunarekin erlazionatutako datuak. Katalogo hauek publikoak dira eta datuen analitikarekin erlazionatutako proiektuek erabili ohi dituzte. Azkenik, erabiltzaileen esperientzia hobetzeko asmoz, algoritmo hau mugikorreko aplikazio batean inplementatuko da.

Gako-hitzak: machine learning, app, bizikleta, open data.

Bike sharing services allow users get to a station and take or return bikes. They have a main drawback, availability is not guaranteed as it can be in other means of transport such as bus or metro. In this paper it will be analyzed the behaviour of Bilbao's system (Bilbon Bizi) using Machine Learning techniques to recognize usage patterns. Creating a neural network that, using previous data, will predict the availability for the next day. The training of the model is possible thanks to the use of Open Data catalogs that store availability data for this project. Those are public and their main use are the creation of data analytic projects like this one. Finally, trying to improve the user experience, the prediction method will be implemented in an app that uses other methods that are much less precise.

Keywords: bike sharing, neural networks, app, open data.

Índice

Lista de figuras

Lista de tablas

Lista de acrónimos

NN Neural Network

ML Machine Learning

API Application Programming Interface

CSV Comma Separated Values

GPU Graphics Processing Unit

DNN Deep Neural Networks

1. Introducción

El concepto de servicios de préstamo de bicicletas se estableció en la década de 1960 en Amsterdam. Es el grupo revolucionario anarquista holandés Provo, harto de la contaminación y del exceso de vehículos en las calles, quien decide poner en marcha una nueva campaña. Colocarán 2.000 bicicletas blancas en las calles de la capital holandesa para que los habitantes puedan desplazarse con ellas por la ciudad.

Figura 1: Bicicletas siendo entregadas del plan Witte Fietsenplan

Así nace el **primer servicio de préstamo de bicicletas**, *Witte Fietsenplan*[1], cualquiera podía acercarse, utilizar una bicicleta hasta su destino y allí dejarla para el siguiente usuario. No había sitios establecidos donde depositarlas por lo que era muy complicado encontrar bicicletas. Las autoridades devolvieron las bicicletas a Provo ya que era ilegal dejar bicicletas sin candado por la ciudad.

Como respuesta a esta acción, se añadieron candados y la combinación de desbloqueo de éstos se pintó en las bicicletas para que pudieran ser utilizadas de nuevo. Como no había ningún sistema de registro de usuarios el experimento acabó mal y la mayoría de las bicicletas fueron robadas.

A pesar de que este experimento no tuviera buen resultado asentó las bases de lo que décadas después serían los servicios públicos de préstamo de bicicletas como los que se usan hoy en día.

En los últimos años se ha evolucionado hasta modelos con registro de usuarios para evitar robos y estaciones donde los usuarios depositan y recogen las bicicletas.

Estos sistemas pueden ser de dos tipos, **anclados** (figura 2) y **no anclados** (figura 3). Los **servicios anclados** suelen ser instalados en su mayoría por ayuntamientos,

[1] https://www.theguardian.com/cities/2016/apr/26/story-cities-amsterdam-bike-share-scheme

Figura 2: Sistema de préstamo de bicicletas ancladas en Bilbao

Figura 3: Sistema de préstamo de bicicletas sin anclajes en Madrid

siendo necesario para dar el servicio la construcción de una infraestructura que contenga los puntos donde pueden ser recogidas y depositadas las bicicletas. Todos los usuarios que quieran hacer uso del sistema deben registrarse, obteniendo una tarjeta con la que identificarse en los puntos de anclaje. Para retirar una bicicleta basta con acercarse a la estación, introducir una tarjeta y retirar el vehículo. Una vez se finalice el trayecto hay que depositar la bicicleta en el anclaje de destino, si bien parece cómodo tiene un gran inconveniente: la disponibilidad de bicicletas no es regular y estaciones de mucho tránsito pueden quedarse vacías o llenas. Cualquiera de estos dos casos supone un problema tanto para los usuarios o los administradores del servicio.

El otro tipo de sistema, correspondiente a los **servicios sin anclaje** está teniendo un auge reciente. Son empresas privadas que desembarcan, literalmente, en ciudades con miles de bicicletas que son depositadas individualmente en puntos de la ciudad. Esta modalidad del servicio es interesante puesto que permite depositar las bicicletas en cualquier sitio. Esto se convierte también en un inconveniente ya que, al no existir estaciones fijas, no se puede asistir a un punto en concreto y esperar encontrar bicicletas. Para ello estos servicios cuentan con aplicaciones de consulta pero no soluciona el problema.

Cualquiera de los posibles sistemas ha supuesto un gran avance en lo que al transporte se refiere, si bien este trabajo se centra en los sistemas con anclajes y todas las menciones a *sistemas de préstamo de bicicletas* o similares estarán referidas a este modelo.

2. Contexto

Cada servicio de préstamo de bicicletas moderno tiene una app complementaria (figura 4), en su mayoría desarrollada por la institución que es propietaria del servicio. Si bien muchas de estas aplicaciones reciben actualizaciones regulares, otras están estancadas y no tienen apoyo ninguno, dejando al usuario de lado y con una sensación de abandono y recurriendo a ayudas de terceros como CityMapper[1].

Figura 4: Apps de consulta oficial (izquierda) y proyecto personal (derecha)

Desde estas mismas instituciones se promueven los catálogos de datos de Open Data, que no son más que una colección de recursos donde encontrar datos actualizados con frecuencia de estos servicios. En el caso de Bilbao, en el catálogo de Open Data hay una URL que muestra los valores de disponibilidad de cada estación, pudiendo ahí extraerse los datos y crear una aplicación que sirva estos valores y aporte más al ciudadano que otras alternativas.

Visto que estos datos son públicos y tras varios años de experiencia realizando desarrollo para iOS, se creó a nivel personal una aplicación disponible en la App Store de manera gratuita (figura 4). Esta cuenta con un mapa en el que se ubican los puntos de retirada de bicicletas junto a su disponibilidad actual, además se muestra un gráfico en

[1] Aplicación de consulta del estado de diferentes servicios de transporte, disponible en diferentes ciudades del mundo. Proporciona información muy detallada sobre la disponibilidad. `https://citymapper.com/`

la parte inferior (figura 5) con la disponibilidad media de esa estación para ese día. Con esto se pretende ayudar al usuario a conocer cuál es el uso de esa estación y permitirle conocer con anterioridad si existe la posibilidad de que esa estación se vaya a quedar vacía. Quedarse sin bicicleta supone un problema si es el medio de transporte en el que se confía para ir o volver de trabajar.

Figura 5: Gráfico de la aplicación que muestra una media del uso diario de una estación

Este gráfico, es útil para identificar patrones de uso o conocer cuándo una estación puede tener problemas de disponibilidad, pero no es muy preciso al ser esta predicción realizada con una media. Siendo este análisis para Bilbao, los valores de predicción se ven muy afectados por los cambios en la meteorología.

Expuestos los fallos que existen ahora mismo en la aplicación, nace la motivación de realizar este trabajo: crear un sistema de predicciones fiable para el usuario que aporte datos útiles para hacer un mejor uso del servicio.

3. Objetivos y alcance

3.1. Objetivos

Este trabajo tiene diferentes finalidades que pueden ser clasificadas según su nivel de impacto.

- **Objetivos de primer nivel**
    1. **Mejorar el sistema de predicciones ya existente en la app** incluyendo información del historial de uso anterior o patrones de uso del servicio basado en redes neuronales.
- **Objetivos secundarios**
    1. Familiarizarse y aprender sobre el entorno de Machine Learning especialmente *Deep Neural Networks* (DNN).
    2. Poder seguir ofreciendo una app gratuita y fiable con funcionalidades añadidas.
    3. **Construir las bases de lo que puede ser un producto exportable a otras ciudades**. Crear una base de un sistema de predicciones para poder utilizarlo al ampliar el servicio a más ciudades.
    4. Acumular datos históricos de disponibilidad en otras ciudades para expandir el proyecto.

3.2. Alcance

- Realizar en una primera fase el sistema para Bilbao y recoger datos en base a los feeds abiertos.
- Conseguir desplegar este sistema de predicciones en dispositivos iOS.
- Recoger únicamente datos de disponibilidad, para futuras iteraciones tener en cuenta los factores climatológicos.

4. Beneficios

4.1. Económicos

Entre los beneficios principales del proyecto, existe una serie de beneficios económicos; es decir, se proporciona una reducción de costes asociados a diferentes procesos o recursos, en menor o mayor medida. Entre este tipo de beneficios, se pueden destacar los siguientes:

- **Mejora en la eficiencia del balanceo de la red**. Al quedarse una estación vacía, son los operarios los que tras un tiempo se acercan con un furgón con bicicletas de otras estaciones para equilibrar esa estación. Al poder realizar predicciones del servicio los administradores pueden equilibrar el servicio antes de que se vacíen estaciones, evitando tener que ir cuando ya hay estaciones vacías y los usuarios han dejado de utilizarlo por esa razón.
- **Conocer el uso real del sistema**. Con estos métodos de predicción se puede conocer en profundidad qué estaciones son usadas con más frecuencia y podrían necesitar ser ampliadas y otras en cambio que no sean utilizadas y posean capacidad en exceso. Pudiendo servir esto para realizar análisis futuros sobre en qué zonas dedicar esfuerzo a construir más estaciones.

Respecto al posible beneficio personal se pueden distinguir los siguientes beneficios:

- **Explotación por publicidad**. A partir de la creación del modelo de predicciones y su implementación en una aplicación no oficial del servicio se podrían cubrir costes mediante publicidad dentro de la aplicación. Similar a otros modelos económicos de otras aplicaciones móviles.
- **Explotación por un modelo de suscripción**. A partir del modelo neuronal construido se puede poner en marcha una API [1] que mediante un modelo de suscripción los administradores del servicio o interesados puedan acceder a datos de patrones de uso y predicciones de la red.

4.2. Sociales

Por último, existen beneficios sociales asociados a este proyecto, aquellos que repercuten de forma directa o indirecta en la sociedad.

[1] Application Programming Interface

- **Mejores servicios de cara a la ciudadanía**. Proporcionando mejores datos a los usuarios se puede conseguir incrementar el uso del servicio al ser considerado más fiable y con más recursos. Se han realizado distintos estudios sobre la predicción de disponibilidad de estaciones con redes neuronales pero a fecha de cierre del proyecto no hay ninguna aplicación o servicio que las implemente.
- **Mejora de imagen de las instituciones**. Al adaptarse las instituciones públicas a nuevos entornos, la ciudadanía puede llegar a adquirir una mejor confianza en las instituciones y los servicios que proveen. Los ciudadanos verían que las instituciones invierten en avances tecnológicos y que Bilbao es una ciudad puntera en este sector al añadir estos sistemas predictivos.
- Con un mejor servicio se fomenta que se usen más las bicicletas pudiendo llegar a un **menor uso del vehículo privado** contaminando menos.

5. Análisis de alternativas

Para cada selección se valorarán los siguientes criterios:

- **Sencillez** (10 %)
- **Rendimiento** (20 %)
- **Flexibilidad** (15 %)
- **Coste Económico** (25 %) Posibles inversiones que hay que realizar para seleccionar esa alternativa.
- **Coste de Aprendizaje** (30 %) Tanto tiempo que se necesita en aprender a utilizar la alternativa como el coste que puede llevar acarreado el aprenderla.

Todos los valores se puntuarán en la tabla sobre diez puntos y serán ponderados según su importancia mencionada en la lista anterior para tomar al final la decisión de qué opción planteada utilizar. Un menor valor refleja que es una peor solución.

5.1. Métodos de predicción

- **Estadísticos clásicos**: Partiendo de que el actual método de predicción es la media se propone añadir varianza. Con esto se puede obtener un gráfico del historial medio junto con las posibles variaciones de valores que habrá hora a hora. Consiguiendo así una mejor adaptación a las variaciones externas.
- **Árbol de decisiones** Entre a los algoritmos de Machine Learning este método era una de las opciones a tener en cuenta. Si bien puede ser una solución adecuada visto en comparación con las redes neuronales está limitado ya que todo se basa en condiciones, similar a `if` en código. Requiere bastante trabajo y es poco adaptable a distintas situaciones.
- **Redes Neuronales** Solución elegida debido a su versatilidad y flexibilidad. Su principal problema es que hay una curva de aprendizaje y que el entrenamiento de los modelos requiere equipos específicos como GPU para que sean rápidos.

En la tabla 1 la solución de los estadísticos y las redes neuronales obtienen prácticamente las mismas puntuaciones. Es el coste económico y el de aprendizaje lo que perjudica en exceso a la última solución pero proporciona mejores resultados.

	Estadísticos	Árbol de Decisiones	Redes Neuronales
Sencillez	10	7	3
Rendimiento	1	5	9
Flexibilidad	0	5	10
Coste Económico	10	2	6
Coste de Aprendizaje	8	6	4
Total	6.1	4.75	**6,3**

Tabla 1: Comparación de los criterios de selección para los métodos de predicción

5.2. Métodos de implementación de redes neuronales

El *core* de este desarrollo se basa en la predicción utilizando redes neuronales. Tras el auge que llevan experimentando las redes neuronales estos años aparecen con frecuencia nuevas tecnologías de desarrollo. Se comentan las que se han planteado para este trabajo.

- **TensorFlow** A día de hoy es la librería con mayor uso. Tiene una comunidad muy grande por detrás pero su curva de aprendizaje es más compleja que las de otras librerías.
- **Keras** Se presenta como una interfaz sencilla para trabajar con redes neuronales. Se puede utilizar sobre Theano o TensorFlow pero no hace falta interactuar con ninguno de esos niveles, sólo con Keras la librería hace todo el resto. Su curva de aprendizaje es menor que en TensorFlow.
- **CoreML** Framework de Apple para trabajar con redes neuronales. Si bien es muy útil ya que puede correr sobre dispositivos móviles y emplearse dentro de las apps es propietaria de Apple y su uso no puede ser más allá de sus plataformas. Al estar diseñada para correr en dispositivos iOS o macOS se escribe en Swift.

	TensorFlow	Keras	CoreML
Sencillez	6	10	7
Rendimiento	10	9	8
Flexibilidad	10	10	2
Coste Económico	10	10	6
Coste de Aprendizaje	6	10	8
Total	8,4	**9,8**	6,5

Tabla 2: Comparación de los criterios de selección para los métodos de implementación de redes neuronales

Si bien la opción ideal para este caso sería implementar CoreML ya que corre en dispositivos iOS y la aplicación para la que se quieren implementar las predicciones corre en esa plataforma. Elegir este caso sería limitarse en flexibilidad y no poder expandirse a más plataformas como crear una app web o migrar a Android. Es por eso que frente a la facilidad que ofrece Keras respecto a TensorFlow se ha elegido la primera.

5.3. Lenguajes de programación

- **Swift**: Al querer mejorar el sistema de predicciones de la aplicación lo más directo sería utilizar este lenguaje y luego el framework de Apple para redes neuronales,

CoreML. La elección de este lenguaje limita bastante las opciones y caminos de escalabilidad para el producto futuro.

- **Python** Lenguaje casi por excelencia empleado para trabajar con redes neuronales.

	Swift	Python
Sencillez	10	10
Soporte	3	10
Flexibilidad	6	10
Coste Económico	10	10
Coste de Aprendizaje	6	10
Total	6,8	**10**

Tabla 3: Comparación de los criterios de selección para los lenguajes de programación

Al haber elegido en el apartado utilizar Keras ya se elige de forma automática el lenguaje, Python.

6. Descripción de la solución

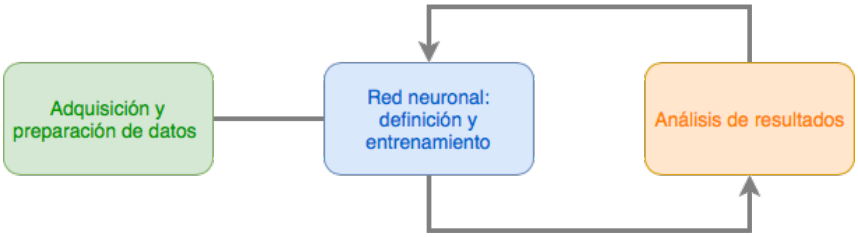


Figura 6: Pasos seguidos en el desarrollo de la solución

Los pasos principales (figura 6) que se van a seguir en el proyecto son los que se detallan a continuación:

1. **Adquisición y preparación de datos**. Creación de scripts en un servidor para que guarden en intervalos de cinco minutos los datos para ir construyendo poco a poco el historial necesario.
    - **Preparación y tratamiento de los datos de entrada al sistema**. Elegir de todos los datos que se están guardando cuáles son relevantes para realizar la predicción.
    - **Codificación cíclica de datos temporales**. Dar formato a aquellos datos que tienen una representación temporal, como el tiempo o los días de la semana, un formato cíclico para que la red neuronal sea capaz de entenderlo.
    - **Normalización de datos**. Entregar al modelo los datos con un rango determinado, se escalan los valores desde 0 hasta 1.
    - **Organización de los conjuntos de datos de entrenamiento**.
2. **Red neuronal**. Definición del modelo y entrenamiento con los datos adquiridos.
    - Se emplearán redes *Long Sort-Term Memory* (LSTM) ideales para este tipo de predicción de secuencias y aprendizaje supervisado.
3. **Análisis de resultados**. Obtenidos los resultados comprobar si son correctos y no y analizar los parámetros de entrada para ver cuáles pueden ser modificados para mejorarlo.

Con todo este proceso se generará una red neuronal que al darle como entrada los dos días previos a la predicción pondrá en la salida los valores predichos para el día siguiente.

Con todo este proceso se generará una red neuronal que tomará como datos de entrada los dos días anteriores y las cinco estaciones próximas a la estación de Zunzunegi.

6.1. Adquisición y preparación de datos

En la figura 7 se detallan las modificaciones que se realizarán a los datos y se detallan en esta sección.

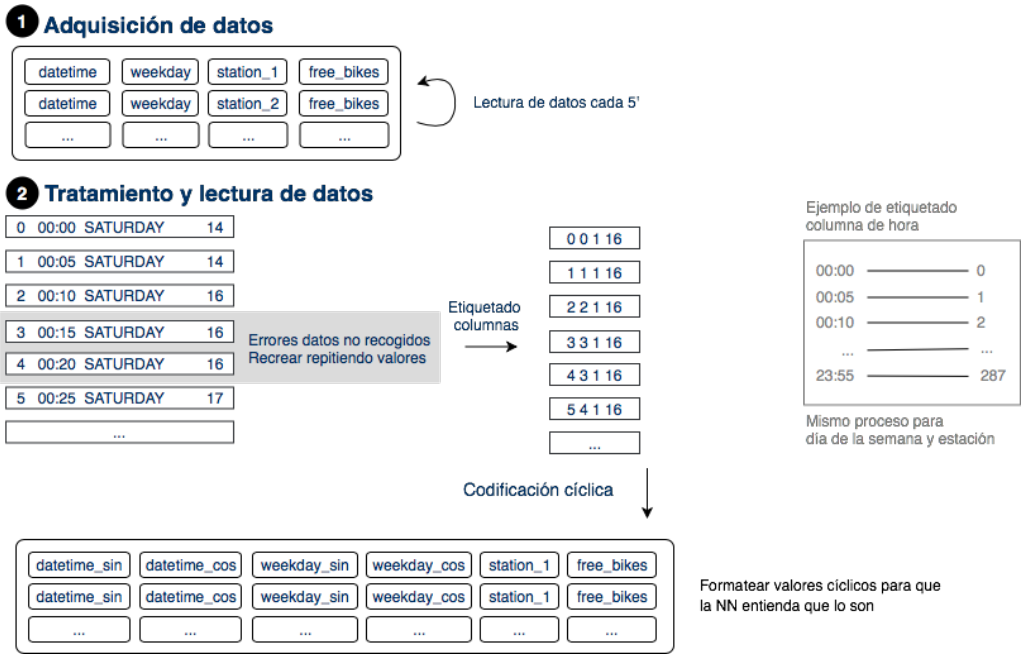


Figura 7: Esquema del procesado de los datos a bajo nivel

Se parte de una base de la cual no se posee ningún historial o nada similar con el que trabajar y poder empezar a entrenar la red neuronal. Como respuesta a eso se crea un script que se ejecute cada cinco minutos en un servidor y que recoja los datos que se le indiquen para ser guardados e ir creando el archivo que se necesita. Para ello se empleará el catálogo de datos abiertos del Ayuntamiento de Bilbao, el cual tiene una URL que muestra la disponibilidad de todas las estaciones en el momento. Estos datos se guardarán en un fichero separado por comas (CSV, *Comma Separated Values*) utilizados frecuentemente para almacenar estos tipos de datos. Los valores que se recogerán son los que se ve en la tabla 4, en secciones posteriores se comentarán cuáles de estos valores son utilizados.

datetime	weekday	identificator	station	free_bikes	free_docks

Tabla 4: Columnas de datos recogidos

A la vez que se realiza la codificación de este programa se contacta con el Ayuntamiento para consultar si existe algún registro del historial como el que se está haciendo manualmente. Se obtiene una respuesta y lo más parecido que existe es un historial de todos los viajes de todos los usuarios, a pesar de que no son los datos que se esperaban se recogen en un CD por si fueran útiles en un futuro.

Al disponer de pocos datos podría hacerse una práctica habitual de recrear datos como dice Goodfellow (2016, p. 233),

> *"La mejor forma de que un modelo de machine learning generalize mejor es entrenarlo con más datos. Obviamente, en la práctica, el tamaño de los datos*

que se tiene es limitado. Una forma de resolver este problema es crear datos falsos y añadirlos al set de entrenamiento."

Esto es útil para resolver ciertos problemas de Machine Learning como identificación de imágenes, en este caso no son aplicables ya que no se pueden inventar con fiabilidad días de disponibilidad. Se usarán los datos recolectados disponibles únicamente.

El alcance de este proyecto sólo se realizará para el servicio Bilbon Bizi de Bilbao pero para implementaciones futuras se están recopilando datos de diferentes ciudades. Para todas éstas se están recogiendo los mismos datos (tabla 4) cada cinco minutos. El proceso de adquisición comienza en septiembre de 2017 y se listan a continuación todas las ciudades que se están observando para futuros casos y el tamaño del fichero con los datos a fecha del cierre documental del proyecto:

- Barcelona (1,73GB)
- Bilbao (116MB)
- Brisbane (710MB)
- Bruselas (1,57GB)
- Goteborg (287MB)
- Lillestrom (21MB)
- Ljubljana (249MB)
- Londres (4GB)
- Lund (56MB)
- Luxemburgo (286MB)
- Lyon (1,67GB)
- Madrid (737MB)
- Marsella (595MB)
- Namur (108MB)
- Nantes (431MB)
- París (1,2GB)
- Santander (76MB)
- Sevilla (1,27GB)
- Toulouse (1,35GB)
- Valencia (1,37GB)

6.1.1. Preparación y tratamiento de los datos de entrada al sistema

Un ejemplo del contenido del archivo de datos es el que se ve en la figura 8. Una parte muy importante al leer los datos y comenzar a prepararlos es comprobar que son consistentes, es decir, comprobar que no hay huecos en las muestras debido a errores como: URL no disponible en el momento que se hace la petición o servidor no estaba operativo en ese momento. En este caso, hay huecos como se puede ver en la columna `time` entre las muestras `00:45` y `01:00`.

```
2017/09/30 00:00,SATURDAY,01,PLAZA LEVANTE,12,3
2017/09/30 00:00,SATURDAY,02,IRUÑA,18,1
2017/09/30 00:00,SATURDAY,03,AYUNTAMIENTO,20,0
2017/09/30 00:00,SATURDAY,04,PLAZA ARRIAGA,20,15
2017/09/30 00:00,SATURDAY,05,SANTIAGO COMPOSTELA,9,5
2017/09/30 00:00,SATURDAY,06,PLAZA REKALDE,9,5
2017/09/30 00:00,SATURDAY,07,DR. AREILZA,14,1
2017/09/30 00:00,SATURDAY,08,ZUNZUNEGI,19,0
2017/09/30 00:00,SATURDAY,09,ASTILLERO,7,8
2017/09/30 00:00,SATURDAY,10,EGUILLOR,10,4
2017/09/30 00:00,SATURDAY,11,S.  CORAZON,14,0
2017/09/30 00:00,SATURDAY,12,PLAZA INDAUTXU,9,6
2017/09/30 00:00,SATURDAY,13,LEHENDAKARI LEIZAOLA,16,3
2017/09/30 00:00,SATURDAY,14,CAMPA IBAIZABAL,8,7
2017/09/30 00:00,SATURDAY,15,POLID. ATXURI,13,1
2017/09/30 00:00,SATURDAY,16,SAN PEDRO,15,0
```

Figura 8: Ejemplo de datos adquiridos con huecos entre muestras

Estos huecos que pueden quedar si no son muy grandes, superiores a varias horas, serán recreados manualmente manteniendo la disponibilidad del último valor conocido. De todos los datos que se han mostrado, no se van a utilizar todos ya que no son relevantes o útiles en el proceso de entrenamiento de la red neuronal. De los que se mencionan en la tabla 4 se van a utilizar únicamente los siguientes:

- Hora del día
- Día de la semana
- Nombre de la estación
- Bicicletas libres

Los valores de las columnas `time` y `weekday` no se pueden entregar al modelo así ya que son cadenas de caracteres, hay que **preparar los datos para que tengan una representación numérica**. El número de posibles valores para estas dos columnas está fijado, para los días de la semana son siete y para las horas del día son 288 valores como se explica en la sección 6.1.2. El proceso que se lleva a cabo es a cada categoría única asignarle un valor entero único. Por ejemplo el día de la semana `LUNES` es `0`, el `MARTES` es `1` y así sucesivamente. Este proceso se conoce como **codificación de etiquetas** y se puede ver el resultado en la figura 9.

La codificación de etiquetas de los valores sólo se hace para los tres primeros (`time`, `weekday` y `station`) ya que la columna que referencia a las bicicletas libres es un valor numérico.

```
[273 0 2 16]
[273 1 2 16]
[273 2 2 16]
[273 3 2 16]
[273 4 2 16]
...
```

Figura 9: Array de ejemplo con datos ya categorizados

Para algunos datos como el nombre de la estación esto puede ser suficiente, en cambio para otras columnas hacen falta más transformaciones como se ven en la siguiente sección.

6.1.2. Codificación Cíclica de datos temporales

Algunos tipos de datos son inherentemente cíclicos. el tiempo es un claro ejemplo de ello: minutos, horas, días de la semana, estaciones... En este caso los datos temporales cíclicos que se están manejando son el día de la semana y la hora del día. Para las personas es muy sencillo comprender que después del domingo viene el lunes o que después de las 23:59 vienen las 00:00 y comienza un nuevo día, si se entregan los datos así al modelo sin hacer ninguna transformación nunca va a entender esa relación y puede que se limite la capacidad de aprendizaje por eso. A continuación se va a explicar el proceso de codificación cíclica para los datos de la hora, el proceso es similar para los días de la semana.

Las muestras ya están codificadas y tienen su representación numérica en lugar de una cadena de caracteres.

Valor	Codificación
00:00	0
00:05	1
...	...
23:50	286
23:55	287

Tabla 5: Muestra de la codificación por etiquetas de los valores de la hora

Los intervalos que están separados cinco minutos, están a una unidad de distancia como se puede ver en la tabla 5, a partir de ahí el modelo es capaz de deducir que de las `13:00` a las `13:15` hay una distancia de 3. Es aquí cuando surge un problema, de esta forma el modelo nunca va a ser capaz de entender los ritmos diarios si piensa que después de las 23:55 (etiqueta 287) viene la etiqueta 288 que no tiene siquiera decodificación.

Se propone como solución **transformar una característica para que tenga dos dimensiones**, haciendo una representación del coseno y del seno de cada columna. Para cada una habrá ahora dos, por ejemplo para `time` habrá `time_sin` y `time_cos` y así para la columna `weekday`. Ninguna de las otras columnas (`station` y `free_bikes`) requiere de esta transformación ya que no son ninguna representación temporal cíclica. La columna `weekday` también tendrá su representación `weekday_sin` y `weekday_cos`.

Son necesarios estos dos valores, el seno y el coseno, ya que analizando los gráficos posteriores se puede ver que en la representación del seno (figura 10) y del coseno (figura 11) tienen las 23:55 y las 00:00 a 5 minutos de distancia. Esto es debido a la periodicidad de

las funciones sinusoidales. A pesar de las ventajas que vemos que supone esto aparecen nuevos problemas.

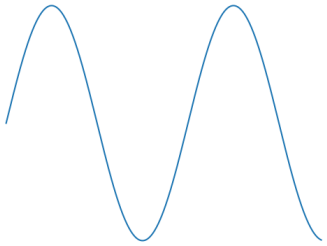


Figura 10: Representación senoidal de una característica

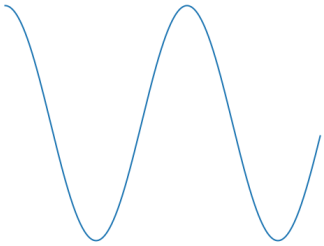


Figura 11: Representación cosenoidal de una característica

Las funciones repiten su periodo cada 2π pero entre medias un mismo valor de y se repite dos veces. Por ejemplo en la función seno (figura 10) para $y = 0$ hay dos valores, las 00:00 y las 12:00, se da el mismo problema para la representación del coseno. ¿Cómo solucionar esto? La forma más intuitiva de mostrarlo es con un gráfico bidimensional, con el coseno en el eje X y el seno en el eje Y consiguiendo una representación en coordenadas polares (12 en la que no hay representación posible con error.

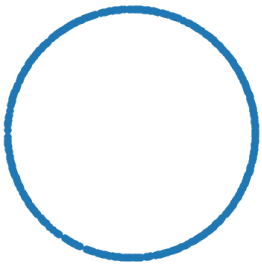


Figura 12: Representación polar de una característica

Tras esta última transformación de las columnas de los datos de entrada los valores conservados son los que se ven en la tabla 6.

time_sin	time_cos	weekday_sin	weekday_cos	station_name	free_bikes

Tabla 6: Columnas de datos recogidos

6.1.3. Normalización de datos

Los datos, una vez transformados, necesitan ser escalados antes de ser utilizados para entrenar la red neuronal. Cuando se emplean datos no normalizados, valores que no están entre cero y uno, es posible que el aprendizaje sea lento y puede que el modelo no converja a una solución deseada. Para esto se utilizará la clase de Keras `MinMaxScaler` y se escalarán los valores de entrada entre 0 y 1, esta misma clase se utilizará posteriormente cuando se obtengan las predicciones para reescalar los datos de salida y obtener los valores correctos.

6.1.4. Organización de los conjuntos de datos de entrenamiento

El problema que se está resolviendo es dado un historial predecir los siguientes valores en una secuencia, en este caso esta secuencia es la disponibilidad de bicicletas de una estación. Lo que hay que hacer es entrenar un modelo que ea capaz de entender estas secuencias y proporcione a la salida un valor adecuado. Para comenzar a trabajar en ello hay que re-estructurar los datos como **problemas de aprendizaje supervisado**, esto es, los datos tienen que ser transformados de una secuencia a pares de entrada X y salida Y. El proceso es similar a entregarle a la red neuronal la pregunta y solución a un problema para que encuentre la relación existente entre ambos.

```
29/10/2018 00:00, 20
29/10/2018 00:05, 25
29/10/2018 00:10, 25
29/10/2018 00:15, 22
...
```

Figura 13: Ejemplo de una secuencia

```
X,                                  y
29/10/2018 00:00, 20 - 29/10/2018 00:05, 25
29/10/2018 00:10, 25 - 29/10/2018 00:15, 22
...
```

Figura 14: Ejemplo de pares de entrada y salida de un problema de aprendizaje supervisado

La figura 15 muestra como tiene que ser el resultado final, dadas unas secuencias de entradas conseguir en la salida los valores de predicción. En la fase de entrenamiento del modelo se darán como datos de entradas los intervalos anteriores y la salida el valor de la predicción que se quiere conseguir, darle la pregunta y la respuesta al problema que se quiere resolver para que la red neuronal encuentre la relación existente entre la entrada y la salida. Una vez terminada la etapa de entrenamiento al modelo se le entregarán únicamente los días previos al día que se quiere predecir y se obtendrán a la salida los valores de la predicción.

Un ejemplo de la transformación de una sucesión de datos temporales (figura 13) a un problema de aprendizaje supervisado como el que se realiza en este trabajo es el que se ve en la figura 15.

Figura 15: Modelado de un problema de aprendizaje supervisado

Por último, se dividirán los datos en los siguientes subconjuntos:

- **Datos de entrenamiento**. Este subconjunto lo ve el modelo una única vez, durante el entrenamiento y es con el que se ajustan los valores del modelo para predecir la salida.
- **Datos de validación**. a set of examples used to tune the parameters of a classifier In the MLP case, we would use the validation set to find the "optimal" number of hidden units or determine a stopping point for the back-propagation algorithmo-delo.

- **Datos de prueba**. Grupo de datos que el modelo no ha visto nunca utilizados para hacer predicciones de prueba y ver cómo se comporta el modelo y ajustar los hiperparámetros y arquitectura.

El fichero cuenta con 2.603.363 líneas, para estos grupos los valores que se han elegido para determinar el tamaño son los siguientes:

Conjunto	Porcentaje (%)	Tamaño (líneas)
Entrenamiento	75	1.952.522
Validación	20	520.672
Prueba	5	130.168

Figura 16: División del conjunto de datos en subconjuntos a utilizar en el entrenamiento

6.2. Red neuronal

Para la selección del modelo, entrenamiento y predicciones que se realizarán existen una serie de pasos:

1. Definir el modelo
2. Compilar el modelo
3. Adaptar el modelo
4. Evaluar el modelo
5. Predecir con el modelo

Una vez ya dado forma al conjunto de datos hay que **definir el modelo** eligiendo el tipo de red neuronal y las capas a asignar, habiendo tres tipos:

- **Capa de entrada**: Cuenta con neuronas de entrada y es responsable de recoger los datos y pasárselos a las capas internas para que realicen el procesado en las siguientes capas.
- **Capas ocultas**: Realizan el procesado y cálculo de valores. Transmiten información entre ellas.
- **Capa de salida**: Última capa de la red que produce las salidas del programa

Como mínimo un modelo está compuesto por tres capas: capa de entrada, capa oculta y capa de salida. Uno de los problemas a resolver en este apartado es el número de capas intermedias a añadir para que el proceso de aprendizaje sea el correcto.

A partir de la adquisición y formato de datos se puede comenzar a definir el modelo que trabajará con estos y a plantear el problema. Elegir el tipo de red neuronal que se va a emplear para resolver el problema de esta predicción.

En este problema es muy importante mantener ordenada la secuencia de observaciones con la que entrenar al modelo y se hacen predicciones. Generalmente, los problemas de predicción que involucran secuencias de datos se conocen como problemas de

predicción de secuencias. Para estos tipos de predicciones se utilizan redes ***Long Short-Term Memory*** (LSTM) que son una variación de las *Recurrent Neural Networks* (RNN).

Para entregar los datos a la red de esta forma se necesita que las entradas se correspondan a las salidas, hay que alinearlas. Es vital que no haya huecos en las muestras como ya se ha comentado en secciones anteriores.

La **definición del modelo** en Keras se realiza como una secuencia de capas (figura 17). El primer paso es crear una instancia de la clase `Sequential` y crear las capas y añadirlas en el orden en el que deberían estar conectadas. La primera capa oculta en el modelo debe definir el número de entradas que se deben esperar. En un modelo LSTM los datos de entrada tienen tres dimensiones muestras, intervalos y características, en ese orden.

- **Muestras**. Son las filas de los datos, una muestra puede ser una secuencia.
- **Time steps**. Observaciones pasadas de una características.
- **Características**. Columnas de datos.

Los modelos LSTM no necesitan que se especifique un número fijo de muestras que deben ser entregadas a la red. El modelo asume una o más muestras, dejando al usuario definir únicamente el número de intervalos y características. Se puede pensar que la clase `Sequential` es como una tubería en la que se entregan los datos a la entrada y las predicciones salen por el otro extremo.

```
model = Sequential()
model.add(LSTM(lstm_neurons, input_shape=(train_x.shape[1], train_x.shape[2]),
return_sequences=True))
model.add(LSTM(lstm_neurons, return_sequences = True))
model.add(LSTM(lstm_neurons))
model.add(Dense(n_out))
```

Figura 17: Definición de las capas del modelo mediante código

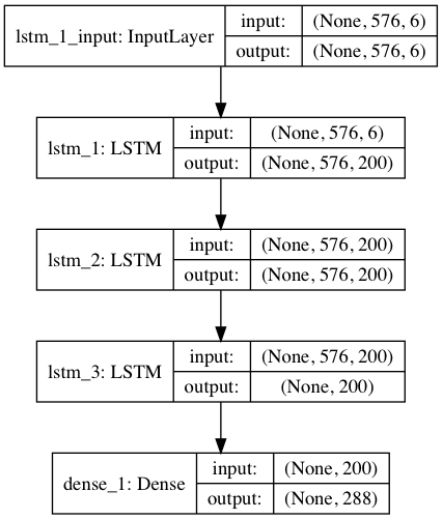


Figura 18: Representación de las capas del modelo neuronal

Tras la definición el segundo paso es **compilar el modelo**, es similar a afinar los últimos parámetros antes de entrenar el modelo. Hay que especificar dos valores:

- **Función de pérdidas** (*loss*) utilizada para evaluar la red. Se ha elegido la función *mean squared error* (error cuadrático medio)
- **Optimizador** (*optimizer*). Algoritmo de optimización con el que entrenar la red. Elegido el optimizador Adam (*Adaptative Movement Estimation*).

```
model.compile(loss='mae', optimizer='adam', metrics = ['mse', 'acc'])
```

Figura 19: Compilación del modelo

Tercer paso, **entrenamiento del modelo**. Una vez se compila la red se puede entrenar, lo que significa adaptar los pesos de las neuronas utilizando un conjunto de datos de entrenamiento. El entrenamiento necesita un conjunto de datos de entrada X y unos de salida Y.

```
history = model.fit(train_x, train_y, batch_size=BATCH, epochs=EPOCHS,
validation_data=(test_x, test_y), verbose=1, shuffle = True)
```

Figura 20: Compilación del modelo

Por último, hay que seleccionar los últimos parámetros antes de entrenar el modelo: *epoch* y *batch*.

- ***epoch***. Número de pasadas que se hacen por el conjunto de datos.
- ***batch***. Número de pasadas que se hacen por subconjuntos de muestras del set de entrenamiento, después de cada una los valores de las neuronas se ajustan. Una *epoch* está compuesta de uno o más *batches*. Se han hecho distintas pruebas con este parámetro que se comentan en la sección de resultados (6.3), destacar los siguientes casos:
  - `batch_size = 1`: Los pesos son actualizados después de cada muestra, conocido como *stochastic gradient descent*.
  - `batch_size = n`: Pesos actualizados después de un número específico `n` de muestras.
  - `batch_size = N`: Igual al número de muestras que tiene el set de entrenamiento, conocido como *batch gradient descent*.

Una vez ha terminado el entrenamiento se devuelve un objeto `history` que provee un resumen del rendimiento del modelo durante el entrenamiento, se utilizarán estos valores en la sección de resultados (6.3) para evaluar la precisión.

Siguiente paso, **evaluar el modelo**. La red puede ser evaluada utilizando el set de entrenamiento pero esto no proveerá información útil sobre el rendimiento de la red como modelo predictivo ya que ese conjunto de datos lo ha visto. Para este proceso se requiere un nuevo conjunto de datos, un subset de pruebas. Esta etapa puede realizarse en un paso independiente o a la vez, en este caso se ha realizado a la vez que se entrena el modelo (figura 20).

Por último, hay que **hacer predicciones con el modelo** una vez el rendimiento sea el adecuado, ahora se puede utilizar para hacer predicciones utilizando nuevos conjuntos de datos.

```
predicted_set = model.predict(scaled)
```

Figura 21: Realizar predicciones utilizando el modelo ya entrenado

6.3. Resultados

Tras haber terminado el proceso de entrenamiento del modelo hay que evaluar el rendimiento de los resultados obtenidos, esto se ha realizado mediante dos vías:

- **Gráficos de rendimiento**. Proporcionados tras el entrenamiento como se explica en el apartado anterior, utilizados para diagnosticar el comportamiento del modelo.
- **Predicciones**: En ciertos casos puede que los gráficos de rendimiento indiquen que el modelo no está correctamente entrenado pero al realizar predicciones éstas sean correctas y proporcionen buenos resultados.

Comenzar comentando el **rendimiento de las predicciones utilizando la media para cada estación** (figura 22). La opinión de los usuarios de este método es que si bien no es fiable debido a las condiciones climatológicas y la poca flexibilidad que ofrece la media ayuda a reconocer los patrones de uso de las estaciones.

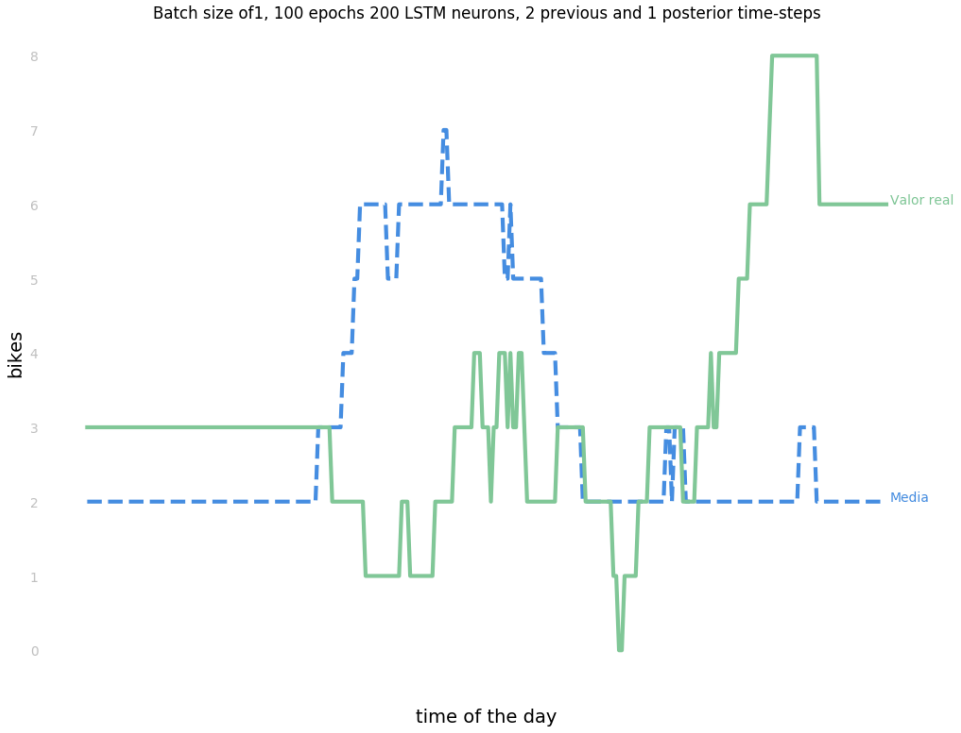


Figura 22: Comparación de la disponibilidad real y la estimada por la media

El **análisis de las predicciones utilizando la red neuronal** se ha realizado utilizando casos reales, tal y como si estuviera el modelo en producción. Del dataset recogido, no se utilizan los últimos días para el entrenamiento que son reservados para estas predicciones, de esta forma el modelo no los ha visto nunca y es un caso real de predicción.

El historial de precisión en el entrenamiento (figura 23) muestra que para el conjunto de datos de entrenamiento la precisión no es muy alta pero para el set de entrenamiento predicho la precisión incrementa poco a poco con el entrenamiento de la red.

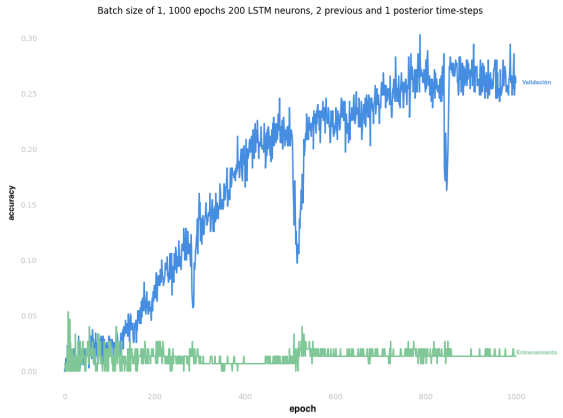


Figura 23: Historial de precisión durante el entrenamiento

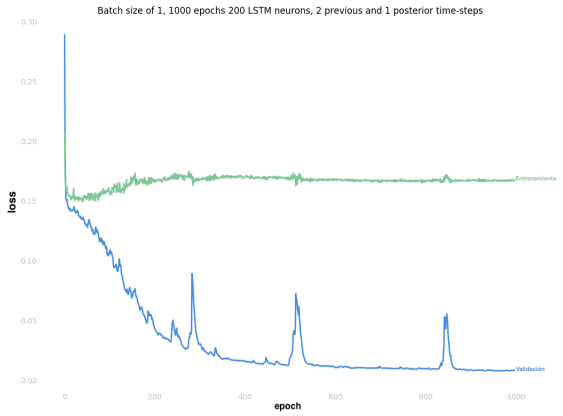


Figura 24: Historial de función de coste durante el entrenamiento

El modelo entrenado muestra valores casi correctos (figura 25), en esta muestra se ha entrenado el modelo con las 5 estaciones más cercanas a la estación de Zunzunegi. No se ha realizado con todas las estaciones al completo por todo el tiempo de procesado que ello conlleva aunque es realizable. Hasta más o menos el primer tercio del día la predicción es acertada en su totalidad. Pero es a partir de ese momento que los caminos difieren. Esto en parte, es debido a la recarga de estaciones por parte de los operarios del sistema. El pico de bicis que decrece en casi 12 unidades se debe a que las bicicletas han sido extraídas para ser llevadas a otra estación.

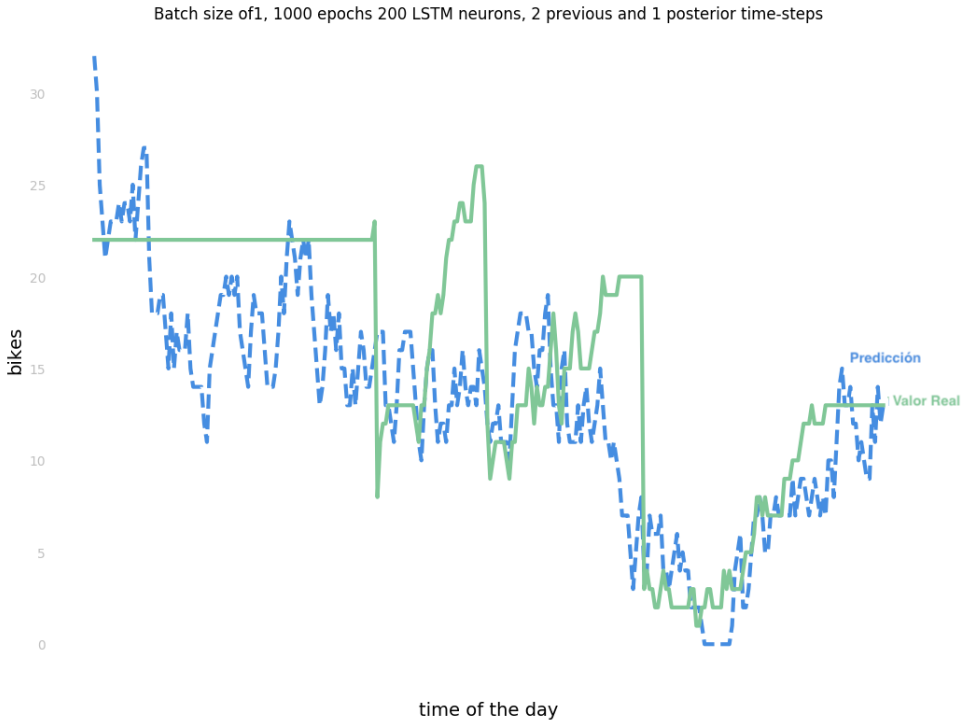


Figura 25: Predicción utilizando la red neuronal entrenada

En diversos casos se ha demostrado que el modelo es capaz de predecir valores mucho más precisos que el anterior método de predicción pero existen fallos: gran variabilidad debido al clima y recargas de estaciones por parte de los operarios.

7. Descripción de tareas

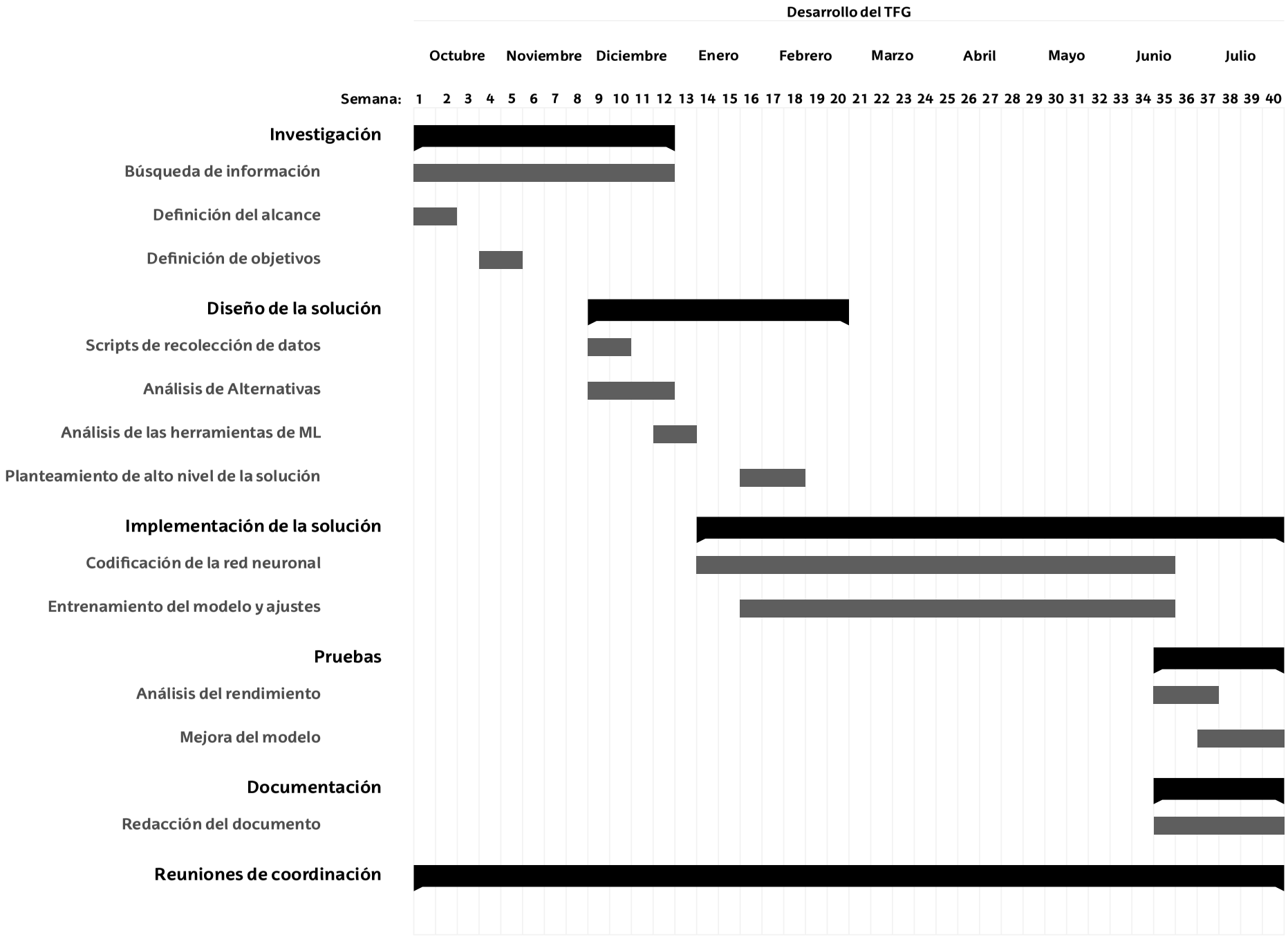
En esta sección se muestran las tareas desempeñadas a lo largo del trabajo. Se describen las diferentes etapas o fases que han sido necesarias en la elaboración del trabajo, detallándose diversos factores como las personas involucradas, los paquetes de trabajo con sus respectivas tareas y los hitos establecidos. Además, también se presenta un diagrama de Gantt con el fin de visualizar la planificación y donde se especifican otros aspectos como la duración de las tareas y la relación entre las mismas. A continuación se detallan las tareas realizadas y posteriormente se muestra el diagrama de Gantt con todas las tareas y el orden en el que se han realizado.

- **Investigación**
  - **Búsqueda de información**. Lectura e investigación sobre los conceptos iniciales necesarios para entender el funcionamiento de redes neuronales.
  - **Definición del alcance**. Establecimiento de los puntos que se realizarán en el proyecto.
  - **Definición de objetivos**. Establecer los límites del trabajo para cubrir el tiempo esperado.
- **Diseño de la solución**
  - **Scripts de recolección de datos**. Scripts con los que recoger los datos de disponibilidad de las distintas ciudades que se analizarán. Posterior colocación en el servidor BIPS de Aholab y configuración mediante la utilidad CRON de UNIX para que se ejecuten cada cinco minutos.
  - **Análisis de alternativas**. Estudio y pruebas con los métodos analizados en la sección para comprobar si pueden llegar a ser soluciones adecuadas.
  - **Planteamiento de alto nivel de la solución**
- **Implementación de la solución**
  - **Codificación de la red neuronal**. Lectura de los datos, formato necesario y adecuado para resolver la problemática.
  - **Entrenamiento del modelo y ajustes**. Realizado en el servidor AhoGPU de Aholab sobre GPUs para ahorrar tiempo.
- **Pruebas**
  - **Análisis del rendimiento**. Comparación de los resultados mediante gráficas como la precisión obtenida durante el entrenamiento, el error obtenido en el entrenamiento y predicciones de valores.

    - **Mejora del modelo**. Comparación de los resultados obtenidos con los esperados y modificación de los hiperparámetros para acercar el modelo a la solución esperada.

- **Documentación**. Redacción del documento final.
- **Reuniones de coordinación**. Reuniones bimensuales con el director del trabajo para analizar las dificultades encontradas y proponer soluciones. El horario se mantuvo durante todo el curso exceptuando los periodos de exámenes que no se realizaron.

7.1. Diagrama de Gantt


Desarrollo del TFG
Octubre
Noviembre
Diciembre
Enero
Febrero
Marzo
Abril
Mayo
Junio
Julio
Semana: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
Investigación
Búsqueda de información
Definición del alcance
Definición de objetivos
Diseño de la solución
Scripts de recolección de datos
Análisis de Alternativas
Análisis de las herramientas de ML
Planteamiento de alto nivel de la solución
Implementación de la solución
Codificación de la red neuronal
Entrenamiento del modelo y ajustes
Pruebas
Análisis del rendimiento
Mejora del modelo
Documentación
Redacción del documento
Reuniones de coordinación

8. Descripción del presupuesto

8.1. Recursos humanos

El coste de las horas dedicadas al proyecto se ha estimado según lo que se cobraría por un trabajo similar, para el alumno se han aproximado a horas de ingeniero junior y para el director de proyecto como ingeniero senior.

Cargo	Coste (€/h)	Dedicación (horas)	Coste total (€)
Director del proyecto	50	60	3.000
Ingeniero Junior	20	340	6.800
		Total	**9.800**

Figura 26: Desgloses de gastos derivados de recursos humanos

8.2. Recursos técnicos

El proyecto se ha desarrollado utilizando los siguientes equipos:

- **Ordenador personal**: Codificación de los programas y documentación.
- **Servidor BIPS**: Guarda los datos recogidos de los servicios de préstamo de bicicletas. Actualmente se están recogiendo datos veinticinco ciudades y ocupando los archivos unos 25GB de espacio. Se hará una estimación de su uso ya que sólo se guardan datos cada cinco minutos y no se usa de forma continuada.
- **Servidor AHOGPU**: Realiza el entrenamiento de las redes neuronales mediante GPUs NVIDIA Titan X y 64 GB de RAM para poder procesar los datos. Se hará una estimación de las horas de procesado utilizadas para el entrenamiento.

No se ha empleado ninguna licencia de software que requiera de compra, todas las librerías empleadas de machine learning son *open source* y no tienen coste alguno.

8.3. Gastos

Se entiende por gastos aquellos recursos que no puede ser reutilizado en ningún otro proyecto. Este trabajo no hace uso de licencias de pago ni requiere ningún equipo específico que no pueda ser usado en otros proyectos, por lo que **los gastos directos**

Concepto	Precio de Adquisición (€)	Vida útil (años)	Utilización (días)	Coste diario (€/día)	Coste total (€)
Ordenador personal	2.000	6	200	0,913	182,64
BIPS	5.000	6	2	2,28	4,56
AHOGPU	5.000	6	15	2,28	34,2
				Total	**284,76**

Figura 27: Material amortizable utilizado durante el proyecto

no amortizables del proyecto se consideran inexistentes. Si que existen gastos adicionales, como puede ser la conexión a internet, se imputará como una estimación de gasto indirecto en el resumen del presupuesto.

8.4. Resumen del presupuesto

Concepto	Coste (€)
Amortizaciones	284,76
Recursos Humanos	9.800
Total de Costes Directos	10.084,76
Estimación de Costes Indirectos	300
Total (sin I.V.A)	**10.384,76**
Total (con I.V.A)	**12.565,55**

Figura 28: Resumen del presupuesto

El coste total del proyecto asciende a 12.565,55€ (I.V.A. incluido).

9. Conclusiones

Tras la realización de este proyecto se han adquirido conocimientos fundamentales para asentar las bases del conocimiento sobre Machine Learning y redes neuronales. Con estos se pretende seguir avanzando en este trabajo ya que fue concebido como parte de un proyecto personal.

Si bien la calidad de las predicciones no es tan alta como se esperaba en un inicio antes de conocer el funcionamiento de las redes neuronales. Esto podría ser considerado un problema pero frente al anterior sistema de predicción es una mejora notable.

Una de las posibilidades que hace que el sistema no sea tan preciso es no tener en cuenta factores climatológicos al ser Bilbao una ciudad con un clima tan especial.

9.1. Resultados

Se puede apreciar por comparación a simple vista los resultados de las figuras 25 (red neuronal) y 22 (media) que los valores son mucho más acertados en el primer caso a pesar de los errores comentados por causas externas.

9.2. Trabajo Futuro

9.2.1. Implementaciones

Al haber entrenado la red neuronal ya está lista para ser desplegada en un producto. Existen varias vías en las que seguir trabajando:

- **Web**. Crear una *webapp* en la que se muestre un mapa con las estaciones del servicio. En cada una se mostrará la disponibilidad del momento junto a la predicción hecha por el modelo.
- **App**. Sustituir el mecanismo actual de predicción por el modelo como se describió al inicio del documento.

9.2.2. Mejoras

El programa que realiza el análisis de datos, los prepara y se los entrega a la red neuronal necesita bastante tiempo y recursos para ejecutarse. El programa ha corrido en un servidor con GPUs de altas prestaciones y 64GB de RAM, aun así en ciertos casos

se alcanzaban los límites y el programa dejaba de ejecutarse, la opción para arreglarlo es recortar el dataset.

Se podrían utilizar hilos paralelos para optimizar el tiempo de procesado empleando uno por cada estación en lugar de realizar todo el procesado de golpe como se está realizando ahora.

Por último, analizando los gráficos de disponibilidad de las estaciones se ve que en algunos casos hay picos que son causados por los empleados al vaciar o rellenar las estaciones para realizar un balance del sistema. Este tipo de variaciones no se han tenido en cuenta por lo que si no son hechas en intervalos regulares las predicciones se verán afectadas.

A partir de septiembre de 2018 el servicio para a tener un **nuevo administrador** que añadirá nuevas estaciones y bicicletas eléctricas. Esto supone una modificación porque habrá nuevas estaciones que no tendrán datos y el proveedor de los datos es probable que sea nuevo.

Por último, se pueden contemplar **distintas formas de entregar los datos en el entrenamiento**. Ahora se están dando por filas cada una con datos de cada estación por lo que la red en el momento del entrenamiento sólo conoce los datos de una única estación. Hay otra forma que sería en cada fila entregarle al modelo el estado de todas las estaciones en un momento determinado. Este cambio puede ser interesante para conocer cómo se comporta el servicio en determinados momentos y ver el balance que se produce.

Toda la documentación junto con el código utilizado se encuentra en un repositorio en GitHub (`https://github.com/javierdemartin/neural-bikes`) en el que se sigue avanzando en el proyecto de forma continua.

Bibliografía

[1] Ian Goodfellow and Yoshua Bengio and Aaron Courville.
Deep Learning. MIT Press. 2016.
https://www.deeplearningbook.org

[2] Jason Brownlee
Long Short-Term Memory Networks with Python. 2017.
https://machinelearningmastery.com/lstms-with-python/

[3] Jason Brownlee
Long Short-Term Memory Networks with Python. 2017.
https://machinelearningmastery.com/lstms-with-python/

[4] Christopher Olah
Understanding LSTM Networks. 2015.
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[5] Christopher Olah
What are hyperparameters in machine learning?. 2016.
https://www.quora.com/What-are-hyperparameters-in-machine-learning

[6] Jason Mayes
Machine Learning 101. 2017.
https://www.quora.com/What-are-hyperparameters-in-machine-learning

[7] Andrew Ng
Machine Learning. 2017.
https://www.quora.com/What-are-hyperparameters-in-machine-learning

10. Anexo I: Código

A continuación se añaden los bloques de código más relevantes en la solución y una breve descripción de su funcionamiento.

10.1. Recolector de datos de disponibilidad

Este primer bloque de código describe el programa en Python que se encarga de recoger los datos de disponibilidad de la URL encontrada en el catálogo de datos abiertos de Bilbao y crear el historial.

```
import xml.etree.ElementTree as ET
import urllib2
import string
import time
import os
import datetime
import MySQLdb
import collections
import codecs

# URL containing the XML feed
url = "http://www.bilbao.eus/WebServicesBilbao/WSBilbao?s=ODPRESBICI&u=
    ↪ OPENDATA&p0=A&p1=A"

# Get current weekday
weekno = -1
weekday = ""
weekdays = ["MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY", "
    ↪ SATURDAY", "SUNDAY"]
weekno = datetime.datetime.today().weekday()
weekday = weekdays[weekno]

request = urllib2.Request(url, headers={"Accept" : "application/xml"})
u = urllib2.urlopen(request)

tree = ET.parse(u)
root = tree.getroot()

id = ""
stationName = ""
```

```
freeBikes = ""
freeDocks = ""
totalQuery = ""

for child in root:
    for detalle in child:

        station_aux = []

        query = ""

        for x in detalle:

            if x.tag == "NOMBRE":
                id = x.text.split("-")[0]
                stationName = x.text.split("-")[1]
            elif x.tag == "ALIBRES":
                freeBikes = x.text
            elif x.tag == "BLIBRES":
                freeDocks = x.text

        query = time.strftime("%Y/%m/%d %H:%M") + "," + weekday + "," +
            ↪ id + "," + stationName + "," + freeBikes + "," + freeDocks +
            ↪  "\n"

        if (",,,") not in query:
                totalQuery += query

print totalQuery

with codecs.open("/home/aholab/javier/tfg/tools/parsers/data/Bilbao.txt",
    ↪ "a", "utf8") as file:
        file.write(totalQuery)
```

10.2. Entrenamiento de la red neuronal

Programa principal que se encarga a llamar al código que realiza el entrenamiento de la red neuronal con los distintos parámetros a probar para realizar tests.

```
import csv
import glob
import matplotlib
matplotlib.use('Agg')
import sys
import matplotlib.pyplot as plt
import os
import gc

# Parameters
# lstm | batch | epochs | n_in | n_out
```

```
epochs = 100
batch_size = 100
lstm_neurons = 200
n_in = 576
n_out = 288


batches = [50]

os.system("rm -rf data_gen/ model/ plots/ encoders/")

class col:
    blue = '\033[94m'
    ENDC = '\033[0m'

for intento in batches:


        os.system("rm -rf model/")

        os.system("python3 script.py " + str(lstm_neurons) + " " + str(
            ↪ intento) + " " + str(epochs) + " " + str(n_in) + " " + str(
            ↪ n_out))

        print("Terminado\a")
        gc.collect()

def generate_plot(plot):

        # Check if the folder for the plots exists, if not create it
        if os.path.isdir("/plots") == False:
                os.system("mkdir plots")
                os.system("chmod 775 plots")

        plt.figure(figsize=(12, 9))
        ax = plt.subplot(111)
        ax = plt.axes(frameon=False)

        plots = []
        legends = []

        for filename in glob.glob('data_gen/' + plot + '/*'):
                print("Reading " + filename)

                with open(filename, 'r') as csvfile:

                        acc = []

                        spamreader = csv.reader(csvfile, delimiter='\n')
                        for row in spamreader:
```

```
                        acc.append(float(row[0]))

                    legends.append(str(int(int(filename.split('data_gen/'
                        ↪  + plot + '/')[1].split("_")[1]) / 288)) + "
                        ↪ Batch Size")

                    aux = plt.plot(acc, label = filename.split('data_gen
                        ↪ /' + plot + '/')[1])
                    plots.append(aux)

                    print(col.blue, "##### Average of " + plot + ": " +
                        ↪ str(sum(acc)/ len(acc)) + " MIN: " + str(min(
                        ↪ acc)) + " MAX: " + str(max(acc)), col.ENDC)


        title = plot + " Batch size of" + str(batch_size) + ", " + str(
            ↪ epochs) + " epochs " + str(lstm_neurons) + " LSTM neurons, "
            ↪  + str(int(n_in/288)) + " previous and " + str(int(n_out
            ↪ /288)) + " posterior time-steps"

        plt.title(plot)
        plt.legend(legends)
        plt.savefig("plots/" + plot + ".png", bbox_inches="tight")

generate_plot('acc')
generate_plot('mean_squared_error')
generate_plot('prediction')
generate_plot('loss')
```

Programa principal, realiza todo lo descrito en la solución seleccionada. Lee los datos, da el formato adecuado y los entrega a la red neuronal para comenzar con el entrenamiento.

```
from math import sqrt
import math
import numpy
import matplotlib
matplotlib.use('Agg') # Needed when running on headless server
import sys
import matplotlib.pyplot as plt
from numpy import concatenate
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.metrics import mean_squared_error
from pandas import concat,DataFrame
from keras.models import Sequential
from keras.utils import plot_model, to_categorical
from keras.layers import Dense, LSTM, Dropout, Activation
from datetime import datetime
import datetime
from numpy import argmax
import matplotlib.ticker as ticker
```

```
import pandas.core.frame
from sklearn.externals import joblib
import os
import csv
import gc

os.system("reset") # Clears the screen

# os.system("rm -rf model/")


# -- Global Parameters & Configuration
# ---------------------------------------

save_model_img = False
# Prints the array
is_in_debug = True
# Station to analyze and read the data from
stationToRead = 'PLAZA ARRIAGA'
weekdays = ["MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY", "
    ↪ SATURDAY", "SUNDAY"]

list_of_stations = ["PLAZA LEVANTE", "IRUÑA", "AYUNTAMIENTO","PLAZA
    ↪ ARRIAGA", "SANTIAGO COMPOSTELA", "PLAZA REKALDE", "DR. AREILZA", "
    ↪ ZUNZUNEGI", "ASTILLERO", "EGUILLOR", "S. CORAZON", "PLAZA INDAUTXU
    ↪ ", "LEHENDAKARI LEIZAOLA", "CAMPA IBAIZABAL", "POLID. ATXURI", "SAN
    ↪  PEDRO", "KARMELO", "BOLUETA", "OTXARKOAGA", "OLABEAGA", "SARRIKO",
    ↪  "HEROS", "EGAÑA", "P.ETXEBARRIA", "TXOMIN GARAT", "ABANDO", "
    ↪ ESTRADA CALEROS", "EPALZA", "IRALA", "S.ARANA", "C.MARIA"]

list_hours = ['00:00', '00:05', '00:10', '00:15', '00:20', '00:25',
    ↪ '00:30', '00:35', '00:40',
 '00:45', '00:50', '00:55', '01:00', '01:05', '01:10', '01:15',
 '01:20', '01:25', '01:30', '01:35', '01:40', '01:45',
 '01:50', '01:55', '02:00', '02:05', '02:10', '02:15', '02:20',
 '02:25', '02:30', '02:35', '02:40', '02:45', '02:50', '02:55', '03:00',
    ↪ '03:05',
 '03:10', '03:15', '03:20', '03:25', '03:30', '03:35', '03:40', '03:45',
    ↪ '03:50',
 '03:55', '04:00', '04:05', '04:10', '04:15', '04:20', '04:25', '04:30',
    ↪ '04:35',
 '04:40', '04:45', '04:50', '04:55', '05:00', '05:05', '05:10', '05:15',
    ↪ '05:20',
 '05:25', '05:30', '05:35', '05:40', '05:45', '05:50', '05:55', '06:00',
    ↪ '06:05',
 '06:10', '06:15', '06:20', '06:25', '06:30', '06:35', '06:40', '06:45',
    ↪ '06:50',
 '06:55', '07:00', '07:05', '07:10', '07:15', '07:20', '07:25', '07:30',
 '07:35', '07:40', '07:45', '07:50', '07:55', '08:00', '08:05', '08:10',
    ↪ '08:15',
 '08:20', '08:25', '08:30', '08:35', '08:40', '08:45', '08:50', '08:55',
```

```
    ↪ '09:00',
 '09:05', '09:10', '09:15', '09:20', '09:25', '09:30', '09:35', '09:40',
 '09:45', '09:50', '09:55', '10:00', '10:05', '10:10', '10:15', '10:20',
    ↪ '10:25',
 '10:30', '10:35', '10:40', '10:45', '10:50', '10:55', '11:00',
 '11:05', '11:10', '11:15', '11:20', '11:25', '11:30', '11:35',
 '11:40', '11:45', '11:50', '11:55', '12:00', '12:05', '12:10', '12:15',
 '12:20', '12:25', '12:30', '12:35', '12:40', '12:45', '12:50', '12:55',
    ↪ '13:00',
 '13:05', '13:10', '13:15', '13:20', '13:25', '13:30', '13:35', '13:40',
 '13:45', '13:50', '13:55', '14:00', '14:05', '14:10', '14:15',
 '14:20', '14:25', '14:30', '14:35', '14:40',
 '14:45', '14:50', '14:55', '15:00', '15:05', '15:10', '15:15',
 '15:20', '15:25', '15:30', '15:35', '15:40', '15:45', '15:50', '15:55',
    ↪ '16:00',
 '16:05', '16:10', '16:15', '16:20', '16:25', '16:30', '16:35', '16:40',
 '16:45', '16:50', '16:55', '17:00', '17:05', '17:10', '17:15', '17:20',
 '17:25', '17:30', '17:35', '17:40', '17:45', '17:50', '17:55',
 '18:00', '18:05', '18:10', '18:15', '18:20', '18:25', '18:30',
 '18:35', '18:40', '18:45', '18:50', '18:55', '19:00', '19:05', '19:10',
 '19:15', '19:20', '19:25', '19:30', '19:35', '19:40', '19:45', '19:50',
 '19:55', '20:00', '20:05', '20:10', '20:15', '20:20', '20:25', '20:30',
    ↪ '20:35',
 '20:40', '20:45', '20:50', '20:55', '21:00', '21:05', '21:10', '21:15',
 '21:20', '21:25', '21:30', '21:35', '21:40', '21:45', '21:50', '21:55',
    ↪ '22:00',
 '22:05', '22:10', '22:15', '22:20', '22:25', '22:30', '22:35', '22:40',
    ↪ '22:45',
 '22:50', '22:55', '23:00', '23:05', '23:10', '23:15', '23:20', '23:25',
 '23:30', '23:35', '23:40', '23:45', '23:50', '23:55']

lstm_neurons = int(sys.argv[1]) # 50
batch_size = 1
epochs = int(sys.argv[3]) # 30
n_in = int(sys.argv[4]) # 10
n_out = int(sys.argv[5]) # 10
new_batch_size = int(sys.argv[2]) * n_in #1000

file_name = str(lstm_neurons) + "_" + str(int(new_batch_size/n_in)) + "_"
    ↪ + str(epochs) + "_" + str(n_in)

model_name = "model_" + str(lstm_neurons) + "_neurons_" + str(
    ↪ new_batch_size) + "_batch_" + str(epochs) + "_epochs_" + str(n_in)
    ↪ + "_n_in_" + str(n_out) + "_n_out"

print(str(lstm_neurons) + " neurons in the LSTM shape, " + str(epochs) + "
    ↪  epochs" + str(n_in) + " previous time-steps and " + str(
    ↪ new_batch_size) + " batch size and" + str(n_out) + " n_out")

################################
# Classes and Functions
```

```
##############################

# Colorful prints in the terminal
class col:
	HEADER = '\033[95m'
	blue = '\033[94m'
	green = '\033[92m'
	yellow = '\033[93m'
	FAIL = '\033[91m'
	ENDC = '\033[0m'
	BOLD = '\033[1m'

# Formatted output
def print_smth(description, x):

	if is_in_debug == True:
		print("", col.yellow)
		print(description)
		print("----------------------------------------", col.ENDC)
		print(x)
		print(col.yellow, "----------------------------------------",
		    ↪ col.ENDC)


# Print an array with a description and its size
def print_array(description, array):

	if is_in_debug == True:
		print("", col.yellow)
		print(description, " ", array.shape)
		print("------------------------------", col.ENDC)
		print(array)
		print(col.yellow, "------------------------", col.ENDC)


def one_plot(xlabel, ylabel, plot_1, plot_2, name, dia):

	min_y = min(plot_1)
	max_y = max(plot_1)

	plt.figure(figsize=(12, 9))
	ax = plt.subplot(111)
	ax = plt.axes(frameon=False)

	ax.spines["top"].set_visible(False)
	ax.spines["bottom"].set_visible(False)
	ax.spines["right"].set_visible(False)
	ax.spines["left"].set_visible(False)
	ax.get_xaxis().tick_bottom()
	ax.get_yaxis().tick_left()
```

```
        plt.xlabel(xlabel, color = 'silver', fontsize = 17)
        plt.ylabel(ylabel, color = 'silver', fontsize = 17)

        lines = plt.plot(plot_1, plot_2, label = 'train', color = '#458DE1
            ↪ ')

        # plt.xticks(dataset.loc [ dataset [ 'datetime' ] == dia ].values
            ↪ [:,1][::24], dataset.loc [ dataset [ 'datetime' ] == dia ].
            ↪ values[:,1][::24])

        plt.setp(lines, linewidth=2)

        texto = stationToRead + " Batch size of " + str(new_batch_size) +
            ↪ ", " + str(epochs) + " epochs " + str(lstm_neurons) + " LSTM
            ↪  neurons, " + str(int(n_in/288)) + " previous and " + str(
            ↪ int(n_out/288)) + " posterior time-steps"
        plt.title(texto,color="black") #, alpha=0.3)
        plt.tick_params(bottom="off", top="off", labelbottom="on", left="
            ↪ off", right="off", labelleft="on", colors = 'silver')

        # plt.show()
        plt.savefig("plots/" + name + ".png", bbox_inches="tight")

        plt.close()
        print(col.HEADER + "> " + name + " plot saved" + col.ENDC)

# Stilish the plot without plot ticks,
def prepare_plot(xlabel, ylabel, plot_1, plot_2, name):

        min_y = min(plot_1)
        max_y = max(plot_1)

        plt.figure(figsize=(12, 9))
        ax = plt.subplot(111)
        ax = plt.axes(frameon=False)

        ax.spines["top"].set_visible(False)
        ax.spines["bottom"].set_visible(False)
        ax.spines["right"].set_visible(False)
        ax.spines["left"].set_visible(False)
        ax.get_xaxis().tick_bottom()
        ax.get_yaxis().tick_left()

        plt.xlabel(xlabel, color = 'silver', fontsize = 14)
        plt.ylabel(ylabel, color = 'silver', fontsize = 14)

        lines = plt.plot(plot_1, label = 'train', color = '#458DE1')

        if len(plot_2) > 0:
                lines += plt.plot(plot_2, label = 'test', color = '#80C797')
```

```
        plt.setp(lines, linewidth=2)

        plt.text((len(plot_1) - 1) * 1.005,
                plot_1[len(plot_1) - 1] + 0.01,
                "Predicted", color = '#458DE1')

        if len(plot_2) > 0:
               plt.text((len(plot_2) - 1) * 1.005,
                plot_2[len(plot_2) - 1],
                "Real", color = '#80C797')

        texto = "Batch size of " + str(new_batch_size) + ", " + str(epochs)
            ↪  + " epochs " + str(lstm_neurons) + " LSTM neurons, " + str(
            ↪ int(n_in/288)) + " previous and " + str(int(n_out/288)) + "
            ↪ posterior time-steps"
        plt.title(texto,color="black") #, alpha=0.3)
        plt.tick_params(bottom="off", top="off", labelbottom="on", left="
            ↪ off", right="off", labelleft="on", colors = 'silver')

        # plt.show()
        plt.savefig("plots/" + name + ".png", bbox_inches="tight")

        plt.close()
        print(col.HEADER + "> " + name + " plot saved" + col.ENDC)

def plot_availability(xlabel, ylabel, plot_1, plot_2, name, label1, label2
    ↪ ):

        min_y = min(plot_1)
        max_y = max(plot_1)

        plt.figure(figsize=(12, 9))
        ax = plt.subplot(111)
        ax = plt.axes(frameon=False)

        ax.spines["top"].set_visible(False)
        ax.spines["bottom"].set_visible(False)
        ax.spines["right"].set_visible(False)
        ax.spines["left"].set_visible(False)
        ax.get_xaxis().tick_bottom()
        ax.get_yaxis().tick_left()

        plt.xlabel(xlabel, color = 'black', fontsize = 14)
        plt.ylabel(ylabel, color = 'black', fontsize = 14)

        lines = plt.plot(plot_1, linestyle = '--', label = 'train', color =
            ↪  '#458DE1')
        lines += plt.plot(plot_2, label = 'test', color = '#80C797')
```

```
        plt.xticks(range(len(plot_1)), hour_encoder.classes_) #, rotation='
            ↪ vertical')

        start, end = ax.get_xlim()
        ax.xaxis.set_ticks(numpy.arange(start, end, 0.125))

        plt.setp(lines, linewidth=3)

        plt.text((len(plot_1) - 1) * 1.005,
                plot_1[len(plot_1) - 1] + 0.01,
                label1 ,color = '#458DE1')

        if len(plot_2) > 0:
               plt.text((len(plot_2) - 1) * 1.005,
                plot_2[len(plot_2) - 1],
                label2, color = '#80C797')

        texto = "Batch size of " + str(new_batch_size) + ", " + str(epochs)
            ↪  + " epochs " + str(lstm_neurons) + " LSTM neurons, " + str(
            ↪ int(n_in/288)) + " previous and " + str(int(n_out/288)) + "
            ↪ posterior time-steps"
        plt.title(texto,color="black") #, alpha=0.3)
        plt.tick_params(bottom="off", top="off", labelbottom="on", left="
            ↪ off", right="off", labelleft="on") #, colors = 'silver')

        # plt.show()
        plt.savefig("plots/" + name + ".png", bbox_inches="tight")

        plt.close()
        print(col.HEADER + "> " + name + " plot saved" + col.ENDC)

# Check if the current directory exists, if not create it.
def check_directory(path):
        if os.path.isdir(path) == False:
               os.system("mkdir " + path)
               os.system("chmod 775 " + path)

def save_file_to(directory, fileName, data):

        with open(directory + fileName, 'w') as file:
               wr = csv.writer(file, delimiter = '\n')
               wr.writerow(data)

# Convert series to supervised learning
# Arguments
# * [Columns]> Array of strings to name the supervised transformation
def series_to_supervised(columns, data, n_in=1, n_out=1, dropnan=False):

        print("COLUMNAS " + str(columns))

        n_vars = 1 if type(data) is list else data.shape[1]
```

```
        dataset = DataFrame(data)
        cols, names = list(), list()
        # input sequence (t-n, ... t-1)
        for i in range(n_in, 0, -1):
                cols.append(dataset.shift(i))
                names += [(columns[j] + '(t-%d)' % (i)) for j in range(
                    ↪ n_vars)]

        # forecast sequence (t, t+1, ... t+n)
        for i in range(0, n_out):
                cols.append(dataset.shift(-i))
                if i == 0:
                        #names += [('var%d(t)' % (j+1)) for j in range(n_vars
                            ↪ )]
                        names += [(columns[j] + '(t)') for j in range(n_vars)
                            ↪ ]
                else:
                        names += [(columns[j] + '(t+%d)' % (i)) for j in
                            ↪ range(n_vars)]
        # put it all together
        agg = concat(cols, axis=1)
        agg.columns = names
        # drop rows with NaN values

        del dataset
        del cols
        gc.collect()

        print("Droppin NAN")

        if dropnan:
                agg.dropna(inplace=True)

        print_array("Reframed dataset after converting series to supervised
            ↪ ", agg.head())

        return agg

# Create the model, used two times
# (1) Batch training the model (batch size = specified as input)
# (2) Making online predictions (batch size = 1)
def create_model(batch_sizee, statefulness):

        model = Sequential()
        # model.add(LSTM(lstm_neurons, batch_input_shape=(batch_sizee,
            ↪ train_x.shape[1], train_x.shape[2]), stateful=statefulness,
            ↪ return_sequences=True))
        model.add(LSTM(lstm_neurons, input_shape=(train_x.shape[1], train_x
            ↪ .shape[2]), stateful=statefulness, return_sequences=True))
        model.add(LSTM(lstm_neurons, return_sequences = True))
        model.add(LSTM(lstm_neurons))
```

```
        model.add(Dense(n_out))
        # model.add(Activation('softmax'))
        model.compile(loss='mae', optimizer='adam', metrics = ['mse', 'acc
            ↪ '])

        return model

def prepare_sequences(x_train, y_train, window_length):
        windows = []
        windows_y = []
        for i, sequence in enumerate(x_train):
                len_seq = len(sequence)
                for window_start in range(0, len_seq - window_length + 1):
                        window_end = window_start + window_length
                        window = sequence[window_start:window_end]
                        windows.append(window)
                        windows_y.append(y_train[i])
        return numpy.array(windows), numpy.array(windows_y)


# Generate same number of columns for the categorization problem as bikes
    ↪ are in this station
def generate_column_array(columns, max_bikes) :

        columns = columns

        for i in range(0, max_bikes + 1):
                columns.append(str(i) + '_free_bikes')

        return columns

# input: number of the day which the average availability is wanted
# output: list with the average for the station
def calculate_average_for(values):

        average_availability = []

        for i in range(288):

                selected = values[numpy.where(values[:,1] == i)]

                average_availability.append(int(sum(selected[:,4]) / len(
                    ↪ selected[:,4])))

        return average_availability


################################################################################
    ↪
# Data preparation
################################################################################
```

```
    ↪

check_directory("model")
check_directory("plots")
check_directory("data_gen")
check_directory("data_gen/acc")
check_directory("data_gen/loss")
check_directory("data_gen/mean_squared_error")
check_directory("data_gen/prediction")
check_directory("encoders")

print(col.HEADER + "Data reading and preparation" + col.ENDC)

#-------------------------------------------------------------------------------
    ↪
# File reading, drop non-relevant columns like station id, station name...
#-------------------------------------------------------------------------------
    ↪

dataset = pandas.read_csv('data/Bilbao.txt')
dataset.columns = ['datetime', 'weekday', 'id', 'station', 'free_docks', '
    ↪ free_bikes']

dataset = dataset.drop(dataset.index[range(400000)])


print_array("Read dataset", dataset)

# dataset = dataset[dataset['station'].isin([stationToRead])]


print(col.HEADER, "> Data from " + dataset['datetime'].iloc[0], " to ",
    ↪ dataset['datetime'].iloc[len(dataset) - 1], col.ENDC)

print_array("Read dataset", dataset.head())
print_array("Read dataset", dataset)

dataset.drop(dataset.columns[[2,4]], axis = 1, inplace = True) # Remove ID
    ↪  of the sation and free docks
dataset = dataset.reset_index(drop = True)


print_array("Read dataset", dataset)


values = dataset.values

#-------------------------------------------------------------------------------
    ↪
#-- Data reading
    ↪ ---------------------------------------------------------------
```

```
#
# Split datetime column into day of the year and time
#------------------------------------------------------------------------------
    ↪

times = [x.split(" ")[1] for x in values[:,0]]

dataset['datetime'] = [datetime.datetime.strptime(x, '%Y/%m/%d %H:%M').
    ↪ timetuple().tm_yday for x in values[:,0]]

dataset.insert(loc = 1, column = 'time', value = times)

print_array("Dataset with unwanted columns removed", dataset.head(15))

dataset = dataset[dataset['time'].isin(list_hours)]

# Get the last n_in samples to predict
today = datetime.datetime.now().timetuple().tm_yday - 1
oldest_day_to_search = today - int(n_in/288)

list_days_to_save = []

for i in range(int(n_in/288)):
       list_days_to_save.append(oldest_day_to_search + i)

print("Recogiendo los dias " + str(list_days_to_save) + " y los de hoy ("
    ↪ + str(today) + ")")
u
datos = dataset[dataset['datetime'].isin(list_days_to_save)][dataset['
    ↪ station'].isin([stationToRead])]
dia_hoy = dataset[dataset['datetime'].isin([today])][dataset['station'].
    ↪ isin([stationToRead])]


print_array("Dataset with unwanted columns removed 2", dataset.head(15))

datos_media = dataset[dataset['station'].isin([stationToRead])].values

# average = calculate_average_for()

values = dataset.values

separated_stations = []

for station_to in list_of_stations:

       separated_stations.append(dataset[dataset['station'].isin([
           ↪ station_to])].values)


misDatos = dataset[dataset['datetime'].isin([datetime.datetime.now().
```

```
    ↪ timetuple().tm_yday])].values


#-----------------------------------------------
#-- Data encoding ----------------------------

hour_encoder = LabelEncoder() # Encode columns that are not numbers
hour_encoder.fit(list_hours)
weekday_encoder = LabelEncoder() # Encode columns that are not numbers
weekday_encoder.fit(weekdays)
station_encoder = LabelEncoder() # Encode columns that are not numbers
station_encoder.fit(list_of_stations)

print_smth("HOUR ENCODER", hour_encoder.classes_)
print(len(hour_encoder.classes_))

print_smth("WEEKDAY ENCODER", weekday_encoder.classes_)
print(len(weekday_encoder.classes_))

print_smth("STATION ENCODER", station_encoder.classes_)
print(len(station_encoder.classes_))


max_bikes = int(max(values[:,4])) # Maximum number of bikes a station
    ↪ holds

del dataset
del values
print("Deleted from memory dataset")

for sta in separated_stations:

       sta[:,1] = hour_encoder.transform(sta[:,1]) # Encode HOUR as an
           ↪ integer value
       sta[:,2] = weekday_encoder.transform(sta[:,2]) # Encode HOUR as int
       sta[:,3] = station_encoder.transform(sta[:,3])
       sta = sta.astype('float')


datos_media[:,1] = hour_encoder.transform(datos_media[:,1])
datos_media[:,2] = weekday_encoder.transform(datos_media[:,2])
datos_media[:,3] = station_encoder.transform(datos_media[:,3])

average = calculate_average_for(datos_media)

max_day = float(max(sta[:,0]))
max_hour = float(max(sta[:,1]))
max_wday = float(max(sta[:,2]))

# Comprobar que no quedan huecos en el dataset, si hay recrearlos
#-----------------------------------------------------------------
```

```
for estacion in separated_stations:

	print(">>> " + str(estacion))

	for i in range(288):
		# Si no esta en el final
		if estacion[i][1] != 287:
			if estacion[i][1] != (estacion[i+1][1] - 1):
				print(">>>> No coincide")

#-----------------------------------------------------------------


print("MAX BIKES " + str(max_bikes))

scaler = MinMaxScaler(feature_range=(0,1)) # Normalize values
scaler.data_max_ = [1.0, 1.0, 1.0, 1.0, 31.0, 35.0]

for i in range(len(separated_stations)):
	new_dataset = DataFrame()
	new_dataset['hour_sin'] = numpy.sin(2. * numpy.pi *
	    ↪ separated_stations[i][:,1].astype('float') / (max_hour + 1))
	new_dataset['hour_cos'] = numpy.cos(2. * numpy.pi *
	    ↪ separated_stations[i][:,1].astype('float') / (max_hour + 1))
	new_dataset['wday_sin'] = numpy.sin(2. * numpy.pi *
	    ↪ separated_stations[i][:,2].astype('float') / (max_wday + 1))
	new_dataset['wday_cos'] = numpy.cos(2. * numpy.pi *
	    ↪ separated_stations[i][:,2].astype('float') / (max_wday + 1))
	new_dataset['station'] = separated_stations[i][:,3]
	new_dataset['free_bikes'] = separated_stations[i][:,4]

	new_dataset = scaler.fit_transform(new_dataset)

	scaler.data_max_ = [1.0, 1.0, 1.0, 1.0, 31.0, 35.0]

	print_smth("NEW DATASET", new_dataset)

	# print_smth("APPENDED", new_dataset)

	# final_stations.append(new_dataset)

	separated_stations[i] = new_dataset

	del new_dataset
	gc.collect()

print("Removed from memory separated_stations")
gc.collect()

print_smth("FINAL DATASET", separated_stations)
```

```
separated_stations = [ scaler.fit_transform(x) for x in separated_stations
    ↪ ]

print_smth("Dataset with normalized values", separated_stations)

#-------------------------------------------------------------
# Generate the columns list for the supervised transformation
#-------------------------------------------------------------
columns = ['hour_sin', 'hour_cos', 'wday_sin', 'wday_cos', 'station', '
    ↪ free_bikes']

for i in range(len(separated_stations)):

       print_smth("pre appending " + str(i), separated_stations[i])

       separated_stations[i] = series_to_supervised(columns,
           ↪ separated_stations[i], n_in, n_out, True)

       print_smth("Appending", separated_stations[i])

       gc.collect()

for i in separated_stations:
       print_smth("HEY PROBANDO CABRON", i)
       print(len(i))

final_drop = []

for i in range(0,n_out):

       position = (len(columns)) * (n_in + i)

       print("Position " + str(position))


       to_drop = range(position, position + 5)
       print(str(i) + " RANGE " + str(to_drop))

       final_drop.append(separated_stations[0].columns[to_drop].tolist())

final_drop = [val for sublist in final_drop for val in sublist]

print_smth("Lista columnas a eliminar", final_drop)

for i in range(len(separated_stations)):

       separated_stations[i].drop(final_drop, axis=1, inplace=True)

       separated_stations[i] = separated_stations[i].values
```

```
lon = 0

for i in separated_stations:

       lon += i.shape[0]


result = separated_stations[0]

for i in range(1,len(separated_stations)):

       result = numpy.append(result, separated_stations[i], axis = 0)

       gc.collect()


shapo = result.shape[1]

values = result

values = values.reshape((lon,shapo))

print_array("VALUES FINAL", values)



# shuffle
numpy.random.shuffle(values)


print_array("Shuffled values", values)

# ------------------------------------------------------------
# -- Calculate the number of samples for each set
# ------------------------------------------------------------

train_size, test_size, prediction_size = int(len(values) * 0.65) , int(len
    ↪ (values) * 0.3), int(len(values) * 0.0)

train_size = int(int(train_size / new_batch_size) * new_batch_size)
test_size = int(int(test_size / new_batch_size) * new_batch_size)
prediction_size = int(int(prediction_size / new_batch_size) *
    ↪ new_batch_size)

print("Train size " + str(train_size) + " Prediction size " + str(
    ↪ prediction_size))

# Divide between train and test sets
train, test, prediction = values[0:train_size,:], values[train_size:
```

```
    ↪ train_size + test_size, :], values[train_size + test_size:
    ↪ train_size + test_size + prediction_size, :]

output_vals = range(len(columns) * (n_in), values.shape[1])
# print("OUTPUT VALS " + str(output_vals))


train_x, train_y = train[:,range(0,len(columns) * n_in)], train[:,
    ↪ output_vals]
test_x, test_y = test[:,range(0,len(columns) * n_in)], test[:,output_vals]
prediction_x, prediction_y = prediction[:,range(0,len(columns) * n_in)],
    ↪ prediction[:,output_vals]

print_array(">>> TRAIN_X 2 ", train_x)
print_array(">>> TRAIN_Y 2 ", train_y)
print_array(">>> TEST_X 2 ", test_x)
print_array(">>> TEST_Y 2 ", test_y)
print_array(">>> PREDICTION_X 2 ", prediction_x)
print_array(">>> PREDICTION_Y 2 ", prediction_y)

# reshape input to be [samples, time_steps, features]
train_x = train_x.reshape((train_x.shape[0], n_in, len(columns)))
test_x = test_x.reshape((test_x.shape[0], n_in, len(columns)))
prediction_x = prediction_x.reshape((prediction_x.shape[0], n_in, len(
    ↪ columns)))

print_array("TRAIN_X 2 ", train_x)
print_array("TRAIN_Y 2 ", train_y)
print_array("TEST_X 2 ", test_x)
print_array("TEST_Y 2 ", test_y)
print_array("PREDICTION_X 2 ", prediction_x)
print_array("PREDICTION_Y 2 ", prediction_y)


train_x_aux = []
train_y_aux = []

for i in range(0,int(len(train_x) / n_in)):
       train_x_aux.append(train_x[i])
       train_y_aux.append(train_y[i])

train_x = train_x_aux
train_y = train_y_aux

test_x_aux = []
test_y_aux = []


for i in range(0,int(len(test_x) / n_in)):

       test_x_aux.append(test_x[i])
```

```
        test_y_aux.append(test_y[i])

test_x = test_x_aux
test_y = test_y_aux

prediction_x_aux = []
prediction_y_aux = []

for i in range(0,int(len(prediction_x) / n_in)):

        prediction_x_aux.append(prediction_x[i])
        prediction_y_aux.append(prediction_y[i])

prediction_x = prediction_x_aux
prediction_y = prediction_y_aux

for i in range(0,10):

        print_array(str(i), train_x[i])

train_x = numpy.asarray(train_x)
train_y = numpy.asarray(train_y)

test_x = numpy.asarray(test_x)
test_y = numpy.asarray(test_y)

prediction_x = numpy.asarray(prediction_x)
prediction_y = numpy.asarray(prediction_y)


print(col.blue, "[Dimensions]> ", "Train X ", train_x.shape, "Train Y ",
    ↪ train_y.shape, "Test X ", test_x.shape, "Test Y ", test_y.shape, "
    ↪ Prediction X", prediction_x.shape, "Prediction Y", prediction_y.
    ↪ shape, col.ENDC)

print_array("TRAIN_X", train_x)

print_array("TRAIN_Y", train_y)

# ----------------------------------------------
# -- Neural Network----------------------------
#
# Model definition
#----------------------------------------------

# If the model is already trained don't do it again
if os.path.isfile("model/" + model_name + ".h5") == False:

        # As the model doesn't exists create the folder to save it there
```

```
        model = create_model(int(new_batch_size/n_in), False)

        list_acc = []
        list_loss = []
        list_mse = []

        history = model.fit(train_x, train_y, batch_size=int(new_batch_size
            ↪  / n_in), epochs=epochs, validation_data=(test_x, test_y),
            ↪ verbose=2, shuffle = True)

        save_file_to("data_gen/acc/", file_name, history.history['acc'])
        save_file_to("data_gen/acc/", file_name + "_validation", history.
            ↪ history['val_acc'])
        save_file_to("data_gen/loss/", file_name, history.history['loss'])
        save_file_to("data_gen/loss/", file_name + "_validation", history.
            ↪ history['loss'])
        save_file_to("data_gen/mean_squared_error/", file_name, history.
            ↪ history['mean_squared_error'])
        save_file_to("data_gen/mean_squared_error/", file_name + "
            ↪ _validation", history.history['val_mean_squared_error'])

        prepare_plot('epoch', 'accuracy', history.history['acc'] ,history.
            ↪ history['val_acc'] , 'accuracy_' + file_name)
        prepare_plot('epoch', 'loss', history.history['loss'] ,history.
            ↪ history['val_loss'] , 'loss_' + file_name)

        model.save_weights("model/" + model_name + ".h5")

        if save_model_img:
                plot_model(model, to_file='model/' + model_name + '.png',
                    ↪ show_shapes = True)

        w = model.get_weights()

        print("Saved model to disk")

print("Loading model from disk")

# Load trained model from disk
model = create_model(batch_size, False)
model.load_weights("model/" + model_name + ".h5")

if save_model_img:
        plot_model(model, to_file='model/new_' + model_name + '.png',
            ↪ show_shapes = True)

check_directory("/data_gen/prediction")

# Predictions
###############################################
```

```
values = datos.values

values[:,1] = hour_encoder.transform(values[:,1]) # Encode HOUR as an
    ↪ integer value
values[:,2] = weekday_encoder.transform(values[:,2]) # Encode HOUR as int
values[:,3] = station_encoder.transform(values[:,3]) # Encode HOUR as int

print_array("DATOS A GUARDAR", values)


# Search for possible holes in the dataset
for i in range(0, int(n_in/288)):

       print("Analizando el día " + str(i))

       for j in range(0,288):

              e = i*288 + j

              print("> Hora " + str(j) + " | " + str(values[e][1]) + " /
                  ↪ len " + str(len(values)))

              # Comprobar si la hora actual y la siguiente están separadas
                  ↪  1 unidad
              if values[e][1] != 287:
                     if values[e][1] != (values[e+1][1] - 1):
                            print("-> No coinciden")

                            element_aux = values[e]
                            element_aux[1] = element_aux[1] + 1

                            values = numpy.insert(values, e, numpy.array(
                                ↪ element_aux), 0)

                            print("--> Insertar " + str(element_aux) + " |
                                ↪  Longitud actual " + str(len(values)))


print_array("RECREADOS DATOS QUE FALTAN", values)

new_dataset = DataFrame()
new_dataset['hour_sin'] = numpy.sin(2. * numpy.pi * values[:,1].astype('
    ↪ float') / (max_hour + 1))
new_dataset['hour_cos'] = numpy.cos(2. * numpy.pi * values[:,1].astype('
    ↪ float') / (max_hour + 1))
new_dataset['wday_sin'] = numpy.sin(2. * numpy.pi * values[:,2].astype('
    ↪ float') / (max_wday + 1))
new_dataset['wday_cos'] = numpy.cos(2. * numpy.pi * values[:,2].astype('
    ↪ float') / (max_wday + 1))
new_dataset['station'] = values[:,3]
new_dataset['free_bikes'] = values[:,4]
```

```
print_array("FINAL DATASET", new_dataset)


values = new_dataset.values

values = values.astype('float32') # Convert al values to floats

print_array("Prescaled values", values)

# scaler = MinMaxScaler(feature_range=(0,1)) # Normalize values
scaled = scaler.transform(values)

print_array("Dataset with normalized values", scaled)

scaled = scaled.reshape((1, n_in, len(columns))) # (...,n_in,4)

print_array("SCALED RESHAPED", scaled)

predicted = model.predict(scaled)[0]

predicted = [int(x * 35.0) for x in predicted]

print_smth("Valores predichos", predicted)

bicis_hoy = dia_hoy.values[:,4]

print_smth("Número de bicicletas hoy", dia_hoy)

plot_availability('time of the day', 'bikes', predicted, bicis_hoy , '
    ↪ real_prediction_' + file_name, "Predicción", "Valor real")
plot_availability('time of the day', 'bikes', average ,bicis_hoy , '
    ↪ real_vs_average_' + file_name, "Media", "Valor real")
```