Trabajo Fin de Máster
Máster Universitario en Ingeniería Industrial

Predicción del uso de bicis compartidas dependiendo de las condiciones climáticas del día

Autor: Andrea Beltrante
Tutor: Dr. Alejandro Escudero Santana

Dpto. Organización Industrial y Gestión de Empresas II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Máster
Máster Universitario en Ingeniería Industrial

Predicción del uso de bicis compartidas dependiendo de las condiciones climáticas del día

Autor:
Andrea Beltrante

Tutor:
Dr. Alejandro Escudero Santana
Profesor Titular de Universidad

Dpto. Organización Industrial y Gestión de Empresas II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo Fin de Máster: Predicción del uso de bicis compartidas dependiendo de las condiciones climáticas del día

Autor: Andrea Beltrante

Tutor: Dr. Alejandro Escudero Santana

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mi familia

A mis amigos, italianos y andaluces

Agradecimientos

Con este Trabajo Fin de Máster concluye una maravillosa experiencia de estudio. Quiero dar mi sincero agradecimiento a todos mis maestros y profesores, quienes con su profesionalidad contribuyeron a formarme como ingeniero y, sobre todo, como persona.

También quiero agradecer a todos los amigos con los que compartí una parte de esta aventura. No puedo sino empezar por Sara y Christian, con quienes tengo una amistad que desde hace tiempo es mayor de edad. Gracias a los compañeros del instituto, con los cuales atravesé infinitas alegrías y tristezas, y especialmente, a aquellos que siguen compartiendo sus experiencias conmigo: Celeste y Nadia, apoyos invaluables en los momentos más obscuros, y Marcello, incansable compañero de exploración. Gracias también a los colegas del Politecnico di Milano, que siempre me han animado a dar lo mejor de mí en todos los desafíos que hemos enfrentado juntos, y, especialmente, a Giusy y Federica, amigas sinceras y siempre atentas.

Agradezco a todos los amigos del Lido delle Nazioni, donde pasé los momentos más despreocupados de mi vida: gracias a Gabriele y Agnese, la pareja más amable que conozca; a Francesco y Marco, de cultura impresionante y excelentes compañeros de conversación, y a los dos Luca, por su perenne jovialidad y la increíble energía. Por supuesto, no me olvido de los compañeros de las "cenas de empresa", pasadas y futuras, por algunas de las mejores risas que he tenido en mi vida. Un enorme gracias, como enorme es nuestro grupo, a todos los amigos del Capo Hoorn, especialmente a Roberto, sin el cual ese grupo no existiría, y a Elisa, compañera de flamas y de ginebra. Gratitud inmensa a Gaia y Vittoria, que me demostraron su inquebrantable amistad cuando menos la merecía.

Estos años españoles han sido maravillosos; por lo tanto, quiero agradecer a los que me han acompañado durante toda o parte de esta experiencia, empezando por los compañeros Erasmus, para las fiestas estupendas. En la universidad encontré a los profesores de español mejores que pudiese desear: mis extraordinarios compañeros de estudio, personas inolvidables, tal y como las palabras que me han enseñado. Gracias a Guillermo "chiste", inagotable fuente de burlas y eminente historiador; a Fernando "pino", guardián de los secretos más recónditos de la ETSI y experto cocinero; a Álvaro "cubata", colega de proyectos, gran motivador y entrenador Pokémon. Agradezco a Manuel Alejandro "orilla" y a su familia, extremadamente generosos y siempre presentes para ayudarme, y a todo el grupo de los Fuerapista. Mil gracias a quien me presentó a todos los demás, guía en las noches sevillanas: Nico "resaca", de quien me acordaré cada vez que se me ocurra la palabra "amistad".

Por último, quiero dedicar un agradecimiento muy especial a mi familia: a mis abuelos, porque han hecho de cualquier vivencia que compartimos un tesoro inolvidable, desde los almuerzos de los domingos a los viajes al otro lado del mundo; a mi hermano, por ser mi mejor amigo y cómplice que hubiese podido desear; a mis padres, por su apoyo magnánimo e inagotable cada vez que les he necesitado, por la confianza que depositaron en mí cuando les pedí autonomía y por su amor incondicional. Siempre.

Resumen

En este Trabajo Fin de Máster se desarrollan algoritmos de aprendizaje automático ("machine learning") con el objetivo de hacer previsiones sobre el utilizo de un sistema de bicicletas compartidas basándose en las condiciones meteorológicas del día. Estos sistemas, que son una componente cada día más importante de la movilidad urbana, necesitan varios sistemas de soporte para que el servicio pueda ser proporcionado fiablemente a los ciudadanos: mejores previsiones permiten de incrementar su eficiencia.

A través de los algoritmos de aprendizaje automático se investiga la compleja relación que une los datos meteorológicos y el calendario con el número de bicis alquiladas, empleando herramientas de aprendizaje supervisado: los algoritmos aprenden a prever el parámetro deseado a través del análisis de un elevado número de ejemplos históricos, contenidos en la base de datos disponible: dicha base se compone de dos años de mediciones del número de bicis alquiladas en Londres y las condiciones climáticas que afectaban la ciudad durante ese periodo.

El desarrollo y la ejecución práctica de los algoritmos se realiza a través del lenguaje de programación Python y sus varias librerías, que proporcionan herramientas útiles tanto en las primeras fases de análisis de los datos como en la generación de previsiones.

Abstract

This Final Thesis presents the development of machine learning algorithms aimed at forecasting the use of a bike sharing service on the basis of the weather conditions. These services, which are more and more important in the urban mobility, require many supporting services to be reliably provided to the citizens: better forecasts allow to improve the overall efficiency.

The machine learning algorithms investigate the complex relations which exists between weather conditions, the calendar and the number of rented bikes, leveraging on the supervised learning: the algorithms learn to predict the desired parameter trough the analysis of a huge number of historic examples, contained in the available database: this database contains two years of records of the number of bikes rented in London and the weather conditions that characterized the city in this period of time.

The development and the execution of the algorithms are based on the Python language and its libraries, which provide useful tools both in the first phases of data analysis and in the building of previsions.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE GRÁFICAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

The best prophet of the future is the past.

- Lord Byron -

Los servicios de bicicletas compartidas son una realidad que se desarrolla cada día más, siendo una concreta y ecológica alternativa a la utilización del coche tanto en las grandes ciudades, donde la congestión del tráfico empuja hacia medios de transporte más agiles, como en los pueblos menos poblados, donde la calidad de los medios de transportes públicos no es suficiente a satisfacer las exigencias de movilidad de los ciudadanos. Los servicios de bicis compartidas contribuyen también a resolver el "problema de la última milla", o sea, a la dificultad que los ciudadanos que se sirven de los medios de transporte público encuentran en llegar desde una parada de estos al destino final, lo que muchas veces impone el empleo del coche para largas rutas que, si no fuese por dicha milla, podrían hacerse de forma conveniente con autobuses y trenes. Por último, un buen servicio de bicicletas compartidas es una alternativa valiosa para la movilidad de los turistas, lo que contribuye a descongestionar la red de transporte público en las áreas centrales de las ciudades, que son las que más sufren fenómenos de hacinamiento.

Ilustración 1 - Metro de Londres en la hora pico

Un sistema de bicicletas compartidas no se limita a la flota y a las estaciones, necesita: un importante soporte de servicios internos de oficina, un mantenimiento cíclico de las bicis y una continua redistribución de éstas entre las diversas áreas de la ciudad. La organización de estos servicios de apoyo tiene que basarse en el número de bicicletas circulantes y, por lo tanto, una previsión precisa de la demanda puede proporcionar una ayuda considerable para optimizar los gastos que el proveedor del servicio debe asumir.

Ilustración 2 - Sistema de redistribución de bicis

La utilización de las bicicletas, y la movilidad en general, es un fenómeno que sigue un patrón: la intensidad de viajes es mayor en los días laborales, concentrándose alrededor de las horas pico, mientras en los días festivos el uso es menor y más estable a lo largo del día. Entre los diferentes medios de transporte, la bicicleta es la que más está influenciada por el clima (Rudloff *et al.*, 2015), por lo que resulta interesante aprovechar los datos climáticos para obtener una previsión más precisa de la utilización de bicis compartidas.

Gracias a las nuevas herramientas informáticas que recaen en el campo del machine learning es posible incrementar la fiabilidad de las previsiones, obteniendo resultados interesantes con medios de cálculos normales. Estas previsiones, si están integradas en la programación de los servicios de soporte, permiten optimizar mucho la gestión de este importante servicio público.

1.1 Objeto y alcance del trabajo

El siguiente Trabajo Fin de Máster trata de estudiar, adaptar y aplicar algoritmos de Machine Learning a una base de datos real, constituida por dos años de observaciones sobre el número de bicicletas compartidas alquiladas en Londres y las condiciones atmosféricas de la ciudad. Estudiar e interpretar la correlación entre las condiciones climáticas y el número de bicicletas alquiladas permite estimar el uso de éstas basándose en previsiones meteorológicas, lo que proporciona un ayuda en la programación de todos los servicios de soporte, incluyendo la redistribución de las bicicletas y las oficinas de atención al cliente.

El alcance del trabajo incluye el análisis de los datos disponibles, con especial énfasis en los sesgos que podrían afectar negativamente al resultado de los algoritmos predictivos, y el desarrollo de herramientas para modificarlos y manejarlos. Entre los algoritmos de machine learning, se pondrá un mayor enfoque en los bosques aleatorios y en las redes neuronales.

1.2 Estructura del trabajo

En el capítulo 2 se introduce la temática enfrentada, los datos a disposición y el concepto de conjuntos de entrenamiento y de prueba. A partir desde el conjunto de prueba se observa la estructura de los datos, sus distribuciones y las varias tipologías de estos. Asimismo, se presentan las estrategias con las cuales se tratarán las diversas categorías de atributos.

En el capítulo 3 se presentan la metodología aplicada, las cuatro etapas principales del proyecto y el concepto de validación cruzada, que permite comprobar la eficacia de los algoritmos de previsión. La primera etapa del proyecto consiste en la preparación de los datos y la aplicación de los primeros modelos predictivos, y se reparte entre el capítulo 4 y 5.

En el capítulo 6 se describen las etapas de refino: tanto en la fase de preparación de los datos como en la implementación de los modelos de regresión se han tomado en consideración varias alternativas posibles,

comparando los resultados para establecer el mejor conjunto de parámetros a utilizar. La última etapa del proyecto consta del desarrollo de la red neuronal y se trata en el capítulo 7, donde se presentan las fases de desarrollo y el refino final de la red.

Los resultados obtenidos se muestran en el capítulo 8, donde se aplican los algoritmos de previsión a un nuevo conjunto de datos, mientras que las conclusiones se encuentran en el capítulo 9 junto a algunas propuestas que plantean nuevos horizontes de desarrollo.

2 DESCRIPCIÓN DEL PROBLEMA Y ANÁLISIS DE DATOS

El problema al que se enfrenta este trabajo es el descubrimiento de la relación entre el empleo de bicis compartidas con las condiciones climáticas del día, con el objetivo de desarrollar algoritmos que predigan la utilización futura basándose en las previsiones meteorológicas.

La herramienta que se utiliza es el Aprendizaje Supervisado, una rama del aprendizaje automático que enseña a un programa cómo inferir una salida dado un conjunto de entradas. El entrenamiento se hace suministrando ejemplos históricos, constituidos por parejas de entradas y salidas.

El programa aprende, entonces, gracias a la experiencia adquirida de la visión de los datos pasados, que son el punto de partida de todo el proceso.

2.1 Datos disponibles

La base de datos que se emplea ha sido obtenida a través de la plataforma "Kaggle" y contiene datos sobre dos años de préstamos de bicis en Londres, desde el 04/01/2015 hasta el 04/01/2017, con mediciones horarias de la cantidad de bicis empleadas y de las condiciones climáticas en dicho momento. La base se presenta como un fichero Excel CSV estándar: en la primera línea se encuentra el listado de todos los parámetros, mientras que, en las siguientes, el valor que cada uno de estos parámetros toma para cada hora de los dos años de observación, separados por comas.

	A	B	C	D	E	F	G
1	timestamp,cnt,t1,t2,hum,wind_speed,weather_code,is_holiday,is_weekend,season						
2	2015-01-04 00:00:00,182,3.0,2.0,93.0,6.0,3.0,0.0,1.0,3.0						
3	2015-01-04 01:00:00,138,3.0,2.5,93.0,5.0,1.0,0.0,1.0,3.0						
4	2015-01-04 02:00:00,134,2.5,2.5,96.5,0.0,1.0,0.0,1.0,3.0						
5	2015-01-04 03:00:00,72,2.0,2.0,100.0,0.0,1.0,0.0,1.0,3.0						
6	2015-01-04 04:00:00,47,2.0,0.0,93.0,6.5,1.0,0.0,1.0,3.0						
7	2015-01-04 05:00:00,46,2.0,2.0,93.0,4.0,1.0,0.0,1.0,3.0						
8	2015-01-04 06:00:00,51,1.0,-1.0,100.0,7.0,4.0,0.0,1.0,3.0						
9	2015-01-04 07:00:00,75,1.0,-1.0,100.0,7.0,4.0,0.0,1.0,3.0						

Ilustración 3 - Base de datos en Excel

Los parámetros a disposición son:

- timestamp: recoge fecha y hora de cada observación. Permite de identificar de manera unívoca cada dato, incluso los que se puedan seguir añadiendo a la base de datos en actualizaciones sucesivas.
- cnt: es el número de alquileres empezados en esa hora. Constituye el "label", es decir, el objetivo a estimar.
- t1 y t2: son la temperatura media de la hora y la temperatura media percibida en cada hora.
- hum y wind_speed: informaciones sobre la humedad media y la velocidad del viento en cada hora.
- weather_code: indica de forma codificada la situación meteorológica.
  - 1 = Despejado / Mayormente despejado
  - 2 = Nubes dispersas / Pocas nubes
  - 3 = Nublado
  - 4 = Muy nublado / Niebla
  - 7 = Lluvia / Lluvia ligera
  - 10 = Lluvia con tormenta
  - 26 = Nevada
  - 94 = Niebla helada
- is_holiday e is_weekend: toman el valor "1" si el día al cual pertenece la observación es festivo/ pertenece al fin de semana.
- season: es la estación.

El fichero se importa en el Python como instancia de la clase pandas.DataFrame:

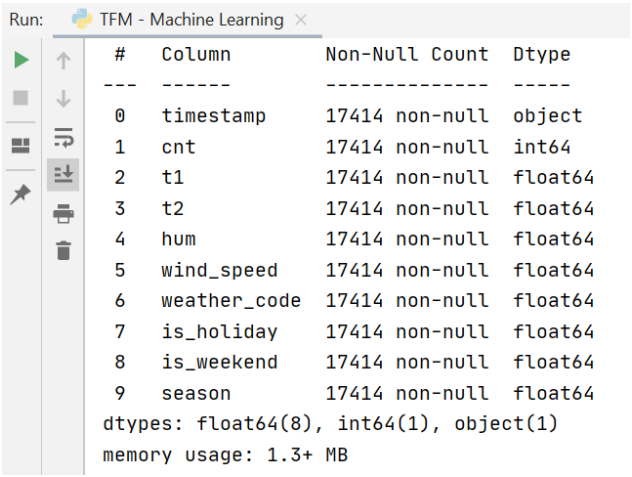


Ilustración 4 - Base de datos en Python

La base de datos está completa, es decir que para cada una de las horas observadas no hay ningún valor nulo, porque se han memorizado todos los atributos considerados. Todos los valores son numéricos, excepto el atributo "timestamp", que el Python interpreta como una variable de texto cualquiera: para extraer informaciones sobre el día y la hora a la cual corresponde cada observación es necesario transformarlo en un atributo de tipo "datetime64". Además, es necesario crear un índice que identifique unívocamente cada instancia, incluidas futuras ampliaciones a la base de datos con nuevas observaciones. Para ello, es conveniente basarse en el "timestamp", siendo en conjunto hora-día-mes-año siempre unívoco.

2.2 Conjunto de entrenamiento y conjunto de prueba

Antes que seguir con las transformaciones de los datos, es fundamental guardar un porcentaje de estos (una quinta parte aproximadamente) y no volver a tocarlos hasta que no se haya desarrollado todo el modelo: esto permite comprobar la eficacia del modelo en las previsiones, que es el objetivo, y no solo en el ajuste a los datos que ya se tienen. Se habla entonces de conjunto de entrenamiento ("training set") para el 80% de datos que se utilizan en las varias fases de construcción de los modelos, y de conjunto de prueba ("test set") para el restante 20%.

La división entre el conjunto de entrenamiento y el conjunto de prueba se logra con un identificativo ("index"), el cual permite aislar el conjunto de prueba durante las ejecuciones del código, evitando así que toda la base de datos sea utilizada por los algoritmos de machine learning.

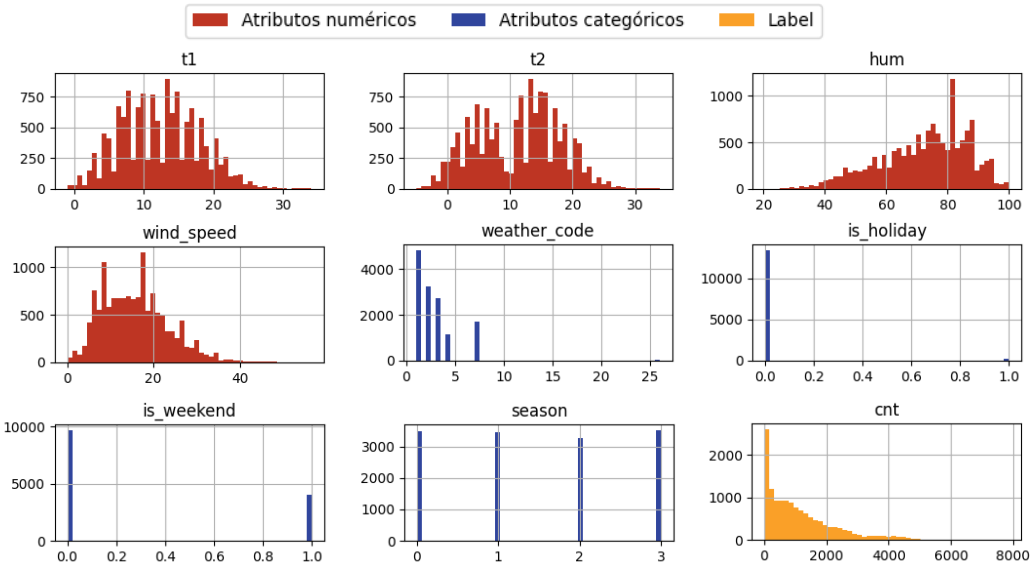
Siendo la utilización diaria de las bicicletas cíclica, se repartirán los datos en función del día (todas las observaciones del mismo día se pondrán bien en el test set, bien en el training set), empleando la función split_train_test_by_id():

```
def split_train_test_by_id(data, test_ratio, id_column, hash = hashlib.md5):
  ids = data[id_column].apply(lambda x : x.isocalendar()[0]*10000 + x.isocalendar()[1]*100 + x.isocalendar()[2])
  in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
  return data.loc[~in_test_set], data.loc[in_test_set]

def test_set_check(identifier, test_ratio, hash):
  return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio
```

La función split_train_test_by_id() recibe los datos, el porcentaje de datos a poner en el test_set ("test_ratio") y la columna identificativa (en este caso "timestamp") sobre la cual se basará la división. Para cada dato se genera un identificativo numérico de forma XXXXYYZZ, siendo XXXX = año, YY = mes y ZZ = día (ese número será igual para cada hora de un mismo día). Una vez establecido a través de test_set_check() si el dato debe pertenecer al test set, la función devuelve el vector de datos que no son parte de este conjunto (o sea el training set) y el mismo test set. El funcionamiento de test_set_check() es simple: para cada número se toma el ultimo byte de su hash, de valor comprendido entre 0 y 255 con igual probabilidad, y si éste es menor que 256*test_ratio se devuelve "1", en caso contrario "0".

2.3 Visualización de los datos

Una primera representación de los datos a través de histogramas muestra ya algunas características interesantes:



Gráfica 1 - Distribución atributos

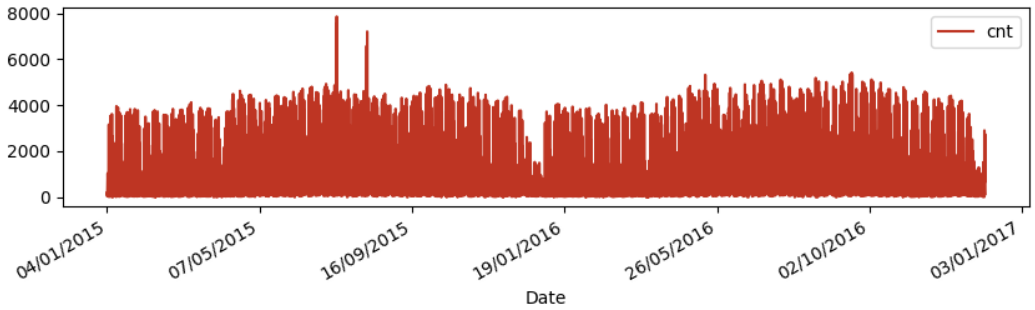
Se observa que los atributos numéricos poseen una distribución próxima a la normal, aunque los atributos "hum" y "wind_speed" parecen algo sesgados. La función skew() confirma esta impresión, midiendo un valor de

asimetría de -0.578 y de 0.661 respectivamente.

En cuanto a los atributos categóricos, se observa como en el "weather_code" la gran mayoría de las observaciones se concentra en cinco clases, mientras que la representación de "is_holiday" indica que la cantidad de días festivos es despreciable.

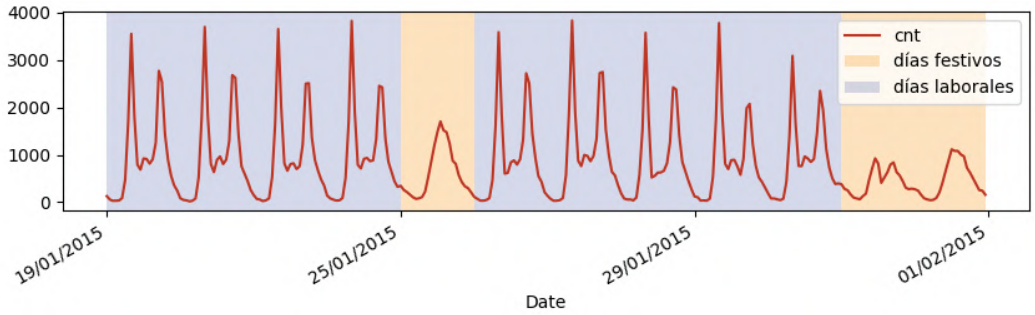
Finalmente, desde la representación de la etiqueta se observa que el "cnt" está fuertemente desequilibrado hacia valores bajos y por lo tanto muy lejos de tener una distribución normal.

Para investigar mejor la naturaleza de "cnt", se ha trazado su evolución a lo largo de los dos años y se ha observado que la media de utilización es más o menos constante, no hay estacionalidad ni ninguna tendencia. También, se advierten claramente algunos valores atípicos, tanto positivos como negativos.



Gráfica 2 - Evolución de "cnt"

Ampliando, se observa una ciclicidad diaria, con importante diferencia entre días laborales y fin de semana (el hecho de que el período no siga el típico patrón de siete días semanales se debe a que los datos analizados pertenecen al conjunto de entrenamiento y faltan, pues, aquellos días que han sido asignados al conjunto de prueba).



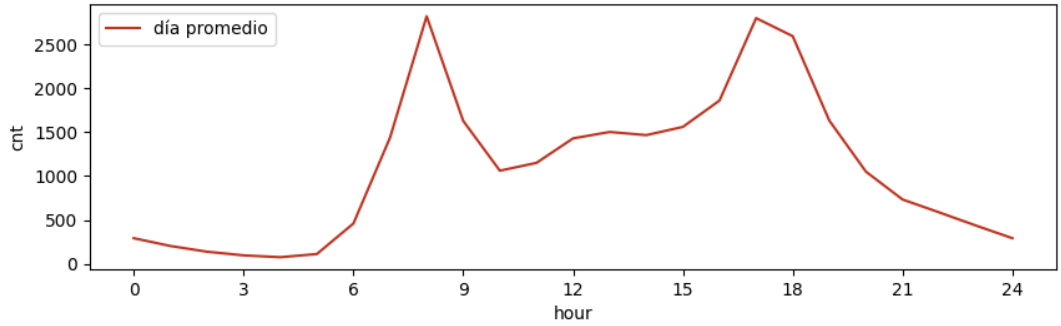
Gráfica 3 - Evolución de "cnt" ampliada, destacando la naturaleza de los días

2.4 Modificación de los atributos

En ocasiones resulta útil modificar los atributos de la base de datos para interpretar mejor algunas características de estos, pudiendo así aprovechar información que, aun estando ya presente, en su condición inicial es de difícil comprensión para los algoritmos.

El ejemplo más claro de este concepto está constituido por la ciclicidad a diario de los datos: que exista una cual cierta similitud entre la misma hora de días distintos es algo intuitivo y también visible en la gráfica anterior, pero esta información se pierde en la complejidad del atributo "timestamp". La creación de un nuevo atributo "hour" ("hora"), que será un número entero entre 0 y 23, permite representar fácilmente dicha similitud. Igualmente, se introducen los atributos "weekday" ("día de la semana") y "month" ("mes"), en los casos en los que existiesen patrones propios de un día específico de la semana o de un mes del año.

Aprovechando el nuevo atributo “hour” es posible dibujar la curva de uso diario promedio:



Gráfica 4 - Patrón de uso diario promedio

Como anteriormente se había notado una diferencia entre el patrón de utilización en los días laborales y los del fin de semana, se han dibujado ambas curvas, confirmando la intuición anterior. Se ha también añadido la curva de utilización en los días festivos, que se asemeja mucho a la de los días del fin de semana:

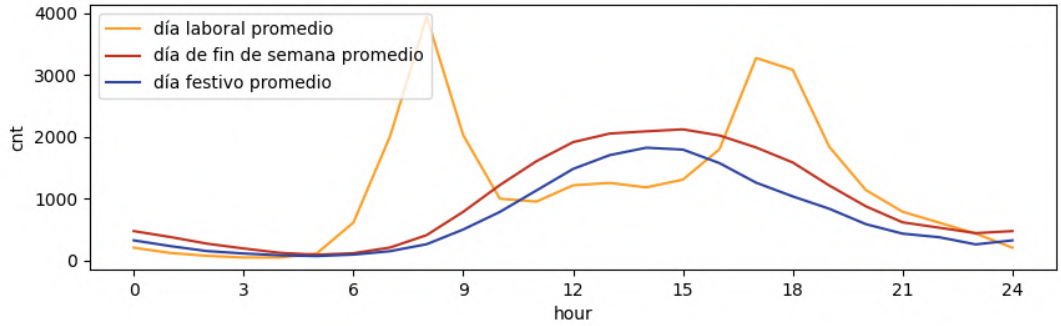


Ilustración 5 - Confronto de varios patrones de uso diario promedio

Algunas modificaciones de la base de datos no son inmediatas o conllevan una contraposición entre beneficios distintos, y en estos casos se ha realizado el análisis tanto con la modificación hecha como sin ésta. Por ejemplo, se observa que el patrón de los días festivos es muy parecido a lo de los días del fin de semana y que el número de días festivos es bastante reducido a lo largo del año, de manera que se ha considerado oportuno fusionar los dos atributos en uno único. Esto permite simplificar el modelo predictivo, aunque implica una pérdida de información.

En las curvas de uso diario promedio se ve cómo la utilización de bicicletas es significativamente mayor en las horas pico de los días laborales:

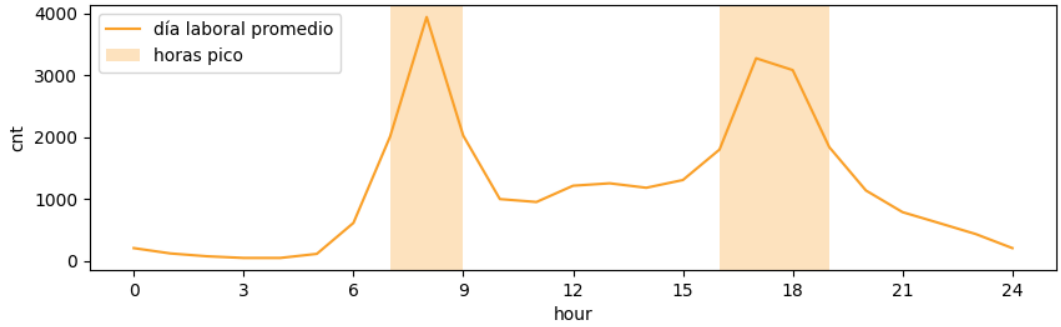


Ilustración 6 - Patrón de uso diario promedio en los días laborales, destacando las horas pico

Esta condición es dada por la combinación de tres atributos distintos, o sea “is_weekend”, “is_holiday” y “hour”, razón por la cual se ha considerado la creación del atributo binario “rush_hour” (“hora pico”) para mejorar la visibilidad de este estado de acuerdo con la relación lógica:

```
data_for_analysis["is_weekend"] == 0
data_for_analysis["is_holiday"] == 0
7 <= data_for_analysis["hour"] < 9
                V                          } data_for_analysis["rush_hour"] = 1
16 <= data_for_analysis["hour"] < 19
```

2.5 Búsqueda de Correlaciones

Es interesante investigar la correlación que existe entre los atributos y el número de bicicletas alquiladas aprovechando el método corr(), que crea una matriz de correlaciones entre todas las columnas de la base de datos; a partir de esta matriz se ha extraído la columna dedicada a la variable dependiente “cnt”.

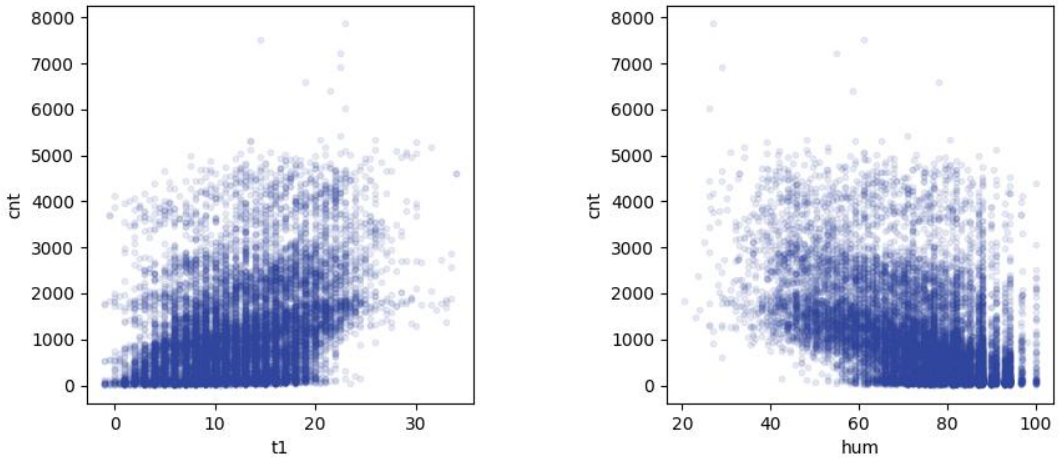
```
corr_matrix = data_for_analysis.corr()
print(corr_matrix["cnt"].sort_values(ascending = False))
```

```
Run:  TFM - Data analysis
cnt             1.000000
t1              0.391911
t2              0.371496
hour            0.325219
wind_speed      0.129930
month           0.062791
is_holiday     -0.057081
weekday        -0.066630
is_weekend     -0.096213
season         -0.118855
weather_code   -0.161517
hum            -0.472819
Name: cnt, dtype: float64
```

Ilustración 7 - Correlación entre “cnt” y el resto de los atributos

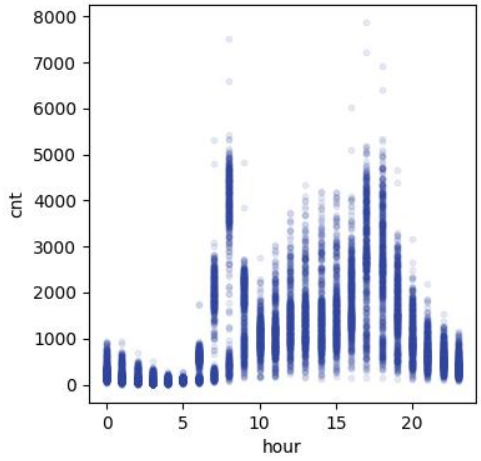
No hay ningún parámetro que tenga una correlación fuerte (>75%) con la variable dependiente “cnt”: la temperatura real y la percibida aparecen ligeramente relacionadas positivamente con el número de bicis alquiladas, así como el atributo “hour” que se ha introducido en la base de datos. Por el contrario, la humedad tiene una correlación negativa.

A través de la librería matplotlib se obtienen los gráficos de dispersión, que para los atributos ya presentes en la base de datos confirman visualmente la ligera correlación que se había observado a través de la matriz:



Gráfica 5 - Correlación de los atributos “t1” y “hum” con la etiqueta “cnt”

Sin embargo, no se puede decir lo mismo de la variable “hour”, cuyo gráfico de dispersión muestra claramente una relación más compleja que la que liga al “cnt” con las variables anteriormente mencionadas:



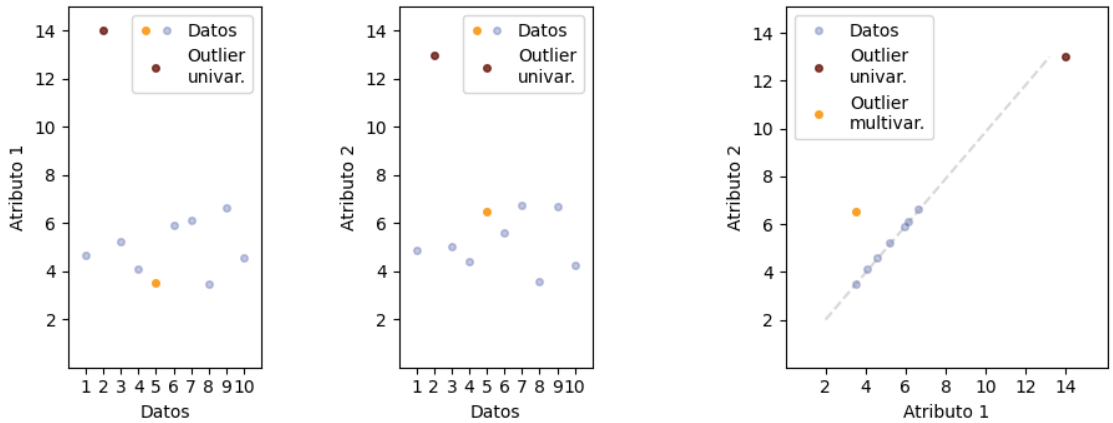
Gráfica 6 - Correlación del atributo “hour” con la etiqueta “cnt”

2.6 Valores atípicos

Los valores atípicos, o outliers, son variaciones extremas en los datos que difieren mucho del patrón general de estos. Pueden generarse por errores en la medición y procesamiento de los datos, pero también pueden ser una natural desviación en el muestreo. Es preciso identificarlos porque pueden influenciar mucho las previsiones futuras del modelo: en el caso de que el origen de estos valores se identifique con un fenómeno puntual, la solución puede ser el descarte del dato o su sustitución por un valor obtenido en función de los datos cercanos.

Un outlier puede ser univariado o multivariado, según si su atipicidad se observa en un solo atributo o en la relación entre varios atribuidos distintos. En la siguiente imagen se muestran los valores de dos atributos del mismo conjunto de datos: es evidente la presencia de un valor atípico en correspondencia del segundo dato, representado en burdeos, que se desvía mucho de la media de los otros datos en ambos atributos.

Sin embargo, representando la relación entre los dos atributos, se manifiesta un patrón de correlación muy fuerte (que parece confirmado también por el outlier univariado, aunque los valores estén fuera del rango), patrón respecto del cual el quinto dato, resaltado en naranja, se aleja mucho. Su atipicidad es clara, pero sólo en el momento en el que se consideran ambos atributos a la vez.



Gráfica 7 - Ejemplificación de outliers univariados y multivariados

En este trabajo se ha elegido fijarse sólo en el análisis de los outliers univariados, siendo la búsqueda de los multivariados mucho más compleja. La identificación de los valores atípicos se ha basado en el método de las vallas de Tukey, que considera outlier cualquier dato que caiga fuera del siguiente intervalo, donde Q_1 y Q_3 son los cuartiles inferior y superior:

$$[Q_1 - 1.5 * (Q_3 - Q_1), \qquad Q_3 + 1.5 * (Q_3 - Q_1)]$$

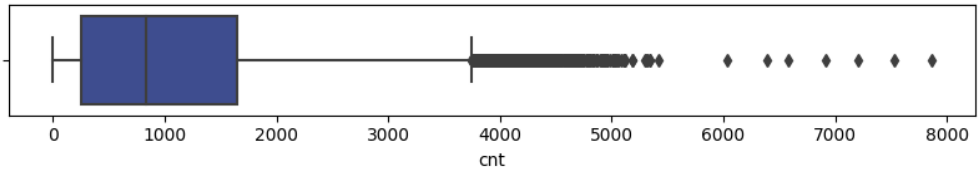
La identificación de los valores atípicos se hace a través de la función find_outlier():

```
def find_outlier(dataframe, attribute, group_by = None):
    df = dataframe.copy()
    if not group_by:
        q1 = df.quantile(q = 0.25)[attribute]
        q3 = df.quantile(q = 0.75)[attribute]
        iqr = q3-q1
        df["outlier"] = ~df[attribute].between(q1-1.5*iqr, q3+1.5*iqr)
        return df[df.outlier==1].index
    else:
        grouped_df = pd.DataFrame(index = np.unique(df[group_by]))
        grouped_df.index.name = group_by
        q1 = df.groupby([group_by]).quantile(q = 0.25)[attribute]
        q3 = df.groupby([group_by]).quantile(q = 0.75)[attribute]
        iqr = q3-q1
        grouped_df["fence_low"]  = q1-1.5*iqr
        grouped_df["fence_high"] = q3+1.5*iqr
        merged_df = pd.merge(df, grouped_df, on = group_by, how = "left").set_index(df.index)
        merged_df["outlier"] = ~merged_df[attribute].between(merged_df["fence_low"],
                                                             merged_df["fence_high"])
        return merged_df[merged_df.outlier==1].index
```

Esta función recibe la base de datos y el nombre de la columna a investigar, y devuelve los índices de los valores atípicos. El parámetro "group_by", opcional, permite encontrar los outliers en subgrupos más homogéneos que la entera base de datos, como en el caso de la etiqueta que se muestra a continuación.

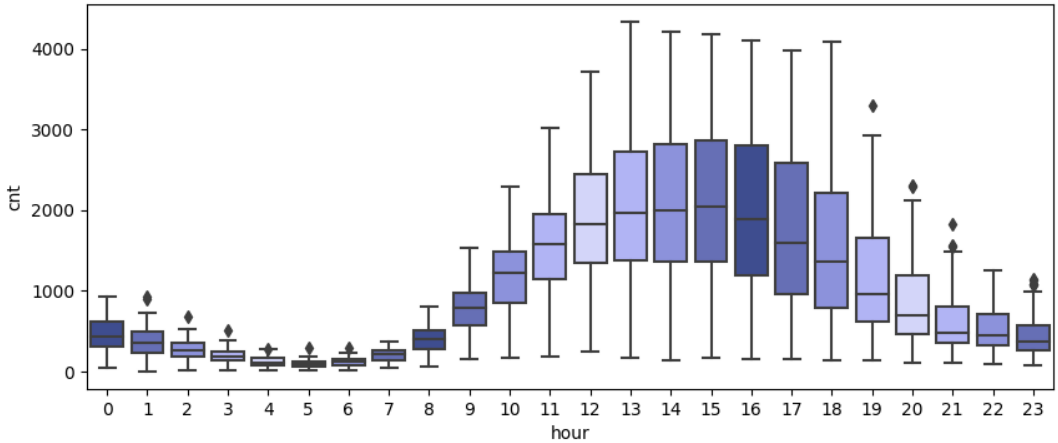
2.6.1 Etiqueta

Con respecto al número de bicis alquiladas, la identificación de los valores atípicos no puede ignorar la fuerte ciclicidad en la utilización de éstas, porque eso conllevaría un aplastamiento de los cuartiles hacia valores bajos, debido al escaso uso de las bicicletas en los horarios nocturnos. Se puede ver el problema en la siguiente gráfica, obtenida a través de la librería seaborn:

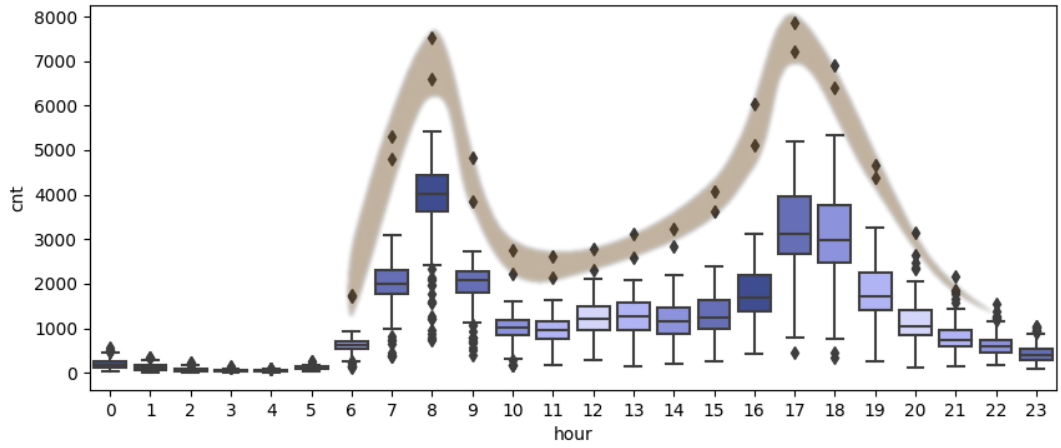


Gráfica 8 - Diagrama de caja de la etiqueta "cnt"

De esta forma, es mejor observar la dispersión de los datos según las horas del día y, como ya se ha observado una fuerte diferencia entre los patrones de uso durante fines de semana y durante días laborales, al considerar estos dos conjuntos por separado:

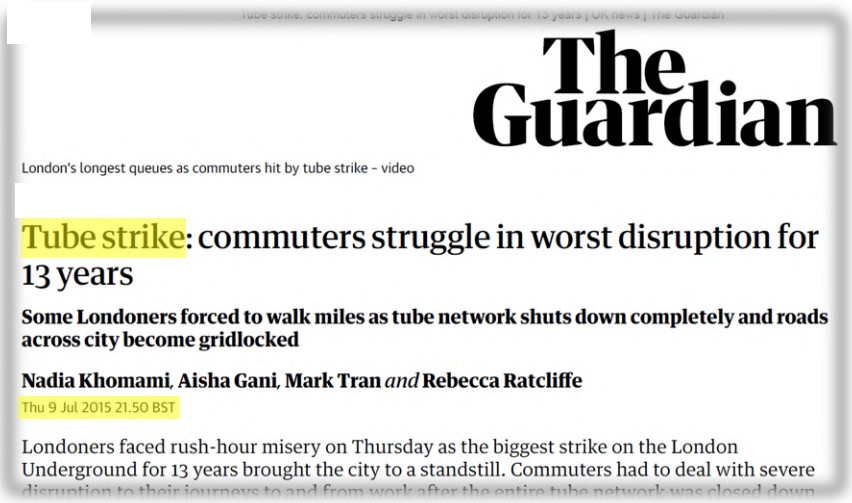


Gráfica 9 - Diagrama de caja de la etiqueta "cnt" por cada hora de los días festivos



Gráfica 10 - Diagrama de caja de la etiqueta "cnt" por cada hora de los días laborales

Mientras en los días festivos la cantidad de valores atípicos es muy limitada y estos no se alejan mucho del límite superior, en los días laborales se observa una presencia más importante tanto de outliers positivos como negativos, con un patrón interesante de parejas de valores muy por encima del promedio. Investigando más sobre estos valores se ha observado que se refieren a dos días específicos, o sea el 9 julio y el 6 agosto del 2015. En dichos días el metro de Londres sufrió las peores huelgas de los últimos años, por lo que parece razonable atribuir a ésta la causa de las desviaciones.



The Guardian

London's longest queues as commuters hit by tube strike – video

Tube strike: commuters struggle in worst disruption for 13 years

Some Londoners forced to walk miles as tube network shuts down completely and roads across city become gridlocked

Nadia Khomami, Aisha Gani, Mark Tran *and* Rebecca Ratcliffe

Thu 9 Jul 2015 21.50 BST

Londoners faced rush-hour misery on Thursday as the biggest strike on the London Underground for 13 years brought the city to a standstill. Commuters had to deal with severe

Ilustración 8 - Crónica de las huelgas de Londres en el periódico "The Guardian"

Dado que estos valores están muy localizados y tienen una explicación tan clara, se ha optado por suprimirlos, descartando del conjunto de entrenamiento los datos correspondientes a estos dos días, de modo que el modelo no resulte influenciado por este fenómeno puntual.

En cuanto a los otros valores atípicos, no es recomendable descartar tal cantidad de datos sin tener una explicación válida de su origen; se ha optado por mantenerlos en la base de datos.

2.7 Tipologías de atributos

Se pueden distinguir tres categorías de atributos, las cuales deben ser manejadas de forma diferente en los algoritmos de machine learning:

- Numéricos ("cnt", "t1", "t2", "hum", "wind_speed")
  Son todos aquellos que son representados por escalares y, por lo tanto, es posible establecer un orden entre valores distintos; dos números consecutivos representan dos condiciones parecidas. Siendo la base de datos una representación limitada de la realidad, pertenecen a esta categoría también magnitudes físicas vectoriales, como la velocidad del viento, ya que no se dispone de información sobre la dirección de éste. Los atributos numéricos deben ser escalados o normalizados para que todos tengan el mismo orden de magnitud, ya que la mayoría de los algoritmos de machine learning no trabaja bien con datos de tamaño muy distinto.

- Categóricos ("weather_code", "is_holiday", "is_weekend")
  Representan una condición unívoca entre un número finito de alternativas, cada una equidistante de las otras. También los atributos booleanos son una subcategoría de esta tipología de atributos, caracterizados por un solo par de alternativas ("es" y "no es"). Consecuencia importante de la equidistancia es que, aunque las varias alternativas puedan ser identificadas con un número, eso no conlleva una relación de semejanza entre números cercanos. Para evitar que los algoritmos de machine learning presupongan una similitud entre dos condiciones identificadas por números cercanos, los atributos categóricos de descomponen en matrices de atributos binarios, con un "1" que designa la categoría a la cual la instancia de la base de datos pertenece y un "0" en las demás.

	weather_code
2016-01-01	1
2016-01-02	2
2016-01-03	2
2016-01-04	4

→

	weather_code_1	weather_code_2	weather_code_3	weather_code_4
2016-01-01	1	0	0	0
2016-01-02	0	1	0	0
2016-01-03	0	1	0	0
2016-01-04	0	0	0	1

Tabla 1 - Ejemplo de gestión de un atributo categórico

- Cíclicos ("hour", "weekday", "month", "season")
  Son atributos numéricos con un límite superior e inferior definido, caracterizados por una condición de ciclicidad, es decir que una vez que se llega al valor más elevado se retorna al límite inferior, que tiene que considerarse semejante (por ejemplo, en el atributo "hour" el valor sucesivo al 23 es el 0, el cual difiere de su predecesor, el 23, tanto como éste del suyo, el 22). Los atributos cíclicos se tratan interpretando cada valor como una posición sobre una circunferencia imaginaria de perímetro igual a la diferencia entre el límite máximo y el mínimo. El valor cíclico queda, pues, traducido en un par de valores; en el seno y en el coseno de dicha posición.

	hour
2016-01-01	3
2016-01-02	6
2016-01-03	12
2016-01-04	23

→

	sen_hour	cos_hour
2016-01-01	0.707	0.707
2016-01-02	0	1
2016-01-03	-1	0
2016-01-04	0.966	-0.259

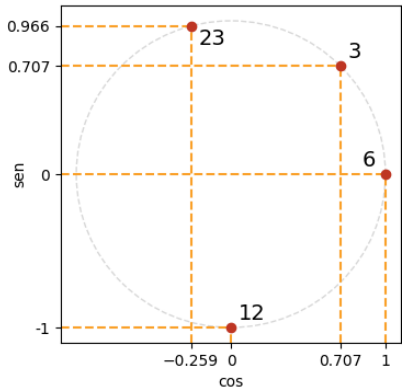


Tabla 2 - Ejemplo de gestión de un atributo cíclico

Es preciso evidenciar que algunos atributos ("weekday", "month" y "season"), que en principio se consideran cíclicos, podrían tratarse también como categóricos. En la etapa de refino de la fase de preparación se han confrontado los resultados de las varias estrategias de modelación para elegir la mejor categorización.

3 METODOLOGÍA

Todo el trabajo se basa en el lenguaje de programación Python y ha sido elaborado a través del entorno de desarrollo integrado (IDE) PyCharm. El Python, además que ser un lenguaje de programación abierto, presenta la gran ventaja de ser extremamente flexible, gracias a la gran cantidad de librerías disponibles que permiten importar sencillamente funciones y algoritmos listos para usar.

El trabajo se desarrolla según el principio de la mejora continua, aplicando una estrategia de prueba y error durante varios ciclos de aprendizaje. Se compone de las cuatro etapas resumidas en el siguiente esquema:

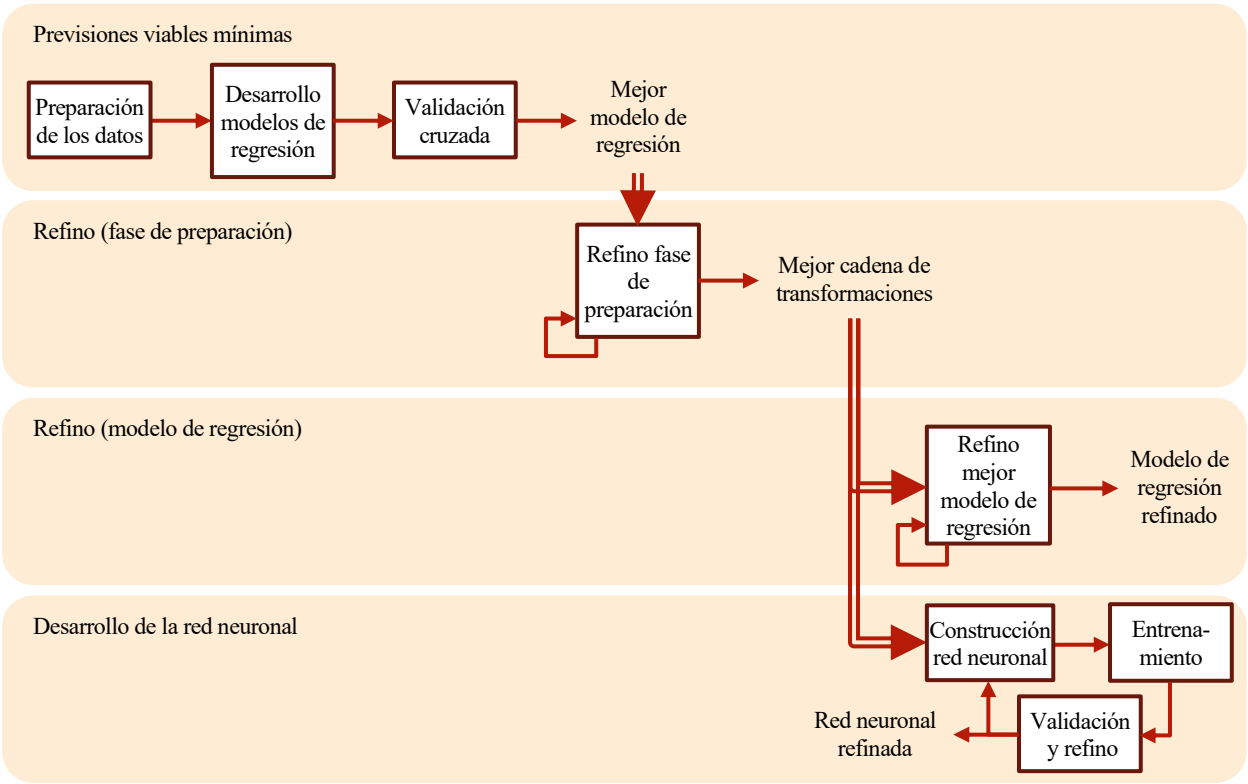


Ilustración 9 - Diagrama de flujo del trabajo

El hilo común de las etapas es el conjunto de entrenamiento, que tras una fase de preparación se emplea para ajustar los modelos de regresión y entrenar la red neuronal. El término "ajustar" hace referencia al hecho de que en esta fase se modifica la estructura de los modelos de regresión, por ejemplo, estableciendo el número de hojas

en el árbol de regresión, mientras la estructura de la red neuronal se establece *a priori*, limitándose a modificar el peso relativo de algunos parámetros en la fase de entrenamiento.

Cada etapa incluye una o varias fases de validación cruzada, que consisten en comprobar las capacidades predictivas del modelo sin emplear el conjunto de prueba. La validación cruzada se basa en partir el conjunto de entrenamiento en ***n*** subconjuntos de datos, de los cuales todos, salvo uno, se emplean para ajustar/entrenar el modelo en análisis, que posteriormente se emplea para hacer previsiones sobre el último paquete. El procedimiento se repite varias veces, dejando fuera de la etapa de entrenamiento un lote siempre distinto, y en cada iteración se mide la desviación entre las previsiones y las etiquetas del último paquete; la media de éstas es una buena aproximación del error que se espera del modelo frente a datos nuevos, proveyendo una medida de la eficacia de la mejora continua.

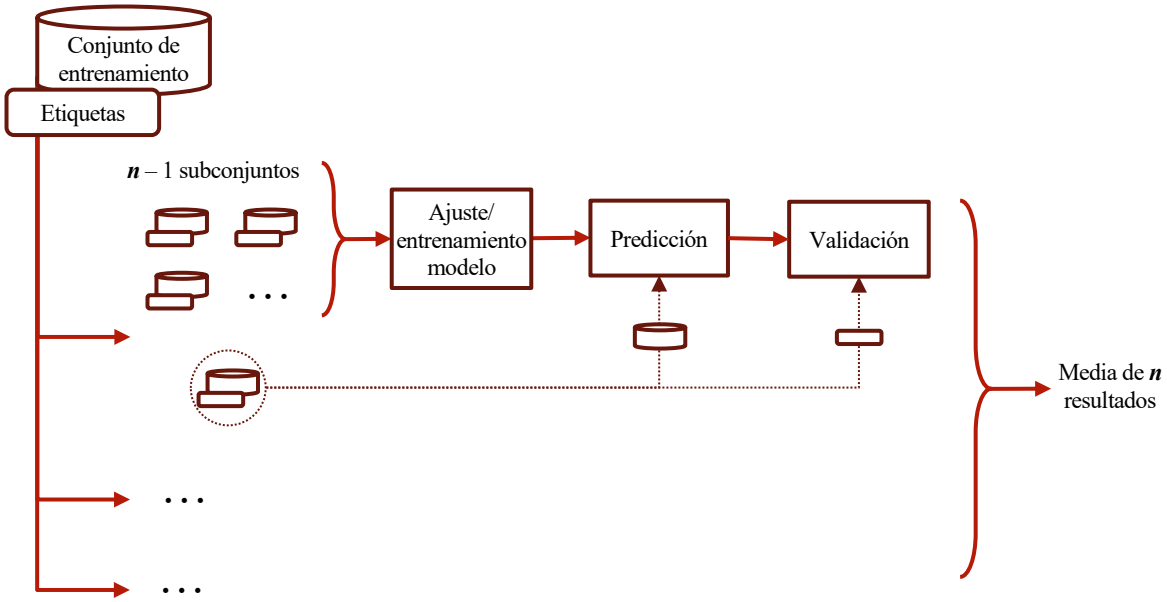


Ilustración 10 - Esquema del proceso de validación cruzada

En la primera etapa se preparan los datos, lo que es imprescindible para aplicar cualquier algoritmo de machine learning, y se aplican los primeros algoritmos predictivos, o sea los modelos de regresión, utilizando sus configuraciones por defecto. El resultado de los distintos algoritmos se compara a través de la validación cruzada. En el desarrollo de estos primeros pasos resulta de gran utilidad la librería Scikit-Learn, donde se encuentran muchas herramientas útiles en la transformación de la base de datos (como la normalización de los valores o el manejo de los atributos categóricos), varios modelos de regresión y la herramienta para ejecutar en automático la validación cruzada. Esta fase concluye con la elección del modelo de regresión más prometedor.

La segunda etapa es el refino de la fase de preparación, que se desarrolla a través de una repetida ejecución del mejor modelo de regresión, ajustándolo cada vez con atributos variadamente transformados y confrontando los resultados obtenidos con la validación cruzada: se ponen a prueba variaciones en la transformación, así como la añadidura o la eliminación de atributos adicionales. Se sigue en el ciclo de aprendizaje hasta el agotamiento de las opciones a experimentar, momento en el cual se selecciona la mejor cadena de transformaciones para emplearla en las dos etapas siguientes.

En la tercera etapa se refina el mejor algoritmo de regresión, a través de un nuevo ciclo de aprendizaje: se ejecutan varias validaciones cruzadas, probando conjuntos de parámetros distintos hasta identificar una solución satisfactoria.

Finalmente, en la última etapa, se desarrolla una red neuronal a través del enésimo ciclo de aprendizaje, constituida por las etapas de construcción, de entrenamiento y de validación y refino. En esta etapa se ha empleado la librería Tensorflow, que contiene todas las herramientas necesarias para la generación de una red neuronal.

4 PREPARACIÓN DE LOS DATOS

La fase de preparación de los datos se compone de varias transformaciones, algunas fijas y necesarias para la correcta ejecución de las etapas siguientes y otras que sólo entrarán en la preparación definitiva tras la obtención de buenos resultados experimentales. Las transformaciones pueden ser desarrolladas a través de modelación propia o empleando algoritmos procedentes de la librería Scikit-Learn, pero resulta imprescindible que todas estén programadas de forma similar para que la fase de preparación pueda ejecutarse de forma sistematizada y automática para toda la base de datos, incluida cualquier nueva instancia que se añada en futuro.

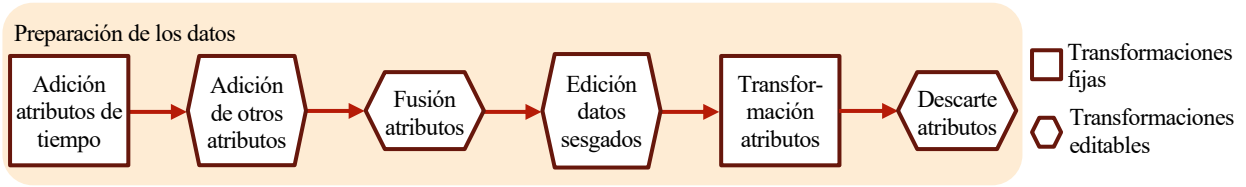


Ilustración 11 - Esquema del capítulo

Cada transformación está programada en Python como clase y varias clases componen un pipeline, donde la salida de una transformación constituye el ingreso de la siguiente. Con los hiperparámetros oportunos, se puede activar, desactivar o modificar todas las transformaciones que son inciertas. Para garantizar la coherencia entre los elementos se ha empleado como guía la estructura de las clases de la librería Scikit-Learn, ya que éstas se han empleado a lo largo de la fase.

La transformación de los datos con Scikit-Learn precisa de dos objetos distintos: los estimadores y los transformadores. Los estimadores son objetos que pueden estimar algunos parámetros valiéndose de una base de datos que se les proporciona mediante el método fit(). Un ejemplo de estimador es el StandardScaler(), que a través de StandardScaler.fit(base_de_datos), almacena promedio y desviación estándar de cada atributo para poder aplicar una normalización posteriormente.

Los transformadores son una subcategoría de estimadores que, partiendo de parámetros calculados a través del fit(), transforman una base de datos con el método transform(). Por ejemplo, la normalización de los datos se hace a través de StandardScaler.transform(base_de_datos). Si la base de datos desde la cual se sacan los parámetros y la base de datos a modificar coinciden, como en el caso del conjunto de entrenamiento, las dos etapas de fit() y transform() se pueden ejecutar simultáneamente a través del método fit_transform().

Para garantizar la correcta ejecución de los pipelines, se ha decidido construir todas las clases con la siguiente estructura, tal que en __init__() se inicializan los parámetros mientras en fit() y transform() se ejecutan los cambios (TransformerMixin permite aguantar también el fit_transform()):

```
class Nombre_Clase(TransformerMixin):
  def __init__(self, hiperparámetro1 = hp1, ... hiperparámetron = hpn):
    self.hiperparámetro1 = hp1
    ...
    self.hiperparámetron = hpn
  def fit(self, base_de_datos, y = None):
    ...
    return self
  def transform(self, base_de_datos, y = None):
    ...
    return base_de_datos_modificada
```

El pipeline completo se compone de seis elementos:

```
Full_Pipeline = Pipeline([('add_time_columns', Time_Columns_Adder()),
                          ('add_other_attributes', Rush_Hour_Adder(add_rush_hour = False)),
                          ('merge_attributes', Merge_Attributes(merge_attributes = False, columns =
                                                ["name1", ... "namen"], new_name = "new_name")),
                          ('modify_skewed_data', Skewed_Data_Modifier(columns_to_modify = ["name1", ...
                                                "namen"], tecnique = "tecnique")),
                          ('transform', All_Categories_Pipeline),
                          ('discard_attributes', Attributes_Discarder(columns_to_discard = ["column1", ...
                                                "columnn"])),
                         ])
```

4.1 'add_time_columns'

La primera etapa a través de la cual cualquier instancia trascurre es la clase Time_Columns_Adder(), que extrae desde el atributo "timestamp" información sobre la hora, el día de la semana y el mes del año, almacenándolas en las respectivas columnas.

```
Class Time_Columns_Adder(BaseEstimator, TransformerMixin):
  def __init__(self):
    self
  def fit(self, X, y = None):
    return self
  def transform(self, X, y = None):
    X.loc[:,"hour"] = X.loc[:,"timestamp"].array.hour
    X.loc[:,"weekday"] = X.loc[:,"timestamp"].array.weekday
    X.loc[:,"month"] = X.loc[:,"timestamp"].array.month
    return X
```

4.2 'add_other_attributes'

La clase Rush_Hour_Adder() desarrolla la misma función que Time_Columns_Adder(), pero dispone de un hiperparámetro booleano ("add_rush_hour") que, en la fase de refinación del modelo, permite excluir o emplear el nuevo atributo, según si ayuda en las previsiones o no, de forma automática.

```
class Rush_Hour_Adder(BaseEstimator, TransformerMixin):
  def __init__(self, add_rush_hour = False):
    self.add_rush_hour = add_rush_hour
  def fit(self, X, y = None):
    return self
  def transform(self, X, y = None):
    if self.add_rush_hour:
      rush_hour = np.where(((X.loc[:,"is_weekend"]==0) & (X.loc[:,"is_holiday"]==0)
                  & (((X.loc[:,"hour"]>=7) & (X.loc[:,"hour"]<9))
                  |((X.loc[:,"hour"]>=16) & (X.loc[:,"hour"]<19)))), 1, 0)
      X.loc[:,"rush_hour"] = rush_hour
    return X
```

4.3 'merge_attributes'

La clase Merge_Attributes() tiene el hiperparámetro booleano ("merge_attributes") para activar o desactivar la transformación, que recibe junto a los nombres de las columnas a fusionar y el nombre de la columna resultado. La fusión se hace según la conexión lógica de disyunción. Claramente, las dos columnas originales se eliminan desde la base de datos.

```
class Merge_Attributes(BaseEstimator, TransformerMixin):
  def __init__(self, merge_attributes = False, columns = None , new_name = None):
    self.merge_attributes = merge_attributes
    self.columns = columns
    self.new_name = new_name
  def fit(self, X, y = None):
    return self
  def transform(self, X, y = None):
    if self.merge_attributes:
      X.loc[:, self.new_name] = np.where((X[self.columns[0]]==1) | (X[self.columns[1]]==1), 1, 0)
      X.drop(self.columns[0], axis = 1, inplace = True)
      X.drop(self.columns[1], axis = 1, inplace = True)
    return X
```

4.4 'modify_skewed_data'

La clase Skewed_Data_Modifier() recibe como hiperparámetros el vector "columns_to_modify", continente de los nombres de las columnas a modificar (en caso de que no se reciba ningún vector, la transformación se queda desactivada), y el "tecnique", que establece qué técnica de normalización se debe emplear. Una vez que se hayan transformado todas las columnas proporcionadas, la clase devuelve la base de datos modificada.

```
class Skewed_Data_Modifier(BaseEstimator, TransformerMixin):
  def __init__(self, columns_to_modify = None, tecnique = "boxcox"):
    self.columns_to_modify = columns_to_modify
    self.tecnique = tecnique
  def fit(self, X, y = None):
    return self
  def transform(self, X, y = None):
    if self.columns_to_modify:
      for e in self.columns_to_modify:
        if self.tecnique=="boxcox":
          X.loc[:,e] = stats.boxcox(X.loc[:,e].replace(0, 0.1))[0]
        elif self.tecnique=="sqrt":
          X.loc[:,e] = np.sqrt(X.loc[:,e])
        elif self.tecnique=="log":
          X.loc[:,e] = np.log(X.loc[:,e].replace(0, 1))
    return X
```

4.5 'transform'

En esta etapa se transforman todas las columnas de la base de datos, tantos las originales como las añadidas en las etapas precedentes, aplicando las transformaciones presentadas en el capítulo 2, distintas según las tres diferentes tipologías de datos. Cada tipología se trata con un pipeline distinto, y se emplea la clase Dataframe_FeatureUnion() para combinar los tres resultados:

```
All_Categories_Pipeline = Dataframe_FeatureUnion(transformer_list = [("num_pipeline", num_pipeline),
                                                                     ("cat_pipeline", cat_pipeline),
                                                                     ("cyc_pipeline", cyclical_pipeline),
                                                                    ])
```

Cada uno de los pipelines se compone de dos etapas: la primera consiste en la selección de las columnas de la base de datos que el pipeline ha de transformar, siendo igual para los tres; la segunda difiere según las

características de estos.

La selección de las columnas se logra a través de la clase DataFrameSelector(), que recibe la base de datos oportunamente ampliada en las fases precedentes y, a través de un hiperparámetro, los nombres de la distintas columnas que deberá extraer desde ésta:

```
class DataFrameSelector(BaseEstimator, TransformerMixin
  def __init__(self, attribute_names = None):
    self.attribute_names = attribute_names
  def fit(self, X, y=None):
    return self
  def transform(self, X, y=None):
    if self.attribute_names:
      return X.loc[:, X.columns & self.attribute_names]
    else:
      return pd.DataFrame(index=X.index)
```

La clase devuelve un subconjunto de la base de datos que sólo contiene las columnas deseadas (en el caso de que no se proporcione el nombre de ninguna columna, se devuelve una base de datos vacía). Con este subconjunto se ingresa en la siguiente etapa que, como se ha dicho, consta de un pipeline para cada tipología de atributo.

4.5.1 'num_pipeline'

Los atributos numéricos deben ser debidamente escalados para que los algoritmos de machine learning actúen correctamente. Esto se logra a través de la clase Dataframe_StandardScaler(), una adaptación de la clase StandardScaler() disponible en la librería Scikit-Learn para hacerla compatible con bases de datos. Durante la ejecución del método transform() la clase sustituye en cada dato de las columnas proporcionadas el valor *Z*:

$$Z = \frac{X - \mu}{\sigma}$$

donde *X* es el valor originario, μ es la media de todos los valores y σ la deviación estándar.

El pipeline se presenta, entonces, como:

```
num_pipeline = Pipeline([('selector', DataFrameSelector(num_attribs)),
                         ('std_scaler', Dataframe_StandardScaler()),
                        ])
```

4.5.2 'cat_pipeline'

Los atributos categóricos deben ser separados en ***n*** variables binarias, donde ***n*** es el número de distintos valores que dichos atributos pueden tomar, a través de la clase Dataframe_OneHotEncoder(), derivada de la clase OneHotEncoder() de la librería Scikit-Learn. Además de crear las variables binarias, la clase Dataframe_OneHotEncoder() se encarga también de nombrar las nuevas columnas a través del método create_new_columns_name() que se activa durante la ejecución del método transform():

```
def create_new_columns_name(self, X):
  new_columns = []
  for i, column in enumerate(X.columns):
    j = 0
    while j < len(self.categories_[i]):
      new_columns.append(str(column) + '['+ str(self.categories_[i][j])+']')
      j += 1
  return new_columns
```

donde X es el subconjunto de la base de datos a trasformar y self.categories_ es una matriz continente del nombre de los distintos valores identificados durante la ejecución del método transform().

Así, el pipeline se presenta como:

```
cat_pipeline = Pipeline([('selector', DataFrameSelector(cat_attribs)),
                         ('one_hot_encoder', Dataframe_OneHotEncoder(sparse = False)),
                        ])
```

4.5.3 'cyc_pipeline'

Los atributos cíclicos se manejan a través de la clase DataFrame_TurnToCyclical(). Durante la ejecución del método transform(), cada atributo se convierte en una pareja de números comprendidos entre -1 y 1, que representan el seno y el coseno del valor equivalente si todas las instancias del atributo se dispusiesen alrededor de una circunferencia imaginaria, con la posición del valor máximo coincidente con la del valor mínimo:

```
def transform(self, X, y = None):
  if len(X.columns):
    elements_in_a_cycle = []
    for _ in range(len(X.columns)):
      elements_in_a_cycle.append((np.amax(X.iloc[:,_])-np.amin(X.iloc[:,_])+1))
    sin_elem = np.sin(2*np.pi*X/elements_in_a_cycle)
    sin_elem.columns = [self.create_new_columns_name(X, "sin")]
    self.columns = self.create_new_columns_name(X, "sin")
    cos_elem = np.cos(2*np.pi*X/elements_in_a_cycle)
    cos_elem.columns = [self.create_new_columns_name(X, "cos")]
    self.columns += self.create_new_columns_name(X, "cos")
    return pd.concat([sin_elem, cos_elem], axis = 1)
  else:
    self.columns = X.columns
    return X
```

donde X es el subconjunto de la base de datos a trasformar. La clase se encarga también de crear un nombre para cada una de las nuevas columnas, a través del método create_new_columns_name():

```
def create_new_columns_name(self, X, sc):
  new_columns = []
  for _ in range(len(X.columns)):
    new_columns.append(sc+"["+X.columns[_]+"]")
  return new_columns
def get_features_name(self):
  return self.columns
```

El pipeline se presenta, pues, como:

```
cyclical_pipeline = Pipeline([('selector', DataFrameSelector(cyclical_attribs)),
                              ('turn_to_cyclical', DataFrame_TurnToCyclical()),
                             ])
```

4.6 'discard_attributes'

En último lugar, la clase Attributes_Discarder() se encarga de borrar de la base de datos las columnas cuyo índice se corresponde con el del vector proporcionado. Esta clase se sitúa después de la etapa de transformación, porque es posible que se requiera suprimir una columna allí generada (por ejemplo, en el caso en que una de las columnas en las cuales se ha transformado un atributo categórico resulte poco significativa).

```
class Attributes_Discarder(BaseEstimator, TransformerMixin):
  def __init__(self, columns_to_discard = None):
    self.columns_to_discard = columns_to_discard
  def fit(self, X, y = None):
    return self
  def transform(self, X, y = None):
    if self.columns_to_discard:
      for atributo in self.columns_to_discard:
        X.drop(atributo, axis = 1, inplace = True)
    return X
```

5 MODELOS DE REGRESIÓN

Los primeros algoritmos de Machine Learning que se prueban son los modelos de regresión, siendo estos de sencilla programación y estando disponibles en la librería Scikit-Learn con una configuración de los parámetros por defecto, lista para usar. En el siguiente esquema se resumen los tres modelos de regresión tomados en consideración y las etapas a desarrollar en este capítulo:

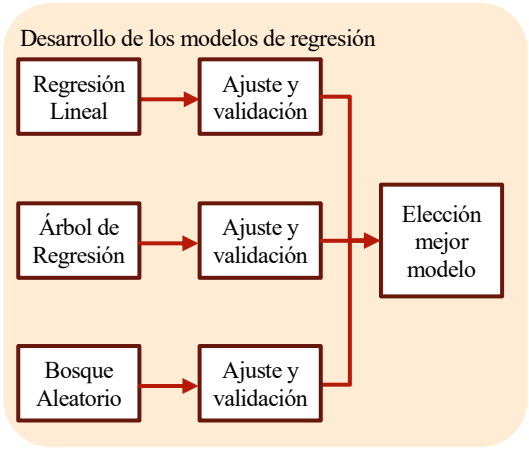
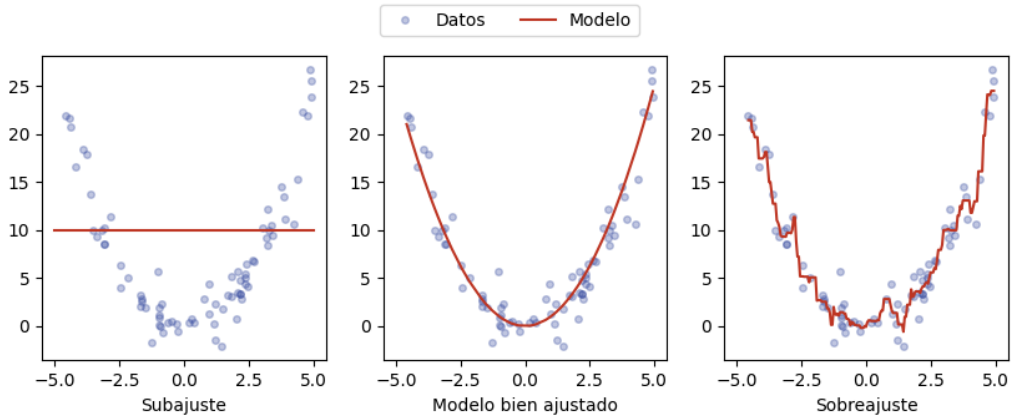


Ilustración 12 - Esquema del capítulo

Un modelo de regresión es un algoritmo que busca determinar la relación que une una variable dependiente, en este caso la etiqueta, con otras variables, en este caso los atributos, con el objetivo de poder prever el valor de dicha variable dependiente dado un conjunto de variables independientes. Para averiguar la relación, el algoritmo tiene que pasar por la fase de ajuste, empleando el conjunto de entrenamiento: en esta fase se modifica la estructura del modelo para que ésta encaje con las variables independientes de manera que se minimice la desviación entre el resultado proporcionado por el algoritmo y las etiquetas del conjunto de entrenamiento.

Un modelo incapaz de lograr un buen resultado es un modelo que sufre de subajuste ("underfitting"), es decir, que no consigue explicar bien la relación entre los datos del conjunto de entrenamiento y sus etiquetas; obviamente, como consecuencia no será capaz de hacer buenas predicciones. Sin embargo, explicar perfectamente los datos del training set no es suficiente e incluso puede ser dañino: el modelo no debe ser despistado por el ruido que caracteriza todas las bases de datos para no caer en el sobreajuste ("overfitting"), o sea, la ausencia de capacidades predictivas debido a una excesiva adherencia del modelo a las fluctuaciones en los datos del training set.

De esta forma, se trata de encontrar un equilibrio entre los extremos; de encontrar un modelo bien ajustado.



Gráfica 11 - Confronto entre modelos con distinto nivel de ajuste a los datos

Mientras que verificar si el modelo explica bien el conjunto de entrenamiento es sencillo, siendo suficiente emplearlo para predecir las etiquetas del mismo, comprobar si sufre de sobreajuste es más complejo y requiere aplicar la validación cruzada a través de la clase cross_val_score():

```
validation_scores = cross_val_score(estimator = modelo,
                                    X = data_prepared,
                                    y = data_labels,
                                    scoring = "neg_mean_squared_error",
                                    cv = 10)
```

La clase recibe el modelo a experimentar, unos datos para aplicarlo y el valor efectivo de las correspondientes etiquetas, juntos a la función de perdida a emplear en la evaluación de las previsiones y el número de rondas de evaluación a ejecutar. Se devuelve en fin un vector conteniente el resultado de cada ronda de evaluación, cuya media es una buena aproximación del error de previsión que se espera desde el modelo en el prever datos nuevos.

Gracias a la anterior etapa de preparación de los datos, resulta fácil comparar varios algoritmos distintos para encontrar el más prometedor, que pasará a la siguiente fase de refino.

5.1 Regresión Lineal

En primer lugar, se ha intentado aplicar a los datos el modelo matemático más simple, o sea la regresión lineal, fundamentalmente para comprobar la eficacia de las etapas anteriores y explicar claramente los pasos que constituyen la fase de ajuste y validación.

Para esta etapa se emplean las clases "predictores" de la librería Scikit-Learn, que tienen como métodos principales el fit() y el predict().

El método fit() se aplica a la instancia de la clase para que aprenda la estructura de los datos. Dada una base de datos oportunamente tratados a través de los Pipelines ("data_prepared") y los respectivos valores de la variable independiente ("data_label"), el fit() busca la estructura del modelo que minimiza el error cuadrático medio con respecto a la etiqueta. Una vez el modelo se ha ajustado, se pueden hacer previsiones con el método predict().

Para aplicar la regresión lineal se ha empleado la clase LinearRegression():

```
lin_reg = LinearRegression()
lin_reg.fit(data_prepared, data_labels)

data_predictions = lin_reg.predict(data_prepared)
lin_mse = mean_squared_error(data_labels, data_predictions)
lin_rmse = np.sqrt(lin_mse)

lin_scores = cross_val_score(lin_reg, data_prepared, data_labels, scoring ="neg_mean_squared_error", cv = 10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse, lin_rmse_scores)
```

Obteniendo como resultado:

Prediction Error on Training Set: 561.98
Typical Prediction Error: 568.27
Standard deviation: 61.21

Se observa que la regresión lineal es inadecuada para explicar la relación entre los datos: un error en la previsión del conjunto de entrenamiento de más de 560 bicicletas indica que el modelo está sufriendo un gran subajuste, considerando que el valor de "cnt" varía entre 0 y 8000. Consecuentemente, el modelo también proporciona pésimos resultados con los conjuntos de validación.

5.2 Árbol de Regresión

El siguiente algoritmo de machine learning con el que se experimenta es el TreeRegressor(), el cual crea un árbol de decisiones binarias que permite asociar una previsión a cada combinación de atributos. Empezando desde el nodo cabecera, se considera el parámetro que permite repartir los datos de la manera más distinta posible (o sea, la que más reduce el error cuadrático medio de los dos grupos obtenidos) y se sigue así hasta llegar a los nodos hojas, cuyo valor es la media de los elementos en ellos contenidos. En la siguiente ilustración está representado un árbol obtenido a través de TreeRegressor() con el número de niveles limitado a tres.

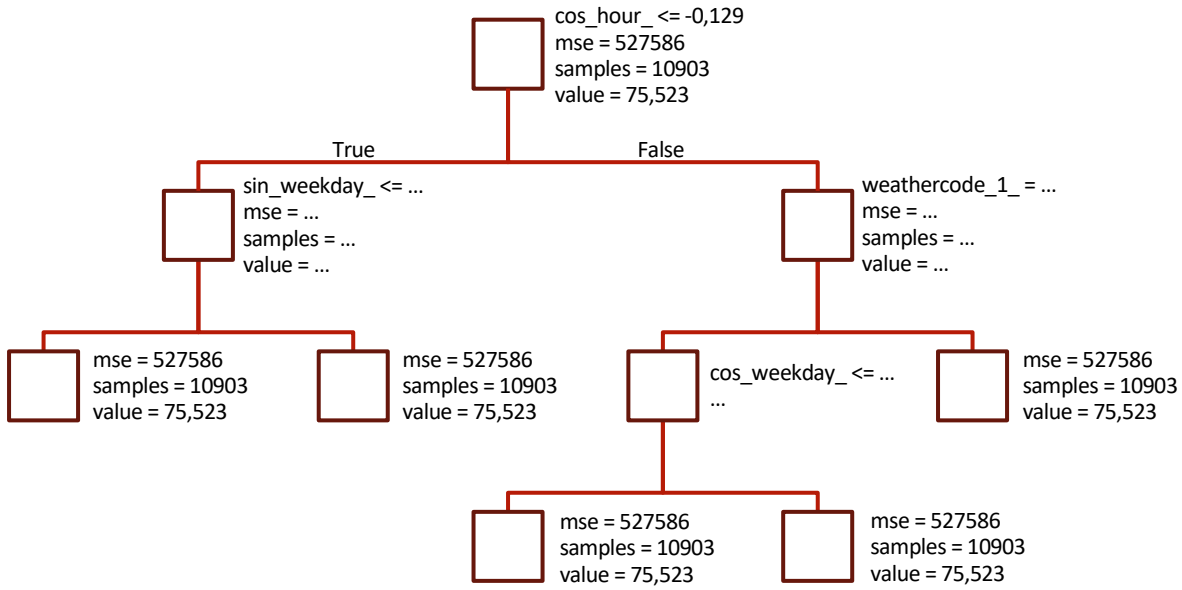


Ilustración 13 - Ejemplo de árbol de regresión

Aplicando el modelo sin limitaciones en el número de niveles se obtiene:

Prediction Error on Training Set: 3.34
Typical Prediction Error: 373.92
Standard deviation: 76.64

Al comparar con el output del LinearRegression(), se observa una notable mejora en la capacidad de explicación del training set. Sin embargo, esta mejora es muy reducida cuando se comprueba la capacidad de previsión empleando los conjuntos de validación: un árbol de decisión sin limitaciones en los hiperparámetros permite alcanzar una precisión próxima al 100% en la explicación de los datos del training set, incurriendo en el sobreajuste: el modelo es demasiado flexible para poder prever bien nuevos datos, porque ha memorizado no sólo las relaciones entre las distintas variables del conjunto de entrenamiento, sino también el ruido de fondo que éste posee.

Para reducir la variancia del modelo, o sea, su dependencia desde el training set con el cual se alimenta, es posible limitar la adaptabilidad del árbol bien acotando la profundidad, bien incrementando el número mínimo de elementos que una hoja ha de tener para ser, a su vez, separada. Cada limitación de estos parámetros hace el modelo más rígido, con el riesgo de no hallar relaciones y características fundamentales para entender las

relaciones entre los datos.

La mejor solución al compromiso entre flexibilidad y reducción de la variancia es crear varios árboles de decisión distintos empleando un bosque aleatorio.

5.3 Bosque Aleatorio

Un bosque aleatorio es un conjunto de árboles que se crean y se emplean de forma combinada para hacer previsiones. El término "aleatorio" hace referencia a que cada árbol se crea a partir de un subconjunto de datos elegidos al azar y a que la división de estos en hojas parte de un subconjunto casual de atributos. La previsión final del modelo es, entonces, la media entre las previsiones de los diversos árboles del bosque.

La idea tras el bosque es que, evaluando conjuntamente un grupo de previsiones hechas considerando parámetros y datos distintos, se obtiene un mejor resultado que empleando un solo método de previsión por preciso que sea.

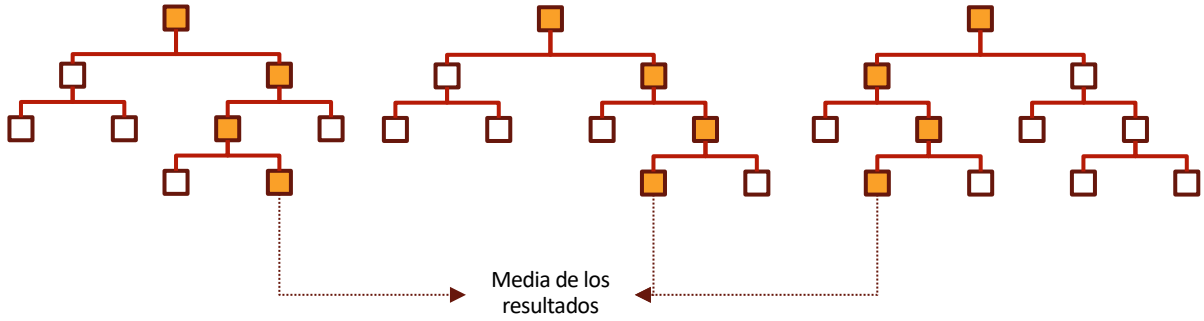


Ilustración 14 - Ejemplo de bosque aleatorio

Cada árbol creado es muy flexible, por lo que explica bien el subconjunto de datos sobre los cuales se construye, pero puede ser influenciado por el ruido de estos. Sin embargo, este segundo efecto, indeseable, está mitigado por la numerosidad del muestreo, que permite compensar la variancia excesiva de los árboles.

El bosque aleatorio se aplica a través de la clase `RandomForestRegressor()` y se obtiene:

```
Prediction Error on Training Set: 85,63
Typical Prediction Error: 282,45
Standard deviation: 67,75
```

Se observa que la capacidad de explicación del training set es peor que la del árbol de regresión, pero la capacidad de previsión ha crecido mucho: la aleatoriedad del bosque reduce la variabilidad, resultando menos influenciada por el ruido de los datos.

6 REFINO DE LOS ALGORITMOS

Una vez se ha encontrado un algoritmo que proporcione un buen resultado al problema, se puede pasar a la fase de refino. Esta fase consiste en la experimentación de varias alternativas, tanto en la etapa de preparación de los datos como en la de ejecución de los algoritmos de Machine Learning, con el fin de encontrar el conjunto de parámetros que mejor se adapte a la situación y perfeccione, así, la capacidad predictiva del modelo.

Las dos clases que se emplean en esta fase son la `GridSearchCV()` y la `RandomizedSearchCV()`:

```
GridSearchCV(estimator = clase o pipeline que aguante el método fit(),
             param_distributions = cuadrícula de hiperparámetros a experimentar,
             scoring = función de evaluación de las previsiones,
             cv = estrategia de división para la validación)
```

```
RandomizedSearchCV(estimator = clase o pipeline que aguante el método fit(),
                   param_distributions = cuadrícula de hiperparámetros a experimentar,
                   n_iter =  número de combinaciones que se muestrean,
                   scoring = función de evaluación de las previsiones,
                   cv = estrategia de división para la validación)
```

Ambas clases reciben un estimador y una lista de hiperparámetros a experimentar. Una vez que se aplica a las clases el método fit(), éstas ejecutan una validación cruzada del estimador aplicando todas las posibles combinaciones de los hiperparámetros (en el caso de la `GridSearchCV()`) o un subconjunto de combinaciones elegido al azar (en el caso de la `RandomizedSearchCV()`). El resultado de las pruebas se recoge en el atributo cv_results_, mediante el cual es posible comparar las diferentes combinaciones, identificando la que ha proporcionado el mejor resultado.

6.1 Refino de la fase de Preparación

En la fase de preparación de los datos es necesario tomar algunas decisiones sobre el tratamiento de los atributos que van a afectar al funcionamiento de los algoritmos de machine learning: no pudiéndose determinar con antelación cuál será la mejor estrategia de transformación, es necesario probar varias opciones para encontrar la que proporcione el mejor resultado. Para experimentar eficazmente a través de las dos clases anteriormente presentadas, es necesario que cada alternativa haya sido modelada con los hiperparámetros oportunos de modo que pueda ser aplicada de forma automática.

El punto de partida es crear un pipeline que contenga tanto la etapa de preparación de los datos como un modelo de previsión. Se ha elegido un bosque aleatorio, siendo el algoritmo que de momento ha proporcionado el mejor

resultado:

```
Preparation_and_Prevision_Pipeline = Pipeline([('data_preparation', Full_Pipeline),
                                               ('prevision', forest_reg)
                                              ])
```

Se crea también una la parrilla de hiperparámetros a experimentar:

```
param_grid = [{'localización hiperparámetro__hiperparámetro₁ a modificar': [Opción₁, ... Opciónₙ],
               ...
               'localización hiperparámetro__hiperparámetroₙ a modificar': [Opción₁, ... Opciónₙ]
              }]
```

6.1.1 Categorización atributos

La primera alternativa que se ensaya en la fase de preparación de los datos es la categorización de los atributos "season", "weekday" y "month", que podrían tratarse como atributos categóricos a semejanza de "weather_code" o como atributos cíclicos a la par de "hour".

El hiperparámetro a modificar es el vector attribute_names, que se proporciona al DataFrameSelector() para establecer qué atributos se manejan como categóricos y cuáles como cíclicos.

```
param_grid = [{ '...__cat_pipeline__...__attribute_names': [['weather_code', 'season', 'weekday', 'month']],
                '...__cyc_pipeline__...__attribute_names': [['hour']]
              },
               ...
              { '...__cat_pipeline__...__attribute_names': [['weather_code']],
                '...__cyc_pipeline__...__attribute_names': [['hour', 'season', 'weekday', 'month']]
               }]
```

Siendo en total ocho combinaciones distintas, es posible experimentarlas todas a través del GridSearchCV():

```
grid_search = GridSearchCV(estimator = Preparation_and_Prevision_Pipeline,
                           param_distributions = param_grid,
                           cv = 10,
                           scoring = 'neg_mean_squared_error')

grid_search.fit(data_attributes, data_labels)
```

Para sacar conclusiones más fiables se ha ejecutado la validación cruzada tres veces; los resultados más importantes se resumen en la siguiente tabla:

		Raíz del error cuadrático medio		
		Primera iteración	Segunda iteración	Tercera iteración
Hiperparámetros	'...__cat_pipeline__...__attribute_names' : ["weather_code", "season", "weekday", "month"] '...__cyc_pipeline__...__attribute_names' : ["hour"]	271,85	271,11	270,63
	'...__cat_pipeline__...__attribute_names' : ["weather_code", "season", "month"] '...__cyc_pipeline__...__attribute_names' : ["hour", "weekday"]	271,86	272,59	271,62
	...	...		
	'...__cat_pipeline__...__attribute_names' : ["weather_code"] '...__cyc_pipeline__...__attribute_names' : ["hour", "season", "weekday", "month"]	282	281,73	282,53

Tabla 3 - Resultados primera aplicación del GridSearchCV()

Se observa que el mejor resultado medio se obtiene cuando los atributos investigados se tratan como categóricos.

6.1.2 Preparación atributos

En la fase de análisis de datos se han hipotetizado algunas transformaciones adicionales que podrían contribuir a un mejor resultado. Las transformaciones adicionales que se experimentan son tres, o sea la incorporación en la base de datos del atributo "rush_hour", la combinación de los atributos "is_holiday" e "is_weekend" en el nuevo atributo "not_workday" y la normalización de los atributos sesgados "hum" y "wind_speed".

```
param_grid = [{ '...__add_rush_hour': [True, False],
                '...__merge_attributes': [True, False],
                '...__columns_to_modify': [None, ["hum"], ["wind_speed"], ["hum", "wind_speed"]]
              }]
```

Siendo en total dieciséis combinaciones distintas, es posible experimentarlas todas a través de GridSearchCV(); para una mayor fiabilidad de las conclusiones, se ha ejecutado la validación cruzada tres veces, obteniéndose los resultados que se resumen en la siguiente tabla:

Hiperparámetros	Raíz del error cuadrático medio		
	Primera iteración	Segunda iteración	Tercera iteración
'...__add_rush_hour' : False '...__merge_attributes' : True '...__columns_to_modify' : ["wind_speed"]	267,98	268,02	267,72
'...__add_rush_hour' : False '...__merge_attributes' : True '...__columns_to_modify' : ["hum"]	268,07	267,98	268,57
'...__add_rush_hour' : False '...__merge_attributes' : True '...__columns_to_modify' : None	268,35	268,5	268,45
...	...		
'...__add_rush_hour' : False '...__merge_attributes' : False '...__columns_to_modify' : None	271,05	272,3	271,81
'...__add_rush_hour' : False '...__merge_attributes' : False '...__columns_to_modify' : ["wind_speed"]	271,79	271,84	273,23

Tabla 4 - Resultados segunda aplicación del GridSearchCV()

Los resultados indican que las mejores previsiones se obtienen con el nuevo atributo "not_workday", empero desaconsejan la introducción del nuevo atributo "rush_hour". No existiendo un claro indicio de mejora de las previsiones gracias a la normalización de "hum" y "wind_speed", se ha elegido rechazar también esta transformación adicional.

6.2 Refino del Bosque Aleatorio

En las fases anteriores, los algoritmos de previsión han sido tratados como si fuesen cajas negras: después de haberlos alimentados con el training set, se han comprobado sus efectividades a través de cross_val_score() e, idealmente, se habrían podido verificar con el conjunto de prueba, así como aplicado para hacer previsiones. En realidad, los algoritmos de machine learning no son una herramienta inmutable, sino pueden ser adaptados a través de una serie de hiperparámetros.

El RandomForestRegressor() tiene seis hiperparámetros principales que establecen el tamaño del bosque que se va a crear y las características principales de los árboles que la pueblan. No teniendo ninguna información sobre los mejores valores de los hiperparámetros, se ha creado el siguiente conjunto de alternativas razonables, empezando por los valores que Python asigna por defecto (en negrita):

- n_estimators = [80, **100**, 120, 140, 160, 180, 200, 500, 1000]
  Define el número de árboles en la selva.
- min_samples_split = [**2**, 4, 6, 8, 10]
  Define el mínimo número de datos que un nodo debe tener antes que ser dividido.
- min_samples_leaf = [**1**, 2, 3, 4]
  Es el mínimo número de datos que cada hoja debe tener.
- max_features = [**'auto'**, 'log2']
  Es el número máximo de atributos que se considera para dividir un nodo en hojas, calculado en función del número de atributos de la base de datos. Con 'auto' se emplean todos los atributos.
- max_depth = [**None**, 100, 200, 500, 1000]
  Es el número máximo de niveles que un árbol del bosque puede tener.
- bootstrap = [**True**, False]
  Es el parámetro que establece si los datos sobre los cuales se basa la creación de los árboles se cogen con reemplazo, o sea, que el mismo dato pueda ser cogido más de una vez o no.

Los hiperparámetros se recogen en la cuadricula param_grid:

```
param_grid = [{ 'n_estimators': n_estimators,
                'min_samples_split': min_samples_split,
                'min_samples_leaf': min_samples_leaf
                'max_features': max_features,
                'max_depth': max_depth,
                'bootstrap': bootstrap,
               }]
```

Claramente, la limitada capacidad de cálculo a disposición no permite encontrar la mejor alternativa entre los miles de combinaciones posibles, por lo que se recurre al RandomizedSearchCV():

```
random_search = RandomizedSearchCV(estimator = forest_reg,
                                   param_distributions = param_grid,
                                   n_iter = 50,
                                   scoring = None,
                                   cv = 3)

random_search.fit(data_prepared, data_labels)
```

Los mejores resultados se resumen en la siguiente tabla:

	Valor hiperparámetros								
'n_estimators'	500	200	500	140	1000	1000	1000	100	…
'min_samples_split'	2	6	4	8	6	10	10	2	…
'min_samples_leaf'	3	4	4	3	4	4	3	3	…
'max_features'	'auto'	'auto'	'auto'	'auto'	'auto'	'auto'	'auto'	'auto'	…
'max_depth'	200	200	500	None	1000	None	None	200	…
'bootstrap'	True	True	True	True	True	True	True	True	…
Raíz del error cuadrático medio	263,01	263,06	263,22	263,23	263,26	263,29	263,56	263,85	…

Tabla 5 - Resultados del RandomizedSearchCV()

Con solamente aplicar hiperparámetros escogidos al azar se observa una mejoría en los resultados; adicionalmente, se observa que en las mejores combinaciones, los hiperparámetros "bootstrap" y "max_features" resultan iguales al valor asignado por defecto, mientras que otros se han estabilizado en un entorno reducido, lo que permite crear una nueva cuadricula de parámetro suficientemente pequeña para ser utilizada en una nueva ronda de GridSearchCV():

```
param_grid = [{ 'n_estimators': [200, 1000],
                'min_samples_split': [2, 6, 10],
                'min_samples_leaf': [3, 4],
                'max_depth': [200, None],
               }]
```

Obteniéndose el siguiente mejor resultado:

	Valor hiperparámetros
'n_estimators'	1000
'min_samples_split'	6
'min_samples_leaf'	3
'max_depth'	None
Raíz del error cuadrático medio	262,95

Tabla 6 - Resultado tercera aplicación del GridSearchCV()

Siendo la mejora del resultado despreciable con respecto al conjunto de parámetros anteriores, se finaliza aquí la fase de refino.

Establecidos todos los hiperparámetros, lo último por hacer es reajustar el modelo empleando todos los datos del conjunto de entrenamiento a la vez, sin guardar un décimo de estos para la validación cruzada, aprovechando así todas las informaciones a disposición:

```
final_regression_model = RandomForestRegressor(n_estimators = 1000,
                                               min_samples_split = 6,
                                               min_samples_leaf = 3,
                                               max_features = 'auto',
                                               max_depth = None,
                                               bootstrap = True)

final_regression_model.fit(data_prepared, data_labels)
```

El final_regression_model, así definido, está listo para ser evaluado con el conjunto de prueba y hacer nuevas previsiones a través del método predict().

7 REDES NEURONALES

Una red neuronal es un modelo computacional que se compone de nodos y conexiones, en los cuales ingresa una señal que fluye y se transforma hasta generar un valor de salida. El elemento constitutivo de la red neuronal es la neurona artificial, o sea, un nodo lógico que, recibida una serie de inputs, genera un output en función de estos siempre que se satisfaga una condición preestablecida.

Las neuronas artificiales son organizadas en capas: la primera tiene como inputs los valores de la base de datos, y los outputs de sus neuronas constituyen el input de la capa siguiente. Esto se repite hasta la última capa, que devuelve el resultado de la previsión. El número de neuronas en cada capa intermedia es variable, mientras que para las dos capas límite coincide con el número de atributos en la primera y con el número de parámetros a estimar en la última (en nuestro caso, uno, o sea, el número de bicis alquiladas).

Todos los ingresos de una neurona están asociados a un parámetro peso ("weight"), que multiplica el valor de estos, y cada neurona también recibe en ingreso una cantidad que no depende de la capa precedente, que actúa como sesgo ("bias"): al principio, el valor de estos parámetros es asignado aleatoriamente, siendo su posterior modificación lo que permite a la red aprender a hacer previsiones.

Las entradas así modificadas son combinadas en un único valor mediante la función de transferencia, cuyo resultado se pasa a la función de activación. Ésta introduce en la transformación de las entradas una cierta no linealidad que permite a la red aprender patrones complejos.

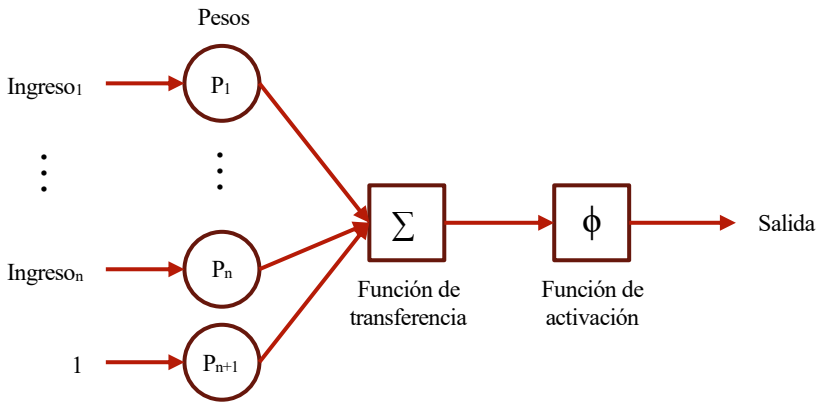


Ilustración 15 - Ejemplo de neurona artificial

El desarrollo de la red neuronal se basa en la librería Tensorflow, lo que requiere sistematizar la base de datos en forma de tensores, o sea, vectores multidimensionales de elementos de tipología uniforme y, consecuentemente, hacer algunos cambios en la etapa de preparación de los datos.

A cada atributo de la base de datos debe corresponderle un tensor distinto, de naturaleza diferente según la tipología del atributo: se emplean tensores de tipo columna numérica ("tf.feature_column.numeric_column") para los atributos numéricos y cíclicos, en este último caso, con dos tensores distintos para las columnas "seno" y "coseno"; se emplea un tensor de tipo columna categórica para los atributos categóricos ("tf.feature_column.categorical_column_with_vocabulary_list").

La existencia de una categoría de tensores apta para tratar atributos categóricos permite descartar del pipeline de transformación el "**cyc_pipeline**", dedicado a la descomposición de los atributos categóricos en matrices de atributos binarios. Se sustituye esta transformación con la simple modificación del nombre de los atributos, enmarcándolos entre dos símbolos de dólares "$" para señalar sus diferentes naturalezas.

La creación de los tensores se hace por medio de la función create_tensores(dataframe = base_de_datos):

```
def create_tensores(dataframe):
  lista_tensores = []
  for header in dataframe.columns:
    if (header[0] is not "$") and (header[-1] is not "$"):
      lista_tensores.append(tf.feature_column.numeric_column(header))
    else:
      lista = tf.feature_column.categorical_column_with_vocabulary_list(header[1:-1],
                                                    crea_diccionario(dataframe.filter([header], axis = 1)))
      lista_tensores.append(tf.feature_column.indicator_column(lista))
      dataframe.rename(columns = {header:header[1:-1]}, inplace = True)
      dataframe[header[1:-1]] = dataframe[header[1:-1]].astype(int)
  return lista_tensores
```

La función recibe una base de datos y devuelve un vector que contiene los tensores de tipología adecuada, diferenciando entre columnas numéricas y categóricas gracias a los símbolos de dólares antedichos. Las categóricas, por su parte, precisan de un vocabulario, el cual es generado con la función crea_diccionario().

El desarrollo de una red neuronal se construye de tres etapas principales: construcción, entrenamiento y evaluación de los resultados. Dado que algunos hiperparámetros del modelo (como el número de capas) se establecen en la etapa de construcción y la eficacia de estos se comprueba sólo en la fase de evaluación, se genera un ciclo de aprendizaje que lleva a repetir varias veces el proceso de desarrollo desde el principio, de acuerdo con los resultados obtenidos.

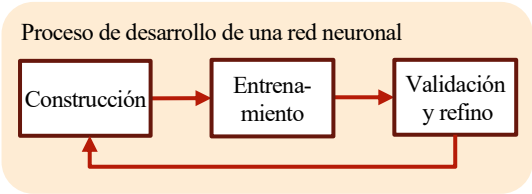


Ilustración 16 - Esquema del capítulo

Una vez la etapa de evaluación ha proporcionado un resultado satisfactorio, se puede emplear la red para hacer previsiones basadas en nuevos datos.

7.1 Construcción

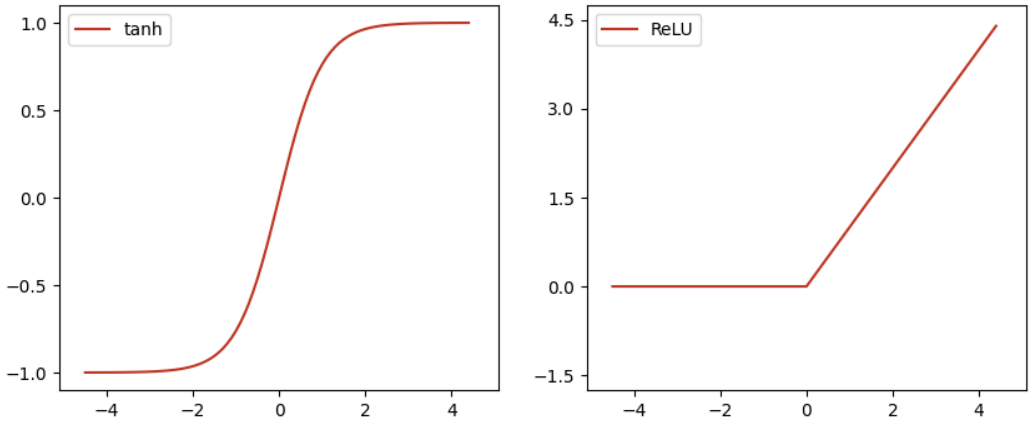
En Tensorflow se dispone de modelos de redes neuronales prefabricados, basados en la clase tf.estimator.Estimator. Partiendo de estos, se pueden construir redes neuronales distintas modificando los hiperparámetros oportunos. Los tres hiperparámetros principales son:

- hidden_units = [N_1, ... N_n]
  Es un vector de ***n*** números enteros, donde ***n*** establece el número de capas entre las dos capas límite, y los números enteros indican cuántas neuronas tendrá cada capa. Las capas pueden tener un número de neuronas decreciente siguiendo el razonamiento de que características de bajo nivel pueden aunarse en

menos características de alto nivel (Géron, 2019), o bien pueden tener el mismo número, para simplificar la etapa de refinación empleando un solo hiperparámetro.
No hay ninguna fórmula para establecer con antelación el numero óptimo de neuronas que se ha da asignar a cada capa, pero generalmente se emplea una cantidad en torno al número de atributos de la base de datos.

- feature_columns
  Constituye el medio para introducir en el modelo los valores contenidos en la base de datos, encajando las columnas de ésta en los tensores correspondientes.

- activation_fn
  Define la función de activación aplicada por la neurona. Las dos más empleadas son la función tangente hiperbólica (tf.nn.tanh) y el Rectificador Lineal Unitario (tf.nn.relu), siendo éste último el más empleado por su rapidez computacional:



$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad f(x) = \max(0, x)$$

Gráfica 12 - Funciones de activación

La función de activación juega un papel clave en la fase de entrenamiento porque su derivada permite establecer cuánto ha contribuido cada neurona al error en la previsión y, consecuentemente, qué inputs deben ser modificados.

7.2 Entrenamiento

La fase de entrenamiento empieza con la generación de varios lotes de datos ("batches") a partir del training set, para emplearlos de forma secuencial durante las dos subfases del paso *hacia adelante* y del paso *hacia atrás*, que se repiten alternativamente.

Durante el paso hacia adelante, los atributos del paquete de datos atraviesan cada capa modificándose según los pesos y los sesgos elegidos aleatoriamente, hasta llegar a la última capa, generando, entonces, una previsión para cada serie de atributos. Dichas previsiones se comparan con las respectivas etiquetas durante el paso hacia atrás, midiendo cuánto defieren a través de una función de pérdida ("loss function"), que por ejemplo puede ser la desviación cuadrática media. El algoritmo computa la derivada de este error frente a todos los pesos y sesgos de la red para identificar y modificar lo que más contribuyó en la generación del error. Tras la corrección, la red resultante recibe el segundo paquete de datos y se repite el ciclo hasta terminar los paquetes.

El número total de iteraciones ("steps") depende, por un lado, del tamaño de los paquetes de datos y, por el otro, del número de repeticiones ("epochs"), o sea, de cuántas veces se pasa al algoritmo el training set entero.

El número de repeticiones influencia la variancia de la red neuronal: para que la red pueda aprender las relaciones entre las variables es necesario iterar numerosas veces, pero sin exceso para no incurrir en el sobreajuste. Igualmente hay que encontrar un equilibrio en el tamaño de los paquetes de datos: para que el algoritmo llegue a una mejora en las previsiones es necesario que tenga una visión general sobre los datos, es decir, necesita paquetes de una cierta dimensión, pero un tamaño demasiado elevado incrementa el riesgo de atascarse en un

óptimo local (Krizhevsky, 2014).

La división del conjunto de entrenamiento y el sucesivo suministro de los paquetes al modelo se hace a través de la función de entrada train_input_fn(). La función recibe el training set, sus etiquetas y los parámetros "batch_size" y "num_epochs", que controlan el tamaño de los paquetes y el número de repeticiones; el parámetro "shuffle" es predeterminado y establece que los paquetes se extraigan desde el training set al azar. Los lotes así creados son suministrados secuencialmente al modelo en fase de entrenamiento:

```
def train_input_fn(train_df, train_labels, batch_size, num_epochs):
  return tf.estimator.inputs.pandas_input_fn(
    x = train_df,
    y = train_labels,
    num_epochs = num_epochs,
    batch_size = batch_size,
    shuffle = True
  )
```

El modelo se entrena a través del método train():

```
trained_model = model.train(input_fn = train_input_fn(train_df = data_prepared,
                                                      train_labels = data_labels,
                                                      num_epochs = num_epochs,
                                                      batch_size = batch_size),
                            max_steps = None)
```

Aparte de los paquetes proporcionados desde la función de entrada, el método recibe el hiperparámetro max_steps, que sirve para limitar el número de veces que la red neuronal será entrenada en aquellos casos en los que se requiera un número inferior de entrenamientos de lo que resulta multiplicando el número de repeticiones por el número de paquetes.

Todas las etapas de la fase de entrenamiento están resumidas en el siguiente esquema:

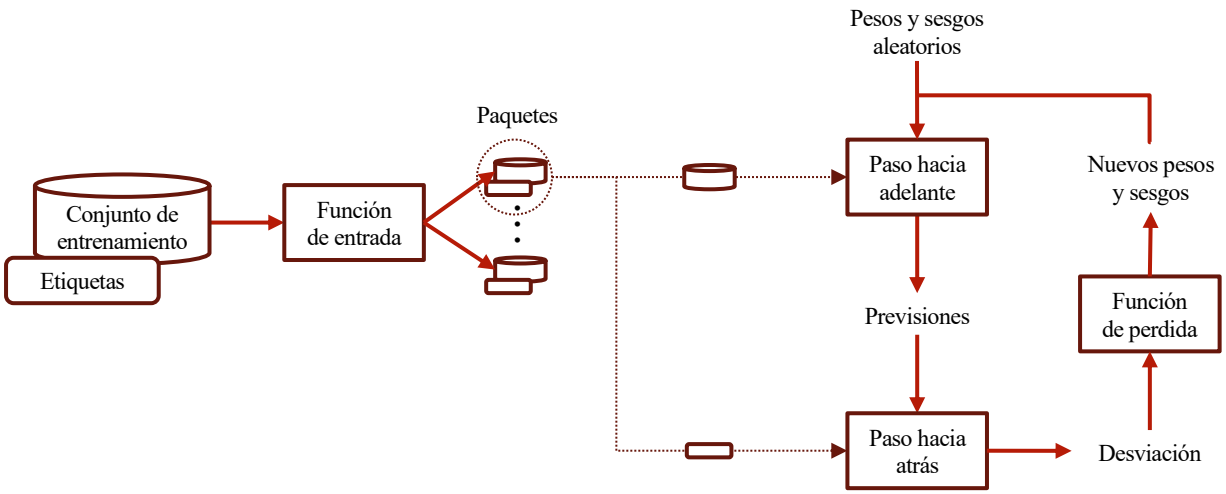


Ilustración 17 - Esquema fase de entrenamiento

Terminada la fase de entrenamiento, el modelo está listo para hacer nuevas previsiones.

7.3 Validación cruzada y refino de los hiperparámetros

La técnica de validación de la red neuronal es la misma que se ha aplicado con los otros algoritmos de machine learning: el conjunto de entrenamiento se fracciona en ***n*** lotes y todos menos uno, se emplean para entrenar el modelo con las respectivas etiquetas, intentando después estimar el último lote y midiendo la diferencia entre el resultado obtenido y el real valor del label. El proceso se repite varias veces, dejando fuera de la etapa de entrenamiento un lote cada vez distinto, para así obtener ***n*** desviaciones diferentes. La media de éstas es una buena aproximación de la capacidad de generalización del modelo.

Dado que el cross_val_score() no se puede emplear con redes neuronales, se ha desarrollado una función propria que pueda hacer lo mismo, la cross_val_score_modificado(). La función recibe el modelo de red neuronal a

entrenar, los datos del training set y los parámetros de entrenamiento, además del número de paquetes en el cual fraccionar el conjunto de entrenamiento para la validación, en este caso, diez:

```
def cross_val_score_modificado(model, train_data, data_labels, num_epochs, batch_size, cv =10):
  ...
  return scores, sum(scores)/len(scores), train_time/cv
```

La función devuelve las diez desviaciones calculadas, su media y el tiempo medio de entrenamiento.

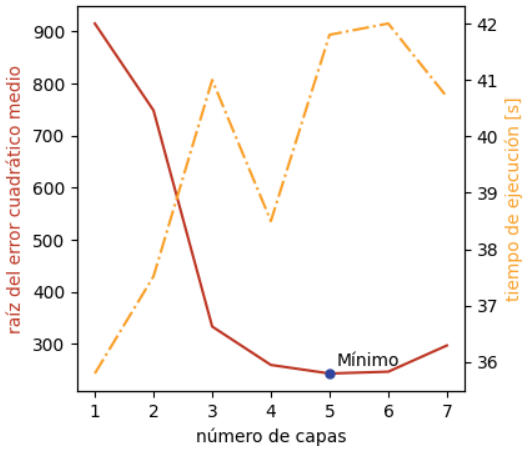
7.4 Ciclo de Aprendizaje

El primer parámetro que se ha tomado en consideración es el número de capas intermedias: se han entrenado y evaluado redes con un número de capas crecientes, empezando por una. Teniendo que fijar los otros parámetros, se han aplicado capas de 37 neuronas (exactamente el número de atributos, considerando que los atributos categóricos se descomponen en un atributo para cada alternativa posible), con la función de activación igual al Rectificador Lineal Unitario.

Finalmente, el entrenamiento se ha hecho con num_epochs = 300 y batch_size = 8000, dos valores no excesivamente desafiantes desde el punto de vista computacional.

Numero de capas intermedias	Raíz del error cuadrático medio
1	915,59
2	748,78
3	333,47
4	259,76
5	243,29
6	246,74
7	297,15

Tabla 7 - Resultados primera ronda de entrenamiento



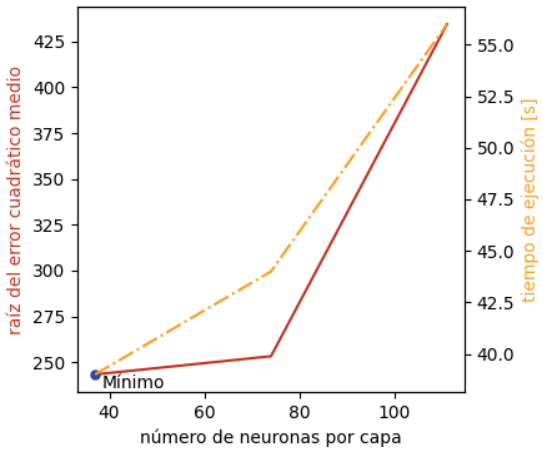
Gráfica 13 - Resultados primera ronda de entrenamiento

El mejor resultado se ha obtenido con cinco capas, estructura que se ha, por lo tanto, mantenido en las etapas siguientes. La razón por la cual la fiabilidad de las previsiones tiene un mínimo es que el número de capas en una red neuronal influye sobre la complejidad de las relaciones que la red neuronal puede investigar: un número no óptimo de capas conduce a fenómenos de subajuste o sobreajuste.

Una vez establecido el número de capas se ha pasado a definir el número óptimo de neuronas, quedándose en la condición de partida de 37 neuronas.

Numero de neuronas por capa	Raíz del error cuadrático medio
37	243,29
74	253,23
111	434,39

Tabla 8 - Resultados segunda ronda de entrenamiento



Gráfica 14 - Resultados segunda ronda de entrenamiento

En cuanto al último parámetro constructivo a establecer en la fase de construcción, esto es, la función de activación de las neuronas, se ha intentado aplicar la tangente hiperbólica. En este caso se ha obtenido una desviación media superior a 1400, mucho peor que cualquier resultado conseguido hasta ahora: se ha confirmado, entonces, el uso del Rectificador Lineal Unitario.

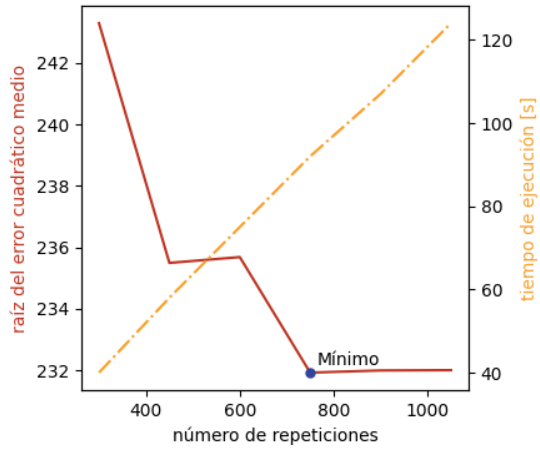
Una vez fijada la estructura de la red en cinco capas intermedias de 37 neuronas, se ha pasado a optimizar los parámetros de entrenamiento. Para este fin se procede en paralelo, modificando uno mientras se fija el otro en el valor inicial.

Numero de repeticiones	Raíz del error cuadrático medio
300	243,29
450	235,49
600	235,68
750	231,92
900	231,99
1050	232

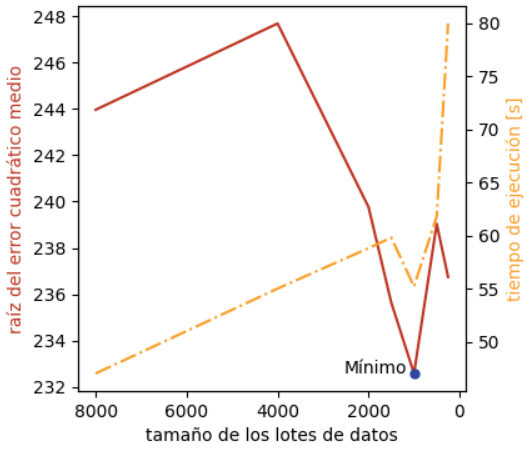
Tabla 9 - Resultados tercera ronda de entrenamiento

Tamaño de los lotes de datos	Raíz del error cuadrático medio
8000	243,97
4000	247,7
2000	239,77
1500	235,64
1000	232,57
500	239,04
250	236,74

Tabla 10 - Resultados cuarta ronda de entrenamiento



Gráfica 15 - Resultados tercera ronda de entrenamiento



Gráfica 16 - Resultados cuarta ronda de entrenamiento

La mejor solución se ha encontrado para un número de repeticiones (“num_epochs”) igual a 750 y un tamaño de lote de datos (“batch_size”) igual a 1000. Con estos dos parámetros a la vez se ha obtenido una desviación media de 230,64, el mejor resultado hasta el momento.

		Valor hiperparámetros
Hiperparámetros de construcción	hidden_units	[111, 111, 111, 111]
	activation_fn	ReLU
Hiperparámetros de entrenamiento	num_epochs	750
	batch_size	1000
Raíz del error cuadrático medio	230,64	

Tabla 11 - Hiperparámetros característicos de la mejor red neuronal encontrada

Terminado el ciclo de aprendizaje, se puede volver a entrenar la red neuronal una última vez aprovechando todos los datos del conjunto de entrenamiento, sin guardar un décimo de estos para la validación cruzada:

```
trained_model = model.train(input_fn = input_fn_train(train_df = data_prepared,
                                                      train_labels = data_labels,
                                                      num_epochs = 600,
                                                      batch_size = 1000),
                            max_steps = None)
```

8 RESULTADOS

Una vez concluida la fase de refino, los mejores modelos están listos para ser probados, haciendo previsiones sobre los datos del conjunto de prueba que se han reservado hasta ahora con este propósito. El primer paso es separar el conjunto de prueba en atributos (“X_test) y etiquetas (y_test), aplicando a los primeros la etapa de preparación:

```
X_test = test_set.drop("cnt", axis=1)
y_test = test_set["cnt"].copy()

X_test_prepared = Full_Pipeline.transform(X_test)
```

Tras ello, se emplean entonces los algoritmos seleccionados para predecir la cantidad de bicis alquiladas en los horarios que pertenecen al conjunto de prueba. Después, el resultado obtenido se confronta con las etiquetas.

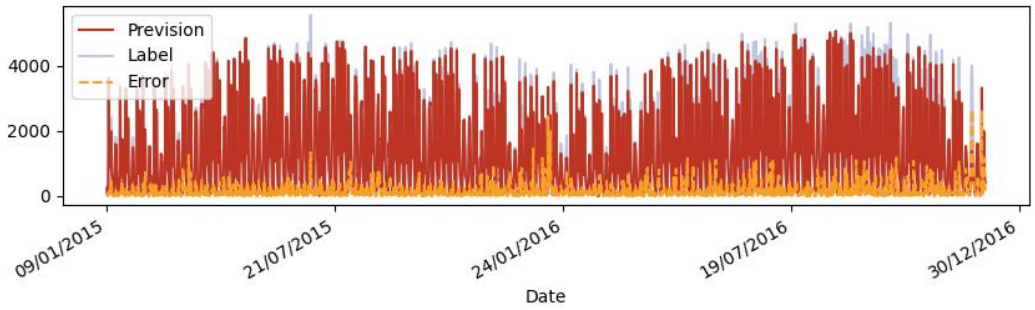
8.1 Resultados del Bosque Aleatorio

El primer algoritmo que se evalúa es el bosque aleatorio, empleando el método predict():

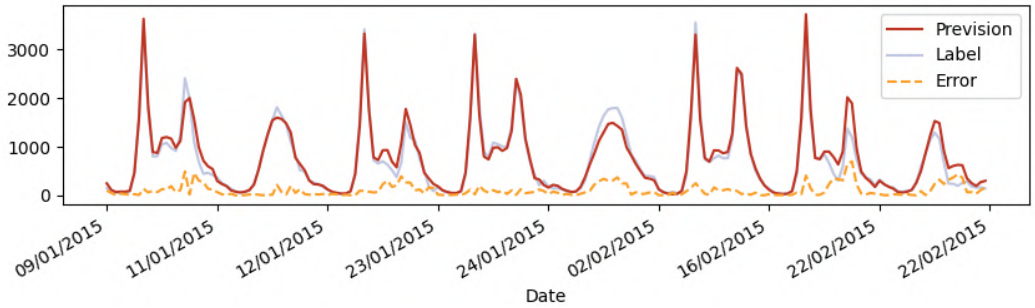
```
final_predictions = refined_forest_regressor.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

Se obtiene un error de previsión medio igual a 260,61, un resultado acorde con el error medio obtenido con la validación cruzada, 262,95, lo que confirma la bondad del modelo y la ausencia de sobreajuste. La ligera mejoría puede ser debida a la dimensión superior de la base de datos sobre la cual el bosque aleatorio ha sido ajustado (todo el conjunto de prueba en lugar del 90% que restaba de reservar el 10% para la validación cruzada). No obstante, no se puede descartar que sea simplemente producto de la aleatoriedad que caracteriza a este algoritmo.

Las previsiones hechas, junto al valor de las etiquetas y al error cometido en la estimación, se han trazado en las siguientes gráficas (ya que es el conjunto de prueba, se trazan sólo algunos días del año):



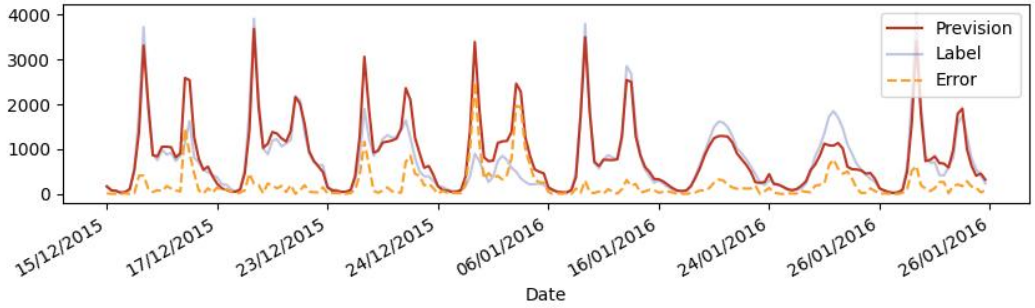
Gráfica 17 - Previsiones del bosque aleatorio, etiquetas y errores con respecto al conjunto de prueba



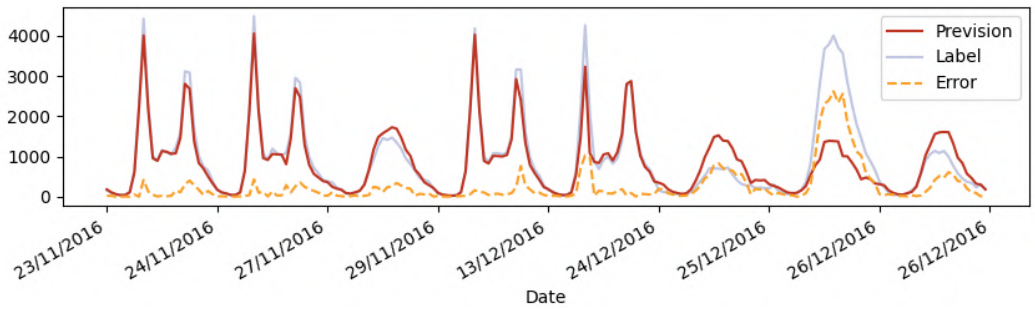
Gráfica 18 - Previsiones del bosque aleatorio, etiquetas y errores con respecto al conjunto de prueba - Aumento

Solamente en la gráfica 17 ya se observa que las previsiones son generalmente buenas, y el aumento muestra cómo las previsiones siguen bien el patrón, tanto el de los días laborales como el de los festivos.

Se advierte un aumento del error en el período navideño, como se muestra en las dos siguientes gráficas:



Gráfica 19 - Previsiones del bosque aleatorio, etiquetas y errores con respecto al conjunto de prueba - Aumento



Gráfica 20 - Previsiones del bosque aleatorio, etiquetas y errores con respecto al conjunto de prueba - Aumento

8.2 Resultados de la Red Neuronal

Se sigue entonces con la evaluación de la red neuronal aplicando el método predict():

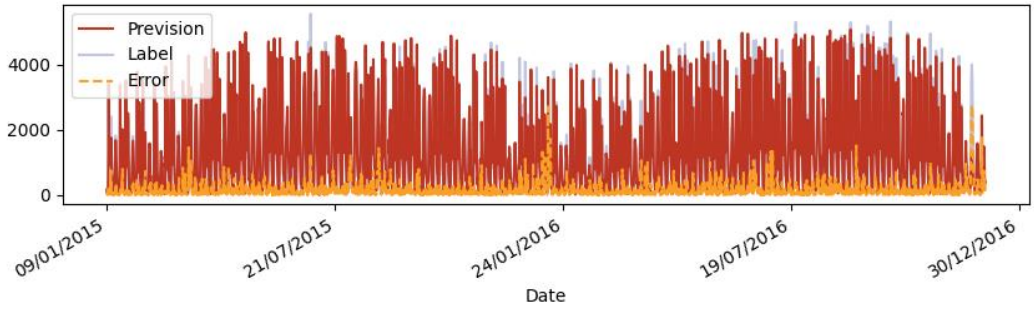
```
predictions_iterator = trained_model.predict(input_fn = input_fn_predict(train_df = X_test_prepared,
                                                                          train_labels = test_label_empty))
```

Análogamente al procedimiento seguido en la etapa de entrenamiento, los datos se suministran al algoritmo en lotes a través de la función input_fn_predict(), que recibe el conjunto de prueba y lo devuelve en paquetes, manteniendo intacto el orden de los datos. La función también recibe y devuelve test_label_empty, o sea, una base de datos vacía donde se almacenarán las previsiones.

A diferencia de en el bosque aleatorio, las previsiones generadas son almacenadas como un objecto iterador, así que es necesario extraerlas en un vector antes de seguir con la evaluación:

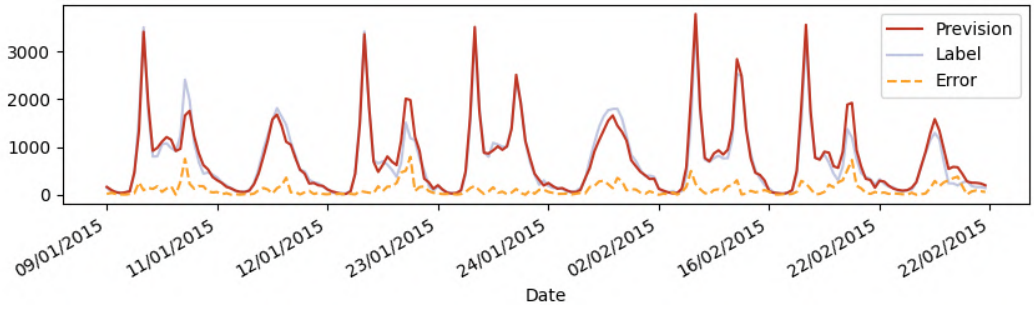
```
final_predictions = []
for prediction in predictions_iterator:
  final_predictions.append(prediction['predictions'][0])

final_mse = mean_squared_error(test_labels, final_predictions)
final_rmse = np.sqrt(final_mse)
```

Se obtiene un error de previsión medio igual a 249,92; mejor que el del bosque aleatorio, pero peor que el de la última ronda de validación cruzada. Las previsiones hechas, junto al valor de las etiquetas y al error cometido en la estimación, se han trazado en la siguiente gráfica (puesto que es el conjunto de prueba, se trazan sólo algunos días del año):



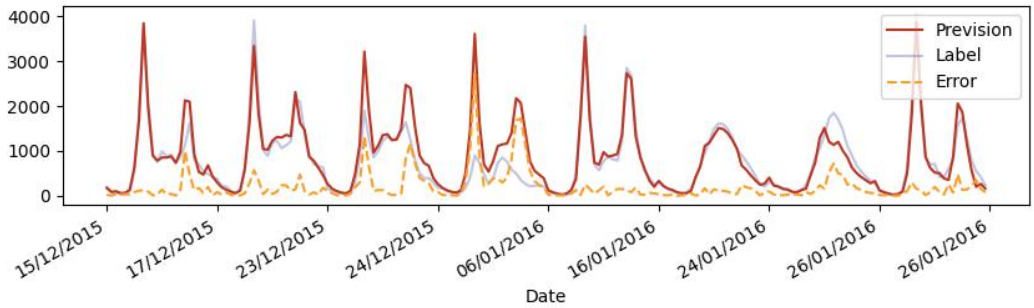
Gráfica 21 - Previsiones de la red neuronal, etiquetas y errores con respecto al conjunto de prueba

Tomando un periodo menor, se observa un buen resultado tanto en los días laborales como en los festivos:

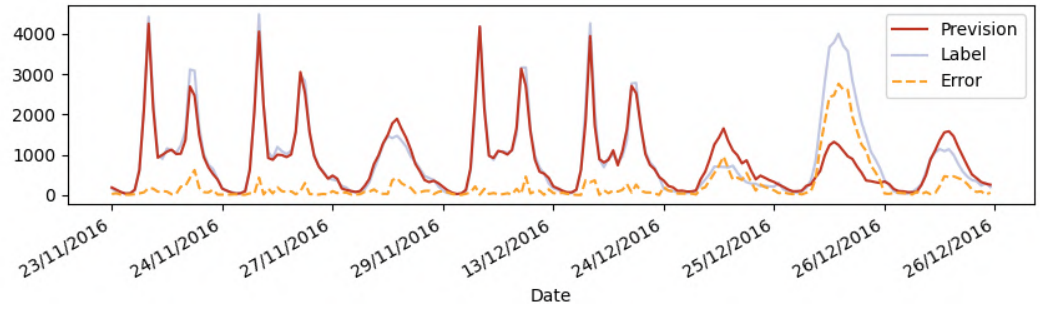


Gráfica 22 - Previsiones de la red neuronal, etiquetas y errores con respecto al conjunto de prueba - Aumento

Tal y como sucede con el bosque aleatorio, los errores más importantes se concentran en torno a las navidades:



Gráfica 23 - Previsiones de la red neuronal, etiquetas y errores con respecto al conjunto de prueba - Aumento



Gráfica 24 - Previsiones de la red neuronal, etiquetas y errores con respecto al conjunto de prueba - Aumento

No sorprende que los errores más importantes se concentren en el periodo de las navidades: de la mera observación de la gráfica se desprende claramente la atipicidad de dicho periodo, caracterizado por días laborales de escasa demanda y valores completamente fuera del estándar el día de Navidad, cuando las bicis son prácticamente el único sistema de transporte público disponible en la ciudad.

TRANSPORT
FOR LONDON

Christmas and New Year travel

Between Wednesday 23 December 2020 and Sunday 3 January 2021, most of our services will be running. However, there will be service changes and planned work on the public transport and road networks.

Travel changes and information

Between Wednesday 23 December 2020 and Sunday 3 January 2021, planned work, closures and service changes may affect your journey.

- The only transport running on Christmas Day will be taxis and private hire; Santander Cycles; and coaches
- There are likely to be changes to timetables on all TfL services

Ilustración 18 - Comunicación del cierre navideño desde Transport for London

9 CONCLUSIONES

Los resultados obtenidos con el conjunto de prueba muestran que los modelos seleccionados consiguen prever la demanda con buena precisión, con una media del error cuadrático comparable a la que se obtenía en las etapas de validación cruzada.

En el caso de la red neuronal, el resultado un poco inferior a las expectativas sugiere un sobreajuste de la red al conjunto de entrenamiento, aunque no se puede descartar que la mayor desviación sea debida a la aleatoriedad que caracteriza la fase de entrenamiento de la red neuronal. En cualquier caso, no es preocupante puesto que el resultado obtenido es muy bueno.

Merecen una mención aparte los errores de previsión en el periodo de las navidades, justificados por el patrón de demanda tan atípica que los caracteriza: limitándose la base de datos a dos años era improbable que los modelos aprendiesen a estimar correctamente el uso de las bicis en fechas tan señaladas.

Ambos algoritmos están ahora listos para hacer nuevas previsiones. Para garantizar un desempeño constante, es necesario seguir actualizando la base de datos para interceptar los futuros cambios en el patrón de uso de las bicis (por ejemplo, un incremento de los alquileres diarios debido a un crecimiento del número de usuarios y de estaciones disponibles).

En cuanto a posibles ampliaciones de búsqueda, es posible dirigirse hacia varias direcciones:

- Si se tuviese una base de datos más detallada, con información sobre la estación de inicio y fin de cada alquiler, se podría intentar predecir los movimientos de las bicicletas entre varias zonas de la ciudad, a fin de organizar mejor el servicio de redistribución de las bicis.
- Habiéndose observado la fuerte influencia que la huelga y el cierre del metro tienen sobre el uso de las bicis, sería interesante ampliar la base de datos con informaciones sobre el transporte público de la ciudad. Noticias sobre el cierre por mantenimiento de estaciones o líneas del metro, así como datos sobre los horarios de servicio, podrían resultar de gran utilidad en el refino de las previsiones.

Por último, está claro que el caso de Londres puede ser fácilmente adaptado a cualquier otra ciudad: basta con disponer de la base de datos adecuada. Es preciso subrayar que las herramientas desarrolladas son suficientemente flexibles para ser empleadas en problemas muy distintos siempre que estos sean modelados como previsión de una variable dependiente basado en un conjunto de atributos independientes.

REFERENCIAS

Géron, A. (2019) *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media.

Krizhevsky, A. (2014) 'One weird trick for parallelizing convolutional neural networks'. Available at: http://arxiv.org/abs/1404.5997.

Rudloff, C. *et al.* (2015) 'Influence of weather on transport demand: Case study from the Vienna, Austria, Region', *Transportation Research Record*, 2482(January), pp. 110–116. doi: 10.3141/2482-14.