Inteligencia de Negocio. Práctica 3: Competición de Kaggle

Patricia Córdoba Hidalgo
patriciacorhid@correo.ugr.es
Grupo 2 (Viernes)

27 de diciembre de 2020

Índice

1. Introducción

Nuestro problema consiste en predecir el precio de un conjunto de instancias de coches a partir de ciertas características de éste. Es un problema de aprendizaje supervisado, más concretamente de clasificación. Las etiquetas correspondientes a la variable `Precio_cat`, aquella que queremos predecir, son las siguientes:

1. Coches baratos.
2. Coches menos baratos.
3. Coches con precio promedio.
4. Coches más caros que el promedio.
5. Coches muy caros.

Las características de los coches usadas para predecir su precio son:

- Nombre del tipo de coche.
- Ciudad de venta del coche.
- Año del coche.
- Kilómetros recorridos del coche.
- Tipo de combustible del coche.
- Tipo de marcha.
- Mano (primera mano, segunda mano, tercera mano, cuarta o más).
- Consumo.
- CC del motor.
- Potencia del motor.
- Asientos del coche.
- Descuento realizado por oferta.

Usaré distintas técnicas y modelos para predecir las etiquetas del conjunto de test. Estas etiquetas serán subidas es un fichero llamado `p3_{número}.csv` a la plataforma Kaggle. La métrica usada para representar la bondad del ajuste es *accuracy*.

2. Lista de intentos

Archivo	Fecha	Pos.	Acc. Train	Acc. Test	Preprocesado	Algoritmos	Parámetros
p3_00.csv	13:20:11 20/12/20	7	0.826657	0.75237	Eliminación de la variable Descuento Uso de Labelencoder	Random Forest	criterion = gini n_estimators = 150 ccp_alpha=0
p3_01.csv	19:03:32 20/12/20	4	0.901557	0.76186	SMOTE	Random Forest	criterion = entropy n_estimators = 220 ccp_alpha=0.00023
p3_02.csv	15:40:15 21/12/20	4	0.900445	0.75582	Cambio de nombres por marcas	Random Forest	criterion = entropy n_estimators = 130 ccp_alpha=0
p3_04.csv	20:49:52 21/12/20	2	0.86529	0.79637	Normalización + OneHotEncoder	MLP	activation = tanh hidden_layer_sizes = =(300,300) alpha=default
p3_05.csv	20:52:47 21/12/20	2	0.881868	0.68162	Normalización + OneHotEncoder	Knn	n_neighbors = 1
p3_06.csv	15:07:36 22/12/20	3	0.920356	0.79982	Normalización + OneHotEncoder	Gradient Boosting	n_estimators = 760 min_sample_leaf = 8 learning_rate = 0.1
p3_07.csv	17:25:41 22/12/20	3	0.9330367	0.80241	Normalización + OneHotEncoder	Stacking con Gradient Boosting MLP y Random Forest	final_estimator = LogisticRegression con max_iters = 400
p3_08.csv	16:53:58 24/12/20	3	0.934805	0.79376	Normalización + OneHotEncoder + LocalOutliersFactor	Stacking con Gradient Boosting MLP y Random Forest	final_estimator = LogisticRegression con max_iters = 400
p3_09.csv	20:34:30 24/12/20	3	0.9339265	0.80845	Normalización + OneHotEncoder	Stacking con Gradient Boosting MLP Random Forest y lightbmClassifier	**Stacking** final_estimator = LogisticRegression con max_iters = 400 **LGBM** boosting_type = = 'goss' num_leaves = 35 max_depth = 10 n_estimators = 150
p3_10.csv	20:36:50 24/12/20	1	0.930033	0.82830	Normalización + OneHotEncoder	Stacking con Gradient Boosting MLP y lightbmClassifier	final_estimator = LogisticRegression con max_iters = 400

NOTA: No consideré oportuno subir el fichero `p3_03.py`, se puede considerar que ese fichero es inexistente. No es ningún error que no aparezca en la tabla de intentos.

3. Intentos

3.1. p3_00

Antes de nada analizamos el conjunto de training. Con la orden `datos.info()` obtenemos los atributos que componen este conjunto, el número de instancias que componen dicho conjunto y el número de valores nulos:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4819 entries, 0 to 4818
Data columns (total 14 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             4747 non-null   float64
 1   Nombre         4747 non-null   object
 2   Ciudad         4747 non-null   object
 3   Año            4747 non-null   float64
 4   Kilometros     4747 non-null   float64
 5   Combustible    4747 non-null   object
 6   Tipo_marchas   4747 non-null   object
 7   Mano           4747 non-null   object
 8   Consumo        4746 non-null   object
 9   Motor_CC       4718 non-null   object
 10  Potencia       4644 non-null   object
 11  Asientos       4713 non-null   float64
 12  Descuento      659 non-null    float64
 13  Precio_cat     4819 non-null   int64
dtypes: float64(5), int64(1), object(8)
memory usage: 527.2+ KB
```

La variable Descuento posee más valores nulos que el resto, un total de 4160 valores perdidos.

Tras esto representamos el número de elementos por clase y observamos que las clases están desbalanceadas:

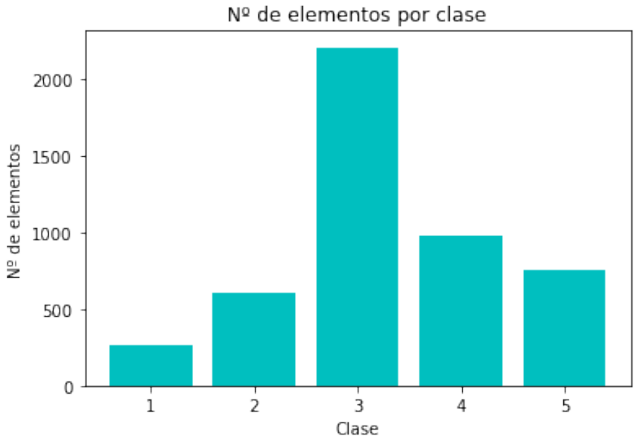


Figura 1: Datos en cada clase

Una vez que tenemos una idea de la distribución de los datos, pasamos a trabajar con ellos. Primero trabajamos con los valores perdidos. Si eliminásemos todos los valores perdidos, contaríamos sólo con el 11.68 % de los originales, con lo que perderíamos mucha información, así que descarté ese preprocesamiento.

El preprocesamiento que realicé finalmente consiste en eliminar la variable Descuento, ya que el 86.32 % de sus valores son perdidos y eliminar el resto de valores perdidos restantes. Esto supone quedarnos con el 81.88 % de los datos originales. Reorganicé los índices para que fuesen del 1 al 3945, dado que posteriormente será necesario esta numeración para recorrer el dataframe con bucles.

Esto fue realizado con el código:

```
p2_datos = datos.copy()
del(p2_datos['Descuento'])
p2_datos = p2_datos.dropna()
p2_datos = p2_datos.reset_index()
del(p2_datos['index'])
```

Tras esto se cuantificaron todas las variables cuaitativas usando `LabelEncoder`. El `LabelEncoder` entrena con la lista total de los valores de cada atributo, que se encuentra en los ficheros *.csv* correspondientes a cada uno, y se usa para transformar los datos *train* y *test*. De esta manera, se aplica la misma transformación en ambos.

Las variables *Consumo, Motor_CC* y *Potencia* están codificadas como `string`. Para guardarlas como un `float`, eliminamos las unidades que acompañan a los valores de estas variables. Por ejemplo, el código usado para eliminar la unidad " kmlp" en la variable *Consumo* es:

```
for i in range(len(p2_datos)):
    p2_datos["Consumo"].iloc[i] = float(p2_datos["Consumo"].iloc[i].strip('␣kmlp'))
```

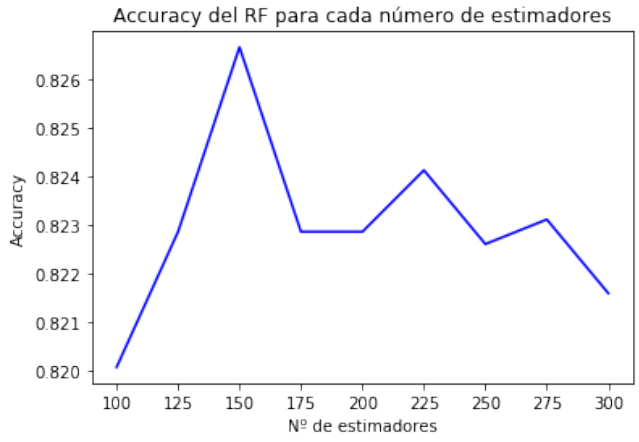
Una vez preprocesados los datos, separamos las etiquetas del resto de variables:

```
cols = [col for col in p2_datos.columns if col not in ['Precio_cat']]
data = p2_datos[cols]
del(data['id'])
target = p2_datos['Precio_cat']
```

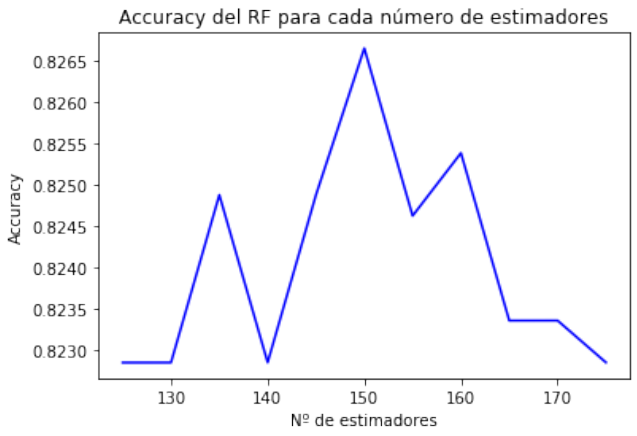
Por último elegí el algoritmo `Random Forest` para resolver este problema y seleccioné los hiperparámetros que iba a usar. Para evaluar la bondad del modelo con cada uno de los diferentes hiperparámetros usé la media de la medida *accuracy* conseguida con cada uno de los cinco conjuntos usados en validación cruzada.

Como criterio de división de nodos, usé el `gini`, ya que la accuracy media obtenida es 0.826657682373137 mientras que con `entropy` consigue un 0.8215879738813753.

Para elegir el número de estimadores usados, probé con diversas cantidades, obteniendo los resultados mostrados en las siguientes gráficas:



(a) Selección de `n_stimators`



(b) Selección de `n_stimators`

En la primera vemos que el máximo se alcanza entorno a 150 estimadores, por lo que realicé intentos menos espaciados entorno a este número. Se aprecia que el máximo se sigue alcanzando en 150 estimadores, que es el valor por defecto de este parámetro.

También intenté identificar el parámetro de poda óptimo. Podemos comprobar que la poda empeora el desempeño del modelo, por lo que este parámetro tampoco lo modificremos.

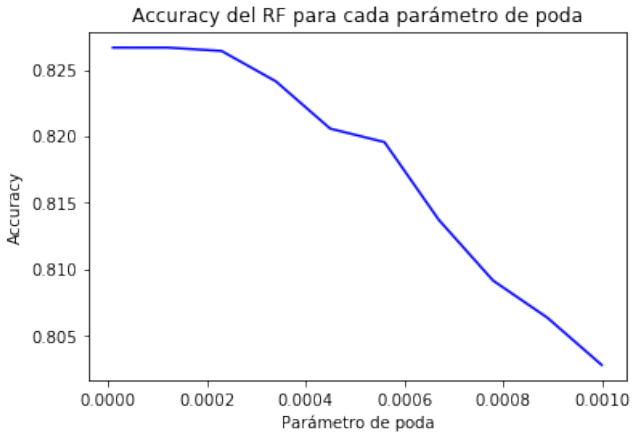


Figura 2: Selección de `ccp_alpha`

Tras todo el ajuste de hiperparámetros, el modelo final es aquel por defecto. Por tanto, para ver la *accuracy* del modelo ejecutamos el siguiente código:

```
rf_clf = RandomForestClassifier(random_state=15, n_estimators=150)
score = cross_val_score(rf_clf, data, target, cv=5)

s = 0
for i in range(len(score)):
    s+= score[i]

print("La accuracy del modelo es: " + str(s/len(score)))
```

La *accuracy* del modelo en el conjunto training es: 0.826657682373137. Tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.75237.

3.2. p3_01

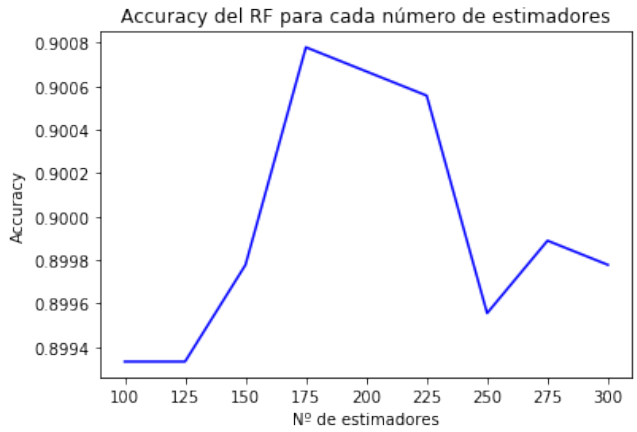
En este intento, tras aplicar el preprocesamiento explicado en la prueba anterior, aplicamos SMOTE (Synthetic Minority Oversampling Technique). Con esto conseguimos que las clases estén balanceadas. El código usado para llevar esto a cabo es:

```
data, target = SMOTE(random_state=15).fit_resample(data, target)
```

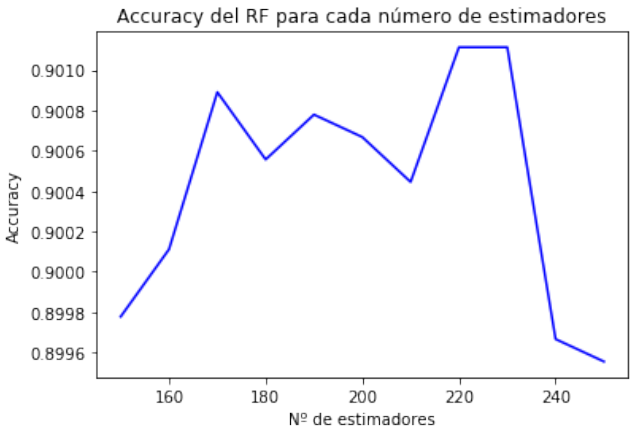
Si ejecutamos la orden `Counter(target)`, comprobamos que todas las clases tienen 1798 instancias.

Después de procesar así los datos, volvemos a ajustar los hiperparámetros del algoritmo `Random Forest`. Esta vez, el criterio de división escogido es `entropy`, que consigue una *accuracy* media de 0.8997775305895438 frente a 0.8972191323692993 que se consigue usando `gini`. Podemos observar que la *accuracy* que consigue el conjunto train con este procesamiento es mayor que sin aplicar oversampling.

Estimamos el número optimo de estimadores de igual manera que en el intento anterior:



(a) Selección de `n_stimators`



(b) Selección de `n_stimators`

A la vista de los resultados tomamos 220 estimadores. También estimamos el valor del parámetro de poda. Escogemos 0.00023 dados los resultados obtenidos en la siguiente gráfica:

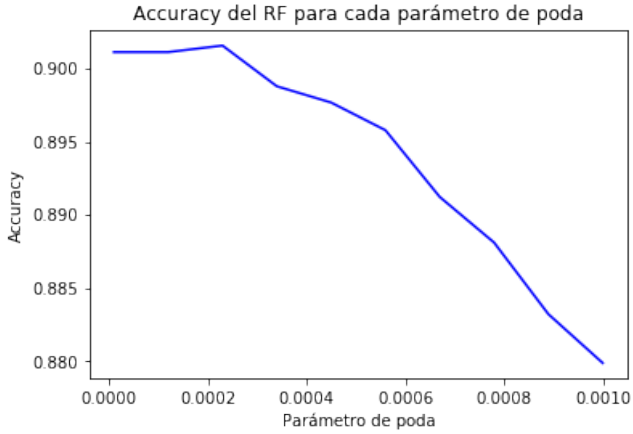


Figura 3: Selección de `ccp_alpha`

La *accuracy* del modelo en el conjunto training es 0.9015572858731924. Tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.76186.

3.3. p3_02

En este intento, después de eliminar la variable Descuento y los valores nulos, y antes de usar `LableEncoder`, sustituí los nombres de los coches por las marcas de los coches usando el siguiente código:

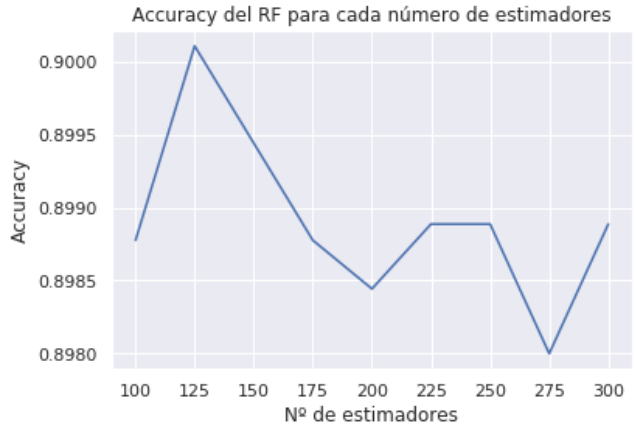
```
for i in range(len(nombre)):
    nombre["Nombre"].iloc[i] = nombre["Nombre"].iloc[i].split(' ')[0]

for i in range(len(p2_datos)):
        p2_datos["Nombre"].iloc[i] = p2_datos["Nombre"].iloc[i].split(' ')[0]
```

El resto del preprocesamiento se mantuvo. Volví a seleccionar los hiperparámetros del `Random Forest`:

El criterio de división elegido fue `entropy`, que consigue una *accuracy* de 0.8994438264738598 frente a 0.8957730812013349 que consigue el criterio `gini`.

Seleccionamos el número de estimadores usado, que en este caso será 130, por la información representada en las siguientes gráficas:



(a) Selección de `n_stimators`

(b) Selección de `n_stimators`

También decidí usar el valor 0 para el parámetro de poda. El motivo se encuentra en la siguiente gráfica:

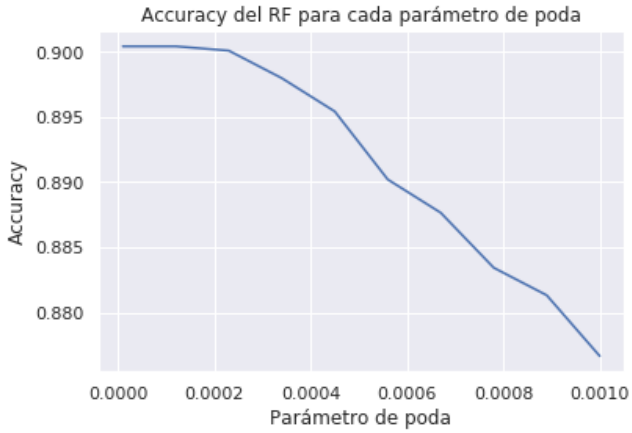


Figura 4: Selección de `ccp_alpha`

La *accuracy* del modelo en el conjunto training es 0.9004449388209121. Aunque es inferior a la obtenida en el intento anterior, pensé que con este procesamiento podría reducirse el sobreajuste y obtener mejores resultados en el conjunto test. Tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.75582, por tanto mis sospechas no eran ciertas.

3.4. p3_04

En este intento quise probar con un modelo diferente, `MLP`. Este modelo requiere un preprocesado diferente, ya que la normalización y el uso de `OneHotVectors` hace que el modelo obtenga mejores resultados.

El procesamiento usado en esta prueba consiste entonces en eliminar, como en las anteriores, la variable Descuento y los valores nulos restantes. Luego pasamos los nombres de los coches a marcas, del mismo modo que hicimos en `p3_02`, para reducir así el número de variables finales (habrá una variable por cada posible valor del atributo cualitativo al que apliquemos `OneHotEncoder`).

En vez de aplicar `LabelEncoder` a las variables cualitativas, usé `OneHotEncoder`. Esto nos crea una nueva variable por cada uno de los posibles valores que tomase la variable a la que se lo aplicamos. Añadimos estas variables al `DataFrame` de datos y eliminamos la variable original.

Un ejemplo del código usado para esto es:

```
encCiudad = OneHotEncoder(handle_unknown='ignore')
encCiudad.fit(ciudad["Ciudad"].to_numpy().reshape(-1, 1))
aux = encCiudad.transform(p2_datos["Ciudad"].to_numpy().reshape(-1, 1)).toarray()
aux = pd.DataFrame(aux)

for i in range(aux.shape[1]):
    p2_datos["Ciudad_" + str(i)] = aux[i]

del(p2_datos['Ciudad'])
```

Tras esto, volvemos a eliminar las unidades de las variables *Consumo, Motor_CC* y *Potencia* como se explicó en `p3_00` y así pasar a `float` sus valores.

La estructura de los datos es ahora:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3946 entries, 0 to 3945
Data columns (total 61 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   id              3946 non-null   float64
 1   Año             3946 non-null   float64
 2   Kilometros      3946 non-null   float64
 3   Consumo         3946 non-null   float64
 4   Motor_CC        3946 non-null   float64
 5   Potencia        3946 non-null   float64
 6   Asientos        3946 non-null   float64
 7   Precio_cat      3946 non-null   int64
 8   Nombre 0        3946 non-null   float64
 9   Nombre 1        3946 non-null   float64
 10  Nombre 2        3946 non-null   float64
 11  Nombre 3        3946 non-null   float64
 12  Nombre 4        3946 non-null   float64
 13  Nombre 5        3946 non-null   float64
 14  Nombre 6        3946 non-null   float64
 15  Nombre 7        3946 non-null   float64
 16  Nombre 8        3946 non-null   float64
 17  Nombre 9        3946 non-null   float64
 18  Nombre 10       3946 non-null   float64
 19  Nombre 11       3946 non-null   float64
 20  Nombre 12       3946 non-null   float64
 21  Nombre 13       3946 non-null   float64
 22  Nombre 14       3946 non-null   float64
 23  Nombre 15       3946 non-null   float64
 24  Nombre 16       3946 non-null   float64
 25  Nombre 17       3946 non-null   float64
 26  Nombre 18       3946 non-null   float64
 27  Nombre 19       3946 non-null   float64
 28  Nombre 20       3946 non-null   float64
 29  Nombre 21       3946 non-null   float64
 30  Nombre 22       3946 non-null   float64
 31  Nombre 23       3946 non-null   float64
 32  Nombre 24       3946 non-null   float64
 33  Nombre 25       3946 non-null   float64
 34  Nombre 26       3946 non-null   float64
 35  Nombre 27       3946 non-null   float64
```

```
 36  Nombre 28       3946 non-null   float64
 37  Nombre 29       3946 non-null   float64
 38  Nombre 30       3946 non-null   float64
 39  Ciudad 0        3946 non-null   float64
 40  Ciudad 1        3946 non-null   float64
 41  Ciudad 2        3946 non-null   float64
 42  Ciudad 3        3946 non-null   float64
 43  Ciudad 4        3946 non-null   float64
 44  Ciudad 5        3946 non-null   float64
 45  Ciudad 6        3946 non-null   float64
 46  Ciudad 7        3946 non-null   float64
 47  Ciudad 8        3946 non-null   float64
 48  Ciudad 9        3946 non-null   float64
 49  Ciudad 10       3946 non-null   float64
 50  Combustible 0   3946 non-null   float64
 51  Combustible 1   3946 non-null   float64
 52  Combustible 2   3946 non-null   float64
 53  Combustible 3   3946 non-null   float64
 54  Combustible 4   3946 non-null   float64
 55  Tipo_marchas 0  3946 non-null   float64
 56  Tipo_marchas 1  3946 non-null   float64
 57  Mano 0          3946 non-null   float64
 58  Mano 1          3946 non-null   float64
 59  Mano 2          3946 non-null   float64
 60  Mano 3          3946 non-null   float64
dtypes: float64(60), int64(1)
memory usage: 1.8 MB
```

Como vemos, ahora hay 61 variables en vez de 13 (14 menos el Descuento) y no hay valores nulos.

Por último, normalizamos los datos de las variables cuantitativas usando `MinMaxScaler`. Como siempre, entrenamos con el archivo con los posibles valores de la variable y luego aplicamos la misma transformación a los datos training y test. Un ejemplo del código usado para normalizar es:

```
scalerKilometros = MinMaxScaler()
scalerKilometros.fit(kilometros["Kilometros"].to_numpy().reshape(-1, 1))
aux = scalerKilometros.transform(p2_datos["Kilometros"].to_numpy().reshape(-1, 1))
aux = pd.DataFrame(aux)
p2_datos["Kilometros"] = aux[0]
```

Una vez hecho esto, separamos los datos de las etiquetas, aplicamos oversampling y ajustamos los hiperparámetros de `MLP`. Para esste algoritmo, el parámetro que estimamos es `hidden_layer_sizes`.

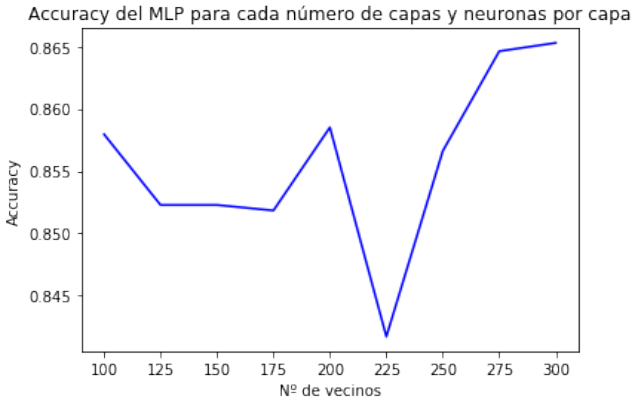


Figura 5: Selección de `hidden_layer_sizes`

La *accuracy* del modelo en el conjunto training es 0.8652947719688543. Tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.79637.

3.5. p3_05

En este intento usé el mismo preprocesamiento de datos que en el anterior, pero el modelo que entrené fue `K-nn`. Ajustamos el parámetro `n_neighbors`, obteniendo que el valor óptimo es 1.

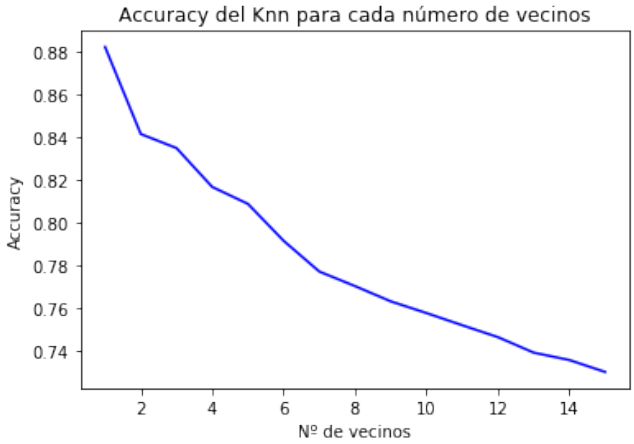


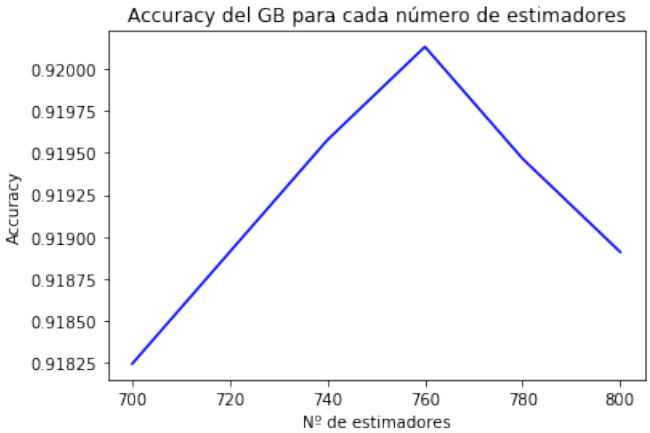
Figura 6: Selección de `n_neighbors`

La *accuracy* del modelo en el conjunto training es 0.8818687430478309, superior a la obtenida con `MLP`. Sin embargo, tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.68162, el peor resultado obtenido.

3.6. p3_06

Con el mismo preprocesamiento usado en `p3_04` entrenamos el algoritmo `Gradient Boosting`. El primer parámetro que seleccionamos es `n_stimators`. Al igual que con `Random Forest` hacemos primero una visión global del comportamiento del algoritmo dependiendo de este parámetro y luego hacemos ejecuciones con valores menos espaciados entorno al máximo encontrado. A la vista de los resultados, escogemos el valor 760 para este parámetro.

(a) Selección de `n_stimators`



(b) Selección de `n_stimators`

Buscamos también el valor óptimo para el parámetro `min_samples_leaf` y observamos que obtenemos los valores máximos de *accuracy* en 1 y en 8. Elegí el valor 8 porque al exigir mayor número de muestras por nodo, evitas nodos con muy pocas muestras y así se puede reducir el sobreajuste.

Figura 7: Selección de `min_samples_leaf`

Por último intentamos ajustar el valor de `learning_rate`, aunque comprobamos que el valor por defecto, 0.1, es el óptimo:

Figura 8: Selección de `min_samples_leaf`

La *accuracy* del modelo en el conjunto training es 0.9203559510567297. Tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.79982, la mejor hasta el momento.

3.7. p3_07

Sigo usando el preprocesamiento de `p3_04`. En este intento uso `StackingClassifier` para combinar los mejores algoritmos entrenados en intentos anteriores, concretamente:

- Gradient Boosting con `n_estimators = 760`, `min_sample_leaf = 8` y `learning_rate = 0.1`.
- MLP con `hidden_layer_sizes=(300,300)`, `activation='tanh'`, `max_iter=800` y `early_stopping=True`
- Random Forest con `n_estimators=230` y `criterion = 'entropy'`.

El clasificador usado para combinar estos tres es `LogisticRegression`, con `max_iter=400`.

La *accuracy* del modelo en el conjunto training es 0.9330367074527253. Tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.80241, la mejor hasta el momento.

3.8. p3_08

En esta prueba intentamos añadir al preprocesado anterior la detección y eliminación de outliers usando `LocalOutlierFactor` con el parámetro `n_neighbors=2`.

La *accuracy* del modelo en el conjunto training es 0.9348057882711348, superior a la obtenida sin aplicar este preprocesado. Sin embargo, tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.79376, que es inferior a la obtenida en el intento anterior. También se consiguió un mejor resultado en `p3_04` y `p3_06`, por lo que decidí no volver a usar `LocalOutlierFactor` en el preprocesado.

3.9. p3_09

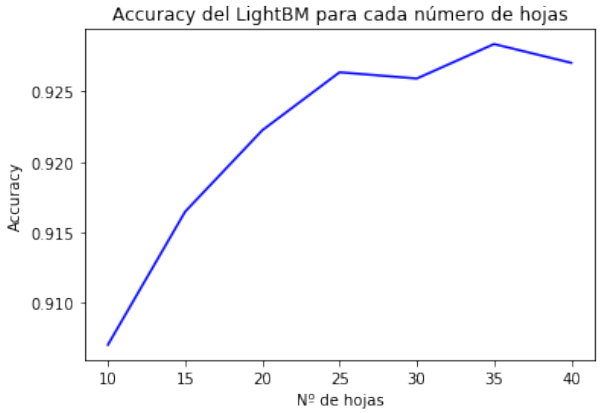
Usando nuevamente el preprocesamiento de `p3_04`, entrené el modelo `LGBMClassifier`.

Primero ajusté el parámetro `boosting_type`. La *accuracy* obtenida con los diferentes valores del parámetro se muestran en la siguiente tabla:

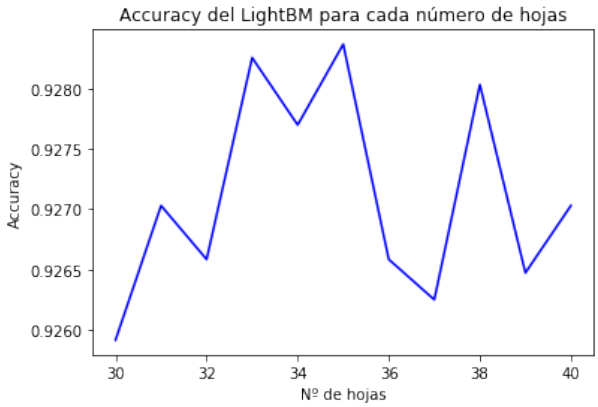
`boosting_type`	*Accuracy*
'gbdt'	0.926807563959556
'dart'	0.9123470522803114
'goss'	0.9270300333704116

En vista de estos resultados, el valor del parámetro usado es 'goss' (Gradient-based One-Side Sampling).

El siguiente parámetro a seleccionar fue `num_leaves`, el número máximo de hojas de cada clasificador base. El procedimiento a seguir es el mismo usado para escoger `n_estimators` en algoritmos anteriores. Como podemos comprobar en estas gráficas, el valor óptimo del parámetro es 35.



(a) Selección de `n_stimators`



(b) Selección de `n_stimators`

También ajusté la profundidad máxima de los estimadores base, es decir, el valor del parámetro `max_depth`.

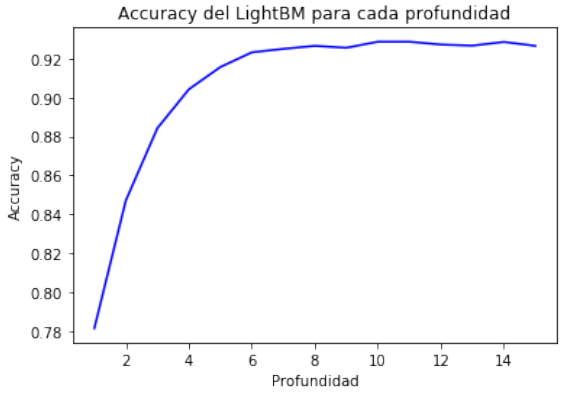
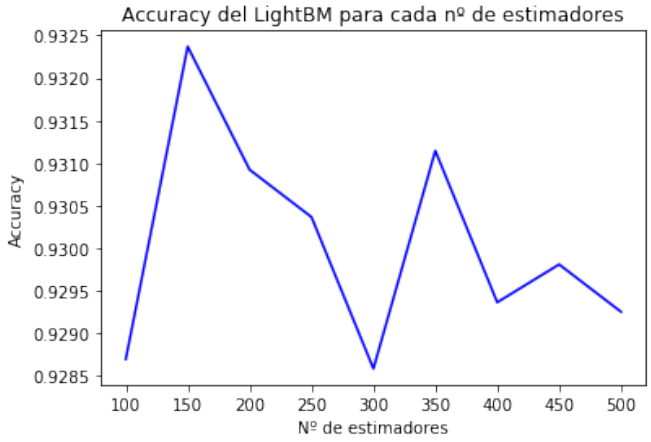


Figura 9: Selección de `max_depth`

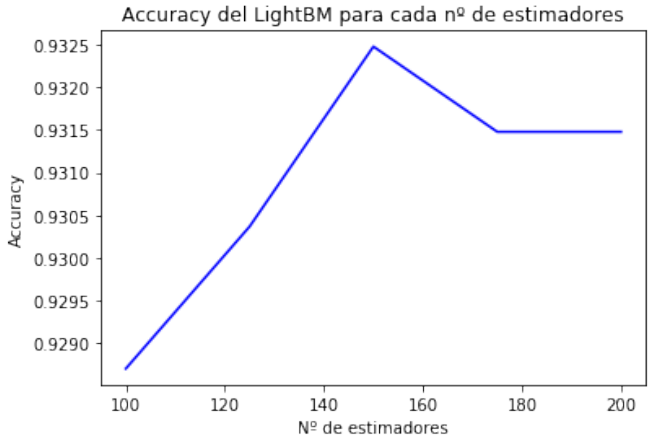
Como en la gráfica apenas se aprecia donde alcanza el máximo, mostraré en la siguiente tabla los valores de *accuracy* obtenidos.

`max_depth`	*Accuracy*
1	0.7813125695216907
2	0.8470522803114571
3	0.8842046718576195
4	0.9042269187986653
5	0.9156840934371525
6	0.9232480533926586
7	0.925027808676307
8	0.9265850945494994
9	0.9255839822024472
10	0.928698553948832
11	0.928698553948832
12	0.9272525027808676
13	0.9266963292547274
14	0.928587319243604
15	0.9265850945494994

Vemos que el máximo se alcanza en `max_depth = 10`, así que ese es el valor que asignaremos al parámetro. El siguiente parámetro que ajusté es `n_estimators`. Los resultados que obtuve se muestran a continuación en las siguientes gráficas:



(a) Selección de `n_stimators`



(b) Selección de `n_stimators`

Escogemos `n_stimators=150`. El último parámetro que ajustamos es `learning_rate`, donde comprobamos que el valor por defecto, 0.1, es el óptimo.

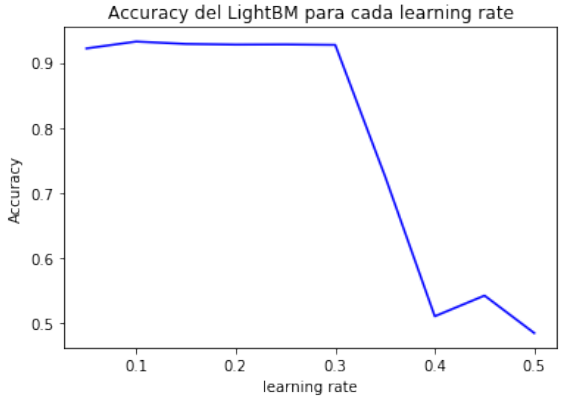


Figura 10: Selección de `learning_rate`

Una vez ajustado el modelo, entrenamos el `StackingClassifier` con los tres modelos anteriores más el `LGBMClasifier` ajustado con los parámetros mencionados anteriormente.

La *accuracy* del modelo en el conjunto training es 0.9339265850945495. Tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.80845, superando así todos los resultados anteriores.

3.10. p3_10

La diferencia de esta prueba con la anterior es la eliminación del algoritmo `Random Forest` de la lista de algoritmos usados en el `StackingClasifier`, dado que era el algoritmo con peor desempeño de los cuatro considerados en `p3_09`.

La *accuracy* del modelo en el conjunto training es 0.9300333704115685, que es menor que la obtenida en la prueba anterior. Sin embargo, tras aplicar el mismo preporcesado al conjunto test, obtenemos que la *accuracy* del conjunto test es 0.82830, la mejor hasta el momento.