

Inteligencia de Negocio: Práctica 3

Competición de Kaggle

David Cabezas Berrido

Grupo 2: Viernes

dxabezas@correo.ugr.es

4 de enero de 2021





















#	Δ pub	Team Name	Notebook	Team Members	Score ?	Entries	Last
1	—	JuanHeliosGarcía			0.83002	15	3d
2	—	PATRICIA CORDOBA 77145053			0.82830	13	3d
3	—	José Alberto García 26513007X			0.82484	43	1d
4	—	Alvaro de Rada 49108766V			0.81622	14	13h
5	—	DAVID CABEZAS 20079906			0.81622	34	2d
6	—	OctavioTorres			0.81622	23	12h
7	—	AlejandroAlonso75577394S			0.81363	10	1d
8	—	Mikhail Raudin 531101855			0.80845	11	12h
9	—	Javier Rodríguez 78306251Z			0.80759	12	13h
10	—	David Martin 75931868J			0.80586	28	21h
11	—	JuanCarlosGonQu			0.79982	54	20h
12	—	Pedro Jiménez 76592485R			0.79810	35	16h
13	—	Ilias_Amar_Ceuta			0.79723	15	15h
14	—	Jose Antonio Martín 77561280J			0.79551	14	16h
15	—	Alberto_Postigo_Ceuta			0.79206	21	2d
16	—	Laura Delgado 20608068E			0.79119	18	15h
17	—	Jose Maria Sánchez Guerrero ...			0.79119	19	14h
18	—	Sergio Fernández Fernández 0...			0.78688	12	13h
19	—	Alejandro Menor Molinero 1317...			0.77911	4	10d
20	—	Antonio Jesús Ruiz 53911182x			0.77825	8	13h

Figura 1: Leaderboard definitiva de la competición.

Índice

1. Pruebas realizadas	4
2. Introducción	7
3. Preprocesados	7
4. Estimación de hiperparámetros	8
4.1. Sobreajuste con oversampling y validación cruzada	8
5. Modelos	10
5.1. Random Forest	10
5.2. MLP	10
5.3. SVM	10
5.4. KNN	11
5.5. Boosting	11
5.6. Stacking	12
6. Otras pruebas fallidas	14
6.1. Experimentos con la columna descuento	14
6.2. Naive-Bayes	15
6.3. Clustering	15
7. Webgrafía	16

1. Pruebas realizadas

Intento	Fecha-hora	Posición	Validación	Test	Preprocesado	Modelo
1	19/12/2020 14:35:08	5	0.8185	0.7420	Preprocesado 1	RandomForest por defecto
2	20/12/2020 10:51:19	4-5	0.8900	0.7580	Preprocesado 2	RandomForest por defecto
3	20/12/2020 12:34:18	4-5	0.9040	0.7498	Preprocesado 2 eliminando de train los ejemplos que inicialmente tenían descuento	RandomForest con 350 estimadores de profundidad máxima 20
4	20/12/2020 17:05:16	3-4	0.9179	0.7645	Preprocesado 3	MLPClassifier con capas ocultas de tamaño (200,200)
5	21/12/2020 09:47:11	3-4	0.8750	0.6877	Preprocesado 3	2-NN con distancia Manhattan y pesos inversamente proporcionales a la distancia
6	21/12/2020 13:30:36	3	0.9143	0.7739	Preprocesado 3	C-SVM con C=65 y kernel RBF
7	21/12/2020 15:32:20	3	0.9144	0.7645	Preprocesado 3	RandomForest por defecto
8	22/12/2020 09:49:44	2	0.9143	0.8059	Preprocesado 3	GradientBoosting con 500 estimadores
9	22/12/2020 09:50:03	2	0.9304	0.8016	Preprocesado 3	Stacking: - RandomForest por defecto - MLPClassifier con capas ocultas (200,200) - C-SVM con C=65 y kernel RBF - GradientBoosting con 500 estimadores
10	22/12/2020 14:40:39	2	0.9312	0.7636	Preprocesado 3	AdaBoost con 500 árboles de profundidad 12 y learning rate de 1.1
11	23/12/2020 10:24:20	2	0.9163	0.7990	Preprocesado 3 sustituyendo los valores perdidos por 0 en la columna descuento en lugar de eliminar la columna	GradientBoosting con 500 estimadores
12	23/12/2020 12:12:38	2		0.7998	Preprocesado 3	Moda (predicción más frecuente) de los intentos 4, 6, 7, 8, 9, 10 y 11
13	24/12/2020 19:54:10	2	0.9205	0.7886	Preprocesado 3	GradientBoosting con 500 árboles de profundidad 6, learning rate de 1.175 y submuestras del 70 %
14	25/12/2020 09:59:17	4	0.8535	0.7239	Preprocesado 3 + PCA con 0.95 de varianza explicada	GradientBoosting con 500 estimadores
15	25/12/2020 10:25:21	4	0.9159	0.8007	Preprocesado 3 tras corregir error en LabelEncoder	GradientBoosting con 500 estimadores

16	25/12/2020 11:33:44	4		0.8093	Preprocesado 3	Stacking de cuatro GradientBoosting con número de estimadores 450, 500, 550 y 600 respectivamente; tasas de aprendizaje 0.14, 0.12, 0.1, 0.08 respectivamente; todos con submuestras del 90 %
17	26/12/2020 10:58:03	4		0.8085	Preprocesado 3	Stacking anterior con la opción passthrough
18	26/12/2020 11:12:21	4	0.9268	0.7886	Preprocesado 3	HistGradientBoosting por defecto
19	26/12/2020 11:31:13	4		0.8024	Preprocesado 3	Stacking de tres GradientBoosting con 500, 550 y 600 estimadores respectivamente; tasas de aprendizaje 0.12, 0.1 y 0.08 respectivamente; todos con submuestras del 90 %. También tres HistGradientBoosting con 100, 150 y 200 iteraciones máximas respectivamente
20	27/12/2020 11:00:35	4		0.8016	Preprocesado 3	Stacking de cuatro GradientBoosting con número de estimadores 450, 500, 550 y 600 respectivamente; tasas de aprendizaje 0.14, 0.12, 0.1, 0.08 respectivamente; todos con submuestras del 90 %. También dos HistGradientBoosting con 100 y 200 iteraciones máximas respectivamente
21	27/12/2020 13:54:7	4	(0.8392)	0.7886	Preprocesado 3	GradientBoosting con 550 árboles de profundidad 2, tasa de aprendizaje de 0.15 y submuestras del 90 %
22	27/12/2020 14:30:41	4	(0.8462)	0.7790	Preprocesado 3	LightGBM con 125 árboles de profundidad máxima 8 y 27 nodos hoja como máximo; tasa de aprendizaje del 0.08
23	28/12/2020 09:21:46	4	0.9290	0.7808	Preprocesado 3	LightGBM con 200 árboles con profundiad máxima 14
24	28/12/2020 09:22:13	4	0.9291	0.7843	Preprocesado 3	LightGBM con 125 árboles con 29 nodos hoja como máximo; tasa de aprendizaje del 0.11
25	28/12/2020 09:16:14	4	0.9266	0.7817	Preprocesado 3	LightGBM por defecto
26	29/12/2020 10:42:25	4	(0.8151) 0.8990	0.7964	Preprocesado 3	MLP con early stopping
27	29/12/2020 10:52:45	4	(0.7920) 0.9135	0.778	Preprocesado 3	SVM con C=40
28	29/12/2020 13:27:51	4	(0.8352) 0.9207	0.8016	Preprocesado 3	XGBoost con 200 árboles de profundidad 3
29	30/12/2020 09:45:53	4	(0.8370) 0.9262	0.7929	Preprocesado 3	HistGradientBoosting con 75 iteraciones máximas

30	30/12/2020 09:43:11	4	0.9300	0.7774	Preprocesado 3	HistGradientBoosting con 200 iteraciones máximas, tasa de aprendizaje del 0.08 y árboles con 29 nodos hoja como máximo
31	30/12/2020 10:08:35	4	0.9304	0.8110	Preprocesado 3	Stacking de GradientBoosting con 500 estimadores, MLP con early stopping, XGBoost con 200 árboles de profundidad 3 y HistGradientBoosting con 75 iteraciones máximas
32	31/12/2020 10:14:38	5	0.9268	0.8162	Preprocesado 3	Stacking de GradientBoosting con 500 estimadores, MLP con early stopping y XGBoost con 200 árboles de profundidad 3
33	31/12/2020 11:29:18	5	0.9225	0.8067	Preprocesado 3	Stacking de GradientBoosting con 500 estimadores y y XGBoost con 200 árboles de profundidad 3

Tabla 1: Pruebas realizadas

Nota: Los valores entre paréntesis en la columna de validación no corresponden a validación cruzada, sino a un conjunto de validación que he separado para paliar un problema de sobreajuste. Ver Sección 4.1.

Nota 2: El intento n , corresponde al archivo `try n .csv`, pero en Kaggle corresponde al $n + 1$ debido a que subí un intento corrupto por un error en el formato de la salida.

2. Introducción

Abordamos en una competición de Kaggle un problema del mundo real como es la predicción del precio de un coche usado, o en este caso la clasificación en categorías por precio. Probamos varias técnicas de preprocesado y varias configuraciones de distintos modelos que hemos estudiado en la asignatura. Realizamos diversos experimentos e intentamos ir mejorando nuestro score sobre test hasta conseguir el nuestro mejor resultado posible. Se trata claramente de un problema de aprendizaje supervisado (clasificación), y aplicaremos las técnicas y modelos propios de este tipo de tarea.

3. Preprocesados

El número de datos que presentaban valores perdidos en alguna de las columnas era bastante bajo, del orden de pocos cientos entre los cerca de 4800 datos, y no hay valores perdidos en los datos de test; es por ello que desechamos las instancias con valores perdidos en alguna de las columnas. Una excepción es la columna Descuento, en la que la gran mayoría de las instancias carecen de valor, por esta razón tomamos la decisión de desechar la columna. Hablaremos sobre ella más tarde.

Una vez tratado el problema de los valores perdidos, probamos distintas técnicas de preprocesado.

El **Preprocesado 1** es el mínimo para que os algoritmos puedan ejecutarse. Codificamos las variables categóricas con `LabelEncoder`, primero nos quedamos con la marca del coche, ya que hay cerca de 2000 modelos y son demasiados para que los algoritmos puedan aprovechar la información sólo con los algo más de 4000 datos de entrenamiento de los que disponemos. Convertimos Consumo, Motor_CC y Potencia a numérica: por ejemplo, el string `23.4 kmpl` se convierte en el flotante `23.4`. También codificamos la mano como numérica (del 1 al 4).

Con este procesamiento sólo hemos probado el intento 1, seguidamente intentamos mejorarlo.

Las clases están desbalanceadas, hay una (3) con casi la mitad de las instancias de entrenamiento y otras con cerca del 5% de las instancias. Para balancearlas he probado dos técnicas que se nos explicaron en el seminario sobre balanceo. La primera ha sido undersampling, y la deseché por obtener resultados bastante peores con Random Forest por defecto que el preprocesado 1. La segunda ha sido oversampling con `SMOTE`, que corresponde al **Preprocesado 2**. Esta técnica si ha supuesto una mejora significativa y he decidido mantenerla. Con este preprocesamiento, sólo he probado los intentos 2 y 3, éste último con una modificación que comentaremos en la Sección 6.1.

El **Preprocesado 3** aplica técnicas que acostumbran a beneficiar a modelos como KNN, SVM y redes neuronales. Estas técnicas son la binarización de características nominales y la estandarización de los datos (reescalarlos y desplazarlos para que tengan media 0 y varianza 1). He mantenido este preprocesamiento por el resto de experimentos, añadiendo pequeñas variaciones en el intento 11 (relativa a la columna Descuento) y en el intento 14, en el que he combinado este preprocesamiento con PCA para el 95% de variabilidad explicada, pero resultados peores.

En el intento 15, he corregido un error en el código que enumeraba incorrectamente los valores de la columna Mano (el 1 debe corresponder a primera mano, el 2 a segunda, el 3 a tercera y el 4 a cuarta o más). También he aprovechado para eliminar la única instancia de entrenamiento correspondiente a un coche eléctrico, ya que el conjunto de test no presenta instancias correspondientes a este tipo de coches. Todos los intentos posteriores se realizan con la nueva versión corregida de este preprocesamiento, aunque no parece que el error tuviese poca relevancia (de hecho el intento 15 obtiene peor score en test que el 8, correspondiente al mismo modelo sin el arreglo en los datos).

4. Estimación de hiperparámetros

Para ajustar los hiperparámetros de los modelos, recurrimos a la función `GridSearchCV`, que realiza validación cruzada (usamos 3, 4 o 5 folds, según lo rápido que se ejecuten los modelos) para cada combinación en una cuadrícula de valores para los hiperparámetros. Por ejemplo, el siguiente código realiza la búsqueda para los hiperparámetros de KNN.

```
param_grid={'n_neighbors':[1,2,3,5,7], 'weights':['uniform','distance'],
            'metric':['euclidean','manhattan']}
searcher = GridSearchCV(model, param_grid, n_jobs=4, verbose=15, cv=4)
search = searcher.fit(train, label)
```

En este caso, prueba $5 \cdot 2 \cdot 2 = 20$ combinaciones, y realiza 4 fits y validaciones para cada una. Podemos obtener los resultados como un DataFrame y ordenarlos por score medio.

param_metric	param_n_neighbors	param_weights	mean_test_score	rank_test_score
manhattan	2	distance	0.861040	1
manhattan	1	distance	0.860931	2
manhattan	1	uniform	0.860931	2
euclidean	2	distance	0.853917	4
euclidean	1	uniform	0.853808	5
euclidean	1	distance	0.853808	5
manhattan	3	distance	0.841206	7
manhattan	5	distance	0.836275	8
euclidean	3	distance	0.831671	9
manhattan	7	distance	0.830686	10

Figura 2: Score medio en CV para distintas configuraciones de KNN.

En ocasiones hemos realizado este proceso varias veces, detallando la cuadrícula o ampliando el rango de valores de los parámetros cuyos óptimos se encontraban en los extremos del rango.

4.1. Sobreajuste con oversampling y validación cruzada

Durante el desarrollo de la práctica, me he percatado de un problema que provocaba sobreajuste en los modelos. Cuando generamos datos sintéticos con `SMOTE` para el oversampling, obtenemos datos muy similares a los datos originales. Por tanto, en el conjunto de datos de entrenamiento hay instancias muy similares entre sí. Cuando hacemos validación cruzada los algoritmos no entrenan con los mismos datos con los que validan, pero si con datos muy similares, lo que los lleva a sobreajustar. Esto provoca que las combinaciones de hiperparámetros que obtienen mejores scores a la hora de validar sean las que presentan bajo sesgo y alta varianza, por lo que no tienen que tener un desempeño mejor sobre el conjunto de test. Según la teoría de aprendizaje, la validación cruzada debería obtener una estimación pesimista de la capacidad de generalización real del modelo. Con scores de validación por encima de 0.9 y scores en test que apenas alcanzan los 0.8, podemos sospechar que este problema está provocando sobreajuste en los modelos y minando su desempeño sobre test en una medida bastante significativa.

Para arreglar esto, separamos un conjunto de validación antes de hacer oversampling, y evaluamos cada configuración de cada algoritmo sobre este conjunto, de modo que los datos de validación no son tan similares a los de entrenamiento y obtenemos validaciones más realistas (rondando 0.8-0.84 de accuracy) y modelos con menos varianza. Las configuraciones de los modelos cambian en gran medida. GradientBoosting, que era mi mejor intento en ese

momento, no consigue superar su desempeño sobre test (los intentos 8 y 16 utilizan árboles sin límite de profundidad, con este nuevo conjunto de validación ajustamos los parámetros para el intento 21, que utiliza árboles con profundidad 2, pero obtiene peor score en test). En cambio, los modelos SVM y MLP encuentran configuraciones con mayor sesgo y menor varianza que logran superar a las anteriores configuraciones. En el caso de SVM, pasamos de $C = 65$ (intento 6) a $C = 40$ (intento 27), a menor C , mayor regularización, y conseguimos una ligera mejora sobre el conjunto de test. Para MLP, obtenemos una configuración más simple en la estructura de la red, de capas ocultas con (200,200) en el intento 4 a (100) en el intento 26, además de la incorporación de Early Stopping para regularizar, se consigue una mejora bastante significativa sobre el conjunto de test. Introducimos este último modelo en el Stacking correspondiente a nuestro mejor intento (el 32).

5. Modelos

Durante el desarrollo de la práctica probamos una amplia gama de modelos de los que hemos estudiado en la asignatura.

5.1. Random Forest

Se trata de un modelo bastante robusto y adecuado para clasificación multietiqueta. Trabaja relativamente bien con variables categóricas, y no requiere que las variables presenten escalas similares, por lo que no necesita un preprocesamiento tan complejo como otros modelos. Es por ello que es el primer modelo que probamos para tener una idea de los resultados que caben esperar. Se ve bastante beneficiado por el balanceo de las clases (oversampling), y curiosamente también por la binarización de características nominales (a pesar de trabajar bien con categóricas) en el intento 7. No le hemos dedicado mucho más trabajo a este modelo y no obtenemos buenos resultados con el ajuste de hiperparámetros, probablemente por el sobreajuste que hemos comentado antes. Los hiperparámetros que ajustamos son el número de estimadores y la profundidad máxima de los mismos.

Para el resto de modelos, mantendremos la binarización de características nominales y la estandarización de los datos, ya que modelos como KNN, MLP y SVM requieren de estos preprocesados para funcionar correctamente.

5.2. MLP

Los parámetros que ajustamos son la estructura de capas de neuronas ocultas y el uso de Early Stopping para regularizar.

Obtiene resultados relativamente buenos en el intento 4 (los mejores hasta el momento), sin Early Stopping y con una estructura compleja, (200,200). Mejora mucho en el intento 26, cuando ajustamos parámetros con el conjunto de validación y concluimos que es conveniente usar una estructura más simple, (100), y regularizar con Early Stopping.

5.3. SVM

Los parámetros que ajustamos son la regularización C y el kernel usado. Rápidamente nos percatamos de que el kernel más efectivo es el de Funciones de Base Radial (RBF), y tras varias búsquedas en grid, elegimos el valor 65 para C .

param_C	param_kernel	mean_test_score	rank_test_score	param_C	mean_test_score	rank_test_score	param_C	mean_test_score	rank_test_score
2.5	rbf	0.895014	1	15	0.910795	1	65	0.914301	1
2.25	rbf	0.894247	2	14	0.909370	2	60	0.914082	2
2	rbf	0.893370	3	13	0.908493	3	70	0.913973	3
1.75	rbf	0.893151	4	12	0.907945	4	75	0.913753	4
1.5	rbf	0.891616	5	11	0.907507	5	35	0.913425	5
1.25	rbf	0.889644	6	10	0.906740	6	55	0.913425	5
1	rbf	0.882630	7	9	0.905315	7	50	0.913315	7
0.75	rbf	0.875616	8	8	0.904658	8	25	0.913096	8
0.5	rbf	0.863452	9	7	0.903671	9	40	0.913096	8
				5	0.902247	10	45	0.912986	10
				6	0.902027	11	30	0.912767	11

(a) En la primera búsqueda nos decantamos por el núcleo RBF.

(b) Los valores altos de C consiguen mejor score en CV.

(c) Acabamos eligiendo $C = 65$.

Figura 3: Búsquedas en cuadrícula de la configuración de SVM con CV.

Esta configuración consiguen en el intento 6 nuestra mejor puntuación hasta el momento. Pero tras descubrir que la validación cruzada es propensa a sobreajustar, realizamos una segunda búsqueda para el parámetro C y elegimos $C = 40$.

param_C	mean_test_score	rank_test_score
35	0.913425	1
50	0.913315	2
25	0.913096	3
40	0.913096	3
45	0.912986	5
30	0.912767	6
20	0.912438	7
15	0.910795	8

Figura 4: Búsqueda en cuadrícula de la configuración de SVM con el conjunto de validación.

Con esta configuración, consigue mejorar su score en el intento 27, pero queda lejos de otros modelos como GradientBoosting.

5.4. KNN

Sólo hemos probado un intento con este modelo, el 5, y los resultados han sido bastante malos tanto en CV como en test. Los parámetros que hemos ajustado son la métrica (euclídea, Manhattan o máximo), el número de vecinos y los pesos (uniformes o inversamente proporcionales a la distancia). En la Figura 2 vemos la última búsqueda en cuadrícula para este modelo y la configuración definitiva con la que realizamos el intento.

5.5. Boosting

Boosting ha sido la técnica más efectiva entre las que he probado. Es por ello que probamos diferentes variantes e implementaciones. Al contrario que en el resto de modelos, algunas de las implementaciones ([Lightgbm](#) y [XGBoost](#)) no son de *sklearn*.

El primer algoritmo que intentamos es GradientBoosting, ajustamos el learning rate (obtenemos 0.1) y el número de estimadores (obtenemos 500) y superamos en el intento 8 el mejor score que habíamos obtenido hasta el momento. El éxito de este modelo nos lleva a considerar otras implementaciones de esta técnica.

Configuramos varios parámetros (número de estimadores, learning rate y profundidad máxima de los estimadores) de AdaBoost (que utiliza una función de pérdida exponencial) para el intento 10, pero conseguimos resultados bastante peores, quizá debido al sobreajuste.

Probamos una configuración más compleja de GradientBoosting para el intento 13, añadiendo a la cuadrícula los parámetros subsample (proporción de la muestra con la que entrena cada estimador simple) y profundidad máxima de los estimadores simples. Pero empeoramos los resultados sobre test. Igual ocurre en el intento 21, donde utilizamos el conjunto de validación para ajustar los parámetros.

También probamos diversas configuraciones de LightGBM, una variante de GradientBoost mucho más rápida, usamos validación cruzada (intentos 23 y 24) y validación simple (intento 22) para ajustar los parámetros. Pero no conseguimos que este modelo iguale el desempeño de GradientBoosting. También probamos la implementación de *sklearn* de esta técnica, HistGradBoost, con configuraciones obtenidas por ambos sistemas de validación (intentos 29, validación; y 30, CV). Igualmente, este modelo se queda por debajo. Ajustamos el learning rate y el número de estimadores, así como la profundidad máxima y el número máximo de nodos hoja de los mismos.

Finalmente, configurando XGBoost con el conjunto de validación, obtenemos un resultado a la altura de GradientBoosting (intento 28).

				n	lr	leaves	d	acc
iter	leaf	lr	acc	125	0.08	27	8	0.846231
75	31	0.10	0.837186	100	0.08	27	8	0.839196
75	33	0.12	0.834171	75	0.10	31	8	0.837186
100	31	0.10	0.833166	100	0.10	27	8	0.836181
100	29	0.10	0.833166	100	0.12	31	8	0.835176
75	27	0.12	0.832161	75	0.08	31	8	0.835176
75	33	0.08	0.831156	125	0.10	27	8	0.835176
100	27	0.12	0.831156	75	0.12	35	8	0.834171
100	27	0.08	0.830151	75	0.08	27	8	0.833166

(a) Configuración de HistGradient-Boosting para el intento 29.

(b) Configuración de LightGBM para el intento 22.

param_learning_rate	param_n_estimators	param_num_leaves	param_max_depth	mean_test_score	rank_test_score
0.1	200	31	14	0.929205	1
0.11	125	29	24	0.929096	2
0.11	125	29	-1	0.929096	2
0.1	150	31	14	0.929096	4
0.1	200	31	24	0.928658	5
0.1	200	31	-1	0.928658	5
0.1	200	29	24	0.928658	5
0.1	200	29	-1	0.928658	5
0.11	200	29	24	0.928438	9

(c) Configuración de LightGBM para el intento 23.

Figura 5: Búsquedas en cuadrícula para configurar distintos algoritmos de boosting.

5.6. Stacking

Stacking es una técnica que combina distintos clasificadores para mejorar el desempeño que tienen por separado, necesitamos varios clasificadores que ya presenten un desempeño decente. Stacking se ve mejorada cuanto mayor sea el número de clasificadores, su calidad y su variedad. Por lo que tenemos que encontrar un compromiso entre cantidad, calidad y variedad de los estimadores para sacarle el máximo partido.

Este modelo entrena los estimadores que apila a la vez que un estimador final (Regresión Logística) sobre los resultados, y utiliza validación cruzada para evaluar a qué estimadores debe de darles más peso. Su ejecución es muy lenta, por lo que en algunos experimentos omitimos la validación cruzada.

Nuestro primer intento de stacking es el 9, que combina el Random Forest por defecto, el MLP con la configuración del intento 4, el SVM con la configuración del intento 6 y el GradientBoosting del intento 8. No consigue superar al GradientBoosting, por lo que alguno o varios del resto de estimadores está minando su eficacia.

Probamos en el intento 16 distintos (cuatro) GradientBoosting con número de estimadores y learning rates diferentes, y fijamos nuestro récord personal. En el siguiente intento permitimos (`passthrough=True`) que el clasificador final

también aprenda sobre la muestra además de sobre los resultados, pero empeoreamos ligeramente el desempeño.

En los intentos 19 y 20 añadimos algunos HistGradientBoosting con diferentes configuraciones, pero también empeoramos el desempeño.

Finalmente en el intento 31, ya tenemos una batería más amplia y variada de modelos con calidad decente. Probamos Stacking con tres algoritmos basados en Boosting (GradientBoosting del intento 8, XGBoost del 28 y HistGradientBoosting del 29) y el nuevo MLP (intento 26). Conseguimos mejorar la mejor puntuación hasta el momento.

Eliminando el que peor desempeño tenía, HistGradientBoosting, conseguimos en el intento 32 nuestra puntuación definitiva. Finalmente, comprobamos en el intento 33 que eliminar el segundo con peor desempeño (MLP) empeora los resultados.

6. Otras pruebas fallidas

Describo otras estrategias que he intentado pero desechado por obtener scores bastante bajos en validación.

6.1. Experimentos con la columna descuento

Al no tener en cuenta el descuento, es lógico pensar que los coches con valores altos en este campo pueden confundir a los algoritmos, ya que podrían estar en una categoría más baja de la que realmente estarían si no tuviesen descuento. Del mismo modo, es posible que fracasemos al clasificar datos de test porque un elevado descuento los sitúe en una categoría más baja que la que le correspondería.

Pruebo dos ideas en esta línea. La primera en el intento 3, en la que elimino los ejemplos de train que presentan algún descuento. Esto evita que los algoritmos sean entrenados erróneamente, pero sigue existiendo el problema en el test. No logro ninguna mejora con esta idea. Además, estoy suponiendo que las instancias con valor NA en descuento se trata de coches sin descuento, lo cual podría no ocurrir.

Bajo esta suposición pruebo una segunda idea en esta línea, que corresponde al intento 14. Sustituyo por 0 los valores perdidos en la columna Descuento en lugar de eliminar la columna, tanto en train como en test. Tampoco logro ninguna mejora con esta idea.

Finalmente, desecho cualquier esperanza de obtener información con el siguiente experimento: Primero separo del conjunto de train un conjunto de validación correspondiente a todos los ejemplos que cuentan con el campo descuento. Entreno mi mejor algoritmo hasta el momento (GradientBoosting con 500 estimadores) con los datos restantes y realizo predicción sobre el conjunto de validación. Obtengo las siguiente tablas:

```
In [36]: diferencias.groupby('Dif').count()
```

Out[36]:

	Cat	Pred	Desc
Dif			
-2.0	1	1	1
-1.0	52	52	52
0.0	469	469	469
1.0	47	47	47
2.0	1	1	1
4.0	1	1	1

(a) Fallos y aciertos en las predicciones.

```
In [35]: diferencias.groupby('Dif').mean()
```

Out[35]:

	Cat	Pred	Desc
Dif			
-2.0	3.000000	1.000000	8.120000
-1.0	3.826923	2.826923	12.164231
0.0	3.840085	3.840085	20.769446
1.0	2.808511	3.808511	16.253617
2.0	3.000000	5.000000	52.460000
4.0	1.000000	5.000000	1.360000

(b) Descuentos en las predicciones acertadas y falladas.

Figura 6: Relación entre los descuentos y la exactitud de las predicciones.

En la tabla de la izquierda, observamos que los fallos se producen aproximadamente en la misma medida por exceso (Dif > 0, predecimos una clase más cara a la correspondiente) que por defecto (Dif < 0, predecimos una clase más barata a la correspondiente). En la tabla de la derecha observamos que los valores de descuento no son mayores en las instancias que fallamos, sino en las que acertamos. Tampoco hay una diferencia significativa entre el descuento medio de las instancias que fallamos por exceso o defecto. Por tanto, no parece que altos descuentos lleven a mayor error en las predicciones.

6.2. Naive-Bayes

El modelo Naive-Bayes requiere un preprocesado algo distinto, ya que asume independencia en las variables y la binarización de variables categóricas impide esta propiedad. Intentamos combinar el preprocesado 2 con PCA para quedarnos con menos variables y además incorreladas. Probando distintos valores para la proporción de variabilidad explicada, no logramos pasar de accuracy 0.6 en validación cruzada, por lo que no aplicamos este modelo a ningún intento.

6.3. Clustering

La validación (y validación cruzada) acostumbra a obtener una estimación pesimista del desempeño de los modelos, debido a que entrenan con menos datos. Claramente, en este caso no la tenemos. En el caso de validación cruzada es lógico pensar que el problema es el que comentamos en la Sección 4.1 con el oversampling, pero en validación simple no se da este problema, así que podría ocurrir que existiese cierto sesgo entre los datos de test y de entrenamiento, normalmente este problema limita nuestras posibilidades y no tiene solución.

En un intento, quizá desesperado, de atajar este problema. Probamos técnicas de aprendizaje no supervisado. Utilizamos un preprocesado como el 3 (binarización de características nominales y reescalado de las variables) pero sin balanceo de clases. Mezclamos los datos de train y test, y aplicamos K-means y Ward con $K = 5$ con la esperanza de que el tamaño de los clusters concuerde con las instancias de train que deberían pertenecer a esos clusters (para que sea factible que se hubiese formado un cluster por clase), pero no logramos nada.

```
Counter(class_labels)
Counter({1.0: 203, 2.0: 502, 3.0: 1825, 4.0: 834, 5.0: 637})

Counter(cluster_labels) # K-means
Counter({0: 2217, 1: 987, 2: 1937, 3: 16, 4: 3})

Counter(cluster_labels) # Ward
Counter({0: 3717, 1: 840, 2: 3, 3: 598, 4: 2})
```

Figura 7: Tamaño de los clusters comparado con el número de instancias de train en cada clase.

Como observamos, los clusters 4 de las segmentaciones no puede contener a ninguna clase.

Probamos a introducir cinco nuevas dimensiones, a los datos de test les ponemos el 0 en todas, mientras que a los datos de entrenamiento les asignamos un valor $r > 0$ en una (relativa a su clase) lo suficientemente grande para forzar que los datos de la misma clase acaben en el mismo cluster. Los datos de test no tendrían predisposición a formar parte de ningún cluster respecto a estas cinco dimensiones. Por tanto, quizá las instancias de test se vean “arrastradas” al cluster correspondiente a su clase según sus valores en el resto de variables (las originales).

```
Counter(class_labels)
Counter({1.0: 203, 2.0: 502, 3.0: 1825, 4.0: 834, 5.0: 637})

Counter(cluster_labels) # K-means
Counter({0: 1825, 1: 1362, 2: 834, 3: 502, 4: 637})

Counter(cluster_labels) # Ward
Counter({0: 1362, 1: 637, 2: 834, 3: 1825, 4: 502})
```

Figura 8: Tamaño de los clusters comparado con el número de instancias de train en cada clase, forzando a que cada clase esté contenida en un cluster.

Como observamos, las instancias de test se ven arrastradas todas al mismo cluster, el cluster 1 de K-means y el 0 de Ward. Luego no hemos obtenido ninguna información.

7. Webgrafía

Algoritmos de *sklearn*:

- Random Forest: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>
- SVM: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>
- Gaussian Naive-Bayes: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
- KNN <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
- MLP: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
- GradientBoosting: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html>
- HistGradientBoosting: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html>
- Stacking: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html>
- K-means: <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>
- Clustering aglomerativo (Ward): <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html>

Otros algoritmos:

- LightGBM: <https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html>
- XGBoost: https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBClassifier