

Práctica 3: Competición en Kaggle

Alejandro Alonso Membrilla






Grupo: viernes

Correo: aalonso99@correo.ugr.es

DNI: 7577394S

Posición final

La posición final obtenida por el alumno en la competición es la de 7º puesto, tal como puede verse en la siguiente captura del Leaderboard de Kaggle:

5	—	DAVID CABEZAS 20079906		0.81622	34	2d
6	—	OctavioTorres		0.81622	23	2h
7	—	AlejandroAlonso75577394S		0.81363	10	1d
8	—	Mikhail Raudin 531101855		0.80845	11	1h
9	—	Javier Rodríguez 78306251Z		0.80759	12	2h

Contents

1	Soluciones propuestas	4
1.1	Primera subida (cuaderno P3_1)	4
1.2	Segunda subida (cuaderno P3_1)	6
1.3	Tercera subida (cuaderno P3_2)	6
1.4	Cuarta subida (cuaderno P3_2)	7
1.5	Quinta subida (cuaderno P3_1_2)	8
1.6	Sexta subida (cuaderno P3_1_2)	8
1.7	Séptima subida (cuaderno P3_1_2)	9
1.8	Octava subida (cuaderno P3_1_2)	9
1.9	Novena subida (cuaderno P3_1_2)	10
1.10	Décima subida (cuaderno P3_1_2)	10

1 Soluciones propuestas

En la siguiente tabla vemos la información básica sobre las distintas soluciones subidas a Kaggle. A continuación incluiremos una subsección muy breve sobre cada una explicándola en mayor detalle:

SUBIDA	FECHA	POSICIÓN	SCORE (Kaggle)
1	28-12-2020-21:05	27	0.72648
2	28-12-2020-21:53	22	0.74978
3	29-12-2020-15:11	26	0.70405
4	29-12-2020-18:33	26	0.67471
5	29-12-2020-00:55	25	0.75496
6	30-12-2020-17:02	9	0.79810
7	31-12-2020-01:42	5	0.81190
8	31-12-2020-13:37	6	0.79551
9	31-12-2020-13:51	6	0.81363
10	01-01-2020-01:22	6	0.79723

El código utilizado para generar las distintas predicciones se encuentra en los cuadernos de Jupyter que se adjuntan con la práctica (se entregan 3). El cuaderno correspondiente a cada subida se indica en la descripción de dicho experimento. A su vez, las predicciones realizadas con cada cuaderno están indicadas al principio de cada uno.

1.1 Primera subida (cuaderno P3_1)

Para el primer experimento realizamos el preprocesado básico de los datos de entrenamiento:

1. Carga y visualización en una tabla de los datos.
2. Eliminación de la columna de Descuento, que contaba únicamente con 659 intancias no nulas.
3. Visualización de los distintos coches separados por categoría de precio en función de la ciudad (ver figura 1). Eliminación de la columna de Ciudades, basándonos en la poca variabilidad que esta variable introducía en el conjunto.
4. Eliminación de las filas restantes con valores perdidos.
5. Cambio de las columnas de *strings* de Consumo, Motor_CC y Potencia por sus correspondientes con valores numéricos (eliminando previamente la parte del texto correspondiente a las unidades).
6. Vemos que hay 1516 modelos de coche diferentes en las filas restantes. Suponemos que incluir la columna Nombre, tal como está, pudiera provocar cierto tipo de sobreaprendizaje (el modelo aprendería la categoría de un coche en función de su nombre), o tal vez ser totalmente ignorado. La solución implementada es sustituir el nombre de cada coche por su marca, quedándonos solo con la primera palabra del nombre. El resultado es una columna Marca con solamente 28 marcas distintas.

A continuación observamos si hay desbalanceo entre las distintas clases del problema. Tras la criba anterior, el número de instancias de cada clase es de 203 para la 1ª, 509 la 2ª, 1853 la 3ª, 848 la 4ª y 646 la 5ª. Hay un claro desbalanceo, especialmente entre la primera clase y la tercera, que tratamos de paliar mediante el procedimiento siguiente:

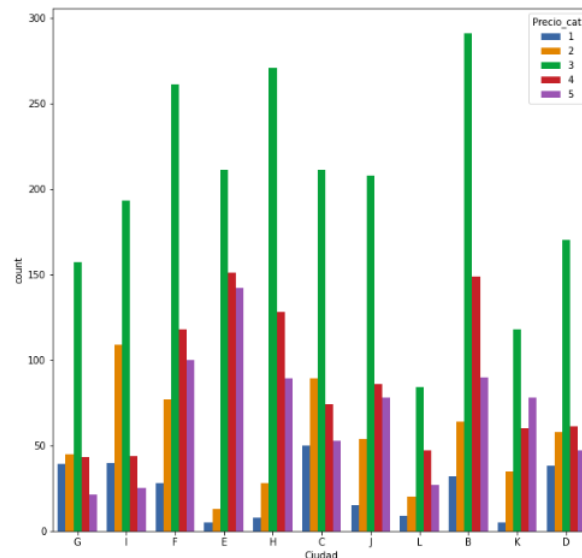


Figure 1: Comparación de precios de coches en distintas ciudades

1. Uso de SMOTENC como método de oversampling para lidiar con las clases categóricas.
2. Codificamos las clases categóricas con un LabelEncoder.
3. Aplicamos undersampling con Tomek's Links para eliminar el ruido inducido por el oversampling.
4. En este caso NO se normalizaron los datos (fue un error corregido en la siguiente subida).

Probamos un modelo de árbol con boosting de gradiente (mediante la biblioteca XGBoost), con 1000 estimadores, una profundidad máxima de 5, la función objetivo específica para clasificación multiclase (*softmax*) y el resto de parámetros por defecto.

Para este modelo y para los siguientes probados se ha elaborado un método llamado `train_test_nombreModelo` para crear y testear de forma más sencilla el modelo en cuestión. En este caso el método se ve así:

```
def train_test_gradientBoosting(X, y, n_estim=500, max_depth=5, lr=0.1, gamma=0,
min_child_weight=1, max_delta_step=0, subsample=0.8, colsample_bytree=1, booster='gbtree'):
    mean_acc = 0.0
    for train_idx, test_idx in kf.split(X):
        X_train, X_test = X[train_idx], X[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        gbClassifier = xgb.XGBClassifier(tree_method='gpu_hist', gpu_id=0,
            objective='multi:softmax', booster=booster, n_estimators=n_estim,
            max_depth=max_depth, learning_rate=lr, gamma=gamma,
            min_child_weight=min_child_weight, max_delta_step=max_delta_step,
            subsample=subsample, colsample_bytree=colsample_bytree,
            colsample_bylevel=1, colsample_bynode=1, reg_alpha=0, reg_lambda=1,
            scale_pos_weight=1, base_score=0.5,
            sampling_method='gradient_based', grow_policy='lossguide')

        gbClassifier.fit(X_train, y_train)
        y_predict = gbClassifier.predict(X_test)
        mean_acc += accuracy_score(y_test, y_predict)
```

```
mean_acc /= n_splits

return mean_acc
```

Esta función de prueba en particular divide el conjunto de entrenamiento para la validación cruzada, define el modelo a partir de una serie de parámetros por defecto y otros pasados como argumento de la función, entrena el modelo y valida los resultados. Como resultado se devuelve el promedio de las precisiones obtenidas.

Aplicando validación cruzada obtenemos resultados muy altos, del orden del 94% de precisión, debido a un problema que se repetirá en los primeros experimentos: hemos aplicado oversampling sobre todo el conjunto de entrenamiento sin separar previamente los conjuntos de validación. El resultado es que el modelo aprende sobre dicho conjunto de validación a partir de la información replicada en el oversampling.

Los resultados en Kaggle, como puede verse en la tabla de puntuaciones, no acompañaron a los obtenidos en entrenamiento.

1.2 Segunda subida (cuaderno P3_1)

Repetimos el experimento anterior, esta vez normalizando el conjunto de datos al intervalo $[0, 1]$ y reduciendo el número de estimadores usados por XGBoost. También se probó a aplicar codificación One Hot de los datos categóricos, pero se decidieron no incluir en la solución subida a Kaggle por el empeoramiento de resultados en validación.

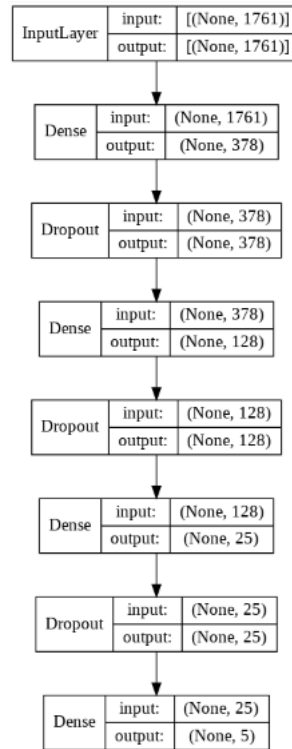
La normalización sube la precisión en torno a un 2%.

1.3 Tercera subida (cuaderno P3_2)

Para este experimento se implementó un perceptrón multicapa en Keras. Previamente se hicieron algunos cambios en el preprocesamiento:

1. No se elimina la columna de ciudades.
2. No se cambia la columna de nombres por la de marcas.
3. La codificación pasa a ser la de One Hot (que había dado peores resultados para XGBoost).
4. Eliminamos la fase de undersampling con Tomek's Links después del oversampling.
5. Al igual que en la prueba anterior (y en todas las posteriores) normalizamos el conjunto de datos de entrenamiento.

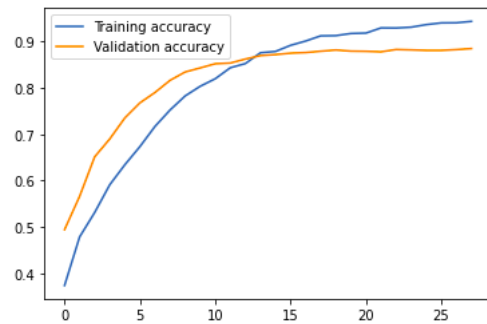
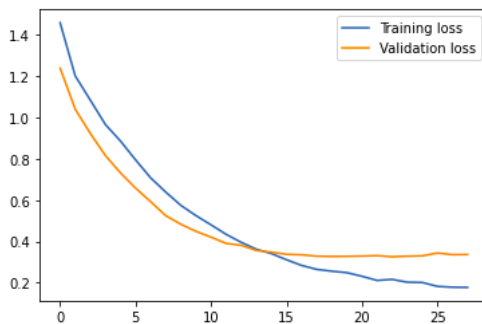
La red neuronal implementada es un MLP con 3 capas ocultas seguidas de *dropouts*. Su arquitectura es la siguiente:



donde el input de entrada es del tamaño de una fila de la tabla codificada como vectores OneHot. Tras una serie de experimentos se decidió probar con una red con 378, 128 y 25 capas ocultas (como se ve en el diagrama anterior). Los resultados aparentaban ser muy buenos (del orden de un 97% de precisión en validación) y, de nuevo, esto no se vio reflejado en Kaggle debido al error de planteamiento inicial al planificar la validación cruzada.

1.4 Cuarta subida (cuaderno P3_2)

Al obtener resultados muy por debajo de lo esperado, el alumno se planteó la posibilidad de que, al entrenar la red con todo el conjunto de entrenamiento, se produjese algún tipo de sobreaprendizaje. La cuarta predicción siguió el mismo proceso que la primera, solo que esta vez la red se entrenó con un 50% elegido aleatoriamente del conjunto de entrenamiento. Las curvas con la evolución de la función de pérdida y la precisión (de entrenamiento y validación) en el entrenamiento de esta red neuronal son las siguientes:



La precisión media en validación rondaba el 88%, pero en test solo tuvo un 67% de precisión.

1.5 Quinta subida (cuaderno P3_1_2)

En el cuaderno P3_1_2 se reimplementa el preprocesado prácticamente desde cero, en forma de una clase llamada `Preprocessor`, para evitar aplicar oversampling sobre los conjuntos de validación del *cross-validation*. Esto logra que las precisiones obtenidas en validación sean mucho más parecidas a las devueltas por Kaggle, lo que nos permita hacer un ajuste de hiperparámetros que pueda ser eficaz en la fase de test.

La clase `Preprocessor` se instancia una vez por cada conjunto de datos, permite aplicar la codificación, normalización y resampling del conjunto, y guarda los parámetros de dichas transformaciones para poder repetirlas (salvo el resampling) sobre los conjuntos de test.

Las funciones del tipo *train_test* empleadas (ejemplo en la primera subida) ahora no necesitan dividir el conjunto en distintos *folds*, puesto que eso ya se ha hecho previamente para no tener que repetir el oversample del conjunto de entrenamiento cada vez que se quiera hacer *cross-validation*. En su lugar, iteran sobre las listas de subconjuntos construidas previamente. El ejemplo original de XGBoost queda de la siguiente forma:

```
def train_test_gradientBoosting(X_train, y_train, X_test, y_test, n_estim=500,
max_depth=5, lr=0.1, gamma=0, min_child_weight=1, max_delta_step=0,
subsample=0.8, colsample_bytree=1, booster='gbtree'):

    mean_acc = 0.0
    for x_tr, x_te, y_tr, y_te in zip(X_train, X_test, y_train, y_test):
        gbClassifier = xgb.XGBClassifier(tree_method='gpu_hist', gpu_id=0,
            objective='multi:softmax', booster=booster, n_estimators=n_estim,
            max_depth=max_depth, learning_rate=lr, gamma=gamma,
            min_child_weight=min_child_weight, max_delta_step=max_delta_step,
            subsample=subsample, colsample_bytree=colsample_bytree,
            colsample_bylevel=1, colsample_bynode=1, reg_alpha=0, reg_lambda=1,
            scale_pos_weight=1, base_score=0.5,
            sampling_method='gradient_based', grow_policy='lossguide')
        gbClassifier.fit(x_tr, y_tr)
        y_predict = gbClassifier.predict(x_te)
        mean_acc += accuracy_score(y_te, y_predict)
    mean_acc /= n_splits

    return mean_acc
```

donde `X_train`, `X_test`, `y_train`, `y_test` son listas con los distintos *folds* y sus etiquetas asociadas a cada instancia. El resampling se aplica solamente sobre `X_train` e `y_train`.

Aplicamos un *grid-search* para buscar parámetros de XGBoost que den precisiones lo más altas posibles. Esto se obtiene para una profundidad máxima de 4 niveles, un ratio de aprendizaje de 0.1, un umbral gamma de 0 y unos ratios de submuestreo de 0.7 a nivel global y 0.7 por cada estimador. La precisión obtenida en validación ronda en torno a 0.82.

1.6 Sexta subida (cuaderno P3_1_2)

Para la sexta subida realizamos un par de cambios en el preprocesado:

1. No se elimina la columna de ciudades (esto ya se había probado para las redes neuronales, no para el XGBoost).
2. La codificación (LabelEncoding) pasa a hacerse antes del oversampling.
3. Para el oversampling dejamos de usar SMOTENC y consideramos las variables categóricas como numéricas. Tras una sencilla experimentación manual, se decide usar BorderlineSMOTE como método de oversampling.

A modo de anotación, tras corregir el error de validación de los experimentos anteriores, se probó a implementar en Keras una red convolucional (para vectores de entrada 1D) que puede verse en el apartado **Redes Neuronales** del cuaderno **P3_1_2**. Después de algunas pruebas y experimentación con sus capas e hiperparámetros, los resultados no fueron buenos (difícilmente llegaban al 70% de precisión de validación) así que se optó por no subir ninguna predicción de este modelo a Kaggle.

Los resultados obtenidos por XGBoost tras este cambio aumentan de 0.82 a 0.83 en validación. En la fase de test este pequeño cambio produce un aumento muy significativo de la precisión (hasta casi un 80%).

1.7 Séptima subida (cuaderno P3_1_2)

Se realiza un stacking con MLP, XGBoost y Random Forest. El MLP se implementa mediante los `MLPClassifiers` de Scikit-Learn para poder integrarlo en un stack más fácilmente. Previamente, se hace un *grid-search* para optimizar los hiperparámetros de esta red neuronal con 3 capas ocultas, obteniendo una red resultante con capas de 500, 150 y 150 nodos y un parámetro de regularización alfa de 0.01.

Como estimador final del `StackingClassifier` se utiliza un `GradientBoostingClassifier` de Scikit-Learn siguiendo este ejemplo, en lugar de la regresión logística usada por defecto.

Se realizan algunos cambios menores en el preprocesado:

1. No se elimina la columna de descuentos. Ahora, si en dicha columna se encuentra un valor perdido este se sustituye por un 0 considerando que, si no se ha tenido en cuenta el descuento, en una situación real lo más natural sería que simplemente no hubiese descuento (lo que equivale a un descuento de 0).
2. Se sustituye la eliminación de filas con valores perdidos por una imputación simple: la mediana en caso de las variables numéricas y la moda en el caso de las categóricas.

Ambos cambios se conservan para los experimentos posteriores.

Los resultados en validación del XGBoost mejoran muy ligeramente hasta el 83.36%. En Kaggle la precisión de test aumenta notablemente, superando el 81%.

1.8 Octava subida (cuaderno P3_1_2)

Añadimos un SVM al stack, tras un ajuste de hiperparámetros del mismo. El clasificador SVM encontrado que aporta mejores resultados por sí solo es un SVC con un parámetro de regularización $C = 1.5$ y un kernel polinomial de grado 9.

Los resultados en validación permanecen iguales pero bajan en Kaggle.

1.9 Novena subida (cuaderno P3_1_2)

Eliminación del SVC y el Random Forest del StackingClassifier, al ser los modelos que, independientemente, obtenían peores resultados.

La puntuación en Kaggle sube a un 81.363% (la más alta obtenida).

1.10 Décima subida (cuaderno P3_1_2)

Se aumentan el número de estimadores usados por el XGBoost en el stack (de 500 a 1000) y el del GradientBoostedClassifier usado como estimador final del stack (de 25 a 40). El resultado baja a menos del 80%, tal vez porque el estimador final tienda, al aumentar este parámetro, a dar más prioridad a los valores predichos por el XGBoost (que era el clasificador con mejores resultados) y a ignorar el de los demás, aunque no podemos confirmarlo sin un estudio más profundo.

En cualquier caso, este último experimento no logra mejorar la máxima precisión obtenida de 81.363%.