Inteligencia de Negocio (2019-2020)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

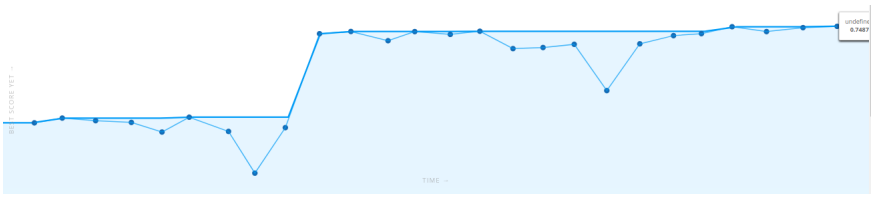
Competición en Driven Data

Grupo de prácticas: martes
Email: simondelosbros@correo.ugr.es

Simón López Vico

7 de enero de 2020

Submissions



SUBMISSIONS

Score	Submitted by	Timestamp
0.6854	Simon_Lopez_UGR	2019-12-17 09:02:15 UTC
0.6885	Simon_Lopez_UGR	2019-12-23 09:34:13 UTC
0.6868	Simon_Lopez_UGR	2019-12-23 09:49:40 UTC
0.6857	Simon_Lopez_UGR	2019-12-23 12:01:08 UTC
0.6794	Simon_Lopez_UGR	2019-12-24 08:16:10 UTC
0.6891	Simon_Lopez_UGR	2019-12-24 09:25:38 UTC
0.6798	Simon_Lopez_UGR	2019-12-24 10:16:03 UTC
0.6524	Simon_Lopez_UGR	2019-12-25 03:55:46 UTC
0.6823	Simon_Lopez_UGR	2019-12-25 10:18:36 UTC
0.7439	Simon_Lopez_UGR	2019-12-25 17:30:57 UTC
0.7454	Simon_Lopez_UGR	2019-12-26 08:45:35 UTC
0.7393	Simon_Lopez_UGR	2019-12-26 10:35:17 UTC
0.7453	Simon_Lopez_UGR	2019-12-26 19:18:43 UTC
0.7435	Simon_Lopez_UGR	2019-12-27 08:21:14 UTC
0.7455	Simon_Lopez_UGR	2019-12-27 10:32:56 UTC
0.7341	Simon_Lopez_UGR	2019-12-27 17:31:54 UTC
0.7348	Simon_Lopez_UGR	2019-12-28 10:19:55 UTC
0.7370	Simon_Lopez_UGR	2019-12-28 16:50:27 UTC
0.7066	Simon_Lopez_UGR	2019-12-28 18:56:44 UTC
0.7373	Simon_Lopez_UGR	2019-12-29 18:25:19 UTC
0.7427	Simon_Lopez_UGR	2019-12-30 10:41:47 UTC
0.7440	Simon_Lopez_UGR	2019-12-30 13:02:09 UTC
0.7485	Simon_Lopez_UGR	2019-12-30 19:02:44 UTC
0.7453	Simon_Lopez_UGR	2019-12-31 09:48:23 UTC
0.7479	Simon_Lopez_UGR	2019-12-31 12:11:55 UTC
0.7487	Simon_Lopez_UGR	2019-12-31 14:37:42 UTC

Índice

1. Introducción

En esta práctica se estudiará el uso de métodos avanzados para aprendizaje supervisado en clasificación sobre una competición real disponible en Driven Data llamada *Richter's Predictor: Modeling Earthquake Damage*. El problema está basado en la predicción del nivel de daños causado en los edificios por el terremoto de Gorkha de 2015 en Nepal, habiendo 3 grados de daño: bajo, medio y casi completamente destruido.

El conjunto de datos consiste principalmente en información sobre la estructura de los edificios y su propiedad legal. Cada fila del dataset representará un edificio específico en la región que fue afectada por el terremoto de Gorkha. Alguna de sus 38 características son:

- `count_floors_pre_eq`: número de pisos en el edificio antes del terremoto.
- `age`: edad del edificio en años.
- `has_superstructure_[...]`: determina si la superestructura está hecha del material que aparezca en `[...]`. Hay 11 variables de este tipo, y cada edificio puede tener una superestructura de varios materiales.
- `count_families`: número de familias que viven en el edificio.
- `has_secondary_use`, `has_secondary_use_[...]`: la primera de ellas indica si el edificio tiene un uso secundario, mientras que con la segunda se especifica cuál es dicho uso en `[...]`. Hay 10 variables de este segundo tipo.

2. Inicio: prueba de distintos algoritmos

Para empezar, notemos que sobre todos los algoritmos ejecutados y enviados a la competición de Driven Data se han pasado las variables categóricas a numéricas. Se han utilizado dos métodos para este preprocesado: uno proporcionado por el profesor cuyo funcionamiento no daba buenos resultados y otro más simple que comentaremos más adelante. En la tabla de resultados podemos ver el aumento en el valor de *Test* al modificar el *Category to Number*.

Para evaluar los resultados de los algoritmos se ha utilizado validación cruzada de 5 particiones. Todos estos resultados se guardarán en el directorio `/code/results_train`, aunque alguno de ellos no se acaben mandando a la competición[1]. El código para aplicar *CV* será:

[1] En dicho directorio hay un fichero que informa de cuáles de los resultados han sido subidos a Driven Data.

```
def validacion_cruzada(modelo, X, y, cv):
    y_test_all = []

    for train, test in cv.split(X, y):
        t = time.time()
        modelo = modelo.fit(X[train],y[train])
        tiempo = time.time() - t
        y_pred = modelo.predict(X[test])
        print("F1 score (tst): {:.4f}, tiempo: {:6.2f} segundos"
        .format(f1_score(y[test],y_pred,average='micro') , tiempo))
        y_test_all = np.concatenate([y_test_all,y[test]])

    return modelo, y_test_all

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED)
algoritmo, result = validacion_cruzada(algoritmo,X,y,skf)
```

Antes de enviar ningún resultado a Driven Data, se han ejecutado diferentes algoritmos de clasificación con sus parámetros por defecto para determinar cuál de ellos puede llegar a dar mejores resultados y escoger uno sobre el que calcular las distintas predicciones para la competición. Sobre todos ellos se ha aplicado el *Category to Number* proporcionado por el profesor, y para los algoritmos basados en distancias se han normalizado los datos. Los resultados obtenidos al ejecutar los algoritmos con validación cruzada son:

	Logistic Reg	**Naive Bayes**	**RandomForest**	**LGBM**	**SGD**	**Ada Boost**	**MLP Classifier**
K=1	0.5691	0.4298	0.6972	0.7054	0.5690	0.6362	0.5676
K=2	0.5691	0.4307	0.6974	0.7095	0.5690	0.6386	0.5646
K=3	0.5692	0.4312	0.6994	0.7058	0.5690	0.6339	0.5671
K=4	0.5692	0.4290	0.7032	0.7095	0.5694	0.6435	0.5700
K=5	0.5692	0.4285	0.7011	0.7068	0.5690	0.6417	0.5690
Time	~3.5 seg	~0.18 seg	~4.05 seg	~9.1 seg	~2.7 seg	~14.7 seg	~167 seg

Podemos ver que los algoritmos con mejores resultados son los basados en árboles, y entre ellos tenemos las mejores predicciones para Random Forest y LGBM, por lo que a partir de ahora trabajaremos con estos dos algoritmos intentando ajustar sus parámetros óptimamente.

3. Características del conjuntos de datos

Nos encontramos ante un dataset con 260601 instancias y 39 atributos, el primero de ellos (`building_id`) desechable, pues toma valores aleatorios únicos por instancia. Por otra parte, nuestro conjunto de test contará con 86868 a predecir. Entre todas las características de los dos conjuntos no nos encontramos ningún valor perdido; esto lo podemos comprobar mediante el código.

```
print("Valores perdido en train:")
print(data_x.isnull().sum())
```

```
print("\n\nValores perdidos en test:")
print(data_x_tst.isnull().sum())
```

Así, se nos mostrará por pantalla el conjunto de atributos y el número de valores perdidos para cada uno de ellos (cero para todos).

Veamos ahora el número de instancias para cada etiqueta del dataset:

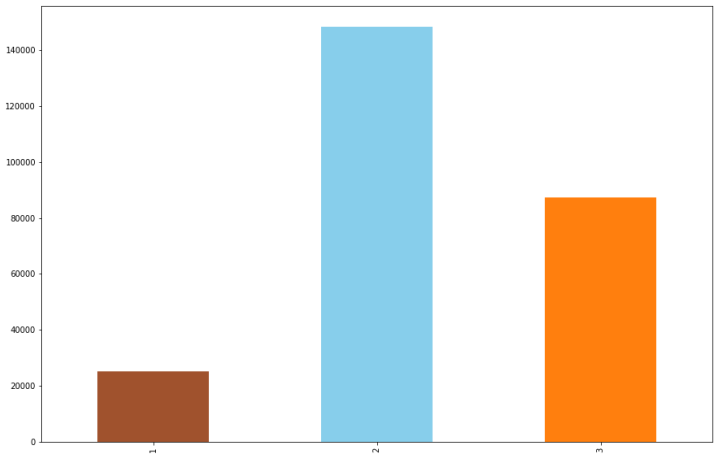


Figura 3.1: Cantidad de valores para *low damage*, *medium damage* y *almost complete destruction*.

Podemos ver que nuestro conjunto de datos está muy desbalanceado, habiendo muy pocos edificios que hayan acabado con pocos daños tras el terremoto. Concretamente contaremos con 25124 instancias para *low damage*, 148259 para *medium damage* y 87218 para *almost complete destruction*. Tendremos que tener esto en cuenta a la hora de predecir sobre el conjunto test.

Finalmente veamos la correlación entre las distintas variables utilizando un heatmap:

```
# Correlacion entre variables
fig,ax=plt.subplots(figsize=(20,20))
sns.heatmap(data_x.corr(), ax=ax,cmap="YlGnBu",cbar=False)

plt.savefig("./plots/correlation.png")
plt.clf()
```

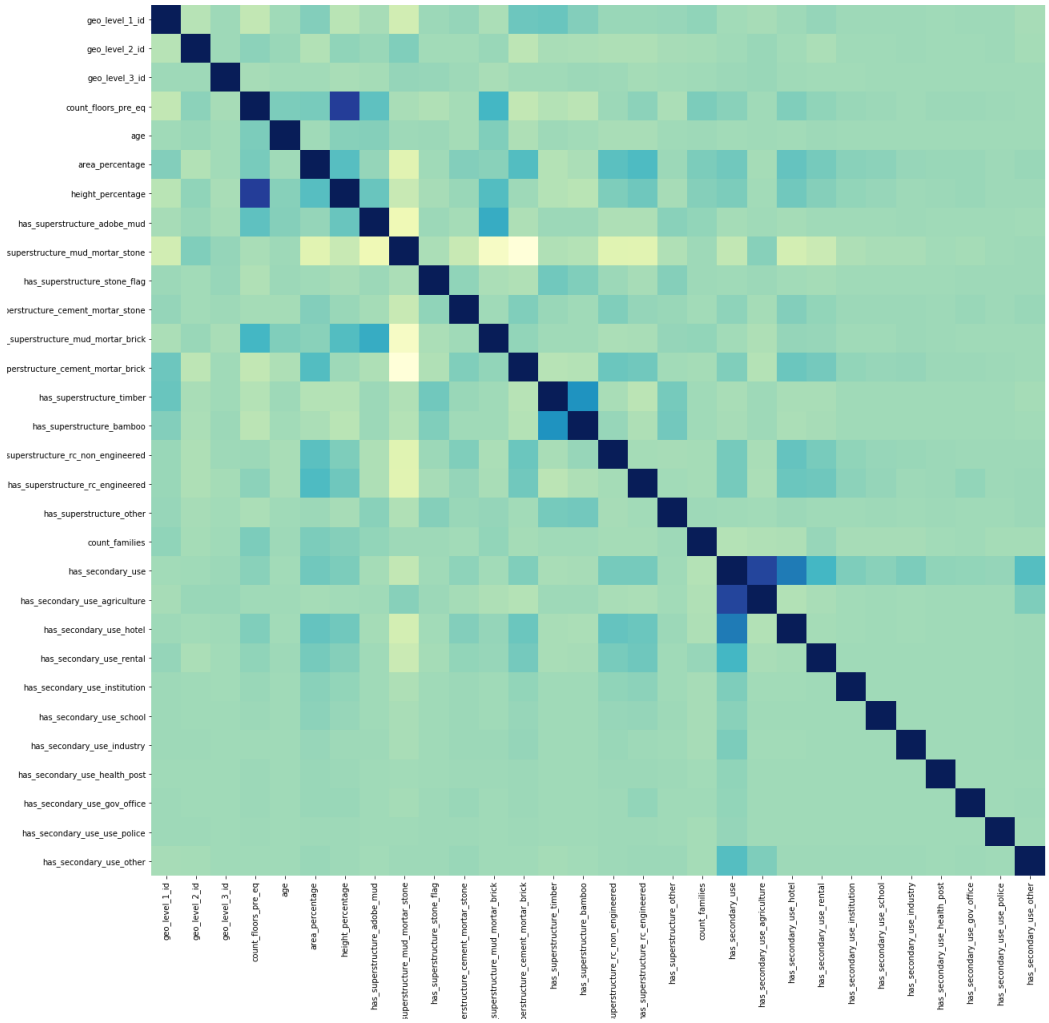


Figura 3.2: Correlación entre las distintas variables de nuestro dataset.

En dicha imagen los valores más oscuros representan una mayor correlación entre variables. En la zona de arriba a la izquierda podemos ver una valor alto de correlación (0.772) entre el número de pisos del edificio y la altura del edificio normalizada. Por otra parte, la correlación entre *has_secondary_use* y *has_secondary_use_agriculture* vale 0.739; los valores altos entre de correlación entre *has_secondary_use* y los distintos atributos del tipo *has_secondary_use_[...]* son debidos a que la primera de estas vale 1 cuando alguno de los segundos vale 1, y vale 0 cuando el edificio no tenga ningún uso secundario. Por tanto, la cantidad de veces que se de el valor 1 en *has_secondary_use* será igual a la suma de las veces que se de un 1 en las variables del segundo tipo. Así, cuantas más veces tomen los atributos del segundo tipo el valor 1, mayor será el valor de correlación con *has_secondary_use*.

4. Preprocesados aplicados

El preprocesado principal aplicado al dataset en cuestión es el encargado de pasar las variables categóricas a numéricas. Como hemos comentado anteriormente, se han utilizado dos algoritmos de *Category to Number*, uno proporcionado por el profesor y otro desarrollado por mi.

```
# ------------------------------------------
# ----- CATEGORICAS A NUMERICAS PROFESOR -----
# ------------------------------------------

from sklearn.preprocessing import LabelEncoder

#mascara para luego recuperar los NaN
mask = data_x.isnull()
#LabelEncoder no funciona con NaN, se asigna un valor no usado
data_x_tmp = data_x.fillna(9999)
#se convierten categoricas en numericas
data_x_tmp = data_x_tmp.astype(str).apply(LabelEncoder().fit_transform)
#se recuperan los NaN
data_x_nan = data_x_tmp.where(~mask, data_x)

#Igual para data_x_tst
#[...]

#guardamos los valores como arrays de numpy
X = data_x_nan.values
X_tst = data_x_tst_nan.values
y = np.ravel(data_y.values)
```

Podemos ver reflejado en la tabla de resultados que con este algoritmo no obtenemos buenos puestos en el ranking de Driven Data; además, consume algo de tiempo, por lo que se ha desarrollado uno más simple y eficiente que vemos a continuación:

```
# ------------------------------------------
# --------- CATEGORICAS A NUMERICAS ---------
# ------------------------------------------

cat_dict = {}

for col in data_x.columns:
  if(type(data_x[col][0]) is str):
    categories = data_x[col].value_counts().index
    add_dict = {}
    for i in range(0,categories.size):
      key = categories[i]
      value = ord(categories[i])
      add_dict[key] = value

    cat_dict[col] = add_dict

data_x.replace(cat_dict, inplace=True)
```

```
data_x_tst.replace(cat_dict, inplace=True)

X = data_x.values
X_tst = data_x_tst.values
y = np.ravel(data_y.values)
```

Con este fragmento de código, recorremos todos los atributos guardados en el dataframe `data_x` y vamos comprobando cuál de ellos es categórico. Cuando encontremos una característica categórica, cogeremos todos los valores que puede tomar dicha característica y los guardaremos en un diccionario junto a su valor numérico ASCII (a=97, b=98, c=99, etcétera). Además, cada uno de estos diccionarios se guardará en otro diccionario que contendrá el nombre del atributo categórico y su diccionario asociado. Finalmente, usaremos el método `replace` de Pandas para modificar todos los valores categóricos por su valor numérico ASCII y transformaremos estos nuevos dataframes en arrays de numpy.

Simplemente con este cambio de preprocesado, se aumentó el porcentaje de acierto en Driven Data de 0.6823 a 0.7439, utilizando el mismo algoritmo y los mismos parámetros.

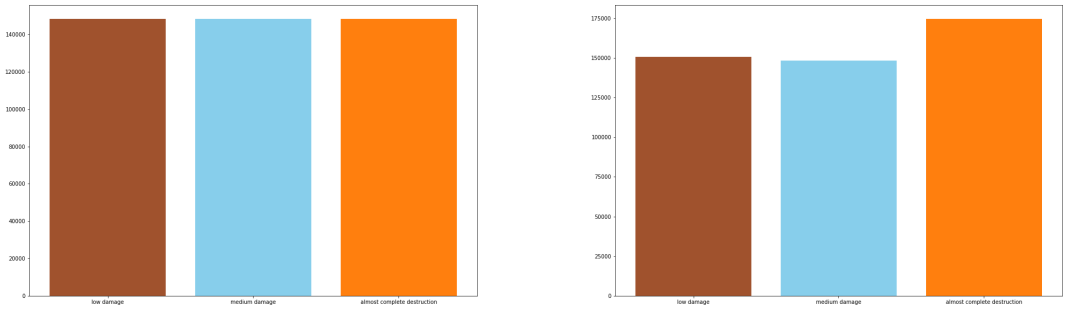
Por otra parte, para solucionar el desbalanceo hemos utilizado SMOTE, un algoritmos que se encargará de crear instancias *sintéticas* mediante diferentes técnicas, la más común de ellas KNN. Para aplicar este método se ha instalado el paquete `smote_variants`[2] que contiene muchas variantes de SMOTE, y la que hemos utilizado ha sido `polynom_fit_SMOTE` probando con sus distintas topologías, *mesh* y *star*. Para aplicar estos algoritmos utilizaremos el siguiente código:

```
import smote_variants as sv

#topology='mesh' o 'star'
oversampler = sv.MulticlassOversampling(
    sv.polynom_fit_SMOTE(topology='mesh',random_state=SEED)
    )

X,y=oversampler.sample(X,y)
```

Vemos a continuación como queda el dataset tras aplicar estos preprocesados.



(a) Conjunto de datos tras aplicarle `polynom_fit_SMOTE` con la topología *mesh* (izda) y *star*(dcha).

La primera topología utilizada, *mesh*, deja el dataset con exactamente el mismo número de elementos para cada clase, mientras que la topología *star* nos creará aproximadamente las misma instancias de *low damage* y *medium damage*, pero en la clase *almost complete destruction* creará más instancias que en el resto, dejando de nuevo el dataset desbalanceado. Aún así, utilizando la topología *star* obtendremos mejores resultados que con *mesh*, lo cual podemos comprobar en las últimas subidas a Driven Data (final de la tabla).

También se ha probado a igualar el número de elementos de cada clase al número de instancias de la clase intermedia, haciendo undersampling a *medium damage* y oversampling a *low damage*. Se ha utilizado el siguiente código:

```
#Juntamos train y test para obtener la misma
#normalizacion en los dos conjuntos
X_all = np.append(X,X_tst,axis=0)

from sklearn import preprocessing
X_all_normalized = preprocessing.normalize(X_all, norm='l2')

#Separamos de nuevo train y test
X_norm = X_all_normalized[:len(X)]
X_tst_norm = X_all_normalized[len(X):]

from imblearn.under_sampling import RandomUnderSampler
under = RandomUnderSampler(random_state=SEED, ratio={2:87218})
X_resampled, y_resampled = under.fit_resample(X_norm, y)

import smote_variants as sv
oversampler = sv.kmeans_SMOTE(proportion=1.0, n_neighbors=3,n_jobs=-1,
    random_state=SEED)
X,y=oversampler.sample(X_resampled,y_resampled)
```

Comenzamos normalizando los datos, pues para el oversampling vamos a utilizar SMOTE KNN, que está basado en distancias. Tras ello aplicamos un `RandomUndersampler` a *medium damage* hasta llegar a 87218 elementos, y con el resultado de esto hacemos

SMOTE KNN a las instancias de la clase *low damage*.

Este último preprocesado fue utilizado con LGBM, pero el resultado en Driven Data fue muy bajo en comparación con el resto ejecuciones del mismo algoritmo (0.7066).

5. Ajuste de parámetros y progreso desarrollado

Una vez vista la estructura del dataset y los distintos preprocesados a aplicar, veamos cómo hemos conseguido nuestro mejor resultado en Driven Data.

Para empezar, como vimos en la segunda sección (2), se ejecutaron varios algoritmos para comprobar cual podría ser más válido sobre nuestro dataset, determinando que las mejores opciones son Random Forest y Ligh GBM. Tras elegir estos algoritmos, se fueron realizando varias subidas a Driven Data; las primeras 9 subidas tienen una baja calidad, pues el método utilizado para pasar de variables categóricas a numéricas hacía que los resultados fueran bajos. En estos primeros intentos se probaron distintos parámetros para los dos algoritmos y diferentes preprocesados, hasta elegir como algoritmo final para la competición Light GBM.

Una vez determinado cuál es el algoritmo a utilizar para la predicción de nuestros datos, es el momento de aplicar el preprocesado y ajustar los parámetros.

5.1. Preprocesado

En esta sección comentaremos la variación de la exactitud en las predicciones en función del preprocesado aplicado. No tendremos en cuenta las 9 primeras subidas a Driven Data, pues utilizan el *Category to Number* que da malos resultados.

La aplicación de SMOTE sobre los datos para la ejecución del clasificador ha hecho mejorar mucho la predicción en el conjunto de entrenamiento, llegando hasta el 91 por ciento de acierto. Aún así, el balanceo de datos no nos está proporcionando un bueno estimador, pues los resultados en Driven Data son iguales o peores a los que se han subido sin aplicar técnicas de balanceo. Por tanto, en las distintas ejecuciones que se han realizado con SMOTE, simplemente se ha sobreentrenado nuestro modelo.

Por otra parte, el balanceo de datos igualando al número de elementos de la clase intermedia nos da un resultado similar al resto en el conjunto de entrenamiento, pero muy malo en función del acierto que se tiene con LGBM en el conjunto test, como hemos comentado anteriormente.

Un bueno preprocesado, que no se ha llegado a aplicar, hubiera sido eliminar la columna `has_secondary_use`, pues como comentamos en la sección tres (3) no aporta

ninguna información y bastaría con las variables `has_secondary_use_[...]`. También sería buena idea comprobar si mejoran los resultados al eliminar `count_floors_pre_eq` o `height_percentage`, variables que aportan un valor similar (altura del edificio).

5.2. Parámetros de LGBMClassifier

Para ajustar los parámetros del algoritmo de clasificación elegido se ha utilizado `GridSearchCV`; con este método, elegiremos todos los valores que queremos probar para cada parámetro elegido, y pasándole el algoritmo a ejecutar, los parámetros y el número de *folds* para la validación cruzada, se nos devolverá la mejor combinación de parámetros para nuestro algoritmo elegido optimizando el valor de `f1_micro`.

```
lgbm = lgb.LGBMClassifier(objective='regression_l1',n_jobs=-1)

params_lgbm = {
    'learning_rate':[i*0.01 for i in range(10,12)],
    'num_leaves':[i*5 for i in range(4,6)],
    'n_estimators':[i*500 for i in range(4,6)]
}

grid = GridSearchCV(lgbm, params_lgbm, cv=2, n_jobs=-1, verbose=1,
          scoring='f1_micro')
grid.fit(X,y)
print("Mejores parametros:")
print(grid.best_params_)
# Devuelve {'learning_rate': 0.1, 'n_estimators': 2500, 'num_leaves': 25}
```

Se han realizado varios Grid Search a lo largo de la competición y se han ido modificando algunos valores manualmente hasta dar con la mejor combinación de parámetros para LGBM. Veamos ahora la función de cada uno de esos parámetros:

- `objective`: especifica la tarea de aprendizaje y el objetivo de aprendizaje correspondiente para el clasificador. Comenzamos utilizando `regression_l1`, ya que era el que venía en el código de ejemplo, pero a lo largo del proyecto se modificó por `multiclassova` (*One-vs-All*).
- `num_leaves`: máximo número de hojas en los árboles utilizados para el ajuste.
- `learning_rate`: ritmo de aprendizaje, determina la velocidad con la que nuestro modelo reacciona ante los datos que están llegando o que está recibiendo.
- `n_estimators`: número de árboles a generar para el ajuste.
- `max_bin`: número máximo de *contenedores* donde los valores de las características serán almacenados durante la ejecución del algoritmo.

Tras ejecutar los distintos Grid Search descubrimos que a mayor número de estimadores, mayor porcentaje de acierto en la competición; además, el objetivo `multiclassova` es el que mejor funciona para nuestro ajuste. Así, los parámetros óptimos para LGBM serán:

```
          (objective='multiclassova', learning_rate=0.09,
0          n_estimators=3250, num_leaves=34, max_bin=400)
```

5.3. Bagging

Una vez determinado el mejor algoritmo a utilizar con sus parámetros óptimos, utilizaremos Bagging (*Bootstrap Aggregating*) para mejorar nuestro modelo. Esta técnica consiste en crear diferentes modelos usando muestras aleatorias con reemplazo y luego combinar o ensamblar los resultados. Así, en las cuatro últimas subidas a Driven Data (al final de la tabla) probamos a elegir los valores y el preprocesado que mejor funcione con Bagging + LGBM.

```
lgbm = lgb.LGBMClassifier([...])
0 bag = BaggingClassifier(base_estimator=lgbm,n_jobs=-1,
            n_estimators=15,random_state=SEED)
2 clf = bag
clf = clf.fit(X,y)
```

Tras ver que Bag + LGBM da muy buenos resultados para test (0.7485) probamos a aplicar SMOTE en dos ocasiones, uno con la topología *mesh* y otro con la topología *star*. De nuevo, vemos que los valores de acierto en el conjunto de entrenamiento mejoran pero el valor para test se queda por debajo del obtenido sin balanceo de las clases (aunque el obtenido con SMOTE star es muy buen resultado).

Por último, ya que disponía de una última subida a Driven Data antes de que cerrara el plazo de competición, probamos a ejecutar Bagging pero esta vez realizando el muestreo sin reemplazo (`bootstrap=False`) y aumentando el número de estimadores de LGBM a 3250. Así, obtenemos un valor final de predicción sobre el conjunto test de 0.7487, lo que se corresponde al puesto número 52 de la competición de Driven Data.

6. Tabla de resultados

Time	Preproc	Alg	Param	Train	Test	Pos
2019-12-17 09:02:15	Categóricas a numéricas Profesor	RF	n_estimators=20 max_features=None Min_samples_leaf=30	0.7553	0.6854	∼**360**
2019-12-23 09:34:13	Categóricas a numéricas Profesor	RF	n_estimators=125 max_features=None Max_depth = 20	0.8863	0.6885	*N.A.*
2019-12-23 09:49:40	Categóricas a numéricas Profesor	LGBM	objective='regression_l1' N_estimators=1000	0.7738	0.6868	*N.A.*
2019-12-23 12:01:08	Categóricas a numéricas Profesor	LGBM	objective='regression_l1' feature_fraction=0.5 n_estimators=1000 Num_leaves=50	0.7849	0.6857	*N.A.*
2019-12-24 08:16:10	Categóricas a numéricas Profesor + polynom fit SMOTE solo sobre la clase Pequeña (mal)	LGBM	objective='regression_l1' n_estimators=2000 Num_leaves=50	0.8868	0.6794	*N.A.*
2019-12-24 09:25:38	Categóricas a numéricas Profesor + polynom fit SMOTE solo sobre la clase Pequeña (mal)	RF	n_estimators=125 max_features=None Max_depth = 20	0.9124	0.6891	∼**300**
2019-12-24 10:16:03	Categóricas a numéricas Profesor + polynom fit SMOTE sobre todas las clases (bien)	RF	Igual al anterior	0.8379	0.6798	*N.A.*
2019-12-25 03:55:46	Categóricas a numéricas Profesor + polynom fit SMOTE (bien)	LGBM	objective='regression_l1' learning_rate= 0.12 n_estimators=1750 num_leaves=25 (GridSearchCV)	0.8522	0.6524	*N.A.*
2019-12-25 10:18:36	Categóricas a numéricas Profesor	LGBM	objective='regression_l1' learning_rate= 0.12 n_estimators=2750 num_leaves=20 (GridSearchCV)	0.8013	0.6823	*N.A.*
Ahora preprocesado funcional						
2019-12-25 17:30:57	Categóricas a numéricas (bueno)	LGBM	Igual al anterior	0.8003	0.7439	**119**
2019-12-26 08:45:35	Categóricas a numéricas	LGBM	objective='regression_l1' learning_rate= 0.1 n_estimators=2750 num_leaves=25	0.8045	0.7454	**99**

[1]N.A. = No Avanza

Time	Preproc	Alg	Param	Train	Test	Pos
2019-12-26 10:35:17	Categóricas a numéricas + polynom fit SMOTE	LGBM	Igual al anterior	0.8598	0.7393	*N.A.*
2019-12-26 19:18:43	Categóricas a numéricas	BAG + LGBM	Bagging: default LGBM: igual al anterior	0.8010	0.7453	*N.A.*
2019-12-27 08:21:14	Categóricas a numéricas	LGBM	objective='regression_l1' learning_rate= 0.09 n_estimators=2700 num_leaves=28 (GridSearchCV)	0.8049	0.7435	*N.A.*
2019-12-27 10:32:56	Categóricas a numéricas	BAG + LGBM	Bagging: n_estimators=20 LGBM: objecti- ve='regression_l1' learning_rate= 0.1 n_estimators=2750 num_leaves=25	0.8012	0.7455	*N.A.*
2019-12-27 17:31:54	Categóricas a numéricas + polynom fit SMOTE	RF	n_estimators=125 max_features=None Max_depth = 20	0.9128	0.7341	*N.A.*
2019-12-28 10:19:55	Categóricas a numéricas + polynom fit SMOTE	RF	n_estimators=500 max_features=None Max_depth = 20	0.9132	0.7348	*N.A.*
2019-12-28 16:50:27	Categóricas a numéricas	RF	Igual al anterior	0.8817	0.7370	*N.A.*
2019-12-28 18:56:44	Categóricas a numéricas + Undersampling a la clase grande + Kmeans SMOTE a la clase pequeña	LGBM	objective='regression_l1' learning_rate= 0.1 n_estimators=2750 num_leaves=25	0.8569	0.7066	*N.A.*
2019-12-29 18:25:19	Categóricas a numéricas	RF	n_estimators=1200 max_features=None Max_depth = 20	0.8819	0.7373	*N.A.*
2019-12-30 10:41:47	Categóricas a numéricas + polynom fit SMOTE	LGBM	objective='regression_l1' learning_rate= 0.1 n_estimators=2750 num_leaves=28 max_bin=400	0.8647	0.7427	*N.A.*
2019-12-30 13:02:09	Categóricas a numéricas + polynom fit SMOTE	LGBM	objective='regression_l1' learning_rate= 0.1 n_estimators=2750 num_leaves=34 max_bin=400	0.8736	0.7440	*N.A.*
2019-12-30 19:02:44	Categóricas a numéricas	BAG + LGBM	Bagging: n_estimators=15 LGBM: objecti- ve='multiclassova' learning_rate=0.09 n_estimators=3000 num_leaves=34 max_bin=400	0.8214	0.7485	**56**

Time	Preproc	Alg	Param	Train	Test	Pos
2019-12-31 09:48:23	Categóricas a numéricas + polynom fit SMOTE MESH	BAG + LGBM	Igual al anterior	0.8715	0.7453	*N.A.*
2019-12-31 12:11:55	Categóricas a numéricas + polynom fit SMOTE STAR	BAG + LGBM	Igual al anterior	0.9018	0.7479	*N.A.*
2019-12-31 14:37:42	Categóricas a numéricas	BAG + LGBM	Bagging: n_estimators=15 bootstrap=False LGBM: objecti- ve='multiclassova' learning_rate=0.09 n_estimators=3250 num_leaves=34 max_bin=400	0.8334	0.7487	**52**

Referencias

[1] Scikit Learn, Supervised Learning, https://scikit-learn.org/stable/supervised_learning.html

[2] SMOTE-variants, https://smote-variants.readthedocs.io/en/latest/

[3] Undersampling and oversampling imbalanced data, https://www.kaggle.com/residentmario/undersampling-and-oversampling-imbalanced-data

[4] Pandas: Python Data Analysis Library, https://pandas.pydata.org/