Inteligencia de Negocio
Práctica 3: Competición en DrivenData

Pablo Jesús Jiménez Ortiz
Grupo IN2 (Jueves)

Enero de 2020

SUBMISSIONS

Score	Submitted by	Timestamp
0.7452	Pablo_Jimenez_Ortiz_UGR	2019-12-31 20:55:03 UTC
0.7459	Pablo_Jimenez_Ortiz_UGR	2019-12-31 21:11:16 UTC
0.7458	Pablo_Jimenez_Ortiz_UGR	2019-12-31 21:37:56 UTC

Figure 1: Resultados plataforma Driven Data

Contents

1 Introducción

Con el objetivo de aproximarnos al mundo real de la Ciencia de Datos, en esta práctica debemos abordar un problema real dentro del entorno de la plataforma **Driven Data**, cuyo propósito es el de realizar competiciones de Ciencia de Datos bajo un enfoque social que hace interesantes muchas de las competiciones propuestas desde esta plataforma.

2 Análisis exploratorio de datos

Para una aplicación eficaz de las técnicas de aprendizaje, hemos de realizar un estudio previo de los datos de cara a conseguir una mejor comprensión de éstos y por tanto enfocar las estrategias de manera más adecuada.

En primer lugar, es necesario conocer la distribución de las clases, que en este problema están desbalanceadas y por tanto hacen que nos encontremos ante un problema de clasificación desbalanceado. Para la visualización de las siguientes gráficas se usaron los paquetes ***matplotlib.pyplot*** y ***seaborn***.

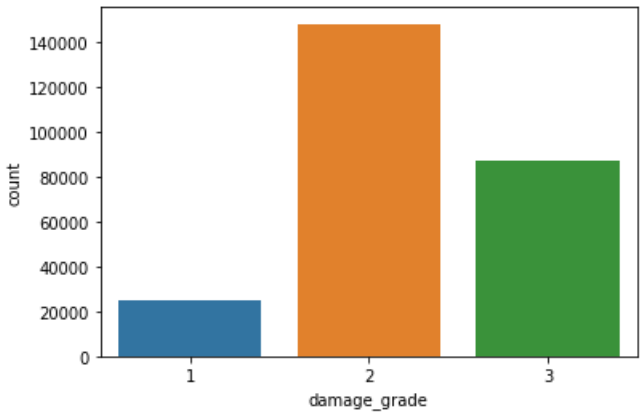

Figure 2: Class Distribution

También es adecuado revisar la importancia de cada variable a la hora de construir el modelo predictivo (del que haremos uso más adelante) como podemos observar a continuación.

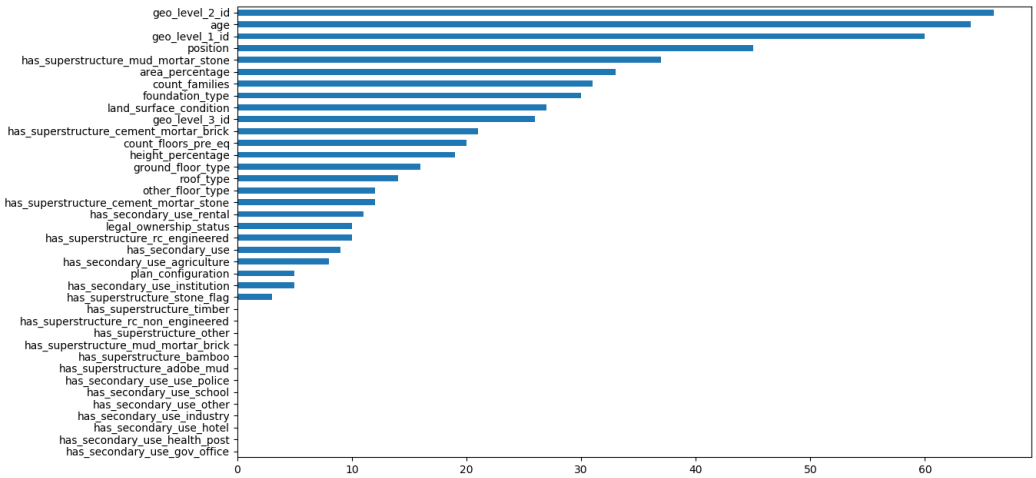

Figure 3: Feature importances

En la siguiente gráfica se ha usado el atributo *feature_importances_* de un modelo XGB y nos da información acerca de qué variables son mas informativas dentro del conjunto de datos.

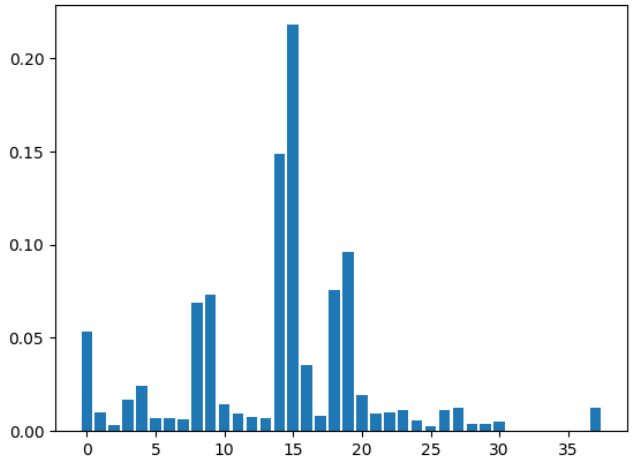

Figure 4: Feature importances

Como se observa a priori, es necesaria la eliminación de algunas de estas variables para guiar de mejor forma a los algoritmos ya que estas variables aportan poca información y posiblemente ruido y no deben participar en el

proceso pues tendrán una influencia negativa en el desarrollo del mismo.

```
model = xgb.XGBClassifier(n_estimators = 200,n_jobs=-1)
model.fit(X, y)
plt.bar(range(len(model.feature_importances_)), model.
    ↪ feature_importances_)
plt.show()
```

Código usado para la visualización de la figura 4

Para continuar, es interesante estudiar la correlación que existe entre las diferentes variables de nuestro conjunto de datos mediante un heatmap. En nuestro caso las correlaciones se muestran en la figura 5

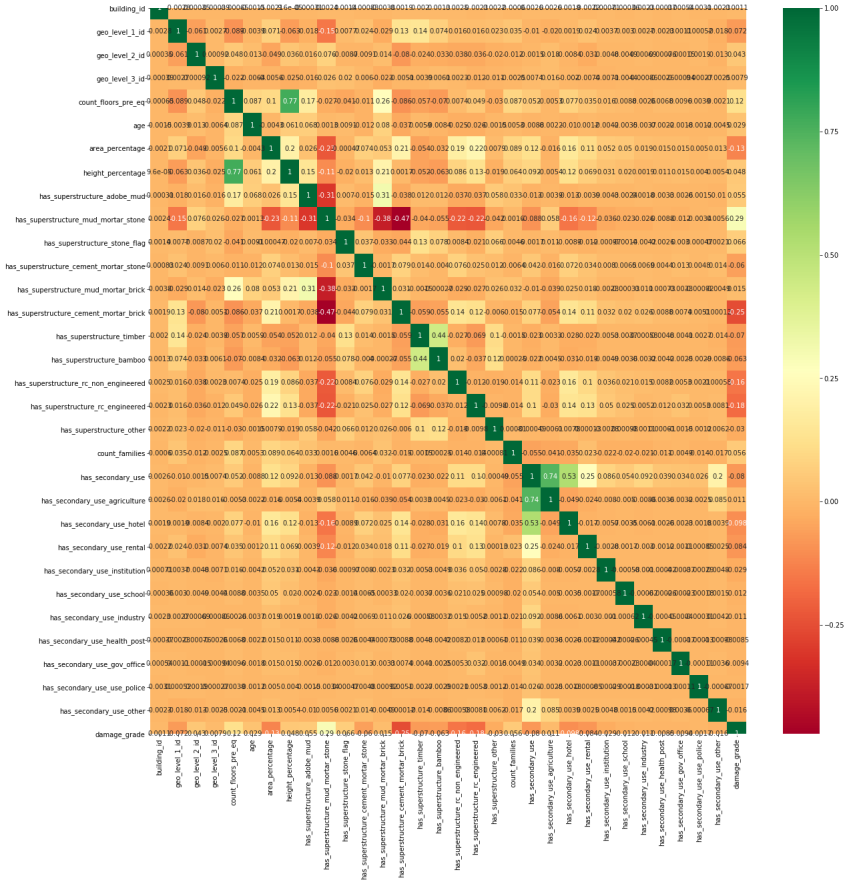

Figure 5: Heatmap del conjunto de datos

A la luz de las visualizaciones, es preciso realizar un preprocesamiento de los

datos para aumentar la calidad de los mismos.

3 Preprocesamiento

3.1 Eliminación de variables

Los criterios elegidos para eliminar las siguientes variables han sido tener poca importancia a la hora de construir el modelo predictivo y por tanto tener un carácter poco informativo o estar muy poco correladas con la variable **damage_grade**. Las variables en cuestión son las siguientes:

- has_secondary_use_hotel.
- has_secondary_use_rental.
- has_superstructure_rc_non_engineered.
- has_superstructure_rc_engineered.
- has_superstructure_cement_mortar_brick.
- has_superstructure_cement_mortar_stone.

3.2 Valores perdidos

Para realizar la imputación de valores perdidos se ha hecho uso de la clase **MissForest**, de la biblioteca **MissingPy**. MissForest imputa los valores perdidos usando Random Forest de forma iterativa. Por defecto, se comienza imputando los valores perdidos de la columna con el menor número de valores perdidos – podemos llamarla columna candidata. El primer paso implica llenar cualquier valor que falte en las columnas restantes, no candidatas, con una suposición inicial, que es la media de la columna para las columnas que representan variables numéricas y el modo de columna para las columnas que representan variables categóricas. Después de eso, el imputador ajusta un modelo Random Forest con la columna candidata como la variable de resultado y las columnas restantes como los predictores en todas las filas en las que no falten los valores de la columna candidata. Después del ajuste, las filas que falten de la columna candidata se imputan usando la predicción del Random Forest ajustado. Las filas de las columnas no candidatas actúan como los datos de entrada para el modelo ajustado. Después de esto, el imputador pasa a la siguiente columna candidata con el segundo número más pequeño de valores perdidos. El proceso se repite para cada columna con un valor perdido hasta que se cumpla el criterio de parada [4].

Nos encontramos ante un conjunto de datos que no contiene valores perdidos y por tanto no es necesario el tratamiento de los mismos. En un principio no se disponía de dicha información y para obtenerla se hizo uso del imputador MissForest, el cual tras ejecutarse proporcionaba un *Warning* indicando que

no se habían encontrado valores perdidos y por tanto el algoritmo **MissForest** devolvía el conjunto de datos intacto.

```
print ("---- Valores perdidos----")
from missingpy import MissForest
imputer = MissForest(n_estimators = 1000, class_weight
    ↪ ={1:6.5,2:1.5,3:2}, warm_start = True,

n_jobs = -1, random_state = 123456)

X_imputed = imputer.fit_transform(X)
```

Código usado para la imputación de valores perdidos

3.3 Numerización de variables

Se ha realizado una transformación de variables de categóricas a numéricas siguiendo el siguiente esquema:

En primer lugar se extraen las variables que son de tipo categórico. A continuación, filtramos de los datos de training estas variables. Posteriormente, eliminamos dichas variables y aplicamos el LabelEncoder, el cual convierte las variables categoricas en numéricas de forma que nuestro modelo predictivo pueda entenderlas mejor.

Seguidamente se muestra el código que nos ha permitido esta conversión:

```
from sklearn.preprocessing import LabelEncoder
def categorical_numer(training, test, y_training):
    # Se extraen las categorias
    columnas_categoricas = list(training.select_dtypes('object').
        ↪ astype(str))
    variables_categoricas = training[columnas_categoricas]

    # Eliminamos variables en cuestion
    training = training.drop(columns = columnas_categoricas)
    #Aplicamos label encoder
    training_cat = variables_categoricas.apply(preprocessing.
        ↪ LabelEncoder().fit_transform)
    #juntamos el conjunto de train
    training_procesado = pd.concat((training, training_cat), axis
        ↪ =1, join='outer', ignore_index=False,
                                levels=None, names=None,
                                    ↪ verify_integrity=False,
                                    ↪ copy=True)

    # Repeticion del proceso para el conjunto de test
    columnas_categoricas = list(test.select_dtypes('object').
        ↪ astype(str))

    variables_categoricas = test[columnas_categoricas]

    test = test.drop(columns = columnas_categoricas) # Eliminamos
        ↪ variables en cuestion

    test_cat = variables_categoricas.apply(preprocessing.
        ↪ LabelEncoder().fit_transform)

    test_procesado = pd.concat((test, test_cat), axis=1, join='
        ↪ outer', ignore_index=False,
                                levels=None, names=None,
                                    ↪ verify_integrity=False,
                                    ↪ copy=True)

    return training_procesado, test_procesado
```

3.4 Normalización de variables

La estandarización de un conjunto de datos es un requerimiento común para muchos estimadores de aprendizaje automático: pueden comportarse mal si las

características individuales no se parecen más o menos a los datos estándar distribuidos normalmente [1].

En este caso se ha usado el **StandardScaler** que estandariza una característica restando la media y luego escalando a la desviación unitaria. La desviación unitaria se consigue dividiendo todos los valores por la desviación estándar [2].

3.5 Selección de instancias

Estamos ante un problema de clasificación desequilibrada y por tanto se hace necesaria una selección de instancias previa al posterior procesamiento. Para tal acometido, en este apartado se hizo uso del paquete **smote-variants** el cual proporciona hasta 85 variantes del SyntheticMinority Oversampling Technique (SMOTE) y suministra algunos códigos de selección y evaluación de modelos.

Para obtener una mejor puntuación en la plataforma, se realizaron varias búsquedas a fin de obtener qué variante SMOTE conseguía la mayor puntuación de F1, que como sabemos equilibra la *precisión* y el *recall* de un clasicador.

El autor de la biblioteca smote-variants también proporciona rankings que aglutinan las diferentes versiones de SMOTE teniendo en cuenta diferentes parámetros de forma que cada cual use una variante u otra dependiendo del uso que se le vaya a dar, como puede verse en la figura 6.

rank	sampler	average score	AUC	AUC rank	G	G rank	F1	F1 rank	P20	P20 rank
1	polynom-fit-SMOTE [43]	2.50	0.9025	6	0.8708	1	0.6952	1	0.9925	2
2	ProWSyn [6]	4.50	0.9044	1	0.8684	4	0.6903	3	0.9911	10
3	SMOTE-IPF [89]	7.50	0.9026	5	0.8687	3	0.6879	9	0.9909	13
4	Lee [62]	8.00	0.9023	7	0.8683	5	0.6881	8	0.9910	12
5	SMOBD [17]	9.25	0.9022	8	0.8677	6	0.6889	4	0.9906	19
6	G-SMOTE [91]	13.50	0.9019	10	0.8651	18	0.6866	12	0.9908	14
7	CCR [57]	14.25	0.9021	9	0.8620	30	0.6879	10	0.9913	8
8	LVQ-SMOTE [78]	14.75	0.9028	3	0.8623	29	0.6836	24	0.9922	3
9	Assembled-SMOTE [111]	15.50	0.9027	4	0.8669	7	0.6886	5	0.9827	46
10	SMOTE-TomekLinks [8]	15.75	0.9010	14	0.8662	9	0.6847	20	0.9906	20

Figure 6: Ranking de mejores sobremuestreadores

Se realizaron varias pruebas, efectuándose ejecuciones con **SMOTE-ENN** y **SMOTE-IPF** en el cual sus autores afirman que *"el desequilibrio de clases no es un problema en sí mismo y la degradación del rendimiento también está asociada a otros factores relacionados con la distribución de los datos. Uno de ellos es la presencia de ejemplos ruidosos y limítrofes, estos últimos situados en las zonas que rodean los límites de las clases"*. Ésta última variante reducía en gran cantidad el número de instancias del conjunto de datos pero tal y como vaticinaba el ranking [3], la variante de SMOTE que conseguía mejor puntuación **f1** era el muestreador **polynom-fit-SMOTE**.

3.6 Selección de características

Para la ocasión, se ha hecho uso del algoritmo Boruta, el cual está construido alrededor del algoritmo **Random Forest**. Intenta capturar todas las características importantes e interesantes que puede tener en su conjunto de datos con respecto a una variable de resultado.

```
from boruta import BorutaPy
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(max_depth=5, n_estimators=200,
    ↪ class_weight={1:5, 2:1.2, 3:1.7}, n_jobs=-1) #warm_start=
    ↪ True, class_weight={1:5, 2:1.2, 3:1.7},

feature_selector = BorutaPy(rf, n_estimators='auto', verbose=0,
    ↪ max_iter=9, random_state=123456)
feature_selector.fit(X_samp, y_samp)
print("Ranking de caracteristicas: ", feature_selector.ranking_)
X_filtered = feature_selector.transform(X_samp)
```

Con la configuración mostrada, la ejcución de este algoritmo no era nada útil pues no descartaba ninguna de las variables y dejaba el conjunto de datos tal cual se encontraba al inicio y por este motivo se ha descartado su uso.

4 Procesamiento

Para el procesamiento de los datos el modelo elegido ha sido **LightGBM** pues aportaba buenos resultados y a su vez una eficiencia mucho mayor que la de otros modelos predictivos probados (Random Forest, AdaBoost...).

Para la parametrización de este modelo se ha seguido la documentación oficial de LightGBM [5], la cual indica, por ejemplo, que para conseguir una buena precisión se debía usar un *learning_rate* bajo junto con un *num_iterations alto* o por otra parte usar un *num_leaves*(número de hojas en un árbol) alto lo cual podría causar over-fitting y se observó que este ocurría a partir de los valores 130-135.

En este problema las clases estan desbalanceadas, y en este sentido es posible usar algoritmo que sea sensible a ese desequilibrio e integre una estrategia pa compensar ese desequilibrio. Existe la posibilidad de poner un peso a cada clase, y cada vez que acierte en una determinada clase multiplico el rendimiento por un número k que yo determino.

El parámetro **classWeight** de LightGBM hace exactamente lo descrito en el anterior párrafo y es adecuado destacar que el peso que se asocia a cada clase tiene que ser más alto cuando la clase es menos frecuente.

Tras varias ejecuciones añadiendo este parámetro se observó una pérdida de rendimiento y para mejorarlo, se usó el algoritmo que se indica a continuación.

4.1 Grid Search

Para hacer ejecuciones más certeras, se decidió usar el algoritmo **GridSearch**, el cual es un proceso que tiene el objetivo de escanear los datos para configurar los parámetros óptimos para un modelo determinado (en mi caso LightGBM). Dependiendo del tipo de modelo utilizado, ciertos parámetros son necesarios. Es importante señalar que la Grid Search puede ser extremadamente costosa desde el punto de vista computacional y puede llevar mucho tiempo de ejecución. Grid-Search construye un modelo en cada combinación de parámetros posible. Se itera a través de cada combinación de parámetros y almacena un modelo para cada combinación.

5 Soluciones

En la siguiente tabla[1] se presentan las soluciones obtenidas a lo largo del desarrollo de esta práctica.

Fecha -Hora	Posición	Score-DrivenData	Preprocesado realizado	Algoritmos empleados	Configuración parámetros
	369	0.6883	PD*	LightGBM	LGBMClassifier(objective='regression_l1',n_estimators=200,n_jobs=5)
	366	0.6886	PD*	Random Forest	RandomForestClassifier(n_estimators = 500, class_weight={1:6.5,2:1.5,3:2}, n_jobs=6)
	358	0.6892	PD*	LightGBM+GridSearch	LGBMClassifier(objective='regression_l1',n_estimators=200,n_jobs=5) params_lgbm = {'feature_fraction':[i/10.0 for i in range(3,6)], 'learning_rate':[0.05,0.1], 'num_leaves':[30,50], 'n_estimators':[200]}
	363	0.6890	PD*	Random Forest+GridSearch	RandomForestClassifier(n_estimators = 500, class_weight={1:6.5,2:1.5,3:2}, n_jobs=6); cw1 = {1:6.5,2:1.5,3:2} cw2 = {1:8,2:1.3,3:1.8} cw3 = {1:7,2:1.3,3:2} params_rf = {'n_estimators':[500, 1000, 1100], 'class_weight' :[cw1, cw2, cw3], 'n_jobs':[-1]}
	342	0.6902	PD*+Boruta+SMOTE-IPF	LightGBM	BorutaPy(rf, max_iter=9, n_estimators=200, SMOTE_IPF(proportion=1.0, n_jobs = -1, random_state=123456) lgb.LGBMClassifier(objective='regression_l1',n_estimators=1000,n_jobs=-1)
	320	0.6945	PD*+polynom_fit_SMOTE+MissForest	LightGBM	polynom_fit_SMOTE(random_state = 123456) MissForest(n_estimators = 1000, class_weight={1:6.5,2:1.5,3:2}, warm_start = True, n_jobs = -1, random_state = 123456) lgb.LGBMClassifier(objective='regression_l1',n_estimators=1000,n_jobs=-1)
	307	0.7003	PD*+polynom_fit_SMOTE+MissForest	Raadom Forest	polynom_fit_SMOTE(random_state = 123456) MissForest(n_estimators = 1000, class_weight={1:6.5,2:1.5,3:2}, warm_start = True, n_jobs = -1, random_state = 123456) RandomForestClassifier(n_estimators = 1000, class_weight={1:6.5,2:1.5,3:2}, n_jobs=-1)
	231	0.7354	Eliminación Variables+ CategoricalToNumeric+ StandardScaler+polynom_fit_SMOTE	LightGBM	polynom_fit_SMOTE(random_state = 123456) lgb.LGBMClassifier(objective='regression_l1', n_estimators=500,n_jobs=-1)
	223	0.7369	Eliminación Variables+ CategoricalToNumeric+ StandardScaler+polynom_fit_SMOTE	LightGBM	polynom_fit_SMOTE(random_state = 123456) LGBMClassifier(objective='regression_l1', num_leaves = 70, n_estimators=500,n_jobs=-1)
2019-12-31 20:55:03 UTC	123	0.7452	Eliminación Variables+ CategoricalToNumeric+ StandardScaler+polynom_fit_SMOTE	LightGBM+GridSearch	LGBMClassifier(objective='regression_l1', num_leaves=130, n_estimators=600,n_jobs=-1) params_lgbm = {'feature_fraction':[i/10.0 for i in range(3,6)], 'learning_rate':[0.1, 0.2], 'num_leaves':[100, 120], 'n_estimators':[700,800]} GridSearchCV(lgbm, params_lgbm, cv=3, n_jobs=-1, verbose=1, scoring='f1_micro')
2019-12-31 21:11:16 UTC	109	0.7459	Eliminación Variables+ CategoricalToNumeric+ StandardScaler+polynom_fit_SMOTE	LightGBM+GridSearch	Todo igual cambiando el parámetro del LGBMClassifier N_estimators = 1000
2019-12-31 21:37:56 UTC	111	0.7458	PD* + Eliminación Variables+ CategoricalToNumeric+ StandardScaler+polynom_fit_SMOTE	LightGBM+GridSearch	Todo igual cambiando el parámetro del LGBMClassifier N_estimators = 1700

[1] PD* : Procesamiento por defecto

References

[1] Sklearn.preprocessing.StandardScaler
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[2] Scale, Standardize, or Normalize with Scikit-Learn
https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02

[3] Research Gate - The top performer oversamplers
https://www.researchgate.net/figure/The-top-performer-oversamplers-ranked-by-the-combination-of-all-scores-Besides-the_tbl3_334732374

[4] missingpy - Project description
https://pypi.org/project/missingpy/

[5] Parameters Tuning - LightGBM Documentation
https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html#for-better-accuracy