

Práctica 3:

Competición en DrivenData

Curso 2019/2020

SOFÍA ALMEIDA BRUNO
sofialmeida@correo.ugr.es

Grupo IN 2
Jueves 9:30-10:30

1. Captura de pantalla de *Submissions*

SUBMISSIONS

Score	Submitted by	Timestamp
0.6883	Sofía_Almeida_UGR	2019-12-05 10:07:58 UTC
0.6874	Sofía_Almeida_UGR	2019-12-19 11:18:10 UTC
0.6894	Sofía_Almeida_UGR	2019-12-22 17:32:55 UTC
0.7227	Sofía_Almeida_UGR	2019-12-22 20:39:08 UTC
0.7245	Sofía_Almeida_UGR	2019-12-23 17:14:45 UTC
0.7228	Sofía_Almeida_UGR	2019-12-23 18:58:58 UTC
0.7253	Sofía_Almeida_UGR	2019-12-23 21:29:30 UTC
0.7167	Sofía_Almeida_UGR	2019-12-24 10:11:57 UTC
0.6969	Sofía_Almeida_UGR	2019-12-24 10:45:07 UTC
0.7177	Sofía_Almeida_UGR	2019-12-24 11:55:27 UTC
0.6823	Sofía_Almeida_UGR	2019-12-25 11:42:49 UTC
0.7388	Sofía_Almeida_UGR	2019-12-25 12:40:41 UTC
0.7412	Sofía_Almeida_UGR	2019-12-25 12:52:50 UTC
0.7444	Sofía_Almeida_UGR	2019-12-26 15:35:06 UTC

Figura 1: Captura de pantalla de *Submissions* - Parte 1.































0.7452	Sofía_Almeida_UGR 	2019-12-26 16:34:32 UTC 
0.7448	Sofía_Almeida_UGR 	2019-12-26 18:24:17 UTC 
0.7443	Sofía_Almeida_UGR 	2019-12-27 12:29:53 UTC 
0.7425	Sofía_Almeida_UGR 	2019-12-27 17:40:36 UTC 
0.7457	Sofía_Almeida_UGR 	2019-12-27 19:01:06 UTC 
0.7416	Sofía_Almeida_UGR 	2019-12-28 20:20:03 UTC 
0.7468	Sofía_Almeida_UGR 	2019-12-28 22:04:41 UTC 
0.7469	Sofía_Almeida_UGR 	2019-12-29 16:21:24 UTC 
0.7240	Sofía_Almeida_UGR 	2019-12-29 22:22:43 UTC 
0.7482	Sofía_Almeida_UGR 	2019-12-29 22:24:07 UTC 
0.7482	Sofía_Almeida_UGR 	2019-12-30 15:57:07 UTC 
0.7475	Sofía_Almeida_UGR 	2019-12-30 19:05:43 UTC 
0.7487	Sofía_Almeida_UGR 	2019-12-31 10:40:04 UTC 
0.7488	Sofía_Almeida_UGR 	2019-12-31 12:34:31 UTC 
0.7479	Sofía_Almeida_UGR 	2019-12-31 15:02:55 UTC 

Figura 2: Captura de pantalla de *Submissions* - Parte 2.

Índice

1. Captura de pantalla de <i>Submissions</i>	1
2. Pruebas realizadas	4
3. Diario de pruebas	10
3.1. p3_00 - Archivo inicial	10
3.2. Análisis exploratorio de los datos	10
3.3. p3_01 - Ajuste de <i>Lightgbm</i>	12
3.4. p3_02 - Parámetros de <i>Lightgbm</i> contra el desbalanceo	12
3.5. p3_03 - Binarización de variables categóricas	12
3.6. p3_04 - Selección de variables con <i>VarianceThreshold</i>	13
3.7. p3_05 - Selección de variables con <i>SelectKBest</i>	14
3.8. p3_06 - Variación de los parámetros de <i>Lightgbm</i>	16
3.9. p3_07 - <i>RandomForest</i>	16
3.10. p3_08 - Ajuste de <i>RandomForest</i>	16
3.11. p3_09 - Ajuste de <i>RandomForest</i>	16
3.12. p3_10 - Binarización con <i>OneHotEncoder</i>	17
3.13. p3_11 - Binarización con <i>DictVectorizer</i>	17
3.14. p3_12 - Ajuste <i>Lightgbm</i>	17
3.15. p3_13 - Ajuste <i>Lightgbm</i>	17
3.16. p3_14 - Ajuste <i>Lightgbm</i>	18
3.17. p3_15 - Análisis sobre las variables <i>geo_level_1_id</i>	18
3.18. p3_16 - Ajuste de <i>Lightgbm</i>	22
3.19. p3_17 - <i>XGBoost</i>	22
3.20. p3_18 - Ajuste <i>XGBoost</i>	22
3.21. p3_19 - <i>Stacking</i>	22
3.22. p3_20 - <i>Stacking</i>	23
3.23. p3_21 - <i>Stacking</i>	23
3.24. p3_sampling - Muestreo aleatorio	23
3.25. p3_22 - Detección de anomalías con <i>PyOD</i>	23
3.26. p3_23 - <i>Stacking</i> sin selección de variables	24
3.27. p3_24 - <i>Stacking</i> sin selección de variables	24
3.28. p3_25 - Ajuste de <i>StackingClassifier</i>	24
3.29. p3_26 - Ajuste de <i>StackingClassifier</i>	24
3.30. p3_27 - Ajuste de <i>StackingClassifier</i>	24
3.31. p3_28 - Ajuste de <i>StackingClassifier</i>	25
3.32. Pruebas fallidas	25
3.32.1. <i>categorical_features</i> de <i>Lightgbm</i>	25
3.32.2. Reducción de la dimensionalidad con <i>UMAP</i>	25
3.32.3. Selección de variables mediante información mutua	25
3.32.4. Selección de variables mediante <i>Boruta</i>	25
3.32.5. Visualización con <i>TSNE</i>	25
3.32.6. p3_naive - <i>NaiveBayes</i>	26
3.32.7. Detección de anomalías + <i>Stacking</i>	26
Referencias	27

2. Pruebas realizadas

Tabla 1: Pruebas realizadas

ID	Fecha-hora	Pos.	Sc.-Training	Sc.-Test	Preprocesado	Algoritmos	Parámetros
00	5/12/2019 10:07:58 UTC	315	0.7264	0.6883		<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2
01	19/12/2019 11:18:10 UTC	356	0.7358	0.6874		<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, feature_fraction = 0.5, learning_rate = 0.1, num_leaves = 50
02	22/12/2019 17:32:55 UTC	360	0.7294	0.6894		<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, num_leaves = 35, scale_pos_weight = 0.1
03	22/12/2019 20:39:08 UTC	243	0.7339	0.7227	get_dummies para todas las variables categóricas	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, num_leaves = 40, scale_pos_weight = 0.1
04	23/12/2019 17:14:45 UTC	242	0.7342	0.7245	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, num_leaves = 40, scale_pos_weight = 0.1
05	23/12/2019 18:58:58 UTC	242	0.7335	0.7228	get_dummies para todas las variables categóricas. Selección de variables con SelectKBest, k = 35	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, num_leaves = 40, scale_pos_weight = 0.1
06	23/12/2019 21:29:30 UTC	234	0.7375	0.7253	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, num_leaves = 45, scale_pos_weight = 0.1

07	24/12/2019 10:11:57 UTC	237	0.8486	0.7167	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.9	<i>RandomForest</i>	n_jobs = -1, max_depth = 20, n_estimators = 300
08	24/12/2019 10:45:07 UTC	237	0.8468	0.6969	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.9	<i>RandomForest</i>	class_weight = 'balanced', max_depth = 20, max_features = 'sqrt', n_estimators = 300
09	24/12/2019 11:55:27 UTC	237	0.9825	0.7177	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>RandomForest</i>	max_depth = 40, max_features = 'sqrt', n_estimators = 500
10	25/12/2019 11:42:49 UTC	239	0.7375	0.6823	OneHotencoder para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, num_leaves = 45, scale_pos_weight = 0.1
11	25/12/2019 12:40:41 UTC	160	0.7693	0.7388	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 500, n_jobs = -1, num_leaves = 55, scale_pos_weight = 0.1
12	25/12/2019 12:52:50 UTC	148	0.7855	0.7412	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 700, n_jobs = -1, num_leaves = 60, scale_pos_weight = 0.1
13	26/12/2019 15:35:06 UTC	117	0.8070	0.7444	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 1000, n_jobs = -1, num_leaves = 65, scale_pos_weight = 0.1

14	26/12/2019 16:34:32 UTC	107	0.8184	0.7452	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 1000, n_jobs = -1, num_leaves = 80, scale_pos_weight = 0.05
15	26/12/2019 18:24:17 UTC	107	0.8259	0.7448	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 1000, n_jobs = -1, num_leaves = 90, scale_pos_weight = 0.05
16	27/12/2019 12:29:53 UTC	110	0.8061	0.7443	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 900, n_jobs = -1, num_leaves = 70, scale_pos_weight = 0.05
17	27/12/2019 17:40:36 UTC	110	0.7913	0.7425	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>XGBoost</i>	predictor = 'cpu_predictor', n_gpus = 0, n_estimators = 200, eta = 0.3, max_depth = 6, max_delta_step = 7
18	27/12/2019 19:01:06 UTC	98	0.8691	0.7457	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	<i>XGBoost</i>	n_estimators = 700, eta = 0.1, max_depth = 10
19	28/12/2019 20:20:03 UTC	104	0.7862	0.7416	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	StackingClassifier (<i>Lightgbm</i> , <i>RandomForest</i> , <i>XGBoost</i>)	<i>Lightgbm</i> : objective = 'regression_l1', n_estimators = 700, n_jobs = -1, num_leaves = 65, scale_pos_weight = 0.05. <i>RandomForest</i> : n_jobs = -1, random_state = 123456, max_depth = 20, n_estimators = 200. <i>XGBoost</i> : n_estimators = 400, eta = 0.1, max_depth = 6

20	28/12/2019 22:04:41 UTC	91	0.8436	0.7468	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	StackingClassifier (<i>Lightgbm</i> , <i>XGBoost</i>)	<i>Lightgbm</i> : objective = 'regression_l1', n_estimators = 1000, n_jobs = -1, num_leaves = 80, scale_pos_weight = 0.05. <i>XGBoost</i> : n_estimators = 700, eta = 0.1, max_depth = 10
21	29/12/2019 16:21:24 UTC	92	0.8427	0.7469	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	StackingClassifier (<i>Lightgbm</i> , <i>RandomForest</i> , <i>XGBoost</i>)	<i>Lightgbm</i> : objective = 'regression_l1', n_estimators = 1000, num_leaves = 80, scale_pos_weight = 0.05. <i>RandomForest</i> : random_state = 123456, max_depth = 20, n_estimators = 300. <i>XGBoost</i> : n_estimators = 700, eta = 0.1, max_depth = 10
22	29/12/2019 22:22:43 UTC	93	0.7387	0.7240	get_dummies para todas las variables categóricas. Isolation Forest con outliers_fraction = 0.05 para eliminar anomalías	<i>Lightgbm</i>	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, num_leaves = 45, scale_pos_weight = 0.1
23	29/12/2019 22:24:07 UTC	58	0.8421	0.7482	get_dummies para todas las variables categóricas	StackingClassifier (<i>Lightgbm</i> , <i>XGBoost</i>)	<i>Lightgbm</i> : objective = 'regression_l1', n_estimators = 1000, n_jobs = -1, num_leaves = 80, scale_pos_weight = 0.05. <i>XGBoost</i> : n_estimators = 700, eta = 0.1, max_depth = 10

24	30/12/2019 15:57:07 UTC	60	0.8498	0.7482	get_dummies para todas las variables categóricas	StackingClassifier (<i>Lightgbm</i> , <i>RandomForest</i> , <i>XGBoost</i>)	<i>Lightgbm</i> : objective = 'regression_l1', n_estimators = 1000, num_leaves = 80, scale_pos_weight = 0.05. <i>RandomForest</i> : random_state = 123456, max_depth = 30, n_estimators = 400. <i>XGBoost</i> : n_estimators = 700, eta = 0.1, max_depth = 10
25	30/12/2019 19:05:43 UTC	66	0.8604	0.7475	get_dummies para todas las variables categóricas	StackingClassifier (<i>Lightgbm</i> , <i>XGBoost</i>)	<i>Lightgbm</i> : objective = 'regression_l1', n_estimators = 1000, n_jobs = -1, num_leaves = 90, scale_pos_weight = 0.05. <i>XGBoost</i> : n_estimators = 900, eta = 0.1, max_depth = 15
26	31/12/2019 10:40:04 UTC	51	0.8627	0.7487	get_dummies para todas las variables categóricas	StackingClassifier (<i>Lightgbm</i> , <i>RandomForest</i> , <i>XGBoost</i>)	<i>Lightgbm</i> : objective = 'regression_l1', n_estimators = 1000, num_leaves = 80, scale_pos_weight = 0.05. <i>RandomForest</i> : random_state = 123456, max_depth = 50, n_estimators = 400. <i>XGBoost</i> : n_estimators = 700, eta = 0.1, max_depth = 10
27	31/12/2019 12:34:31 UTC	51	0.8653	0.7488	get_dummies para todas las variables categóricas	StackingClassifier (<i>Lightgbm</i> , <i>RandomForest</i> , <i>XGBoost</i>)	<i>Lightgbm</i> : objective = 'regression_l1', n_estimators = 1000, num_leaves = 80, scale_pos_weight = 0.05. <i>RandomForest</i> : random_state = 123456, max_depth = 60, n_estimators = 450. <i>XGBoost</i> : n_estimators = 800, eta = 0.1, max_depth = 10

28	31/12/2019 15:02:55 UTC	51	0.8720	0.7479	get_dummies para todas las variables categóricas	StackingClassifier (Lightgbm, RandomForest, XGBoost)	Lightgbm: objective = 'regression_l1', n_estimators = 1000, num_leaves = 80, scale_pos_weight = 0.05. RandomForest: random_state = 123456, max_depth = 65, n_estimators = 500. XGBoost: n_estimators = 850, eta = 0.1, max_depth = 12
----	----------------------------	----	--------	--------	--	--	--

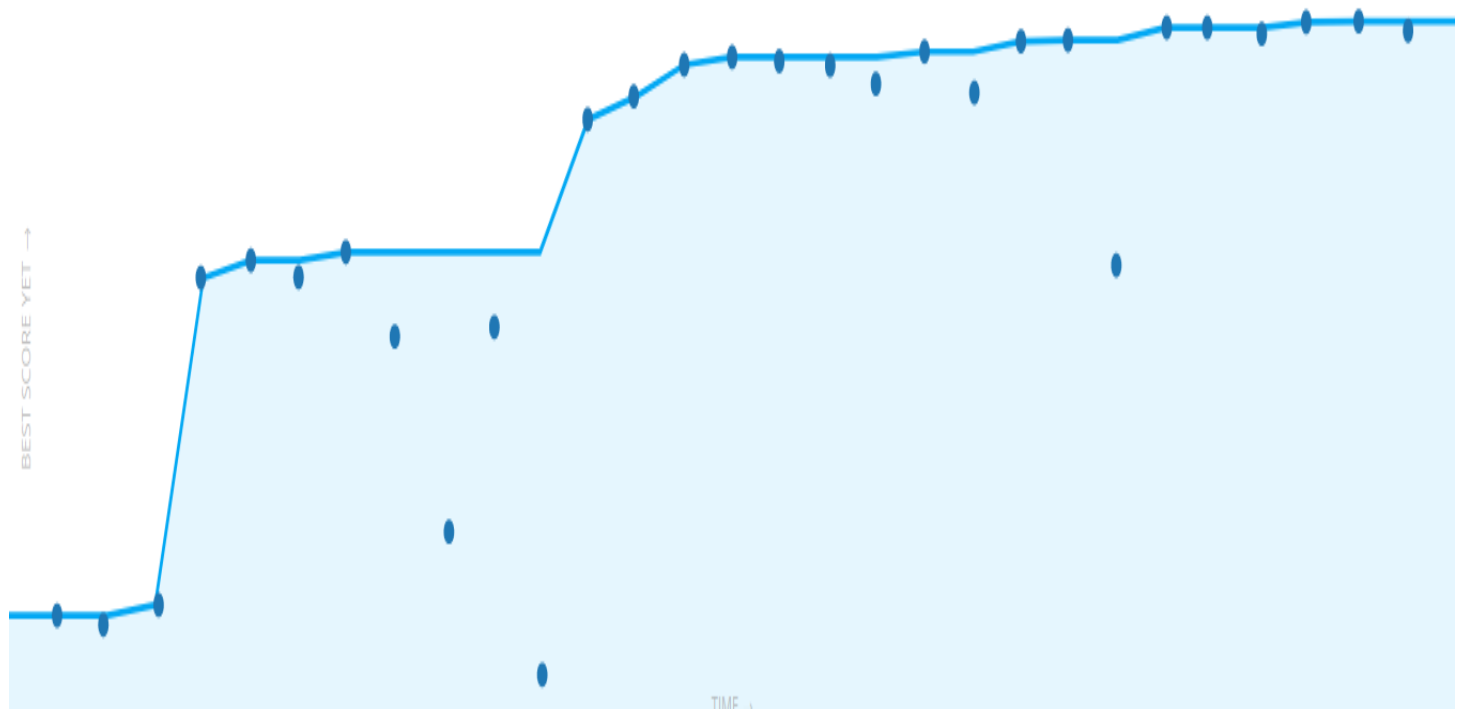


Figura 3: Gráfica de los *submission* de la plataforma DrivenData.

3. Diario de pruebas

3.1. p3_00 - Archivo inicial

Comenzamos aprendiendo a subir los resultados de *test* a la plataforma para que puedan ser validados. El *script* utilizado en esta ocasión es el proporcionado por el profesor de la asignatura. No se realiza ningún preprocesado y el algoritmo a utilizar es *Lightgbm*. Este es un algoritmo de *boosting* que se caracteriza por ser bastante rápido.

3.2. Análisis exploratorio de los datos

Antes de decidir qué hacer a continuación debemos conocer cierta información sobre los datos con los que estamos tratando. Pensamos que seguramente necesitemos algún tipo de preprocesado, pues es lo habitual en este tipo de problema, pero sin conocer exactamente cómo son los datos, si tienen o no ruido, la cantidad de valores perdidos, correlación entre las variables, ... no podremos decidir cómo enfocar el preprocesado ni qué alternativas podrían venir bien al conjunto. Para ello comenzamos a escribir algunas funciones de visualización que nos permitan conocer esta información, se encuentran en el *script* `visualization.py`.

Inspirándonos en [18], observamos en la Figura 4 la distribución de las clases.

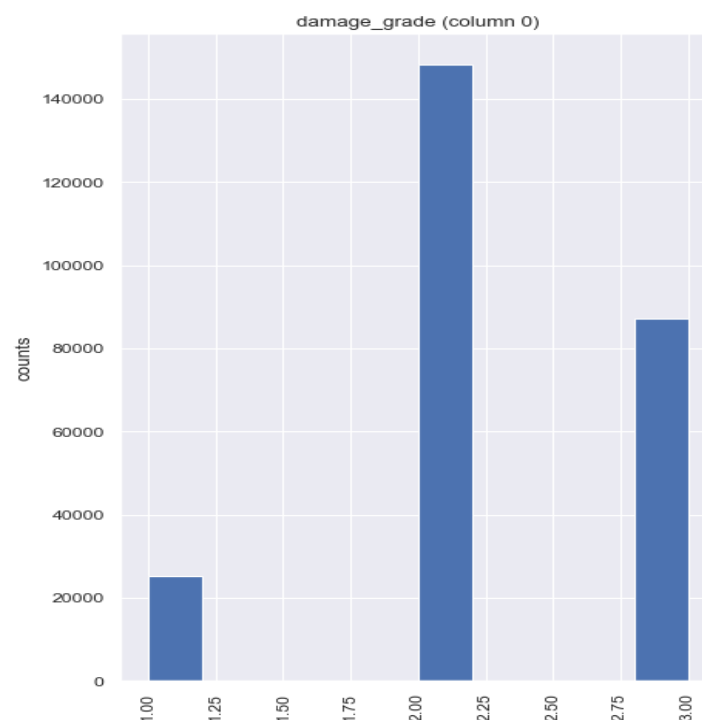


Figura 4: Tamaño de las clases.

Las clases del problema a tratar están claramente desbalanceadas. En la Tabla 2 observamos con exactitud el número de ejemplos que tenemos de cada clase. Ante esta situación se nos ocurren dos opciones: elegir un algoritmo que esté diseñado para manejar clases no balanceadas o utilizar técnicas específicas para paliar el desbalanceo.

Tabla 2: Tamaño de las clases.

Clase	Número de elementos	Tamaño de la clase
1	25124	9.64 %
2	148259	56.89 %
3	87218	33.47 %

También podemos observar la correlación entre las variables en la Figura 5. No hay parejas de variables altamente correladas.

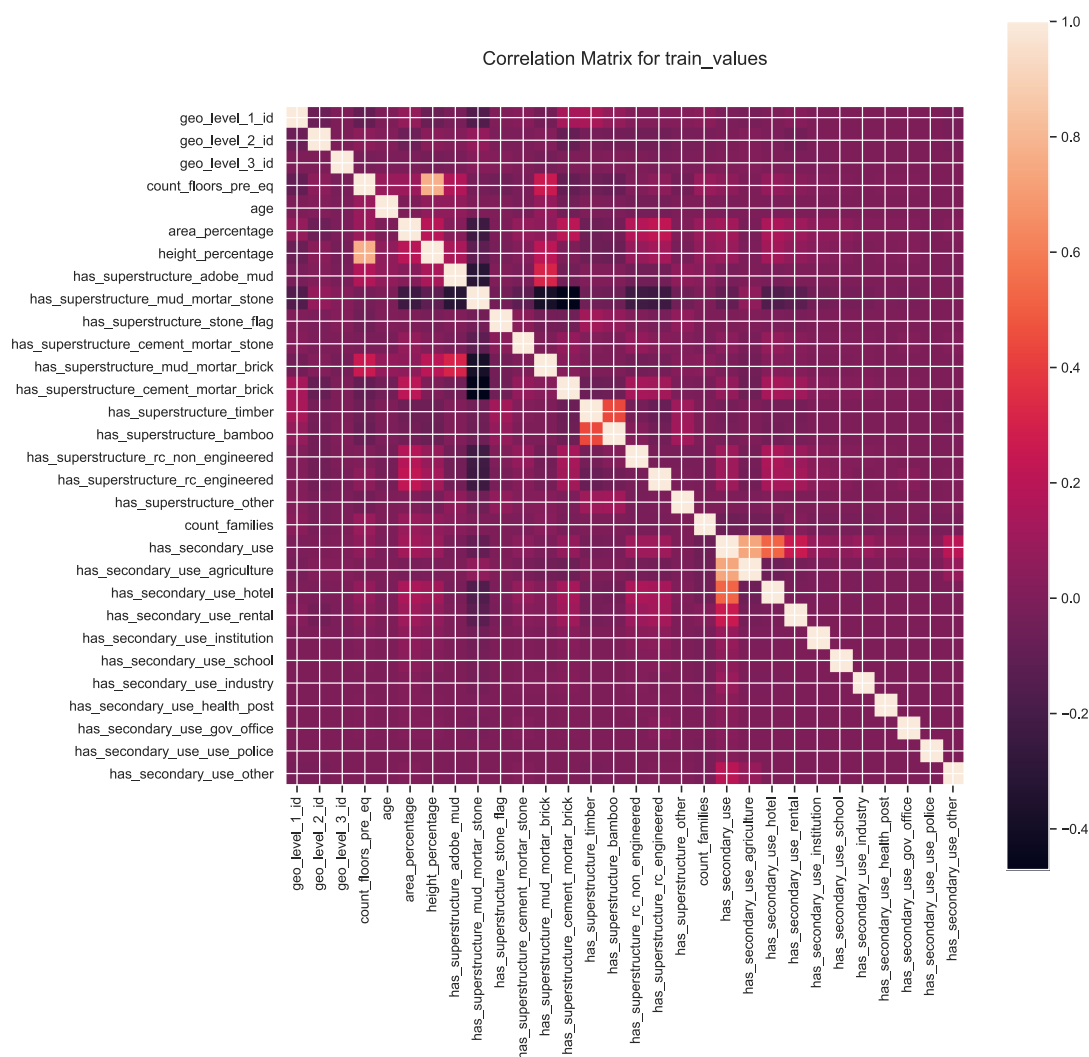


Figura 5: Matriz de correlación.

Continuamos el análisis tomando como referencia [7]. Mediante `data.x.info()` vemos que de las 38 variables 30 son numéricas y 8 categóricas, accedemos a la descripción del problema en [5] para conocer cuántos posibles valores toman las variables categóricas. Tienen entre 3 y 10 posibles valores. En esta página también nos percatamos de que, de las variables numéricas, muchas son de tipo binario.

Ejecutando `data.x.isnull().any()` nos damos cuenta de que nuestras variables no contienen valores

perdidos, nos ahorraremos la imputación de valores perdidos.

3.3. p3_01 - Ajuste de *Lightgbm*

Para tratar de conseguir mejores resultados tenemos, a priori, dos caminos: preprocesar los datos o mejorar el algoritmo. Para mejorar el algoritmo hay, a su vez, que tomar dos decisiones: elección del algoritmo y ajuste de sus parámetros.

En primer lugar, trataremos de ajustar los parámetros de *Lightgbm* a ver si mejora el resultado. Usando el código de muestra `ejemplo_ds_avanzado.py`, implementamos `p3_01.py`, donde nos centraremos en los parámetros `feature_fraction` (que tomará valores entre 0.3 y 0.5), `learning_rate` (probaremos 0.05 y 0.1) y `num_leaves` (comprobaremos los valores 30 y 50), fijando `n_estimators = 200`. Tras realizar el `GridSearch` se obtiene (tras 10 minutos de ejecución) que los mejores parámetros son: `feature_fraction = 0.5`, `learning_rate = 0.1`, `n_estimators = 200`, `num_leaves = 50`.

Obtenemos un resultado en *training* de 0.7358 y en *test* de 0.6874, disminuyendo con respecto al anterior envío, a pesar de haber aumentado en *test*. ¿Se está produciendo sobreajuste?

3.4. p3_02 - Parámetros de *Lightgbm* contra el desbalanceo

Nos preguntamos qué está provocando este sobreajuste, así comprobamos los valores de las variables por defecto y los utilizados. El que más se diferencia es `num_leaves`, por defecto es 31 y estamos tomando 50. Es complicado saber hasta qué punto podemos aumentar el número de hojas sin que se llegue a producir el sobreajuste que provoca malos resultados en la fase de *test*, más teniendo en cuenta que el número de pruebas a realizar es limitado. Por ello, proseguiré tratando de mejorar el rendimiento de este algoritmo de otro modo.

Leemos en [9] que *Lightgbm* tiene dos parámetros que nos permiten tratar de ajustar el desbalanceo de las clases: `is_unbalance` o `scale_pos_weight`. Probaremos a configurar ambos parámetros a ver con cuál conseguimos mejor resultado.

En `p3_02_unbalance.py` partimos del código utilizado en `p3_00.py` añadiendo las variables `num_classes = 3` y `is_unbalance = True`. Conseguimos exactamente el mismo resultado que en el archivo de partida (los archivos `submission` correspondientes no se diferencian) a pesar de que por defecto (en `p3_00.py`) se asumía que no había desbalanceo.

Probamos con la otra opción: variamos el peso de la clase positiva (asumimos que está tomando la mayoritaria como positiva) entre 0.1 y 0.6 modificando el parámetro `scale_pos_weight`. Así, terminamos ejecutando el algoritmo con `n_estimators = 200`, `num_leaves = 35`, `scale_pos_weight = 0.1`, consiguiendo una puntuación de 0.7294 en entrenamiento y 0.6894 al realizar el envío (mejorando la mejor solución obtenida hasta el momento).

3.5. p3_03 - Binarización de variables categóricas

Una opción que podemos utilizar para preprocesar es, en vez de convertir las variables categóricas a numéricas, realizar un *one-to-many* en el que binarizamos las variables categóricas, obteniendo una variable nueva por cada posible valor de las variables categóricas. Como observamos que eran 8 variables categóricas y que no tomaban demasiados valores, además de que *Lightgbm* es un algoritmo rápido, probamos este preprocesado.

Pandas ofrece una función que realiza la transformación sobre el conjunto de datos, es la función `get_dummies` [8]. En el archivo `p3_03.py` encontramos el código correspondiente a esta ejecución. Pasamos de 38 a 68 características.

Conseguimos una puntuación en *training* de 0.7339 y en *test* de 0.7227 (mejorando los resultados anteriores).

3.6. p3_04 - Selección de variables con VarianceThreshold

Aunque *Lightgbm* es un algoritmo ligero y se puede ejecutar con las 68 variables que conseguimos tras binarizar las categóricas, puede que no todas ellas sean importantes para clasificar si el edificio ha sido dañado o no, alejando el modelo del modelo ideal. Por ello, partiendo del código anterior en el que utilizamos *Lightgbm* con 0.1 para *scale_pos_weight* añadimos un método de selección de variables al preprocesado. En *scikit-learn* vienen implementados distintos métodos de selección de características [2].

Nuestro primer intento consiste en eliminar las variables con varianza baja [15]. Por defecto, elimina las variables que tengan varianza nula, esto es, aquellas variables que tengan el mismo valor para todos los ejemplos. Indicamos un umbral para que elimine. Por ejemplo, para 0.9 obtenemos de *f1-score* en *training* 0.7319 (22 variables, en torno a 15 segundos por partición). Con un umbral de 0.95 nos quedamos con 33 variables finalmente, los tiempos por partición van desde 16 hasta 20 segundos, pero la puntuación se ve favorecida (en *training*) siendo ahora 0.7342. La puntuación al subirlo en la web es de 0.7245 (mejorando en milésimas al programa base sin selección de variables).

Tratamos de ver la importancia de cada variable mediante la función *plotImp* que generará un gráfica como la que podemos ver en la Figura 6. En ella se representa el número de veces que una variable se usa en un modelo.

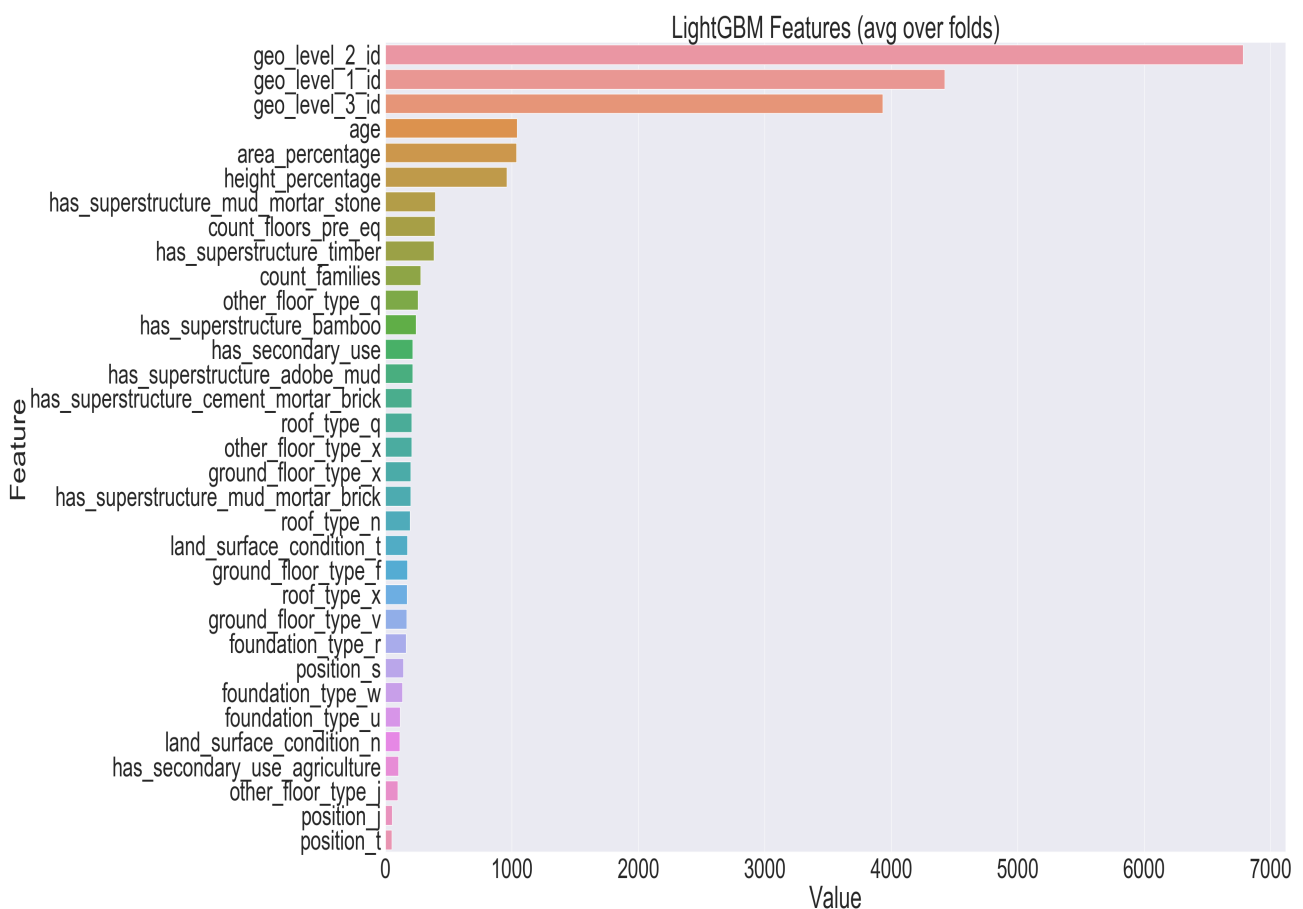


Figura 6: Importancia de las variables - VarianceThreshold, umbral = 0.95.

3.7. p3_05 - Selección de variables con SelectKBest

Seleccionando variables a partir de su varianza hemos mejorado algunas milésimas el resultado, igual utilizando algún otro método de selección más complejo logramos ajustarnos un poco mejor a las variables realmente determinantes en nuestro problema.

Probamos a seleccionar las k mejores variables según *tests* estadísticos univariantes [14].

Notamos que los tiempos de ejecución disminuyen, pasan a rondar los 10 segundos. El resultado en entrenamiento es una puntuación *f1-score* de 0.7251 para 10 características. Observamos en la Figura 7 que las cuatro primeras variables son las mismas que seleccionando según la varianza, pero a partir de ahí varían.

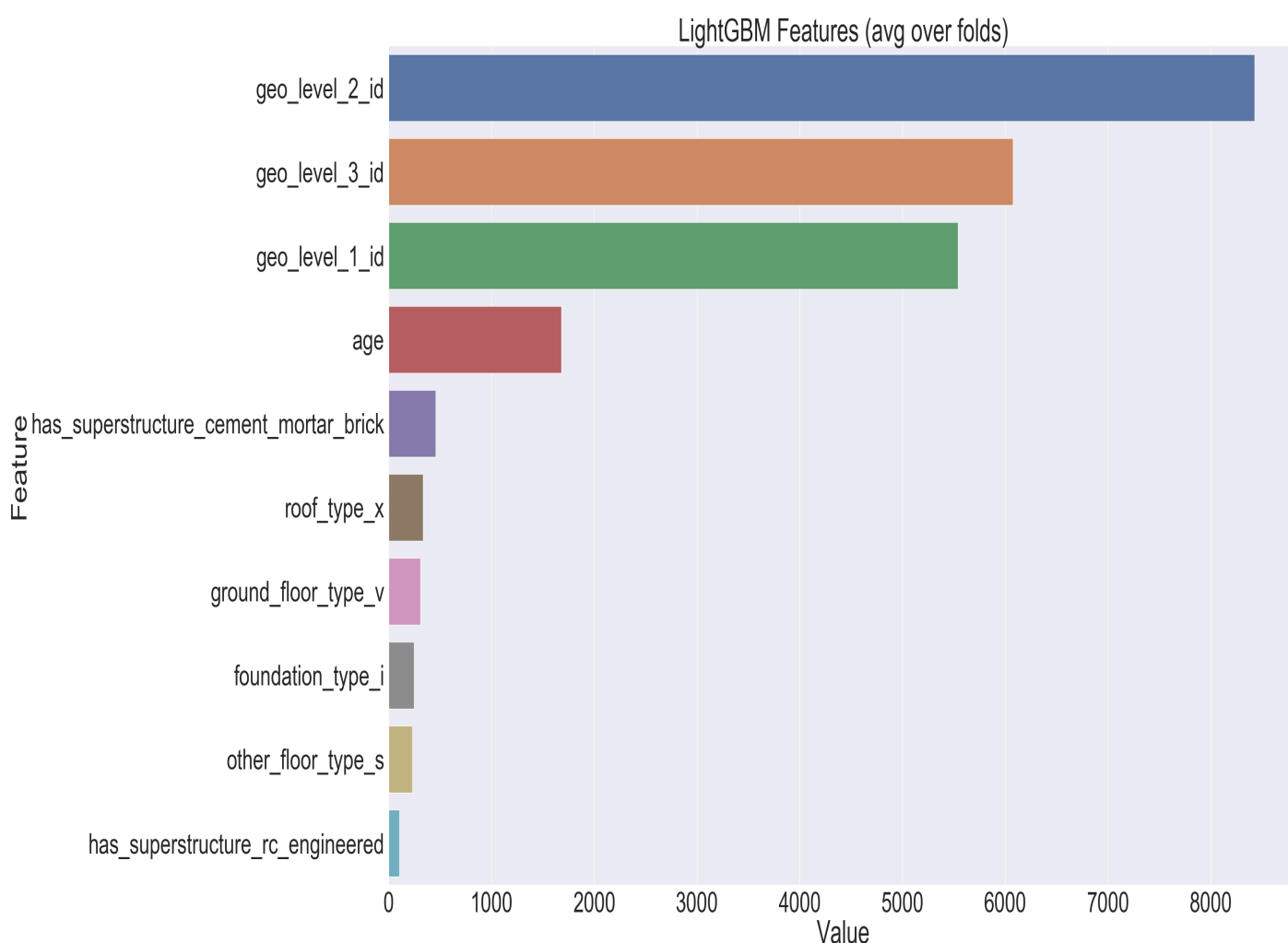


Figura 7: Importancia de las variables tras seleccionar las 10 mejores - SelectKBest.

Si nos quedamos solo con 4 variables no selecciona estas cuatro (se queda con `geo_level_2_id`, `geo_level_3_id`, `age` y `ground_floor_type_v`) y el rendimiento baja a 0.6772.

Como no sabemos cuál es la mejor forma de seleccionar el valor k del preprocesado realizamos pruebas con algunos valores para elegir el mejor. En la Tabla 3 podemos ver los diferentes valores probados.

Tabla 3: Selección de las k mejores características.

Nº de variables	F1-Score	Tiempo por partición (s)
4	0.6772	7
10	0.7251	9.5 - 13
20	0.7296	10 - 15
25	0.7317	12 - 17
30	0.7325	17 - 18
35	0.7335	18 - 20
37	0.7294	19 - 21
40	0.7302	18 - 23

Nos quedamos con 35 variables consiguiendo una puntuación en *test* de 0.7228 (parecida a la conseguida sin realizar la selección de variables). En la Figura 8 observamos las variables más utilizadas para clasificar en esta ocasión.

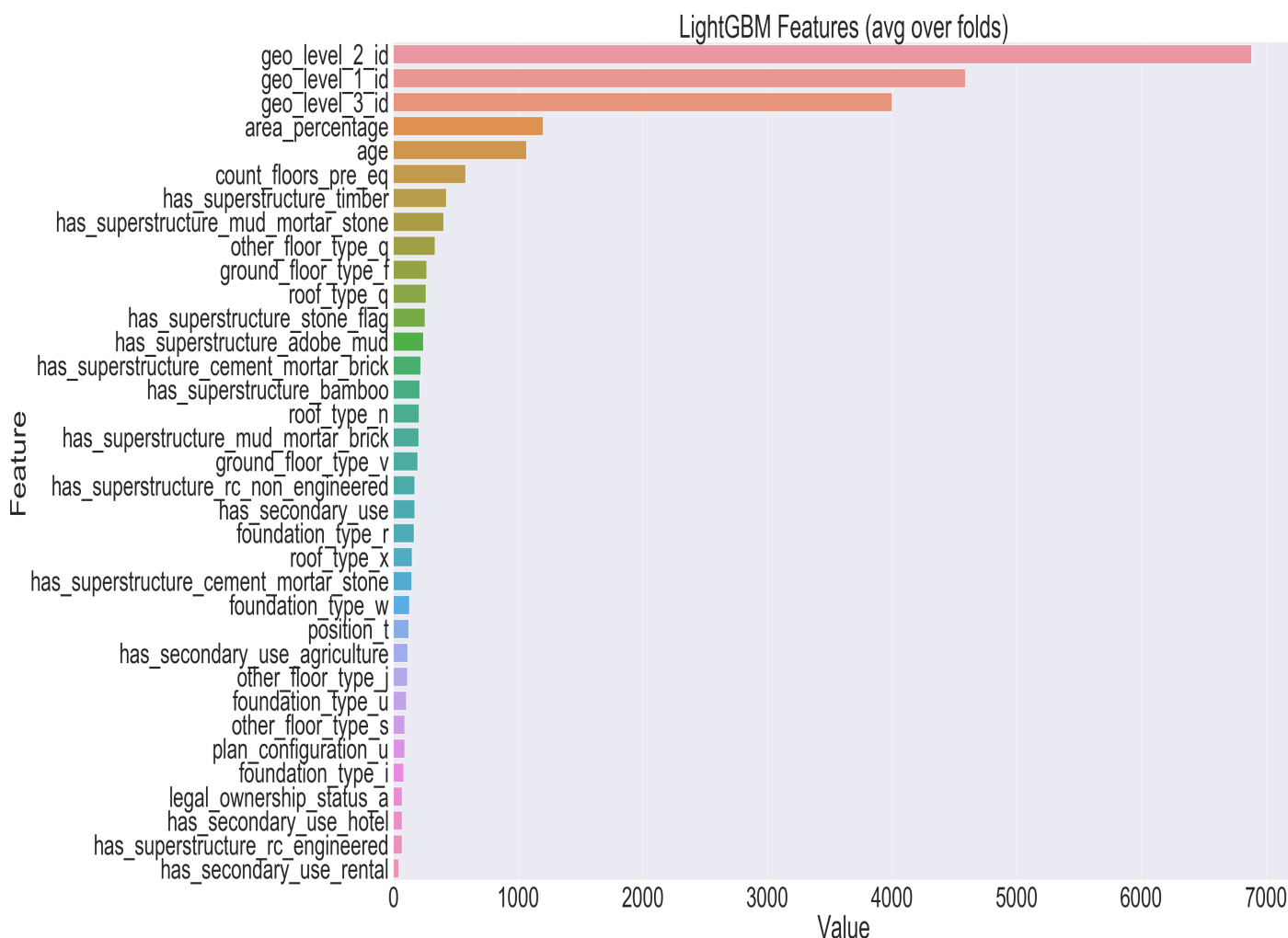


Figura 8: Importancia de las variables tras seleccionar las 35 mejores.

Comparando con las seleccionadas en p3.04 nos damos cuenta de que aunque las primeras son similares y coinciden en algunas más, *height_percentage* es de las más utilizadas en el caso anterior y en este no

fue seleccionada.

3.8. p3_06 - Variación de los parámetros de *Lightgbm*

Tras perder el rato con algunas pruebas fallidas, decidimos no desperdiciar la última subida del día. Utilizaremos `get_dummies` y `VarianceThreshold` en el preprocesado y el algoritmo *Lightgbm* aumentando el número de hojas a 45. La puntuación en *training* es de 0.7375 y en *test* de 0.7253 (mejor hasta el momento).

3.9. p3_07 - *RandomForest*

Pasamos ahora a probar otro algoritmo: *RandomForest* [3].

Investigando un poco vemos que aunque en teoría los árboles de decisión puedan trabajar con todo tipo de variables, en la práctica la implementación de los algoritmos no lo permite. Así que en este caso también debemos numerizar las variables categóricas.

Empezamos de forma similar a como lo hicimos con *Lightgbm*. Numerizamos las variables categóricas, haciendo un preprocesado mínimo. Realizamos un ajuste de parámetros mínimo para ver qué variables puede ser más interesante afinar.

Como tarda mucho tiempo, decido probar las opciones que funcionaron mejor con *Lightgbm*, a ver si en este caso también dan buenos resultados. Utilizando como criterio de selección la varianza e imponiendo un umbral de 0.9 nos quedamos con 22 variables que serán las utilizadas con *RandomForest*. Tarda unos 8 minutos con el `GridSearch` en el que se probó `max_depth`: [10, 20] y `n_estimators`: [200, 300]. Los mejores parámetros resultan ser `max_depth` = 20 y `n_estimators` = 300. Con un tiempo de ejecución de más de un minuto por partición (en torno a los 70 segundos) consigue una puntuación en *training* de 0.8486 y en *test* de 0.7167.

3.10. p3_08 - Ajuste de *RandomForest*

Para tratar de paliar el sobreaprendizaje, siguiendo la recomendación de [1] se fija el número máximo de características en la raíz cuadrada del número de características, confiando también que esto disminuya un poco el tiempo de cómputo. Además, se modifica la variable `class_weight` con la que se actuará sobre el desbalanceo. Se consideran distintas opciones desde el balanceo automático calculado por el algoritmo hasta algunas combinaciones de pesos puestas manualmente. Tras realizar el ajuste de parámetros mediante `GridSearch` (que llevó unos 12 minutos), se elige la siguiente combinación de los mismos: `class_weight` = 'balanced', `max_depth` = 20, `max_features` = 'sqrt', `n_estimators` = 300.

Cada partición tarda unos 70 segundos. Se consigue una puntuación en entrenamiento de 0.8468 (menor que antes), confiando en que el sobreajuste sea menor lo subimos y obtenemos una puntuación de 0.6969 (¡peor que sin balancear!).

3.11. p3_09 - Ajuste de *RandomForest*

Probamos a quitar `max_features` = 'sqrt', obtenemos una puntuación en *training* de 0.8468 (igual que cuando sí que estaba).

Si lo mantenemos pero quitamos el `class_weight`= 'balanced' la puntuación en entrenamiento es de 0.8486, así que decidimos quitar este parámetro pues sin él mejora un poco el resultado.

Por último, tratamos de mejorar un poco aumentando la profundidad máxima, número de estimadores y número de variables. Para ello, se toma en la selección de variables `threshold=(.95 * (1 - .95))`, `max_depth = 40`, `n_estimators = 500`, `max_features = 'sqrt'`. Las iteraciones aumentan su tiempo de ejecución hasta superar los dos minutos (en torno a 150 segundos). La puntuación en entrenamiento es de 0.9825, ¿estará sobreajustando? Lo comprobamos subiendo los resultados y obteniendo una puntuación de 0.7177. ¿Cómo podemos evitar este sobreajuste?

3.12. p3_10 - Binarización con OneHotEncoder

El gran sobreajuste de *RandomForest* y su tiempo elevado de ejecución hace que abandonemos este algoritmo de momento y volvamos a *Lightgbm*. A partir de [11] descubrimos que `get_dummies` no es la única forma de categorizar variables. Como este cambio hizo que mejoraran los resultados probamos diferentes formas de binarizar [6] a ver si varían los resultados.

Partiendo del código `p3_06.py` que fue el que mejores resultados dio, realizaremos la binarización con `OneHotEncoder`. El resultado en *training* es similar al del código original aunque si comparamos los archivos *submission* correspondientes nos damos cuenta de que difieren en 10273/86867¹ objetos, probamos a subirlo a ver cuál es mejor. Sin embargo, el resultado en *test* es bastante peor, de 0.6823².

3.13. p3_11 - Binarización con DictVectorizer

Volvemos a partir del código de `p3_06.py` y probamos la otra alternativa de [6]: `DictVectorizer` [13].

Tras la binarización y selección de variables se consiguen otra vez 33 y una puntuación en *training* de 0.7375, pero en este caso tenemos 0 diferencias con el original. Guardamos este código en el archivo `p3_dict.py` y tratamos de ajustar un poco más los parámetros utilizados en `p3_06.py`.

Así, probamos las combinaciones de los parámetros: `learning_rate = 0.1`, `num_leaves = [45, 50, 55]`, `n_estimators = [200, 300, 400, 500]`.

Tras largo rato de ejecución para probar todas las combinaciones (36), se concluye que los mejores parámetros son: `num_leaves = 55` y `n_estimators = 500`. El aumento de estos valores provoca un incremento en el tiempo de ejecución (unos 40 segundos por partición). Consiguiendo un *f1-score* en entrenamiento de 0.7693, que se traduce en un valor en *test* de 0.7388 (mejor hasta el momento).

3.14. p3_12 - Ajuste Lightgbm

Parece que aumentar el número de hojas y estimadores es positivo para el resultado, pruebo a ejecutar una variación del archivo `p3_11`, esta vez con `num_leaves = 60` y `n_estimators = 700`. El tiempo de ejecución se ve afectado, cada partición tarda entre 50 y 60 segundos. Consigo una puntuación en *training* de 0.7855 y en *test* de 0.7412 (mejor que en el archivo de partida, reforzando la hipótesis).

3.15. p3_13 - Ajuste Lightgbm

Nos preguntamos cuánto podemos aumentar estos valores antes de que se produzca sobreajuste. Ejecutamos, volviendo a aumentar ambos valores: `num_leaves = 65` y `n_estimators = 1000`. Los tiempos de ejecución pasan a estar en el rango 60-75 segundos por partición. Conseguimos una puntuación al entrenar de 0.8070 y al realizar un envío de 0.7444, mejorando el resultado anterior.

¹Realizamos este cálculo mediante `diff -U 0 file1 file2 | grep ^ @ | wc -l`.

²Aquí nos quedamos con la duda de si subimos el archivo correcto, esperábamos resultados similares al menos a los obtenidos con `get_dummies`. La limitación en el número de envíos hizo que nunca lo comprobara.

3.16. p3_14 - Ajuste *Lightgbm*

Podemos hacer pruebas para tratar de ajustar el resto de parámetros, para ello partimos del archivo `p3_06.py` (para evitar tiempos de ejecución demasiado grandes) y asumiremos que si mejora en este caso, mejorará al aumentar el número de estimadores y hojas.

Pasamos a comprobar qué umbral es el más adecuado en la selección de variables, en la Tabla 4 podemos observar los resultados de las ejecuciones para el distinto valor del umbral.

Tabla 4: Selección de características variando el umbral.			
Umbral	Nº variables	F1-Score - Trainig	Tiempo por partición (s)
0.9	22	0.7341	14-16
0.91	24	0.7344	14-17
0.92	26	0.7348	15-19
0.93	27	0.7356	16-22
0.94	30	0.7364	21-26
0.95	33	0.7375	14-18
0.96	37	0.7372	17-20

Parece que el umbral más adecuado es el que habíamos elegido, 0.95.

El siguiente parámetro a ajustar será `scale_pos_weight`, relativo al desbalanceo de las clases. Para ajustarlo utilizaremos un `GridSearch` en el que comprobaremos qué valor es mejor entre `[0.05, 0.075, 0.1, 0.15, 0.175]` (5 min para las pruebas, archivo `p3_14_sc.py`). El que mejor resultados da es 0.05, con un resultado en *training* de 0.7375 (tiempo entre 19 y 25 segundos por partición), si lo comparamos con `p3_06.py` es exactamente igual. ¿Será mejor si lo bajamos más?

Modificamos el archivo para probar valores menores `[0.05, 0.04, 0.03, 0.02, 0.01]`, el mejor parámetro sigue siendo 0.05.

Partiremos del archivo `p3_13.py` con este parámetro ya ajustado y aumentando el número de hojas a 80. El tiempo de ejecución por partición es de 75 - 100 segundos. La puntuación en *training* es 0.8184 y la puntuación en *test* es de 0.7452 (mejorando levemente la de partida).

3.17. p3_15 - Análisis sobre las variables `geo_level_1_id`

En nuestras gráficas de selección de características destacaba que las primeras variables coincidían en todas ellas, eran las llamadas `geo_level_1_id`. Estas variables representan las regiones geográficas en las que están situadas los edificios.

Investigando un poco descubrimos que Nepal está formado por 7 provincias, cada una de las cuales tiene una serie de distritos (entre 8 y 14), en total hay 77 distritos. La variable `geo_level_1_id` toma valores entre 0 y 30, no logro saber bien a qué hace referencia.

En las Figuras 9 y 10 observamos la relación entre los valores que toma esta variable y el grado de daño del edificio, que resulta ser un poco dispar.

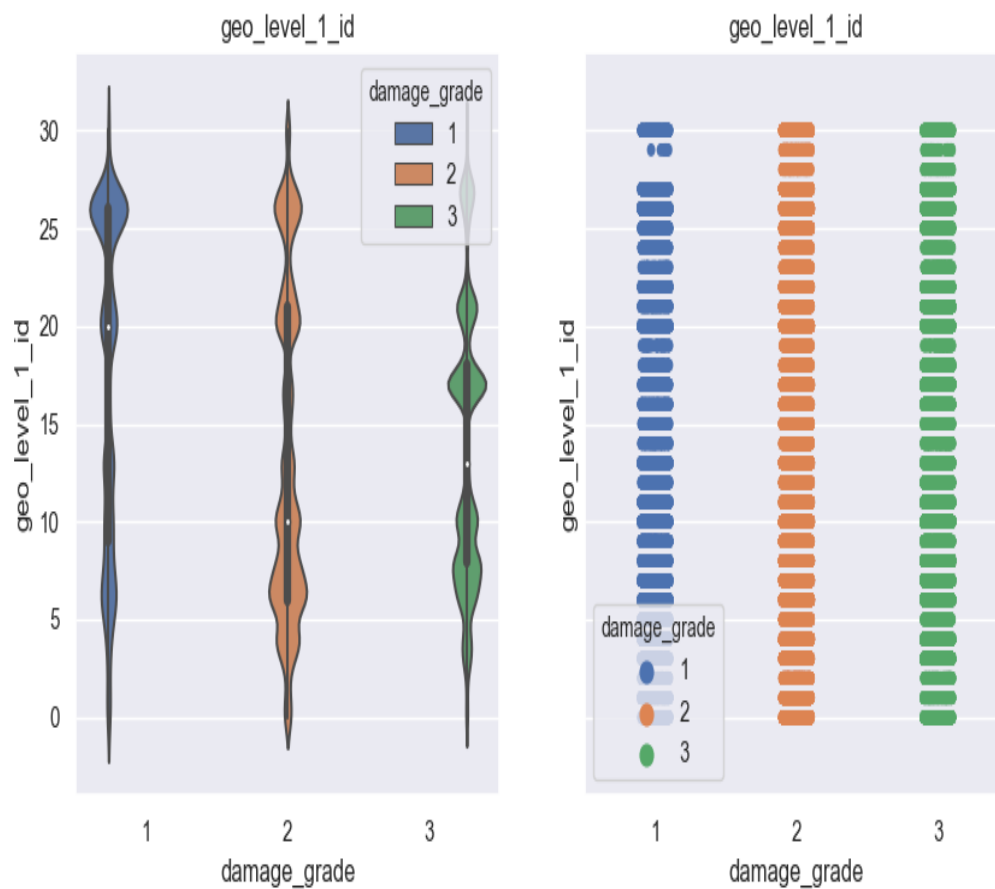


Figura 9: Relación geo_level_1_id y damage_grade.

En cualquier caso, es una variable categórica que está tomando valores numéricos, es decir, se le está induciendo cierto orden que a priori no tienen. Podemos probar a binarizar esta variable a ver si los resultados mejoran. Si hiciéramos lo mismo con geo_level_2 y 3 aumentaría demasiado el número de variables, además, en esos casos la relación entre su valor y el grado de daño es uniforme, como podemos ver en las Figuras 11 y 12.

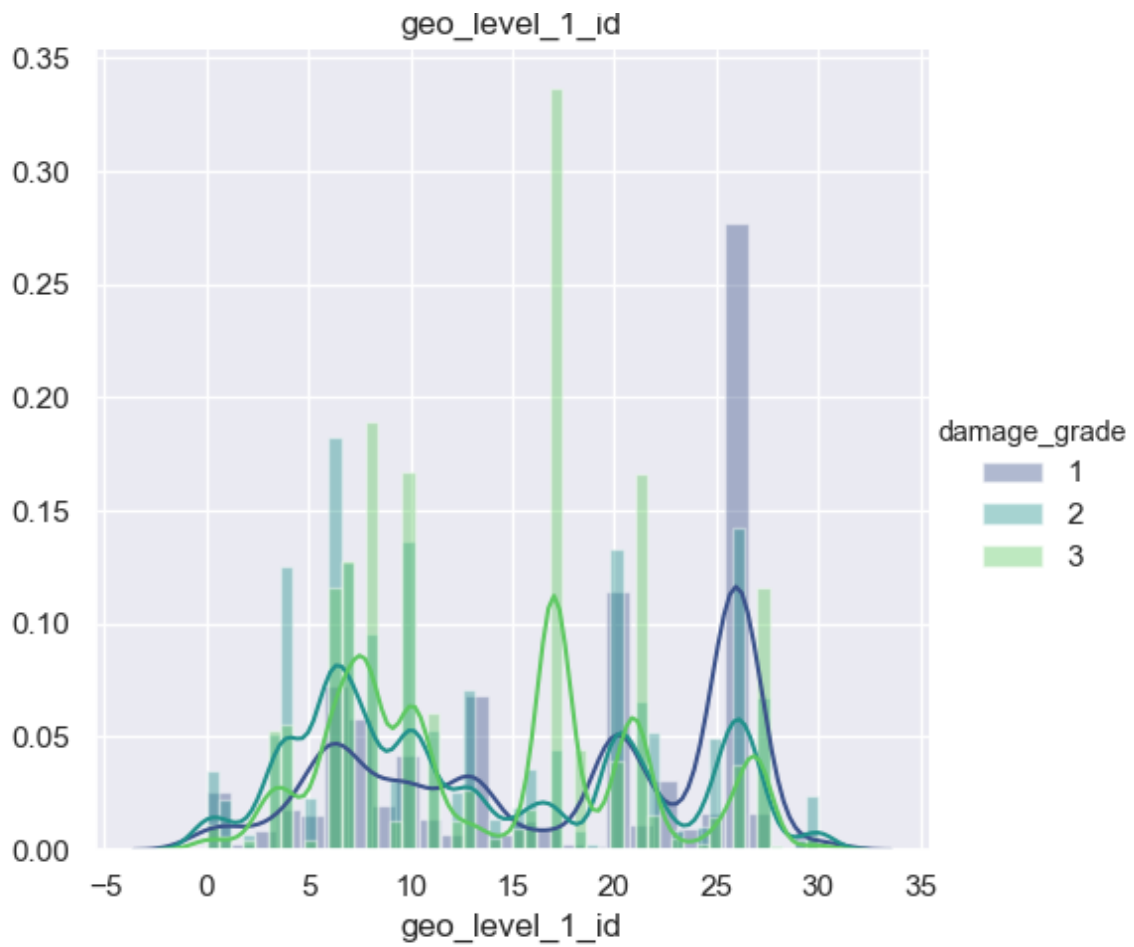
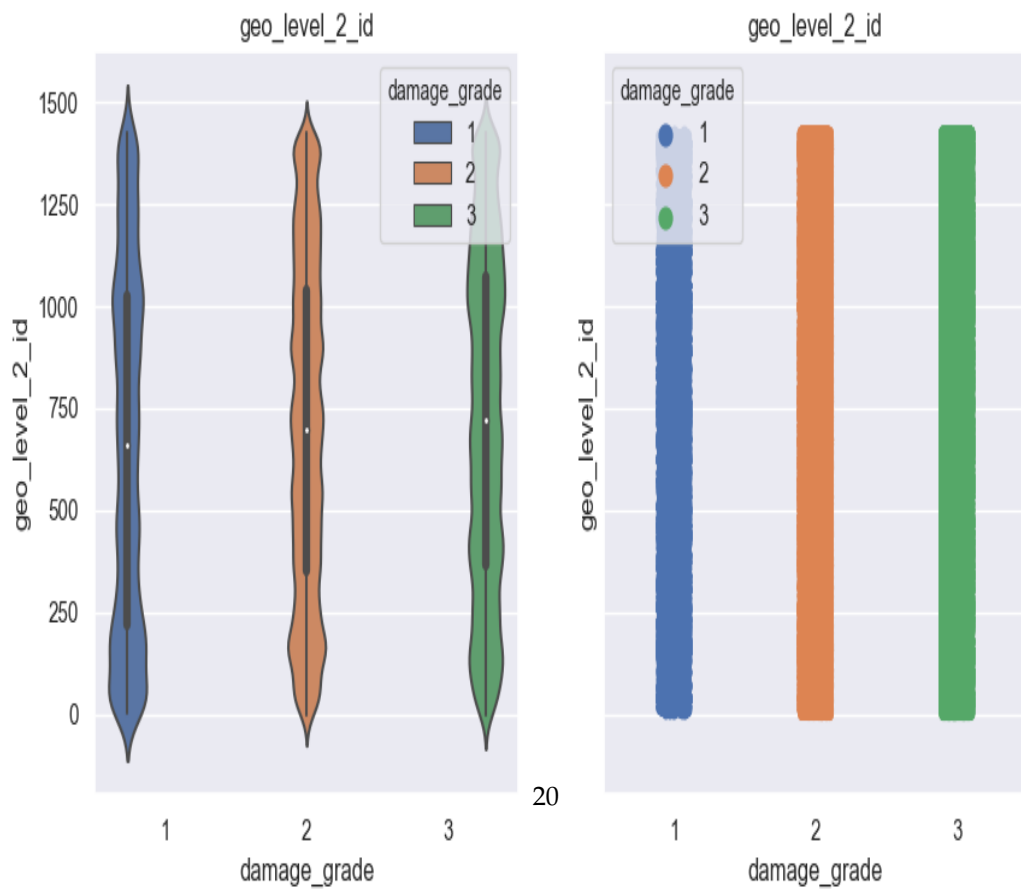


Figura 10: Relación geo_level_1_id y damage_grade.



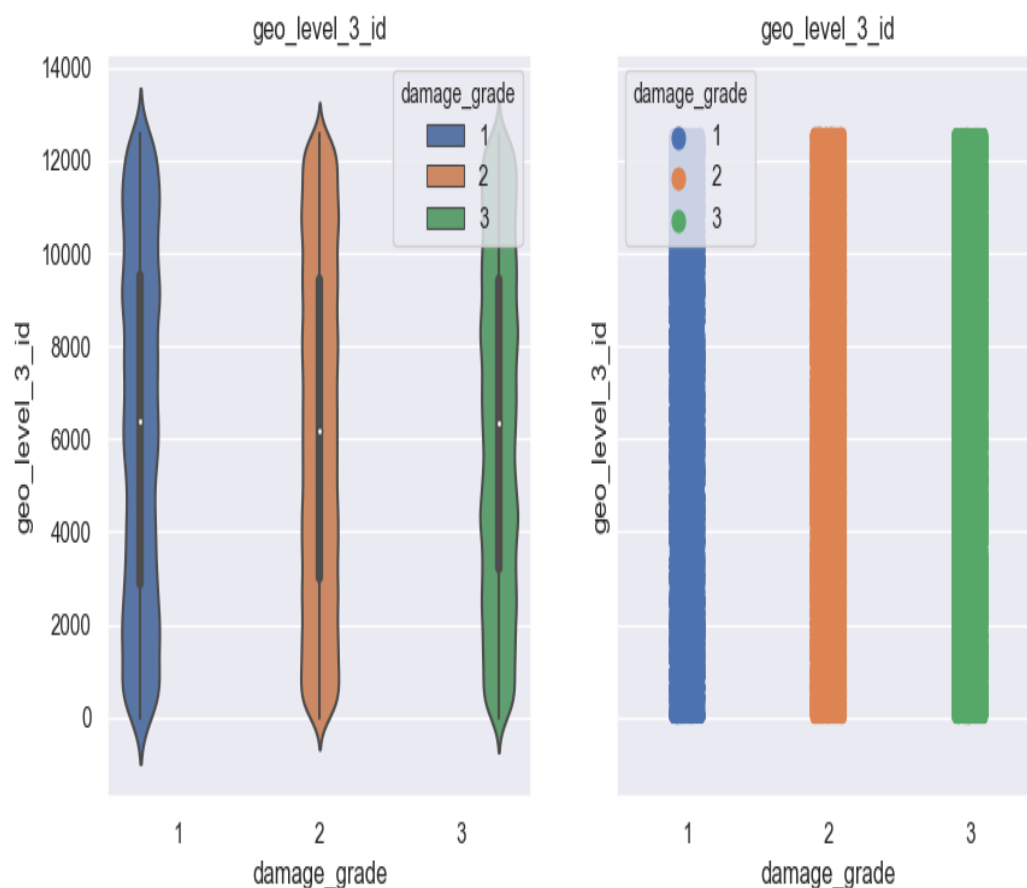


Figura 12: Relación geo_level_3_id y damage_grade.

Tras binarizar geo_level_1_id (además de las categóricas) tenemos 98 variables que al seleccionar se quedan en 41. Las pruebas iniciales se hacen partiendo de p3_06.py por evitar tiempos de ejecución demasiado elevados. Conseguimos un resultado en *training* de 0.7291, menor que el 0.7375 obtenido en el intento 06 con tiempos entre 17 y 22 segundos.

Probamos a realizar lo mismo, aumentando el umbral de selección a 0.97, en este caso nos quedamos con 52 variables, los tiempos de ejecución van de 18 segundos hasta 23. El resultado en *training* es ahora de 0.7337 (sigue siendo inferior al de partida).

Un umbral de 0.99, nos deja con 70 variables y tiempos entre 20 y 24 segundos por partición. Siendo los resultados en entrenamiento otra vez inferiores a los de partida: 0.7369.

Si nos quedamos con las 98 variables los tiempos van desde 19 hasta 24 y el resultado es de 0.7361 (menor que en el caso anterior).

No conseguimos una mejora, con lo que concluimos que al numerizar igual sí que estaban ordenadas según algún criterio (situación, proximidad, ...).

Como no nos queda mucho más tiempo hoy, probamos a aumentar el número de hojas para tratar de mejorar el intento 14. Tarda unos 80 segundos por partición. El resultado en *training* es de 0.8259 y en *test* de 0.7448 (un poco peor que el intento 14 de partida). Debe de ser que se está produciendo sobreajuste.

3.18. p3_16 - Ajuste de *Lightgbm*

Hemos visto que los parámetros `num_leaves` y `n_estimators` son determinantes para la buena ejecución del algoritmo *Lightgbm*, por ello, realizaremos un ajuste de parámetros específico a ver qué combinación de parámetros proporciona mejores resultados. Probaremos los valores de los parámetros: `num_leaves` $\in \{50, 60, 70, 80, 90, 100\}$, `n_estimators` $\in \{200, 300, 400, 500, 600, 700, 800, 900, 1000\}$, `scale_pos_weight` $\in \{0.05, 0.1\}$. Tras 142.4 minutos de ejecución, se llega a la conclusión de que los mejores parámetros son: `n_estimators` = 900, `num_leaves` = 70, `scale_pos_weight` = 0.05. En torno a los 70 segundos de ejecución por partición, consigue un resultado en entrenamiento de 0.8061.

Si probamos la binarización de `geo_level_1_id` (`p3_16_geo.py`) con estos parámetros se queda con 41 variables, tiempo por partición entre 65 y 80 segundos. El resultado en *training* es de 0.7995, un poco menor que sin la binarización.

Así, subimos los resultados de la ejecución sin binarizar esta variable obteniendo un resultado en *test* de 0.7443 (menor que el obtenido en p3.14).

3.19. p3_17 - *XGBoost*

Pasamos ahora a probar otro algoritmo. Sabemos que *Lightgbm* tiene la ventaja de ser muy eficiente, pero no es el más potente, así, probaremos con *XGBoost* [21], a ver si obtiene resultados similares o incluso mejores. Utilizaremos el preprocesado anterior.

En primer lugar, nos encontramos con un error inesperado. El algoritmo trata de usar la GPU y por algún motivo no puede. Para solucionarlo añadimos los parámetros `predictor` = `cpu_predictor` y `n_gpus` = 0.

Ejecutamos el algoritmo con la siguiente configuración de parámetros: `predictor` = `'cpu_predictor'`, `n_gpus` = 0, `n_estimators` = 200, `eta` = 0.3, `max_depth` = 6, `scale_pos_weight` = 1. Con un tiempo que ronda los 300 segundos por iteración, conseguimos un resultado de 0.7308 en entrenamiento.

Probamos a tratar el imbalanceo de las clases. Por un lado, variando el parámetro `scale_pos_weight`, probamos a darle el valor 0.5 y 1 pero no afecta a los resultados. Por otro lado, utilizando el parámetro `max_delta_step` que vemos que afecta al desbalanceo de clases. Ejecutamos el algoritmo con los parámetros: `predictor` = `'cpu_predictor'`, `n_gpus` = 0, `n_estimators` = 200, `eta` = 0.3, `max_depth` = 6, `max_delta_step` = 7. El tiempo por ejecución oscila entre 296 segundos y 319. El resultado en *training* exactamente igual que cuando no estaba este parámetro.

Probamos a aumentar un poco el resto de parámetros, disminuyendo el valor `eta` para evitar el sobreajuste. `n_estimators` = 400, `eta` = 0.1, `max_depth` = 8. Conseguimos una puntuación en entrenamiento de 0.7913 que se tradujo en 0.7425 en *test*.

3.20. p3_18 - Ajuste *XGBoost*

Probamos a aumentar todavía más los parámetros: `n_estimators` = 700, `eta` = 0.1, `max_depth` = 10. El resultado obtenido en entrenamiento es de 0.8691 y en *test* de 0.7457. El aumento en el número de estimadores y profundidad, provocó una mejora en los resultados.

3.21. p3_19 - *Stacking*

Ahora que ya hemos probado varios algoritmos es el momento de utilizar el algoritmo recomendado por los profesores de la asignatura: *stacking* [12], que consiste en utilizar varios clasificadores para mejorar la clasificación final.

A partir de aquí, como los algoritmos llevan tanto tiempo, dejamos de realizar la validación cruzada.

En primer lugar, probaremos los 3 algoritmos con los parámetros que mejores resultados dieron. Tras obtener varias veces problemas de memoria, tenemos que ir disminuyendo los parámetros de los algoritmos hasta conseguir ejecutarlos. Con una advertencia sobre la convergencia, consigue un resultado en entrenamiento de 0.7862 tras 984 segundos de ejecución (unos 16 minutos). Esto se tradujo en un resultado en *test* de 0.7416.

3.22. p3_20 - *Stacking*

Probamos ahora a quitar el clasificador *RandomForest*, ya que consiguió unos resultados ligeramente inferiores a los de *XGBoost* y *Lightgbm* en las pruebas de los clasificadores por separado.

En esta ocasión, sí que pudimos mantener los parámetros que dieron mejores resultados de ambos clasificadores. En un tiempo cercano a una hora conseguimos un resultado en entrenamiento de 0.8436 que se convirtió en 0.7468 en *test* (mejor resultado hasta el momento).

Puebo a ejecutar el mismo algoritmo eliminando la selección de variables, el resultado en *training* es de 0.8421, en un tiempo de 14452.72 segundos (cuatro horas). Da una puntuación menor en *training* a pesar de haber estado entrenando durante mucho más rato. Por lo pronto no subiré estos resultados (estarán en p3_23).

3.23. p3_21 - *Stacking*

Conseguimos evitar los errores de memoria eliminando los parámetros *n_jobs* (siguiendo el consejo de un compañero). Así, podemos probar el *Stacking* con los tres algoritmos en los parámetros que mejor resultado dieron. En un tiempo de 2967 segundos (50 minutos), consiguió un resultado en *training* de 0.8427 (ligeramente menor que sin *RandomForest*). Al subir el *submission* correspondiente obtenemos una puntuación de 0.7469 (0.0001 más que sin *RandomForest*).

3.24. p3_sampling - Muestreo aleatorio

Probamos a realizar un muestreo aleatorio, a ver si disminuyen los tiempos de ejecución y mejoran los resultados. Partimos del archivo p3_6.py eliminando la selección de variables. Archivo p3_sampling.py.

Seleccionando 3/4 de las variables, 195450 instancias, los tiempos de ejecución son de 14-18 segundos por partición y obtenemos un resultado en *training* de 0.7388 (una pequeña mejora respecto al 0.7375 de partida).

Añadiendo la selección de características, los tiempos son de entre 11-13 segundos por partición. El resultado en *training* del total de instancias es de 0.7397, que mejora un poco el resultado inicial.

El muestreo podría resultar más interesante en los algoritmos más lentos. Sin embargo, un muestreo aleatorio puede no ser de mucho interés, ya que muchos algoritmos funcionan mejor con mayor cantidad de instancias (aunque tarden más de lo que nos gustaría), convendría quitar aquellas que por algún motivo estén generando algún tipo de ruido y empeorando el modelo.

3.25. p3_22 - Detección de anomalías con PyOD

Se considera una anomalía a quel punto que difiere demasiado del resto de observaciones del conjunto. Detectarlas y eliminarlas puede producir mejores resultados, ya que generaremos un modelo que no trate de adecuarse a estas (eliminar las anomalías no siempre resulta en mejores resultados, puede que haya puntos diferentes en el conjunto y debamos adaptarnos a ellos).

Utilizaremos PyOD para detectarlas [19]. En particular, el clasificador Isolation Forest, pues tiene un buen rendimiento en datos multidimensionales. En el archivo `p3_anomalías.py` está la adaptación del archivo `p3_06.py` en el que se cambia la selección de variables y binarización de variables categóricas por la eliminación de anomalías. Tras seleccionar 247571 instancias (13030 anomalías). El resultado en *training* es de 0.7392 (ligeramente superior al 0.7375 de partida) con tiempos de ejecución entre 17 y 20 segundos por partición.

Añadimos la binarización de las variables categóricas antes de la eliminación de anomalías, pues antes de aplicar la binarización, en las primeras pruebas con *Lightgbm*, conseguimos resultados del orden de 0.73 en *training* que en *test* luego no llegaron a superar el 0.7. Añadiendo el código correspondiente, actualizado en el archivo `p3_22.py`, pasamos a probarlo. Con unos tiempos de 20 segundos por partición, el resultado en *training* es de 0.7387, se tradujo en un *f1-score* en *test* de 0.7240 (que tenemos que comparar con 0.7253), un ligero empeoramiento.

3.26. p3_23 - *Stacking* sin selección de variables

Para no desperdiciar la última prueba del día, subiré los resultados comentados en la Sección 3.22.

Son los resultados de un *stacking* de *XGBoost* y *LightGBM* eliminando la selección de variables. Esto da un resultado en *training* de 0.8421 y en *test* de 0.7482. Aunque la selección de variables quitara bastante tiempo de ejecución, los resultados en *test* se ven favorecidos, dando la puntuación más alta hasta el momento.

3.27. p3_24 - *Stacking* sin selección de variables

Viendo que sin selección de variables los resultados del *Stacking* son ligeramente mejores, decidimos probar también el *Stacking* con *RandomForest*. Tras un tiempo de ejecución de 4775.06 segundos (1 hora y 20 minutos aproximadamente), consigue un *f1-score* de 0.8498 (superior al 0.8427 de `p3_21.py` y al 0.8421 de `p3_23.py`). El resultado en *test* es exactamente igual que el de la ejecución anterior, 0.7482 (aunque el archivo *submission* correspondiente difiriera del anterior en 556 elementos).

3.28. p3_25 - Ajuste de *StackingClassifier*

Partiendo de `p3_23.py` aumentamos un poco los parámetros de profundidad, a ver si con esto conseguimos mejorar los resultados. En un tiempo de 9187 segundos (casi 3 horas), consigue un resultado en *training* de 0.8604 (mayor que el 0.8421 de partida). Empeora un poco, posiblemente por problemas de sobreajuste.

3.29. p3_26 - Ajuste de *StackingClassifier*

Partimos ahora del archivo `p3_24.py`, aumentando los parámetros de profundidad de *RandomForest*. El resultado conseguido tras 4811 segundos de ejecución (menos de hora y media) en *training* es de 0.8627, en *test* es de 0.7487.

3.30. p3_27 - Ajuste de *StackingClassifier*

Partiendo del archivo `p3_27.py` modificamos algunos parámetros a ver si conseguimos mejorar el resultado. En un tiempo de 5462 segundos (una hora y media) consigue una puntuación en *training* de 0.8653, que se traduce en *test* a 0.7488 (mejor resultado conseguido).

3.31. p3_28 - Ajuste de *StackingClassifier*

Partimos del archivo anterior y modificamos levemente los valores de los parámetros. Obtenemos un resultado en entrenamiento de 0.8720, que se transforma en una puntuación en *test* de 0.7479.

3.32. Pruebas fallidas

3.32.1. `categorical_features` de *Lightgbm*

Tras observar los parámetros de *Lightgbm* en [10], nos damos cuenta de que tiene una opción en la que el propio algoritmo trata este tipo de variables. Partiendo de `p3_06.py` adaptamos el algoritmo para usar el parámetro `categorical_features`, el código correspondiente se encuentra en `p3_categorical.py`, pero no conseguimos ajustarlo debido a varios errores en la ejecución.

3.32.2. Reducción de la dimensionalidad con UMAP

Por recomendación de un compañero, pruebo a utilizar UMAP [20] para reducir la dimensionalidad. Tras instalar el paquete mediante `pip install umap-learn`, pruebo su funcionamiento de forma básica en `p3_umap.py` (usando a posteriori el algoritmo *Lightgbm*).

Tras largo tiempo de ejecución (media hora por lo menos) consigue exactamente los mismos resultados que `p3_00.py`.

3.32.3. Selección de variables mediante información mutua

Partiendo de `p3_04.py`, probamos a utilizar como criterio de selección la información mutua, que representa el grado de dependencia entre dos variables, la información que una contiene sobre la otra. Si vale 0 es porque las variables son independientes. Encontramos el código ejecutado en `p3_mi.py`, notamos que tarda mucho, puede ser porque basa sus cálculos en herramientas del estilo a *K-nn* que con el alto número de instancias es excesivamente lento.

3.32.4. Selección de variables mediante Boruta

Para terminar los experimentos de selección de variables con *Lightgbm* utilizaremos Boruta, que pretende seleccionar las variables más importantes sin que tengamos que fijar de antemano el número de variables a utilizar.

Partiendo del código proporcionado por el profesor de prácticas utilizaremos un *RandomForest* para seleccionar las características más importantes. Nuestra primera impresión es negativa debido al alto tiempo de cómputo para seleccionar.

Tras 2700 segundos (45 minutos) de ejecución, me da un error. Lo abandono pensando que fue una mala idea utilizar *RandomForest* con tantas variables.

3.32.5. Visualización con TSNE

Antes de atacar el problema del ruido, descubro la existencia de TSNE [16] una herramienta de scikit-learn para reducir la dimensionalidad y poder visualizar los datos. Añadimos las líneas necesarias a `visualization.py` para obtener la representación correspondiente en dos dimensiones [4]. En la Figura 13 vemos dicha representación.

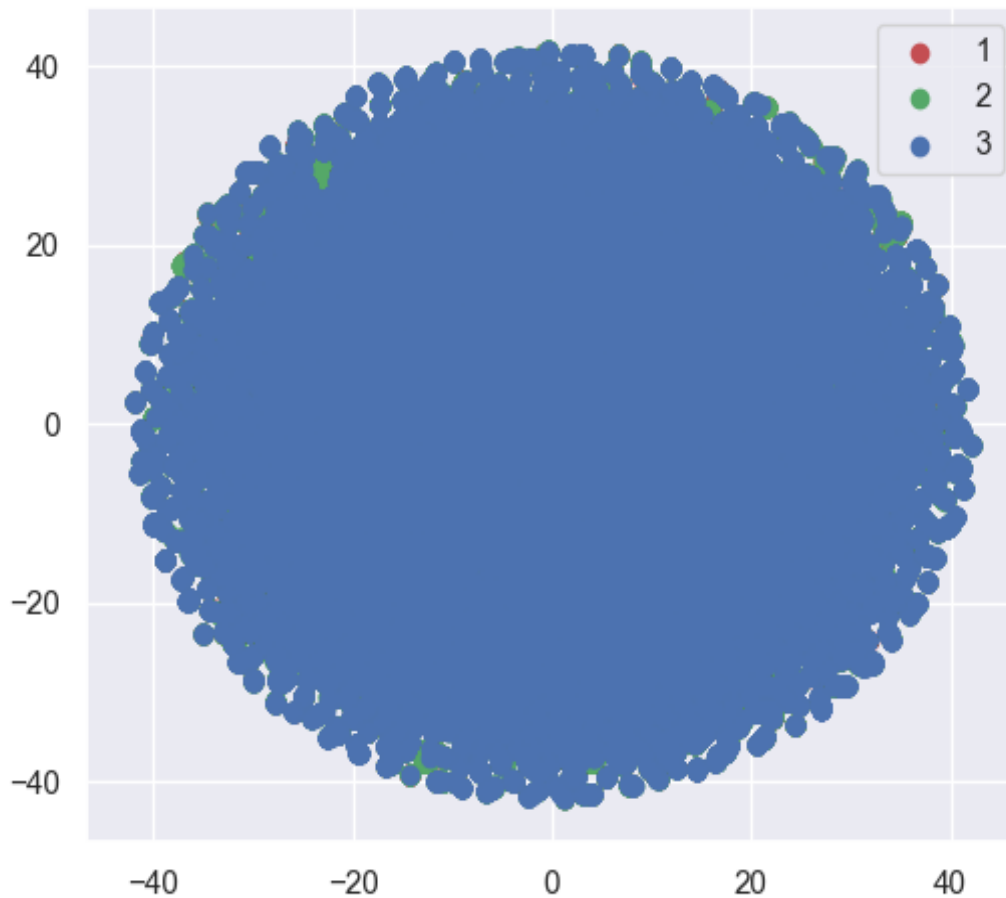


Figura 13: Visualización de los datos usando TSNE en 2 dimensiones.

Como no podemos obtener ninguna conclusión sobre la representación en dos dimensiones, probamos a dibujarla en tres dimensiones. Tras largo tiempo de cómputo obtenemos un error que no sabemos solucionar tras una corta búsqueda.

3.32.6. p3_naive - NaiveBayes

Decidimos realizar algunas pruebas con ComplementNB, ya que [17] dice que es especialmente adecuado para conjuntos de datos desbalanceados. Los resultados en *training* están alrededor de 0.2, incluso tratando de ajustar algún parámetro. Así, desistimos de profundizar más en el ajuste de este clasificador. Podemos encontrar el código de prueba en `p3_naive.py`.

3.32.7. Detección de anomalías + Stacking

Partiendo del preprocesado del *script* `p3_22.py`, lo adaptamos al *Stacking* de *Lightgbm* y *XGBoost*. Partimos del archivo `p3_20.py` y cambiamos el preprocesado al recién probado. Sin embargo, esto genera un error que no fuimos capaces de arreglar inmediatamente. Tras ver los resultados de `p3_22.py` al subirlo a la web, decidimos no seguir investigando por este camino.

Referencias

- [1] 1.11. Ensemble methods — scikit-learn 0.22 documentation. URL: <https://scikit-learn.org/stable/modules/ensemble.html#forest> (Último acceso 24-12-2019).
- [2] 1.13. Feature selection — scikit-learn 0.22 documentation. URL: https://scikit-learn.org/stable/modules/feature_selection.html (Último acceso 23-12-2019).
- [3] 3.2.4.3.1. `sklearn.ensemble.RandomForestClassifier` — scikit-learn 0.22 documentation. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier> (Último acceso 24-12-2019).
- [4] 3.6.10.5. *tSNE to visualize digits* — Scipy lecture notes. URL: https://scipy-lectures.org/packages/scikit-learn/auto_examples/plot_tsne.html (Último acceso 28-12-2019).
- [5] DrivenData. *Richter's Predictor: Modeling Earthquake Damage*. DrivenData. URL: <https://www.drivendata.org/competitions/57/nepal-earthquake/page/136/> (Último acceso 14-12-2019).
- [6] Yang Liu. *Encoding Categorical Features*. Medium. 20 de sep. de 2018. URL: <https://towardsdatascience.com/encoding-categorical-features-21a2651a065c> (Último acceso 25-12-2019).
- [7] *Model for Nepal Earthquake Damage*. URL: <https://kaggle.com/jaylaksh94/model-for-nepal-earthquake-damage> (Último acceso 14-12-2019).
- [8] `pandas.get_dummies` — pandas 0.25.3 documentation. URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html (Último acceso 22-12-2019).
- [9] *Parameters* — LightGBM 2.3.2 documentation. URL: https://lightgbm.readthedocs.io/en/latest/Parameters.html#weight_column (Último acceso 19-12-2019).
- [10] *Parameters* — LightGBM 2.3.2 documentation. URL: https://lightgbm.readthedocs.io/en/latest/Parameters.html#categorical_feature (Último acceso 31-12-2019).
- [11] *Richter's Predictor*. URL: <https://kaggle.com/alekseyeliseev/richter-s-predictor> (Último acceso 25-12-2019).
- [12] `sklearn.ensemble.StackingClassifier` — scikit-learn 0.22 documentation. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html> (Último acceso 28-12-2019).
- [13] `sklearn.feature_extraction.DictVectorizer` — scikit-learn 0.22 documentation. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html (Último acceso 25-12-2019).
- [14] `sklearn.feature_selection.SelectKBest` — scikit-learn 0.22 documentation. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest (Último acceso 31-12-2019).
- [15] `sklearn.feature_selection.VarianceThreshold` — scikit-learn 0.22 documentation. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.VarianceThreshold.html (Último acceso 31-12-2019).
- [16] `sklearn.manifold.TSNE` — scikit-learn 0.22 documentation. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html> (Último acceso 28-12-2019).
- [17] `sklearn.naive_bayes.ComplementNB` — scikit-learn 0.22 documentation. URL: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.ComplementNB.html (Último acceso 30-12-2019).
- [18] *Starter: Richter's Predictor: Modeling* e7f51e9e-e. URL: <https://kaggle.com/kerneler/starter-richter-s-predictor-modeling-e7f51e9e-e> (Último acceso 14-12-2019).
- [19] *Tutorial on Outlier Detection in Python using the PyOD Library*. Analytics Vidhya. 14 de feb. de 2019. URL: <https://www.analyticsvidhya.com/blog/2019/02/outlier-detection-python-pyod/> (Último acceso 29-12-2019).
- [20] *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction* — umap 0.3 documentation. URL: <https://umap-learn.readthedocs.io/en/latest/> (Último acceso 31-12-2019).

- [21] *XGBoost Parameters* — *xgboost 1.0.0-SNAPSHOT documentation*. URL: <https://xgboost.readthedocs.io/en/latest/parameter.html> (Último acceso 27-12-2019).