Práctica 3:
Competición en DrivenData

Curso 2019/2020

SOFÍA ALMEIDA BRUNO
sofialmeida@correo.ugr.es

Grupo IN 2
Jueves 9:30-10:30

Índice

1. Captura de pantalla de Submissions

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	-	Lightgbm	`objective = 'regression_l1', n_estimators = 200, n_jobs = 2`
01	19/12/2019 11:18:10 UTC	356	0.7358	0.6874	-	Lightgbm	`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	-	Lightgbm	`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	Lightgbm	`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	Lightgbm	`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`, $k = 35$	Lightgbm	`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	Lightgbm	`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	RandomForest	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	RandomForest	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	RandomForest	max_depth = 40, max_features = 'sqrt', n_estimators = 500
10	2019/12/25 11:42:49 UTC	239	0.7375	0.6823	OneHotencoder para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	Lightgbm	objective = 'regression_l1', n_estimators = 200, n_jobs = 2, num_leaves = 45, scale_pos_weight = 0.1
11	2019/12/25 12:40:41 UTC	160	0.7693	0.7388	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	Lightgbm	objective = 'regression_l1', n_estimators = 500, n_jobs = -1, num_leaves = 55, scale_pos_weight = 0.1
12	2019/12/25 12:52:50 UTC	148	0.7855	0.7412	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	Lightgbm	objective = 'regression_l1', n_estimators = 700, n_jobs = -1, num_leaves = 60, scale_pos_weight = 0.1
13	2019/12/26 15:35:06 UTC	117	0.8070	0.7444	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	Lightgbm	objective = 'regression_l1', n_estimators = 1000, n_jobs = -1, num_leaves = 65, scale_pos_weight = 0.1

14	2019/12/26 16:34:32 UTC	107	0.8184	0.7452	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	Lightgbm	objective = 'regression_l1', n_estimators = 1000, n_jobs = -1, num_leaves = 80, scale_pos_weight = 0.05
15	2019/12/26 18:24:17 UTC	107	0.8259	0.7448	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	Lightgbm	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	Lightgbm	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	XGBoost	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	0.7425	get_dummies para todas las variables categóricas. Selección de variables con VarianceThreshold, umbral 0.95	XG-Boostn_estimators = 700, eta = 0.1, max_depth = 10	

3. Diario de pruebas

3.1. p3_00

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 un algoritmo de boosting.

3.2. p3_01

3.2.1. Análisis exploratorio de los datos

Antes de decidir qué hacer a continuación debemos conocer cierta información sobre los datos. Podemos pensar que necesitamos algún tipo de preprocesado, pues es lo habitual en este tipo de problema, pero sin conocer exactamente cómo son nuestros datos, si tienen o no ruido, la cantidad de valores perdidos, correlación entre las variables, ... no podremos decidir cómo enfocar el preprocesado ni qué necesidades tiene el 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 https://www.kaggle.com/kerneler/starter-richter-s-predictor-modeling-e7f51e9e-e observamos en la Figura 1 la distribución de las clases.

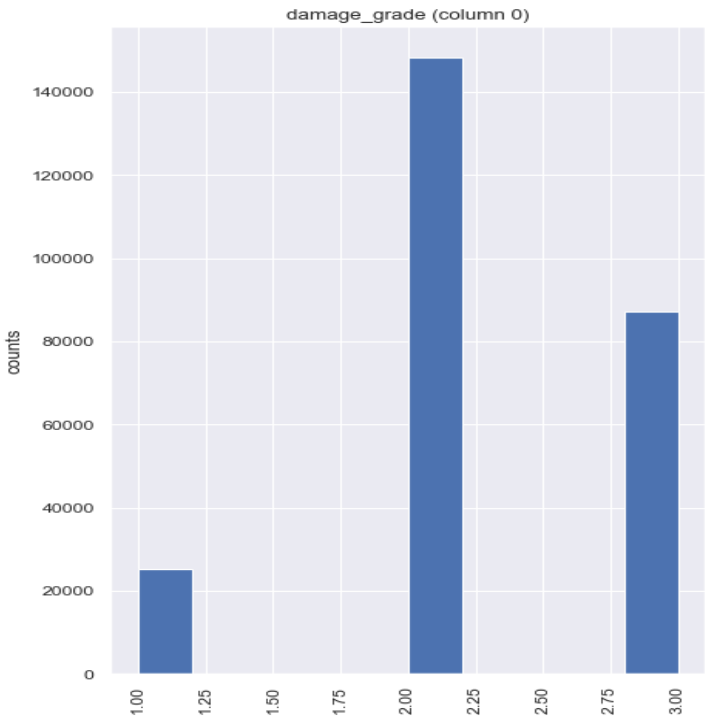

Figura 1: Tamaño de las clases

Nuestras clases 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 esta situación, utilizar técnicas específicas para paliarlo.

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 2.

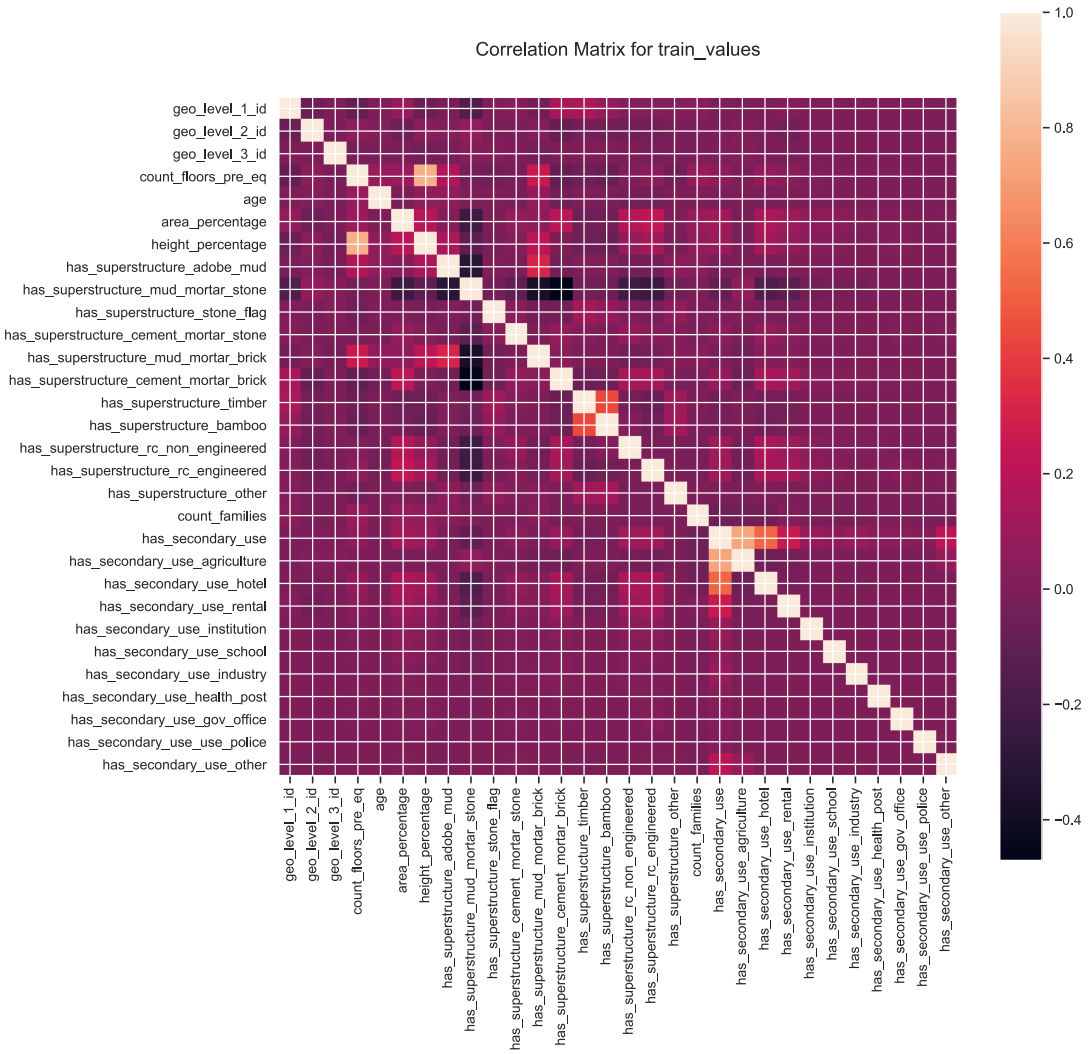

Figura 2: Matriz de correlación

Mediante `data_x.info()` conocemos que de las 38 variables 30 son numéricas y 8 categóricas, accedemos a la descripción del problema en para conocer cuántos posibles valores toman las variables categóricas. Toman 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, podemos ahorrarnos la imputación de valores perdidos.

3.2.2. Ajuste de Lightgbm

Para tratar de conseguir mejores resultados tenemos, a priori, dos caminos: preprocesar los datos, mejorar el algoritmo. Para mejorar el algoritmo hay, a su vez, dos opciones: elegir el algoritmo y ajustar sus parámetros.

En primer lugar trataremos de ajustar los parámetros de Lightgbm a ver si mejora el resultado. Usando el código de ejemplo `ejemplo_ds_avanzado` nos centraremos en los parámetros `feature_fraction` (que tomará ..., `learning_rate`, `num_leaves`. 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 sobre ajuste?

3.3. p3_02

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 sobre ajuste 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.

Para paliar el desbalanceamiento de las clases se pueden usar dos parámetros: `is_unbalance` o `scale_pos_weight`. Probaremos a configurar ambos parámetros a ver con cuál conseguimos realmente mejor resultado.

Así, 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 se asumía que no había desbalanceo.

Probamos con la otra opción: variamos el peso de la clase 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` que consiguió una puntuación de 0.7294 en entrenamiento y 0.6894 al realizar el envío (mejorando la mejor solución obtenida hasta el momento).

3.4. p3_03

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. Como observamos que eran 8 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, mediante la función `get_dummies`. En el archivo `p3_03.py` encontramos el código correspondiente a esta ejecución. Pasamos de 38 a 68 características.

Conseguimos una puntuación en test de 0.7339 y en prueba de 0.7227.

3.5. p3_04

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 utlizamos Lightgbm con 0.1 para `scale_pos_weight` añadimos un método de selección de variables al preprocesado.

Nuestro primer intento consiste en eliminar las variables con varianza baja. 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 adaptando...en la Figura 3 podemos ver el número de veces que una variable se usa en un modelo.

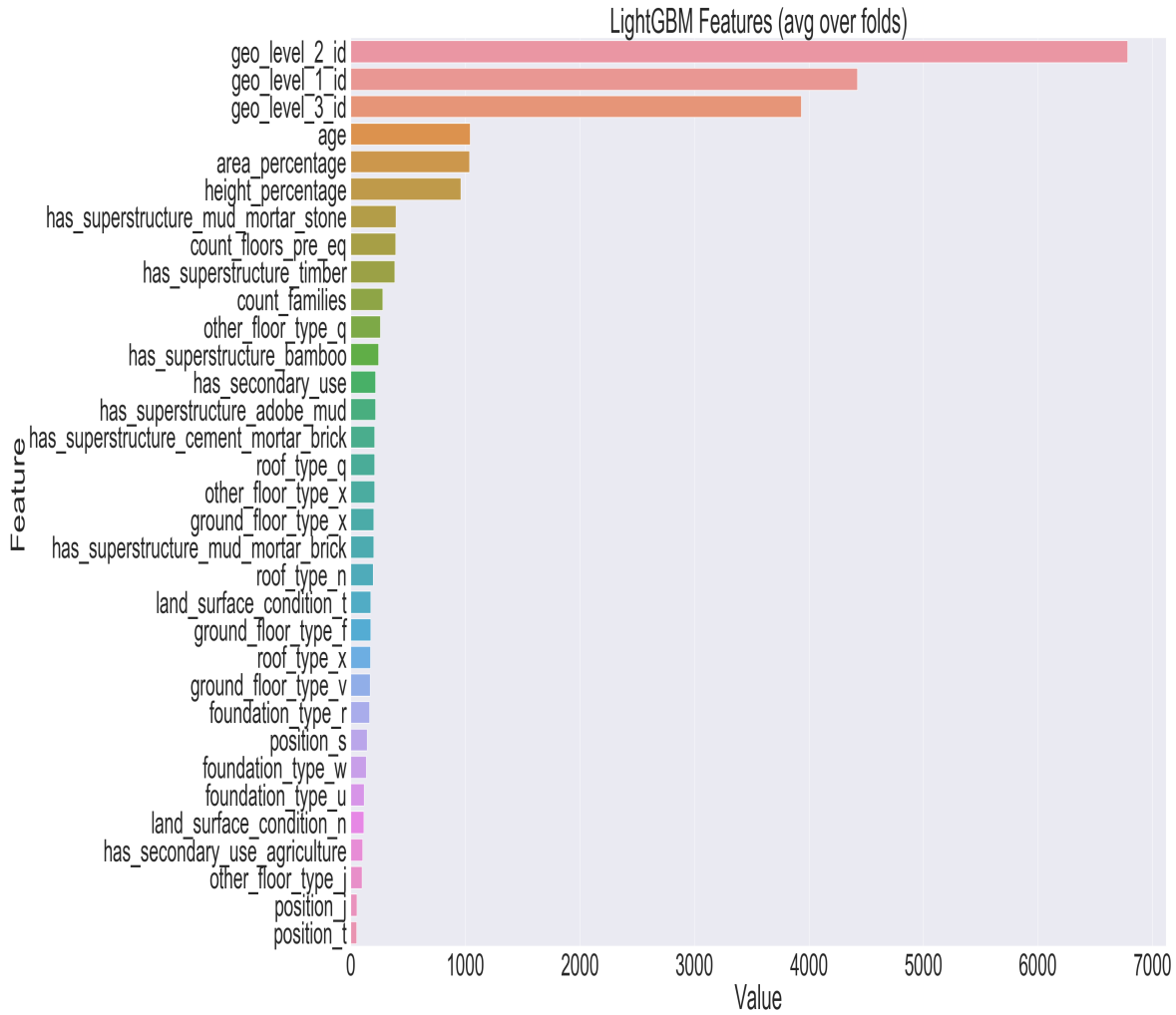

Figura 3: Importancia de las variables

3.6. p3_05

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 los k mejores

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 que las cuatro primeras variables son las mismas que seleccionando según la varianza, pero a partir de ahí varían.

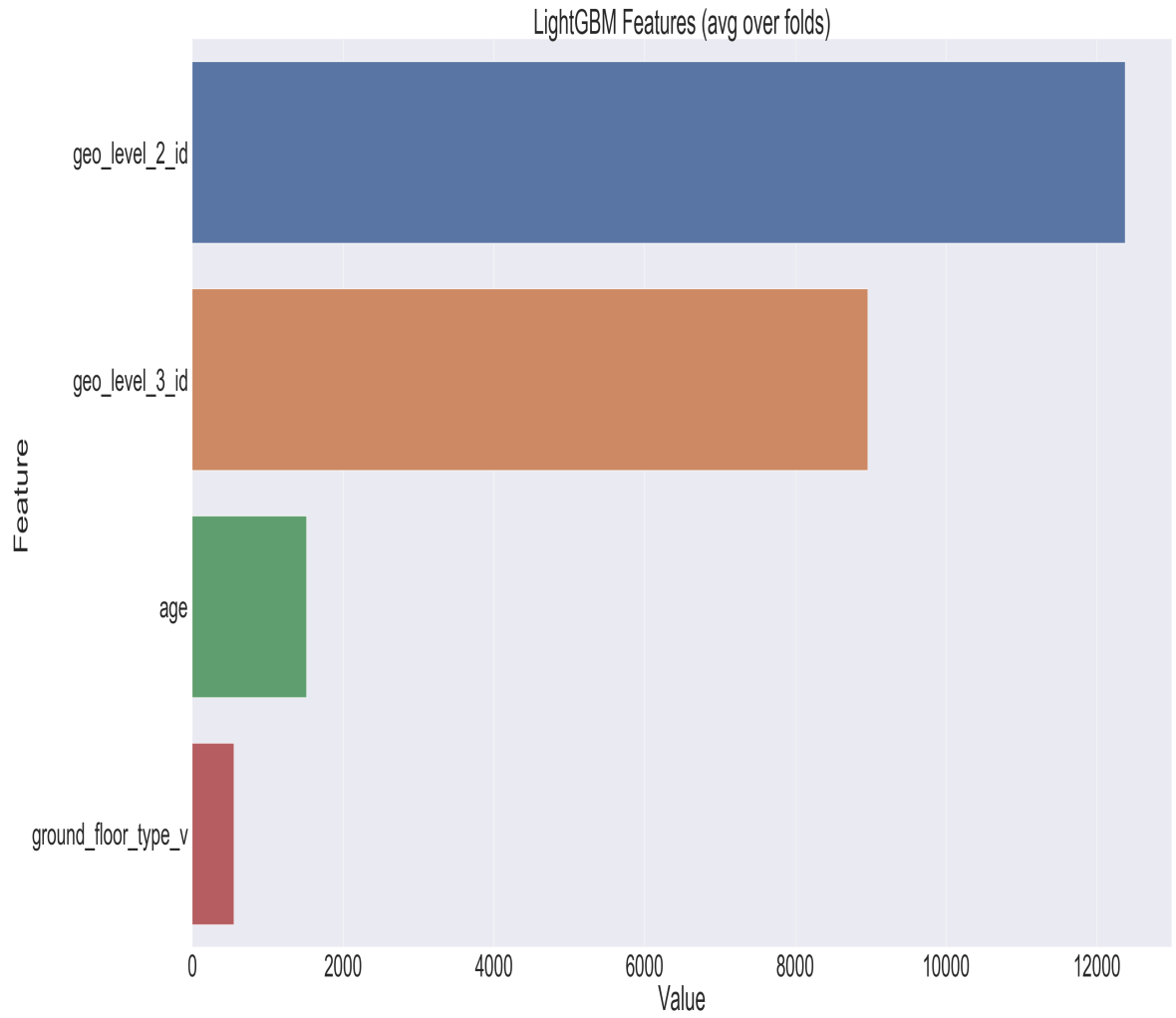

Figura 4: Importancia de las variables tras seleccionar las 10 mejores.

Si nos quedamos solo con 4 variables no selecciona estas cuatro 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 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 observamos las variables más utilizadas para clasificar.

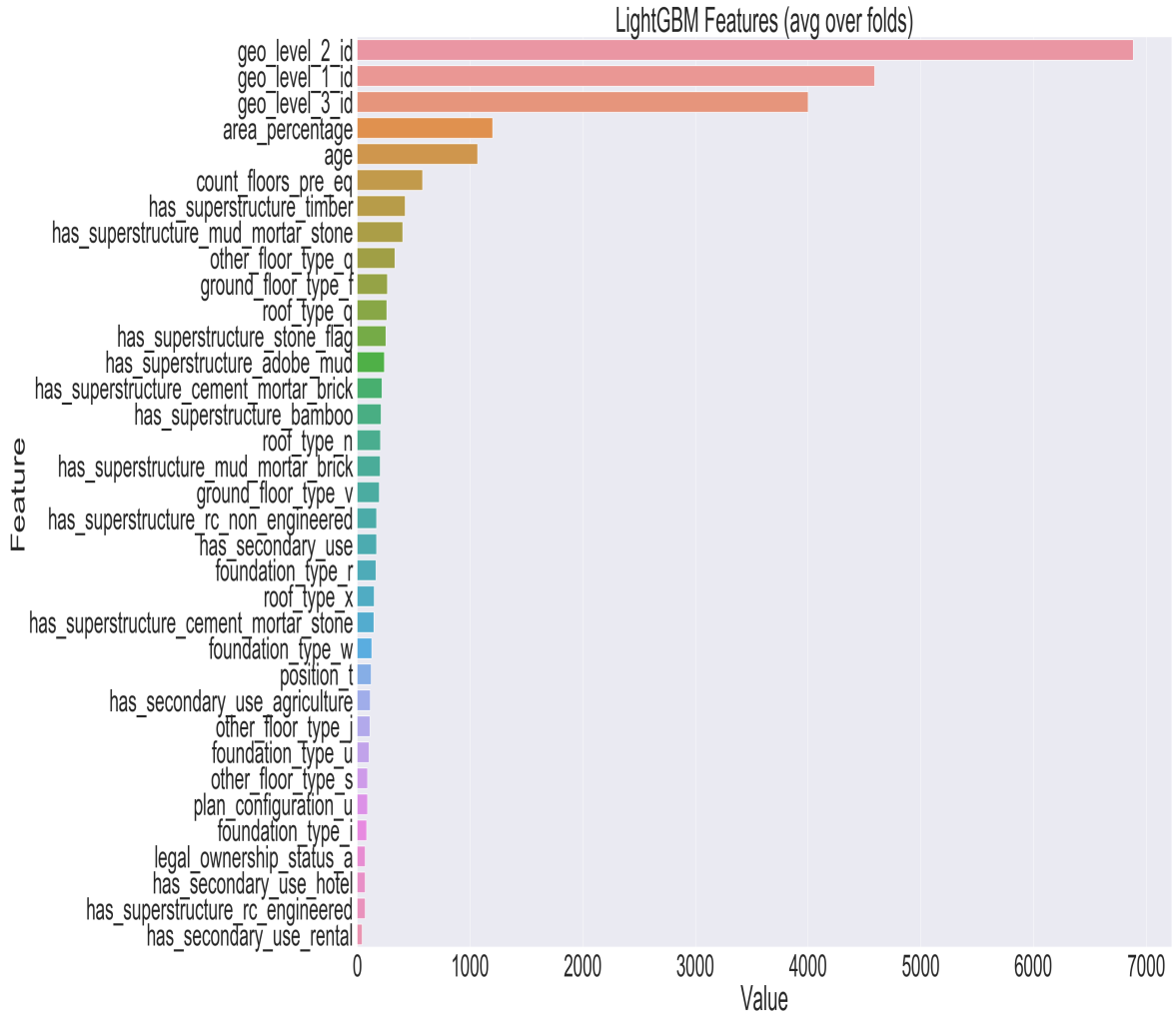

Figura 5: Importancia de las variables tras seleccionar las 35 mejores.

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.

3.7. p3_06

Tras perder el rato con algunas pruebas fallidas, decido no desperdiciar la última subida del día tratando de mejorar aumentando el número de hojas a 45. Puntuación en training: 0.7375, puntuación en test: 0.7253.

3.8. p3_07

Pasamos ahora a probar otro algoritmo: RandomForest.

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. Unos 8 minutos para el GridSearch (y eso que no puse valores muy grandes... `max_depth: [10, 20]`, `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.

3.9. p3_08

Para tratar de paliar el sobreaprendizaje, siguiendo la recomendación de 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 menor sobre ajuste lo subimos y obtenemos una puntuación de 0.6969 (¡peor que sin balancear!).

3.10. p3_09

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 sobre ajuste?

3.11. p3_10

El gran sobreajuste de RandomForest y su tiempo elevado de ejecución hace que desestimemos este algoritmo de momento y volvamos a Lightgbm. De descubrimos que `get_dummies` no es la única forma de categorizar variables... como este cambio hizo que mejoraran los resultados voy a probar diferentes formas de binarizar 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 (10273/86867) , probamos a subirlo a ver cuál es mejor. Sin embargo el resultado en test es bastante peor, de 0.6823.

3.12. p3_11

Volvemos a partir del código de `p3_06.py` y probamos esta vez la otra alternativa de DictVectorizer Se consiguen otra vez 33 y una puntuación en training de 0.7375, pero en este caso tenemos 0 diferencias. 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).

3.13. p3_12

Parece que aumentar el número de hojas y estimadores es positivo para el resultado, pruebo a ejecutar 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.

3.14. p3_13

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.

3.15. p3_14

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 utilizaré un GridSearch en el que comprobaré qué valor es mejor entre `[0.05, 0.075, 0.1, 0.15, 0.175]` (5 min para las pruebas, archivo `p3_14_sc`). 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. Tiempo de ejecución por partición: 75 - 100 segundos. Puntuación en training: 0.8184, la puntuación en test es de 0.7452.

3.16. p3_15

En nuestras gráficas de selección de características destacaba que las primeras variables coincidían en todas ellas, eran las llamadas `geo_level_i_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 6 y 7 observamos la relación entre los valores que toma esta variable y el grado de daño del edificio.

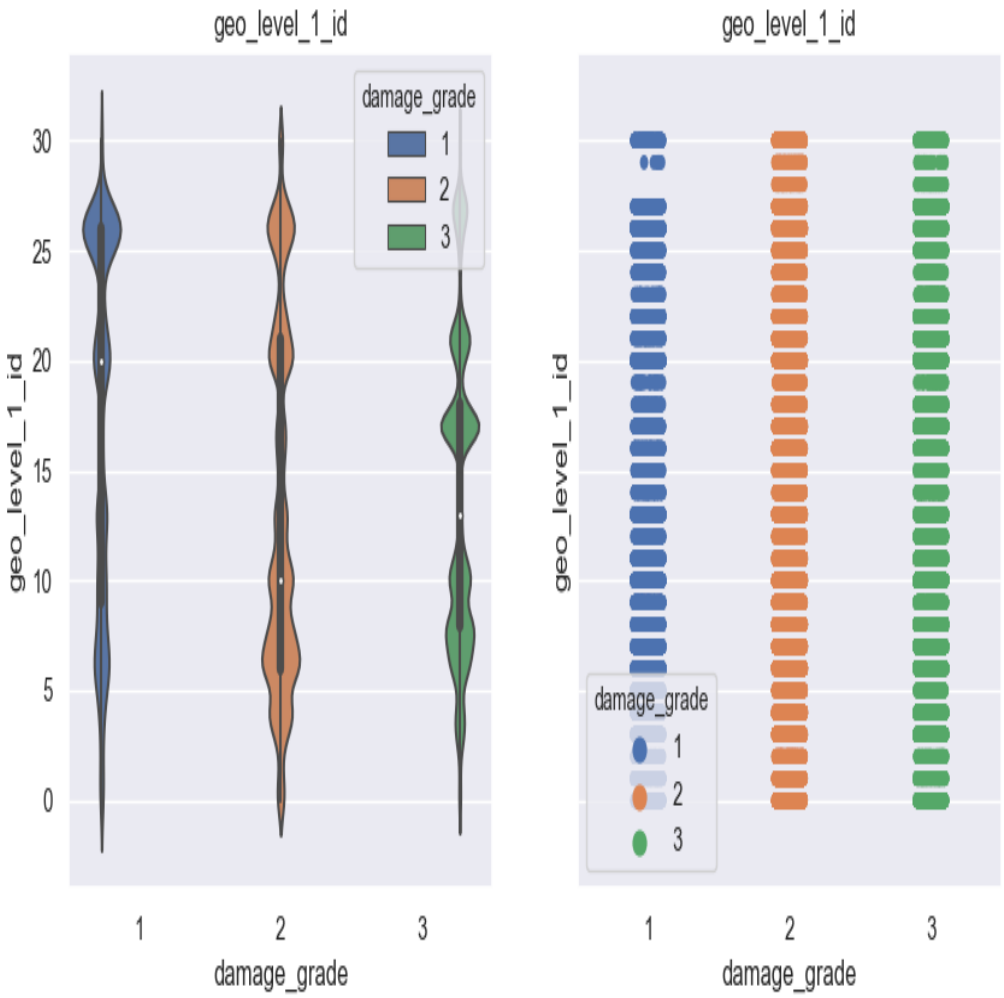

Figura 6: Relación `geo_level_1_id` y `damage_grade`.

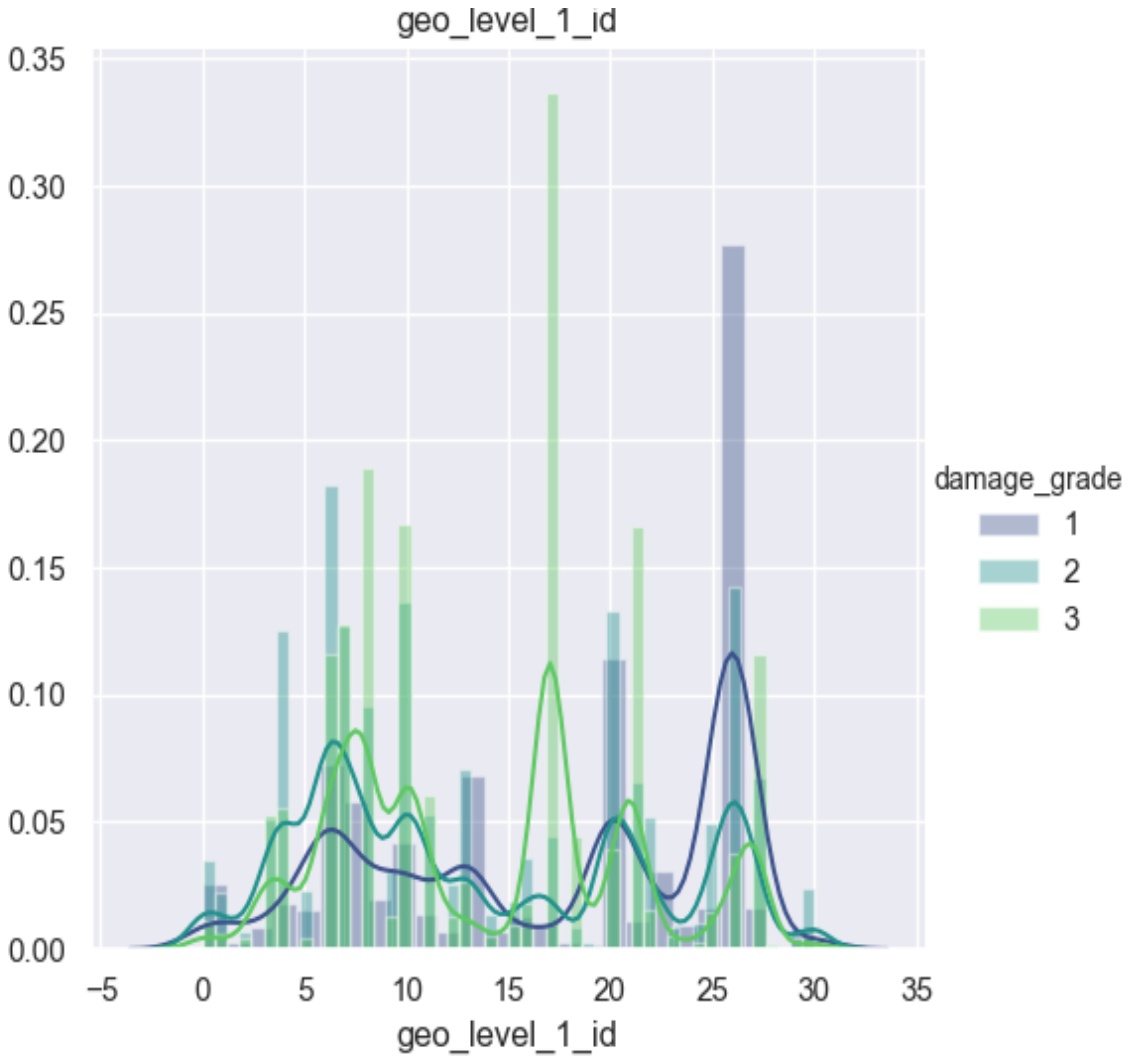

Figura 7: Relación `geo_level_1_id` y `damage_grade`.

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 (con geo_level_2 y 3 aumentaría demasiado el número de variables).

Tras binarizar 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. Tiempos entre 17 y 22 segundos.

Pruebo 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.

Umbral 0.99, nos deja con 70 variables tiempos entre 20 y 24 segundos 0.7369.

Si nos quedamos con las 98 variables los tiempos van desde 19 hasta 24 y el resultado es de 0.7361.

No conseguimos una mejora, ni tampoco nos queda mucho más tiempo hoy, así probamos a aumentar el número de hojas para tratar de mejorar el intento 14. Tarda unos segundos 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 ser que se está produciendo sobreajuste.

3.17. p3_16

Hemos visto que los parámetros num_leaves y n_estimators son determinantes para la buena ejecución de este algoritmo, por ello, realizaremos un ajuste de parámetros específico a ver qué combinación de parámetros proporciona mejores resultados. Probaremos las siguientes combinaciones: 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 142.4min 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, cosigue 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...

Así, subimos los resultados de la ejecución sin binarizar esta variable... resultado en test de 0.7443 :(

3.18. p3_17

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, a ver si obtiene resultados similares o incluso mejores. Utilizamos 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, pruebo a darle el valor 0.5 y 1 pero no afecta a los resultados. 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 y aumentamos el número de estimadores y la profundidad máxima. n_estimators = 400, eta = 0.1, max_depth = 8. Conseguimos una puntuación en entremaiento de 0.7913 que se tradujo en 0.7425 en test.

3.19. p3_18

Pruebo 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.

3.20. Pruebas fallidas

Tras observar los parámetros de Lightgbm 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. No funciona (p3_categorical.py) ValueError: could not convert string to float: 't' solo acepta tipos enteros... Una vez solucionado ese error nos encontramos con: [LightGBM]

[Fatal] categorical_feature is not a number, if you want to use a column name, please add the prefix ”name:”to the column name

3.20.1. umap

Por recomendación de un compañero, pruebo a utilizar umap para reducir la dimensionalidad. Tras instalar el paquete mediante `pip install umap-learn`, pruebo su funcionamiento de forma básica en `p3_umap.py` (algoritmo: lightgbm).

Tras largo tiempo de ejecución (media hora por lo menos) consigue exactamente los mismos resultados que p3_00 :((resultados en submissions_ap.csv)

3.20.2. Información mutua

Tras p3_04...

Probamos a utilizar como criterio 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... p3_mi.py notamos que tarda mucho, puede ser porque se basa en cosas tipo knn que con el alto número de instancias es excesivamente lento...

3.20.3. 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 142.44 segundos para seleccionar.

Tarda del orden de 2700 segundos y me da un error :/ mala idea un random forest con tantas variables.