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

Práctica 3: Competición en DrivenData

Alberto Jesús Durán López
DNI: 54142189-M
Email: albduranlopez@correo.ugr.es
Grupo Prácticas: Martes

4 de enero de 2020

SUBMISSIONS

Score	Submitted by	Timestamp
0.6883	Jesus_Duran_UGR	2019-12-22 20:00:51 UTC
0.6776	Jesus_Duran_UGR	2019-12-22 20:39:41 UTC
0.6920	Jesus_Duran_UGR	2019-12-22 20:55:09 UTC
0.7024	Jesus_Duran_UGR	2019-12-23 13:29:17 UTC
0.7081	Jesus_Duran_UGR	2019-12-23 13:34:33 UTC
0.7360	Jesus_Duran_UGR	2019-12-23 13:47:21 UTC
0.6535	Jesus_Duran_UGR	2019-12-24 08:53:17 UTC
0.6887	Jesus_Duran_UGR	2019-12-24 09:52:28 UTC
0.7444	Jesus_Duran_UGR	2019-12-24 10:24:11 UTC
0.7454	Jesus_Duran_UGR	2019-12-25 18:24:51 UTC
0.7454	Jesus_Duran_UGR	2019-12-25 19:52:09 UTC
0.7466	Jesus_Duran_UGR	2019-12-25 20:16:23 UTC
0.7466	Jesus_Duran_UGR	2019-12-26 07:57:59 UTC
0.7475	Jesus_Duran_UGR	2019-12-26 12:35:38 UTC
0.7478	Jesus_Duran_UGR	2019-12-26 23:48:17 UTC
0.7476	Jesus_Duran_UGR	2019-12-27 06:52:34 UTC
0.7456	Jesus_Duran_UGR	2019-12-27 21:45:25 UTC
0.7480	Jesus_Duran_UGR	2019-12-28 06:03:34 UTC
0.7463	Jesus_Duran_UGR	2019-12-28 07:35:57 UTC
0.7478	Jesus_Duran_UGR	2019-12-28 13:36:22 UTC
0.7471	Jesus_Duran_UGR	2019-12-29 16:54:57 UTC
0.7478	Jesus_Duran_UGR	2019-12-30 13:00:18 UTC
0.7486	Jesus_Duran_UGR	2019-12-30 16:50:06 UTC
0.7484	Jesus_Duran_UGR	2019-12-30 18:59:06 UTC
0.7486	Jesus_Duran_UGR	2019-12-31 00:03:45 UTC
0.7485	Jesus_Duran_UGR	2019-12-31 01:23:59 UTC
0.7496	Jesus_Duran_UGR	2019-12-31 08:59:03 UTC

Figura 0.1: Submissions

Índice

1. Introducción

En esta práctica usaremos métodos avanzados para aprendizaje supervisado en clasificación y participaremos en una competición real de DrivenData (`https://www.drivendata.org/`
El problema a resolver es *Richter's Predictor: Modeling Earthquake Damaga*, disponible en `https://www.drivendata.org/competitions/57/nepal-earthquake/`

El objetivo es el de predecir el nivel de daño a los edificios causados por un terremoto, `Gorkha`, en 2015 en Nepal. Se usarán los datos proporcionados por la Oficina Central de Estadística y se trabajará especialmente con el conjunto de datos de entrenamiento.
A partir de un preprocesado de los datos junto a diferentes pruebas de algoritmos, se intentará mejorar nuestro modelo.

Para realizar una primera primera ejecución contamos únicamente con la descripción de las características, disponible en `https://www.drivendata.org/competitions/57/nepal-earthquake/page/136/` y con un ranking de las primeras 50 posiciones, desconociendo el procedimiento usado para cada uno de ellos.

2. Estudio del DataSet

El conjunto de datos de entrenamiento consta de 260.601 instancias y 39 atributos categóricos, binarios y enteros.
Se predecirá la variable `damage_grade` la cual representa la cantidad de daño que se ha producido al edificio. Esta variable puede tomar los valores 1, 2 y 3, que representan un daño bajo, una cantidad media de daño y una destrucción casi completa, respectivamente.

Estudiando las variables de los datos proporcionados y, con la ayuda de una pequeña función en python, observamos que no presentan valores perdidos, por lo que no es necesario imputar dichos valores.

```
print("Valores perdidos:")
print(data_training.isnull().sum())
data_training.isnull().sum().plot.bar()
plt.show()
```

El gráfico correspondiente al código anterior muestra que todas las variables contienen un total de 0 valores perdidos.

Sin embargo, esto no supondría ningún problema usando la clase *Imputer* que nos proporciona sklearn. Se podría realizar la siguiente función para imputar dichos valores por la estrategia seleccionada, es decir, *mean*, *median* o *most_frequent*.

Mostramos a continuación el fragmento de código referido:

```
def ImputarStrategy(strat):
    imp = Imputer(missing_values='NaN', strategy=strat)
    imp = imp.fit(X)
    X_train_imp = imp.transform(X)
    imp = imp.fit(X_tst)
    X_tst_imp = imp.transform(X_tst)

    return X_tst_imp, X_train_imp
```

Por otro lado, mostramos las clases con la ayuda del siguiente diagrama de barras:

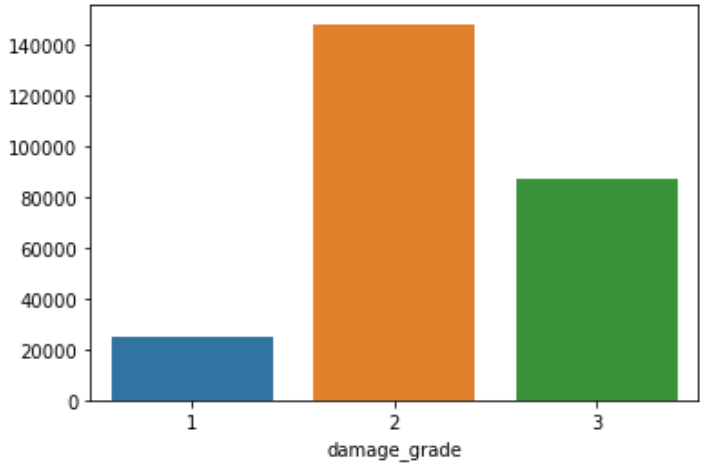
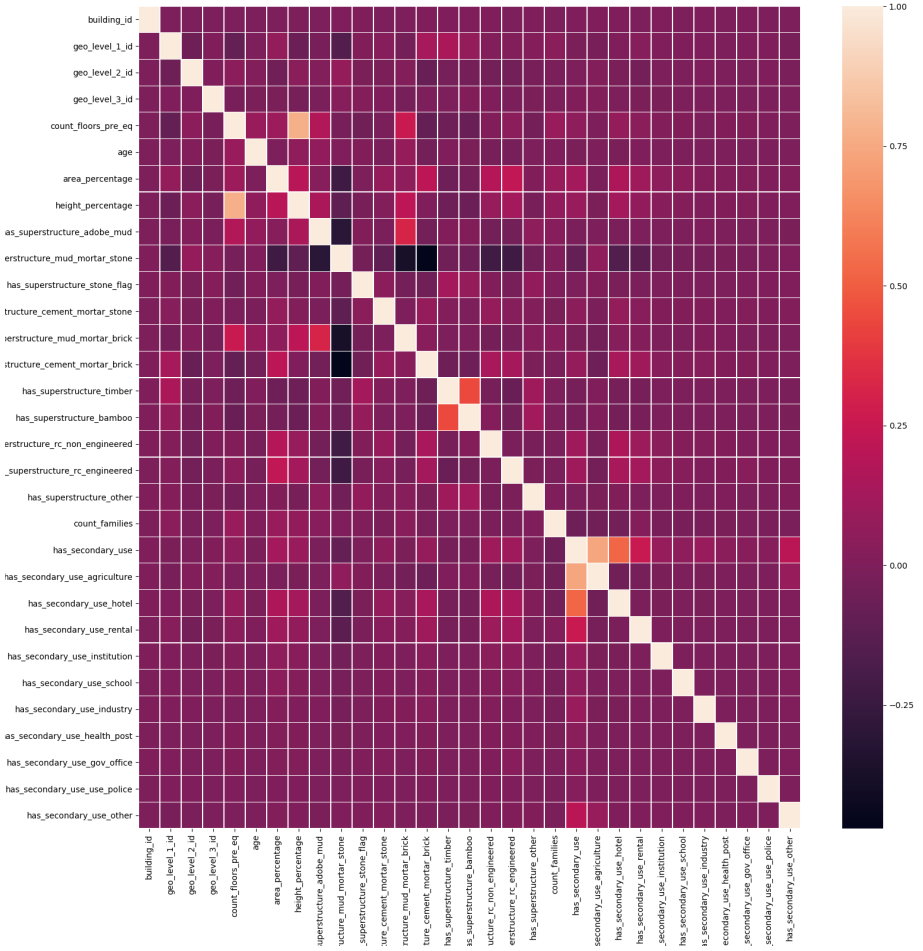


Figura 2.1: Clases

Podemos observar (en color naranja), que la clase predominante es la número 2 (con más de 140.000 instancias), seguida de la clase 3 y dejando con apenas 20.000 instancias la clase 1. El desbalanceo de clases supone un grave problema en el aprendizaje automático y obtener un muestreo equilibrado de las clases supondría una mejora en muchos escenarios.
Dejaremos el problema del desbalanceo de clases para secciones posteriores, donde se explicará con profundidad como hacer frente a dicho problema.

Por último, procedemos a mostrar un heatmap o mapa de calor donde se muestra la correlación de las variables:



En el heatmap anterior los colores más claros representan una correlación mayor respecto a los colores más oscuros.
Se ha eliminado la variable `building_id`, que contiene un identificador único para cada edificio y no aporta nada al conjunto del dataset. Por otro lado, en nuestro dataset existen muchos atributos del tipo *has_ secondary_ use*, variable que está relacionada con el resto de variables que se llaman de forma similar: *has_ secondary_ use*....
Tras realizar algunos intentos de eliminar alguna otra variable (usando por ejemplo la herramienta *Boruta*), se ha comprobado que no hay apenas ruido en la base de datos y que en resumen, todas las variables 'aportan' algo por lo que los intentos de eliminar las variables menos significativas han sido en vano.

3. Preprocesado

Realizamos distintos tipos de preprocesado a nuestros datos, explicaremos cada uno de ellos detalladamente:

3.1. Variables numéricas y categóricas

- **Preprocesado Inicial**: Se trata del preprocesado que venía por defecto, proporcionado por el profesor. Transforma variables numéricas a categóricas. Sin embargo, los algoritmos no se veían potenciados, por lo que procedemos a realizar otro tipo de preprocesado a las variables categóricas.

- **Biyección con números naturales - `[Preprocesado1]`**: Establecemos una biyección con los números naturales para transformar las variables categóricas a numéricas. A cada letra del abecedario le hacemos corresponder su número natural asociado, es decir:

  a $\rightarrow$ 1 , b $\rightarrow$ 2 , c $\rightarrow$ 3 , ... , z $\rightarrow$ 27

  Se realiza tal y como se muestra a continuación:

```
biyeccion = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7,
  'h':8, 'i':9, 'j':10, 'k':11, 'l':12, 'm':13, 'n':14,
  '/n':15,'o':16, 'p':17, 'q':18, 'r':19, 's':20,'t':21,
  'u':22, 'v':23, 'w':24, 'x':25, 'y':26, 'z':27}

preprocesado1 = {"roof_type": biyeccion,
  "land_surface_condition": biyeccion,
  "position": biyeccion,
  "other_floor_type": biyeccion,
  "legal_ownership_status": biyeccion,
  "foundation_type": biyeccion,
  "ground_floor_type": biyeccion,
  "plan_configuration": biyeccion,
}
```

- **Restringir dominio variables - `[Preprocesado2]`**: Observando el archivo .csv proporcionado para realizar la práctica, podemos comprobar que las variables numéricas toman pocos valores. Es por esto por lo que se podría realizar una modificación a la biyección anterior y restringir el dominio que toman las variables. Ordenamos por orden alfabético los valores que pueden tomar las variables categóricas y le asignamos los números naturales, empezando por el 1 y de forma ascendente, tal y como mostramos a continuación:

```
preprocesado2 = {"roof_type": {'n': 1, 'q': 2, 'x': 3},
  "land_surface_condition": {'n': 1, 'o': 2, 't':3},
  "position": {'j': 1, 'o': 2, 's': 3, 't': 4},
```

```
    "other_floor_type": {'j': 1, 'q': 2, 's': 3, 'x': 4},
    "legal_ownership_status": {'a':1,'r':2,'v':3,'w':4},
    "foundation_type":{'h':1,'i':2,'r':3,'u':4,'w':5},
    "ground_floor_type": {'f':1,'m':2,'v':3,'x':4, 'z':5},
    "plan_configuration": {'a': 1, 'c': 2, 'd': 3, 'f': 4,
         'm': 5, 'n': 6, 'o': 7, 'q': 8, 's': 9, 'u': 10}
    }

```

El uso de este preprocesado obtiene prácticamente los mismos resultados que el preprocesado1, no estableciendo una clara mejora.

- **Get Dummies -** `[Preprocesado3]`: Se puede usar este método para convertir variables categóricas en variables numéricas. Con esto, se obtiene la columna categórica y genera de manera automática una lista de números (que pueden ser 1 si contiene la cadena o 0 en caso contrario), cada uno correspondiente a una categoría particular de la variable.
  Sin embargo, aplicamos getDummies al conjunto de variables total, tal y como se muestra a continuación:

```
data_training = pd.get_dummies(data_training)
data_test = pd.get_dummies(data_test)

```

3.2. Balanceo de clases

Otro problema de los datos proporcionados es que las clases están desbalanceadas como hemos comentado anteriormente, lo que se ve reflejado en un peor resultado, cosa que se podría evitar balanceando las clases
Principalmente existen estas dos formas:

- **Algoritmos preparados para clases desbalanceadas**: Como bien se indica, existen algoritmos preparados con un parámetro para tratar el problema del desbalanceo de clases. Cuando se incluye este parámetro, el modelo por lo general actúa de forma automática aplicando una métrica por defecto, como es el caso de `is_unbalance` de *LightGBM*.

  Por otro lado, existen algoritmos como el propio *LightGBM* o *Random Forest* donde podemos ajustar los pesos de las clases de forma automática o manual con el parámetro `class_weight` o `scale_pos_weight`.

- **Smote**: Herramienta muy útil para realizar un oversampling de los datos de forma sencilla. Es necesario instalar el paquete `smote_variants`.

Existen bastantes variantes disponibles en : `https://github.com/gykovacs/smote_variants`, además, se incluye una descripción y ejemplos de cada uno, mostrando incluso un ranking con los mejores resultados:

sampler	overall	auc	rank_auc	gacc	rank_gacc	f1
polynom-fit-SMOTE	2.5	0.902538	6	0.870753	1	0.695154
ProWSyn	4.5	0.904389	1	0.868449	4	0.690284
SMOTE-IPF	7.5	0.902565	5	0.868715	3	0.687935
Lee	8	0.902318	7	0.868324	5	0.688082
SMOBD	9.25	0.902247	8	0.86766	6	0.688885
G-SMOTE	13.5	0.901916	10	0.865103	18	0.686613
CCR	14.25	0.902112	9	0.861994	30	0.687886
LVQ-SMOTE	14.75	0.902799	3	0.862295	29	0.683646
Assembled-SMOTE	15.5	0.902691	4	0.866914	7	0.688614
SMOTE-TomekLinks	15.75	0.901016	14	0.866174	9	0.684708

Figura 3.1: Ranking mejores Smote

Como norma general, equilibrar las clases realizando un oversampling no es recomendable, lo adecuado y óptimo sería obtener un equilibrio en las clases, realizando un undersampling de la clase superior y un oversampling de la inferior, obteniendo así el número de instancias de la clase intermedia.

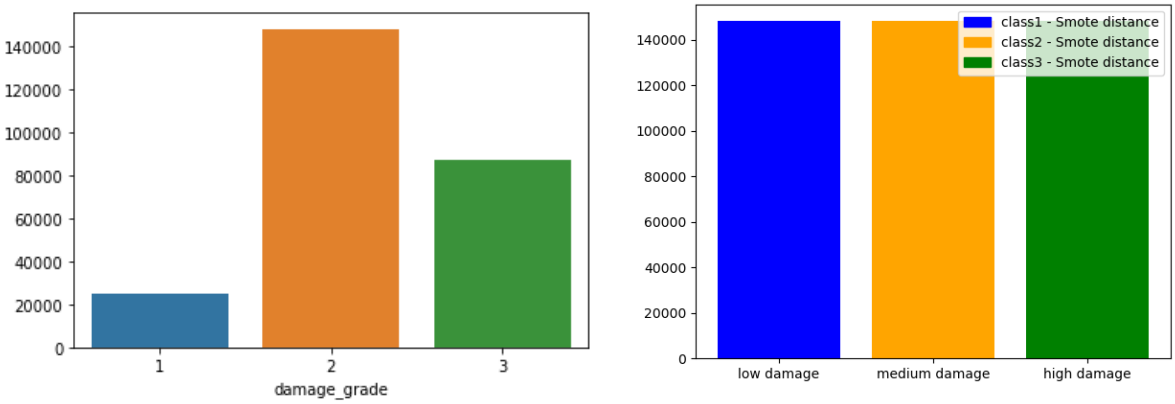
Sin embargo, debido a que nuestro dataset no presenta outliers ni ruido, en las últimas submissions se ha realizado un oversampling a la clase superior y se han obtenido resultado bastante prometedores.

En nuestro caso probamos dos variantes, *distance_SMOTE Polynom-fit-SMOTE* que explicaremos a continuación:

- **Distance SMOTE**

```
oversampler= sv.MulticlassOversampling(sv.distance_SMOTE())
```

El número de instancias de las 3 clases queda totalmente balanceado, obteniendo un total de instancias de la clase superior para cada una de ellas. Desde la submission 23 hasta la 26 se hace uso de esta variante.



(a) Unbalanced

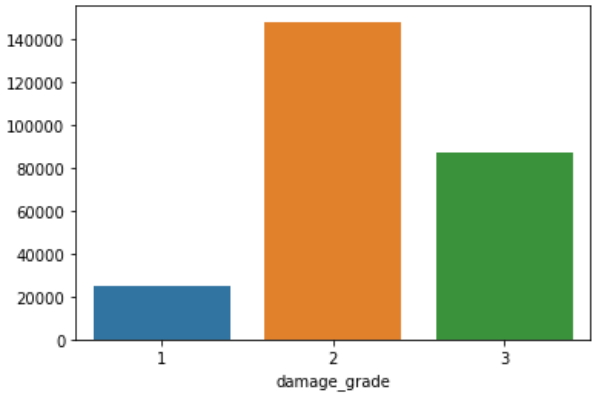
(b) Balanced

La documentación oficial disponible quizás es algo escasa y al incluir el término *distance* pensamos que quizás era necesario normalizar los datos (proceso que no hacemos porque estamos ante un algoritmo basado en árboles). Por esto mismo, preferimos probar otra variante y comprobar si los resultados mejoran.

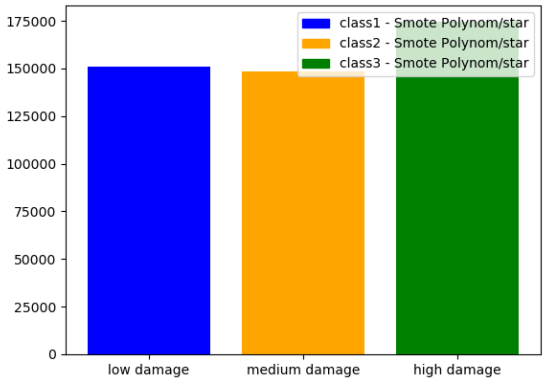
- **Polynom-fit-SMOTE**: Ocupa el rank1 en la clasificación de los SMOTE que se ha incluído en la página anterior.

```
oversampler=sv.MulticlassOversampling(sv.polynom_fit_SMOTE(
    topology='star'))
```

Como parámetro, incluímos la topología *star*, aunque dicha elección no es única, pudiendo elegir entre otras opciones como *mesh* o *bus*



(c) Unbalanced



(d) Balanced

En este caso, las clases no están completamente balanceadas. La clase 1 y 2 tienen una ligera diferencia, rondando las 140.000 instancias, mientras que la clase 3 llega incluso a las 175.000 instancias. Aunque pueda parecer extraño, con este ligero desbalanceo se obtienen los mejores resultados de todos.

4. Algoritmos

En esta sección explicaremos los diferentes algoritmos usados, así como ventajas y desventajas encontradas en el estudio de los modelos.

Antes de adentrarnos en dicho estudio, cabe destacar el empleo de técnicas como `GridSearch`, herramienta que ante una serie de parámetros, escoge los mejores para el modelo. Principalmente se eligen de forma exhaustiva cuales son los mejores parámetros para entrenar el modelo, evitando a toda costa el underfitting y el overfitting que se produce en el proceso de entrenamiento.

A continuación mostramos un ejemplo realizado de `GridSearch` en el que calculamos de entre los parámetros indicados, los mejores:

```
print("------ LightGBM + GRID...")

lgbm = lgb.LGBMClassifier(objective='multiclass',
                          max_depth=20,
                          n_jobs=-1)
parametros = {
"n_estimators": [3150, 3200, 3270, 3400],

"learning_rate": [0.107, 0.108, 0.109, 0.111, 0.112,
                  0.113, 0.114, 0.115, 0.116, 0.117,
                  0.118, 0.119, 0.121, 0.122, 0.123,
                  0.124, 0.125, 0.126]}

devolver = GridSearchCV(estimator=lgbm,
          param_grid=parametros,
          scoring='f1_micro',#puede ser macro, weighted
          cv=3,
          n_jobs=-1)

print("Obteniendo mejor configuración: ")
devolver.fit(X,y)
```

```
##################################
##################################
Ejecutando algoritmo número 7
##################################
##################################

------ LightGBM + GRID...
Obteniendo mejor configuración:
{"learning_rate": 0.108, "n_estimators": 3200}
---------------------------------
Learning rate óptimo: 0.108
N_estimador óptimo: 3200
---------------------------------
Calculando algoritmo con mejores parámetros
F1 score (tra): 0.8300
Creando archivo submission número 7
Descanso durante 1 minuto antes de empezar la siguiente ejecución

In [1]:
```

Figura 4.1: Ejecución GridSearchCV

Otra técnica que se podría haber usado es `RandomSearch`, útil cuando se desea ahorrar en tiempo y el número de instancias no supone un problema. Se han realizado ejecuciones con grid para ajustar parámetros usuales como *n_ estimators, learning_ rate, max_ depth* y otros menos usuales como *objective, is_ unbalance* o incluso *class_ weight*, para ajustar de forma óptima el peso que le asignamos a cada clase.

- **XGBoost**: Extreme Gradient Boosting. Se trata de una implementación de árboles de decisión con Gradient Boosting diseñada para minimizar la velocidad de ejecución y maximizar el rendimiento.

  En nuestras pruebas no obteníamos un score en el training lo suficientemente alto como para subirlo a Drivendata, ya que teníamos únicamente 3 submission al día y no había que arriesgar, además, el tiempo de ejecución era bastante superior al resto por lo que el ajuste de hiperparámetros se complicaba.

- **LightGBM**: Light Gradient Boosting Machine. Basada en árboles de decisión y paralelización masiva de modelos. Se caracteriza por ser muy eficaz a la hora de ejecutar los modelos.

  Es el algoritmo que mejor resultados nos ha dado con diferencia. Su eficiencia destacaba por encima del resto, además, hablando en términos de eficacia y bajo los mismos parámetros, era incluso superior a otros modelos, resultando en un gran ahorro de tiempo a la hora de realizar el ajuste de hiperparámetros.

  Algunas de estas ejecuciones se han guardado en el archivo **RESULTADOS.CSV**, disponible en la carpeta **Submissions** entregada en la práctica. El resto de resultados de grid se han incluído como comentarios en el código *earthquake.py*.

```
lgbm = lgb.LGBMClassifier(objective='multiclassova',
                          learning_rate=0.108,
                          n_estimators=3198,
                          max_depth=20,
                          random_state=seed,
                          class_weight={1: 1, 2: 0.8, 3: 0.7})
```

En el fragmento de código anterior se muestra un ejemplo de ejecución tras haber realizado un ajuste óptimo de los parámetros, correspondiente a la submission 17.

- **Random Forest**: Se trata de un meta estimador que ajusta un número de árboles de decisión en muestras del dataset para después realizar un promedio y mejorar la precisión a la vez que se controla el over-fitting.

  Es un método versátil de aprendizaje automático capaz de realizar tanto tareas de regresión como de clasificación. Está incluído en la biblioteca de *scikit-learn* y se construye de la siguiente forma:

```
rf = RandomForestClassifier(n_estimators=500,
                            learning_rate=0.1, max_depth=20,
                            min_samples_split=2,
                            random_state=seed,
                            class_weight='balanced')
```

  Los principales parámetros de este clasificador son `n_estimators` y `learning_rate`, teniendo que un aumento del número de estimadores resulta en un tiempo mayor de ejecución pero quizás no una mayor precisión. Por otro lado, un mayor `learning_rate` se ve reflejado en un modelo que reacciona más rápido ante los datos que van llegando pero propenso a un overfitting superior. En el caso de este algoritmo, era bastante complicado establecer un *learning_rate* óptimo, obteniendo en la mayoría de ejecuciones un resultado con bastante overfitting en el training.

- **CatBoost**: Algoritmo de aprendizaje automático basado en potenciación del gradiente *'Gradient Boosting'*. El nombre de CatBoost proviene de la unión de Category y Boosting, donde el primer término hace referencia al hecho de que la librería funciona perfectamente con variables categóricas.

```
cbc= CatBoostClassifier(learning_rate=0.1,
                        n_estimators=550,
                        loss_function='MultiClass',
                        eval_metric='TotalF1',
                        od_pval=0.001,
                        od_type='IncToDec',
                        random_seed=54142189,
                        bootstrap_type='MVS', #Probar Bayesian
                        best_model_min_trees=250,
                        max_depth=12)
```

Con la configuración mostrada anteriormente, se llegó a obtener un F1-Score de 0.7360 en la submission 6, sin embargo, el tiempo de ejecución era bastante superior al resto y se descartó por las mismas razones que `XGBoost`

- **BaggingClassifier**: **B**ootstrap **Agg**regat**ing**. Técnica que entrena diferentes muestras del dataset original (con la misma cardinalidad) usando combinaciones con repetición, combinando todas ellas y obteniendo una predicción final en la cual la varianza se ha reducido.

  Ha resultado ser una técnica esencial en el desarrollo de nuestro modelo final, mejorando por varias centésimas nuestro Score final. Para dicho clasificador, se han realizado variaciones y ejecuciones con `GridSearch` en el número de estimadores y en el parámetro booleano *Bootstrap*, que indica si las muestras se extraen con reemplazamiento (True), o en caso contrario, sin él (False). Tras probar y ejecutar durante varios días, no se obtuvo una configuración adecuada, dejando todo azar.

```
lgbm = lgb.LGBMClassifier(...)

clf=BaggingClassifier(
        base_estimator=lgbm,
        n_estimators=15,
        bootstrap=True) #A veces se obtiene mejor resultado con False
```

- **VotingClassifier**: Herramienta que requiere el entrenamiento de diversos algoritmos para después unirlos y predecir la salida final.
  Para un resultado óptimo de este clasificador es conveniente el uso y entrenamiento de algoritmos diferentes (basados en árboles, reglas, probabilisticos...etc).

En nuestro caso usamos únicamente *LightGBM* con diferentes parámetros potenciando el resultado final uniendo dos bagging (uno de ellos con Bootstrap=True y otro con esta variable tomando un valor False) y juntándolo en este Voting, tal y como se muestra a continuación:

```
lgbm1=lgb.LGBMClassifier(learning_rate=0.1, objective='multiclassova',
                         n_estimators=3198, n_jobs=-1,
                         num_leaves=34, max_bin=500,
                         max_depth=22, random_state=seed)

lgbm2=lgb.LGBMClassifier(learning_rate=0.1, objective='multiclassova',
                         n_estimators=3198, n_jobs=-1,
                         num_leaves=34, max_bin=500,
                         max_depth=24, random_state=seed)

clf1 = BaggingClassifier(base_estimator=lgbm1, n_estimators=20,
                         bootstrap=False, random_state=seed)

clf2 = BaggingClassifier(base_estimator=lgbm2, n_estimators=15,
                         bootstrap=False, random_state=seed)

devolver = EnsembleVoteClassifier(clfs=[clf1, clf2],
                                  weights=[1, 1], voting='soft')
```

El resultado tras ejecutar este modelo combinado, superando incluso las 5 horas de ejecución, ha sido la combinación perfecta (junto con oversampling de SMOTE). Es el seleccionado para la submission final, obteniendo un F1-Score de 0.7496 y un rank final de 34.

BEST	CURRENT RANK	# COMPETITORS	SUBS. MADE
0.7496	34	1650	3 of 3

Como nota aclaratoria, `EnsembleVoteClassifier` tiene un parámetro *voting* que puede tomar valor 'hard' o 'soft'. En el caso de 'hard', el clasificador cuenta el número de instancias más votada y toma dicho valor. Por otro lado, cuando los modelos están mejor entrenados (como es nuestro caso, debido a la gran cantidad de GridSearch que se esconden tras él) es recomendable el uso de 'soft' en el que se indica al clasificador que realice una media de las probabilidades de cada instancia.

- **Otros**: Se han usado otros algoritmos como LogisticRegression, pero a la hora de realizar pruebas con sus parámetros no se han obtenido buenos resultados en el training. Esto es sucede porque al no ser algoritmos basados en árboles, necesitan un preprocesado básico de normalización de variables.

5. Resultados

Mostramos una tabla a modo de resumen de cada una de las subidas realizadas a Drivendata, incluyendo la fecha de subida, el rank tras subir la submission, algoritmo usado, configuración&preprocesado y tanto el score obtenido en el training como en el test

Submission	Date	Rank	Algorithm	Score tra	Score tst	Configuration and Preprocessing
1	22/12/2019-8pm	377	LightGBM/CrossValidation	0.7264	0.6883	Configuración y preprocesado inicial
2	22/12/2019-9pm	377	XGBoost/Cross Validation	0.6886	0.6776	Configuración y preprocesado inicial
3	22/12/2019-9pm	356	Random Forest	0.7692	0.6920	Configuración y preprocesado inicial
4	23/12/2019-1pm	332	CatBoost	0.7856	0.7024	Configuración y preprocesado inicial
5	23/12/2019-2pm	316	Random Forest	0.7659	0.7081	Preprocesado1, 350 estimators
6	23/12/2019-2pm	180	CatBoost	0.7903	0.7360	Preprocesado1, 550 estimators
7	24/12/2019-9am	180	CatBoost	0.8555	0.6535	Preprocesado inicial, oversampling smote
8	24/12/2019-10am	180	Random Forest	0.7548	0.6887	Preprocesado1, asignando pesos(class_weight)
9	24/12/2019-10am	110	LightGBM	0.8011	0.7444	Preprocesado2, is_unbalance, 2000 estimators
10	25/12/2019-6pm	98	LightGBM/GridSearch	0.8272	0.7454	Preprocesado2, is_unbalance, 2950 estimators
11	25/12/2019-8pm	98	LightGBM/CrossValidation	0.8272	0.7454	Preprocesado2, is_unbalance, 3198 estimators
12	25/12/2019-8pm	81	LightGBM/GridSearch	0.8299	0.7466	Conf. anterior ajustando con GridSearch
13	26/12/2019-8am	81	LightGBM/GridSearch	0.8299	0.7466	GridSearch, scale_pos_weight=0.5
14	26/12/2019-1pm	72	LightGBM/Bagging	0.8213	0.7475	Preprocesado2, 3198 estimators
15	26/12/2019-00	69	LightGBM/Bagging	0.8250	0.7478	Conf. Anterior, 15 estimators Bagging
16	27/12/2019-7am	69	LightGBM/Bagging	0.8253	0.7476	Conf Anterior, is_unbalance
17	27/12/2019-10pm	69	LightGBM	0.8328	0.7456	Preprocesado2, objective='multiclassova'
18	28/12/2019-6am	59	LightGBM/Bagging/Voting	0.8296	0.7480	Preprocesado2, doble Bagging, Voting soft
19	28/12/2019-8am	59	LightGBM	0.8286	0.7463	Preprocesado2, GridSearch, class_weight
20	28/12/2019-2pm	60	LightGBM/Bagging/Voting	0.8298	0.7478	Preprocesado2, doble Bagging, Voting hard
21	29/12/2019-5pm	60	LightGBM/GridSearch	0.8321	0.7471	GetDummies, max_bin=500, num_leaves=34
22	30/12/2019-1pm	62	LightGBM	0.8333	0.7478	Conf. Anterior, Distance_SMOTE, num_leaves=35
23	30/12/2019-5pm	48	LightGBM/Bagging	0.8270	0.7486	Conf. Anterior,bagging 15 estimators,bootstrap=false
24	30/12/2019-7pm	48	LightGBM/Bagging/Voting	0.8310	0.7484	Anterior,bagging 15 estimators,bootstrap=true,vote=soft
25	31/12/2019-00	49	LightGBM/Bagging/Voting	0.8295	0.7486	Anterior,doble Bagging (Bootstrap true/false), vote=hard
26	31/12/2019-1am	51	LightGBM/Bagging/Voting	0.8310	0.7485	Anterior, vote=soft
27	**31/12/2019-9am**	**34**	**LightGBM/Bagging/Voting**	**0.8371**	**0.7496**	**Anterior, polynom_fit_SMOTE, topology=star**

Tabla 5.1: DrivenData Submissions

Para explicar la tabla de resultados, dividiremos el trabajo realizado en 4 secciones:

- `Etapa 1`: Se engloban las 4 primeras submissions. Consistió en un primer contacto con nuestro dataset, familiarizandonos con las variables y funciones que se iban realizando. No modificamos el preprocesado proporcionado por el profesor, nos limitamos a probar distintos algoritmos *LightGBM, XGBoost, Random Forest y CatBoost* y a jugar con sus parámetros.

- `Etapa 2`: Desde la submission 5 hasta la 9. Tras comprobar que las variables categóricas tienen un dominio reducido, realizamos los dos preprocesados comentados en la sección anterior. Además, buscamos información sobre los distintos algoritmos y mejores parámetros a usar para finalmente elegir uno de ellos. Tras probar, nos quedamos con *LightGBM*, que destaca por su eficiencia, pudiendo ejecutar un modelo de 3000 estimadores en pocos minutos, a diferencia del resto de algoritmos que necesitan varias horas para realizar el mismo trabajo. Esto nos dará mucho juego en la futuras submissions ya que se podrá realizar ajuste de hiperparámetros y probar estas configuraciones en un tiempo muy reducido.

- `Etapa 3`: Desde la submission 10 hasta la 20. Esta etapa consistió en optimizar nuestro modelo y obtener el número óptimo de *n_estimators, learning_rate...etc* usando

la herramienta `GridSearchCV`. El tiempo en algunas ejecuciones superó incluso las 24h, pero el resultado fue bastante prometedor. Además, se realizó la búsqueda de la mejor configuración de las clases (asignandoles pesos con [`class_weights`], poniendo o quitando `is_unbalance`), sin embargo, no hubo una configuración que sobresaliera entre el resto, produciéndose fluctuaciones en el score del test.

Por otro lado, hicimos uso de las herramientas `Bagging` y `Voting`, comentadas en la sección anterior. A estas herramientas se le aplicaron numerosos `GridSearchCV` y los resultados mejoraron considerablemente, llegando incluso al rank 59 en la submission 18.

- `Etapa 4`: Desde la submission 21 hasta la última. En estos momentos nos encontrábamos a 2 días para la finalización de la competición y más o menos el 5º o 6º respecto de la clase. Necesitábamos usar alguna herramienta que nos permitiese sumar algunas centésimas más. Es aquí cuando decidimos cambiar nuestro preprocesado a `GetDummies`, además, siguiendo la información proporcionada en el siguiente link `https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html` modificamos los parámetros *max_ bin* y *num_ leaves*. Estos parámetros mejoran considerablemente nuestro score pero producían overfitting (como se puede ver en los scores del training), por ello nuestros resultados del Grid no eran realmente fiables al 100 %.

  Para lidiar con el overfitting usamos la herramienta Smote y realizamos un oversampling Multiclass-Distance. Sin embargo, usando Multiclass-Polynom con la topología *star* conseguimos un F1-score de 0.7496, llegando al rank 34 de la competición.

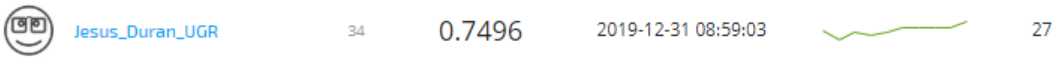


Figura 5.1: F1-Score 0.7496, rank 34

Referencias

[1] Página web de la asignatura - http://sci2s.ugr.es/graduateCourses/in

[2] http://scikit-learn.org/stable/modules/clustering.html

[3] GitHub de SMOTE - https://github.com/gykovacs/smote_variants