Práctica-3: Competición en DrivenData.

UNIVERSIDAD DE GRANADA
E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

UNIVERSIDAD
DE GRANADA

Departamento de Ciencias de la
Computación e Inteligencia Artificial

Inteligencia de negocio (2019-2020)

Daniel Bolaños Martínez
danibolanos@correo.ugr.es
Grupo 2 - Jueves 09:30h

Índice

1. Introducción.

Esta tercera práctica consistirá en la aplicación de métodos avanzados para aprendizaje supervisado en clasificación sobre una competición real disponible en DrivenData.

La competición se llama *Richter's Predictor: Modeling Earthquake Damage* y está basada en aspectos de la ubicación y construcción de edificios teniendo como objetivo la predicción del nivel de daño causado por el terremoto de Gorkha de 2015 en Nepal.

Los datos se recopilaron mediante encuestas realizadas por la Oficina Central de Estadística de Nepal. El conjunto de entrenamiento consta de 260.601 instancias y 39 atributos de tipo categóricos, enteros y binarios. El valor **building_id** toma valores únicos y solo sirve para identificar cada ejemplo, por lo que no será relevante en nuestro estudio, pudiendo eliminarlo. Algunas de las variables más relevantes son:

- **geo_level_1_id**, **geo_level_2_id**, **geo_level_3_id**: región geográfica donde se encuentra el edificio. Desde una región más general (nivel 1), hasta más específica (nivel 3). Representa un valor numérico que va desde: nivel 1: 0-30, nivel 2: 0-1.427, nivel 3: 0-12.567.
- **count_floors_pre_eq**: número de pisos del edificio antes del terremoto.
- **age**: edad en años del edificio.
- **area_percentage**, **height_percentage**: área y altura respectivamente del edificio.
- **ground_floor_type**: categoría del tipo de suelo donde está construido el edificio. Posibles valores: f, m, v, x, z.
- **has_superstructure_bamboo**: indica con 1 (True) o 0 (False) si la estructura tiene como material bambú. (Existen más por cada material de construcción).

Trataremos de predecir la variable ordinal **damage_grade**, que representa el nivel de daño al edificio afectado por el terremoto. Hay 3 grados: 1, representa un daño bajo; 2, representa una cantidad media de daño; y 3, representa la destrucción casi completa.

Para medir el rendimiento de nuestros algoritmos, utilizaremos la puntuación F1 que equilibra la precisión y el *recall* de un clasificador. Normalmente, el F1-score se utiliza para evaluar el rendimiento de un clasificador binario, pero como tenemos tres posibles etiquetas, utilizaremos una variante llamada puntuación F1 micropromediada.

2. Estudio general de los datos.

Antes de probar algoritmos, realizaremos un estudio sobre la estructura de la base de datos y comprobaremos como se comporta frente a problemas como el desbalanceo de clases, valores perdidos o correlación de variables. Este estudio en un futuro nos podrá ser útil para preprocesar los datos o elegir entre un algoritmo u otro, mejorando el rendimiento de nuestra solución y pudiendo obtener mejor puntuación de F1.

2.1. Distribución de las etiquetas.

Mostraremos la distribución de las etiquetas del conjunto de entrenamiento para ver si partimos de un problema con balanceo de clases o no. Usaremos el siguiente código para obtener la gráfica que se muestra a continuación.

```
def GraficoComprobarVar(data_y, path="./imagenes/"):
  plt.figure(figsize=(10,8))
  ax = sns.countplot('damage_grade', data=data_y)
  for i in ax.patches:
    ax.text(i.get_x()+0.2, i.get_height()+3, str(round((i.get_height()),
        2)), fontsize=14, color='dimgrey')
  plt.savefig(path+'damage-grade'+".png")
  plt.clf()
```

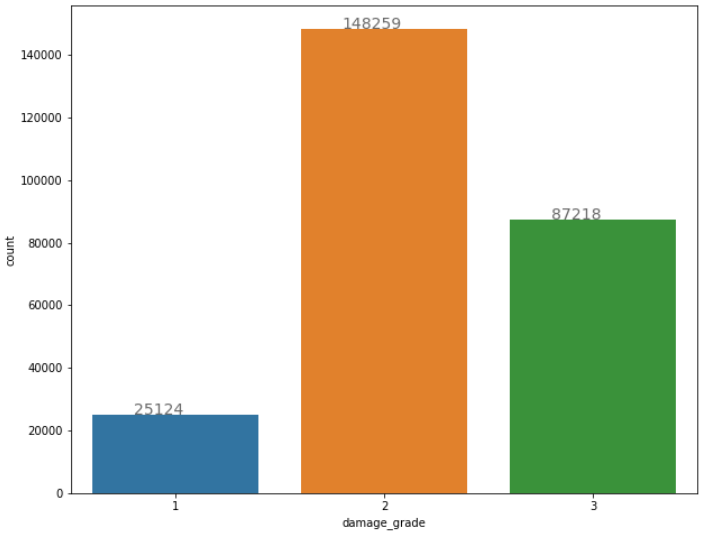

Figura 1: Distribución de etiquetas en el conjunto de entrenamiento.

En la Figura 2.1, podemos observar como existe una gran desproporción siendo la mayoría de casos de la clase 2, mientras que de la clase 1 existen muy pocas muestras. Esto quiere decir que el terremoto de 2015 causó daños de nivel 2 o más en la gran mayoría de casos y que pocos edificios recibieron daños mínimos. Al conocer que las clases están desbalanceadas, más adelante discutiremos si será más interesante usar oversampling para balancearlas o es más conveniente utilizar un algoritmo preparado para el desbalanceo.

2.2. Valores perdidos.

Mostraremos a partir de la siguiente función si el conjunto de datos de entrenamiento presenta valores perdidos para alguna variable.

```
def ComprobarValPer(data_tra , path="./imagenes/"):
  plt.subplots(figsize=(17,17))
  data_tra.isnull().sum().plot.bar()
  plt.savefig(path+"valores-perdidos"+".png")
  plt.clf()
```

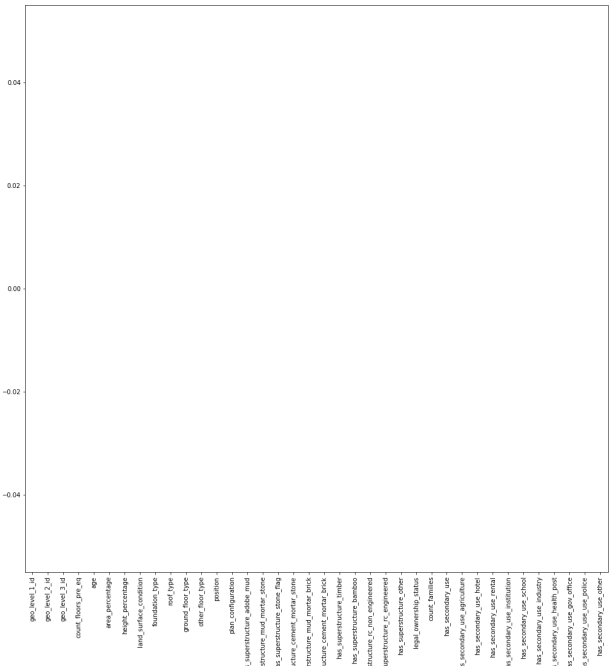

Figura 2: Características con valores perdidos.

Como podemos observar en la Figura 2.2, no tenemos valores perdidos en el conjunto de entrenamiento, por lo que será un problema menos. La no existencia de valores perdidos, no quiere decir que existan variables con valores que se salgan del rango o que tengan valores raros que se hayan utilizado como sustituto a dejar el valor como desconocido.

2.3. Correlación entre variables.

A continuación haremos un estudio sobre la correlación entre las variables. Este estudio será interesante a la hora de eliminar variables que nos proporcionen la misma información en el problema y por tanto sean irrelevantes en el cálculo de la predicción.

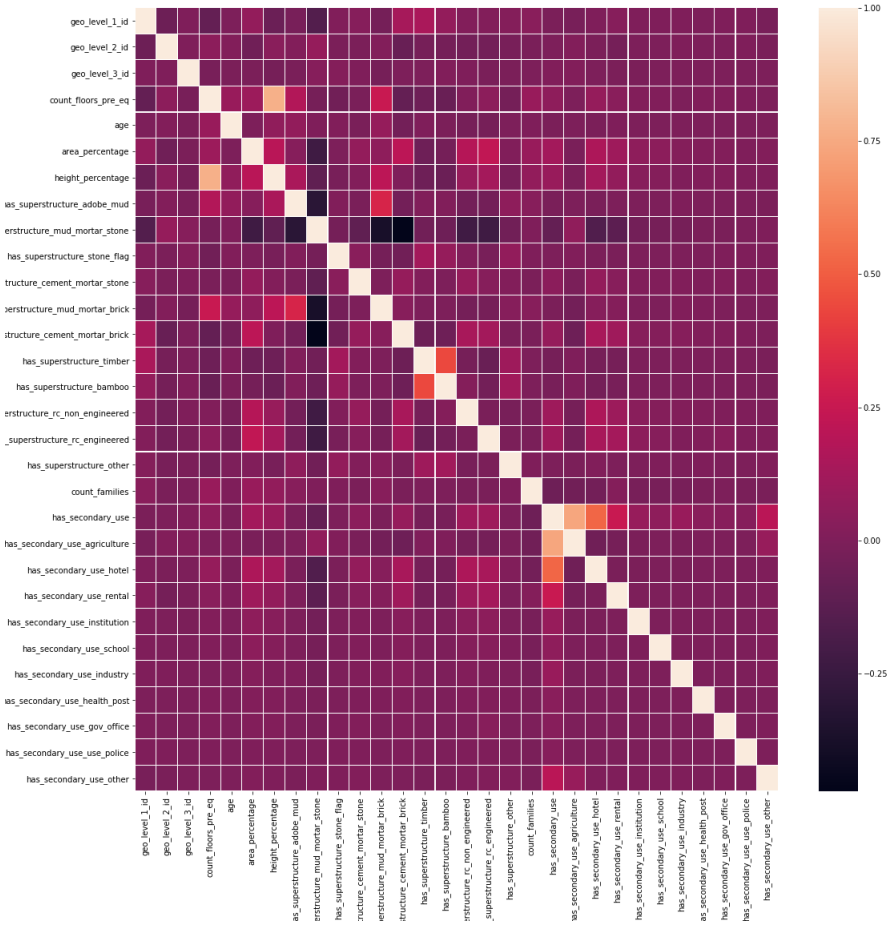

Figura 3: Matriz de correlación entre variables.

El código utilizado para obtenter la Figura 2.3 ha sido el siguiente:

```
def MatrizCorrelacion(data_tra, path="./imagenes/"):
  correlations = data_tra.corr()
  fig, ax = plt.subplots(figsize=(18,18))
  sns.heatmap(correlations, linewidths=0.125, ax=ax)
  plt.savefig(path+"matriz-corr"+".png")
  plt.clf()
```

Podemos observar que no tenemos ninguna correlación entre variables que sea total (1.0) por lo que en principio, no sería conveniente eliminar ninguna variable, puesto que aunque tengamos algunas que tengan más correlación entre ellas que la media, puede que la información que nos ofrezcan ambas variables sea importante.

Por ejemplo, podemos apreciar cierta relación entre las variables **height_percentage** y **count_floors_pre_eq**, esto se debe a que a mayor número de pisos tenga un edificio mayor altura alcanzará. También presentan cierto grado de correlación las variables **has_secondary_use** y **has_secondary_use_agriculture** probablemente porque la mayoría de casas en Nepal, sean de agricultores.

3. Preprocesado.

A continuación se mostrarán las diversas técnicas de preprocesado que se han utilizado para el problema. Aunque se hará una breve descripción de todas las utilizadas junto con el código relacionado, algunas han obtenido peores resultados que otras y por tanto se han ido descartando.

3.1. Elimina columnas.

La única columna que eliminaremos será **building_id** como ya comentamos en la introducción. A partir del estudio de la correlación y alguna prueba realizada, se ha visto que la eliminación de las variables mayor correladas perjudica a la clasificación. Esto significa que todas las variables son útiles a la hora de aportar información a la hora de clasificar los datos.

```
def eliminaLabels(x, y, tst, etiquetas):
  x.drop(labels=etiquetas, axis=1, inplace = True)
  y.drop(labels=etiquetas[0], axis=1, inplace = True)
  tst.drop(labels=etiquetas, axis=1, inplace = True)
```

3.2. Categóricas a numéricas.

Este preprocesado se dió con la plantilla por defecto, se ha utilizado en algunas pruebas pero se ha visto que no beneficia a la obtención de mejores resultados por lo que se ha optado por eliminarlo.

```
def catToNum(data):
  mask = data.isnull()
  data_tmp = data.fillna(9999)
  data_tmp = data_tmp.astype(str).apply(LabelEncoder().fit_transform)
  return data_tmp.where(~mask, data)
```

3.3. Categóricas a dummies.

Este preprocesado se ha realizado para sustituir al **category to number**. Es el equivalente a **One to many** que realizábamos en KNIME. Crea nuevas columnas con las opciones de las variables categóricas e indica con 0 o 1 si presenta esa característica. Este preprocesado es muy bueno ya que contamos con pocas variables categóricas cuyo dominio de definición es bastante reducido, por lo que el número de columnas no crece de forma exponencial y funciona bastante bien con cualquier algoritmo.

```
def dummies(data_x, data_y, data_x_tst):
  X = pd.get_dummies(data_x)
  X_tst = pd.get_dummies(data_x_tst)
  y = np.ravel(data_y.values)
  return X, y, X_tst
```

3.4. Oversampling con SMOTE.

Hace un oversample de las clases minoritarias una por una hasta igualar su cardinalidad a la de la clase mayoritaria, usando diversas ténicas para la creación de nuevas muestras. En general, el aumento de datos y balanceo de clases es bastante bueno con algoritmos que producen sobreentrenamiento, como por ejemplo **Random Forest**. Pero se ha visto que no es muy útil con otros algoritmos que no sobreentrenan tanto y que de por sí, producen mejores resultados.

```
  oversampler = sv.MulticlassOversampling()
  X_sample, y_sample = oversampler.sample(X,y)
```

3.5. Normalizar valores en un intervalo.

Podemos observar como algunas variables toman valores fuera de rango o que cuentan con *outliers*. En el caso de la edad (**age**) de los edificios, hay muy pocos que se correspondan con más de 100 años, por lo que se podría intentar normalizar esa variable en un intervalo reducido que no genere valores atípicos. Lo mismo ocurre con las variables **count_families** o **count_floors_pre_eq** las cuales de media no están definidas más allá de 3 y 4 respectivamente.

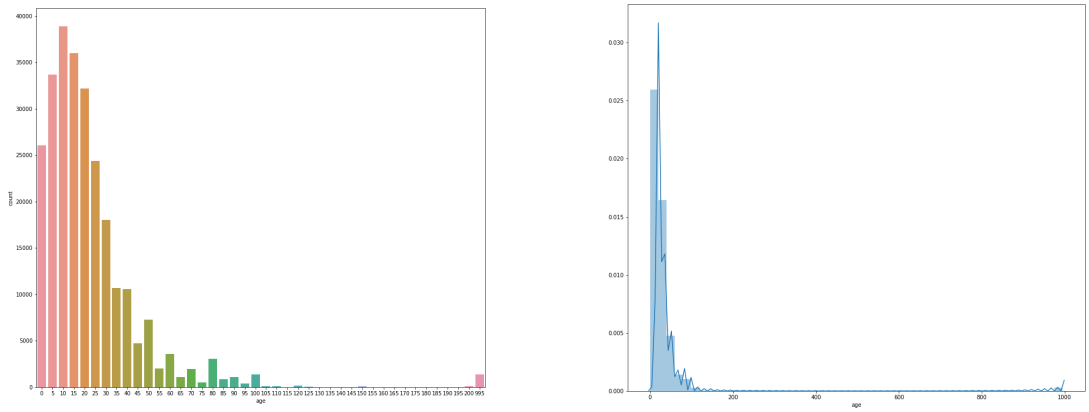

Figura 4: Variable age.

Para ello se ha utilizado el siguiente código, de forma que normalice el intervalo de manera que desaparezcan los outliers a partir de ese valor en el nuevo intervalo definido.

```
def normIntData(data, label, n):
  etq = data[label].values.tolist()
  new_etq = [n if x >= n else x for x in etq]
  data.drop([label], axis = 1, inplace = True)
  new_etq = np.array(new_etq)
  etq = (new_etq - np.mean(new_etq)) / np.std(new_etq)
  data[label] = etq.T
  return data
```

A continuación, mostraremos las gráficas de la variable **count_floors_pre_eq** y las gráficas de **age** una vez hemos normalizado el conjunto en el nuevo intervalo.

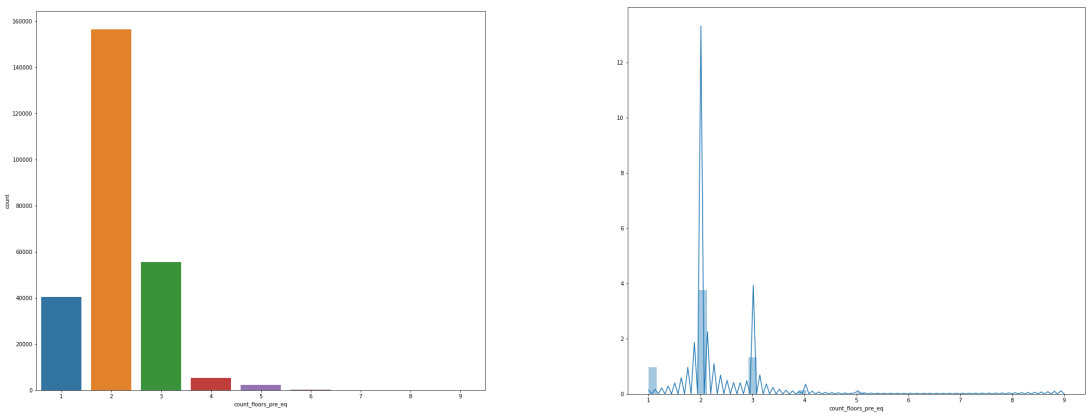

Figura 5: Variable count_floors_pre_eq.

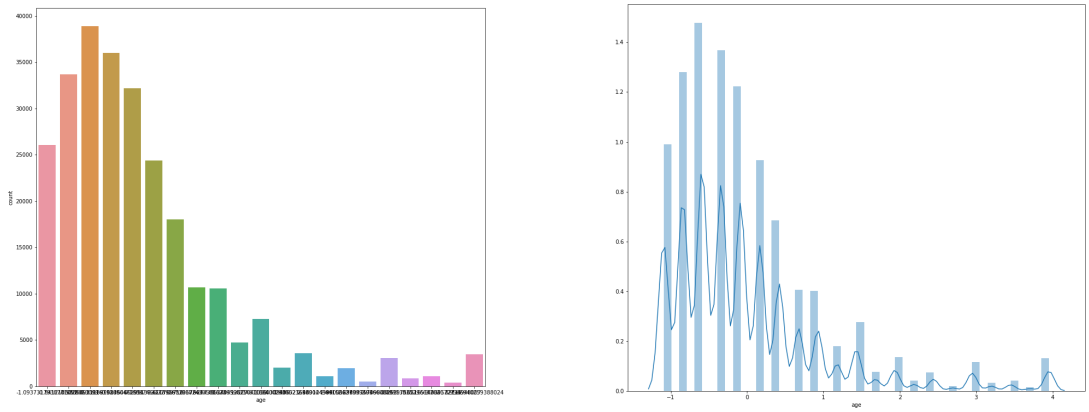

Figura 6: Variable age una vez aplicada la función.

3.6. Normalizar valores entre 0-1.

El siguiente código realiza una normalización (esta vez entre 0-1) de los valores numéricos que se pasen como parámetro. En particular se ha hecho con las variables que representan el porcentaje de la altura y área de los edificios.

```
def normData01(data, label):
  etq = data[label].values
  etq = (etq - min(etq)) / (max(etq) - min(etq))
  data.drop([label], axis = 1, inplace = True)
```

```
data[label] = etq.T
return data
```

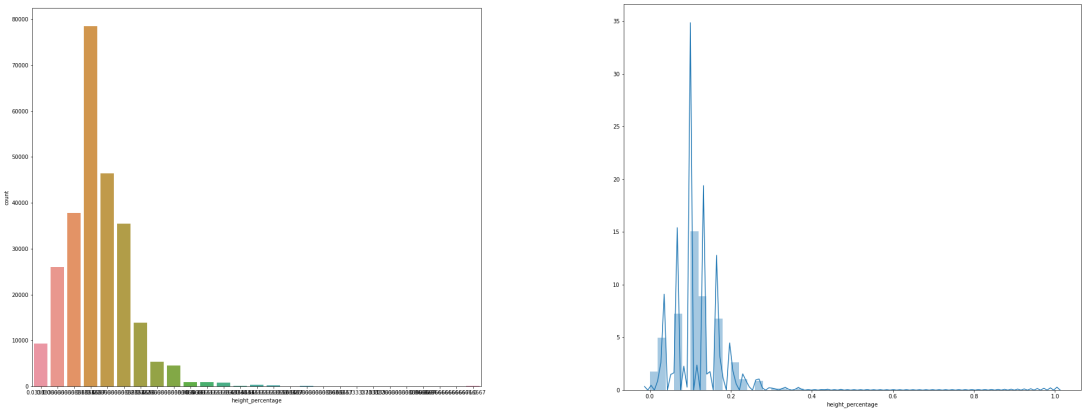

Figura 7: Variable height_percentage normalizada a 0-1.

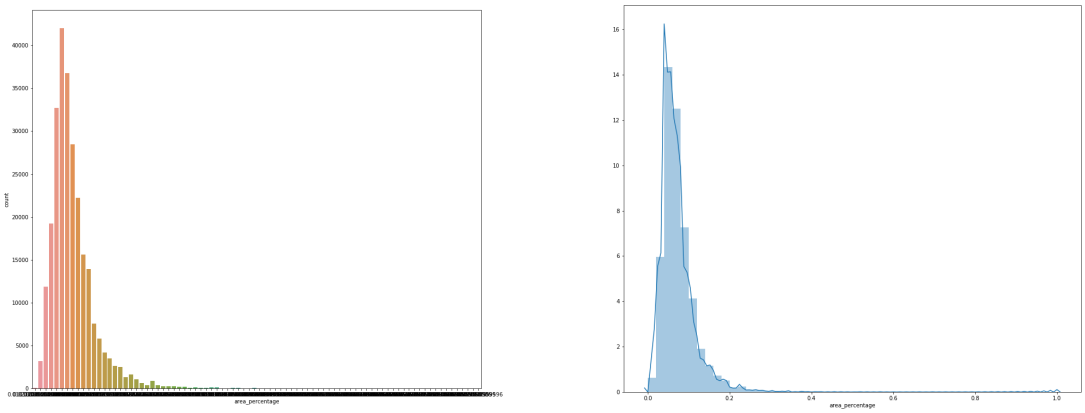

Figura 8: Variable area_percentage normalizada a 0-1.

Las pruebas realizadas nos han servido para darnos cuenta de que cualquier preprocesado más allá del **get_dummies**, perjudica en gran medida a la clasificación. Hay ciertos algoritmos que se ven beneficiados de ellos, pero en general existen muchos otros que sin preprocesamiento de ese tipo, mejoran los resultados de los primeros.

4. Validación cruzada.

Para realizar la validación cruzada he creado una función que además de implementar lo que ya venía por defecto en la plantilla, añade un parámetro para especificar el número de validaciones que queremos hacer (por defecto 5) y además muestra el score de train y test_validation para cada validación.

De esta forma, podremos acercarnos más al score que obtengamos en la web sin preocuparnos de que pueda haber sobreentrenamiento reflejado en el score para el conjunto de train.

```
def validacion_cruzada(clf, X, y, splits=5):
  cv = StratifiedKFold(n_splits=splits, shuffle=True, random_state
      =76592621)
  validation = 1
  y_train_all, y_test_all = [], []
  for train, test in cv.split(X, y):
    X_train, X_test = X[train], X[test]
    y_train, y_test = y[train], y[test]
    t = time.time()
    clf = clf.fit(X_train,y_train)
    training_time = time.time() - t
    predictions_train = clf.predict(X_train)
    predictions = clf.predict(X_test)
    print("---------- Validation ", validation, " ---------- ")
    print("Tiempo en segundos: ", training_time)
    print ("Train F1-score: ", f1_score(y_train, predictions_train,
        average='micro'))
    print ("Test F1-score: ", f1_score(y_test, predictions, average='
        micro'))
    print("")
    y_test_all = np.concatenate([y_test_all, y_test])
    y_train_all = np.concatenate([y_train_all, y_train])
    validation += 1

  return clf, y_train_all, y_test_all
```

Para las primeras pruebas se realizaba **cross-validation** para 5 validaciones pero conforme el tiempo de ejecución iba aumentando, era insostenible realizar las 5 validaciones para cada prueba por lo que se optó por pasar directamente el algoritmo al modelo y hacer las predicciones sin validaciones. Según se ha visto en la práctica, este proceso ha mejorado el valor de F1-score en la mayoría de ocasiones.

5. Algoritmos utilizados.

A continuación se comentarán los tres algoritmos utilizados en la práctica y se hará una breve reseña de sus parámetros y uso a lo largo de la misma.

5.1. Random Forest.

Para **Random Forest**, utilicé el preprocesado de **catToNum** que venía por defecto en la plantilla y apliqué un oversampling con SMOTE para intentar reducir el overfitting. La configuración de la mejor solución que obtuve fue la siguiente:

```
rfm = RandomForestClassifier(max_features = 'sqrt', criterion='gini',
    n_estimators=400, max_depth=23, random_state=76592621, n_jobs=-1)
```

En general, pienso que se podría haber optimizado mejor este algoritmo, pero su principal desventaja es que sobreentrena mucho y para una competición donde contamos con 3 subidas diarias y tienes que descartar pruebas en función de un valor sobre el entrenamiento, supone un gran problema.

Además usar **GridSearch** o **RandomSearch** con un algoritmo con alto overfitting es poco recomendable debido a que no siempre obtienes unos buenos parámetros que te proporcionen soluciones fiables al subirlo a la web.

5.2. Catboost.

Este algoritmo presenta una mejora respecto a **Random Forest** puesto que ofrece unos resultados bastante buenos pero sin generar tanto sobreentrenamiento. Para obtener unos resultados buenos es necesario pasar a la función **fit** del modelo, una lista con las posiciones de las variables categóricas del conjunto:

```
index_categories = np.where(data_x.dtypes =='object')[0]
```

Además, es recomendable eliminar el preprocesado de **catToNum**, puesto que Catboost funciona mejor con variables categóricas. Para que no de error al subir el *submission* generado a la web con **Catboost**, es necesario pasar las predicciones de float a valores enteros:

```
y_pred_tst = y_pred_tst.astype(np.uint8)
```

El preprocesado con SMOTE, en general no ofrecía mejores resultados. Además se pudo observar que lo que mejoraba los resultados era aumentar el número de estimadores (a partir de 1000). La configuración de la mejor solución que obtuve fue la siguiente:

```
cbc = CatBoostClassifier(n_estimators=1000, learning_rate=0.12,
    eval_metric='TotalF1', od_pval=0.001, random_seed=76592621,
    bootstrap_type='MVS', max_depth=12, best_model_min_trees=250,
    loss_function='MultiClass')
```

Además usé **shuffle** para mezclar las muestras antes de entrenar lo que se reflejó en una pequeña mejora sobre varias pruebas realizadas.

Para este algoritmo realicé diversas pruebas con **GridSearch** para intentar ajustar los parámetros a la mejor solución posible:

```
fit_alg = CatBoostClassifier(eval_metric='TotalF1', od_pval=0.001,
    random_seed=76592621, loss_function='MultiClass', cat_features=
    index_categories)
param_dist = {
'max_depth': [12,13],
'n_estimators': [700, 800, 900, 1000],
'learning_rate': [0.12, 0.11, 0.1],
'best_model_min_trees':[250, 300, 280]
}
clf = GridSearchCV(fit_alg, param_dist, verbose=1, cv=2, scoring='
    f1_micro', n_jobs=2)
clf = clf.fit(X_train,y_train, index_categories)
```

Para mostrar por pantalla el mejor valor para cada parámetro procedemos de la siguiente forma:

```
best_param1 = clf.best_params_['max_depth']
best_param2 = clf.best_params_['n_estimators']
best_param3 = clf.best_params_['learning_rate']
best_param4 = clf.best_params_['best_model_min_trees']
print ("Mejor valor para max_depth: ", best_param1)
print ("Mejor valor para n_estimators: ", best_param2)
print ("Mejor valor para learning_rate: ", best_param3)
print ("Mejor valor para best_model_min_trees: ", best_param4)
```

5.3. LightGBM.

Este algoritmo ofrece unos resultados bastante buenos a partir del ajuste de parámetros con **GridSearch** y el preprocesado de **get_dummies**, ofreciendo un score cercano al 0.74 simplemente con un valor alto de estimadores (más de 1000). La configuración de la mejor solución que obtuve fue la siguiente:

```
lgbm = lgb.LGBMClassifier(objective='multiclassova', n_estimators=2695,
    n_jobs=2, num_leaves=32, max_bin=300, random_state=76592621,
    learning_rate=0.09)
```

Fueron necesarios diversas pruebas con **GridSearch** hasta obtener unos buenos valores para cada parámetro. En la documentación de **LightGBM**, podemos encontrar un apartado que habla sobre el ajuste de parámetros y que fue clave para las diversas pruebas con **GridSearch**. [5]

Existen varios parámetros para hacer más rápido el algoritmo, mejorar el score y reducir el overfitting. En mi caso, la rapidez no era indispensable, por lo que me centré en buscar un equilibrio entre mejorar el score y reducir el overfitting. Algunos parámetros que mejoran la puntuación al aumentarlos son **max_bin** o **num_leaves**. Un valor muy alto para estos parámetros, también puede producir overfitting, por lo que hay que buscar un valor adecuado para ambos. Además la documentación recomienda usar un gran número de **estimadores** con un **learning_rate** bajo.

Una vez conseguido un algoritmo con un buen ajuste de parámetros, se utilizó un algoritmo de **Bagging** para mejorar los resultados:

```
clf = BaggingClassifier(base_estimator=lgbm, n_estimators=35, n_jobs=2,
    bootstrap=False, random_state=76592621)
```

Este algoritmo realiza un número establecido de ejecuciones (**n_estimators**) con el algoritmo que se le pase como **base_estimator** en nuestro caso el **LightGBM** anterior y a partir del promedio o voting establece una nueva predicción en base de las predicciones obtenidas en cada ejecución. Utilizaremos **bootstrap**=False para realizar muestreo sin reemplazo.

La primera prueba usando **LighGBM+Bagging** nos dió un resultado sobre el 0.7475 en la web, que es un resultado bastante bueno. Pero mejoró cuando aumenté a 40 el número de hojas del algoritmo base. Esto hizo que el sobreentrenamiento producido al elevar **num_leaves** a 40, se equilibre con el método utilizado por el **Bagging**. De esta manera, hemos obtenido un resultado muy superior al anterior: 0.7486.

6. Progreso realizado.

El progreso realizado ya se ha ido explicando con los algoritmos utilizados pero se hará un resumen del camino recorrido durante esta práctica:

El cambio de algoritmos se ha ido realizando conforme observaba que un algoritmo no podía mejorar más con los medios que tenía o encontraba algún algoritmo o preprocesado que podía mejorar la puntuación actual.

Se realizó una prueba con **StackingClassifier** que mezcla varios algoritmos de diferentes tipos y que con un meta clasificador de regresión logística obtiene la predicción final utilizando las ventajas de cada uno, pero no fue muy buena idea en mi caso.

Como tenía unos buenos parámetros para **LightGBM** la mejor opción fue utilizar el algoritmo de **Bagging**. Probablemente si hubiese podido hacer pruebas antes con **LightGBM+Bagging** podría haber mejorado la puntuación añadiendo más estimadores a **BaggingClassifier** o usando voting por ejemplo.

7. Tabla de submissions.

Fecha	Pos.	Score tra.	Score tst.	Preprocesado	Algoritmos	Parámetros
12/12 10:01	334	0.7264	0.6883	CatToNum +crossVal(5)	LightGBM	n_estimators=200, n_jobs=2 objective='regression_l1'
22/12 21:43	352	0.8322	0.6949	CatToNum +crossVal(5)	RandomForest	n_estimators=130, max_depth=22 random_state=5
22/12 22:16	346	0.8480	0.6969	CatToNum +crossVal(5)	RandomForest+ RandomSearch	n_estimators=200, max_depth=23 random_state=5
22/12 23:56	346	0.8777	!	CatToNum +crossVal(5)	Catboost	iteration=80, learning_rate=1 depth=15, loss_function='MultiClass'
23/12 19:16	349	0.8777	0.6657	CatToNum +crossVal(5)	Catboost	iteration=80, learning_rate=1 depth=15, loss_function='MultiClass'
23/12 20:57	347	0.8478	0.6972	CatToNum +crossVal(5)	RandomForest+ RandomSearch	n_estimators=500, max_depth=23 random_state=5
23/12 22:28	347	0.9021	0.6953	CatToNum+ MultiClassOversampling +crossVal(5)	RandomForest	n_estimators=400, max_depth=23 random_state=DNI, class_weight='balanced' criterion='gini', n_jobs=-1

Tabla 1: Tabla submissions 1.

Fecha	Pos.	Score tra.	Score tst.	Preprocesado	Algoritmos	Parámetros
24/12 13:05	354	0.918	0.6974	CatToNum+ MultiClassOversampling (proportion=0.25) +crossVal(5)	RandomForest	n_estimators=400, max_depth=23 random_state=DNI, criterion='gini', n_jobs=-1
24/12 13:06	353	0.919	0.6975	CatToNum+ MultiClassOversampling (proportion=0.75) +crossVal(5)	RandomForest	n_estimators=400, max_depth=23 random_state=DNI, criterion='gini', n_jobs=-1
24/12 13:46	353	0.8747	0.6561	CatToNum+ MultiClassOversampling (proportion=0.75) +crossVal(2)	Catboost	n_estimators=450, learning_rate=0.12, eval_metric='TotalF1', random_seed=DNI, bootstrap_type='Bayesian', max_depth=12, best_model_min_trees=250
25/12 15:19	359	0.8815	0.6406	CatToNum+ MultiClassOversampling (proportion=0.25) +crossVal(2)	Catboost	n_estimators=600, learning_rate=0.12, eval_metric='TotalF1', random_seed=DNI, bootstrap_type='MVS', max_depth=12, best_model_min_trees=250
25/12 15:28	351	0.7912	0.7003	CatToNum +sin crossVal	Catboost+ GridSearch	n_estimators=450, learning_rate=0.12, eval_metric='TotalF1', random_seed=DNI, bootstrap_type='MVS', max_depth=12, best_model_min_trees=250
25/12 15:49	351	0.8694	0.6307	CatToNum+ MultiClassOversampling (proportion=0.5) +sin crossVal	Catboost	n_estimators=450, learning_rate=0.12, eval_metric='TotalF1', random_seed=DNI, bootstrap_type='MVS', max_depth=12, best_model_min_trees=250
26/12 13:25	357	0.8010	0.7011	CatToNum+ shuffle+train_tst_split (proportion=0.5) +sin crossVal	Catboost	n_estimators=450, learning_rate=0.12, eval_metric='TotalF1', random_seed=DNI, bootstrap_type='MVS', max_depth=12, best_model_min_trees=250
26/12 15:15	223	0.7836	0.7301	categóricas+ shuffle+train_tst_split (proportion=0.5) +sin crossVal	Catboost	n_estimators=450, learning_rate=0.12, eval_metric='TotalF1', random_seed=DNI, bootstrap_type='MVS', max_depth=12, best_model_min_trees=250
26/12 17:57	201	0.8231	0.7349	categóricas+ shuffle+train_tst_split (proportion=0.5) +sin crossVal	Catboost+ GridSearch	n_estimators=1000, learning_rate=0.12, eval_metric='TotalF1', random_seed=DNI, bootstrap_type='MVS', max_depth=12, best_model_min_trees=250

Tabla 2: Tabla submissions 2.

Fecha	Pos.	Score tra.	Score tst.	Preprocesado	Algoritmos	Parámetros
27/12 22:12	205	0.8199	0.7324	categóricas+ shuffle+train_tst_split (proportion=0.5) +sin crossVal	Catboost	n_estimators=750, learning_rate=0.105, eval_metric='TotalF1', random_seed=DNI, bootstrap_type='MVS', max_depth=13
27/12 22:13	167	0.7722	0.7397	get_dummies +sin crossVal	LightGBM	n_estimators=1000, learning_rate=0.1, objective='multiclass', random_seed=DNI
27/12 22:14	167	0.7523	0.7320	get_dummies +sin crossVal	LightGBM	n_estimators=510, learning_rate=0.08, objective='multiclassova', random_seed=DNI, num_leaves=40 class_weight={1:0.9,2:0.8,3:0.7}
28/12 12:36	109	0.8153	0.7455	get_dummies +sin crossVal	LightGBM	n_estimators=2700, learning_rate=0.09, objective='multiclassova', random_seed=DNI
28/12 15:47	109	0.8210	0.7452	get_dummies +sin crossVal	LightGBM	n_estimators=2690, learning_rate=0.09, objective='multiclassova', random_seed=DNI, num_leaves=34
28/12 16:01	88	0.8174	0.7469	get_dummies +sin crossVal	LightGBM	n_estimators=2690, learning_rate=0.09, objective='multiclassova', max_bin=300, random_seed=DNI, num_leaves=32
29/12 10:30	91	0.8183	0.7466	get_dummies +sin crossVal	LightGBM	n_estimators=2695, learning_rate=0.09, objective='multiclassova', max_bin=400, random_seed=DNI, num_leaves=32
29/12 11:01	91	0.8869	0.7453	get_dummies+ MulticlassOversampling(0.5) +sin crossVal	LightGBM	n_estimators=2700, learning_rate=0.08, objective='multiclassova', max_bin=350, random_seed=DNI, num_leaves=34
29/12 11:34	91	0.8897	0.7459	get_dummies+ MulticlassOversampling(0.5) +sin crossVal	LightGBM	n_estimators=3000, learning_rate=0.08, objective='multiclassova', max_bin=350, random_seed=DNI, num_leaves=33
30/12 00:00	93	0.8181	0.7470	get_dummies+ normalizado+truncar +sin crossVal	LightGBM+ GridSearch	n_estimators=2695, learning_rate=0.09, objective='multiclassova', max_bin=300, random_seed=DNI, num_leaves=32
30/12 00:01	93	0.8105	0.7378	get_dummies+ normalizado+truncar+ +preprocesado ids + sin crossVal	LightGBM	n_estimators=2695, learning_rate=0.09, objective='multiclassova', max_bin=300, random_seed=DNI, num_leaves=32
30/12 01:43	93	0.8214	0.7395	get_dummies+ normalizado+truncar+ +sin crossVal	LightGBM+ Catboost+ LogisticRegression	mejores parámetros obtenidos a lo largo de la práctica para cada algoritmo
31/12 00:00	51	0.8183	0.7475	get_dummies+ sin crossVal	LightGBM+ BaggingClassifier	n_estimators=2695, learning_rate=0.09, objective='multiclassova', max_bin=300, random_seed=DNI, num_leaves=32 Bagging: n_estimators=35
31/12 00:00	51	0.8329	0.7486	get_dummies+ sin crossVal	LightGBM+ BaggingClassifier	n_estimators=2695, learning_rate=0.09, objective='multiclassova', max_bin=300, random_seed=DNI, num_leaves=40 Bagging: n_estimators=35
31/12 18:29	55	0.8224	0.7478	get_dummies+ sin crossVal	LightGBM+ BaggingClassifier	n_estimators=2695, learning_rate=0.09, objective='multiclassova', max_bin=300, random_seed=DNI, num_leaves=34 Bagging: n_estimators=45

Tabla 3: Tabla submissions 3.

Referencias

[1] Diapositivas de clase.

[2] https://www.drivendata.org/competitions/57/nepal-earthquake/page/136/

[3] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html

[4] https://smote-variants.readthedocs.io/en/latest/multiclass_oversampling.html

[5] https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html

[6] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html