Clasificación del conjunto de datos *Chess King-Rook vs King* usando redes neuronales y árboles de decisión con *TensorFlow*

Antonio Ruiz Molero

June 14, 2021

1 Abstract

En este documento utilizamos dos tipos de algoritmos de Machine Learning: redes neuronales *feedforward* y árboles de clasificación para clasificar el conjunto de datos *Chess (King-Rook vs. King) Data Set.* Dado el desbalanceo que existe entre las clases de la columna a clasificar, también utilizamos el algoritmo *SMOTE* para homogeneizar el conjunto.

Todas las pruebas se realizan con TensorFlow con el fin de implementar técnicas vistas en clase. Se estudian distintas configuraciones de redes neuronales como el número de capas y profundidad de las mismas, y el uso de técnicas como *Dropout* para evitar el sobreentrenamiento. Los árboles de clasificación son implementados con una nueva *API* de Tensorflow: *TensorFlowDecisionTrees.* Comparamos el rendimiento de los árboles de clasificación de esta librería con los de *Scikit-Learn* y las redes neuronales anteriores.

Se puede acceder a los archivos desde el siguiente repositorio: `https://github.com/Rumoa/Optional_tensorflow`

Contents

2 Análisis exploratorio de datos

El conjunto de datos con el que trabajaremos es una recopilación de situaciones de ajedrez en las cuales el rey y la torre blanca intentan dar mate al rey negro en un determinado número de jugadas. El conjunto se puede obtener en https://archive.ics.uci.edu/ml/datasets/Chess+%28King-Rook+vs.+King%29 .

El objetivo es determinar, a partir de las posiciones del rey y la torre negra, el número de movimientos necesario para dar mate al rey negro. Es posible que la configuración de piezas no dé lugar a un mate y acabe en tablas. Además, todos los datos están generados asumiendo un estilo de juego óptimo usando el estimador de teoría de juegos *Minimax*.

Nuestro conjunto de datos está formado por 7 columnas, de las cuales las 6 primeras serán las variables predictoras y la última la variable a predecir:

1. Columna del rey blanco (wkc)
2. Fila del rey blanco (wkr)
3. Columna de la torre blanca (wrc)
4. Fila de la torre blanca (wrr)
5. Columna del rey negro (bkc)
6. Fila del rey negor (bkr)
7. Número de movimientos óptimos para que ganen las blancas. Varían del 0 al 16 más la posibilidad de empate.

Nuestro trabajo será construir modelos que sean capaces de predecir el número de movimientos óptimos para ganar a partir de las posiciones de las tres piezas.

[11]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
```

Importamos los datos usando pandas:

[12]:
```python
data = pd.read_csv("krkopt.data", header=None)
data.columns = ["wkc", "wkr", "wrc", "wrr", "bkc", "bkr", "opt rank" ]
```

Veamos la estructura del conjunto:

[13]:
```python
data
```

[13]:
```
      wkc  wkr wrc  wrr bkc  bkr opt rank
0       a    1   b    3   c    2     draw
1       a    1   c    1   c    2     draw
2       a    1   c    1   d    1     draw
3       a    1   c    1   d    2     draw
4       a    1   c    2   c    1     draw
…     ..   …  ..   …  ..   …        …
28051   b    1   g    7   e    5  sixteen
28052   b    1   g    7   e    6  sixteen
28053   b    1   g    7   e    7  sixteen
28054   b    1   g    7   f    5  sixteen
28055   b    1   g    7   g    5  sixteen

[28056 rows x 7 columns]
```

Existe un total de 28056 casos para 7 columnas.

Comprobamos si existen valores perdidos:

[14]:
```
np.sum(data.isnull())
```

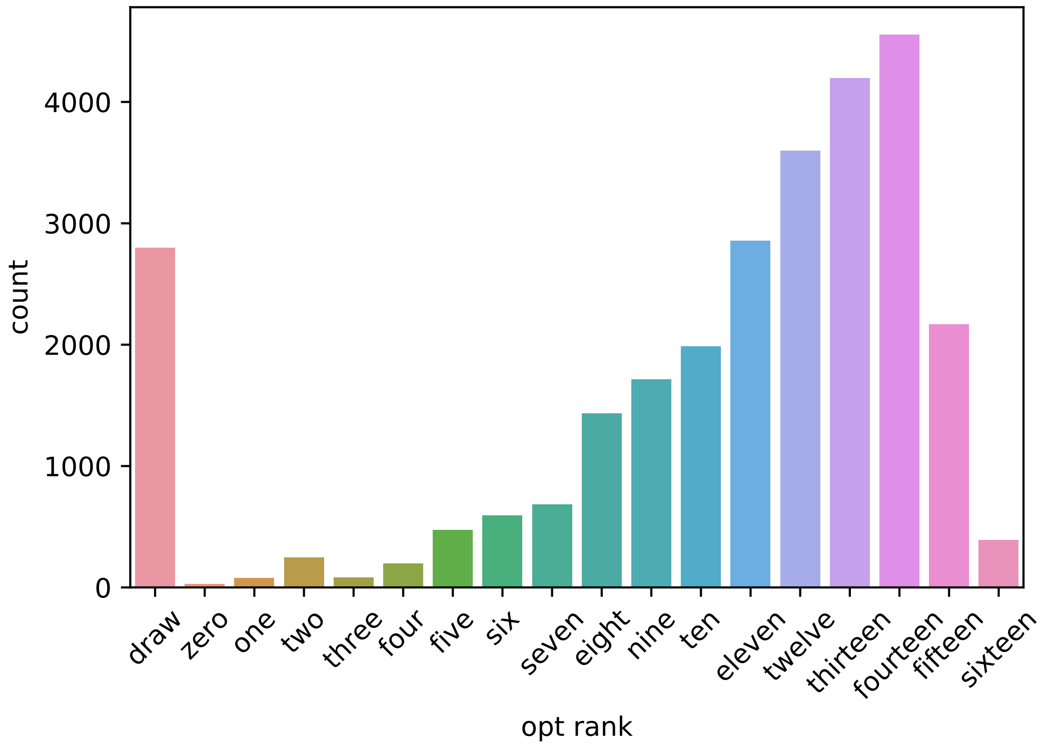
[14]:
```
wkc         0
wkr         0
wrc         0
wrr         0
bkc         0
bkr         0
opt rank    0
dtype: int64
```

No existen valores omitidos, por lo que podemos usar todos los casos sin ningún problema.

Estamos interesados en predecir los valores de *otp rank*, nos preguntamos cuál es la distribución de los casos para esta columna:

[15]:
```
plt.xticks(rotation=45)
sns.countplot(x='opt rank',
             data=data)
```

[15]:
```
<AxesSubplot:xlabel='opt rank', ylabel='count'>
```



[16]:
```python
from collections import Counter
counter = Counter(data['opt rank'])
for k,v in counter.items():
        per = v / len(data['opt rank']) * 100
        print('Class=%s, n=%d (%.3f%%)' % (k, v, per))
```

```
Class=draw, n=2796 (9.966%)
Class=zero, n=27 (0.096%)
Class=one, n=78 (0.278%)
Class=two, n=246 (0.877%)
Class=three, n=81 (0.289%)
Class=four, n=198 (0.706%)
Class=five, n=471 (1.679%)
Class=six, n=592 (2.110%)
Class=seven, n=683 (2.434%)
Class=eight, n=1433 (5.108%)
Class=nine, n=1712 (6.102%)
Class=ten, n=1985 (7.075%)
Class=eleven, n=2854 (10.173%)
Class=twelve, n=3597 (12.821%)
Class=thirteen, n=4194 (14.949%)
```

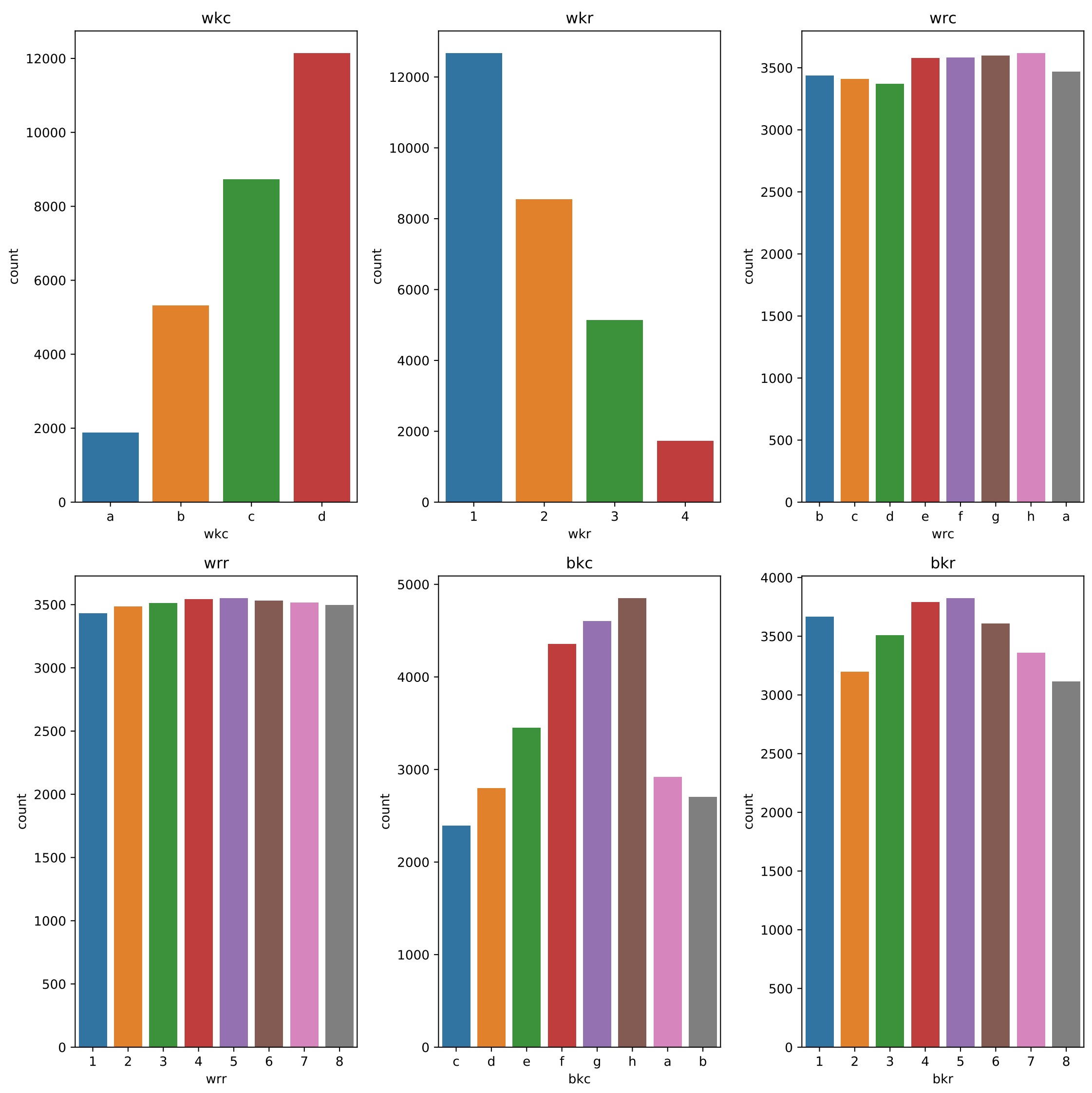
```
Class=fourteen, n=4553 (16.228%)
Class=fifteen, n=2166 (7.720%)
Class=sixteen, n=390 (1.390%)
```

Tal y como vemos, se trata de un problema de clasificación desbalanceada. Existe una gran tendencia en la distribución a necesitar un número de entre diez y catorce pasos para acabar la partida. Destacamos, de igual manera, la importante cantidad de veces que acaba en tablas. Hay muy poca representanción de partidas que puedan acabar en el rango de uno a siete movimientos.

Este tipo de conjuntos de datos son problemáticos ya que el gran desbalanceo existente entre las clases impide que los modelos construidos consigan buenas métricas en la clasificación. Por ello, usaremos, además de los datos *raw*, el algoritmo SMOTE, que realiza un sobremuestreo de los datos. Su funcionamiento es parecido a la interpolación, pero mucho más complejo. Esta técnica funciona mejor en problemas binarios con predictores continuos, donde la interpolación es más natural. Sin embargo, como también acepta datos categóricas, vemos interesante su uso y ver qué resultados ofrece.

Continuamos con la distribución de las demás clases en cada variable predictora:

[18]:
```python
f, ax = plt.subplots(2,3, figsize=(12,12) )
sns.countplot(ax= ax[0,0], x='wkc',
             data=data)
sns.countplot(ax= ax[0,1], x='wkr',
             data=data)
sns.countplot(ax= ax[0,2], x='wrc',
             data=data)
sns.countplot(ax= ax[1,0], x='wrr',
             data=data)
sns.countplot(ax= ax[1,1], x='bkc',
             data=data)
sns.countplot(ax= ax[1,2], x='bkr',
             data=data)



ax[0,0].set_title("wkc")
ax[0,1].set_title("wkr")
ax[0,2].set_title("wrc")
ax[1,0].set_title("wrr")
ax[1,1].set_title("bkc")
ax[1,2].set_title("bkr")

plt.tight_layout()
plt.savefig('foo.pdf', transparent=True)
```



Existe una mayor homogeneidad en las variables predictoras correspondiente a la posición de la torres. Es curioso que el rey blanco suele estar en la columna *d* en la primera fila, mientras que el rey negro presenta una distribución mucho más variada.

Todavía no hemos realizado ningúna conversión de los tipos ya que preferimos aplazarlo hasta la aplicación de determinados algoritmos, que nos exijan la codificación de las columnas en un deterimnado tipo.

2.1 Importancia de variables

Podemos utilizar algoritmos basados en árboles de decisión como *RandomForest* para estimar qué variables predictoras importan más a la hora de determinar la variable a predecir.

Para ello utilizaremos el paquete *Scikit-learn*. En primer lugar codificaremos numéricamente las variables categóricas.

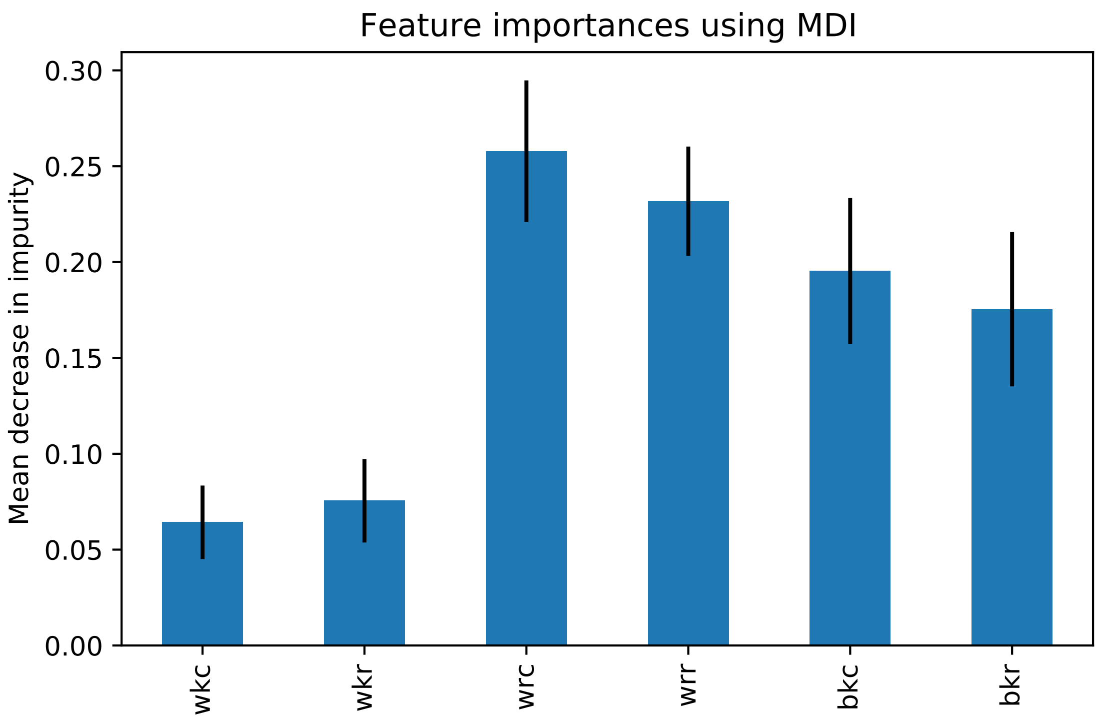
```
[19]: data_aux = data.copy()
      data_aux[['wkc', 'wrc', 'bkc']] = data_aux[['wkc', 'wrc', 'bkc']].
       ↪astype('category')

      data_aux['wkc'] = data_aux['wkc'].cat.codes
      data_aux['wrc'] = data_aux['wrc'].cat.codes
      data_aux['bkc'] = data_aux['bkc'].cat.codes
```

Entrenamos un modelo de *RandomForest*:

```
[20]: from sklearn.ensemble import RandomForestClassifier
      from sklearn.model_selection import train_test_split

      X_train, X_test, y_train, y_test = train_test_split(data_aux.drop("opt␣
       ↪rank", axis=1), data_aux['opt rank']
                                                                ,test_size=0.2)
      feature_names = [f'feature {i}' for i in range(data_aux.drop('opt rank',␣
       ↪axis=1).shape[1])]

      forest = RandomForestClassifier(random_state=0)
      forest.fit(X_train, y_train)
```

```
[20]: RandomForestClassifier(random_state=0)
```

```
[22]: importances = forest.feature_importances_
      std = np.std([
          tree.feature_importances_ for tree in forest.estimators_], axis=0)

      forest_importances = pd.Series(importances, index=feature_names)

      fig, ax = plt.subplots()
      forest_importances.plot.bar(yerr=std, ax=ax)
      ax.set_title("Feature importances using MDI")
      ax.set_ylabel("Mean decrease in impurity")
      ax.set_xticks([0,1,2,3,4,5]) # values
      labels = list(data_aux.columns[:-1])
      ax.set_xticklabels(labels) # labels
      positions = (1, 2, 3,4, 5)



      fig.tight_layout()
```



Las variables más importantes a la hora de determinar el número de pasos son las coordenadas de la torre junto a las coordenadas del rey negro. Parece que la posición del rey blanco no es tan importante como el papel de la torre a la hora de ir *acorralando* al rey negro.

Podemos dar por finalizado el análisis exploratorio de datos, ya que no necesitamos hacer ningún test estadístico o análisis continuo de los datos debido a la naturaleza categórica de los mismos. Además, la distribución de clases por columna no sigue ninguna distribución normal, atendiendo a las figuras obtenidas.

3 Creación de modelos con redes neuronales con TensorFlow

De acuerdo a lo aprendido en la asignatura, utilizaremos TensorFlow en su versión 2.5 para crear distintos modelos de redes neuronales para abordar este problema de clasificación. Los datos que usaremos son los originales y los balanceados usando SMOTE.

Para cada conjunto crearemos distintos modelos de perceptrón multicapa en los cuales variaremos la profundidad del mismo (número de capas) así como la cantidad de neuronas por capa.

Haremos uso de técnicas como dropout y callbacks. La técnica del dropout consiste en desactivar ciertas neuronas por capa de forma aleatoria en cada época con el fin de evitar que la red neuronal sobreaprenda. Con esta idea, se debe poder conseguir redes neuronales de mayor tamaño que permitan aprender más características del conjunto de datos aún sin sobreaprender.

Los callbacks son utilidades que incluye Tensorflow que facilitan el entrenamiento de los modelos. Podemos guardar en disco el modelo que mejor resultado da en función de una métrica de control. En nuestro caso, estamos interesados en la función *EarlyStopping* que permite acabar el entrenamiento de forma automática si, tras un número determinado a elegir de épocas, la métrica que elijamos no ha mejorado. Esta ventaja es fundamental ya que nos ahorra tener que incluir el número de épocas dentro de los parámetros tuneables durante el entrenamiento.

Para realizar el entrenamiento, usamos Linux junto a una tarjeta gráfica Nvidia 1070 max-q que nos ayuda a acelerar el trabajo de forma drástica. El entorno de Linux permite la configuración de las librerías CUDA de forma sencilla.

De entre los modelos vistos en clase, nos hemos decantado por el perceptrón multicapa ya que es el que más fácilmente se adapta a este problema. Por un lado, no podemos usar las redes convolucionales, joya de la corona del deep learning actualmente, ya que se usan en reconomiento de imágenes. No sabemos cómo utilizar las capas convolucionales en este tipo de datos. Muchas funcionalidades de TensorFlow no son usadas, como *DataAugmentation*, que permite variar el conjunto de entrenamiento para que la red neuronal tenga más variedad durante el mismo. Por otro lado, los mapas autogenerativos así como las redes neuronales con funciones de base radial no tienen suficiente documentación en internet para este tipo de problemas. Las redes neuronales recurrentes no se suelen usar en estas situaciones.

3.1 Datos *raw* con *smote* y *dropout*

Importamos los datos y separamos en variables predictoras: X y variable a predecir: y.

```
[4]: import pandas as pd
     import numpy as np
     data = pd.read_csv("krkopt.data", header=None)
     data.columns = ["wkc", "wkr", "wrc", "wrr", "bkc", "bkr", "opt rank" ]
     X = data.iloc[:, 0:6]
     y = data['opt rank']
     X
```

```
[4]:       wkc  wkr wrc  wrr bkc  bkr
     0       a    1   b    3   c    2
     1       a    1   c    1   c    2
     2       a    1   c    1   d    1
     3       a    1   c    1   d    2
     4       a    1   c    2   c    1
     …     ..   …   ..   …  ..   …
     28051   b    1   g    7   e    5
     28052   b    1   g    7   e    6
```

```
28053    b    1   g    7   e    7
28054    b    1   g    7   f    5
28055    b    1   g    7   g    5

[28056 rows x 6 columns]
```

Codificamos los valores “a”, “b”, “c” de las variables categóricas a enteros para que puedan ser utilizadas por las redes neuronales.

```
[5]: X["wkc"]=X["wkc"].astype('category')
     X["wrc"]=X["wrc"].astype('category')
     X["bkc"]=X["bkc"].astype('category')
     X["wkc"]=X["wkc"].cat.codes
     X["wrc"]=X["wrc"].cat.codes
     X["bkc"]=X["bkc"].cat.codes
     y = y.astype('category')
     y = y.cat.codes
     X
```

```
[5]:        wkc  wkr  wrc  wrr  bkc  bkr
     0        0    1    1    3    2    2
     1        0    1    2    1    2    2
     2        0    1    2    1    3    1
     3        0    1    2    1    3    2
     4        0    1    2    2    2    1
     …     …   …   …   …   …   …
     28051    1    1    6    7    4    5
     28052    1    1    6    7    4    6
     28053    1    1    6    7    4    7
     28054    1    1    6    7    5    5
     28055    1    1    6    7    6    5

     [28056 rows x 6 columns]
```

```
[6]: from sklearn.preprocessing import OneHotEncoder, LabelEncoder
     from sklearn.model_selection import train_test_split
     from tensorflow import keras
     import tensorflow as tf
     from tensorflow.keras.utils import to_categorical
```

Creamos los conjuntos de entrenamiento y test con una proporción 80-20 usando herramientas de *sklearn* y utilizamos oversampling con *smote*.

```
[7]: X_train, X_test, y_train, y_test = train_test_split(X,
                                                          y, test_size=0.2,
                                                                    random_state = 1)

from imblearn.over_sampling import SMOTE
oversample = SMOTE()


X_smote, y_smote = oversample.fit_resample(X, y)

X_train_smote, X_test_smote, y_train_smote, y_test_smote =␣
 ↪train_test_split(X_smote,
                                                                    y_smote, test_size=0.2,
                                                                    random_state = 1)




y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)
y_train_smote = to_categorical(y_train_smote)
y_test_smote = to_categorical(y_test_smote)
```

Como vemos, *smote* permite utilizar datos categóricos aunque no sea lo más recomendable.

[9]:
```
X_smote
```

[9]:
```
         wkc  wkr  wrc  wrr  bkc  bkr
0          0    1    1    3    2    2
1          0    1    2    1    2    2
2          0    1    2    1    3    1
3          0    1    2    1    3    2
4          0    1    2    2    2    1
…     …    …    …    …    …    …
81949      2    3    0    1    2    1
81950      2    2    0    7    0    2
81951      2    1    0    8    0    1
81952      2    1    0    8    0    1
81953      2    3    6    1    2    1

[81954 rows x 6 columns]
```

Estandarizamos los datos de entrada usando *zscore*.

[10]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
```

```
X_test = scaler.transform(X_test)

scaler = StandardScaler().fit(X_train_smote)
X_train_smote = scaler.transform(X_train_smote)
X_test_smote = scaler.transform(X_test_smote)
```

3.1.1 Definición de funciones

Definimos tres funciones que nos serán útiles para automatizar el proceso:

- *make_my_model_multi* permite crear un perceptrón multicapa proporcionándole la estructura de la forma $[n_1, n_2, ..., n_i, ..., n_N]$ donde n_i es el número de neuronas de la capa oculta i. Además de otros parámetros como la forma de entrada, salida y la función de activación que queramos usar.
- *compile_fit_multiclass* entrena un modelo de entrada con el conjunto de entrenamiento usando un conjunto de validación 80-20 y produce predicciones con un conjunto test. Utilizamos la herramienta *ModelCheckpoint* para guardar el mejor modelo durante el entrenamiento y *EarlyStopping* para parar el entrenamiento si el valor de *loss* en el conjunto de validación no mejora en 10 épocas. Así evitamos el sobreaprendizaje.
- *compute_metrics_multiclass* calcula las métricas precision, recall, F1 y Kappa a partir de las predicciones y los valores exactos de test.
- *make_my_model_multi_dropout* es una modificación que permite crear un modelo introduciendo capas internas de dropout. La estructura de la red se introduce de la forma $[n_1, n_2, ..., n_i, ..., n_N]$ donde n_i es el número de neuronas de la capa i si escribimos un número entero o una capa dropout con un valor de desactivación n_i si introducimos un elemento de tipo carácter. Por ejemplo, [50, “0.2”, 50, “0.2”] creará unas capas ocultas de la forma: capa con 50 neuronas, dropout 0.2, capa con 50 neuronas y dropout de 0.2.

[11]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


checkpoint_filepath = '/tmp/checkpoint'
from sklearn.metrics import confusion_matrix, precision_score, \
f1_score, cohen_kappa_score, recall_score

def make_my_model_multi( units_per_layer, input_s, output_s,␣
 ↪activation_='relu'):
    model = Sequential()
    depth = len(units_per_layer)
    model.add(Dense(units_per_layer[0], activation=activation_,␣
 ↪input_shape=(input_s,)))
```

```
    for i in range(1, depth):
        model.add(Dense(units_per_layer[i], activation=activation_))
    model.add(Dense(output_s, activation = 'softmax'))

    return model




def compile_fit_multiclass(modelo, X_train, X_test, y_train, batch,␣
 ↪epochs, verbose=0):
    modelo.compile(loss='categorical_crossentropy',
             optimizer='adam',
             metrics=['accuracy'])

    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',␣
 ↪patience=10, verbose=True)

    model_checkpoint = tf.keras.callbacks.
 ↪ModelCheckpoint(filepath=checkpoint_filepath,
                                                       ␣
 ↪save_weights_only=True,
                                                       ␣
 ↪monitor='val_loss',
                                                        mode='min',
                                                       ␣
 ↪save_best_only=True,
                                                        verbose=False)

    modelo.fit(X_train, y_train, epochs=epochs, batch_size=batch,␣
 ↪verbose=verbose, validation_split=0.2, callbacks = [early_stopping,␣
 ↪model_checkpoint])
    model.load_weights(checkpoint_filepath)
    predictions = modelo.predict(X_test)
    return predictions

def compute_metrics_multiclass(y_test, y_pred):
    results=[]
    results.append(precision_score(y_test, np.round(y_pred),␣
 ↪average="micro"))
    results.append(recall_score(y_test, np.round(y_pred),␣
 ↪average="micro"))
    results.append(f1_score(y_test, np.round(y_pred), average="micro"))
    results.append(cohen_kappa_score(y_test, np.round(y_pred)))
    return results


from tensorflow.keras.layers import Dropout

def make_my_model_multi_dropout( units_per_layer, input_s, output_s,␣
 ↪activation_='relu'):
    model = Sequential()
    depth = len(units_per_layer)
    model.add(Dense(units_per_layer[0], activation=activation_,␣
 ↪input_shape=(input_s,)))
    for i in range(1, depth):
        if isinstance(units_per_layer[i], str):
            a = units_per_layer[i]
            dropout_r = float(a)
            model.add(Dropout(dropout_r))
        else:
            model.add(Dense(units_per_layer[i], activation=activation_))
    model.add(Dense(output_s, activation = 'softmax'))

    return model
```

Comprobamos que los datos tienen las dimensiones correctas

[22]:
```
X_train.shape, X_train_smote.shape, y_train.shape, y_train_smote.shape,␣
 ↪X_test.shape, y_test.shape
```

[22]:
```
((22444, 6), (65563, 6), (22444, 18), (65563, 18), (5612, 6), (5612, 18))
```

3.1.2 Pruebas

Datos *raw* Creamos un total de treinta experimentos en las que crearemos redes neuronales de la misma cantidad de neuronas por capas con distinto número de neuronas y distinto número de capas. El número de neuronas será: [50, 100, 150, 200, 250] y el tamaño variará de entre una y seis capas ocultas. Guardamos las predicciones junto a las métricas y la matriz de confusión. Escribimos en disco el objeto mediante *joblib* para su posterior análisis.

[15]:
```
results = []
seed = 1
from sklearn.metrics import confusion_matrix
```

[54]:
```
size_config = [50, 100, 150, 200, 250]
for size in size_config:
    layer_config = [[size], [size]*2, [size]*3, [size]*4, [size]*5,␣
 ↪[size]*6]
    for layers in layer_config:
        np.random.seed(seed)
        tf.random.set_seed(seed)
        print(layers)
        model = make_my_model_multi(layers, 6, 18, activation_='relu' )
        preds = compile_fit_multiclass(model, X_train, X_test, y_train,␣
 ↪256, 300, verbose=0)
        metrics = compute_metrics_multiclass(np.argmax(preds, axis = 1),␣
 ↪np.argmax(y_test, axis = 1))
        confusion = confusion_matrix(np.argmax(preds, axis = 1), np.
 ↪argmax(y_test, axis = 1))
        aux = { "layer config" : layers,
               #"Model": model,
               "Predictions" : preds,
               "Metrics" : metrics,
               "Confusion" : confusion

        }
        print(metrics)
        results.append(aux)
```

```
[50]
[0.5563079116179616, 0.5563079116179616, 0.5563079116179616, 0.
 ↪5014778486804247]
[50, 50]
Epoch 00278: early stopping
[0.6776550249465432, 0.6776550249465432, 0.6776550249465432, 0.
 ↪6394872591777784]
[50, 50, 50]
Epoch 00189: early stopping
[0.7004632929436921, 0.7004632929436921, 0.7004632929436921, 0.
 ↪665270507458l3]
[50, 50, 50, 50]
Epoch 00135: early stopping
[0.7033143264433357, 0.7033143264433357, 0.7033143264433357, 0.
 ↪6683596780441357]
[50, 50, 50, 50, 50]
Epoch 00083: early stopping
```

```
[0.7140057020669993, 0.7140057020669993, 0.7140057020669993, 0.
 ↪6802859773047576]
[50, 50, 50, 50, 50, 50]
Epoch 00101: early stopping
[0.7325374198146828, 0.7325374198146828, 0.7325374198146829, 0.
 ↪7014741703016625]
[100]
[0.5841054882394868, 0.5841054882394868, 0.5841054882394868, 0.
 ↪532926380750063]
[100, 100]
Epoch 00226: early stopping
[0.7145402708481825, 0.7145402708481825, 0.7145402708481825, 0.
 ↪6810448690167336]
[100, 100, 100]
Epoch 00145: early stopping
[0.7808267997148967, 0.7808267997148967, 0.7808267997148967, 0.
 ↪7552060507593273]
[100, 100, 100, 100]
Epoch 00117: early stopping
[0.7770848182466144, 0.7770848182466144, 0.7770848182466144, 0.
 ↪7511752665391203]
[100, 100, 100, 100, 100]
Epoch 00083: early stopping
[0.7854597291518175, 0.7854597291518175, 0.7854597291518175, 0.
 ↪7603159232769513]
[100, 100, 100, 100, 100, 100]
Epoch 00089: early stopping
[0.7861724875267284, 0.7861724875267284, 0.7861724875267283, 0.
 ↪7611696219845376]
[150]
[0.6051318602993585, 0.6051318602993585, 0.6051318602993585, 0.
 ↪5571977645455048]
[150, 150]
Epoch 00196: early stopping
[0.7462580185317177, 0.7462580185317177, 0.7462580185317177, 0.
 ↪7163812931399909]
[150, 150, 150]
Epoch 00117: early stopping
[0.7813613684960798, 0.7813613684960798, 0.7813613684960798, 0.
 ↪7561423259186348]
[150, 150, 150, 150]
Epoch 00087: early stopping
[0.7957947255880257, 0.7957947255880257, 0.7957947255880257, 0.
 ↪7718799039500605]
```

```
[150, 150, 150, 150, 150]
Epoch 00094: early stopping
[0.8209194583036351, 0.8209194583036351, 0.8209194583036351, 0.
 ↪799991822447679]
[150, 150, 150, 150, 150, 150]
Epoch 00083: early stopping
[0.8184248039914469, 0.8184248039914469, 0.8184248039914469, 0.
 ↪7973288336155344]
[200]
[0.6286528866714184, 0.6286528866714184, 0.6286528866714184, 0.
 ↪5841443006982328]
[200, 200]
Epoch 00160: early stopping
[0.7530292230933714, 0.7530292230933714, 0.7530292230933714, 0.
 ↪7243038312259522]
[200, 200, 200]
Epoch 00094: early stopping
[0.7794012829650748, 0.7794012829650748, 0.7794012829650748, 0.
 ↪7533991727673046]
[200, 200, 200, 200]
Epoch 00072: early stopping
[0.8029223093371347, 0.8029223093371347, 0.8029223093371347, 0.
 ↪7797459386027457]
[200, 200, 200, 200, 200]
Epoch 00064: early stopping
[0.8048823948681397, 0.8048823948681397, 0.8048823948681397, 0.
 ↪7819079964441699]
[200, 200, 200, 200, 200, 200]
Epoch 00061: early stopping
[0.8191375623663578, 0.8191375623663578, 0.8191375623663577, 0.
 ↪7979856944628764]
[250]
[0.6416607270135424, 0.6416607270135424, 0.6416607270135424, 0.
 ↪5990285359827496]
[250, 250]
Epoch 00145: early stopping
[0.7528510334996437, 0.7528510334996437, 0.7528510334996438, 0.
 ↪7238185938600421]
[250, 250, 250]
Epoch 00065: early stopping
[0.7694226657163221, 0.7694226657163221, 0.7694226657163221, 0.
 ↪7424900002897106]
[250, 250, 250, 250]
Epoch 00072: early stopping
```

```
[0.8161083392729864, 0.8161083392729864, 0.8161083392729864, 0.
 ↪7947878390730547]
[250, 250, 250, 250, 250]
Epoch 00060: early stopping
[0.8251960085531005, 0.8251960085531005, 0.8251960085531005, 0.
 ↪8045311760051372]
[250, 250, 250, 250, 250, 250]
Epoch 00057: early stopping
[0.8218104062722738, 0.8218104062722738, 0.8218104062722738, 0.
 ↪80115129573033]
```

[65]:
```
import joblib

joblib.dump(results, 'results_1_joblib')
```

[65]:
```
['results_1_joblib']
```

Datos con *smote* Repetimos el mismo esquema con los datos con *smote*.

[30]:
```
results_smote = []
seed = 1
```

[31]:
```
size_config = [50, 100, 150, 200, 250]
for size in size_config:
    layer_config = [[size], [size]*2, [size]*3, [size]*4, [size]*5,␣
 ↪[size]*6]
    for layers in layer_config:
        np.random.seed(seed)
        tf.random.set_seed(seed)
        print(layers)
        model = make_my_model_multi(layers, 6, 18, activation_='relu' )
        preds = compile_fit_multiclass(model, X_train_smote, X_test,␣
 ↪y_train_smote, 256, 300, verbose=0)
        metrics = compute_metrics_multiclass(np.argmax(preds, axis = 1),␣
 ↪np.argmax(y_test, axis = 1))
        confusion = confusion_matrix(np.argmax(preds, axis = 1), np.
 ↪argmax(y_test, axis = 1))
        aux = { "layer config" : layers,
               #"Model": model,
               "Predictions" : preds,
               "Metrics" : metrics,
               "Confusion" : confusion

        }
        print(metrics)
        results_smote.append(aux)
```

```
[50]
[0.2735210263720599, 0.2735210263720599, 0.2735210263720599,
0.21056283287164057]
[50, 50]
Epoch 00195: early stopping
[0.3016749821810406, 0.3016749821810406, 0.3016749821810406,
0.23858740268705947]
[50, 50, 50]
Epoch 00149: early stopping
[0.319672131147541, 0.319672131147541, 0.319672131147541, 0.
 ↪2573189896196104]
[50, 50, 50, 50]
Epoch 00150: early stopping
[0.3257305773342837, 0.3257305773342837, 0.3257305773342837,
0.26329646382574623]
[50, 50, 50, 50, 50]
Epoch 00105: early stopping
[0.32216678545972915, 0.32216678545972915, 0.32216678545972915,
0.26004197522733685]
[50, 50, 50, 50, 50, 50]
Epoch 00170: early stopping
[0.3310762651461155, 0.3310762651461155, 0.3310762651461155, 0.
 ↪2689915712533639]
[100]
[0.29971489665003564, 0.29971489665003564, 0.29971489665003564,
0.23722592685369615]
[100, 100]
Epoch 00174: early stopping
[0.32323592302209553, 0.32323592302209553, 0.32323592302209553,
0.26046775820865786]
[100, 100, 100]
Epoch 00133: early stopping
[0.3341054882394868, 0.3341054882394868, 0.3341054882394868,
0.27209407806805874]
[100, 100, 100, 100]
Epoch 00065: early stopping
[0.34052031361368496, 0.34052031361368496, 0.34052031361368496,
0.2793803604511679]
[100, 100, 100, 100, 100]
Epoch 00065: early stopping
```

```
[0.33856022808268, 0.33856022808268, 0.33856022808268, 0.2775702647152132]
[100, 100, 100, 100, 100, 100]
Epoch 00078: early stopping
[0.3447968638631504, 0.3447968638631504, 0.3447968638631504, 0.
 ↪2820060064899931]
[150]
[0.30024946543121883, 0.30024946543121883, 0.30024946543121883,
0.23828899508645218]
[150, 150]
Epoch 00145: early stopping
[0.32555238774055595, 0.32555238774055595, 0.32555238774055595,
0.26301755677369276]
[150, 150, 150]
Epoch 00095: early stopping
[0.323592302209551, 0.323592302209551, 0.323592302209551, 0.
 ↪26068327259513024]
[150, 150, 150, 150]
Epoch 00065: early stopping
[0.3469351389878831, 0.3469351389878831, 0.3469351389878831,
0.28626249332475373]
[150, 150, 150, 150, 150]
Epoch 00076: early stopping
[0.3551318602993585, 0.3551318602993585, 0.3551318602993585,
0.29535007953313264]
[150, 150, 150, 150, 150, 150]
Epoch 00089: early stopping
[0.3544191019244476, 0.3544191019244476, 0.3544191019244476, 0.
 ↪2932453533302928]
[200]
[0.31272273699215963, 0.31272273699215963, 0.31272273699215963,
0.2508736929301466]
[200, 200]
Epoch 00155: early stopping
[0.3362437633642195, 0.3362437633642195, 0.3362437633642195, 0.
 ↪2741266338548447]
[200, 200, 200]
Epoch 00072: early stopping
[0.33481824661439774, 0.33481824661439774, 0.33481824661439774,
0.2737471663026608]
[200, 200, 200, 200]
Epoch 00059: early stopping
[0.34337134711332856, 0.34337134711332856, 0.34337134711332856,
0.28165430613441933]
[200, 200, 200, 200, 200]
```

```
Epoch 00065: early stopping
[0.34746970776906627, 0.34746970776906627, 0.34746970776906627,
0.2864768326195607]
[200, 200, 200, 200, 200, 200]
Epoch 00078: early stopping
[0.35655737704918034, 0.35655737704918034, 0.3565573770491803,
0.2961375972429202]
[250]
[0.32020669992872414, 0.32020669992872414, 0.32020669992872414,
0.2589827289284935]
[250, 250]
Epoch 00116: early stopping
[0.3271560940841055, 0.3271560940841055, 0.3271560940841055,
0.26503000969397217]
[250, 250, 250]
Epoch 00084: early stopping
[0.3394511760513186, 0.3394511760513186, 0.3394511760513186, 0.
 ↪2769489283726123]
[250, 250, 250, 250]
Epoch 00068: early stopping
[0.35477548111190305, 0.35477548111190305, 0.3547754811119031,
0.29475622872925544]
[250, 250, 250, 250, 250]
Epoch 00045: early stopping
[0.34978617248752675, 0.34978617248752675, 0.34978617248752675,
0.2885785810340571]
[250, 250, 250, 250, 250, 250]
Epoch 00057: early stopping
[0.3617248752672844, 0.3617248752672844, 0.36172487526728436,
0.3008943441645957]
```

[33]:
```
joblib.dump(results_smote, 'results_smote_joblib')
```

[33]:
```
['results_smote_joblib']
```

Datos raw con dropout A la hora de realizar experimentos con dropout, creamos el mismo esquema que en casos anteriores intercalando capas dropout de 3 posibles valores: 0.1, 0.2 y 0.3.

[33]:
```
results_dropout = []
seed = 1
```

Mostramos la configuración de los experimentos. En total, realizaremos noventa casos.

```
[61]: size_config = [50, 100, 150, 200, 250]
      dropout_rate = ["0.1", "0.2", "0.3"]

      for size in size_config:
          for size_d in (dropout_rate):
              layer_config_dense = [[size], [size]*2, [size]*3, [size]*4,␣
       ↪[size]*5, [size]*6]
              layer_config_dropout = [[size_d], [size_d]*2, [size_d]*3,␣
       ↪[size_d]*4, [size_d]*5, [size_d]*6]
              for layers_dense, layers_dropout in zip(layer_config_dense,␣
       ↪layer_config_dropout):
                  final_design = [None]*(len(layers_dense)+len(layers_dropout))
                  final_design[::2] = layers_dense
                  final_design[1::2] = layers_dropout
                  print(final_design)
```

```
[50, '0.1']
[50, '0.1', 50, '0.1']
[50, '0.1', 50, '0.1', 50, '0.1']
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
[50, '0.2']
[50, '0.2', 50, '0.2']
[50, '0.2', 50, '0.2', 50, '0.2']
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
[50, '0.3']
[50, '0.3', 50, '0.3']
[50, '0.3', 50, '0.3', 50, '0.3']
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
[100, '0.1']
[100, '0.1', 100, '0.1']
[100, '0.1', 100, '0.1', 100, '0.1']
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
[100, '0.2']
[100, '0.2', 100, '0.2']
[100, '0.2', 100, '0.2', 100, '0.2']
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
```

```
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
[100, '0.3']
[100, '0.3', 100, '0.3']
[100, '0.3', 100, '0.3', 100, '0.3']
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
[150, '0.1']
[150, '0.1', 150, '0.1']
[150, '0.1', 150, '0.1', 150, '0.1']
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
[150, '0.2']
[150, '0.2', 150, '0.2']
[150, '0.2', 150, '0.2', 150, '0.2']
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
[150, '0.3']
[150, '0.3', 150, '0.3']
[150, '0.3', 150, '0.3', 150, '0.3']
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
[200, '0.1']
[200, '0.1', 200, '0.1']
[200, '0.1', 200, '0.1', 200, '0.1']
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
[200, '0.2']
[200, '0.2', 200, '0.2']
[200, '0.2', 200, '0.2', 200, '0.2']
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
[200, '0.3']
[200, '0.3', 200, '0.3']
[200, '0.3', 200, '0.3', 200, '0.3']
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
[250, '0.1']
```

```
[250, '0.1', 250, '0.1']
[250, '0.1', 250, '0.1', 250, '0.1']
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
[250, '0.2']
[250, '0.2', 250, '0.2']
[250, '0.2', 250, '0.2', 250, '0.2']
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
[250, '0.3']
[250, '0.3', 250, '0.3']
[250, '0.3', 250, '0.3', 250, '0.3']
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
```

[62]:
```
size_config = [50, 100, 150, 200, 250]
dropout_rate = ["0.1", "0.2", "0.3"]

for size in size_config:
    for size_d in (dropout_rate):
        layer_config_dense = [[size], [size]*2, [size]*3, [size]*4,␣
↪[size]*5, [size]*6]
        layer_config_dropout = [[size_d], [size_d]*2, [size_d]*3,␣
↪[size_d]*4, [size_d]*5, [size_d]*6]
        for layers_dense, layers_dropout in zip(layer_config_dense,␣
↪layer_config_dropout):
            final_design = [None]*(len(layers_dense)+len(layers_dropout))
            final_design[::2] = layers_dense
            final_design[1::2] = layers_dropout
            np.random.seed(seed)
            tf.random.set_seed(seed)
            print(final_design)
            model = make_my_model_multi_dropout(final_design, 6, 18,␣
↪activation_='relu' )
            preds = compile_fit_multiclass(model, X_train, X_test,␣
↪y_train, 256, 300, verbose=0)
            metrics = compute_metrics_multiclass(np.argmax(preds, axis =␣
↪1), np.argmax(y_test, axis = 1))
            confusion = confusion_matrix(np.argmax(preds, axis = 1), np.
↪argmax(y_test, axis = 1))
            aux = { "layer config" : final_design,
                #"Model": model,
                "Predictions" : preds,
                "Metrics" : metrics,
                "Confusion" : confusion

            }
            print(metrics)
            results_dropout.append(aux)
```

```
[50, '0.1']
[0.5306486101211689, 0.5306486101211689, 0.5306486101211689,
0.47299466324494865]
[50, '0.1', 50, '0.1']
[0.6582323592302209, 0.6582323592302209, 0.6582323592302209, 0.
↪6168927965574147]
[50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00274: early stopping
[0.7109764789736279, 0.7109764789736279, 0.7109764789736278, 0.
↪677058966107785]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00187: early stopping
[0.7066999287241625, 0.7066999287241625, 0.7066999287241625, 0.
↪6719743342215174]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00152: early stopping
[0.6903064861012117, 0.6903064861012117, 0.6903064861012117, 0.
↪6533659428663383]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00176: early stopping
[0.6915538132573058, 0.6915538132573058, 0.6915538132573058, 0.
↪6548578250956504]
[50, '0.2']
Epoch 00253: early stopping
[0.5172843905915895, 0.5172843905915895, 0.5172843905915895,
0.45668070032612573]
[50, '0.2', 50, '0.2']
[0.607448325017819, 0.607448325017819, 0.607448325017819, 0.
↪5594133149504249]
[50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00155: early stopping
[0.6181397006414825, 0.6181397006414825, 0.6181397006414825, 0.
↪5718332632332058]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
```

```
Epoch 00221: early stopping
[0.642551674982181, 0.642551674982181, 0.642551674982181, 0.
↪5987753202495634]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00135: early stopping
[0.5915894511760513, 0.5915894511760513, 0.5915894511760513, 0.
↪5418785789667531]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00100: early stopping
[0.577512473271561, 0.577512473271561, 0.577512473271561, 0.
↪5258937398977566]
[50, '0.3']
Epoch 00249: early stopping
[0.5110477548111191, 0.5110477548111191, 0.5110477548111191,
0.44936967418461227]
[50, '0.3', 50, '0.3']
Epoch 00214: early stopping
[0.5673556664290805, 0.5673556664290805, 0.5673556664290805, 0.
↪5134787826514988]
[50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00235: early stopping
[0.5616535994297933, 0.5616535994297933, 0.5616535994297933, 0.
↪5066628382810214]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00124: early stopping
[0.5342124019957234, 0.5342124019957234, 0.5342124019957234, 0.
↪47517196797274831]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00127: early stopping
[0.5263720598717035, 0.5263720598717035, 0.5263720598717035, 0.
↪4667612314609778]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00108: early stopping
[0.5226300784034212, 0.5226300784034212, 0.5226300784034212,
0.46306875130107517]
[100, '0.1']
Epoch 00249: early stopping
[0.5461511047754811, 0.5461511047754811, 0.5461511047754811,
0.49062978889937336]
[100, '0.1', 100, '0.1']
[0.7521382751247327, 0.7521382751247327, 0.7521382751247327, 0.
↪7227156116282598]
[100, '0.1', 100, '0.1', 100, '0.1']
```

```
[0.8257305773342837, 0.8257305773342837, 0.8257305773342837, 0.
 ↪8054057526695507]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00165: early stopping
[0.7993585174625801, 0.7993585174625801, 0.7993585174625801, 0.
 ↪775965204492387]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00221: early stopping
[0.8182466143977192, 0.8182466143977192, 0.8182466143977192, 0.
 ↪7970575500307149]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00199: early stopping
[0.8155737704918032, 0.8155737704918032, 0.8155737704918032, 0.
 ↪7940122031851752]
[100, '0.2']
Epoch 00249: early stopping
[0.5386671418389166, 0.5386671418389166, 0.5386671418389166,
0.48189218191606287]
[100, '0.2', 100, '0.2']
[0.7248752672843906, 0.7248752672843906, 0.7248752672843906, 0.
 ↪6919934078576849]
[100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00238: early stopping
[0.7674625801853172, 0.7674625801853172, 0.7674625801853173, 0.
 ↪740127140644157]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00191: early stopping
[0.7610477548111191, 0.7610477548111191, 0.7610477548111191, 0.
 ↪7329919066201204]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00158: early stopping
[0.7273699215965788, 0.7273699215965788, 0.7273699215965788, 0.
 ↪6954064061016627]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00089: early stopping
[0.6945830363506771, 0.6945830363506771, 0.6945830363506771, 0.
 ↪6587686427382717]
[100, '0.3']
Epoch 00214: early stopping
[0.5340342124019958, 0.5340342124019958, 0.5340342124019958,
0.47652760627391544]
[100, '0.3', 100, '0.3']
Epoch 00252: early stopping
```

```
[0.6764076977904491, 0.6764076977904491, 0.6764076977904491, 0.
 ↪637487713970418]
[100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00195: early stopping
[0.7033143264433357, 0.7033143264433357, 0.7033143264433357, 0.
 ↪6684001878880355]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00157: early stopping
[0.6847826086956522, 0.6847826086956522, 0.6847826086956522, 0.
 ↪6471773664755471]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00136: early stopping
[0.6781895937277262, 0.6781895937277262, 0.6781895937277262, 0.
 ↪6394933393602311]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00124: early stopping
[0.6370277975766215, 0.6370277975766215, 0.6370277975766215, 0.
 ↪5935140862849402]
[150, '0.1']
Epoch 00272: early stopping
[0.5620099786172488, 0.5620099786172488, 0.5620099786172488, 0.
 ↪508341442519633]
[150, '0.1', 150, '0.1']
Epoch 00266: early stopping
[0.7749465431218817, 0.7749465431218817, 0.7749465431218816, 0.
 ↪7484450378452736]
[150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00200: early stopping
[0.8355310049893087, 0.8355310049893087, 0.8355310049893087, 0.
 ↪8162182870942731]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00197: early stopping
[0.8597647897362795, 0.8597647897362795, 0.8597647897362795, 0.
 ↪8433538331608326]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00110: early stopping
[0.8241268709907341, 0.8241268709907341, 0.8241268709907341, 0.
 ↪8035528526761304]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00230: early stopping
[0.8624376336421953, 0.8624376336421953, 0.8624376336421952, 0.
 ↪846293252489121]
[150, '0.2']
```

```
[0.5563079116179616, 0.5563079116179616, 0.5563079116179616, 0.
 ↪5015463823385387]
[150, '0.2', 150, '0.2']
[0.7615823235923022, 0.7615823235923022, 0.7615823235923023, 0.
 ↪7335657779063354]
[150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00204: early stopping
[0.8095153243050606, 0.8095153243050606, 0.8095153243050606, 0.
 ↪7871495134285849]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00264: early stopping
[0.8342836778332146, 0.8342836778332146, 0.8342836778332146, 0.
 ↪8149513387287599]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00148: early stopping
[0.8022095509622238, 0.8022095509622238, 0.8022095509622238, 0.
 ↪778780229359274]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00098: early stopping
[0.7696008553100498, 0.7696008553100498, 0.7696008553100498, 0.
 ↪7424088375196287]
[150, '0.3']
Epoch 00249: early stopping
[0.5411617961511048, 0.5411617961511048, 0.5411617961511048, 0.
 ↪4847149956069786]
[150, '0.3', 150, '0.3']
[0.7403777619387027, 0.7403777619387027, 0.7403777619387029, 0.
 ↪7093364064692692]
[150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00248: early stopping
[0.785816108339273, 0.785816108339273, 0.785816108339273, 0.
 ↪7605921518542224]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00140: early stopping
[0.7521382751247327, 0.7521382751247327, 0.7521382751247327, 0.
 ↪722902071589828]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00187: early stopping
[0.7599786172487527, 0.7599786172487527, 0.7599786172487527, 0.
 ↪731518863771794]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00147: early stopping
[0.7337847469707769, 0.7337847469707769, 0.733784746970777, 0.
 ↪7018631579115805]
```

```
[200, '0.1']
[0.5771560940841055, 0.5771560940841055, 0.5771560940841055, 0.
↪5250361092757381]
[200, '0.1', 200, '0.1']
Epoch 00227: early stopping
[0.7902708481824662, 0.7902708481824662, 0.7902708481824661, 0.
↪765865311916265]
[200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00246: early stopping
[0.8709907341411262, 0.8709907341411262, 0.8709907341411262, 0.
↪8558632550430805]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00150: early stopping
[0.8661796151104776, 0.8661796151104776, 0.8661796151104776, 0.
↪8504775774300332]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00171: early stopping
[0.8736635780470421, 0.8736635780470421, 0.8736635780470421, 0.
↪858903886573265]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00143: early stopping
[0.8647540983606558, 0.8647540983606558, 0.8647540983606559, 0.
↪8489033371956115]
[200, '0.2']
[0.5718104062722738, 0.5718104062722738, 0.5718104062722738, 0.
↪5193521921585287]
[200, '0.2', 200, '0.2']
[0.7906272273699216, 0.7906272273699216, 0.7906272273699216, 0.
↪7660980388359336]
[200, '0.2', 200, '0.2', 200, '0.2']
[0.8549536707056308, 0.8549536707056308, 0.8549536707056308, 0.
↪8381041473725426]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00131: early stopping
[0.8232359230220955, 0.8232359230220955, 0.8232359230220955, 0.
↪8025074171854785]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00219: early stopping
[0.8558446186742694, 0.8558446186742694, 0.8558446186742694, 0.
↪8391218900502941]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00152: early stopping
[0.8330363506771205, 0.8330363506771205, 0.8330363506771205, 0.
↪8132935419648581]
```

```
[200, '0.3']
Epoch 00253: early stopping
[0.5481111903064861, 0.5481111903064861, 0.5481111903064861, 0.
 ↪4926703109101981]
[200, '0.3', 200, '0.3']
Epoch 00268: early stopping
[0.7617605131860299, 0.7617605131860299, 0.7617605131860298, 0.
 ↪7338285623633978]
[200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00245: early stopping
[0.8200285103349965, 0.8200285103349965, 0.8200285103349965, 0.
 ↪7989672711956026]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00211: early stopping
[0.8134354953670706, 0.8134354953670706, 0.8134354953670706, 0.
 ↪7917231289447606]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00133: early stopping
[0.770848182466144, 0.770848182466144, 0.770848182466144, 0.
 ↪7438576324656899]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00159: early stopping
[0.7790449037776194, 0.7790449037776194, 0.7790449037776194, 0.
 ↪7530774131803226]
[250, '0.1']
[0.5851746258018532, 0.5851746258018532, 0.5851746258018532, 0.
 ↪5348236817948813]
[250, '0.1', 250, '0.1']
Epoch 00251: early stopping
[0.8057733428367784, 0.8057733428367784, 0.8057733428367784, 0.
 ↪7828834924115987]
[250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00150: early stopping
[0.8490734141126158, 0.8490734141126158, 0.8490734141126158, 0.
 ↪8314265685216257]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00113: early stopping
[0.8610121168923734, 0.8610121168923734, 0.8610121168923734, 0.
 ↪8447712258954628]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00109: early stopping
[0.8740199572344975, 0.8740199572344975, 0.8740199572344975, 0.
 ↪8592490994133825]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
```

```
Epoch 00133: early stopping
[0.8713471133285816, 0.8713471133285816, 0.8713471133285816, 0.
 ↪8562902115842986]
[250, '0.2']
Epoch 00249: early stopping
[0.5712758374910906, 0.5712758374910906, 0.5712758374910906, 0.
 ↪5190729423874725]
[250, '0.2', 250, '0.2']
[0.8034568781183179, 0.8034568781183179, 0.8034568781183179, 0.
 ↪7804932079709793]
[250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00192: early stopping
[0.8526372059871703, 0.8526372059871703, 0.8526372059871702, 0.
 ↪8354667548029857]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00129: early stopping
[0.8483606557377049, 0.8483606557377049, 0.8483606557377049, 0.
 ↪8308142882391145]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00179: early stopping
[0.8635067712045617, 0.8635067712045617, 0.8635067712045617, 0.
 ↪8475984137698004]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00136: early stopping
[0.8390947968638631, 0.8390947968638631, 0.8390947968638631, 0.
 ↪8203757068695277]
[250, '0.3']
[0.5659301496792587, 0.5659301496792587, 0.5659301496792587, 0.
 ↪5130749928933578]
[250, '0.3', 250, '0.3']
Epoch 00266: early stopping
[0.7783321454027085, 0.7783321454027085, 0.7783321454027085, 0.
 ↪7522834553134814]
[250, '0.3', 250, '0.3', 250, '0.3']
[0.8531717747683535, 0.8531717747683535, 0.8531717747683535, 0.
 ↪8360999258915827]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00262: early stopping
[0.8465787598004276, 0.8465787598004276, 0.8465787598004277, 0.
 ↪8288494408217892]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00249: early stopping
[0.8406985032074127, 0.8406985032074127, 0.8406985032074127, 0.
 ↪8221270376207139]
```

```
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00210: early stopping
[0.8127227369921597, 0.8127227369921597, 0.8127227369921597, 0.
 ↪7909136938135756]
```

[64]:
```
joblib.dump(results_dropout, 'results_dropout')
```

[64]:
```
['results_dropout']
```

Dropout* usando *smote Repetimos el procedimiento con estos datos.

[25]:
```
results_dropout_smote = []
seed = 1
```

[26]:
```
size_config = [50, 100, 150, 200, 250]
dropout_rate = ["0.1", "0.2", "0.3"]

for size in size_config:
    for size_d in (dropout_rate):
        layer_config_dense = [[size], [size]*2, [size]*3, [size]*4,␣
 ↪[size]*5, [size]*6]
        layer_config_dropout = [[size_d], [size_d]*2, [size_d]*3,␣
 ↪[size_d]*4, [size_d]*5, [size_d]*6]
        for layers_dense, layers_dropout in zip(layer_config_dense,␣
 ↪layer_config_dropout):
            final_design = [None]*(len(layers_dense)+len(layers_dropout))
            final_design[::2] = layers_dense
            final_design[1::2] = layers_dropout
            np.random.seed(seed)
            tf.random.set_seed(seed)
            print(final_design)
            model = make_my_model_multi_dropout(final_design, 6, 18,␣
 ↪activation_='relu' )
            preds = compile_fit_multiclass(model, X_train_smote, X_test,␣
 ↪y_train_smote, 256, 300, verbose=0)
            metrics = compute_metrics_multiclass(np.argmax(preds, axis =␣
 ↪1), np.argmax(y_test, axis = 1))
            confusion = confusion_matrix(np.argmax(preds, axis = 1), np.
 ↪argmax(y_test, axis = 1))
            aux = { "layer config" : final_design,
                   #"Model": model,
                   "Predictions" : preds,
                   "Metrics" : metrics,
                   "Confusion" : confusion
            }
            print(metrics)
            results_dropout_smote.append(aux)
```

```
[50, '0.1']
[0.26300784034212404, 0.26300784034212404, 0.26300784034212404,
0.20085519448743994]
[50, '0.1', 50, '0.1']
Epoch 00265: early stopping
[0.2998930862437634, 0.2998930862437634, 0.2998930862437634,
0.23713346452369544]
[50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00199: early stopping
[0.3205630791161796, 0.3205630791161796, 0.3205630791161796, 0.
 ↪258903190675154]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00162: early stopping
[0.30648610121168923, 0.30648610121168923, 0.30648610121168923,
0.2430873107926549]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00142: early stopping
[0.3075552387740556, 0.3075552387740556, 0.3075552387740556,
0.24394819172656956]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00099: early stopping
[0.28367783321454026, 0.28367783321454026, 0.28367783321454026,
0.22122127360587496]
[50, '0.2']
Epoch 00271: early stopping
[0.2494654312188168, 0.2494654312188168, 0.2494654312188168, 0.
 ↪18757690658737980]
[50, '0.2', 50, '0.2']
Epoch 00175: early stopping
[0.28421240199572345, 0.28421240199572345, 0.28421240199572345,
0.22168654599358906]
[50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00163: early stopping
[0.27619387027797576, 0.27619387027797576, 0.27619387027797576,
0.2133303494383716]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00111: early stopping
[0.2729864575908767, 0.2729864575908767, 0.2729864575908767,
0.21067733393453925]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00222: early stopping
[0.2920527441197434, 0.2920527441197434, 0.2920527441197434, 0.
 ↪2270944345574314]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00159: early stopping
[0.2719173200285103, 0.2719173200285103, 0.2719173200285103,
0.20919106407082955]
[50, '0.3']
Epoch 00216: early stopping
[0.2459016393442623, 0.2459016393442623, 0.2459016393442623,
0.18529233154432556]
[50, '0.3', 50, '0.3']
Epoch 00154: early stopping
[0.2656806842480399, 0.2656806842480399, 0.2656806842480399,
0.20272738533056844]
[50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00121: early stopping
[0.26069137562366357, 0.26069137562366357, 0.26069137562366357,
0.19940871426469609]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00122: early stopping
[0.24376336421952957, 0.24376336421952957, 0.24376336421952957,
0.17933947264543937]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00177: early stopping
[0.2275481111903065, 0.2275481111903065, 0.2275481111903065, 0.
 ↪1614652947632501]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00160: early stopping
[0.21596578759800428, 0.21596578759800428, 0.21596578759800428,
0.1511632883484848]
[100, '0.1']
[0.2594440484675695, 0.2594440484675695, 0.2594440484675695, 0.
 ↪1955059345370388]
[100, '0.1', 100, '0.1']
[0.3447968638631504, 0.3447968638631504, 0.3447968638631504,
0.28340547320298026]
[100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00219: early stopping
[0.372416250890948, 0.372416250890948, 0.372416250890948, 0.
 ↪31370181177426715]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
```

```
Epoch 00185: early stopping
[0.35673556664290806, 0.35673556664290806, 0.35673556664290806,
0.2965402996283344]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00164: early stopping
[0.36689237348538845, 0.36689237348538845, 0.3668923734853885,
0.3079995227352187]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00100: early stopping
[0.35263720598717035, 0.35263720598717035, 0.35263720598717035,
0.29222509720780787]
[100, '0.2']
Epoch 00290: early stopping
[0.25659301496792586, 0.25659301496792586, 0.25659301496792586,
0.19235916152006827]
[100, '0.2', 100, '0.2']
Epoch 00251: early stopping
[0.3143264433357092, 0.3143264433357092, 0.3143264433357092,
0.25120042985546787]
[100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00129: early stopping
[0.328225231646471 8, 0.3282252316464718, 0.3282252316464718,
0.26708859834267673]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00105: early stopping
[0.315751960085531, 0.315751960085531, 0.315751960085531, 0.
 ↪2545872547346073]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00191: early stopping
[0.3321454027084818, 0.3321454027084818, 0.3321454027084818,
0.27051519260022494]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00215: early stopping
[0.3225231646471846, 0.3225231646471846, 0.3225231646471846, 0.
 ↪2600452708848978]
[100, '0.3']
[0.2553456878118318, 0.2553456878118318, 0.2553456878118318,
0.19117397326811747]
[100, '0.3', 100, '0.3']
Epoch 00267: early stopping
[0.3036350677120456, 0.3036350677120456, 0.3036350677120456,
0.24018452699497383]
[100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00131: early stopping
```

```
[0.32430506058446185, 0.32430506058446185, 0.32430506058446185,
0.2625623102105059]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00178: early stopping
[0.3014967925873129, 0.3014967925873129, 0.3014967925873129,
0.23783108719859625]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00093: early stopping
[0.2965074839629366, 0.2965074839629366, 0.2965074839629366,
0.23485587148830267]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00111: early stopping
[0.33018531717747684, 0.33018531717747684, 0.33018531717747684,
0.26997768943588174]
[150, '0.1']
[0.27049180327868855, 0.27049180327868855, 0.27049180327868855,
0.2070689926269983]
[150, '0.1', 150, '0.1']
Epoch 00236: early stopping
[0.36617961511047753, 0.36617961511047753, 0.36617961511047753,
0.3065969957363909]
[150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00179: early stopping
[0.3544191019244476, 0.3544191019244476, 0.3544191019244476,
0.29432310566920317]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00151: early stopping
[0.38025659301496795, 0.38025659301496795, 0.38025659301496795,
0.3217359080760902]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00199: early stopping
[0.38934426229508196, 0.38934426229508196, 0.38934426229508196,
0.3320630520189505]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00179: early stopping
[0.39468995010691377, 0.39468995010691377, 0.39468995010691377,
0.33666606146261224]
[150, '0.2']
[0.2557020669992872, 0.2557020669992872, 0.2557020669992872,
0.19120997021688535]
[150, '0.2', 150, '0.2']
Epoch 00201: early stopping
[0.3499643620812545, 0.3499643620812545, 0.3499643620812545,
0.28962387662704503]
[150, '0.2', 150, '0.2', 150, '0.2']
```

```
Epoch 00227: early stopping
[0.3643977191732003, 0.3643977191732003, 0.3643977191732003, 0.
 ↪3047718177375881]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00263: early stopping
[0.3832858161083393, 0.3832858161083393, 0.3832858161083393, 0.
 ↪3251691504162134]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00151: early stopping
[0.3891660727013542, 0.3891660727013542, 0.3891660727013542, 0.
 ↪3318250780448826]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00069: early stopping
[0.35548823948681396, 0.35548823948681396, 0.3554882394868139,
0.29658654328800893]
[150, '0.3']
Epoch 00300: early stopping
[0.2540983606557377, 0.2540983606557377, 0.2540983606557377,
0.18919799124540182]
[150, '0.3', 150, '0.3']
Epoch 00230: early stopping
[0.3396293656450463, 0.3396293656450463, 0.3396293656450463, 0.
 ↪2786682676628811]
[150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00197: early stopping
[0.3677833214540271, 0.3677833214540271, 0.36778332145402703,
0.3088770225577919]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00140: early stopping
[0.3462223806129722, 0.3462223806129722, 0.3462223806129722,
0.28653161153670004]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00129: early stopping
[0.3579828937990021, 0.3579828937990021, 0.35798289379900206,
0.29920706564380195]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00137: early stopping
[0.3549536707056308, 0.3549536707056308, 0.35495367070563083,
0.2956493499086582]
[200, '0.1']
[0.2758374910905203, 0.2758374910905203, 0.2758374910905203,
0.21182638080300087]
[200, '0.1', 200, '0.1']
Epoch 00203: early stopping
```

```
[0.36065573770491804, 0.36065573770491804, 0.3606557377049181,
0.30122411585270636]
[200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00187: early stopping
[0.37829650748396293, 0.37829650748396293, 0.37829650748396293,
0.319156284144949]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00136: early stopping
[0.39522451888809695, 0.39522451888809695, 0.395224518888097,
0.3373918663980583]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00082: early stopping
[0.40021382751247325, 0.40021382751247325, 0.40021382751247325,
0.34222347641272977]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00109: early stopping
[0.38649322879543835, 0.38649322879543835, 0.38649322879543835,
0.32731143396239426]
[200, '0.2']
[0.268888096935139, 0.268888096935139, 0.268888096935139, 0.
 ↪20499839988310808]
[200, '0.2', 200, '0.2']
Epoch 00167: early stopping
[0.35655737704918034, 0.35655737704918034, 0.3565573770491803,
0.2967274796130971]
[200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00232: early stopping
[0.40324305060584464, 0.40324305060584464, 0.40324305060584464,
0.3467882041439525]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00161: early stopping
[0.3955808980755524, 0.3955808980755524, 0.3955808980755524, 0.
 ↪3376063898548457]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00241: early stopping
[0.3907697790449038, 0.3907697790449038, 0.3907697790449038, 0.
 ↪3324561672984019]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00141: early stopping
[0.3679615110477548, 0.3679615110477548, 0.36796151104775476,
0.3083622483336965]
[200, '0.3']
Epoch 00153: early stopping
[0.26924447612259444, 0.26924447612259444, 0.26924447612259444,
0.2058282571547384]
[200, '0.3', 200, '0.3']
Epoch 00175: early stopping
[0.3556664290805417, 0.3556664290805417, 0.3556664290805417, 0.
 ↪2950894640652668]
[200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00232: early stopping
[0.39326443335709194, 0.39326443335709194, 0.39326443335709194,
0.33580720228153926]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00232: early stopping
[0.37954383464005703, 0.37954383464005703, 0.37954383464005703,
0.3216607699137475]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00140: early stopping
[0.3638631503920171, 0.3638631503920171, 0.3638631503920171,
0.30498056948130214]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00129: early stopping
[0.36689237348538845, 0.36689237348538845, 0.3668923734853885,
0.3069702792714195]
[250, '0.1']
Epoch 00285: early stopping
[0.2753029223093371, 0.2753029223093371, 0.2753029223093371,
0.21098176024808302]
[250, '0.1', 250, '0.1']
Epoch 00271: early stopping
[0.36350677120456165, 0.36350677120456165, 0.36350677120456165,
0.3031787683148758]
[250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00117: early stopping
[0.39861012116892375, 0.39861012116892375, 0.39861012116892375,
0.3411751823555649]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00116: early stopping
[0.3923734853884533, 0.3923734853884533, 0.3923734853884533, 0.
 ↪3343374374058219]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00126: early stopping
[0.3907697790449038, 0.3907697790449038, 0.3907697790449038,
0.33151015653713056]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00116: early stopping
[0.4000356379187455, 0.4000356379187455, 0.4000356379187456,
0.34147287960435413]
[250, '0.2']
Epoch 00244: early stopping
[0.2662152530292231, 0.2662152530292231, 0.2662152530292231,
0.20255805666422488]
[250, '0.2', 250, '0.2']
Epoch 00252: early stopping
[0.3699215965787598, 0.3699215965787598, 0.3699215965787598, 0.
 ↪312080858448489]
[250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00234: early stopping
[0.3995010691375624, 0.3995010691375624, 0.3995010691375624, 0.
 ↪3421950134241337]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00159: early stopping
[0.3923734853884533, 0.3923734853884533, 0.3923734853884533, 0.
 ↪3339574861445036]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00117: early stopping
[0.3914825374198147, 0.3914825374198147, 0.3914825374198147,
0.33271842857445766]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00132: early stopping
[0.3995010691375624, 0.3995010691375624, 0.3995010691375624,
0.34119409276086266]
[250, '0.3']
Epoch 00285: early stopping
[0.2690662865288667, 0.2690662865288667, 0.2690662865288667, 0.
 ↪2052302756253448]
[250, '0.3', 250, '0.3']
Epoch 00242: early stopping
[0.3602993585174626, 0.3602993585174626, 0.3602993585174626, 0.
 ↪3011976641288431]
[250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00216: early stopping
[0.39611546685673554, 0.39611546685673554, 0.39611546685673554,
0.3388907927897745]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00160: early stopping
[0.3923734853884533, 0.3923734853884533, 0.3923734853884533,
0.33487742605489634]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00174: early stopping
```

```
[0.3766928011404134, 0.3766928011404134, 0.3766928011404134, 0.
 ↪317371421227289]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00160: early stopping
[0.387384176764077, 0.387384176764077, 0.387384176764077, 0.
 ↪3296291053670306]
```

[27]:
```
import joblib
joblib.dump(results_dropout_smote, 'results_dropout_smote')
```

[27]:
```
['results_dropout_smote']
```

3.2 Datos codificados con *one_hot* con *smote* y *dropout*

Repetimos los mismos pasos que en el apartado anterior la diferencia de que codificamos los datos de las variables predictoras usando *dummy variables* o codificación *one-hot*. Así, compararemos el rendimiento de los modelos.

[6]:
```
import numpy as np
import pandas as pd
data = pd.read_csv("krkopt.data", header=None)
data.columns = ["wkc", "wkr", "wrc", "wrr", "bkc", "bkr", "opt rank" ]
X = data.iloc[:, 0:6]
y = data['opt rank']
X["wkc"]=X["wkc"].astype('category')
X["wrc"]=X["wrc"].astype('category')
X["bkc"]=X["bkc"].astype('category')
X["wkc"]=X["wkc"].cat.codes
X["wrc"]=X["wrc"].cat.codes
X["bkc"]=X["bkc"].cat.codes
y = y.astype('category')
y = y.cat.codes

from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.model_selection import train_test_split

cat_cols = list(X.columns)
```

Codificamos usando *get_dummies* de pandas. Veremos que ahora tenemos 40 variables predictoras.

[7]:
```
X = pd.get_dummies(X,columns=cat_cols)
X
```

```
[7]:        wkc_0  wkc_1  wkc_2  wkc_3  wkr_1  wkr_2  wkr_3  wkr_4  wrc_0 ␣
    ↪wrc_1  \
    0          1      0      0      0      1      0      0      0      0    ␣
    ↪1
    1          1      0      0      0      1      0      0      0      0    ␣
    ↪0
    2          1      0      0      0      1      0      0      0      0    ␣
    ↪0
    3          1      0      0      0      1      0      0      0      0    ␣
    ↪0
    4          1      0      0      0      1      0      0      0      0    ␣
    ↪0
    …      …    …    …    …    …    …    …    …    …    …
    28051      0      1      0      0      1      0      0      0      0    ␣
    ↪0
    28052      0      1      0      0      1      0      0      0      0    ␣
    ↪0
    28053      0      1      0      0      1      0      0      0      0    ␣
    ↪0
    28054      0      1      0      0      1      0      0      0      0    ␣
    ↪0
    28055      0      1      0      0      1      0      0      0      0    ␣
    ↪0

          …  bkc_6  bkc_7  bkr_1  bkr_2  bkr_3  bkr_4  bkr_5  bkr_6  bkr_7  \
    0     …      0      0      0      1      0      0      0      0      0
    1     …      0      0      0      1      0      0      0      0      0
    2     …      0      0      1      0      0      0      0      0      0
    3     …      0      0      0      1      0      0      0      0      0
    4     …      0      0      1      0      0      0      0      0      0
    …    …    …    …    …    …    …    …    …    …    …
    28051 …      0      0      0      0      0      0      1      0      0
    28052 …      0      0      0      0      0      0      0      1      0
    28053 …      0      0      0      0      0      0      0      0      1
    28054 …      0      0      0      0      0      0      1      0      0
    28055 …      1      0      0      0      0      0      1      0      0

          bkr_8
    0         0
    1         0
    2         0
    3         0
    4         0
```

```
…        …
28051      0
28052      0
28053      0
28054      0
28055      0

[28056 rows x 40 columns]
```

[15]:
```
from tensorflow import keras
import tensorflow as tf
from tensorflow.keras.utils import to_categorical

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y, test_size=0.2,
                                                    random_state = 1)

from imblearn.over_sampling import SMOTE
oversample = SMOTE()


X_smote, y_smote = oversample.fit_resample(X, y)

X_train_smote, X_test_smote, y_train_smote, y_test_smote =␣
 ↪train_test_split(X_smote,
                                                    y_smote, test_size=0.2,
                                                    random_state = 1)




y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)
y_train_smote = to_categorical(y_train_smote)
y_test_smote = to_categorical(y_test_smote)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

scaler = StandardScaler().fit(X_train_smote)
X_train_smote = scaler.transform(X_train_smote)
X_test_smote = scaler.transform(X_test_smote)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


checkpoint_filepath = '/tmp/checkpoint'

from sklearn.metrics import confusion_matrix, precision_score, \
f1_score, cohen_kappa_score, recall_score
```

3.2.1 Pruebas

Todos los casos son los mismos que en los datos con la anterior codificación. Guardaremos en archivos todos los objetos de cada conjunto de experimentos.

Datos *one_hot*

[48]:
```
results = []
seed = 1
```

[49]:
```
size_config = [50, 100, 150, 200, 250]
for size in size_config:
    layer_config = [[size], [size]*2, [size]*3, [size]*4, [size]*5,␣
 ↪[size]*6]
    for layers in layer_config:
        np.random.seed(seed)
        tf.random.set_seed(seed)
        print(layers)
        model = make_my_model_multi(layers, 40, 18, activation_='relu' )
        preds = compile_fit_multiclass(model, X_train, X_test, y_train,␣
 ↪256, 300, verbose=0)
        metrics = compute_metrics_multiclass(np.argmax(preds, axis = 1),␣
 ↪np.argmax(y_test, axis = 1))
        confusion = confusion_matrix(np.argmax(preds, axis = 1), np.
 ↪argmax(y_test, axis = 1))
        aux = { "layer config" : layers,
               #"Model": model,
               "Predictions" : preds,
               "Metrics" : metrics,
               "Confusion" : confusion

        }
        print(metrics)
        results.append(aux)
```

```
[50]
[0.7132929436920884, 0.7132929436920884, 0.7132929436920883, 0.
↪6795940399725571]
[50, 50]
Epoch 00118: early stopping
[0.7359230220955096, 0.7359230220955096, 0.7359230220955095, 0.
↪7051213677686434]
[50, 50, 50]
Epoch 00075: early stopping
[0.7464362081254454, 0.7464362081254454, 0.7464362081254454, 0.
↪7169690264820239]
[50, 50, 50, 50]
Epoch 00059: early stopping
[0.7293300071275838, 0.7293300071275838, 0.7293300071275838, 0.
↪6976471559645523]
[50, 50, 50, 50, 50]
Epoch 00053: early stopping
[0.7323592302209551, 0.7323592302209551, 0.7323592302209551, 0.
↪7011602714040706]
[50, 50, 50, 50, 50, 50]
Epoch 00054: early stopping
[0.7245188880969351, 0.7245188880969351, 0.7245188880969351, 0.
↪6923109986216298]
[100]
Epoch 00176: early stopping
[0.7405559515324305, 0.7405559515324305, 0.7405559515324305, 0.
↪7101055066873646]
[100, 100]
Epoch 00053: early stopping
[0.7624732715609408, 0.7624732715609408, 0.7624732715609408, 0.
↪7348246606533331]
[100, 100, 100]
Epoch 00042: early stopping
[0.7859942979330007, 0.7859942979330007, 0.7859942979330007, 0.
↪761075013060205]
[100, 100, 100, 100]
Epoch 00039: early stopping
[0.7845687811831789, 0.7845687811831789, 0.7845687811831789, 0.
↪7594270506445013]
[100, 100, 100, 100, 100]
Epoch 00034: early stopping
```

```
[0.7802922309337135, 0.7802922309337135, 0.7802922309337134, 0.
 ↪7544084918576515]
[100, 100, 100, 100, 100, 100]
Epoch 00029: early stopping
[0.7590876692801141, 0.7590876692801141, 0.7590876692801141, 0.
 ↪7307269833855377]
[150]
Epoch 00150: early stopping
[0.7587312900926586, 0.7587312900926586, 0.7587312900926585, 0.
 ↪7304863274796372]
[150, 150]
Epoch 00059: early stopping
[0.8080898075552387, 0.8080898075552387, 0.8080898075552387, 0.
 ↪7856176438592615]
[150, 150, 150]
Epoch 00038: early stopping
[0.8071988595866001, 0.8071988595866001, 0.8071988595866, 0.
 ↪7847174514580395]
[150, 150, 150, 150]
Epoch 00030: early stopping
[0.7995367070563079, 0.7995367070563079, 0.799536707056308, 0.
 ↪7760993871183968]
[150, 150, 150, 150, 150]
Epoch 00024: early stopping
[0.785816108339273, 0.785816108339273, 0.785816108339273, 0.760816977877812]
[150, 150, 150, 150, 150, 150]
Epoch 00026: early stopping
[0.8013186029935851, 0.8013186029935851, 0.8013186029935851, 0.
 ↪7778395069749614]
[200]
Epoch 00108: early stopping
[0.7633642195295794, 0.7633642195295794, 0.7633642195295794, 0.
 ↪7355328449643446]
[200, 200]
Epoch 00041: early stopping
[0.8031004989308624, 0.8031004989308624, 0.8031004989308624, 0.
 ↪7800639075355223]
[200, 200, 200]
Epoch 00032: early stopping
[0.8096935138987883, 0.8096935138987883, 0.8096935138987883, 0.
 ↪787236216263784]
[200, 200, 200, 200]
Epoch 00031: early stopping
```

```
[0.8308980755523877, 0.8308980755523877, 0.8308980755523878, 0.
↪8111400326514505]
[200, 200, 200, 200, 200]
Epoch 00028: early stopping
[0.8145046329294369, 0.8145046329294369, 0.8145046329294369, 0.
↪7932395201038133]
[200, 200, 200, 200, 200, 200]
Epoch 00020: early stopping
[0.7856379187455452, 0.7856379187455452, 0.7856379187455452, 0.
↪7609006763147126]
[250]
Epoch 00093: early stopping
[0.7756593014967926, 0.7756593014967926, 0.7756593014967926, 0.
↪74921524924313]
[250, 250]
Epoch 00039: early stopping
[0.8241268709907341, 0.8241268709907341, 0.8241268709907341, 0.
↪8035364436115411]
[250, 250, 250]
Epoch 00030: early stopping
[0.8301853171774768, 0.8301853171774768, 0.8301853171774768, 0.
↪8104398918945435]
[250, 250, 250, 250]
Epoch 00023: early stopping
[0.8171774768353528, 0.8171774768353528, 0.8171774768353528, 0.
↪7955487752872001]
[250, 250, 250, 250, 250]
Epoch 00022: early stopping
[0.8038132573057734, 0.8038132573057734, 0.8038132573057732, 0.
↪7810034973873443]
[250, 250, 250, 250, 250, 250]
Epoch 00022: early stopping
[0.8084461867426942, 0.8084461867426942, 0.8084461867426942, 0.
↪7861858456085945]
```

[50]:
```python
import joblib

joblib.dump(results, 'results_1_onehot_joblib')
```

[50]:
```
['results_1_onehot_joblib']
```

3.2.2 Datos *one_hot* con *SMOTE*

```
[51]: results_smote = []
      seed = 1
```

```
[52]: size_config = [50, 100, 150, 200, 250]
      for size in size_config:
          layer_config = [[size], [size]*2, [size]*3, [size]*4, [size]*5,␣
       ↪[size]*6]
          for layers in layer_config:
              np.random.seed(seed)
              tf.random.set_seed(seed)
              print(layers)
              model = make_my_model_multi(layers, 40, 18, activation_='relu' )
              preds = compile_fit_multiclass(model, X_train_smote, X_test,␣
       ↪y_train_smote, 256, 300, verbose=0)
              metrics = compute_metrics_multiclass(np.argmax(preds, axis = 1),␣
       ↪np.argmax(y_test, axis = 1))
              confusion = confusion_matrix(np.argmax(preds, axis = 1), np.
       ↪argmax(y_test, axis = 1))
              aux = { "layer config" : layers,
                      #"Model": model,
                      "Predictions" : preds,
                      "Metrics" : metrics,
                      "Confusion" : confusion

              }
              print(metrics)
              results_smote.append(aux)
```

```
[50]
[0.09853884533143265, 0.09853884533143265, 0.09853884533143265,
0.05990480607973814]
[50, 50]
Epoch 00270: early stopping
[0.06931575196008553, 0.06931575196008553, 0.06931575196008553,
0.03700690402698103 6]
[50, 50, 50]
Epoch 00155: early stopping
[0.06967213114754098, 0.06967213114754098, 0.06967213114754098,
0.03441175321042145]
[50, 50, 50, 50]
Epoch 00153: early stopping
[0.07448325017818959, 0.07448325017818959, 0.07448325017818959,
0.04118489587487906]
[50, 50, 50, 50, 50]
Epoch 00120: early stopping
[0.06789023521026372, 0.06789023521026372, 0.06789023521026372,
0.028956419894087593]
[50, 50, 50, 50, 50, 50]
Epoch 00200: early stopping
[0.05434782608695652, 0.05434782608695652, 0.05434782608695652,
0.018471319233337335]
[100]
[0.11101211689237349, 0.11101211689237349, 0.11101211689237349,
0.07134215115338971]
[100, 100]
Epoch 00285: early stopping
[0.1097647897362794, 0.1097647897362794, 0.1097647897362794,
0.07580671664360639]
[100, 100, 100]
Epoch 00226: early stopping
[0.11279401282965075, 0.11279401282965075, 0.11279401282965075,
0.0702367187003482]
[100, 100, 100, 100]
Epoch 00157: early stopping
[0.09764789736279401, 0.09764789736279401, 0.09764789736279401,
0.058277308802495265]
[100, 100, 100, 100, 100]
Epoch 00162: early stopping
[0.07947255880256593, 0.07947255880256593, 0.07947255880256593,
0.04168589497606212]
[100, 100, 100, 100, 100, 100]
Epoch 00154: early stopping
[0.07662152530292231, 0.07662152530292231, 0.07662152530292231,
0.041859363311582354]
[150]
[0.11119030648610122, 0.11119030648610122, 0.11119030648610122,
0.07057217306767738]
[150, 150]
Epoch 00192: early stopping
[0.12918745545260157, 0.12918745545260157, 0.12918745545260157,
0.09148213445074216]
[150, 150, 150]
Epoch 00166: early stopping
[0.13809693513898788, 0.13809693513898788, 0.13809693513898788,
0.0914795402998081]
[150, 150, 150, 150]
Epoch 00140: early stopping
```

```
[0.10816108339272987, 0.10816108339272987, 0.10816108339272985,
0.06738482220517195]
[150, 150, 150, 150, 150]
Epoch 00109: early stopping
[0.06699928724162509, 0.06699928724162509, 0.06699928724162509,
0.03164283938023682]
[150, 150, 150, 150, 150, 150]
Epoch 00093: early stopping
[0.06254454739843193, 0.06254454739843193, 0.06254454739843193,
0.02622167248184981]
[200]
Epoch 00192: early stopping
[0.09461867426942266, 0.09461867426942266, 0.09461867426942266,
0.0597149180355222]
[200, 200]
Epoch 00254: early stopping
[0.1876336421952958, 0.1876336421952958, 0.1876336421952958,
0.14429603051691353]
[200, 200, 200]
Epoch 00120: early stopping
[0.15217391304347827, 0.15217391304347827, 0.15217391304347827,
0.10779708923931663]
[200, 200, 200, 200]
Epoch 00129: early stopping
[0.1003207412687099, 0.1003207412687099, 0.1003207412687099,
0.06150141024915745]
[200, 200, 200, 200, 200]
Epoch 00081: early stopping
[0.08357091945830364, 0.08357091945830364, 0.08357091945830364,
0.047671217979086355]
[200, 200, 200, 200, 200, 200]
Epoch 00082: early stopping
[0.0792943692088382, 0.0792943692088382, 0.0792943692088382, 0.
 ↪0425269634720844]
[250]
Epoch 00254: early stopping
[0.10655737704918032, 0.10655737704918032, 0.10655737704918032,
0.07031624765828526]
[250, 250]
Epoch 00174: early stopping
[0.14593727726300784, 0.14593727726300784, 0.14593727726300784,
0.10398057068418642]
[250, 250, 250]
Epoch 00106: early stopping
```

```
[0.17783321454027085, 0.17783321454027085, 0.17783321454027085,
0.13176963959600485]
[250, 250, 250, 250]
Epoch 00087: early stopping
[0.1172487526728439, 0.1172487526728439, 0.1172487526728439,
0.07717883510268653]
[250, 250, 250, 250, 250]
Epoch 00081: early stopping
[0.08802565930149679, 0.08802565930149679, 0.08802565930149679,
0.05372309817465015]
[250, 250, 250, 250, 250, 250]
Epoch 00058: early stopping
[0.08214540270848182, 0.08214540270848182, 0.08214540270848182,
0.047641839366908134]
```

[53]:
```
joblib.dump(results_smote, 'results_smote_onehot_joblib')
```

[53]:
```
['results_smote_onehot_joblib']
```

3.2.3 Datos *one_hot* con *dropout*

[54]:
```
results_dropout = []
seed = 1
```

[55]:
```
size_config = [50, 100, 150, 200, 250]
dropout_rate = ["0.1", "0.2", "0.3"]

for size in size_config:
    for size_d in (dropout_rate):
        layer_config_dense = [[size], [size]*2, [size]*3, [size]*4,␣
 ↪[size]*5, [size]*6]
        layer_config_dropout = [[size_d], [size_d]*2, [size_d]*3,␣
 ↪[size_d]*4, [size_d]*5, [size_d]*6]
        for layers_dense, layers_dropout in zip(layer_config_dense,␣
 ↪layer_config_dropout):
            final_design = [None]*(len(layers_dense)+len(layers_dropout))
            final_design[::2] = layers_dense
            final_design[1::2] = layers_dropout
            np.random.seed(seed)
            tf.random.set_seed(seed)
            print(final_design)
            model = make_my_model_multi_dropout(final_design, 40, 18,␣
 ↪activation_='relu' )
```

```
            preds = compile_fit_multiclass(model, X_train, X_test,␣
↪y_train, 256, 300, verbose=0)
            metrics = compute_metrics_multiclass(np.argmax(preds, axis =␣
↪1), np.argmax(y_test, axis = 1))
            confusion = confusion_matrix(np.argmax(preds, axis = 1), np.
↪argmax(y_test, axis = 1))
            aux = { "layer config" : final_design,
                   #"Model": model,
                   "Predictions" : preds,
                   "Metrics" : metrics,
                   "Confusion" : confusion

            }
            print(metrics)
            results_dropout.append(aux)
```

```
[50, '0.1']
[0.6911974340698503, 0.6911974340698503, 0.6911974340698503, 0.
↪6544424729231852]
[50, '0.1', 50, '0.1']
Epoch 00249: early stopping
[0.7549893086243763, 0.7549893086243763, 0.7549893086243763, 0.
↪7260805176671765]
[50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00244: early stopping
[0.7685317177476836, 0.7685317177476836, 0.7685317177476836, 0.
↪7412666342935919]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00204: early stopping
[0.7628296507483963, 0.7628296507483963, 0.7628296507483963, 0.
↪7349489199206065]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00204: early stopping
[0.755167498218104, 0.755167498218104, 0.755167498218104, 0.726566737550184]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00138: early stopping
[0.7446543121881682, 0.7446543121881682, 0.7446543121881682, 0.
↪7145851936293055]
[50, '0.2']
[0.6797933000712758, 0.6797933000712758, 0.6797933000712758, 0.
↪6415147506179972]
[50, '0.2', 50, '0.2']
Epoch 00194: early stopping
```

```
[0.7211332858161084, 0.7211332858161084, 0.7211332858161085, 0.
 ↪6880538576802332]
[50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00202: early stopping
[0.7189950106913756, 0.7189950106913756, 0.7189950106913756, 0.
 ↪6857954888177956]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00177: early stopping
[0.7049180327868853, 0.7049180327868853, 0.7049180327868853, 0.
 ↪6696354479193032]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00184: early stopping
[0.7040270848182466, 0.7040270848182466, 0.7040270848182466, 0.
 ↪6688250607918782]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00225: early stopping
[0.6983250178189594, 0.6983250178189594, 0.6983250178189594, 0.
 ↪6625351015506182]
[50, '0.3']
Epoch 00290: early stopping
[0.6584105488239487, 0.6584105488239487, 0.6584105488239487, 0.
 ↪6172402972361986]
[50, '0.3', 50, '0.3']
Epoch 00182: early stopping
[0.6837134711332858, 0.6837134711332858, 0.6837134711332858, 0.
 ↪6456665282835925]
[50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00152: early stopping
[0.66928011404134, 0.66928011404134, 0.66928011404134, 0.6293258262824244]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00182: early stopping
[0.655559515324305, 0.655559515324305, 0.655559515324305, 0.
 ↪6142799530678658]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00192: early stopping
[0.6407697790449037, 0.6407697790449037, 0.6407697790449037, 0.
 ↪5974540868981456]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00216: early stopping
[0.6076265146115467, 0.6076265146115467, 0.6076265146115467, 0.
 ↪5617307756281984]
[100, '0.1']
Epoch 00267: early stopping
```

```
[0.7537419814682823, 0.7537419814682823, 0.7537419814682823, 0.
 ↪7247488644953375]
[100, '0.1', 100, '0.1']
Epoch 00193: early stopping
[0.8221667854597291, 0.8221667854597291, 0.8221667854597291, 0.
 ↪8014009505335618]
[100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00182: early stopping
[0.8504989308624377, 0.8504989308624377, 0.8504989308624377, 0.
 ↪8330632580919843]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00146: early stopping
[0.8480042765502495, 0.8480042765502495, 0.8480042765502495, 0.
 ↪8302515913483358]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00148: early stopping
[0.8478260869565217, 0.8478260869565217, 0.8478260869565218, 0.
 ↪8300096411794345]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00150: early stopping
[0.8460441910192444, 0.8460441910192444, 0.8460441910192444, 0.
 ↪8281644104059738]
[100, '0.2']
[0.7535637918745546, 0.7535637918745546, 0.7535637918745546, 0.
 ↪7243112074616954]
[100, '0.2', 100, '0.2']
Epoch 00200: early stopping
[0.8059515324305061, 0.8059515324305061, 0.8059515324305061, 0.
 ↪7831509826329692]
[100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00258: early stopping
[0.8403421240199572, 0.8403421240199572, 0.8403421240199572, 0.
 ↪8216186407532786]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00189: early stopping
[0.8145046329294369, 0.8145046329294369, 0.8145046329294369, 0.
 ↪7927907643878691]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00129: early stopping
[0.792943692088382, 0.792943692088382, 0.792943692088382, 0.
 ↪7687186237795841]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00236: early stopping
```

```
[0.8111190306486101, 0.8111190306486101, 0.8111190306486101, 0.
 ↪7890115939135013]
[100, '0.3']
[0.7442979330007128, 0.7442979330007128, 0.7442979330007128, 0.
 ↪7138441190567824]
[100, '0.3', 100, '0.3']
Epoch 00208: early stopping
[0.7888453314326443, 0.7888453314326443, 0.7888453314326443, 0.
 ↪7640197609347319]
[100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00221: early stopping
[0.8061297220242338, 0.8061297220242338, 0.8061297220242338, 0.
 ↪7833711703582897]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00153: early stopping
[0.7676407697790449, 0.7676407697790449, 0.7676407697790449, 0.
 ↪7403424377519607]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00156: early stopping
[0.7667498218104063, 0.7667498218104063, 0.7667498218104063, 0.
 ↪7393739781195857]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00176: early stopping
[0.7414468995010691, 0.7414468995010691, 0.7414468995010691, 0.
 ↪7111252105230017]
[150, '0.1']
[0.7820741268709908, 0.7820741268709908, 0.7820741268709906, 0.
 ↪7564243977333958]
[150, '0.1', 150, '0.1']
Epoch 00129: early stopping
[0.855488239486814, 0.855488239486814, 0.855488239486814, 0.
 ↪8385669050046001]
[150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00079: early stopping
[0.8590520313613685, 0.8590520313613685, 0.8590520313613685, 0.
 ↪8426740003139367]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00109: early stopping
[0.8781183178902352, 0.8781183178902352, 0.8781183178902352, 0.
 ↪8639675844777932]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00094: early stopping
[0.8700997861724875, 0.8700997861724875, 0.8700997861724875, 0.
 ↪8549745938221411]
```

```
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00093: early stopping
[0.8695652173913043, 0.8695652173913043, 0.8695652173913043, 0.
 ↪8543679160066875]
[150, '0.2']
Epoch 00248: early stopping
[0.7751247327156094, 0.7751247327156094, 0.7751247327156094, 0.
 ↪7486044176314802]
[150, '0.2', 150, '0.2']
Epoch 00220: early stopping
[0.860655737704918, 0.860655737704918, 0.860655737704918, 0.844376829162256]
[150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00231: early stopping
[0.8761582323592302, 0.8761582323592302, 0.8761582323592302, 0.
 ↪8616730939921385]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00199: early stopping
[0.8679615110477548, 0.8679615110477548, 0.8679615110477547, 0.
 ↪852580315610961]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00136: early stopping
[0.8601211689237348, 0.8601211689237348, 0.8601211689237348, 0.
 ↪8437704300336455]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00139: early stopping
[0.8535281539558089, 0.8535281539558089, 0.8535281539558089, 0.
 ↪8364105315970465]
[150, '0.3']
Epoch 00253: early stopping
[0.7713827512473271, 0.7713827512473271, 0.7713827512473271, 0.
 ↪744263007539312]
[150, '0.3', 150, '0.3']
Epoch 00214: early stopping
[0.8458660014255167, 0.8458660014255167, 0.8458660014255168, 0.
 ↪8278767855942375]
[150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00220: early stopping
[0.840520313613685, 0.840520313613685, 0.840520313613685, 0.
 ↪8219800613018564]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00173: early stopping
[0.8300071275837491, 0.8300071275837491, 0.8300071275837491, 0.
 ↪8101729751256426]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
```

```
Epoch 00171: early stopping
[0.8218104062722738, 0.8218104062722738, 0.8218104062722738, 0.
 ↪8010098078549712]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00130: early stopping
[0.7952601568068425, 0.7952601568068425, 0.7952601568068425, 0.
 ↪7712983060607658]
[200, '0.1']
Epoch 00228: early stopping
[0.7961511047754811, 0.7961511047754811, 0.796151104775481, 0.
 ↪7720837157452116]
[200, '0.1', 200, '0.1']
Epoch 00097: early stopping
[0.8569137562366358, 0.8569137562366358, 0.8569137562366358, 0.
 ↪8402549944374698]
[200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00098: early stopping
[0.8884533143264434, 0.8884533143264434, 0.8884533143264434, 0.
 ↪8754311944881142]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00065: early stopping
[0.8700997861724875, 0.8700997861724875, 0.8700997861724875, 0.
 ↪854908986256218]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00077: early stopping
[0.8845331432644333, 0.8845331432644333, 0.8845331432644333, 0.
 ↪8710306582427136]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00068: early stopping
[0.8734853884533144, 0.8734853884533144, 0.8734853884533144, 0.
 ↪8588150087272403]
[200, '0.2']
Epoch 00292: early stopping
[0.7993585174625801, 0.7993585174625801, 0.7993585174625801, 0.
 ↪7756947986927558]
[200, '0.2', 200, '0.2']
Epoch 00173: early stopping
[0.8761582323592302, 0.8761582323592302, 0.8761582323592302, 0.
 ↪8616832826121679]
[200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00178: early stopping
[0.8929080541696365, 0.8929080541696365, 0.8929080541696365, 0.
 ↪8804258424474269]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
```

```
Epoch 00122: early stopping
[0.8889878831076266, 0.8889878831076266, 0.8889878831076266, 0.
 ↪876025663923951]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00113: early stopping
[0.8752672843905915, 0.8752672843905915, 0.8752672843905915, 0.
 ↪86073847814212]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00162: early stopping
[0.8863150392017106, 0.8863150392017106, 0.8863150392017106, 0.
 ↪8731436557602646]
[200, '0.3']
[0.7986457590876693, 0.7986457590876693, 0.7986457590876693, 0.
 ↪7748684094073761]
[200, '0.3', 200, '0.3']
Epoch 00230: early stopping
[0.8674269422665716, 0.8674269422665716, 0.8674269422665717, 0.
 ↪8519922476261953]
[200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00214: early stopping
[0.8788310762651461, 0.8788310762651461, 0.8788310762651462, 0.
 ↪8646983800933996]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00172: early stopping
[0.8717034925160371, 0.8717034925160371, 0.8717034925160371, 0.
 ↪8566902616204991]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00223: early stopping
[0.8738417676407698, 0.8738417676407698, 0.8738417676407699, 0.
 ↪8591722739548441]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00165: early stopping
[0.8581610833927299, 0.8581610833927299, 0.8581610833927299, 0.
 ↪8415628528427186]
[250, '0.1']
Epoch 00175: early stopping
[0.7995367070563079, 0.7995367070563079, 0.799536707056308, 0.
 ↪7759209362036418]
[250, '0.1', 250, '0.1']
Epoch 00089: early stopping
[0.8727726300784034, 0.8727726300784034, 0.8727726300784034, 0.
 ↪8579218239707193]
[250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00071: early stopping
```

```
[0.8864932287954383, 0.8864932287954383, 0.8864932287954383, 0.
 ↪873233938730633]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00054: early stopping
[0.8807911617961511, 0.8807911617961511, 0.8807911617961511, 0.
 ↪866858669993142]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00069: early stopping
[0.8875623663578047, 0.8875623663578047, 0.8875623663578046, 0.
 ↪8744596060520364]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00087: early stopping
[0.894511760513186, 0.894511760513186, 0.894511760513186, 0.
 ↪8821913607432853]
[250, '0.2']
Epoch 00177: early stopping
[0.7922309337134711, 0.7922309337134711, 0.7922309337134711, 0.
 ↪7676712411254928]
[250, '0.2', 250, '0.2']
Epoch 00183: early stopping
[0.8918389166072701, 0.8918389166072701, 0.8918389166072701, 0.
 ↪8792620364600945]
[250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00093: early stopping
[0.8877405559515325, 0.8877405559515325, 0.8877405559515325, 0.
 ↪8746325738298955]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00096: early stopping
[0.8872059871703493, 0.8872059871703493, 0.8872059871703494, 0.
 ↪874061027009303]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00090: early stopping
[0.8800784034212402, 0.8800784034212402, 0.8800784034212402, 0.
 ↪8661096664833664]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00124: early stopping
[0.8866714183891661, 0.8866714183891661, 0.8866714183891661, 0.
 ↪8735009741103202]
[250, '0.3']
Epoch 00273: early stopping
[0.8107626514611547, 0.8107626514611547, 0.8107626514611547, 0.
 ↪7883434831162922]
[250, '0.3', 250, '0.3']
Epoch 00182: early stopping
```

```
[0.8809693513898789, 0.8809693513898789, 0.8809693513898789, 0.
 ↪8671143715434837]
[250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00133: early stopping
[0.8847113328581611, 0.8847113328581611, 0.884711332858161, 0.
 ↪8712432971205801]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00164: early stopping
[0.8850677120456165, 0.8850677120456165, 0.8850677120456164, 0.
 ↪871636611045392]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00189: early stopping
[0.8875623663578047, 0.8875623663578047, 0.8875623663578046, 0.
 ↪874455680487535]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00142: early stopping
[0.8711689237348539, 0.8711689237348539, 0.871168923734854, 0.
 ↪856223700873793]
```

```
[56]: joblib.dump(results_dropout, 'results_dropout_one_hot')
```

```
[56]: ['results_dropout_one_hot']
```

Datos *one_hot SMOTE dropout*

```
[28]: results_dropout_smote = []
      seed = 1
```

```
[29]: size_config = [50, 100, 150, 200, 250]
      dropout_rate = ["0.1", "0.2", "0.3"]

      for size in size_config:
          for size_d in (dropout_rate):
              layer_config_dense = [[size], [size]*2, [size]*3, [size]*4,␣
       ↪[size]*5, [size]*6]
              layer_config_dropout = [[size_d], [size_d]*2, [size_d]*3,␣
       ↪[size_d]*4, [size_d]*5, [size_d]*6]
              for layers_dense, layers_dropout in zip(layer_config_dense,␣
       ↪layer_config_dropout):
                  final_design = [None]*(len(layers_dense)+len(layers_dropout))
                  final_design[::2] = layers_dense
                  final_design[1::2] = layers_dropout
                  np.random.seed(seed)
                  tf.random.set_seed(seed)
```

```
            print(final_design)
            model = make_my_model_multi_dropout(final_design, 40, 18,␣
↪activation_='relu' )
            preds = compile_fit_multiclass(model, X_train_smote, X_test,␣
↪y_train_smote, 256, 300, verbose=0)
            metrics = compute_metrics_multiclass(np.argmax(preds, axis =␣
↪1), np.argmax(y_test, axis = 1))
            confusion = confusion_matrix(np.argmax(preds, axis = 1), np.
↪argmax(y_test, axis = 1))
            aux = { "layer config" : final_design,
                    #"Model": model,
                    "Predictions" : preds,
                    "Metrics" : metrics,
                    "Confusion" : confusion

            }
            print(metrics)
            results_dropout_smote.append(aux)
```

```
[50, '0.1']
[0.17925873129009265, 0.17925873129009265, 0.17925873129009265,
0.12232832383335368]
[50, '0.1', 50, '0.1']
[0.1261582323592302, 0.1261582323592302, 0.1261582323592302,
0.08089709192596017]
[50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00248: early stopping
[0.0650392017106201, 0.0650392017106201, 0.0650392017106201,
0.03040751666739172]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00189: early stopping
[0.042230933713471135, 0.042230933713471135, 0.042230933713471135,
0.01270048583862049]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00194: early stopping
[0.038132573057733425, 0.038132573057733425, 0.038132573057733425,
0.005452686722108413]
[50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1', 50, '0.1']
Epoch 00232: early stopping
[0.0345687811831789, 0.0345687811831789, 0.0345687811831789,
0.008312049551169154]
[50, '0.2']
[0.1626870990734141, 0.1626870990734141, 0.1626870990734141,
0.10466572690499598]
[50, '0.2', 50, '0.2']
Epoch 00255: early stopping
[0.13863150392017107, 0.13863150392017107, 0.13863150392017107,
0.085656992318495]
[50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00128: early stopping
[0.05737704918032787, 0.05737704918032787, 0.05737704918032787,
0.02477694026158095]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00191: early stopping
[0.03189593727726301, 0.03189593727726301, 0.03189593727726301,
0.0039150488723641574]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00126: early stopping
[0.033856022808267994, 0.033856022808267994, 0.033856022808267994,
0.009683362769514314]
[50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2', 50, '0.2']
Epoch 00153: early stopping
[0.043478260869565216, 0.043478260869565216, 0.043478260869565216,
0.021719267754409355]
[50, '0.3']
[0.17587312900926586, 0.17587312900926586, 0.17587312900926586,
0.12135664118870049]
[50, '0.3', 50, '0.3']
Epoch 00250: early stopping
[0.101568068424804, 0.101568068424804, 0.101568068424804, 0.
 ↪05585754164033441]
[50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00186: early stopping
[0.049002138275124736, 0.049002138275124736, 0.04900213827512473,
0.025606993164014047]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00152: early stopping
[0.023521026372059873, 0.023521026372059873, 0.023521026372059876,
-0.0008839246776410903]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00122: early stopping
[0.019600855310049892, 0.019600855310049892, 0.019600855310049892,
0.0035310058483916107]
[50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3', 50, '0.3']
Epoch 00110: early stopping
[0.02690662865288667, 0.02690662865288667, 0.02690662865288667,
0.009679007414853946]
```

```
[100, '0.1']
[0.20224518888096935, 0.20224518888096935, 0.20224518888096935,
0.14544292498918487]
[100, '0.1', 100, '0.1']
Epoch 00240: early stopping
[0.24679258731290094, 0.24679258731290094, 0.24679258731290094,
0.1934243712900997]
[100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00198: early stopping
[0.09230220955096223, 0.09230220955096223, 0.09230220955096224,
0.05140908127877386]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
[0.07305773342836779, 0.07305773342836779, 0.07305773342836779,
0.03229868220347709]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00239: early stopping
[0.05238774055595153, 0.05238774055595153, 0.05238774055595153,
0.01593217893389176]
[100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1', 100, '0.1']
Epoch 00220: early stopping
[0.05327868852459016, 0.05327868852459016, 0.05327868852459016,
0.018773343435891765]
[100, '0.2']
[0.2241625089094797, 0.2241625089094797, 0.2241625089094797,
0.16380451668419682]
[100, '0.2', 100, '0.2']
[0.20990734141126158, 0.20990734141126158, 0.20990734141126158,
0.1564546087020312]
[100, '0.2', 100, '0.2', 100, '0.2']
[0.0935495367070563, 0.0935495367070563, 0.0935495367070563,
0.049653103710961655]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00172: early stopping
[0.039379900213827514, 0.039379900213827514, 0.039379900213827514,
0.010444041860423137]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
[0.06147540983606557, 0.06147540983606557, 0.06147540983606557,
0.016954035230797748]
[100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2', 100, '0.2']
Epoch 00139: early stopping
[0.05915894511760513, 0.05915894511760513, 0.05915894511760513,
0.018481805134154428]
[100, '0.3']
[0.20420527441197434, 0.20420527441197434, 0.20420527441197434,
0.146919703577035]
```

```
[100, '0.3', 100, '0.3']
[0.17230933713471133, 0.17230933713471133, 0.17230933713471133,
0.11901675774047382]
[100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00193: early stopping
[0.042943692088382036, 0.042943692088382036, 0.042943692088382036,
0.01206205191427645]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00164: early stopping
[0.03991446899501069, 0.03991446899501069, 0.03991446899501069,
0.00613134448216468]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00189: early stopping
[0.04240912330719886, 0.04240912330719886, 0.04240912330719886,
0.00991607580343623]
[100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3', 100, '0.3']
Epoch 00179: early stopping
[0.03563791874554526, 0.03563791874554526, 0.03563791874554526,
0.012061464890697371]
[150, '0.1']
[0.1970776906628653, 0.1970776906628653, 0.1970776906628653,
0.14580931580526268]
[150, '0.1', 150, '0.1']
[0.3145046329294369, 0.3145046329294369, 0.3145046329294369,
0.26160453102930736]
[150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00217: early stopping
[0.17836778332145403, 0.17836778332145403, 0.17836778332145403,
0.13225478504378496]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
[0.09390591589451176, 0.09390591589451176, 0.09390591589451176,
0.057542665344226474]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
[0.0718104062722737, 0.0718104062722737, 0.0718104062722737,
0.033419143155595354]
[150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1', 150, '0.1']
Epoch 00233: early stopping
[0.0616535994297933, 0.0616535994297933, 0.0616535994297933,
0.02521055524335547]
[150, '0.2']
[0.22042052744119744, 0.22042052744119744, 0.22042052744119744,
0.1651157706698071]
[150, '0.2', 150, '0.2']
Epoch 00264: early stopping
[0.23431931575196008, 0.23431931575196008, 0.23431931575196008,
0.1814566324589243]
[150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00181: early stopping
[0.12883107626514612, 0.12883107626514612, 0.12883107626514612,
0.08123675934587382]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00227: early stopping
[0.08125445473984319, 0.08125445473984319, 0.08125445473984319,
0.042625704486619176]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00284: early stopping
[0.05666429080541696, 0.05666429080541696, 0.05666429080541696,
0.022088509992709504]
[150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2', 150, '0.2']
Epoch 00206: early stopping
[0.04686386315039202, 0.04686386315039202, 0.04686386315039202,
0.013243538873656369]
[150, '0.3']
[0.2166785459729152, 0.2166785459729152, 0.2166785459729152,
0.16013192583393254]
[150, '0.3', 150, '0.3']
Epoch 00269: early stopping
[0.20153243050605846, 0.20153243050605846, 0.20153243050605846,
0.1489163218797086]
[150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00233: early stopping
[0.09141126158232359, 0.09141126158232359, 0.09141126158232359,
0.05088555218453383]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00235: early stopping
[0.048467569493941556, 0.048467569493941556, 0.048467569493941556,
0.01460299355630068]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00239: early stopping
[0.04436920883820385, 0.04436920883820385, 0.044369208838203854,
0.0130637778773115]
[150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3', 150, '0.3']
Epoch 00229: early stopping
[0.04526015680684248, 0.04526015680684248, 0.045260156806842484,
0.007616860340662668]
[200, '0.1']
[0.2533856022808268, 0.2533856022808268, 0.2533856022808268, 0.
 ↪197391515941576]
[200, '0.1', 200, '0.1']
```

```
[0.3708125445473984, 0.3708125445473984, 0.3708125445473984, 0.
 ↪3210530887142903]
[200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00276: early stopping
[0.22202423378474698, 0.22202423378474698, 0.22202423378474698,
0.17342244981580968]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00187: early stopping
[0.11546685673556664, 0.11546685673556664, 0.11546685673556664,
0.07693413882061018]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00295: early stopping
[0.09248039914468995, 0.09248039914468995, 0.09248039914468995,
0.054437648776313186]
[200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1', 200, '0.1']
Epoch 00223: early stopping
[0.07056307911617961, 0.07056307911617961, 0.07056307911617961,
0.03598036748298006]
[200, '0.2']
[0.2569493941553813, 0.2569493941553813, 0.2569493941553813, 0.
 ↪1990989622048731]
[200, '0.2', 200, '0.2']
Epoch 00300: early stopping
[0.28813257305773343, 0.28813257305773343, 0.28813257305773343,
0.23566194103112692]
[200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00297: early stopping
[0.216143977191732, 0.216143977191732, 0.216143977191732, 0.
 ↪16476544862583697]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00206: early stopping
[0.09016393442622951, 0.09016393442622951, 0.09016393442622953,
0.051738070452585716]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
[0.08107626514611546, 0.08107626514611546, 0.08107626514611546,
0.04782415240770388]
[200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2', 200, '0.2']
Epoch 00194: early stopping
[0.06290092658588739, 0.06290092658588739, 0.06290092658588739,
0.02886584321690111]
[200, '0.3']
[0.2753029223093371, 0.2753029223093371, 0.2753029223093371, 0.
 ↪2168872081718204]
[200, '0.3', 200, '0.3']
```

```
[0.231111903064861, 0.231111903064861, 0.231111903064861, 0.
↪17615806450636173]
[200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00246: early stopping
[0.10727013542409124, 0.10727013542409124, 0.10727013542409124,
0.06523663598466534]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
[0.061831789023521024, 0.061831789023521024, 0.061831789023521024,
0.03089862212446237]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00245: early stopping
[0.043300071275837494, 0.043300071275837494, 0.043300071275837494,
0.011053024973720404]
[200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3', 200, '0.3']
Epoch 00280: early stopping
[0.040805416963649324, 0.040805416963649324, 0.040805416963649324,
0.011774770560165515]
[250, '0.1']
[0.26300784034212404, 0.26300784034212404, 0.26300784034212404,
0.20673535816629096]
[250, '0.1', 250, '0.1']
Epoch 00241: early stopping
[0.3558446186742694, 0.3558446186742694, 0.35584461867426936,
0.3066293157389701]
[250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00169: early stopping
[0.24358517462580184, 0.24358517462580184, 0.24358517462580184,
0.1961415507087051]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00167: early stopping
[0.1443335709194583, 0.1443335709194583, 0.1443335709194583,
0.10284190805322913]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00163: early stopping
[0.08267997148966501, 0.08267997148966501, 0.08267997148966501,
0.04849906766953527]
[250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1', 250, '0.1']
Epoch 00245: early stopping
[0.08856022808267996, 0.08856022808267996, 0.08856022808267996,
0.05350068877068037]
[250, '0.2']
[0.2966856735566643, 0.2966856735566643, 0.2966856735566643,
0.23811703510767068]
[250, '0.2', 250, '0.2']
```

```
[0.3670705630791162, 0.3670705630791162, 0.3670705630791161, 0.
 ↪3162973225364475]
[250, '0.2', 250, '0.2', 250, '0.2']
[0.17979330007127584, 0.17979330007127584, 0.17979330007127584,
0.1300558613010846]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00276: early stopping
[0.12081254454739843, 0.12081254454739843, 0.12081254454739844,
0.08267604380040405]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
Epoch 00188: early stopping
[0.07323592302209551, 0.07323592302209551, 0.07323592302209551,
0.039733431057557]
[250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2', 250, '0.2']
[0.07377049180327869, 0.07377049180327869, 0.07377049180327869,
0.04105625909595611 6]
[250, '0.3']
[0.3132573057733428, 0.3132573057733428, 0.3132573057733428,
0.2529062743595671 4]
[250, '0.3', 250, '0.3']
Epoch 00284: early stopping
[0.2425160370634355, 0.2425160370634355, 0.2425160370634355,
0.19115487345844973]
[250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00270: early stopping
[0.10762651461154668, 0.10762651461154668, 0.10762651461154668,
0.0654504653495317]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00211: early stopping
[0.0778688524590164, 0.0778688524590164, 0.0778688524590164,
0.04145904731879446]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
Epoch 00232: early stopping
[0.05167498218104063, 0.05167498218104063, 0.05167498218104063,
0.019842027765843095]
[250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3', 250, '0.3']
[0.05434782608695652, 0.05434782608695652, 0.05434782608695652,
0.026805608899612032]
```

[30]:
```
import joblib
joblib.dump(results_dropout_smote, 'results_dropout_smote_one_hot')
```

[30]:
```
['results_dropout_smote_one_hot']
```

3.3 Análisis de los resultados

Una vez que hemos producido los experimentos, leemos los conjuntos de datos y creamos un dataset con todas las configuraciones y las métricas obtenidas. Para ello hacemos uso de la función *ReadAndCreate*.

[23]:
```
import numpy as np
import pandas as pd
import joblib
```

[24]:
```
layer_config = []
Precision = []
Recall = []
F1 = []
Cohen_kappa = []
```

[25]:
```
def ReadAndCreate(file):
    results_1 = joblib.load(file)
    layer_config = []
    Precision = []
    Recall = []
    F1 = []
    Cohen_kappa = []
    for i in range(len(results_1)):
        aux = results_1[i]
        layer_config.append(aux['layer config'])
        Precision.append(aux['Metrics'][0])
        Recall.append(aux['Metrics'][1])
        F1.append(aux['Metrics'][2])
        Cohen_kappa.append(aux['Metrics'][3])

    data = {'Hidden layers' : layer_config,
            'Precision' : Precision,
           'Recall' : Recall,
            'F1' : F1,
           'Cohen kappa' : Cohen_kappa}
    data = pd.DataFrame(data)
    return data
```

3.3.1 Resultados con datos raw

[26]:
```
results_data = ReadAndCreate("results_1_joblib")
results_data.sort_values("Precision", ascending=False).head(6)
```

```
[26]:                      Hidden layers  Precision    Recall        F1  Cohen␣
     ↪kappa
     28       [250, 250, 250, 250, 250]   0.825196  0.825196  0.825196     0.
     ↪804531
     29  [250, 250, 250, 250, 250, 250]   0.821810  0.821810  0.821810     0.
     ↪801151
     16       [150, 150, 150, 150, 150]   0.820919  0.820919  0.820919     0.
     ↪799992
     23  [200, 200, 200, 200, 200, 200]   0.819138  0.819138  0.819138     0.
     ↪797986
     17  [150, 150, 150, 150, 150, 150]   0.818425  0.818425  0.818425     0.
     ↪797329
     27            [250, 250, 250, 250]   0.816108  0.816108  0.816108     0.
     ↪794788
```

```
[27]: results_data.sort_values("Precision", ascending=True).head(6)
```

```
[27]:   Hidden layers  Precision    Recall        F1  Cohen kappa
     0            [50]   0.556308  0.556308  0.556308     0.501478
     6           [100]   0.584105  0.584105  0.584105     0.532926
     12          [150]   0.605132  0.605132  0.605132     0.557198
     18          [200]   0.628653  0.628653  0.628653     0.584144
     24          [250]   0.641661  0.641661  0.641661     0.599029
     1        [50, 50]   0.677655  0.677655  0.677655     0.639487
```

Estudiando los datos con mayor puntuación en Precision descubrimos que los modelos con mayor número de neuronas y capas obtienen buenas puntuaciones, sin llegar al 83%. Inicialmente podríamos esperar que mayor número de neuronas está directamente relacionado con puntuación en metricas, pero vemos que no es así. Configuraciones de capas distintas con números de neuronas por encima de 150 dan lugar a resultados similares. Probablemente, la diferencia en puntuación se deba a la naturaleza aleatoria de las redes neuronales a la hora de inicializar los pesos así como en el algoritmo de minimización. Todos los modelos con características parecidas dan lugar a resultados muy similares.

Por otro lado, en los peores resultados obtenemos lo esperado. Los modelos de una capa con pocas neuronas clasifican muy mal y mejoran su puntuación en orden creciente de neuronas.

3.3.2 Resultados datos raw one-hot

```
[28]: results_data = ReadAndCreate("results_1_onehot_joblib")
      results_data.sort_values("Precision", ascending=False).head(6)
```

```
[28]:                Hidden layers  Precision    Recall        F1  Cohen kappa
     21       [200, 200, 200, 200]   0.830898  0.830898  0.830898     0.811140
     26            [250, 250, 250]   0.830185  0.830185  0.830185     0.810440
```

```
25                [250, 250]   0.824127  0.824127  0.824127     0.803536
27      [250, 250, 250, 250]   0.817177  0.817177  0.817177     0.795549
22  [200, 200, 200, 200, 200]  0.814505  0.814505  0.814505     0.793240
20           [200, 200, 200]   0.809694  0.809694  0.809694     0.787236
```

[29]:
```
results_data.sort_values("Precision", ascending=True).head(6)
```

[29]:
```
                Hidden layers  Precision    Recall        F1  Cohen kappa
0                        [50]   0.713293  0.713293  0.713293     0.679594
5  [50, 50, 50, 50, 50, 50]   0.724519  0.724519  0.724519     0.692311
3            [50, 50, 50, 50]   0.729330  0.729330  0.729330     0.697647
4        [50, 50, 50, 50, 50]   0.732359  0.732359  0.732359     0.701160
1                    [50, 50]   0.735923  0.735923  0.735923     0.705121
6                       [100]   0.740556  0.740556  0.740556     0.710106
```

Estudiamos los modelos con mayor puntuación. El uso de usar la codificación *one-hot* de las variables dan lugar a prácticamente los mismos resultados que sin usar esta transformación. Además, los modelos que consiguen las mayores puntuaciones también son muy parecidos.

Sin embargo, en los modelos que obtienen los peores resultados, éstos son muchos mejores que los obtenidos por los peores modelos sin usar *one-hot*. Las peores métricas obtenidas está acotadas inferiormente, en el caso de una sola capa oculta de 50 neuronas, en el 71%. Esto nos puede indicar que, mientras que usar *one-hot* es beneficioso en este tipo de modelos, sea difícil mejorar los resultados obtenidos en los mejores casos.

3.3.3 Resultados datos raw smote

[30]:
```
results_data = ReadAndCreate("results_smote_joblib")
results_data.sort_values("Precision", ascending=False).head(6)
```

[30]:
```
                        Hidden layers  Precision    Recall        F1  Cohen␣
 ↪kappa
29  [250, 250, 250, 250, 250, 250]   0.361725  0.361725  0.361725     0.
 ↪300894
23  [200, 200, 200, 200, 200, 200]   0.356557  0.356557  0.356557     0.
 ↪296138
16       [150, 150, 150, 150, 150]   0.355132  0.355132  0.355132     0.
 ↪295350
27            [250, 250, 250, 250]   0.354775  0.354775  0.354775     0.
 ↪294756
17  [150, 150, 150, 150, 150, 150]   0.354419  0.354419  0.354419     0.
 ↪293245
28       [250, 250, 250, 250, 250]   0.349786  0.349786  0.349786     0.
 ↪288579
```

```
[31]: results_data.sort_values("Precision", ascending=True).head(6)
```

```
[31]:    Hidden layers  Precision    Recall        F1  Cohen kappa
      0           [50]   0.273521  0.273521  0.273521     0.210563
      6          [100]   0.299715  0.299715  0.299715     0.237226
      12         [150]   0.300249  0.300249  0.300249     0.238289
      1       [50, 50]   0.301675  0.301675  0.301675     0.238587
      18         [200]   0.312723  0.312723  0.312723     0.250874
      2   [50, 50, 50]   0.319672  0.319672  0.319672     0.257319
```

En un intento de mejorar el desbalanceo existente en las clases a clasificar, utilizamos el algoritmo SMOTE para mejorar el dataset. El motivo por el que usamos un algoritmo de sobremuestreo en vez de *tirar a la baja* con undersampling es por el beneficio que tienen las redes neuronales al aumentar el tamaño del dataset. Además, en pruebas realizadas con árboles de decisión en otras asignaturas, descubrimos que, a pesar de romper el problema de desbalanceo, obtenemos peores resultados al disponer de menos casos.

No ha mejorado la clasificación. Los resultados son muy malos. Las mejores redes obtienen métricas alrededor del 35%, mientras que las mejoras obtienen un 27%. Al haber tan poca diferencia entre los mejores y peores resultados, podemos estar seguros que no se debe a un número insuficiente de neuronas y capas.

En los problemas en los que las clases están muy desbalanceadas, SMOTE no funciona bien. Esta es una de esas ocasiones. Además, este algoritmo funciona mejor en problemas de clasificación binarios, donde fue concebido.

Como hemos mencionado antes, SMOTE está diseñado para problemas de predictores continuos, donde es más natural la interpolación. Sin embargo, como vimos que no daba ningún error al ejecutar la librería, vimos interesante ver el desempeño de la misma en problemas categóricos.

3.3.4 Resultados datos smote one-hot

```
[32]: results_data = ReadAndCreate("results_smote_onehot_joblib")
      results_data.sort_values("Precision", ascending=False).head(6)
```

```
[32]:       Hidden layers  Precision    Recall        F1  Cohen kappa
      19       [200, 200]   0.187634  0.187634  0.187634     0.144296
      26  [250, 250, 250]   0.177833  0.177833  0.177833     0.131770
      20  [200, 200, 200]   0.152174  0.152174  0.152174     0.107797
      25       [250, 250]   0.145937  0.145937  0.145937     0.103981
      14  [150, 150, 150]   0.138097  0.138097  0.138097     0.091480
      13       [150, 150]   0.129187  0.129187  0.129187     0.091482
```

```
[33]: results_data.sort_values("Precision", ascending=True).head(6)
```

```
[33]:                     Hidden layers  Precision    Recall        F1  Cohen␣
     ↪kappa
     5          [50, 50, 50, 50, 50, 50]   0.054348  0.054348  0.054348     0.
     ↪018471
     17  [150, 150, 150, 150, 150, 150]   0.062545  0.062545  0.062545     0.
     ↪026222
     16       [150, 150, 150, 150, 150]   0.066999  0.066999  0.066999     0.
     ↪031643
     4              [50, 50, 50, 50, 50]   0.067890  0.067890  0.067890     0.
     ↪028956
     1                          [50, 50]   0.069316  0.069316  0.069316     0.
     ↪037007
     2                      [50, 50, 50]   0.069672  0.069672  0.069672     0.
     ↪034412
```

La codificación one-hot ha empeorado los resultados con SMOTE. Las métricas están acotadas entre un 18% y un 5%.

3.3.5 Resultados datos raw con dropout

```
[34]: results_data = ReadAndCreate("results_dropout")
      results_data.sort_values("Precision", ascending=False).head(6)
```

```
[34]:                                         Hidden layers  Precision    ␣
     ↪Recall  \
     94  [250, 0.1, 250, 0.1, 250, 0.1, 250, 0.1, 250, …   0.874020  0.874020
     76  [200, 0.1, 200, 0.1, 200, 0.1, 200, 0.1, 200, …   0.873664  0.873664
     95  [250, 0.1, 250, 0.1, 250, 0.1, 250, 0.1, 250, …   0.871347  0.871347
     74                   [200, 0.1, 200, 0.1, 200, 0.1]   0.870991  0.870991
     75        [200, 0.1, 200, 0.1, 200, 0.1, 200, 0.1]   0.866180  0.866180
     77  [200, 0.1, 200, 0.1, 200, 0.1, 200, 0.1, 200, …   0.864754  0.864754

               F1  Cohen kappa
     94  0.874020     0.859249
     76  0.873664     0.858904
     95  0.871347     0.856290
     74  0.870991     0.855863
     75  0.866180     0.850478
     77  0.864754     0.848903
```

```
[35]: results_data.sort_values("Precision", ascending=True).head(6)
```

```
[35]:                                         Hidden layers  Precision    ␣
     ↪Recall  \
30                                       [50, 0.3]   0.511048  0.511048
24                                       [50, 0.2]   0.517284  0.517284
35  [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, …   0.522630  0.522630
34      [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3]   0.526372  0.526372
0                                        [50, 0.1]   0.527263  0.527263
18                                       [50, 0.1]   0.530649  0.530649

          F1  Cohen kappa
30  0.511048     0.449370
24  0.517284     0.456681
35  0.522630     0.463069
34  0.526372     0.466761
0   0.527263     0.469118
18  0.530649     0.472995
```

Los mejores resultados mejoran levemente los obtenidos sin dropout. Al igual que en estos últimos, las modelos más grandes obtienen mejores resultados. Notemos que todos los que aparecen con mejores métricas tienen un porcentaje de desactivación muy pequeño.

En el polo opuesto, los modelos que obtuvieron peor resultado fueron los modelos con el mayor porcentaje de desactivación en dropout: 30% y menor número de neuronas. Esto es lógico, ya que no parece ser un dataset lo suficientemente grande como para que se produzca un sobreaprendizaje durante el entrenamiento.

Recordamos que el número de épocas en el entrenamiento viene controlado por tensorflow mediante los *Callbacks*, como comentamos anteriormente, por lo que si no mejora la *accuracy* del conjunto de validación en 10 épocas, finaliza el entrenamiento.

3.3.6 Resultados datos raw one-hot con dropout

```
[36]: results_data = ReadAndCreate("results_dropout_one_hot")
      results_data.sort_values("Precision", ascending=False).head(6)
```

```
[36]:                                        Hidden layers  Precision  ␣
  ↪Recall  \
77  [250, 0.1, 250, 0.1, 250, 0.1, 250, 0.1, 250, …   0.894512  0.894512
62                     [200, 0.2, 200, 0.2, 200, 0.2]   0.892908  0.892908
79                               [250, 0.2, 250, 0.2]   0.891839  0.891839
63           [200, 0.2, 200, 0.2, 200, 0.2, 200, 0.2]   0.888988  0.888988
56                     [200, 0.1, 200, 0.1, 200, 0.1]   0.888453  0.888453
80                     [250, 0.2, 250, 0.2, 250, 0.2]   0.887741  0.887741

          F1  Cohen kappa
77  0.894512     0.882191
62  0.892908     0.880426
```

```
79  0.891839     0.879262
63  0.888988     0.876026
56  0.888453     0.875431
80  0.887741     0.874633
```

[37]:
```
results_data.sort_values("Precision", ascending=True).head(6)
```

[37]:
```
                                        Hidden layers  Precision   ␣
 ↪Recall  \
17  [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, …   0.607627  0.607627
16      [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3]   0.640770  0.640770
15                [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3]   0.655560  0.655560
12                                           [50, 0.3]   0.658411  0.658411
14                          [50, 0.3, 50, 0.3, 50, 0.3]   0.669280  0.669280
6                                            [50, 0.2]   0.679793  0.679793

          F1  Cohen kappa
17  0.607627     0.561731
16  0.640770     0.597454
15  0.655560     0.614280
12  0.658411     0.617240
14  0.669280     0.629326
6   0.679793     0.641515
```

Los resultados son esperables vistos los casos anteriores. Dropout mejora también el caso en el que usamos *one-hot* encoding. Estos son los mejores resultados que obtenemos.

3.3.7 Resultados datos raw smote con dropout

[38]:
```
results_data = ReadAndCreate("results_dropout_smote")
results_data.sort_values("Precision", ascending=False).head(6)
```

[38]:
```
                                        Hidden layers  Precision   ␣
 ↪Recall  \
62                     [200, 0.2, 200, 0.2, 200, 0.2]   0.403243  0.403243
58  [200, 0.1, 200, 0.1, 200, 0.1, 200, 0.1, 200, …   0.400214  0.400214
77  [250, 0.1, 250, 0.1, 250, 0.1, 250, 0.1, 250, …   0.400036  0.400036
83  [250, 0.2, 250, 0.2, 250, 0.2, 250, 0.2, 250, …   0.399501  0.399501
80                     [250, 0.2, 250, 0.2, 250, 0.2]   0.399501  0.399501
74                     [250, 0.1, 250, 0.1, 250, 0.1]   0.398610  0.398610

          F1  Cohen kappa
62  0.403243     0.346788
58  0.400214     0.342223
```

```
77  0.400036     0.341473
83  0.399501     0.341194
80  0.399501     0.342195
74  0.398610     0.341175
```

[39]:
```
results_data.sort_values("Precision", ascending=True).head(6)
```

[39]:
```
                                        Hidden layers  Precision  ␣
 ↪Recall  \
17  [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, …   0.215966  0.215966
16      [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3]   0.227548  0.227548
15               [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3]   0.243763  0.243763
12                                          [50, 0.3]   0.245902  0.245902
6                                           [50, 0.2]   0.249465  0.249465
48                                         [150, 0.3]   0.254098  0.254098

          F1  Cohen kappa
17  0.215966     0.151163
16  0.227548     0.161465
15  0.243763     0.179339
12  0.245902     0.185292
6   0.249465     0.187577
48  0.254098     0.189198
```

Se obtienen mejores resultados que sin incluir dropout, pero siguen siendo malos.

3.3.8 Resultados datos smote one-hot con dropout

[40]:
```
results_data = ReadAndCreate("results_dropout_smote_one_hot")
results_data.sort_values("Precision", ascending=False).head(6)
```

[40]:
```
             Hidden layers  Precision    Recall        F1  Cohen kappa
55  [200, 0.1, 200, 0.1]   0.370813  0.370813  0.370813     0.321053
79  [250, 0.2, 250, 0.2]   0.367071  0.367071  0.367071     0.316297
73  [250, 0.1, 250, 0.1]   0.355845  0.355845  0.355845     0.306629
37  [150, 0.1, 150, 0.1]   0.314505  0.314505  0.314505     0.261605
84            [250, 0.3]   0.313257  0.313257  0.313257     0.252906
78            [250, 0.2]   0.296686  0.296686  0.296686     0.238117
```

[41]:
```
results_data.sort_values("Precision", ascending=True).head(6)
```

[41]:
```
                                        Hidden layers  Precision  ␣
 ↪Recall  \
16      [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3]   0.019601  0.019601
```

```
15               [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3]   0.023521  0.023521
17  [50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, 50, 0.3, …   0.026907  0.026907
9                [50, 0.2, 50, 0.2, 50, 0.2, 50, 0.2]   0.031896  0.031896
10      [50, 0.2, 50, 0.2, 50, 0.2, 50, 0.2, 50, 0.2]   0.033856  0.033856
5   [50, 0.1, 50, 0.1, 50, 0.1, 50, 0.1, 50, 0.1, …   0.034569  0.034569

          F1  Cohen kappa
16  0.019601     0.003531
15  0.023521    -0.000884
17  0.026907     0.009679
9   0.031896     0.003915
10  0.033856     0.009683
5   0.034569     0.008312
```

Se repite el comportamiento, usar dropout con un porcentaje de desactivación pequeño mejora los resultados, pero éstos eran muy malos per se.

3.4 Conclusiones

Hemos construido modelos de redes neuronales del tipo perceptrón multicapa con distintas configuraciones de capas ocultas y neuronas. Los datos utilizados han sido los originales, la codificación de las variables predictoras mediante *one-hot* y sus equivalentes tras usar *SMOTE*. También se ha implementado *callbacks* para parar el entrenamiento si el valor de *accuracy* en el set de validación no mejora tras 10 épocas, evitando el sobreaprendizaje y retirando el número de épocas como parámetro a estudiar y modificar. A todos estos modelos, se hicieron experimentos introduciendo capas intermedias con *dropout* con tres posibles valores de desactivación de neuronas: 10, 20 y 30%.

Los resultados son los siguientes. El uso de *SMOTE* no aporta ningún beneficio. Las métricas obtenidas son mucho peores que usando datos sin *SMOTE*. La razón puede deberse al gran desbalanceo entre clases que existe en la variable a predecir y la naturaleza categórica de las predictoras.

Introducir *dropout* mejora el resultado en todos los casos, incluso en los peores.

Creemos haber llegado a resultados próximos a los máximos posibles usando perceptrones multicapa ya que, en los casos con mejor puntuación, no existe una relación directa entre mayor número de neuronas y rendimiento. Se dan casos en los que configuraciones con menos capas obtienen mejores resultados que equivalentes con más. Sin embargo, está claro que un número grande neuronas es beneficioso para el modelo.

Todas las métricas se comportan de forma parecida: si un modelo obtiene mejor resultado que otro, todas sus métricas son mejores que las del otro.

4 Creación de árboles de clasificación con TensorFlowDecisionTrees

Con el fin de investigar más y aprendiendo sobre TensorFlow, descubrimos que, recientemente (Mayo de 2021), se ha incluido en la librería una API en la que se implementan tres algoritmos muy famosos de árboles de decisión: Random Forest, Gradient Boosted Trees y CART.

Estos algoritmos, los cuales hemos visto en el máster, se ejecutan en C++ con la librería, también de Google, Yggdrasil Decision Forests. En realidad, al igual que la API en Python de TensorFlow, estamos utilizando un wrapper en el cual creamos los modelos fácilmente en Python y posteriormente serán entrenados en C++. De momento, esta librería no hace uso de la tarjeta gráfica.

La API se llama TensorFlow Decision Forest y puede ser instalada con `pip3 install tensorflow_decision_forests --upgrade`.

Probaremos los tres algoritmos y modificaremos ciertos parámetros para el entrenamiento.

Al ser una nueva librería, nos parece interesante ir comentando cómo construir los modelos.

La lectura de datos se realiza de la misma forma que en casos anteriores:

[1]:
```
import pandas as pd
import numpy as np

import tensorflow_decision_forests as tfdf
from wurlitzer import sys_pipes
```

[2]:
```
data = pd.read_csv("krkopt.data", header=None)
data.columns = ["wkc", "wkr", "wrc", "wrr", "bkc", "bkr", "opt rank" ]
```

[3]:
```
data.head()
```

[3]:
```
  wkc  wkr wrc  wrr bkc  bkr opt rank
0   a    1   b    3   c    2     draw
1   a    1   c    1   c    2     draw
2   a    1   c    1   d    1     draw
3   a    1   c    1   d    2     draw
4   a    1   c    2   c    1     draw
```

Es necesario destacar cuál va a ser la columna que contenga la variable a predecir para usar *tfdf*. En nuestro caso, *opt rank*.

Mostramos también las clases de esta variable:

[4]:
```
label = "opt rank"

classes = data[label].unique().tolist()
```

```
print(f"Label classes: {classes}")

#convert to integer category

data[label] = data[label].map(classes.index)
```

```
Label classes: ['draw', 'zero', 'one', 'two', 'three', 'four', 'five',␣
↪'six',
'seven', 'eight', 'nine', 'ten', 'eleven', 'twelve', 'thirteen', 'fourteen',
'fifteen', 'sixteen']
```

Creamos el conjunto de entrenamiento y test con una proporción 80-20.

[5]:
```
seed = 1
np.random.seed(seed)
```

[6]:
```
def split_dataset(dataset, test_ratio=0.2):
    test_indices = np.random.rand(len(dataset)) < test_ratio
    return dataset[~test_indices], dataset[test_indices]

train, test = split_dataset(data)
test_aux = test.copy()
```

[7]:
```
print("{} examples in training, {} examples for testing.".format(
    len(train), len(test)))
```

```
22450 examples in training, 5606 examples for testing.
```

Encontramos las primeras pecularidades. Es necesario utilizar la función *pd_dataframe_to_tf_dataset* para transformar nuestro conjunto de datos a un objeto que pueda ser procesado por la librería *yggrdrasil*. Más adelante comentaremos un *bug* que hemos encontrado en esta función que nos impide realizar ciertas pruebas.

[8]:
```
train = tfdf.keras.pd_dataframe_to_tf_dataset(train, label=label)
test = tfdf.keras.pd_dataframe_to_tf_dataset(test, label=label)
```

4.1 Creación de Random Forest con datos *raw*

Creamos un modelo de Random Forest usando la función *RandomForestModel*. Es necesario compilar el modelo con la métrica que nosotros queramos. Inicialmente intentamos usar Precision, para poder comparar con las redes neuronales, pero obtuvimos error ya que no parece completamente compatible esta métrica aún. Por ello, usamos *accuracy* y más tarde usamos precision.

Además, en este primer modelo, mostramos un log usando *sys_pipes* que permite ver los entresijos y detalles del entrenamiento. Como vemos, la propia librería detecta las categorías de las variables predictoras.

En el caso de los árboles de decisión, no se utilizan épocas ya que los distintos árboles que componen el bosque de árboles se entrenan con todo el conjunto de entrenamiento. Además, el propio algoritmo tiene una medida de estimación de validación, por lo que tampoco hay que usar conjunto de validación.

[9]:
```
from tensorflow.keras.metrics import Precision
model_rf= tfdf.keras.RandomForestModel()

model_rf.compile(
                metrics=["accuracy"])

with sys_pipes():
    model_rf.fit(x=train)
```

```
2021-06-10 17:12:50.174954: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:176] None of the␣
 ↪MLIR
Optimization Passes are enabled (registered 2)
2021-06-10 17:12:50.195009: I
tensorflow/core/platform/profile_utils/cpu_utils.cc:114] CPU Frequency:
2199995000 Hz

351/351 [==============================] - 3s 592us/step

[INFO kernel.cc:746] Start Yggdrasil model training
[INFO kernel.cc:747] Collect training examples
[INFO kernel.cc:392] Number of batches: 351
[INFO kernel.cc:393] Number of examples: 22450
[INFO kernel.cc:769] Dataset:
Number of records: 22450
Number of columns: 7

Number of columns by type:
        CATEGORICAL: 4 (57.1429%)
        NUMERICAL: 3 (42.8571%)

Columns:

CATEGORICAL: 4 (57.1429%)
        0: "bkc" CATEGORICAL has-dict vocab-size:9 zero-ood-items most-
frequent:"h" 3859 (17.1893%)
        2: "wkc" CATEGORICAL has-dict vocab-size:5 zero-ood-items most-
frequent:"d" 9677 (43.1047%)
        4: "wrc" CATEGORICAL has-dict vocab-size:9 zero-ood-items most-
```

```
frequent:"f" 2930 (13.0512%)
        6: "__LABEL" CATEGORICAL integerized vocab-size:19 no-ood-item

NUMERICAL: 3 (42.8571%)
        1: "bkr" NUMERICAL mean:4.45768 min:1 max:8 sd:2.24698
        3: "wkr" NUMERICAL mean:1.84909 min:1 max:4 sd:0.924757
        5: "wrr" NUMERICAL mean:4.51042 min:1 max:8 sd:2.27881

Terminology:
        nas: Number of non-available (i.e. missing) values.
        ood: Out of dictionary.
        manually-defined: Attribute which type is manually defined by the␣
 ↪user
i.e. the type was not automatically inferred.
        tokenized: The attribute value is obtained through tokenization.
        has-dict: The attribute is attached to a string dictionary e.g. a
categorical attribute stored as a string.
        vocab-size: Number of unique values.

[INFO kernel.cc:772] Configure learner
[INFO kernel.cc:797] Training config:
learner: "RANDOM_FOREST"
features: "bkc"
features: "bkr"
features: "wkc"
features: "wkr"
features: "wrc"
features: "wrr"
label: "__LABEL"
task: CLASSIFICATION
[yggdrasil_decision_forests.model.random_forest.proto.random_forest_config]␣
 ↪{
  num_trees: 300
  decision_tree {
    max_depth: 16
    min_examples: 5
    in_split_min_examples_check: true
    missing_value_policy: GLOBAL_IMPUTATION
    allow_na_conditions: false
    categorical_set_greedy_forward {
      sampling: 0.1
      max_num_items: -1
      min_item_frequency: 1
    }
    growing_strategy_local {
    }
    categorical {
      cart {
      }
    }
    num_candidate_attributes_ratio: -1
    axis_aligned_split {
    }
  }
  winner_take_all_inference: true
  compute_oob_performances: true
  compute_oob_variable_importances: false
  adapt_bootstrap_size_ratio_for_maximum_training_duration: false
}

[INFO kernel.cc:800] Deployment config:

[INFO kernel.cc:837] Train model
[INFO random_forest.cc:303] Training random forest on 22450 example(s) and 6
feature(s).
[INFO random_forest.cc:578] Training of tree  1/300 (tree index:1) done
accuracy:0.663478 logloss:12.1295
[INFO random_forest.cc:578] Training of tree  11/300 (tree index:13) done
accuracy:0.691763 logloss:5.05263
[INFO random_forest.cc:578] Training of tree  21/300 (tree index:21) done
accuracy:0.730233 logloss:2.65216
[INFO random_forest.cc:578] Training of tree  31/300 (tree index:30) done
accuracy:0.742316 logloss:1.95862
[INFO random_forest.cc:578] Training of tree  41/300 (tree index:40) done
accuracy:0.750379 logloss:1.58524
[INFO random_forest.cc:578] Training of tree  51/300 (tree index:50) done
accuracy:0.753497 logloss:1.37915
[INFO random_forest.cc:578] Training of tree  61/300 (tree index:60) done
accuracy:0.755768 logloss:1.26819
[INFO random_forest.cc:578] Training of tree  71/300 (tree index:70) done
accuracy:0.758708 logloss:1.17384
[INFO random_forest.cc:578] Training of tree  81/300 (tree index:80) done
accuracy:0.75902 logloss:1.11116
[INFO random_forest.cc:578] Training of tree  91/300 (tree index:90) done
accuracy:0.760713 logloss:1.0598
[INFO random_forest.cc:578] Training of tree  101/300 (tree index:100) done
accuracy:0.761336 logloss:1.02395
[INFO random_forest.cc:578] Training of tree  111/300 (tree index:110) done
accuracy:0.764276 logloss:0.992012
```

```
[INFO random_forest.cc:578] Training of tree  121/300 (tree index:121) done
accuracy:0.763964 logloss:0.961497
[INFO random_forest.cc:578] Training of tree  131/300 (tree index:131) done
accuracy:0.763563 logloss:0.944803
[INFO random_forest.cc:578] Training of tree  141/300 (tree index:139) done
accuracy:0.764009 logloss:0.928943
[INFO random_forest.cc:578] Training of tree  151/300 (tree index:150) done
accuracy:0.764365 logloss:0.913852
[INFO random_forest.cc:578] Training of tree  161/300 (tree index:160) done
accuracy:0.764811 logloss:0.89613
[INFO random_forest.cc:578] Training of tree  171/300 (tree index:170) done
accuracy:0.765479 logloss:0.881226
[INFO random_forest.cc:578] Training of tree  181/300 (tree index:180) done
accuracy:0.764944 logloss:0.866638
[INFO random_forest.cc:578] Training of tree  191/300 (tree index:191) done
accuracy:0.765345 logloss:0.863178
[INFO random_forest.cc:578] Training of tree  201/300 (tree index:200) done
accuracy:0.764855 logloss:0.848468
[INFO random_forest.cc:578] Training of tree  211/300 (tree index:212) done
accuracy:0.765345 logloss:0.843659
[INFO random_forest.cc:578] Training of tree  221/300 (tree index:218) done
accuracy:0.766414 logloss:0.837226
[INFO random_forest.cc:578] Training of tree  231/300 (tree index:230) done
accuracy:0.76588 logloss:0.834492
[INFO random_forest.cc:578] Training of tree  241/300 (tree index:240) done
accuracy:0.765924 logloss:0.828863
[INFO random_forest.cc:578] Training of tree  251/300 (tree index:250) done
accuracy:0.766503 logloss:0.822741
[INFO random_forest.cc:578] Training of tree  261/300 (tree index:262) done
accuracy:0.766102 logloss:0.819977
[INFO random_forest.cc:578] Training of tree  271/300 (tree index:270) done
accuracy:0.766503 logloss:0.815393
[INFO random_forest.cc:578] Training of tree  281/300 (tree index:281) done
accuracy:0.767261 logloss:0.809227
[INFO random_forest.cc:578] Training of tree  291/300 (tree index:290) done
accuracy:0.767305 logloss:0.804649
[INFO random_forest.cc:578] Training of tree  300/300 (tree index:297) done
accuracy:0.765702 logloss:0.803169
[INFO random_forest.cc:645] Final OOB metrics: accuracy:0.765702
logloss:0.803169
[INFO kernel.cc:856] Export model in log directory: /tmp/tmpusz_27m4
[INFO kernel.cc:864] Save model in resources
[INFO kernel.cc:929] Loading model from path
[INFO decision_forest.cc:590] Model loaded with 300 root(s), 1374910␣
 ↪node(s),
and 6 input feature(s).
[INFO abstract_model.cc:876] Engine "RandomForestGeneric" built
[INFO kernel.cc:797] Use fast generic engine
```

Se han creado trescientos árboles. El log nos muestra algunos de estos. Vemos que en general se obtiene un 76% de *accuracy*. Podemos obtener un resumen más reducido usando *summary*

[10]:
```
model_rf.summary()
```

```
Model: "random_forest_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Total params: 1
Trainable params: 0
Non-trainable params: 1
_________________________________________________________________
Type: "RANDOM_FOREST"
Task: CLASSIFICATION
Label: "__LABEL"

Input Features (6):
        bkc
        bkr
        wkc
        wkr
        wrc
        wrr

No weights

Variable Importance: NUM_NODES:
    1. "wrr" 216342.000000 ################
    2. "wrc" 165397.000000 ###########
    3. "bkr" 119478.000000 #######
    4. "bkc" 112101.000000 #######
    5. "wkc" 51255.000000 ##
    6. "wkr" 22732.000000

Variable Importance: NUM_AS_ROOT:
    1. "bkr" 205.000000 ################
    2. "wkr" 78.000000 #####
    3. "wkc" 12.000000
    4. "bkc"  5.000000
```

```
Variable Importance: SUM_SCORE:
    1. "wrr" 2938632.366295 ################
    2. "wrc" 2833222.359850 ##############
    3. "bkr" 2734362.605383 #############
    4. "bkc" 2527649.034721 ###########
    5. "wkr" 1702105.100839 ###
    6. "wkc" 1365919.120243

Variable Importance: MEAN_MIN_DEPTH:
    1. "__LABEL" 12.360672 ################
    2.     "wrr"  7.176756 #########
    3.     "wrc"  5.558875 ######
    4.     "wkc"  4.410273 #####
    5.     "wkr"  2.251511 ##
    6.     "bkc"  2.117759 ##
    7.     "bkr"  0.481818



Winner take all: true
Out-of-bag evaluation: accuracy:0.765702 logloss:0.803169
Number of trees: 300
Total number of nodes: 1374910

Number of nodes by tree:
Count: 300 Average: 4583.03 StdDev: 81.4853
Min: 4331 Max: 4813 Ignored: 0
----------------------------------------------
[ 4331, 4355)  1   0.33%   0.33%
[ 4355, 4379)  0   0.00%   0.33%
[ 4379, 4403)  1   0.33%   0.67%
[ 4403, 4427)  2   0.67%   1.33% #
[ 4427, 4451)  6   2.00%   3.33% ##
[ 4451, 4475) 14   4.67%   8.00% ####
[ 4475, 4500) 25   8.33%  16.33% ######
[ 4500, 4524) 27   9.00%  25.33% #######
[ 4524, 4548) 24   8.00%  33.33% ######
[ 4548, 4572) 40  13.33%  46.67% ##########
[ 4572, 4596) 38  12.67%  59.33% ##########
[ 4596, 4620) 29   9.67%  69.00% #######
[ 4620, 4644) 30  10.00%  79.00% ########
[ 4644, 4669) 18   6.00%  85.00% #####
[ 4669, 4693) 15   5.00%  90.00% ####
[ 4693, 4717) 12   4.00%  94.00% ###
[ 4717, 4741)  8   2.67%  96.67% ##
[ 4741, 4765)  4   1.33%  98.00% #
[ 4765, 4789)  1   0.33%  98.33%
[ 4789, 4813]  5   1.67% 100.00% #

Depth by leafs:
Count: 687605 Average: 12.3605 StdDev: 1.81321
Min: 5 Max: 15 Ignored: 0
----------------------------------------------
[  5,  6)     10   0.00%   0.00%
[  6,  7)    215   0.03%   0.03%
[  7,  8)   2159   0.31%   0.35%
[  8,  9)  10417   1.51%   1.86% #
[  9, 10)  31503   4.58%   6.44% ##
[ 10, 11)  68800  10.01%  16.45% #####
[ 11, 12) 108362  15.76%  32.21% ########
[ 12, 13) 129972  18.90%  51.11% ##########
[ 13, 14) 130129  18.92%  70.04% ##########
[ 14, 15) 106086  15.43%  85.46% ########
[ 15, 15]  99952  14.54% 100.00% ########

Number of training obs by leaf:
Count: 687605 Average: 9.79487 StdDev: 10.2582
Min: 5 Max: 257 Ignored: 0
----------------------------------------------
[   5,  17) 623807  90.72%  90.72% ##########
[  17,  30)  40317   5.86%  96.59% #
[  30,  42)  11791   1.71%  98.30%
[  42,  55)   5563   0.81%  99.11%
[  55,  68)   2509   0.36%  99.47%
[  68,  80)   1172   0.17%  99.64%
[  80,  93)    775   0.11%  99.76%
[  93, 106)    492   0.07%  99.83%
[ 106, 118)    269   0.04%  99.87%
[ 118, 131)    202   0.03%  99.90%
[ 131, 144)    173   0.03%  99.92%
[ 144, 156)    111   0.02%  99.94%
[ 156, 169)     73   0.01%  99.95%
[ 169, 182)     67   0.01%  99.96%
[ 182, 194)     65   0.01%  99.97%
[ 194, 207)    113   0.02%  99.98%
[ 207, 220)     62   0.01%  99.99%
[ 220, 232)     41   0.01% 100.00%
[ 232, 245)      2   0.00% 100.00%
[ 245, 257]      1   0.00% 100.00%
```

```
Attribute in nodes:
        216342 : wrr [NUMERICAL]
        165397 : wrc [CATEGORICAL]
        119478 : bkr [NUMERICAL]
        112101 : bkc [CATEGORICAL]
        51255 : wkc [CATEGORICAL]
        22732 : wkr [NUMERICAL]

Attribute in nodes with depth <= 0:
        205 : bkr [NUMERICAL]
        78 : wkr [NUMERICAL]
        12 : wkc [CATEGORICAL]
        5 : bkc [CATEGORICAL]

Attribute in nodes with depth <= 1:
        328 : bkr [NUMERICAL]
        225 : wkr [NUMERICAL]
        176 : bkc [CATEGORICAL]
        162 : wkc [CATEGORICAL]
        9 : wrr [NUMERICAL]

Attribute in nodes with depth <= 2:
        630 : bkc [CATEGORICAL]
        508 : wkr [NUMERICAL]
        455 : bkr [NUMERICAL]
        399 : wkc [CATEGORICAL]
        96 : wrr [NUMERICAL]
        12 : wrc [CATEGORICAL]

Attribute in nodes with depth <= 3:
        1211 : bkc [CATEGORICAL]
        982 : wkr [NUMERICAL]
        875 : wkc [CATEGORICAL]
        779 : bkr [NUMERICAL]
        411 : wrr [NUMERICAL]
        242 : wrc [CATEGORICAL]

Attribute in nodes with depth <= 5:
        4253 : wrc [CATEGORICAL]
        3484 : bkc [CATEGORICAL]
        3272 : bkr [NUMERICAL]
        2809 : wrr [NUMERICAL]
        2720 : wkc [CATEGORICAL]
        2352 : wkr [NUMERICAL]
```

```
Condition type in nodes:
        358552 : HigherCondition
        328753 : ContainsBitmapCondition
Condition type in nodes with depth <= 0:
        283 : HigherCondition
        17 : ContainsBitmapCondition
Condition type in nodes with depth <= 1:
        562 : HigherCondition
        338 : ContainsBitmapCondition
Condition type in nodes with depth <= 2:
        1059 : HigherCondition
        1041 : ContainsBitmapCondition
Condition type in nodes with depth <= 3:
        2328 : ContainsBitmapCondition
        2172 : HigherCondition
Condition type in nodes with depth <= 5:
        10457 : ContainsBitmapCondition
        8433 : HigherCondition

Training OOB:
        trees: 1, Out-of-bag evaluation: accuracy:0.663478 logloss:12.1295
        trees: 11, Out-of-bag evaluation: accuracy:0.691763 logloss:5.05263
        trees: 21, Out-of-bag evaluation: accuracy:0.730233 logloss:2.65216
        trees: 31, Out-of-bag evaluation: accuracy:0.742316 logloss:1.95862
        trees: 41, Out-of-bag evaluation: accuracy:0.750379 logloss:1.58524
        trees: 51, Out-of-bag evaluation: accuracy:0.753497 logloss:1.37915
        trees: 61, Out-of-bag evaluation: accuracy:0.755768 logloss:1.26819
        trees: 71, Out-of-bag evaluation: accuracy:0.758708 logloss:1.17384
        trees: 81, Out-of-bag evaluation: accuracy:0.75902 logloss:1.11116
        trees: 91, Out-of-bag evaluation: accuracy:0.760713 logloss:1.0598
        trees: 101, Out-of-bag evaluation: accuracy:0.761336 logloss:1.02395
        trees: 111, Out-of-bag evaluation: accuracy:0.764276 logloss:0.
 ↪992012
        trees: 121, Out-of-bag evaluation: accuracy:0.763964 logloss:0.
 ↪961497
        trees: 131, Out-of-bag evaluation: accuracy:0.763563 logloss:0.
 ↪944803
        trees: 141, Out-of-bag evaluation: accuracy:0.764009 logloss:0.
 ↪928943
        trees: 151, Out-of-bag evaluation: accuracy:0.764365 logloss:0.
 ↪913852
        trees: 161, Out-of-bag evaluation: accuracy:0.764811 logloss:0.89613
        trees: 171, Out-of-bag evaluation: accuracy:0.765479 logloss:0.
 ↪881226
```

```
        trees: 181, Out-of-bag evaluation: accuracy:0.764944 logloss:0.
↪866638
        trees: 191, Out-of-bag evaluation: accuracy:0.765345 logloss:0.
↪863178
        trees: 201, Out-of-bag evaluation: accuracy:0.764855 logloss:0.
↪848468
        trees: 211, Out-of-bag evaluation: accuracy:0.765345 logloss:0.
↪843659
        trees: 221, Out-of-bag evaluation: accuracy:0.766414 logloss:0.
↪837226
        trees: 231, Out-of-bag evaluation: accuracy:0.76588 logloss:0.834492
        trees: 241, Out-of-bag evaluation: accuracy:0.765924 logloss:0.
↪828863
        trees: 251, Out-of-bag evaluation: accuracy:0.766503 logloss:0.
↪822741
        trees: 261, Out-of-bag evaluation: accuracy:0.766102 logloss:0.
↪819977
        trees: 271, Out-of-bag evaluation: accuracy:0.766503 logloss:0.
↪815393
        trees: 281, Out-of-bag evaluation: accuracy:0.767261 logloss:0.
↪809227
        trees: 291, Out-of-bag evaluation: accuracy:0.767305 logloss:0.
↪804649
        trees: 300, Out-of-bag evaluation: accuracy:0.765702 logloss:0.
↪803169
```

También recibimos información de las variables más importantes, que coinciden con el resultado de EDA (lógico ya que usa randomForest). También podemos ver el número de nodos y de hojas.

Procedemos a probar el árbol con el conjunto test:

[11]:
```python
evaluation = model_rf.evaluate(test, return_dict=True)
print()

for name, value in evaluation.items():
    print(f"{name}: {value:.4f}")
```

```
88/88 [==============================] - 1s 6ms/step - loss: 0.0000e+00 -
accuracy: 0.7683

loss: 0.0000
accuracy: 0.7683
```

Obtenemos las predicciones para poder utilizar distintas métricas.

```
[56]: preds = model_rf.predict(test)
```

```
[57]: y_test_aux = test_aux["opt rank"]
```

```
[14]: from sklearn.metrics import confusion_matrix, precision_score, \
      f1_score, cohen_kappa_score, recall_score

      def compute_metrics_multiclass(y_test, y_pred):
          results=[]
          results.append(precision_score(y_test, np.round(y_pred),␣
       ↪average="micro"))
          results.append(recall_score(y_test, np.round(y_pred),␣
       ↪average="micro"))
          results.append(f1_score(y_test, np.round(y_pred), average="micro"))
          results.append(cohen_kappa_score(y_test, np.round(y_pred)))
          return results
```

Utilizamos la función argmax para poder obtener la clase predicha en cada caso.

```
[58]: preds = np.argmax(preds, axis=1)

      metrics_rf = compute_metrics_multiclass(y_test_aux, preds)
      metrics_rf
```

```
[58]: [0.7682839814484481, 0.7682839814484481, 0.768283981448448, 0.
       ↪7408866826454885]
```

Los resultados oscilan el 76% en todas las métricas. No son mejores que los obtenidos por las redes neuronales usando datos por defecto.

Podemos realizar una visualización del árbol promedio de los 300, aunque, al ser multiclase, no nos es fácil interpretarlo.

```
[16]: tfdf.model_plotter.plot_model_in_colab(model_rf, tree_idx=0, max_depth=3)
```

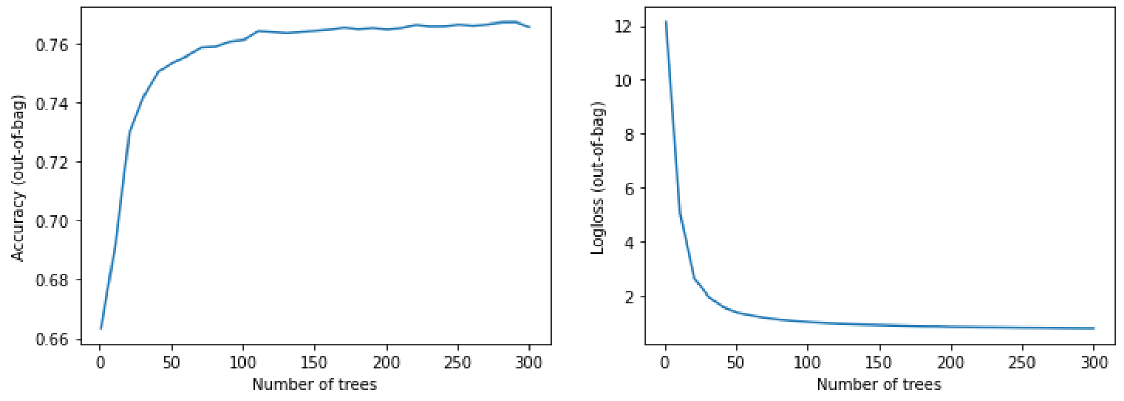
```
[16]: <IPython.core.display.HTML object>
```

Podemos ver cómo ha evolucionado el entrenamiento del modelo:

```
[17]: import matplotlib.pyplot as plt

      def make_figure(model):
          logs = model.make_inspector().training_logs()

          plt.figure(figsize=(12, 4))
```

```
    plt.subplot(1, 2, 1)
    plt.plot([log.num_trees for log in logs], [log.evaluation.accuracy␣
↪for log in logs])
    plt.xlabel("Number of trees")
    plt.ylabel("Accuracy (out-of-bag)")

    plt.subplot(1, 2, 2)
    plt.plot([log.num_trees for log in logs], [log.evaluation.loss for␣
↪log in logs])
    plt.xlabel("Number of trees")
    plt.ylabel("Logloss (out-of-bag)")

    plt.show()
```

[18]:
```
make_figure(model_rf)
```



4.2 Random Forest con SMOTE

Repetimos el procedimiento con los datos SMOTE. Esperamos obtener un mal rendimiento, al igual que en redes neuronales.

[19]:
```
from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state=2)
```

Convertimos los datos de las columnas categóricas a números para poder usar *SMOTE*:

[20]:
```
cat_columns = ['wkc', 'wrc', 'bkc']

data[cat_columns]=data[cat_columns].astype("category")
data[cat_columns]=data[cat_columns].apply(lambda x: x.cat.codes)
data.astype('int64')
```

[20]:
```
       wkc  wkr  wrc  wrr  bkc  bkr  opt rank
0        0    1    1    3    2    2         0
1        0    1    2    1    2    2         0
2        0    1    2    1    3    1         0
3        0    1    2    1    3    2         0
4        0    1    2    2    2    1         0
…     …   …   …   …   …   …          …
28051    1    1    6    7    4    5        17
28052    1    1    6    7    4    6        17
28053    1    1    6    7    4    7        17
28054    1    1    6    7    5    5        17
28055    1    1    6    7    6    5        17

[28056 rows x 7 columns]
```

Aplicamos *SMOTE*:

[21]:
```python
xsmote, ysmote = sm.fit_resample(data.drop(label, axis=1), data[label])
data_smote = pd.concat([xsmote, ysmote], axis=1)
data_smote
```

[21]:
```
       wkc  wkr  wrc  wrr  bkc  bkr  opt rank
0        0    1    1    3    2    2         0
1        0    1    2    1    2    2         0
2        0    1    2    1    3    1         0
3        0    1    2    1    3    2         0
4        0    1    2    2    2    1         0
…     …   …   …   …   …   …          …
81949    0    1    5    7    3    2        17
81950    0    1    1    2    4    3        17
81951    0    1    6    6    4    3        17
81952    0    1    7    6    5    3        17
81953    0    1    3    6    5    3        17

[81954 rows x 7 columns]
```

[22]:
```python
train_smote, test_smote = split_dataset(data_smote)
print("{} examples in training, {} examples for testing.".format(
    len(train_smote), len(test_smote)))
```

```
65416 examples in training, 16538 examples for testing.
```

Creamos los datasets train y test para que pueda usarlos la librería. Notemos que los datos de las variables numéricas que hemos convertido se asignan a int8 automáticamente, a diferencia de las demás, que usan int64.

[23]:
```
train_smote = tfdf.keras.pd_dataframe_to_tf_dataset(train_smote,␣
↪label=label, )
test_smote = tfdf.keras.pd_dataframe_to_tf_dataset(test_smote,␣
↪label=label)
```

[24]:
```
train_smote
```

[24]:
```
<BatchDataset shapes: ({wkc: (None,), wkr: (None,), wrc: (None,), wrr:␣
↪(None,),
bkc: (None,), bkr: (None,)}, (None,)), types: ({wkc: tf.int8, wkr: tf.
↪int64,
wrc: tf.int8, wrr: tf.int64, bkc: tf.int8, bkr: tf.int64}, tf.int64)>
```

[25]:
```
model_1_smote = tfdf.keras.RandomForestModel()

model_1_smote.compile(
                metrics=["accuracy"])

model_1_smote.fit(x=train_smote)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last
<ipython-input-25-18e6e4ad8e58> in <module>
      4                 metrics=["accuracy"])
      5
----> 6 model_1_smote.fit(x=train_smote)

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/
 ↪tensorflow_decision_forests/keras/core.py in fit(self, x, y, callbacks,
 ↪**kwargs)
    769
    770     try:
--> 771       history = super(CoreModel, self).fit(

    772           x=x, y=y, epochs=1, callbacks=callbacks, **kwargs)
    773     finally:

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/keras
 ↪engine/training.py in fit(self, x, y, batch_size, epochs, verbose,␣
 ↪callbacks, validation_split, validation_data, shuffle, class_weight,␣
 ↪sample_weight, initial_epoch, steps_per_epoch, validation_steps,␣
 ↪validation_batch_size, validation_freq, max_queue_size, workers,␣
 ↪use_multiprocessing)
   1181                 _r=1):
   1182               callbacks.on_train_batch_begin(step)
-> 1183               tmp_logs = self.train_function(iterator)
```

```
   1184                 if data_handler.should_sync:
   1185                   context.async_wait()

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/eager/
 ↪def_function.py in __call__(self, *args, **kwds)
    887
    888       with OptionalXlaContext(self._jit_compile):
--> 889         result = self._call(*args, **kwds)
    890
    891       new_tracing_count = self.experimental_get_tracing_count()

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/eager/
 ↪def_function.py in _call(self, *args, **kwds)
    931       # This is the first call of __call__, so we have to␣
 ↪initialize.
    932       initializers = []
--> 933       self._initialize(args, kwds, add_initializers_to=initializers)
    934     finally:
    935       # At this point we know that the initialization is complete␣
 ↪(or less

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/eager/
 ↪def_function.py in _initialize(self, args, kwds, add_initializers_to)
    761     self._graph_deleter = FunctionDeleter(self.
 ↪_lifted_initializer_graph)
    762     self._concrete_stateful_fn = (
--> 763         self._stateful_fn.
 ↪_get_concrete_function_internal_garbage_collected(  # pylint:␣
 ↪disable=protected-access

    764             *args, **kwds))
    765

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/eager/
 ↪function.py in _get_concrete_function_internal_garbage_collected(self,␣
 ↪*args, **kwargs)
   3048       args, kwargs = None, None
   3049     with self._lock:
-> 3050       graph_function, _ = self._maybe_define_function(args, kwargs
   3051     return graph_function
   3052

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/eager/
 ↪function.py in _maybe_define_function(self, args, kwargs)
   3442
   3443           self._function_cache.missed.add(call_context_key)
```

```
-> 3444            graph_function = self._create_graph_function(args, kwargs)
   3445            self._function_cache.primary[cache_key] = graph_functio
   3446

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/eager/
 ↪function.py in _create_graph_function(self, args, kwargs,␣
 ↪override_flat_arg_shapes)
   3277       arg_names = base_arg_names + missing_arg_names
   3278       graph_function = ConcreteFunction(
-> 3279           func_graph_module.func_graph_from_py_func(
   3280               self._name,
   3281               self._python_function,

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/
 ↪framework/func_graph.py in func_graph_from_py_func(name, python_func,␣
 ↪args, kwargs, signature, func_graph, autograph, autograph_options,␣
 ↪add_control_dependencies, arg_names, op_return_value, collections,␣
 ↪capture_by_value, override_flat_arg_shapes)
    997         _, original_func = tf_decorator.unwrap(python_func)
    998
--> 999       func_outputs = python_func(*func_args, **func_kwargs)
   1000
   1001       # invariant: `func_outputs` contains only Tensors,␣
 ↪CompositeTensors,

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/eager/
 ↪def_function.py in wrapped_fn(*args, **kwds)
    670         # the function a weak reference to itself to avoid a␣
 ↪reference cycle.
    671         with OptionalXlaContext(compile_with_xla):
--> 672           out = weak_wrapped_fn().__wrapped__(*args, **kwds)
    673         return out
    674

~/anaconda3/envs/tf2.5/lib/python3.8/site-packages/tensorflow/python/
 ↪framework/func_graph.py in wrapper(*args, **kwargs)
    984           except Exception as e:  # pylint:disable=broad-except
    985             if hasattr(e, "ag_error_metadata"):
--> 986               raise e.ag_error_metadata.to_exception(e)
    987             else:
    988               raise

ValueError: in user code:
    /home/antonio/anaconda3/envs/tf2.5/lib/python3.8/site-packages/
  ↪tensorflow/python/keras/engine/training.py:855 train_function  *
        return step_function(self, iterator)
    /home/antonio/anaconda3/envs/tf2.5/lib/python3.8/site-packages/
  ↪tensorflow_decision_forests/keras/core.py:646 train_step  *
        normalized_semantic_inputs = tf_core.
  ↪normalize_inputs(semantic_inputs)
    /home/antonio/anaconda3/envs/tf2.5/lib/python3.8/site-packages/
  ↪tensorflow_decision_forests/tensorflow/core.py:255 normalize_inputs  *
        raise ValueError(

    ValueError: Non supported tensor dtype <dtype: 'int8'> for semantic␣
  ↪Semantic.CATEGORICAL of feature wkc
```

Sorpresa. No somos capaces de crear el modelo ya que las variables deben estar codificadas en tf.int64. Pero al usar la función para crear el dataset, como hemos comentado antes, no es posible cambiar el tipo. Aunque definamos inicialmente todas las columnas como int64, la función las cambia a tf.int8.

Este bug nos impide comprobar el rendimiento de SMOTE. Suponemos que se solucionará en versiones siguientes. El método con *one-hot* encoding tampoco funcionará ya que también es convertido a tf.int8.

4.3 Datos raw con boosted trees

El siguiente algoritmo que vamos a probar es *Gradient Boosted Trees*. En estos modelos, se crean un conjunto de árboles los cuales son mejorados iterativamente mediante *boosting*. A diferencia de *AdaBoost*, se mejoran las predicciones reduciendo el error con una función de coste.

Este algoritmo permite modificar ciertos parámetros, por los que realizaremos distintas pruebas y ver su rendimiento.

[26]:
```
model_bt_1 = tfdf.keras.GradientBoostedTreesModel(num_trees=500,
                                                   ␣
↪growing_strategy="BEST_FIRST_GLOBAL",
                                                   max_depth=8)
model_bt_1.fit(x=train)
model_bt_1.summary()
```

```
351/351 [==============================] - 0s 1ms/step
Model: "gradient_boosted_trees_model"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Total params: 1
Trainable params: 0
Non-trainable params: 1
_________________________________________________________________
Type: "GRADIENT_BOOSTED_TREES"
Task: CLASSIFICATION
Label: "__LABEL"

Input Features (6):
        bkc
        bkr
        wkc
        wkr
        wrc
        wrr

No weights

Variable Importance: NUM_NODES:
    1. "wrr" 50164.000000 ################
    2. "wrc" 46244.000000 #############
    3. "bkc" 39981.000000 ##########
    4. "bkr" 37646.000000 #########
    5. "wkr" 27136.000000 ###
    6. "wkc" 21299.000000

Variable Importance: NUM_AS_ROOT:
    1. "bkr" 2894.000000 ################
    2. "bkc" 1723.000000 ########
    3. "wkc" 980.000000 ###
    4. "wkr" 932.000000 ###
    5. "wrr" 499.000000
    6. "wrc" 388.000000

Variable Importance: SUM_SCORE:
    1. "bkr" 9447.080979 ################
    2. "wrr" 9369.968841 ###############
    3. "bkc" 9339.258444 ###############
    4. "wrc" 8868.002580 #############
    5. "wkr" 6195.950017 ####
    6. "wkc" 4883.574089

Variable Importance: MEAN_MIN_DEPTH:
    1. "__LABEL"  6.596429 ################
    2.     "wkr"  3.999949 #######
    3.     "wkc"  3.672179 ######
    4.     "wrr"  3.451561 #####
    5.     "wrc"  3.013544 ####
    6.     "bkc"  2.044103
    7.     "bkr"  1.752188



Loss: MULTINOMIAL_LOG_LIKELIHOOD
Validation loss value: 0.398818
Number of trees per iteration: 18
Number of trees: 7416
Total number of nodes: 452356

Number of nodes by tree:
Count: 7416 Average: 60.9973 StdDev: 0.232229
Min: 41 Max: 61 Ignored: 0
----------------------------------------------
[ 41, 42)    1   0.01%   0.01%
[ 42, 43)    0   0.00%   0.01%
[ 43, 44)    0   0.00%   0.01%
[ 44, 45)    0   0.00%   0.01%
[ 45, 46)    0   0.00%   0.01%
[ 46, 47)    0   0.00%   0.01%
[ 47, 48)    0   0.00%   0.01%
[ 48, 49)    0   0.00%   0.01%
[ 49, 50)    0   0.00%   0.01%
[ 50, 51)    0   0.00%   0.01%
[ 51, 52)    0   0.00%   0.01%
[ 52, 53)    0   0.00%   0.01%
[ 53, 54)    0   0.00%   0.01%
[ 54, 55)    0   0.00%   0.01%
[ 55, 56)    0   0.00%   0.01%
[ 56, 57)    0   0.00%   0.01%
[ 57, 58)    0   0.00%   0.01%
[ 58, 59)    0   0.00%   0.01%
[ 59, 60)    0   0.00%   0.01%
[ 60, 61] 7415  99.99% 100.00% ##########

Depth by leafs:
Count: 229886 Average: 6.59651 StdDev: 1.8032
Min: 1 Max: 8 Ignored: 0
----------------------------------------------
[ 1, 2)   4180   1.82%   1.82%
[ 2, 3)   6308   2.74%   4.56% #
[ 3, 4)   8914   3.88%   8.44% #
[ 4, 5)  13219   5.75%  14.19% #
[ 5, 6)  19659   8.55%  22.74% ##
[ 6, 7)  29002  12.62%  35.36% ###
[ 7, 8)  41108  17.88%  53.24% ####
[ 8, 8] 107496  46.76% 100.00% ##########

Number of training obs by leaf:
Count: 229886 Average: 0 StdDev: 0
Min: 0 Max: 0 Ignored: 0
----------------------------------------------
[ 0, 0] 229886 100.00% 100.00% ##########

Attribute in nodes:
        50164 : wrr [NUMERICAL]
        46244 : wrc [CATEGORICAL]
        39981 : bkc [CATEGORICAL]
        37646 : bkr [NUMERICAL]
        27136 : wkr [NUMERICAL]
        21299 : wkc [CATEGORICAL]

Attribute in nodes with depth <= 0:
        2894 : bkr [NUMERICAL]
        1723 : bkc [CATEGORICAL]
        980 : wkc [CATEGORICAL]
        932 : wkr [NUMERICAL]
        499 : wrr [NUMERICAL]
        388 : wrc [CATEGORICAL]

Attribute in nodes with depth <= 1:
        5303 : bkr [NUMERICAL]
        4945 : bkc [CATEGORICAL]
        2146 : wrc [CATEGORICAL]
        2073 : wkc [CATEGORICAL]
        1903 : wrr [NUMERICAL]
        1698 : wkr [NUMERICAL]

Attribute in nodes with depth <= 2:
        8155 : bkc [CATEGORICAL]
        7709 : bkr [NUMERICAL]
        5435 : wrc [CATEGORICAL]
        4617 : wrr [NUMERICAL]
        4162 : wkc [CATEGORICAL]
        2986 : wkr [NUMERICAL]
```

```
Attribute in nodes with depth <= 3:
        12463 : bkc [CATEGORICAL]
        11071 : bkr [NUMERICAL]
        9659 : wrc [CATEGORICAL]
        9207 : wrr [NUMERICAL]
        6393 : wkc [CATEGORICAL]
        5349 : wkr [NUMERICAL]

Attribute in nodes with depth <= 5:
        26050 : wrr [NUMERICAL]
        24139 : wrc [CATEGORICAL]
        23218 : bkc [CATEGORICAL]
        20910 : bkr [NUMERICAL]
        13953 : wkr [NUMERICAL]
        13024 : wkc [CATEGORICAL]

Condition type in nodes:
        114946 : HigherCondition
        107524 : ContainsBitmapCondition
Condition type in nodes with depth <= 0:
        4325 : HigherCondition
        3091 : ContainsBitmapCondition
Condition type in nodes with depth <= 1:
        9164 : ContainsBitmapCondition
        8904 : HigherCondition
Condition type in nodes with depth <= 2:
        17752 : ContainsBitmapCondition
        15312 : HigherCondition
Condition type in nodes with depth <= 3:
        28515 : ContainsBitmapCondition
        25627 : HigherCondition
Condition type in nodes with depth <= 5:
        60913 : HigherCondition
        60381 : ContainsBitmapCondition
```

Hemos creado y entrenado un modelo con 300 árboles y una profundidad máxima de 8. Veamos las métricas obtenidas:

[28]:
```
preds = model_bt_1.predict(test)
preds = np.argmax(preds, axis=1)

metrics_bt_1= compute_metrics_multiclass(y_test_aux, preds)
metrics_bt_1
```

[28]:
```
[0.8913663931501962,
 0.8913663931501962,
 0.8913663931501962,
 0.8785774841048029]
```

Los resultados son mucho mejores. Obtenemos un 89% en Precision, Recall y F1. Hemos alcanzado los mejores resultados de las redes neuronales en el caso de dropout y 6 capas internas de 250 neuronas con un modelo mucho más rápido y simple de construir. Recordemos que los resultados son deterministas y que sólo es necesario una "época" a diferencia de las redes neuronales.

Creamos otro modelo en el que aportamos más parámetros para el entrenamiento, los cuales se recomiendan en la documentación y tutoriales de la librería.

[29]:
```
model_bt_2 = tfdf.keras.GradientBoostedTreesModel(
    num_trees=500,
    growing_strategy="BEST_FIRST_GLOBAL",
    max_depth=8,
    split_axis="SPARSE_OBLIQUE",
    categorical_algorithm="RANDOM",
    )
model_bt_2.fit(x=train)
model_bt_2.summary()
```

```
351/351 [==============================] - 0s 1ms/step
Model: "gradient_boosted_trees_model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Total params: 1
Trainable params: 0
Non-trainable params: 1
_________________________________________________________________
Type: "GRADIENT_BOOSTED_TREES"
Task: CLASSIFICATION
Label: "__LABEL"

Input Features (6):
        bkc
        bkr
        wkc
        wkr
        wrc
        wrr
```

```
No weights

Variable Importance: NUM_NODES:
    1. "bkr" 80207.000000 ################
    2. "wrc" 44066.000000 #######
    3. "bkc" 39708.000000 ######
    4. "wkr" 24787.000000 ###
    5. "wkc" 19797.000000 ##
    6. "wrr" 9051.000000

Variable Importance: NUM_AS_ROOT:
    1. "bkr" 3933.000000 ################
    2. "bkc" 1479.000000 #####
    3. "wkc" 933.000000 ###
    4. "wkr" 606.000000 ##
    5. "wrc" 251.000000
    6. "wrr" 52.000000

Variable Importance: SUM_SCORE:
    1. "bkr" 16701.633258 ################
    2. "bkc" 10920.684873 #########
    3. "wrc" 9608.707738 ########
    4. "wkr" 6900.316011 #####
    5. "wkc" 4510.972891 ##
    6. "wrr" 1757.327341

Variable Importance: MEAN_MIN_DEPTH:
    1. "__LABEL"  6.491117 ################
    2.     "wrr"  5.667415 #############
    3.     "wkr"  4.415438 #########
    4.     "wkc"  3.703789 #######
    5.     "wrc"  2.917051 #####
    6.     "bkc"  2.008529 ##
    7.     "bkr"  1.172906



Loss: MULTINOMIAL_LOG_LIKELIHOOD
Validation loss value: 0.37488
Number of trees per iteration: 18
Number of trees: 7254
Total number of nodes: 442486

Number of nodes by tree:
Count: 7254 Average: 60.9989 StdDev: 0.0939229
```

```
Min: 53 Max: 61 Ignored: 0
----------------------------------------------
[ 53, 54)    1   0.01%   0.01%
[ 54, 55)    0   0.00%   0.01%
[ 55, 56)    0   0.00%   0.01%
[ 56, 57)    0   0.00%   0.01%
[ 57, 58)    0   0.00%   0.01%
[ 58, 59)    0   0.00%   0.01%
[ 59, 60)    0   0.00%   0.01%
[ 60, 61)    0   0.00%   0.01%
[ 61, 61] 7253  99.99% 100.00% ##########

Depth by leafs:
Count: 224870 Average: 6.49114 StdDev: 1.77628
Min: 1 Max: 8 Ignored: 0
----------------------------------------------
[ 1, 2)  3757   1.67%   1.67%
[ 2, 3)  5718   2.54%   4.21% #
[ 3, 4)  9162   4.07%   8.29% #
[ 4, 5) 14169   6.30%  14.59% ##
[ 5, 6) 22241   9.89%  24.48% ##
[ 6, 7) 33055  14.70%  39.18% ####
[ 7, 8) 43372  19.29%  58.47% #####
[ 8, 8] 93396  41.53% 100.00% ##########

Number of training obs by leaf:
Count: 224870 Average: 0 StdDev: 0
Min: 0 Max: 0 Ignored: 0
----------------------------------------------
[ 0, 0] 224870 100.00% 100.00% ##########

Attribute in nodes:
        80207 : bkr [NUMERICAL]
        44066 : wrc [CATEGORICAL]
        39708 : bkc [CATEGORICAL]
        24787 : wkr [NUMERICAL]
        19797 : wkc [CATEGORICAL]
        9051 : wrr [NUMERICAL]

Attribute in nodes with depth <= 0:
        3933 : bkr [NUMERICAL]
        1479 : bkc [CATEGORICAL]
        933 : wkc [CATEGORICAL]
        606 : wkr [NUMERICAL]
        251 : wrc [CATEGORICAL]
        52 : wrr [NUMERICAL]

Attribute in nodes with depth <= 1:
        7577 : bkr [NUMERICAL]
        4510 : bkc [CATEGORICAL]
        2157 : wrc [CATEGORICAL]
        1997 : wkc [CATEGORICAL]
        1328 : wkr [NUMERICAL]
        436 : wrr [NUMERICAL]

Attribute in nodes with depth <= 2:
        12508 : bkr [NUMERICAL]
        7941 : bkc [CATEGORICAL]
        5815 : wrc [CATEGORICAL]
        3821 : wkc [CATEGORICAL]
        2771 : wkr [NUMERICAL]
        933 : wrr [NUMERICAL]

Attribute in nodes with depth <= 3:
        19810 : bkr [NUMERICAL]
        12619 : bkc [CATEGORICAL]
        10928 : wrc [CATEGORICAL]
        5966 : wkc [CATEGORICAL]
        5059 : wkr [NUMERICAL]
        1813 : wrr [NUMERICAL]

Attribute in nodes with depth <= 5:
        45160 : bkr [NUMERICAL]
        25541 : wrc [CATEGORICAL]
        24738 : bkc [CATEGORICAL]
        13406 : wkr [NUMERICAL]
        12158 : wkc [CATEGORICAL]
        4880 : wrr [NUMERICAL]

Condition type in nodes:
        114045 : ObliqueCondition
        103571 : ContainsBitmapCondition
Condition type in nodes with depth <= 0:
        4591 : ObliqueCondition
        2663 : ContainsBitmapCondition
Condition type in nodes with depth <= 1:
        9341 : ObliqueCondition
        8664 : ContainsBitmapCondition
Condition type in nodes with depth <= 2:
        17577 : ContainsBitmapCondition
        16212 : ObliqueCondition
Condition type in nodes with depth <= 3:
        29513 : ContainsBitmapCondition
        26682 : ObliqueCondition
Condition type in nodes with depth <= 5:
        63446 : ObliqueCondition
        62437 : ContainsBitmapCondition
```

[30]:
```
preds = model_bt_2.predict(test)
preds = np.argmax(preds, axis=1)

metrics_bt_2= compute_metrics_multiclass(y_test_aux, preds)
metrics_bt_2
```

[30]:
```
[0.881198715661791, 0.881198715661791, 0.881198715661791, 0.
 ↪8672254847683257]
```

Los resultados siguen siendo muy buenos pero empeoran levemente con respecto a los parámetros por defecto.

La librería implementa algunas plantillas de parámetros que, según la documentación, proveen mejores resultados que los parámetros por defecto.

Estos son:

[31]:
```
print(tfdf.keras.GradientBoostedTreesModel.predefined_hyperparameters())
```

```
[HyperParameterTemplate(name='better_default', version=1,
parameters={'growing_strategy': 'BEST_FIRST_GLOBAL'}, description='A
configuration that is generally better than the default parameters without␣
 ↪being
more expensive.'), HyperParameterTemplate(name='benchmark_rank1', version=1,
parameters={'growing_strategy': 'BEST_FIRST_GLOBAL',␣
 ↪'categorical_algorithm':
'RANDOM', 'split_axis': 'SPARSE_OBLIQUE', 'sparse_oblique_normalization':
'MIN_MAX', 'sparse_oblique_num_projections_exponent': 1.0}, description='Top
ranking hyper-parameters on our benchmark slightly modified to run in␣
 ↪reasonable
time.')]
```

Podemos probar "benchmark_rank1", que ofreció los mejores resultados en sus benchmarks.

[32]:
```
model_bt_3 = tfdf.keras.GradientBoostedTreesModel(
    hyperparameter_template="benchmark_rank1")

model_bt_3.fit(x=train)
```

```
model_bt_3.summary()
```

```
351/351 [==============================] - 0s 1ms/step
Model: "gradient_boosted_trees_model_2"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
Total params: 1
Trainable params: 0
Non-trainable params: 1
_________________________________________________________________
Type: "GRADIENT_BOOSTED_TREES"
Task: CLASSIFICATION
Label: "__LABEL"

Input Features (6):
        bkc
        bkr
        wkc
        wkr
        wrc
        wrr

No weights

Variable Importance: NUM_NODES:
    1. "bkr" 59503.000000 ################
    2. "wrc" 34820.000000 ########
    3. "bkc" 33116.000000 ########
    4. "wkc" 14859.000000 ##
    5. "wkr" 14090.000000 ##
    6. "wrr" 5583.000000

Variable Importance: NUM_AS_ROOT:
    1. "bkr" 3037.000000 ################
    2. "bkc" 958.000000 ####
    3. "wkc" 830.000000 ####
    4. "wkr" 418.000000 #
    5. "wrc" 90.000000
    6. "wrr" 67.000000

Variable Importance: SUM_SCORE:
    1. "bkr" 16677.602664 ################
    2. "bkc" 9835.501402 ########
    3. "wrc" 7970.335011 #######
    4. "wkr" 4682.182032 ###
    5. "wkc" 4022.472972 ###
    6. "wrr" 1099.078474

Variable Importance: MEAN_MIN_DEPTH:
    1. "__LABEL"  5.490134 ################
    2.     "wrr"  4.978168 ##############
    3.     "wkr"  4.237620 ###########
    4.     "wkc"  3.482901 ########
    5.     "wrc"  3.052854 #######
    6.     "bkc"  2.224792 ####
    7.     "bkr"  0.950627



Loss: MULTINOMIAL_LOG_LIKELIHOOD
Validation loss value: 0.441028
Number of trees per iteration: 18
Number of trees: 5400
Total number of nodes: 329342

Number of nodes by tree:
Count: 5400 Average: 60.9893 StdDev: 0.332047
Min: 45 Max: 61 Ignored: 0
----------------------------------------------
[ 45, 46)    1   0.02%   0.02%
[ 46, 47)    0   0.00%   0.02%
[ 47, 48)    0   0.00%   0.02%
[ 48, 49)    0   0.00%   0.02%
[ 49, 50)    1   0.02%   0.04%
[ 50, 51)    0   0.00%   0.04%
[ 51, 52)    1   0.02%   0.06%
[ 52, 53)    0   0.00%   0.06%
[ 53, 54)    0   0.00%   0.06%
[ 54, 55)    0   0.00%   0.06%
[ 55, 56)    2   0.04%   0.09%
[ 56, 57)    0   0.00%   0.09%
[ 57, 58)    1   0.02%   0.11%
[ 58, 59)    0   0.00%   0.11%
[ 59, 60)    2   0.04%   0.15%
[ 60, 61)    0   0.00%   0.15%
[ 61, 61] 5392  99.85% 100.00% ##########

Depth by leafs:
```

```
Count: 167371 Average: 5.49019 StdDev: 0.978449
Min: 1 Max: 6 Ignored: 0
----------------------------------------------
[ 1, 2)    597   0.36%   0.36%
[ 2, 3)   3533   2.11%   2.47%
[ 3, 4)   6480   3.87%   6.34% #
[ 4, 5)  12596   7.53%  13.87% #
[ 5, 6)  23579  14.09%  27.95% ##
[ 6, 6] 120586  72.05% 100.00% ##########

Number of training obs by leaf:
Count: 167371 Average: 0 StdDev: 0
Min: 0 Max: 0 Ignored: 0
----------------------------------------------
[ 0, 0] 167371 100.00% 100.00% ##########

Attribute in nodes:
        59503 : bkr [NUMERICAL]
        34820 : wrc [CATEGORICAL]
        33116 : bkc [CATEGORICAL]
        14859 : wkc [CATEGORICAL]
        14090 : wkr [NUMERICAL]
        5583 : wrr [NUMERICAL]

Attribute in nodes with depth <= 0:
        3037 : bkr [NUMERICAL]
        958 : bkc [CATEGORICAL]
        830 : wkc [CATEGORICAL]
        418 : wkr [NUMERICAL]
        90 : wrc [CATEGORICAL]
        67 : wrr [NUMERICAL]

Attribute in nodes with depth <= 1:
        7210 : bkr [NUMERICAL]
        3307 : bkc [CATEGORICAL]
        1801 : wkc [CATEGORICAL]
        1770 : wrc [CATEGORICAL]
        1108 : wkr [NUMERICAL]
        407 : wrr [NUMERICAL]

Attribute in nodes with depth <= 2:
        13003 : bkr [NUMERICAL]
        7191 : bkc [CATEGORICAL]
        5527 : wrc [CATEGORICAL]
        3460 : wkc [CATEGORICAL]
        2379 : wkr [NUMERICAL]
        916 : wrr [NUMERICAL]

Attribute in nodes with depth <= 3:
        22026 : bkr [NUMERICAL]
        13209 : bkc [CATEGORICAL]
        12257 : wrc [CATEGORICAL]
        5894 : wkc [CATEGORICAL]
        4529 : wkr [NUMERICAL]
        1827 : wrr [NUMERICAL]

Attribute in nodes with depth <= 5:
        59503 : bkr [NUMERICAL]
        34820 : wrc [CATEGORICAL]
        33116 : bkc [CATEGORICAL]
        14859 : wkc [CATEGORICAL]
        14090 : wkr [NUMERICAL]
        5583 : wrr [NUMERICAL]

Condition type in nodes:
        82795 : ContainsBitmapCondition
        79176 : ObliqueCondition
Condition type in nodes with depth <= 0:
        3522 : ObliqueCondition
        1878 : ContainsBitmapCondition
Condition type in nodes with depth <= 1:
        8725 : ObliqueCondition
        6878 : ContainsBitmapCondition
Condition type in nodes with depth <= 2:
        16298 : ObliqueCondition
        16178 : ContainsBitmapCondition
Condition type in nodes with depth <= 3:
        31360 : ContainsBitmapCondition
        28382 : ObliqueCondition
Condition type in nodes with depth <= 5:
        82795 : ContainsBitmapCondition
        79176 : ObliqueCondition
```

[33]:
```
preds = model_bt_3.predict(test)
preds = np.argmax(preds, axis=1)

metrics_bt_3= compute_metrics_multiclass(y_test_aux, preds)
metrics_bt_3
```

[33]:
```
[0.8555119514805566,
 0.8555119514805566,
 0.8555119514805566,
 0.8384821966216377]
```

Obtenemos los peores resultados. Es curioso que la mejor configuración de parámetros en las pruebas de Google haya producido estas métricas.

4.4 Creación de árboles con modelos CART con datos *raw*

Probamos ahora la función *CartModel*, variación del famoso C4.5. Este tipo de clasificadores han aparecido en otras asignaturas del máster y provee muy buenos resultados.

[35]:
```
model_CART_1 = tfdf.keras.CartModel()

model_CART_1.fit(x=train)
model_CART_1.summary()
```

```
351/351 [==============================] - 0s 1ms/step
Model: "cart_model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Total params: 1
Trainable params: 0
Non-trainable params: 1
_________________________________________________________________
Type: "RANDOM_FOREST"
Task: CLASSIFICATION
Label: "__LABEL"

Input Features (6):
        bkc
        bkr
        wkc
        wkr
        wrc
        wrr

No weights

Variable Importance: NUM_NODES:
    1. "wrr" 267.000000 ################
    2. "wrc" 202.000000 ###########
    3. "bkr" 128.000000 #####
    4. "bkc" 113.000000 ####
    5. "wkc" 79.000000 ##
    6. "wkr" 45.000000

Variable Importance: NUM_AS_ROOT:
    1. "bkr"  1.000000

Variable Importance: SUM_SCORE:
    1. "bkr" 6537.829253 ################
    2. "wrc" 6219.915177 ##############
    3. "wrr" 6025.801832 #############
    4. "bkc" 5914.132091 ############
    5. "wkr" 4833.035964 ######
    6. "wkc" 3554.173828

Variable Importance: MEAN_MIN_DEPTH:
    1. "__LABEL" 10.990419 ################
    2.     "wrr"  6.985629 ##########
    3.     "wrc"  5.141317 #######
    4.     "wkc"  5.123353 #######
    5.     "wkr"  3.002395 ####
    6.     "bkc"  1.723353 ##
    7.     "bkr"  0.000000



Winner take all: false
Out-of-bag evaluation disabled.
Number of trees: 1
Total number of nodes: 1669

Number of nodes by tree:
Count: 1 Average: 1669 StdDev: 0
Min: 1669 Max: 1669 Ignored: 0
----------------------------------------------
[ 1669, 1669] 1 100.00% 100.00% ##########

Depth by leafs:
Count: 835 Average: 10.9904 StdDev: 1.98072
Min: 5 Max: 15 Ignored: 0
----------------------------------------------
[  5,  6)   1   0.12%   0.12%
[  6,  7)   6   0.72%   0.84%
[  7,  8)  23   2.75%   3.59% #
[  8,  9)  50   5.99%   9.58% ###
[  9, 10) 112  13.41%  22.99% ######
[ 10, 11) 154  18.44%  41.44% #########
[ 11, 12) 173  20.72%  62.16% ##########
[ 12, 13) 125  14.97%  77.13% #######
[ 13, 14)  88  10.54%  87.66% #####
[ 14, 15)  65   7.78%  95.45% ####
[ 15, 15]  38   4.55% 100.00% ##

Number of training obs by leaf:
Count: 835 Average: 24.2132 StdDev: 24.0984
Min: 5 Max: 229 Ignored: 0
----------------------------------------------
[   5,  16) 397  47.54%  47.54% ##########
[  16,  27) 195  23.35%  70.90% #####
[  27,  38) 108  12.93%  83.83% ###
[  38,  50)  50   5.99%  89.82% #
[  50,  61)  26   3.11%  92.93% #
[  61,  72)  16   1.92%  94.85%
[  72,  83)  16   1.92%  96.77%
[  83,  95)  10   1.20%  97.96%
[  95, 106)   4   0.48%  98.44%
[ 106, 117)   2   0.24%  98.68%
[ 117, 128)   5   0.60%  99.28%
[ 128, 140)   1   0.12%  99.40%
[ 140, 151)   0   0.00%  99.40%
[ 151, 162)   0   0.00%  99.40%
[ 162, 173)   2   0.24%  99.64%
[ 173, 185)   2   0.24%  99.88%
[ 185, 196)   0   0.00%  99.88%
[ 196, 207)   0   0.00%  99.88%
[ 207, 218)   0   0.00%  99.88%
[ 218, 229]   1   0.12% 100.00%

Attribute in nodes:
        267 : wrr [NUMERICAL]
        202 : wrc [CATEGORICAL]
        128 : bkr [NUMERICAL]
        113 : bkc [CATEGORICAL]
        79 : wkc [CATEGORICAL]
        45 : wkr [NUMERICAL]

Attribute in nodes with depth <= 0:
        1 : bkr [NUMERICAL]

Attribute in nodes with depth <= 1:
        1 : wkr [NUMERICAL]
        1 : bkr [NUMERICAL]
        1 : bkc [CATEGORICAL]

Attribute in nodes with depth <= 2:
        3 : bkc [CATEGORICAL]
        1 : wrc [CATEGORICAL]
        1 : wkr [NUMERICAL]
        1 : wkc [CATEGORICAL]
        1 : bkr [NUMERICAL]

Attribute in nodes with depth <= 3:
        3 : wkr [NUMERICAL]
        3 : wkc [CATEGORICAL]
        3 : bkr [NUMERICAL]
        3 : bkc [CATEGORICAL]
        2 : wrc [CATEGORICAL]
        1 : wrr [NUMERICAL]

Attribute in nodes with depth <= 5:
        16 : wrc [CATEGORICAL]
        10 : wrr [NUMERICAL]
        10 : bkr [NUMERICAL]
        9 : wkc [CATEGORICAL]
        9 : bkc [CATEGORICAL]
        8 : wkr [NUMERICAL]

Condition type in nodes:
        440 : HigherCondition
        394 : ContainsBitmapCondition
Condition type in nodes with depth <= 0:
        1 : HigherCondition
Condition type in nodes with depth <= 1:
        2 : HigherCondition
        1 : ContainsBitmapCondition
Condition type in nodes with depth <= 2:
        5 : ContainsBitmapCondition
        2 : HigherCondition
Condition type in nodes with depth <= 3:
        8 : ContainsBitmapCondition
        7 : HigherCondition
Condition type in nodes with depth <= 5:
        34 : ContainsBitmapCondition
        28 : HigherCondition
```

[36]:
```
preds = model_CART_1.predict(test)
preds = np.argmax(preds, axis=1)

metrics_CART_1= compute_metrics_multiclass(y_test_aux, preds)
metrics_CART_1
```

[36]:
```
[0.6307527648947556,
 0.6307527648947556,
 0.6307527648947556,
 0.5866906674682764]
```

Obtenemos unos malos resultados con un 63% de Precision, F1 y Recall. Es lógico, ya que tanto RandomForest como BoostedTrees ofrecen mejores resultados que CART al usar éste último un único árbol.

4.5 Comparación eficiencia RandomForest de Tensorflow vs Scikit-Learn

En el apartado de análisis exploratorio de datos usamos RandomForest con la librería Scikit-Learn para estudiar las variables predictoras con más importancia. Podemos pensar qué ventajas aporta esta nueva librería de TensorFlow frente a la ya mencionada anterioremente.

Hay aspectos claros, la librería de TensorFlow ofrece una mayor integración en el ecosistema y permite crear modelos de árboles que interactúen con redes neuronales de Keras. Además, el número de métodos de las clases así como los parámetros de customización son mayores (recomendamos ver la documentación de la API).

Nos preguntamos si, al usar la librería Yggdrasil *under the hood*, obtenemos una mayor rápidez.

Para ello creamos dos modelos de RandomForest con ambas librerías. Los dos tienen la misma configuración: 300 árboles y una profundidad máxima de 8 nodos.

[45]:
```
from sklearn.ensemble import RandomForestClassifier

train_sk, test_sk = split_dataset(data)

clf = RandomForestClassifier(max_depth=8,
                             random_state=0,
                             n_estimators=300)

%timeit clf.fit(train_sk.drop('opt rank', axis=1), train_sk['opt rank'])
```

```
2.21 s ± 541 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[46]:
```
model = tfdf.keras.RandomForestModel(max_depth=8,
                                     num_trees=300)

%timeit model.fit(train)
```

```
351/351 [==============================] - 0s 1ms/step
351/351 [==============================] - 0s 1ms/step
351/351 [==============================] - 0s 1ms/step
351/351 [==============================] - 0s 1ms/step
WARNING:tensorflow:5 out of the last 92 calls to <function
CoreModel.make_predict_function.<locals>.predict_function_trained at
0x7f521052ee50> triggered tf.function retracing. Tracing is expensive and␣
↪the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has␣
↪experimental_relax_shapes=True
option that relaxes argument shapes that can avoid unnecessary retracing.␣
↪For
(3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.

WARNING:tensorflow:5 out of the last 92 calls to <function
CoreModel.make_predict_function.<locals>.predict_function_trained at
0x7f521052ee50> triggered tf.function retracing. Tracing is expensive and␣
↪the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has␣
↪experimental_relax_shapes=True
option that relaxes argument shapes that can avoid unnecessary retracing.␣
↪For
(3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.

351/351 [==============================] - 0s 1ms/step
WARNING:tensorflow:6 out of the last 93 calls to <function
CoreModel.make_predict_function.<locals>.predict_function_trained at
0x7f521042a8b0> triggered tf.function retracing. Tracing is expensive and␣
↪the
excessive number of tracings could be due to (1) creating @tf.function
```

```
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has␣
↪experimental_relax_shapes=True
option that relaxes argument shapes that can avoid unnecessary retracing.␣
↪For
(3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.

WARNING:tensorflow:6 out of the last 93 calls to <function
CoreModel.make_predict_function.<locals>.predict_function_trained at
0x7f521042a8b0> triggered tf.function retracing. Tracing is expensive and␣
↪the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has␣
↪experimental_relax_shapes=True
option that relaxes argument shapes that can avoid unnecessary retracing.␣
↪For
(3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.

351/351 [==============================] - 0s 1ms/step
351/351 [==============================] - 0s 1ms/step
351/351 [==============================] - 0s 1ms/step
449 ms ± 4.69 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Mientras que la función de la librería Scikit-learn ha necesitado 2.21 s (de media) para realizar el entrenamiento, la función de TensorFlow lo ha hecho en 449 ms, unas cinco veces más rápido.

Vemos que la nueva librería ofrece resultados más rápidos. Intuimos que la diferencia en tiempos aumentará de forma notable al usar dataset más grandes y con configuraciones de árboles más complejas.

4.6 Conclusiones

Mostramos los resultados de las métricas obtenidas por los algoritmos usados en este apartado:

```
[60]: metricas = [metrics_rf, metrics_bt_1, metrics_bt_2, metrics_bt_3,␣
       ↪metrics_CART_1]
      precision = []
```

```
recall = []
f1 = []
cohen_kappa = []

for metrica in metricas:
    precision.append(metrica[0])
    recall.append(metrica[1])
    f1.append(metrica[2])
    cohen_kappa.append(metrica[3])

data = {"Modelo" : ["Random Forest", "BT por defecto", "BT más complejo",␣
 ↪"BT mejor benchmark","CART"],
         "Precision" : precision,
        "Recall": recall,
        "F1" : f1,
        "Cohen kappa" : cohen_kappa}
pd.DataFrame(data)
```

```
[60]:                Modelo  Precision    Recall        F1  Cohen kappa
      0       Random Forest   0.768284  0.768284  0.768284     0.740887
      1      BT por defecto   0.891366  0.891366  0.891366     0.878577
      2     BT más complejo   0.881199  0.881199  0.881199     0.867225
      3  BT mejor benchmark   0.855512  0.855512  0.855512     0.838482
      4                CART   0.630753  0.630753  0.630753     0.586691
```

En la peor posición encontramos el algoritmo de tipo CART. Lógico, ya que solo crea un árbol mientras que los demás usan ensembles. Random Forest obtiene aproximadamente un 76% en las métricas, pero es superado por rendimiento por todas las pruebas de Boosted Trees.

En las configuraciones utilizadas, los parámetros por defecto obtienen mejores resultados (89% en Precision, Recall y F1) que el modelo más complejo y usando la plantilla recomendada por la librería que ofrecía mejores resultados.

Además hemos comprobado que esta librería es más eficiente en tiempo computacional que los algoritmos incluidos en la librería Scikit-Learn.

En conclusión, hemos obtenido unos resultados que igualan a los mejores conseguidos con redes neuronales con unos modelos mucho más fáciles de configurar y más rápidos de entrenar.