Universidad de Granada

Inteligencia de Negocio

Práctica 3: Competición en DrivenData.

Miguel Ángel Torres López (Grupo 1)

torresma1996@correo.ugr.es

January 10, 2019

Pump it Up: Data Mining the Water Table

HOSTED BY DRIVENDATA

SUBMISSIONS

Score	Submitted by	Timestamp
0.7585	MiguelAngel_Torres_UGR	2018-12-24 17:30:26 UTC
0.7588	MiguelAngel_Torres_UGR	2018-12-24 18:07:03 UTC
0.7566	MiguelAngel_Torres_UGR	2018-12-25 11:30:24 UTC
0.7705	MiguelAngel_Torres_UGR	2018-12-25 11:57:01 UTC
0.7747	MiguelAngel_Torres_UGR	2018-12-25 12:01:38 UTC
0.6685	MiguelAngel_Torres_UGR	2018-12-26 20:10:54 UTC
0.6747	MiguelAngel_Torres_UGR	2018-12-26 20:13:43 UTC
0.6640	MiguelAngel_Torres_UGR	2018-12-26 20:17:37 UTC
0.7759	MiguelAngel_Torres_UGR	2018-12-27 09:02:13 UTC
0.7615	MiguelAngel_Torres_UGR	2018-12-27 10:23:03 UTC
0.7776	MiguelAngel_Torres_UGR	2018-12-28 11:07:39 UTC
0.7769	MiguelAngel_Torres_UGR	2018-12-28 11:21:48 UTC
0.7470	MiguelAngel_Torres_UGR	2018-12-28 16:57:59 UTC
0.8053	MiguelAngel_Torres_UGR	2018-12-29 10:51:02 UTC
0.8109	MiguelAngel_Torres_UGR	2018-12-29 11:11:41 UTC
0.8146	MiguelAngel_Torres_UGR	2018-12-29 17:39:59 UTC
0.7514	MiguelAngel_Torres_UGR	2018-12-30 10:55:53 UTC
0.7517	MiguelAngel_Torres_UGR	2018-12-30 10:57:14 UTC
0.7510	MiguelAngel_Torres_UGR	2018-12-30 11:29:41 UTC
0.8102	MiguelAngel_Torres_UGR	2018-12-31 11:30:21 UTC
0.8104	MiguelAngel_Torres_UGR	2018-12-31 11:31:58 UTC
0.8143	MiguelAngel_Torres_UGR	2018-12-31 11:36:51 UTC
0.8145	MiguelAngel_Torres_UGR	2019-01-01 17:09:00 UTC
0.8134	MiguelAngel_Torres_UGR	2019-01-01 17:30:14 UTC
0.7716	MiguelAngel_Torres_UGR	2019-01-01 17:36:56 UTC
0.8166	MiguelAngel_Torres_UGR	2019-01-02 14:30:37 UTC
0.8141	MiguelAngel_Torres_UGR	2019-01-02 14:34:23 UTC
0.8185	MiguelAngel_Torres_UGR	2019-01-02 14:37:28 UTC

Contents

1 Introducción **4**
1.1 Problema abordado 4
1.2 Valoración de una predicción 4
1.3 Envíos a DrivenData 6
1.3.1 Estrategia seguida 6

2 Algoritmos **8**
2.1 LightGBM 8
2.2 Random Forest 8
2.3 SVC 9
2.4 Linear SVC 10
2.5 XGBoost 11

3 Preprocesamiento de datos **13**

4 Algoritmos tras preprocesado **16**
4.1 XGBoost 16
4.2 Random Forest 16
4.3 Random Forest con características polinómicas 17

5 Bibliografía **18**

List of Figures

1 Similitud entre clase real y predicción. 4
2 Similitud entre clase real y predicción con casos ordenados. . 5
3 Predicción sobre el conjunto de training en LightGBM. 8
4 Predicción sobre el conjunto de test en LightGBM. 8
5 Predicción sobre el conjunto de training en Random Forest. . 9
6 Predicción sobre el conjunto de test en Random Forest. . . . 9
7 Predicción sobre el conjunto de training con SVC. 10
8 Predicción sobre el conjunto de test con SVC. 10
9 Predicción sobre el conjunto de training con Linear SVC. . . 10
10 Predicción sobre el conjunto de test con Linear SVC. 11
11 Predicción sobre el conjunto de training con XGBoost. 11
12 Predicción sobre el conjunto de test con XGBoost. 11
13 Atributos añadidos después del preprocesado. 15
14 Predicción sobre el conjunto de training preprocesado con XGBoost. 16
15 Predicción sobre el conjunto de test preprocesado con XGBoost. 16
16 Predicción sobre el conjunto de training preprocesado con Random Forest. 16

17 Predicción sobre el conjunto de test preprocesado con Random Forest. 16
18 Predicción sobre el conjunto de training preprocesado con Random Forest y características polinómicas. 17
19 Predicción sobre el conjunto de test preprocesado con Random Forest y características polinómicas. 17

1 Introducción

1.1 Problema abordado

Esta práctica tiene como objetivo realizar una predicción lo más acertada posible del problema *Pump it Up: Data Mining the Water Table*, que puede encontrarse en *DrivenData*.

El problema consiste en predecir el estado de un punto de extracción de agua en Tanzania a partir de un total de 59400 casos de ejemplo. De cada caso se aportan 39 atributos y su estado real, a saber funcional, no funcional o funcional pero necesita reparación.

Dado un conjunto de 14850 pozos de los que conocemos sus atributos, se pide predecir el estado de cada uno de ellos.

1.2 Valoración de una predicción

Es algo natural comprobar cómo de buenas son las predicciones realizadas con unos u otros algoritmos para poder compararlos y afinar los resultados. Pero en esta competición tenemos un problema, no podemos comprobar como de buena es la predicción sobre el conjunto de test hasta que no se envíe la solución a *DrivenData*. Cómo solo tenemos tres envíos al día, tendremos que aprovecharlos para enviar las mejores soluciones de entre todas las realizaciones.

Para poder elegir las mejores ejecuciones vamos a partir, de forma estratificada, los datos de entrenamiento que nos dan, ya que de estos si conocemos la clasificiación correcta. Bien con un *cross validation* o bien con una partición simple, vamos a medir la puntuación de cada algoritmo como en la precisión simple de dicho algoritmo.

Además, se ha realizado, con ayuda de la librería PIL (`http://effbot.org/zone/pil-index.htm`), un método de desarrollo propio para realizar gráficas en las que se contrasta mejor los fallos y aciertos del algoritmo. A continuación se muestra un ejemplo:

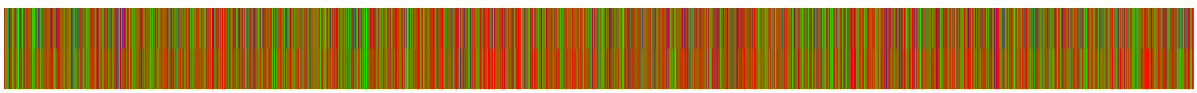

Figure 1: Similitud entre clase real y predicción.

En la parte superior se muestra el color de la clasificación correcta y en la inferior la predicción realizada por el algoritmo. De ahora en adelante se utilizará el mismo código de colores: rojo para *functional*, verde para *non*

functional y azul para *functional needs repair*. Nótese que cuando hacemos la predicción sobre el conjunto de test desconocemos la clase real. Por eso solo se mostrará la mitad inferior del gráfico.

Debido a la cantidad de datos y a la reducción de la imagen no se diferencian los colores de cada individuo. Se ha realizado un segundo método que muestra los mismos indiviudos con los casos ordenados por colores:

Figure 2: Similitud entre clase real y predicción con casos ordenados.

El código para realizar dichas gráficas se muestra a continuación, aunque no se entra en detalle por carecer de interés real para esta práctica.

```
# -*- coding: utf-8 -*-

from PIL import Image

def similarityBar(y_real, y_prediction, height):
  print(y_real)
  print(y_prediction)

  img = Image.new( 'RGB', (y_real.size, height), 'black')
  pixels = img.load()
  for i in range(y_real.size):
    for j in range(int(height/2)):
      if y_real[i] == 'functional' :
        pixels[i,j] = (255, 0, 0)
      elif y_real[i] == 'non functional' :
        pixels[i,j] = (0, 255, 0)
      else:
        pixels[i,j] = (0, 0, 255)

      if y_prediction[i] == 'functional' :
        pixels[i,j+int(height/2)] = (255, 0, 0)
      elif y_prediction[i] == 'non functional' :
        pixels[i,j+int(height/2)] = (0, 255, 0)
      else:
        pixels[i,j+int(height/2)] = (0, 0, 255)

  img.save('image.bmp')


def orderedSimilarityBar(y_real, y_prediction, height, point_per_pixel):
  img = Image.new( 'RGB', (int(y_real.size/point_per_pixel), height), 'black')
  pixels = img.load()
  counter = [[0,0,0],[0,0,0],[0,0,0]]  # counter[i][j] means y_real=i and y_prediction=j
  color = [(255,0,0),(0,255,0),(0,0,255)]  # functional = red, non functional = green, functional need repair

  for i in range(y_real.size):
    if y_real[i] == 'functional' :
      y_real[i] = 0
    elif y_real[i] == 'non functional':
      y_real[i] = 1
    else:
      y_real[i] = 2

    if y_prediction[i] == 'functional' :
      y_prediction[i] = 0
    elif y_prediction[i] == 'non functional':
      y_prediction[i] = 1
    else:
      y_prediction[i] = 2

  for i in range(y_real.size):
```

```
    counter[y_real[i]][y_prediction[i]] = counter[y_real[i]][y_prediction[i]] + 1

aux = 0

for i in range(3) :
  for j in range(3) :
    counter[i][j] = int(counter[i][j] / point_per_pixel)
    for l in range(counter[i][j]) :
      for k in range(int(height/2)) :
        pixels[aux + l,k] = color[i]
        pixels[aux + l,k+int(height/2)] = color[j]
    aux += counter[i][j]

img.save('image.bmp')
```

1.3 Envíos a DrivenData

En *DrivenData*, cada usuario puede enviar tres submissions al da. Por tanto, es bastante importante medir las posibilidades de acierto que se tienen antes de enviar una nueva solución. En este apartado se explicarán los motivos por los que se escogieron cada una de las respuestas enviadas a *DrivenData*.

Fecha subida	Posición subida	Score training	Score test	Procesado	Algoritmo	Configuración
24-12	1407	0'825	0'7585	Categóricas a numéricas	LightGBM	Binario, 200 estimadores
24-12	1396	0'84	0'7588	Categóricas a numéricas	LightGBM	Binario, 300 estimadores
25-12	1396	0'8585	0'7566	Categóricas a numéricas	LightGBM	Binario, 500 estimadores
25-12	1327	0'9562	0'7705	Categóricas a numéricas, not NAN	RandomForest	100 árboles, 6 indiv para split
25-12	1316	0'9242	0'7747	Categóricas a numéricas, not NAN	RandomForest	300 árboles, 10 indiv para split
26-12	1318	0'9242	0.6685	Anteriores y Tiers fundadores	RandomForest	300 árboles, 10 indiv para split
26-12	1318	0'8849	0.6747	Anteriores fundadores	RandomForest	500 árboles, 20 indiv para split
26-12	1318	0'8847	0.6640	Anterior y Tiers fundadores numéricos	RandomForest	500 árboles, 20 indiv para split
27-12	1311	0'8849	0.7759	Anterior y Tiers fundadores en test	RandomForest	500 árboles, 20 indiv para split
27-12	1311	0'893	0.7615	Anterior	LightGBM	Multiclase, 300 estimadores, máximo 61 hojas
28-12	1303	0'9971	0.7776	Anterior con Nan=mediana	RandomForest	700 árboles 2 indiv para split
28-12	1303	1	0.7769	Anterior	RandomForest	Multiclase, 1000 árboles
28-12	1303	0'9207	0.747	Anterior	XGBoost	learn-rate=0.17, max_depth=10, árboles=200
29-12	906	0'9508	0.8053	Anterior y funder group rates	XGBoost	learn-rate=0.17, max_depth=10, árboles=200
29-12	792	0'9653	0.8109	Anterior	Random Forest	árboles=700, 6 individuos para split
29-12	677	0'9778	0.8146	Anterior, lga tiers y lga rates	Random Forest	árboles=700, 6 individuos para split
30-12	678	0'9692	0.7514	Anterior y característica polinómica	XGBoost	learn-rate=0.17, max_depth=10, árboles=200
30-12	678	0'9871	0.7517	Anterior	Random Forest	árboles=700, 6 individuos para split
30-12	678	0'9912	0.7510	Anterior	Random Forest	árboles=1500, 20 individuos para split
31-12	678	0'9834	0.8102	Anterior y limpieza del año de construcción	XGBoost	learn-rate=0.17, max_depth=10, árboles=200
31-12	678	0'9856	0.8104	Anterior	XGBoost	learn-rate=0.17, max_depth=10, árboles=200
31-12	678	0'9923	0.8143	Anterior	Random Forest	árboles=1500, 20 individuos para split
1-1	679	0'9817	0.8145	Anterior y limpieza de 'permit'	Random Forest	árboles=700, 20 individuos para split
1-1	679	0'9643	0.8134	Anterior	Random Forest	árboles=200, 4 individuos para split
1-1	679	0'9545	0.7716	Anterior y algo más	Random Forest	árboles=200, 2 individuos para split
2-1	570	0'9713	0.8166	Anterior	Random Forest	árboles=200, 20 de profundidad máxima
2-1	570	0'9548	0.8141	Anterior	Random Forest	árboles=200, 18 de profundidad máxima
2-1	460	0'9812	0.8185	Anterior	Random Forest	árboles=200, 22 de profundidad máxima

1.3.1 Estrategia seguida

Al empezar la competición no tenemos ninguna estimación de cual es una buena puntuación en el problema. Sí, tenemos la tabla de clasificiación con las mejores 50 soluciones, pero no tenemos ni idea de que algoritmos o preprocesamientos de datos fueron llevados a cabo para sacar esta solución. Por lo tanto, las primeras submissions en *DrivenData* tiene como objetivo establecer una puntuación base de la que partimos para mejorar.

Por facilidad, se ha usado el ejemplo que se nos facilitó en la página de la web, un uso simple del framework LightGBM. Se enviaron tres respuestas basadas en distintas configuraciones del mismo. Se observa poca variación entre las tres, pero se intuye que en la tercera (LightGBM de 500 estimadores) se provoca un ligero sobreaprendizaje.

Justo después, se prueba con un algoritmo que funciona bastante bien en la mayor parte de los problemas, el Random Forest. Los dos primeros envíos tienen el mismo preprocesamiento de datos que en el caso anterior, a excepción de la eliminación de los valores perdidos, que no se pueden utilizar en los árboles. En sendos envíos mejoramos la puntuación base de partida, que será un 0'7747 de puntuación que nos sitúa en la posición 1316.

Una vez que tenemos la puntuación base, se empieza a probar con prepocesado de datos. Los cambios sobre los datos se han ido haciendo de forma gradual. Al principio se rellenan los valores perdidos con la mediana de la columna. Luego se han hecho los *Tiers* de fundadores y las proporciones de estado de los fundadores. Siguiendo el mismo patrón se han añadido el tamaño de los lga y las proporciones de estado de los mismos.

Por último, con menos efectividad, se han realizado características polinómicas, es decir, se han calculado los productos de distintas parejas de atributos y se colocado como nuevos atributos.

2 Algoritmos

2.1 LightGBM

El algoritmo LightGBM es un algoritmo basado en árboles bastante robusto. De entre sus muchos parámetros se han probado numerosas combinaciones, aunque los resultados de una otra ejecución no tienen diferencias significativas.

Se obtiene una precisión moderadamente buena en el conjunto de entrenamiento. También alta en conjunto de test, pero un poco baja frente a nuestro objetivo.

En entrenamiento obtenemos una precisión de 0'893. En el conjunto de datos de test, el que hay que predecir en *DrivenData*, la precisión cae a 0'7615. En cuanto a los aciertos y fallos separados en clases tenemos un reparto ligeramente desigual. Como se observa en la figura 3, la clase *functional* se predice con bastante frecuencia bien, pero se introduce una cantidad significativa de pozos de las otras dos clases que se predicen de esta.

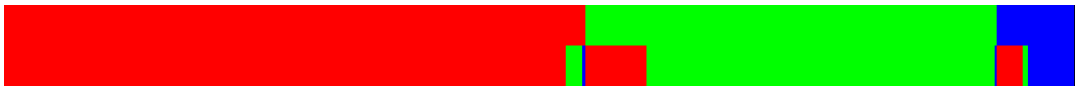

Figure 3: Predicción sobre el conjunto de training en LightGBM.

En cuanto a la predicción sobre el conjunto de test, podemos mencionar que se clasifican muchos pozos como *functional* pero pocos como *functional needs repair*. Suponiendo que el conjunto de test está balanceado igual que el de training, debería haber pozos de la tercera clase clasificados como de la primera de forma incorrecta.

Figure 4: Predicción sobre el conjunto de test en LightGBM.

2.2 Random Forest

Este es una algoritmo que funciona bien en la mayoría de los problemas y, como no se esperaba, en este también. Se pueden modificar varios parámetros, pero principalmente vamos a trabajar con el número de árboles y el mínimo de individuos que se necesitan para dividir una hoja. El número de árboles mejorará el caracter aleatorio del algoritmo y el mínimo de individuos contralará la profundidad del aprendizaje (impide que lleguemos al sobreapren-

dizaje).

Obtenemos unos valores similares para la precisión en el caso de training y test. Usando 1000 árboles y 10 individuos mínimos en el primero se queda en 0'9237 y en el segundo a 0'7747.

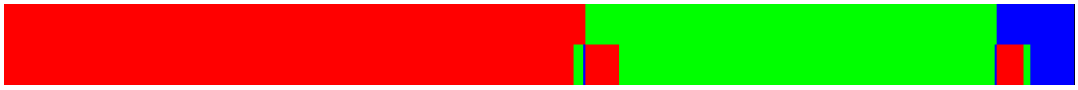

Figure 5: Predicción sobre el conjunto de training en Random Forest.

Figure 6: Predicción sobre el conjunto de test en Random Forest.

Este algoritmo predice un poco mejor la primera y segunda clase que el anterior. No obstante, sigue cometiendo bastantes fallos en la tercera clase, que la confunde, en mayor parte, con la primera.

2.3 SVC

Lo primero que se debe destacar del algoritmo SVC de scikit-learn es la complejidad. Como se puede leer en https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC, tiene una complejidad superior a cuadrática, lo que provocará que para nuestro caso, con casi 60000 pozos, sea inviable su ejecución.

Por ello, se ha optado por hacer un partición estratificada e intentar predecir el conjunto de test con solo una porción de los datos de training que se nos facilitan:

```
sss = StratifiedShuffleSplit(n_splits=1, test_size=0.8)
for train_index, test_index in sss.split(X,y):
  X_train, y_train = X[train_index], y[train_index]
```

Probamos con distintos parámetros para este algoritmo. Con kernel *rbf* de gamma *auto* y *scale* y *poly* de distintos grados.

Tras probar con sendas configuraciones, podemos comprobar el modelo aprende bastante bien sobre los datos. Su puntuacin sobre los datos de training es de 0'98 en media, es decir, aprende a predecir los datos sobre los que ha aprendido bastante bien con todas las configuraciones. Puede observarser este hecho en la figura 7 Pero a la hora de inferir la clase de los pozos del conjunto de test es bastante desastroso. Marca todos los pozos como *functional*, como puede verse en la figura 8. Por este motivo, no vamos

ni si quiera a subir la solución a *DrivenData*.

Figure 7: Predicción sobre el conjunto de training con SVC.

Figure 8: Predicción sobre el conjunto de test con SVC.

Es díficil asegurar por qué se realiza esta mala predicción. Una de las posibilidades es que el conjunto de test contenga datos que, a la hora de hacer la separación con SVC, estén orientados a caer en la región de los pozos funcionales. Otra posible explicación es el sobreaprendizaje que que puede provocar el algoritmo SVC. Aunque a priori es más posible la primera opción, por el caracter clasificatorio de este tipo de algoritmos de región.

2.4 Linear SVC

Este algoritmo es una relajación del anterior. Es de complejidad menor y, por tanto, podemos utilizar una mayor cantidad de datos para entrenar, aunque las regiones que conforman las clases van a ser más simples. Como es un algoritmo iterativo basado en convergencia y queremos la mejor aproximación, vamos a poner un límite de iteraciones alto. Preferimos que el algoritmo se pare al superar la tolerancia de convergencia y no al superar el límite de pasos.

Usamos el modo dual. Se recomienda en la documentación que se use cuando el número de muestras es mayor que el número de atributos.

Tras probar con distintas configuraciones de tolerancia, la mejor configuración nos da una precisión en training de 0'6577, algo bastante alejado de lo requerido. Además, observando al figura 9, donde se comparan las predicciones con la clase real, es claro que hay un problema al predecir la clase *functional needs repair*.

Figure 9: Predicción sobre el conjunto de training con Linear SVC.

Figure 10: Predicción sobre el conjunto de test con Linear SVC.

La clase *functional*, de color rojo, se predice demasiadas veces cuando la clase real es *functional needs repair*, e incluso con *non functional*.

Esto refuerza la idea que ya se ha planteado anteriormente, es bastante complicado separar los pozos con algoritmos basados en regiones. Mientras que el LightGBM y Random Forest consiguen una precisión aceptable, los algoritmos basados en SVC fallan en la diferenciación de una de las clases frente a las otras.

2.5 XGBoost

Desde la biblioteca XGBoost podemos usar el algoritmo con su mismo nombre. Está basado en un Random Forest ponderando a los individuos que se clasificaron mal en árboles anteriores. Con este algoritmo, por tanto, se esperan mejores resultados a la hora de clasificar datos conflictivos. En nuestro caso, podríamos intentar clasificar mejor los pozos que tengan la etiqueta *functional needs repair*.

El algoritmo tiene una amplia gama de configuraciones. Vamos a dejar fijo el número de árboles en 200, para no demorar mucho las pruebas. Y realizamos varias ejecuciones cambiando los parámetros *colsample_bytree*, *learning_rate* y *max_depth*, midiendo la precisión que tiene el modelo aprendido con el conjunto de training para no enviar todas las pruebas a *DrivenData*.

La mejor se obtiene con $colsample\_bytree = 1$, $learning\_rate = 0.17$ y $max\_depth = 10$.

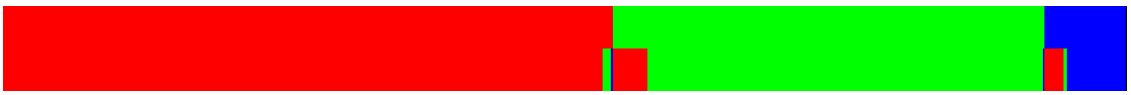

Figure 11: Predicción sobre el conjunto de training con XGBoost.

Figure 12: Predicción sobre el conjunto de test con XGBoost.

Obtenemos unas ejecuciones similares a las del Random Forest aunque, como se esperaba, los fallos en la clase *functional needs repair* se han re-

ducido considerablemente. En la predicción sobre el test también se observan algunos individuos más clasificados en esta tercera clase.

Se podría intentar hacer una solución combinada entre un buen resultado en Random Forest y otro en XGBoost. Dando más valor a los individuos que se clasifiquen en la tercera clase con XGBoost.

3 Preprocesamiento de datos

Tras probar varias configuraciones y hacer varias pruebas más, salta a la vista que los datos de la tercera clase se confunden, a menudo, con los de la primera. Es neceario modificar los datos para intentar crear alguna diferencia entre sendas clases.

- **Limpieza de la columna permit**. En esta columna se supone que hay valores booleanos, pero vienen representados de distintas formas, con 1 o 0 y con TRUE o FALSE. Unificamos el sistema de representación para que los algoritmos no tomen 4 valores como posibles en lugar de 2.

- **Limpieza de latitud y longitud**. Al observar estos dos valores, nos damos cuenta de que hay muchos valores cercanos 0. El país se situa en las coordenadas 32,-5 aproximadamente. Por lo que las coordenadas cercanas a 0,0 pueden tomarse como valores perdidos.

  Para sustituir estos valores por otros con algo más de sentido, vamos a imponer como valor la media. Aunque no será la media de todos los pozos, si no la media de los pozos que están en la misma región que el pozo con latitud y longitud erróneas. Esto va a producir datos más fiables.

- **Tiers de fundadores**. Entre la información que se puede percibir de los datos hay varias que los algoritmos no pueden extraer. Vamos a hacer varios tiers de fundadores según el número de pozos que han instalado. Así los fundadores de pozos que más han instalado serán situados en el tier 10, y los que menos en el tier 1.

  Para ello, haremos un dataframe auxiliar en el que el índice será el nombre del fundador y una columna values tendrá la suma de pozos con ese nuevo índice. El código exacto puede verse junto con el código del siguiente punto.

- **Proporciones de fundadores**. Siguiendo la idea del punto anterior, podemos ampliar aun más la información que tenemos de los fundadores con información implícita en los datos. Podemos calcular la proporción de pozos funcionales, no funcionales y no funcionales pero reparables de cada fundador y añadirla a cada uno de los pozos. Esto nos aporta un conocimiento previo bastante fuerte sobre la clase del pozo, ya que si un fundador tiene todos los pozos conocidos rotos, lo más probable es que un nuevo pozo, que desconocíamos, de este fundador también esté roto.

Claramente, estamos añadiendo información redundante con estos nuevos 4 atributos. Así que, si se quiere, se puede eliminar ahora el atributo *funder*, del que ya tenemos bastante información implícita.

```
def funder_cat(x):
  if x <= 1:
    return 1
  elif (x>1) & (x<=5):
    return 2
  elif (x>5) & (x<=10):
    return 3
  elif (x>10) & (x<=20):
    return 4
  elif (x>20) & (x<=50):
    return 5
  elif (x>50) & (x<=100):
    return 6
  elif (x>100) & (x<=150):
    return 7
  elif (x>150) & (x<=200):
    return 8
  elif (x>200) & (x<=300):
    return 9
  elif (x>300):
    return 10

def funder_clean(X, y, X_tst):
  aux = pd.concat([X, y], axis=1, sort=False)
  funders = aux.pivot_table(index=['funder'], values='id', aggfunc=len)
  X['funder_size'] = np.nan
  X['funct_rate'] = np.nan
  X['non_funct_rate'] = np.nan
  X['needs_repair_rate'] = np.nan
  X_tst['funder_size'] = np.nan
  X_tst['funct_rate'] = np.nan
  X_tst['non_funct_rate'] = np.nan
  X_tst['needs_repair_rate'] = np.nan

  for index, row in funders.iterrows():
    funct = non_funct = funct_repair = 0
    funct = aux.loc[(aux['funder']==index) & (aux['status_group']=='functional')].size
    non_funct = aux.loc[(aux['funder']==index) &
       (aux['status_group']=='non functional')].size
    funct_repair = aux.loc[(aux['funder']==index) &
       (aux['status_group']=='functional needs repair')].size
    total = funct + non_funct + funct_repair

    X.loc[X['funder']==index,'funder_size'] = funder_cat(total)
    X.loc[X['funder']==index,'funct_rate'] = funct_rate
    X.loc[X['funder']==index,'non_funct_rate'] = non_funct_rate
    X.loc[X['funder']==index,'needs_repair_rate'] = needs_repair_rate
    X_tst.loc[X_tst['funder']==index,'funder_size'] = funder_cat(total)
    X_tst.loc[X_tst['funder']==index,'funct_rate'] = funct_rate
    X_tst.loc[X_tst['funder']==index,'non_funct_rate'] = non_funct_rate
    X_tst.loc[X_tst['funder']==index,'needs_repair_rate'] = needs_repair_rate
    print('Procesado el fundador: ' + index )

  return X, X_tst
```

- **Proporciones de lga**. Al igual que con los fundadores, tiene sentido pensar que los pozos de una zona u otra tengan un destino parecido. En una zona seca o despoblada será más comúm encontrarse pozos abandonados y rotos.

  Hacemos un preprocesado similar con este atributo, aunque se espera que sea algo menos fructrífero que los dos puntos anteriores. Igualmente se adjuntarán a los datos el tamaño de cada lga (el número de pozos en el lga) y las tres proporciones de las clasificaciones.

Tras el preprocesamiento hemos adherido a los datos 8 nuevos atributos. En la figura 13 tenemos un ejemplo de los 8 atributos calculados para ciertos pozos del conjunto de test.

funder_size	funct_rate	non_funct_rate	needs_repair_rate	lga_size	lga_funct_rate	lga_non_funct_rate	lga_needs_repair_rate
10.0	0.8602150537634409	0.1021505376344086	0.03763440860215054	4.0	0.6430976430976431	0.32323232323232326	0.03367003367003367
10.0	0.4095112285336856	0.5133201232937032	0.07716864817261118	9.0	0.6988817891373802	0.2627955271565495	0.038338658146964855
10.0	0.5500963391136802	0.3988439306358382	0.05882352941176471	8.0	0.4402010050251256	0.49748743718592964	0.062311557788944726
10.0	0.060606060606060615	0.9393939393939394	0.0	2.0	0.2532467532467532	0.7272727272727273	0.01948051948051948

Figure 13: Atributos añadidos después del preprocesado.

4 Algoritmos tras preprocesado

4.1 XGBoost

El algoritmo XGBoost gana poco en precisión sobre el training. Obtiene, para los mismos parámetros, valores bastantes similares después de meter en juego los 8 nuevos atributos.

Pero, al enviar la solución a *DrivenData* podemos observar como se supera la barrera del 0'78 en puntuación y se llega a 0'8053. Valor que esta algo más proximo al 0'8296 que corresponde a la puntuación actual más alta.

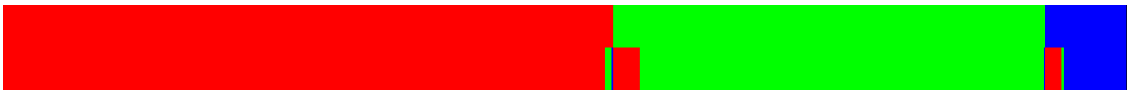

Figure 14: Predicción sobre el conjunto de training preprocesado con XGBoost.

Figure 15: Predicción sobre el conjunto de test preprocesado con XGBoost.

4.2 Random Forest

Viendo la mejora del algoritmo anterior con preprocesado, es de esperar que este también suba en puntuación. En la figura 16 vemos como la precisión ha aumentado en training sin alterar los parámetros del algortimo, esto es buen síntoma.

Figure 16: Predicción sobre el conjunto de training preprocesado con Random Forest.

Figure 17: Predicción sobre el conjunto de test preprocesado con Random Forest.

Las sospechas son ciertas, al subir una nueva solución a la web, obtenemos un 0'8109. Alterando los parámetros ajustamos un poco más la puntuación, llegando a los 0'8185 puntos.

4.3 Random Forest con características polinómicas

Una buena práctica es intentar crear nuevos atributos multiplicando parejas de los atributos que tenemos. En ocasiones es una forma de generar información que los algoritmos antes no podían tener en cuenta.

En este caso, sin embargo, aunque podemos ver en las figuras de aciertos que el modelo sobre el conjunto de training se aprende muy bien, la puntuación obtenida en *DrivenData* es baja.

Esto podría ser síntoma de un exceso de preprocesamiento. Aplicar este procedimiento sobre atributos que ya han sido creados artificialmente puede que desvirtue un poco el origen de los datos. Estamos creando relaciones que se cumplen bien en training pero no en test.

Figure 18: Predicción sobre el conjunto de training preprocesado con Random Forest y características polinómicas.

Figure 19: Predicción sobre el conjunto de test preprocesado con Random Forest y características polinómicas.

A pesar de que la precisión en training es similar a la del Random Forest sin característcas polinómicas, la precisión en test es bastante mala. Incluso se puede observar que la proporción de clasificado es similar en test, pero están mal clasificados.

5 Bibliografía

- Documentación de Scikit-learn (https://scikit-learn.org/stable/)
- Documentación de XGBoost (https://xgboost.readthedocs.io/en/latest/python/python_intro.html)
- Documentación de LightGBM (https://lightgbm.readthedocs.io/en/latest/Python-Intro.html)