Random Forest

Alain Lobato

30 de Marzo del 2025

1 Introducción

Un Random Forest es un algoritmo de aprendizaje automático de uso común, el cual utiliza distintos árboles de decisión para obtener un resultado único. Este surge debido al *overfitting* en los árboles de decisión. Se crea este modelo para evitar este problema, memorizando soluciones en lugar de realizar un aprendizaje generalizado, trabajando distintos árboles en conjunto.

2 Metodología

Primero clonamos nuestro repositorio de la materia en nuestra máquina y luego creamos una carpeta llamada "Random Forest". Descargamos el archivo CSV para extraer los datos y creamos un archivo llamado `random_forest.py` donde desarrollamos nuestro código usando Visual Studio Code.

Iniciamos importando las librerías necesarias, instalando las dependencias en caso de ser necesario.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeClassifier
from pylab import rcParams
from imblearn.under_sampling import NearMiss
from imblearn.over_sampling import RandomOverSampler
from imblearn.combine import SMOTETomek
from imblearn.ensemble import BalancedBaggingClassifier
from collections import Counter
from sklearn.ensemble import RandomForestClassifier
```

Figure 1: Importación de librerías

Ahora cargaremos los datos e imprimimos con `head` los primeros 5 registros, así como sus dimensiones.

```
#set up graphic style in this case I am using the color scheme from xkcd.com
rcParams['figure.figsize'] = 14, 8.7 # Golden Mean
LABELS = ["Normal","Fraud"]
#col_list = ["cerulean","scarlet"]# https://xkcd.com/color/rgb/
#sns.set(style='white', font_scale=1.75, palette=sns.xkcd_palette(col_list))

df = pd.read_csv("creditcard.csv")
print(df.head())
print(df.shape)

print(df['Class'].value_counts(sort=True))
```

Figure 2: Carga de datos

```
   Time        V1        V2        V3        V4        V5        V6  ...       V24       V25       V26       V27       V28  Amount  Cl
ass
0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  ...  0.066928  0.128539 -0.189115  0.133558 -0.021053  149.62
  0
1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361  ... -0.339846  0.167170  0.125895 -0.008983  0.014724    2.69
  0
2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  ... -0.689281 -0.327642 -0.139097 -0.055353 -0.059752  378.66
  0
3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  ... -1.175575  0.647376 -0.221929  0.062723  0.061458  123.50
  0
4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  ...  0.141267 -0.206010  0.502292  0.219422  0.215153   69.99
  0

[5 rows x 31 columns]
(284807, 31)
```

Figure 3: Resultados de la carga de datos

Ahora bipartimos los datos en clases, donde 0 es un acceso válido y 1 es un acceso fraudulento:

```
Class
0    284315
1       492
```

Figure 4: Clasificación de datos

Podemos ver cómo se encuentra un registro más cargado que el otro, con demasiados registros válidos y pocos registros fraudulentos.

Ahora creamos el dataset, separamos los datos válidos y fraudulentos, así como hacemos una división entre los datos independientes y nuestro *target*.

```
normal_df = df[df.Class == 0] #registros normales
fraud_df = df[df.Class == 1] #casos de fraude
y = df['Class']
X = df.drop('Class', axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7)

def mostrar_resultados(y_test, pred_y):
    conf_matrix = confusion_matrix(y_test, pred_y)
    plt.figure(figsize=(8, 8))
    sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d")
    plt.title("Confusion matrix")
    plt.ylabel('True class')
    plt.xlabel('Predicted class')
    plt.show()
    print (classification_report(y_test, pred_y))
```

Figure 5: Separación de datos

Ahora crearemos nuestro Random Forest usando los datos que ya separamos y lo entrenamos.

```
# Crear el modelo con 100 arboles
model = RandomForestClassifier(n_estimators=100,bootstrap = True, verbose=2,max_features = 'sqrt')
# a entrenar!
model.fit(X_train, y_train)
def run_model_balanced(X_train, X_test, y_train, y_test):
    clf = LogisticRegression(C=1.0,penalty='l2',random_state=1,solver="newton-cg",class_weight="balanced")
    clf.fit(X_train, y_train)
    return clf

model = run_model_balanced(X_train, X_test, y_train, y_test)
```

Figure 6: Creación y entrenamiento del modelo

3 Resultados

Vamos a predecir nuestros datos de test y mostramos los resultados usando la función que anteriormente declaramos:

```
pred_y = model.predict(X_test)
mostrar_resultados(y_test, pred_y)
```

Figure 7: Predicción de datos de prueba

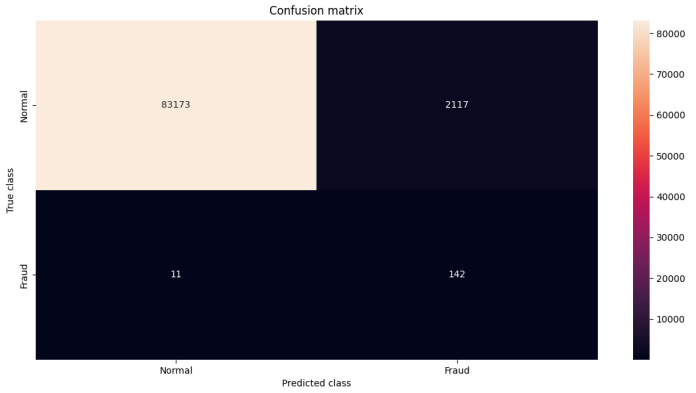

Figure 8: Resultados de la predicción

Podemos ver que se tienen buenos resultados. El reporte de clasificación arroja un *recall* de 0.98 y un F1-score macro promedio de 0.55, siendo muy buenos resultados.

	precision	recall	f1-score	support
0	1.00	0.98	0.99	85290
1	0.06	0.93	0.12	153
accuracy			0.98	85443
macro avg	0.53	0.95	0.55	85443
weighted avg	1.00	0.98	0.99	85443

Figure 9: Reporte de clasificación

4 Conclusión

Gracias a los Random Forest, se pudo evitar el problema de *overfitting* que ocurre al usar solo un árbol de decisión, lo que garantiza un mejor resultado a la hora de predecir algún dato. No fue necesario hacer ningún ajuste al código y todo resultó sencillo de realizar. Se repasó muy bien la información en esta práctica.

5 Referencias

Material de clase (2025). UANL.
Bagnato J. (2010). *Aprende Machine Learning.* Leanpub.