Random Forest en Python

[Carlos Oswaldo Gonzalez Garza]

March 31, 2025

1 Introducción

Random Forest es un modelo de Ensamble que combina varios árboles de decisión. La predicción final es la opción más votada. Además, calcula la importancia de las características, identificando las más relevantes para las predicciones.

2 Metodología

Se siguieron los siguientes pasos para realizar el análisis y modelado con técnicas de balanceo, empleando diversos modelos (incluyendo Regresión Logística, métodos de resampling y Random Forest):

1. **Carga y Exploración de Datos**
   Se importan las librerías necesarias, se carga el conjunto de datos y se visualiza una muestra y la forma del dataframe, así como la distribución de la variable objetivo.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, 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
```

```
# Cargar los datos
df = pd.read_csv("creditcard.csv")
df.head(n=5)

# Imprimir la forma del dataframe y las frecuencias de las clases
print(df.shape)
count_classes = pd.Series(df['Class']).value_counts(sort=True)
print(count_classes)
```

2. **Visualización de la Distribución de Clases**
   Se definen las etiquetas para el gráfico y se grafica la frecuencia de cada clase.

```
# Definir etiquetas para el gr\'afico
LABELS = ['Class 0', 'Class 1']

# Graficar la distribuci\'on de clases
count_classes.plot(kind='bar', rot=0)
plt.xticks(range(2), LABELS)
plt.title("Frequency by observation number")
plt.xlabel("Class")
plt.ylabel("Number of Observations")
plt.show()
```

3. **Preparación de los Datos**
   Se separan las características (X) y la variable objetivo (y) y se dividen los datos en conjuntos de entrenamiento y prueba.

```
# Definir las caracter\'isticas y las etiquetas
y = df['Class']
X = df.drop('Class', axis=1)

# Dividir en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7)
```

4. **Entrenamiento del Modelo Base**
   Se define una función que entrena un modelo de Regresión Logística y se ejecuta para obtener el modelo base.

```
# Crear una funci\'on para entrenar el modelo
def run_model(X_train, X_test, y_train, y_test):
    clf_base = LogisticRegression(C=1.0, penalty='l2',
                                  random_state=1, solver="newton-cg")
```

```
    clf_base.fit(X_train, y_train)
    return clf_base

# Ejecutar el modelo base
model = run_model(X_train, X_test, y_train, y_test)
```

5. **Evaluación del Modelo Base**
   Se define una función para mostrar la matriz de confusión y el reporte de clasificación, y se evalúa el modelo base.

```
# Definir una funci\'on para mostrar los resultados
def mostrar_resultados(y_test, pred_y):
    conf_matrix = confusion_matrix(y_test, pred_y)
    plt.figure(figsize=(12, 12))
    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))

# Predicci\'on con el modelo base
pred_y = model.predict(X_test)
mostrar_resultados(y_test, pred_y)
```

6. **Modelo Balanceado con `class_weight`**
   Se crea y entrena un modelo de Regresión Logística utilizando el parámetro `class_weight="balanced"` para abordar el desbalanceo.

```
# Crear el modelo balanceado
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

# Ejecutar el modelo balanceado
model = run_model_balanced(X_train, X_test, y_train, y_test)
pred_y = model.predict(X_test)
mostrar_resultados(y_test, pred_y)
```

7. **Aplicación de NearMiss (Undersampling)**
   Se utiliza la técnica NearMiss para reducir la cantidad de muestras de

la clase mayoritaria y se evalúa el modelo entrenado con los datos re-balanceados.

```
# Aplicar NearMiss (undersampling)
us = NearMiss(ratio=0.5, n_neighbors=3, version=2, random_state=1)
X_train_res, y_train_res = us.fit_resample(X_train, y_train)

print("Distribution before resampling {}".format(Counter(y_train)))
print("Distribution after resampling {}".format(Counter(y_train_res)))

# Ejecutar el modelo con el dataset balanceado por NearMiss
model = run_model(X_train_res, X_test, y_train_res, y_test)
pred_y = model.predict(X_test)
mostrar_resultados(y_test, pred_y)
```

8. **Aplicación de RandomOverSampler (Oversampling)**
   Se aplica RandomOverSampler para aumentar la cantidad de muestras de la clase minoritaria y se evalúa el modelo.

```
# Aplicar RandomOverSampler (oversampling)
os = RandomOverSampler(sampling_strategy=0.5)
X_train_res, y_train_res = os.fit_resample(X_train, y_train)

print("Distribution before resampling {}".format(Counter(y_train)))
print("Distribution after resampling {}".format(Counter(y_train_res)))

# Ejecutar el modelo con el dataset balanceado por RandomOverSampler
model = run_model(X_train_res, X_test, y_train_res, y_test)
pred_y = model.predict(X_test)
mostrar_resultados(y_test, pred_y)
```

9. **Aplicación de SMOTETomek (Combinación de Oversampling y Undersampling)**
   Se utiliza SMOTETomek para re-balancear el conjunto de entrenamiento combinando ambas técnicas, y se evalúa el modelo.

```
# Aplicar SMOTETomek
os_us = SMOTETomek(sampling_strategy=0.5)
X_train_res, y_train_res = os_us.fit_resample(X_train, y_train)

print("Distribution before resampling {}".format(Counter(y_train)))
print("Distribution after resampling {}".format(Counter(y_train_res)))

# Ejecutar el modelo con el dataset balanceado por SMOTETomek
```

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

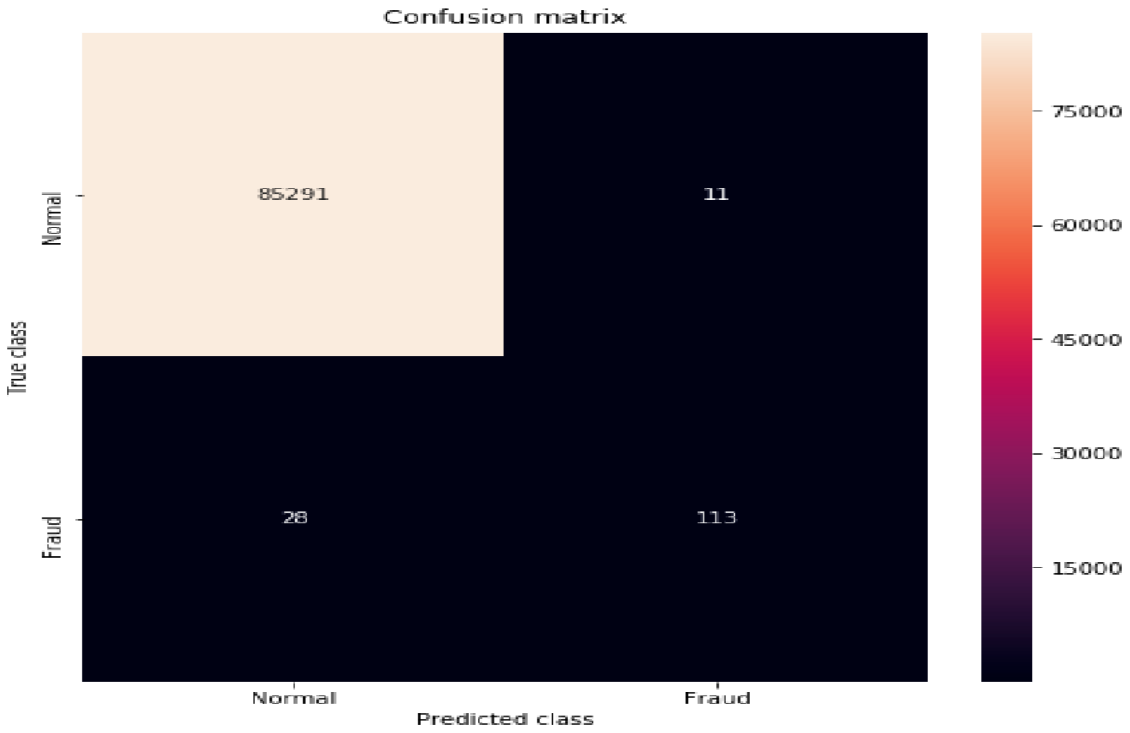
10. **Uso de BalancedBaggingClassifier**
    Se entrena un clasificador de ensamble utilizando BalancedBaggingClassifier con un árbol de decisión como estimador base.

```
# Crear el BalancedBaggingClassifier
bbc = BalancedBaggingClassifier(base_estimator=DecisionTreeClassifier(),
                                sampling_strategy='auto',
                                replacement=False, random_state=0)

# Entrenar el clasificador
bbc.fit(X_train, y_train)
pred_y = bbc.predict(X_test)
mostrar_resultados(y_test, pred_y)
```

11. **Entrenamiento de un Modelo Random Forest**
    Finalmente, se entrena un modelo de Random Forest utilizando 100 árboles.

```
from sklearn.ensemble import RandomForestClassifier

# Crear el modelo con 100 \'arboles
model = RandomForestClassifier(n_estimators=100,
                               bootstrap=True, verbose=2,
                               max_features='sqrt')
# Entrenar el modelo
model.fit(X_train, y_train)
```

3 Resultados



4 Conclusión

El Random Forest es un modelo rápido y sencillo que, aunque pierde la interpretabilidad de un único árbol de decisión, ofrece la ventaja de evitar el overfitting y proporcionar un clasificador más robusto. Existen varios algoritmos basados en árboles (Tree-Based) que mejoran la idea del árbol de decisión mediante técnicas y enfoques de ensamble. La actividad demuestra cómo manejar el desbalanceo de clases en modelos de clasificación utilizando técnicas como NearMiss, RandomOverSampler, SMOTETomek y BalancedBaggingClassifier. Estas mejoran la precisión al equilibrar las clases, y la regresión logística balanceada también muestra buenos resultados en la predicción.