Asignatura	Datos del alumno	Fecha
Machine Learning	**Apellidos:** Avilés Cahill	27/02/2024
	Nombre: Adán	

Laboratorio: Resolver un problema de clasificación

Índice

Asignatura	Datos del alumno	Fecha
Machine Learning	**Apellidos:** Avilés Cahill	27/02/2024
	Nombre: Adán	

1. Introducción

En esta actividad, vamos a resolver un problema de clasificación, creando primero un dataset ficticio.

2. Creación del dataset

El primer paso consiste en crear ese dataset, en el cual emplearemos el DNI para tener un conjunto de datos distintos. Para que los dataset sean comparables entre todos:

- Si el número de identidad tiene menos de 8 cifras replicaremos las primeras hasta obtener exactamente 8 (de forma manual)
- Si alguna de las cifras es menor que 2 la sustituiremos por ese número (automatizado)

. Crearemos una clase que nos transforme el DNI en la forma adecuada (fallando si no contiene 8 dígitos o si es un entero en vez de una string) y a partir de él, se creará el dataset. En el anexo 7.1 se puede encontrar todos los imports necesarios.

```
class DatasetGenerator:
    def __init__(self,
                 dni: str):
        self.dni = dni
        self._check_dni()
        self._prepare_dni()
        self.dataset = None

    def _check_dni(self):
        """
        Check if DNI is well defined
        Returns: The error if is not

        """
        if not isinstance(self.dni, str):
```

```
            raise TypeError('DNI is not an string')
        if not self.dni.isnumeric():
            raise ValueError('DNI has characters that are not numbers')
        if len(self.dni) != 8:
            raise ValueError('DNI has not 8 digits')

    def _prepare_dni(self):
        """
        Change the 0 and 1 to 2
        Returns:

        """
        self.dni.replace('1', '2').replace('0', '2')

    def _create_dataset(self):
        X, y = make_classification(n_samples=200+10*int(self.dni[0]),
                                   n_features=10+int(self.dni[1])+int(self.dni[2]),
                                   n_informative=10+int(self.dni[1]),
                                   shuffle=False,
                                   random_state=int(self.dni))
        list_of_features = [
            f'feature_{number}' for number in
            range(0, 10+int(self.dni[1])+int(self.dni[2]))
        ]
        dataset = pd.DataFrame(X, columns=list_of_features)
        dataset['target'] = y
        self.dataset = dataset
```

```
    def get_clean_dni(self):
        return self.dni

    def get_dataset(self):
        if self.dataset is None:
            self._create_dataset()
        return self.dataset
```

Después simplemente hemos de importar la clase donde la necesitemos y ejecutar

```
dataset = DatasetGenerator(dni='48778094').get_dataset()
```

3. Preparación y EDA

3.1. Preparación

Separaremos en primer lugar el dataset en dos sub datasets, train y test. El train es el que usaremos durante el resto de la práctica mientras que el test lo dejaremos intacto hasta el final.

```
dataset_features = dataset.drop('target', axis=1)
target = dataset[['target']]
# Split the dataset into a training set and a test set
X_train, X_test, y_train, y_test = train_test_split(dataset_features,
                                                    target,
                                                    train_size=200,
                                                    random_state=48778094)
```

3.2. EDA

Hemos ahora de aplicar los métodos *info()* y *describe()* para entender un poco más nuestro dataset.

```
X_train.info()
# <class 'pandas.core.frame.DataFrame'>
# Int64Index: 200 entries, 180 to 118
# Data columns (total 25 columns):
#  #   Column      Non-Null Count  Dtype
# ---  ------      --------------  -----
#  0   feature_0   200 non-null    float64
#  1   feature_1   200 non-null    float64
#  2   feature_2   200 non-null    float64
#                     ...
```

Podemos observar que no tenemos missings en ninguna de las variables, y que ninguna es categórica, con lo cual no tendremos problema alguno, ni que imputar missings o tratar variables. Pasemos a ver la información que nos brinda describe()

```
X_train.describe()
#          feature_0   feature_1   feature_2  ...  feature_23  feature_24
# count  200.000000  200.000000  200.000000  ...  200.000000  200.000000
# mean     0.796240    0.697819    0.487957  ...    0.057566   -0.156741
# std      2.423918    2.490853    2.386239  ...    1.025249    0.998709
# min     -5.139975   -5.449480   -5.034585  ...   -2.377526   -3.080426
# 25%     -0.824737   -1.077084   -1.172061  ...   -0.761771   -0.808426
# 50%      0.880678    0.795573    0.272227  ...    0.053310   -0.132491
# 75%      2.655636    2.361335    2.097005  ...    0.855447    0.501123
# max      6.806898    7.344300    8.096904  ...    3.171954    2.449390
```

Podemos notar que las features tienen todas una media cercana al 0 y una desviación típica de 1, haciendo que en general los valores oteen entre -3 y 3 (aunque las primeras features no cumplen esto, hablamos en forma general). Por otro lado, el target no se comporta como ellas, aunque en este

caso, los árboles de decisión son resilientes ante medidas dispares, así que no hemos de estandarizar los datos.

3.3. Histograma y correlación

Realizamos el histograma ahora de las variables y el target.

```
df_train.hist(figsize = (16,18))
plt.show()
```

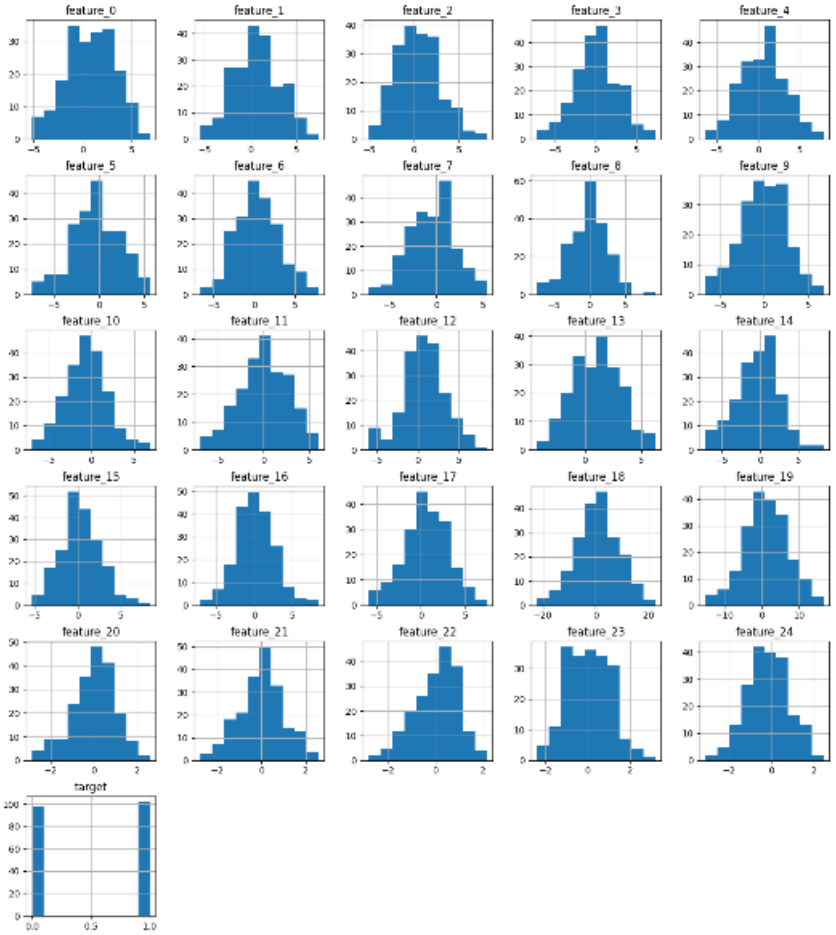

Figura 1: Histograma

Que nos devuelve el histograma entero (que también hemos hecho variable por variable para ver su distribución). Podemos ver como el target claramente es binario.

4. Clasificación por árbol de decisión

Haremos ahora las dos clasificaciones por árboles de decisión.

4.1. Árbol con hiperparámetros default

Comprobaremos, haciendo una regresión simple y usando el paquete **statsmodels** si todas las variables son significativas.

```
clf = DecisionTreeClassifier(random_state=48778094)
clf = clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
y_prob_pred = clf.predict_proba(X_test)[:, 1]
print(roc_auc_score(y_test, y_prob_pred))
```

He usado tanto la accuracy como el AUC para medir cómo de bueno es el modelo (en general no me gusta usar la accuracy como métrica porque si tenemos un dataset desbalanceado, no será una buena métrica). Igualmente en este caso obtenemos un AUC de 0.701.

4.2. Árbol con maxdepth=3

Por defecto el DecissionTreeClasifier no tiene límite de profundidad, al setear el max depth a 3, estamos haciendo que nuestro árbol sea mucho más pequeño. Sin embargo, esto no tiene por qué ser malo, pues evita el overfitting a los datos del train.

```
clf3 = DecisionTreeClassifier(random_state=48778094, max_depth=3)
```

En este caso, tanto la accuracy como el AUC en el test han mejorado. ¿Pero qué pasa si miramos en el train?

```
print(accuracy_score(y_train, clf.predict(X_train)))
print(accuracy_score(y_train, clf3.predict(X_train)))
```

Podremos ver como el primer árbol, tiene una accuracy perfecta ante el train, habiendo overfiteado los datos, sin embargo el segundo árbol tiene una accuracy de 0.845 en el train. Como podemos observar, mejor resultadod en el train, no implica mejor resultado en el test.

5. Métodos de ensemble

Ahora, aplicaremos métodas de ensemble learning al mismo conjunto de datos.

5.1. Bagging

El método de bagging utiliza árboles con reemplazamiento de los datos, como ya sabemos.

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=48778094),
    bootstrap=True)
bag_clf = bag_clf.fit(X_train, y_train)
y_prob_pred = bag_clf.predict_proba(X_test)[:, 1]
print(roc_auc_score(y_test, y_prob_pred))
```

Con este método, obtenemos un AUC de 0.899, el mejor hasta ahora. Veamos qué ocurre con el pasting.

5.2. Pasting

El método de bagging utiliza árboles sin reemplazamiento de los datos.

```
past_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=48778094),
    bootstrap=False)
# ajustar el modelo-----------------------------------------------
past_clf = past_clf.fit(X_train, y_train)
y_prob_pred = past_clf.predict_proba(X_test)[:, 1]
print(roc_auc_score(y_test, y_prob_pred))
```

Con este método, obtenemos un AUC de 0.68, bastante peor que con el bagging, aunque hemos de tener en cuenta que el pasting puede ser más eficaz que el bagging en ciertos escenarios en los que

es importante evitar el sobreajuste y tener muestras de entrenamiento únicas y de tamaño fijo. Que el pasting haya tenido un peor resultado puede deberse al tener solo 200 datos.

5.3. Random Forest

Los Random Forest, a diferencia de los métodos anteriores, eligen sobre un subset de las variables favoreciendo también a evitar el overfitting o evitar que si una variable tiene data leakage, aparezca siempre, tomando también árboles más débiles y menos correlacionados.

```
rnd_clf = RandomForestClassifier(max_leaf_nodes=4,
                                 random_state=48778094,)
rnd_clf = rnd_clf.fit(X_train, y_train)
y_prob_pred = rnd_clf.predict_proba(X_test)[:, 1]
print(roc_auc_score(y_test, y_prob_pred))
```

En este caso, el Random Forest obtiene un AUC de 0,92, superando a los otros métodos.

5.4. Gradient Boosting

A diferencia de los tres métodos anteriores, los modelos de boosting construyen un modelo aditivo en forma de un conjunto de árboles de decisión, donde cada árbol se ajusta a los errores residuales del modelo anterior.

```
gb_clf = GradientBoostingClassifier(random_state=48778094)
gb_clf = gb_clf.fit(X_train, y_train)
y_prob_pred = gb_clf.predict_proba(X_test)[:, 1]
print(roc_auc_score(y_test, y_prob_pred))
```

Dándonos el mejor AUC, de 0.95, un score que ya quisiera yo ver en un entorno de producción. Dicho sea, que hayamos obtenido mejores resultados con él, no implica que sea mejor. Dependiendo del caso de uso, RandomForest puede ser mucho más óptimo.

6. Feature importance de los modelos

Veamos la feature importance de los modelos. Como Bagging no tiene feature importance implementada, hemos usado la media de sus árboles.

```
importancias_clf = clf.feature_importances_
    indices_clf = np.argsort(importancias_clf)[::-1]

    importancias_clf3 = clf3.feature_importances_
    indices_clf3 = np.argsort(importancias_clf3)[::-1]

    importancias_bag_clf = np.mean([
        tree.feature_importances_ for tree in bag_clf.estimators_
    ], axis=0)
    indices_bag_clf = np.argsort(importancias_bag_clf)[::-1]

    importancias_past_clf=  np.mean([
        tree.feature_importances_ for tree in past_clf.estimators_
    ], axis=0)
    indices_past_clf = np.argsort(importancias_past_clf)[::-1]

    importancias_rnd_clf = rnd_clf.feature_importances_
    indices_rnd_clf = np.argsort(importancias_rnd_clf)[::-1]

    importancias_gb_clf = gb_clf.feature_importances_
    indices_gb_clf = np.argsort(importancias_gb_clf)[::-1]

    plt.figure(figsize=(12, 13))
    plt.subplot(6, 1, 1)
    plt.title("Importancia de las variables Arbol Decision")
```

```
plt.bar(range(X_train.shape[1]), importancias_clf[indices_clf],
        color="r", align="center")
plt.xticks(range(X_train.shape[1]), indices_clf)
plt.xlim([-1, X_train.shape[1]])

plt.subplot(6, 1, 2)
plt.title("Importancia de las variables Arbol Decision depth3")
plt.bar(range(X_train.shape[1]), importancias_clf3[indices_clf3],
        color="r", align="center")
plt.xticks(range(X_train.shape[1]), indices_clf3)
plt.xlim([-1, X_train.shape[1]])

plt.subplot(6, 1, 3)
plt.title("Importancia de las variables Bagging")
plt.bar(range(X_train.shape[1]), importancias_bag_clf[indices_bag_clf],
        color="r", align="center")
plt.xticks(range(X_train.shape[1]), indices_bag_clf)
plt.xlim([-1, X_train.shape[1]])

plt.subplot(6, 1, 4)
plt.title("Importancia de las variables Pasting")
plt.bar(range(X_train.shape[1]), importancias_past_clf[indices_past_clf],
        color="r", align="center")
plt.xticks(range(X_train.shape[1]), indices_past_clf)
plt.xlim([-1, X_train.shape[1]])

plt.subplot(6, 1, 5)
plt.title("Importancia de las variables RandomForest")
```

```
plt.bar(range(X_train.shape[1]), importancias_rnd_clf[indices_rnd_clf],
        color="r", align="center")
plt.xticks(range(X_train.shape[1]), indices_rnd_clf)
plt.xlim([-1, X_train.shape[1]])

plt.subplot(6, 1, 6)
plt.title("Importancia de las variables GradientBoosting")
plt.bar(range(X_train.shape[1]), importancias_gb_clf[indices_gb_clf],
        color="r", align="center")
plt.xticks(range(X_train.shape[1]), indices_gb_clf)
plt.xlim([-1, X_train.shape[1]])
```

Que nos da como resultado el siguiente plot

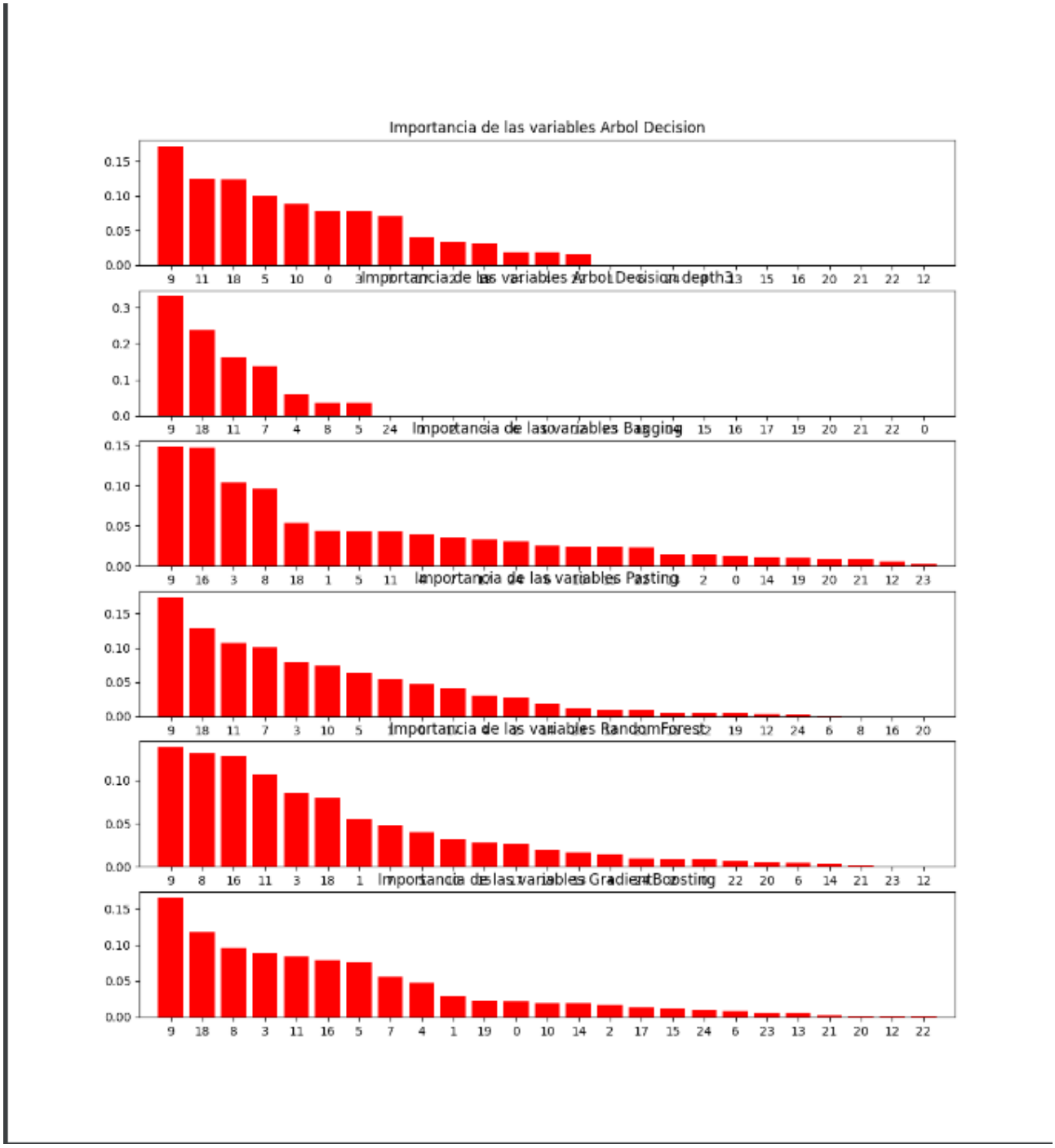

Figura 2: Histograma

En él podemos comprobar como la variable 9 siempre es la que más peso tiene. La segunda variable no tiene por qué coincidir en cada modelo, pero sí que es interesante ver como las variables 0, 15,

20, 21, 22 y 12 suelen tener las peores posiciones. Notar también que debido a la naturaleza de coger la mejor variable para el split de los árboles de decisión, hay variables que no aparecen ni una sola vez. Teniendo en cuenta que la importancia de la variable se calcula en función de la cantidad de veces que se utiliza dicha característica para dividir los nodos de los árboles de decisión que conforman el modelo, podemos decir que la variable 9, en general contribuye con un 15 por ceinto a la mejora del modelo. Haciendo que, aunque tengamos 25 variables, pudieramos obtener los mismos resultados con un subconjunto de estas más pequeño. Cabe destacar también que una importancia de 0 no necesariamente significa que la característica no tiene ningún valor predictivo, sino que en el contexto del modelo en particular, esa característica no es útil para mejorar la precisión de la predicción.

7. Anexos: Código

7.1. Lista de imports

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.ensemble import (BaggingClassifier, RandomForestClassifier,
GradientBoostingClassifier)
from sklearn.model_selection import (train_test_split )

from Machine_Learning.practica_2.dataset_generator import DatasetGenerator
```

8. Bibilografia

Referencias

[1] Apuntes de la asignatura de MachineLearning

[2] Documentación Oficial de Scikit-Learn

[3] Documentación Oficial de StatsModels