MACHINE LEARNING

EDA
AND
CLASSIFICATION

Author

Sina Heydari

Winter 2024

Table Of Contents

1 Introduction **3**
- 1.1 What Is Machine Learning ? ... 3
- 1.2 Three Main Categories Of Machine Learning ... 4
- 1.3 What Is Exploratory Data Analysis ? ... 5
- 1.4 Why Exploratory Data Analysis Is Important ? ... 5
- 1.5 The Foremost Goals of EDA ... 5
- 1.6 Types of EDA ... 6

2 Abstract **9**
- 2.1 The Problem ... 9
- 2.2 The Data ... 9
- 2.3 Importing Libraries ... 9

3 The Dataset **12**
- 3.1 Loading The Data ... 12
- 3.2 Data Preprocessing ... 13
  - 3.2.1 Filling Missing Values In 'Sale' Column ... 14
  - 3.2.2 Filling Missing Values In 'product price' Column ... 15
  - 3.2.3 Filling Missing Values In 'product country' Column ... 17
  - 3.2.4 Filling Missing Values In 'product price' Column Based On 'product country' Column With Median ... 19

3.2.5 Filling Missing Values In 'product price' Column Based On 'audience id' Column With Median 22
3.3 Handling Time Information 25
3.4 Cleaning The Dataset 28

4 Training The Models **30**
4.1 The Models We Used 30
4.1.1 Decision Tree 30
4.1.2 Implementation 31
4.1.3 Logistic Regression 36
4.1.4 Implementation 37
4.1.5 SVC (Support Vector Classifier) 41
4.1.6 Implementation 42
4.1.7 XGBoost 44
4.1.8 Implementation 44
4.1.9 Random Forest 48
4.1.10 Implementation 49
4.1.11 k-Nearest Neighbors 52
4.1.12 Implementation 53

5 References **56**

1 Introduction

1.1 What Is Machine Learning ?

Machine learning which we will be referring as(ML), is a branch of artificial intelligence (AI) that focuses on the development of algorithms and statistical models that enable computers to learn and improve their performance on a specific task without being explicitly programmed. The primary goal of machine learning is to enable computers to learn from data and make predictions or decisions based on that data.

Data Collection:

The ML algorithms require data to learn from. This data can be structured (e.g., tabular data) or unstructured (e.g., text, images, audio). The quality and quantity of data play a crucial role in the effectiveness of the machine learning model.

Feature Extraction and Selection:

Features are specific pieces of information or characteristics derived from the data that are relevant to the task at hand. In this step, relevant features are extracted from the raw data, and unnecessary or redundant features may be removed to simplify the model and improve its performance.

Model Training:

During the training phase, the machine learning algorithm is presented with labeled examples from the data set. The algorithm learns patterns and relationships in the data by adjusting its internal parameters to minimize the difference between its predictions and the actual labels or outcomes.

Evaluation:

Once the model is trained, it is evaluated using a separate dataset that it hasn't seen before (the test set). Evaluation metrics such as accuracy, precision, recall, or mean squared error are used to assess the performance of the model and determine how well it generalizes to new, unseen data.

Model Deployment:

If the model performs satisfactorily during evaluation, it can be deployed to make predictions or decisions on new, unseen data. Deployment may involve integrating the model into software applications, web services, or other systems where it can be used to automate tasks or provide insights.

1.2 Three Main Categories Of Machine Learning

Supervised Learning:

In supervised learning, the algorithm is trained on labeled data, where each example is associated with a corresponding label or outcome. The goal is to learn a mapping from input features to output labels, enabling the algorithm to make predictions on new data.

Unsupervised Learning:

Unsupervised learning involves training the algorithm on unlabeled data, where the goal is to discover hidden patterns or structures in the data. Clustering, dimensionality reduction, and association rule learning are common tasks in unsupervised learning.

Reinforcement Learning:

Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties based on its actions, and its goal is to learn a policy that maximizes cumulative rewards over time.

Conclusively, machine learning enables computers to learn from data and make intelligent decisions or predictions, making it a powerful tool for a wide range of applications in various fields, including finance, health care, transportation, and more.

1.3 What Is Exploratory Data Analysis ?

Exploratory data analysis (EDA) is used by data scientists to analyze and investigate data sets and summarize their main characteristics, often employing data visualization methods.

EDA helps determine how best to manipulate data sources to get the answers you need, making it easier for data scientists to discover patterns, spot anomalies, test a hypothesis, or check assumptions.

EDA is primarily used to see what data can reveal beyond the formal modeling or hypothesis testing task and provides a provides a better understanding of data set variables and the relationships between them. It can also help determine if the statistical techniques you are considering for data analysis are appropriate. Originally developed by American mathematician John Tukey in the 1970s, EDA techniques continue to be a widely used method in the data discovery process today.

1.4 Why Exploratory Data Analysis Is Important ?

The main purpose of EDA is to help look at data before making any assumptions. It can help identify obvious errors, as well as better understand patterns within the data, detect outliers or anomalous events, find interesting relations among the variables.

Data scientists can use exploratory analysis to ensure the results they produce are valid and applicable to any desired business outcomes and goals. EDA also helps stakeholders by confirming they are asking the right questions. EDA can help answer questions about standard deviations, categorical variables, and confidence intervals. Once EDA is complete and insights are drawn, its features can then be used for more sophisticated data analysis or modeling, including machine learning.

1.5 The Foremost Goals of EDA

1. Data Cleaning:

EDA involves examining the information for errors, lacking values, and inconsistencies. It includes techniques including records imputation, managing missing statistics, and figuring out and getting rid of outliers.

2. Descriptive Statistics:

EDA utilizes precise records to recognize the important tendency, variability, and distribution of variables. Measures like suggest, median, mode, preferred deviation, range, and percentiles are usually used.

3. Data Visualization:

EDA employs visual techniques to represent the statistics graphically. Visualizations consisting of histograms, box plots, scatter plots, line plots, heatmaps, and bar charts assist in identifying styles, trends, and relationships within the facts.

4. Feature Engineering:

EDA allows for the exploration of various variables and their adjustments to create new functions or derive meaningful insights. Feature engineering can contain scaling, normalization, binning, encoding express variables, and creating interplay or derived variables.

5. Correlation and Relationships:

EDA allows discover relationships and dependencies between variables. Techniques such as correlation analysis, scatter plots, and pass-tabulations offer insights into the power and direction of relationships between variables.

6. Data Segmentation:

EDA can contain dividing the information into significant segments based totally on sure standards or traits. This segmentation allows advantage insights into unique subgroups inside the information and might cause extra focused analysis.

7. Hypothesis Generation:

EDA aids in generating hypotheses or studies questions based totally on the preliminary exploration of the data. It facilitates form the inspiration for in addition evaluation and model building.

8. Data Quality Assessment:

EDA permits for assessing the nice and reliability of the information. It involves checking for records integrity, consistency, and accuracy to make certain the information is suitable for analysis.

1.6 Types of EDA

Depending on the number of columns we are analyzing we can divide EDA into two types.

1. Univariate Analysis:

This sort of evaluation makes a speciality of analyzing character variables inside the records set. It involves summarizing and visualizing a unmarried variable at a time to understand its distribution, relevant tendency, unfold, and different applicable records. Techniques like histograms, field plots, bar charts, and precis

information are generally used in univariate analysis.

2. Bivariate Analysis:

Bivariate evaluation involves exploring the connection between variables. It enables find associations, correlations, and dependencies between pairs of variables. Scatter plots, line plots, correlation matrices, and move-tabulation are generally used strategies in bivariate analysis.

3. Multivariate Analysis:

Multivariate analysis extends bivariate evaluation to encompass greater than variables. It ambitions to apprehend the complex interactions and dependencies among more than one variables in a records set. Techniques inclusive of heatmaps, parallel coordinates, aspect analysis, and primary component analysis (PCA) are used for multivariate analysis.

4. Time Series Analysis:

This type of analysis is mainly applied to statistics sets that have a temporal component. Time collection evaluation entails inspecting and modeling styles, traits, and seasonality inside the statistics through the years. Techniques like line plots, autocorrelation analysis, transferring averages, and ARIMA (AutoRegressive Integrated Moving Average) fashions are generally utilized in time series analysis.

5. Missing Data Analysis:

Missing information is a not unusual issue in datasets, and it may impact the reliability and validity of the evaluation. Missing statistics analysis includes figuring out missing values, know-how the patterns of missingness, and using suitable techniques to deal with missing data. Techniques along with lacking facts styles, imputation strategies, and sensitivity evaluation are employed in lacking facts evaluation.

6. Outlier Analysis:

Outliers are statistics factors that drastically deviate from the general sample of the facts. Outlier analysis includes identifying and knowledge the presence of outliers, their capability reasons, and their impact at the analysis. Techniques along with box plots, scatter plots, z-rankings, and clustering algorithms are used for outlier evaluation.

7. Data Visualization:

Data visualization is a critical factor of EDA that entails creating visible representations of the statistics to facilitate understanding and exploration. Various

visualization techniques, inclusive of bar charts, histograms, scatter plots, line plots, heatmaps, and interactive dashboards, are used to represent exclusive kinds of statistics.

These are just a few examples of the types of EDA techniques that can be employed at some stage in information evaluation. The choice of strategies relies upon on the information traits, research questions, and the insights sought from the analysis.

2 Abstract

Today, predicting users' reactions to commercial advertisements has become a significant issue, attracting considerable investment. Predicting how users will react to each of our advertisements will enable us to reduce advertising costs or increase the effectiveness of each advertisement. In this project, we intend to tackle one of the fundamental issues in this field.

2.1 The Problem

We intend to predict whether clicking on an internet advertisement ultimately leads to a purchase or not, given certain features of the advertisement, using machine learning. Note that we know not every displayed advertisement is clicked on, and we aim to distinguish between clicked advertisements resulting in purchases or not, which is the essence of the seller's profit. Essentially, the input to our model is a row of features of an internet advertisement, and the desired output is our model's prediction of whether the product is purchased after clicking on a specific advertisement.

2.2 The Data

In this project, we use Critco live data, which contains information about advertisements for various products over a period of 90 days. Each row of this data contains information about a click made on these advertisements. These details include whether the click led to the purchase of the product, the time elapsed between clicking on the advertisement and purchasing the product, user characteristics, and product attributes.

2.3 Importing Libraries

For the start, we import necessary libraries for data analysis, machine learning modeling, and visualization.

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import xgboost as xgb
from imblearn.under_sampling import RandomUnderSampler
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
```

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.preprocessing import StandardScaler,
 OneHotEncoder
from sklearn.metrics import accuracy_score,
 classification_report, confusion_matrix, roc_auc_score,
 roc_curve, f1_score
from sklearn.svm import SVC
from datetime import datetime, timedelta


```

necessary libraries

numpy (np): NumPy is a fundamental package for scientific computing with Python. It provides support for mathematical functions to operate on arrays and matrices. It's widely used for numerical operations and data manipulation.

pandas (pd): Pandas is a powerful library for data manipulation and analysis. It offers data structures like DataFrame and Series, which are highly efficient for handling structured data. Pandas is commonly used for data preprocessing and exploratory data analysis (EDA).

matplotlib.pyplot (plt): Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. The pyplot module of Matplotlib provides a MATLAB-like interface for creating plots and visualizations.

xgboost: XGBoost is an optimized distributed gradient boosting library designed for efficiency and performance. It's widely used for supervised learning tasks, particularly in machine learning competitions and data science projects.

imblearn.under_sampling.RandomUnderSampler: Imbalanced-learn is a Python package offering various strategies for dealing with imbalanced datasets. RandomUnderSampler is used here for under-sampling the majority class to handle class imbalance in classification tasks.

sklearn.model_selection.train_test_split: Scikit-learn (sklearn) is a popular machine learning library in Python. The train_test_split function is used to split datasets into training and testing sets for model evaluation and

validation.

sklearn.tree.DecisionTreeClassifier: DecisionTreeClassifier is a classifier based on decision trees. It's used for classification tasks and is part of the scikit-learn library.

sklearn.preprocessing.OneHotEncoder: OneHotEncoder is used for converting categorical variables into binary vectors.

sklearn.compose.ColumnTransformer: ColumnTransformer allows for applying different transformations to different columns of a dataset. It's commonly used in preprocessing pipelines.

sklearn.pipeline.Pipeline: Pipeline is used to chain multiple processing steps together, such as preprocessing and modeling, into a single object.

sklearn.impute.SimpleImputer: SimpleImputer is used for imputing missing values in datasets.

sklearn.linear_model.LogisticRegression: LogisticRegression is a linear model for binary classification tasks.

sklearn.preprocessing.StandardScaler: StandardScaler is used for standardizing features by removing the mean and scaling to unit variance.

sklearn.metrics: This imports various metrics for evaluating machine learning models, such as accuracy_score, classification_report, confusion_matrix, roc_auc_score, roc_curve, and f1_score.

sklearn.svm.SVC: SVC stands for Support Vector Classifier, which is a type of support vector machine for classification tasks.

datetime.datetime, datetime.timedelta: These are modules for working with dates and times in Python. They are used here for handling time-related information.

3 The Dataset

3.1 Loading The Data

Here we load the data from a CSV file named 'data.csv' into a pandas DataFrame named Dataframe, ensuring that the entire file is read into memory at once for better performance.

```
# Read data from CSV into a pandas Dataframe
Dataframe = pd.read_csv('data.csv', low_memory=False)

```

Loading The Data

pd.read_csv('data.csv', low_memory=False): This function is from the pandas library (pd) and is used to read data from a CSV file. It takes several parameters:

'data.csv': This is the name of the CSV file from which the data will be read.

low_memory=False: This parameter is set to False to ensure that pandas reads the entire file into memory at once rather than trying to determine the data types of each column based on a small sample of the data. This can be useful for large datasets where memory usage may be a concern.

Dataframe: This is the variable name assigned to the pandas DataFrame that will store the data read from the CSV file.

3.2 Data Preprocessing

Here removes the specified columns from the DataFrame, eliminating the columns deemed unnecessary for further analysis or modeling.

Due to the fact that, we found out the category columns were not thoroughly encoded and prduct_age_group, product_gender, product_brand, mostly were not filled with the data.

```
# Drop unnecessary columns
Dataframe = Dataframe.drop(columns=['Unnamed: 0',
                                    ' product_age_group ',
                                    '  product_gender   ',
                                    '   product_brand   ',
                                    ' product_category1 ',
                                    ' product_category2 ',
                                    ' product_category3 ',
                                    ' product_category4 ',
                                    ' product_category5 ',
                                    ' product_category6 ',
                                    'product_category7'],axis
 =1)

```

Dropping Unnecessary Columns

columns: This parameter specifies the list of column names to be dropped from the DataFrame. In this case, it includes columns such as 'Unnamed: 0', 'product_age_group', 'product_gender', and so on.

axis=1: This parameter indicates that the operation should be performed along the columns (axis=1). Dropping columns means removing them from the DataFrame.

Dataframe: After dropping the specified columns, the modified DataFrame is assigned back to the variable Dataframe, overwriting the original DataFrame.

3.2.1 Filling Missing Values In 'Sale' Column

We found out that there are missing values in 'Sale' column, So we ensured that missing values in the 'Sale' column are filled based on the value of 'SalesAmountInEuro' column, with 0.0 indicating no sale and 1.0 indicating a sale.

```
# Define a function to fill missing values in 'Sale' column based on 'SalesAmountInEuro'
def fill_sale_column(row):
    if pd.isna(row['Sale']):
        if row['SalesAmountInEuro'] == -1:
            return 0.0
        else:
            return 1.0
    else:
        return row['Sale']

# Apply the function to fill missing values in 'Sale' column
Dataframe['Sale'] = Dataframe.apply(fill_sale_column, axis=1)

```

Filling Misiing Values In 'Sale' column

We defined a function fill_sale_column() to fill missing values in the 'Sale' column of the Dataframe based on the 'SalesAmountInEuro' column.

def fill_sale_column(row): This line defines a function named fill_sale_column that takes a row from the DataFrame as input.

if pd.isna(row['Sale']): This condition checks if the value in the 'Sale' column of the current row is missing (NaN).

if row['SalesAmountInEuro'] == -1: This nested condition checks if the value in the 'SalesAmountInEuro' column of the current row is equal to -1.

return 0.0: If both conditions are true, meaning 'Sale' is missing and 'SalesAmountInEuro' is -1, the function returns 0.0, indicating that there was no sale.

else: return 1.0: If the 'Sale' value is missing but 'SalesAmountInEuro' is not -1, the function returns 1.0, indicating that there was a sale.

else: return row['Sale']: If the 'Sale' value is not missing, the function returns the existing value in the 'Sale' column for that row.

Dataframe['Sale'] = Dataframe.apply(fill_sale_column, axis=1): This line applies the fill_sale_column function to each row of the Dataframe along the axis 1 (rows), and the result is assigned back to the 'Sale' column of the DataFrame. This effectively fills missing values in the 'Sale' column based on the logic defined in the fill_sale_column function.

3.2.2 Filling Missing Values In 'product price' Column

Here we calculate the mean price based on the 'product_id' and fill missing values in the 'product_price' column using this mean price and ensures that missing values in the 'product_price' column are filled based on the mean price calculated for each 'product_id'.

```
  # Calculate mean price based on product_id
mean_price_based_on_product_id = Dataframe[(Dataframe['
    product_price'] != -1) & (Dataframe['product_id'] != '-1'
    )][['product_id','product_price']].groupby(by='product_id'
    ).mean()

# Define a function to fill missing values in 'product_price'
    column based on 'product_id'
def fill_price_based_on_product_id(row):
    if row['product_price'] == -1:
        if row['product_id'] != '-1':
            try:
                return mean_price_based_on_product_id.loc[row
    ['product_id']].item()
            except:
                return -1
        else:
            return -1
    else:
        return row['product_price']

# Apply the function to fill missing values in 'product_price
    ' column
Dataframe['product_price'] = Dataframe.apply(
    fill_price_based_on_product_id, axis=1)

```

Filling Misiing Values In 'product price' column

mean_price_based_on_product_id: This line calculates the mean price for each unique 'product_id' where the 'product_price' is not -1 and the 'product_id' is not '-1'. It uses the groupby function to group the DataFrame by 'product_id' and then calculates the mean price for each group.

def fill_price_based_on_product_id(row): This function is defined to fill missing values in the 'product_price' column based on the 'product_id'. It takes a row from the DataFrame as input.

if row['product_price'] == -1: This condition checks if the value in the 'product_price' column of the current row is -1, indicating a missing value.

if row['product_id'] != '-1': This nested condition checks if the value in the 'product_id' column of the current row is not '-1'. If the 'product_id' is valid, it proceeds to fill the missing 'product_price'.

try: return mean_price_based_on_product_id.loc[row['product_id']].item(): This line attempts to retrieve the mean price corresponding to the 'product_id' from the mean_price_based_on_product_id DataFrame. If successful, it returns the mean price for that 'product_id'.

except: return -1: If an error occurs (e.g., 'product_id' not found in mean_price_based_on_product_id), it returns -1.

else: return -1: If the 'product_id' is '-1' (indicating an invalid value), it returns -1.

else: return row['product_price']: If the 'product_price' is not -1 (i.e., it's not a missing value), it returns the existing 'product_price' value for that row.

Dataframe.apply(fill_price_based_on_product_id, axis=1): This line applies the fill_price_based_on_product_id function to each row of the Dataframe along the axis 1 (rows), and the result is assigned back to the 'product_price' column of the DataFrame. This effectively fills missing values in the 'product_price' column based on the mean price calculated for each 'product_id'.

3.2.3 Filling Missing Values In 'product country' Column

Here we calculate unique product countries per partner_id and fill missing values in the 'product_country' column based on the 'partner_id'.

```
  # Define a function to fill missing values in '
    product_country' column based on 'partner_id'
def fill_country_based_on_partner_id(row):
    if row['product_country'] == '-1':
        if row['partner_id'] != '-1':
            try:
                x = country_partner_uniques_dataframe.loc[row
    ['partner_id']]
                if isinstance(x, np.ndarray):
                    return x[0]
                else:
                    return x
            except:
                return '-1'
        else:
            return '-1'
    else:
        return  row['product_country']

# Apply the function to fill missing values in '
    product_country' column
Dataframe['product_country'] = Dataframe.apply(
    fill_country_based_on_partner_id, axis=1)

```

Filling Missing Values In 'product country' Column

country_partner_uniques_dataframe: This line creates a DataFrame that contains unique product countries per partner_id.

It filters out rows where 'product_country' is not −1, selects the columns 'product_country' and 'partner_id', groups the DataFrame by 'partner_id', and then applies the unique() function to get unique product countries for each partner_id.

def fill_country_based_on_partner_id(row): This function is defined to fill missing values in the 'product_country' column based on the 'partner_id'. It takes a row from the DataFrame as input.

if row['product_country'] == '-1': This condition checks if the value in the 'product_country' column of the current row is 1−, indicating a missing value.

if row['partner_id'] != '-1': This nested condition checks if the value in the 'partner_id' column of the current row is not −1. If the 'partner_id' is valid, it proceeds to fill the missing 'product_country'.

try: x = country_partner_uniques_dataframe.loc[row['partner_id']]: This line attempts to retrieve the unique product countries corresponding to the 'partner_id' from the country_partner_uniques_dataframe.

if isinstance(x, np.ndarray): return x[0]: If the retrieved value is an instance of NumPy array (indicating multiple unique product countries), it returns the first country in the array.

else: return x: If there's only one unique product country for the 'partner_id', it returns that country.

except: return '-1': If an error occurs (e.g., 'partner_id' not found in country_partner_uniques_dataframe), it returns −1.

else: return '-1': If the 'partner_id' is −1 (indicating an invalid value), it returns −1.

Dataframe.apply(fill_country_based_on_partner_id, axis=1): This line applies the fill_country_based_on_partner_id function to each row of the DataFrame along the axis 1 (rows), and the result is assigned back to the 'product_country' column of the DataFrame. This effectively fills missing values in the 'product_country' column based on the unique product countries per 'partner_id'.

3.2.4 Filling Missing Values In 'product price' Column Based On 'product country' Column With Median

For this part, we calculate and visualized statistics related to product prices for different countries, ensuring that missing prices are filled based on the median price of the corresponding country.

```
# Calculate statistics based on product country and price
Dataframe1 = Dataframe[(Dataframe['product_country']!='-1') &
    (Dataframe['product_price']!=-1)][['product_country','
    product_price']]
ag_Dataframe1 = Dataframe1.groupby(by='product_country')

res_table = ag_Dataframe1.mean()
res_table['max'] = ag_Dataframe1.max()['product_price'].
    values
res_table['min'] = ag_Dataframe1.min()['product_price'].
    values
res_table['count'] = ag_Dataframe1.count()['product_price'].
    values
res_table['median'] = ag_Dataframe1.median()['product_price'
    ].values

#Plot histograms to visualize distribution of product prices
    for different countries
Dataframe[(Dataframe['product_country'] == '57
    A1D462A03BD076E029CF9310C11FC5') & (Dataframe['
    product_price']!= -1)]['product_price'].plot(kind='hist',
    bins=100)
plt.show()
Dataframe[(Dataframe['product_country'] == '2
    AC62132FBCFA093B9426894A4BC6278') & (Dataframe['
    product_price']!= -1)]['product_price'].plot(kind='hist',
    bins=100)
plt.show()

# Calculate median price based on product country
median_price_based_on_country = ag_Dataframe1.median()

# Define a function to fill missing values in 'product_price'
    column based on 'product_country'
def fill_price_based_on_product_country(row):
    if row['product_price'] == -1:
        if row['product_country'] != '-1':
            try:
                return median_price_based_on_country.loc[row[
    'product_country']].item()
            except:
                return -1
```

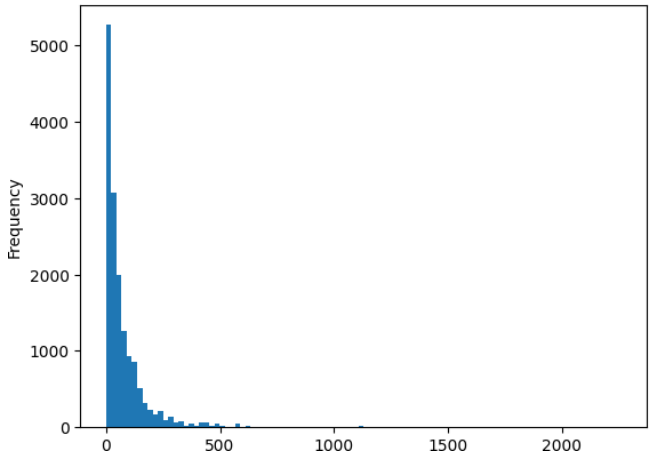
```
        else:
            return -1
    else:
        return row['product_price']

# Apply the function to fill missing values in 'product_price' column
Dataframe['product_price'] = Dataframe.apply(fill_price_based_on_product_country, axis=1)
```

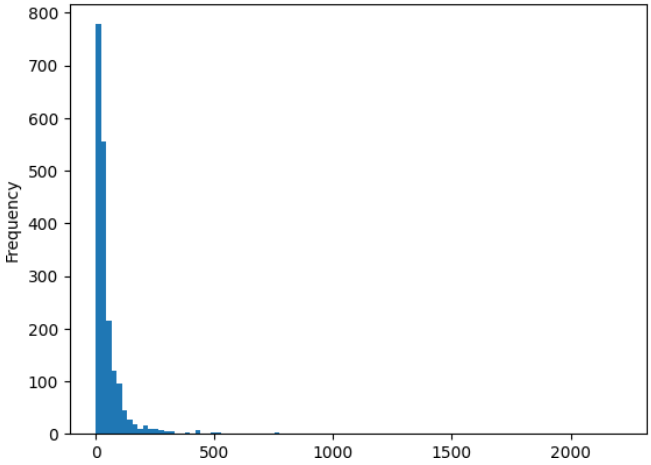
Filling Missing Values In 'product price' Column

Filtering Dataframe and Grouping by 'product_country':It creates a new Dataframe by filtering out rows where 'product_country' is not −1 and 'product_price' is not equal to −1. It selects only the columns 'product_country' and 'product_price'.It then groups this filtered DataFrame by 'product_country' using the groupby() function, resulting in a grouped DataFrame.

Calculating Statistics: It calculates various statistics such as mean, maximum, minimum, count, and median of product prices for each country using the grouped DataFrame. These statistics are stored in a DataFrame called res_table.

Visualizing Distribution of Product Prices: It plots histograms to visualize the distribution of product prices for specific countries. Two histograms are plotted, one for each country. The histograms show the frequency distribution of product prices for the selected countries.





Calculating Median Price Based on Product Country: It calculates the median price for each product country using the grouped DataFrame and stores it in a variable called median_price_based_on_country.

Filling Missing Values in 'product_price' Column Based on 'product_country': It defines a function fill_price_based_on_product_country(row) to fill missing values in the 'product_price' column based on the corresponding 'product_country'. Inside the function, if the 'product_price' is missing (−1) and the 'product_country' is valid (not −1), it fills the missing price with the median price of that country. This function is then applied to each row of the DataFrame using the apply() function along the axis 1 (rows), effectively filling missing values in the 'product_price' column based on the median price of the respective country.

3.2.5 Filling Missing Values In 'product price' Column Based On 'audience id' Column With Median

Here we calculate and visualized statistics related to product prices for different audience IDs, ensuring that missing prices are filled based on the median price of the respective audience ID.

```
  # Calculate statistics based on audience_id and price
Dataframe1 = Dataframe[(Dataframe['audience_id']!='-1') & (
    Dataframe['product_price']!=-1)][['audience_id','
    product_price']]
ag_Dataframe1 = Dataframe1.groupby(by='audience_id')

res_table = ag_Dataframe1.mean()
res_table['max'] = ag_Dataframe1.max()['product_price'].
    values
res_table['min'] = ag_Dataframe1.min()['product_price'].
    values
res_table['count'] = ag_Dataframe1.count()['product_price'].
    values
res_table['median'] = ag_Dataframe1.median()['product_price'
    ].values
print(res_table)

# Plot histograms to visualize distribution of product prices
    for different audience IDs
Dataframe[(Dataframe['audience_id'] == '013
    F1DD80B0848F555FBC82C981E9747') & (Dataframe['
    product_price']!=-1)]['product_price'].plot(kind='hist',
    bins=100)
plt.show()
Dataframe[(Dataframe['audience_id'] == '
    ED37757B4EBDAC15881F1E9B29A35096') & (Dataframe['
    product_price']!=-1)]['product_price'].plot(kind='hist',
    bins=100)
plt.show()

# Calculate median price based on audience ID
median_price_based_on_audience_id = ag_Dataframe1.median()

# Define a function to fill missing values in 'product_price'
    column based on 'audience_id'
def fill_price_na_based_on_audience_id(row):
    if row['product_price'] == -1:
        if row['audience_id'] != '-1':
            try:
                return median_price_based_on_audience_id.loc[
    row['audience_id']].item()
```

```
            except:
                return -1
        else:
            return -1
    else:
        return row['product_price']

# Apply the function to fill missing values in 'product_price' column
x = Dataframe.apply(fill_price_na_based_on_audience_id, axis=1)

```

Filling Missing Values In 'product price'

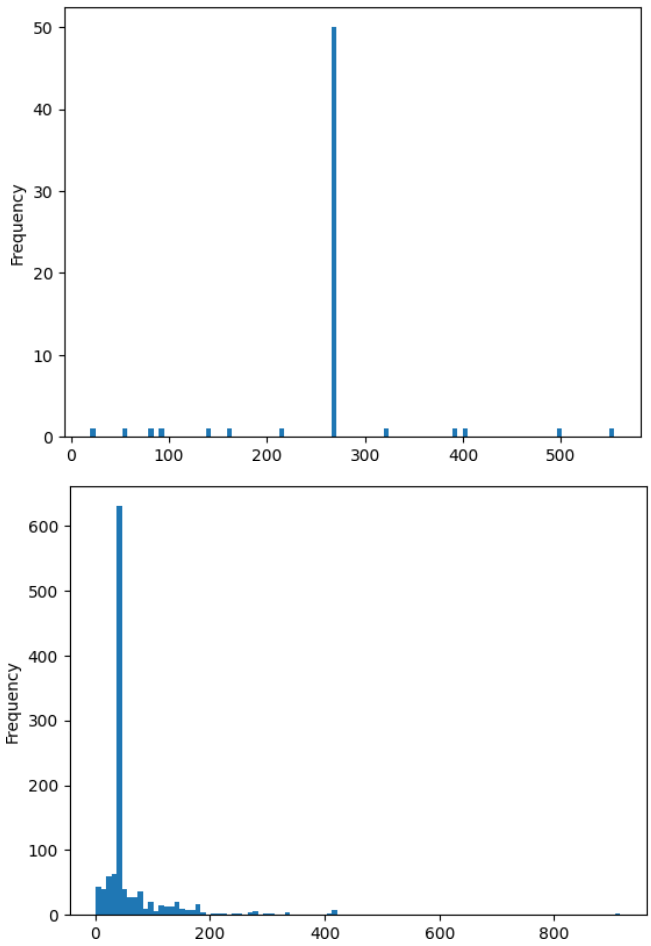
Filtering Dataframe and Grouping by 'audience_id': It creates a new DataFrame by filtering out rows where 'audience_id' is not −1 and 'product_price' is not equal to -1. It selects only the columns 'audience_id' and 'product_price'. It then groups this filtered DataFrame by 'audience_id' using the groupby() function, resulting in a grouped DataFrame.

Calculating Statistics: It calculates various statistics such as mean, maximum, minimum, count, and median of product prices for each audience ID using the grouped DataFrame. These statistics are stored in a DataFrame called res_table.

audience_id	product_price	max	min	count	median
004ADDA473468947099E099CB564585C	57.456667	74.95	29.99	6	64.95
00510B0DD11101C9D91DD8C2838C2AF8	266.510000	266.51	266.51	83	266.51
006AA023F5B71D441617389F20A2AC57	27.480000	27.48	27.48	9	27.48
008084BE5FA06B5CCA3C282840613D14	266.510000	266.51	266.51	4	266.51
00AA443DF7F3418942601C441C68566A	60.000000	65.00	55.00	3	60.00
...	...	...	...	...	...
FEF3C006C7313BCCC1B1599CD78BF7CC	30.000000	35.00	25.00	2	30.00
FEFA5FB2144E716E686C0FF81524EB7B	237.188000	266.51	119.90	5	266.51
FF5093021A395A2E96D1E4EF06B7C0A5	56.480000	182.00	40.79	9	40.79
FF95E22D7E5FE894F1E5BB89EAF50C46	40.790000	40.79	40.79	1	40.79
FFDBAEF9336F72DE5A7191A1BB9A93E8	23.798000	27.00	17.99	5	27.00

[1332 rows x 5 columns]

Visualizing Distribution of Product Prices: It plots histograms to visualize the distribution of product prices for specific audience IDs. Two histograms are plotted, one for each audience ID. The histograms show the frequency distribution of product prices for the selected audience IDs.//



Calculating Median Price Based on Audience ID: It calculates the median price for each audience ID using the grouped DataFrame and stores it in a variable called median_price_based_on_audience_id.

Filling Missing Values in 'product_price' Column Based on 'audience_id': It defines a function fill_price_na_based_on_audience_id(row) to fill missing values in the 'product_price' column based on the corresponding 'audience_id'. Inside the function, if the 'product_price' is missing (−1) and the
'audience_id' is valid (not '-1'), it fills the missing price with the median price of that audience ID. This function is then applied to each row of the DataFrame using the apply() function along the axis 1 (rows), effectively filling missing values in the 'product_price' column based on the median price of the respective audience ID.

3.3 Handling Time Information

Here we obtain the last recorded date from the 'click_timestamp' column of the DataFrame in the form of a datetime object and store it in the variable last_date.

```
# Extract last date from 'click_timestamp'
last_date = Dataframe['click_timestamp'].max()
last_date = datetime.fromtimestamp(last_date)

```

Extracting Last Date From 'click timestamp'

Extracting Last Date from 'click_timestamp': It retrieves the maximum value (latest timestamp) from the 'click_timestamp' column of the DataFrame using the max() function. The extracted timestamp.

Converting Timestamp to Datetime Object: It converts the extracted timestamp into a datetime object using the datetime.fromtimestamp() function. This conversion facilitates further manipulation and analysis of the date and time information.

Then we compute the start date of the week based on the last recorded date extracted earlier This calculation is useful for filtering and analyzing data within a specific time frame, such as the last week.:

```
# Calculate start of the week date
start_of_week_date = last_date - timedelta(days=7)
start_of_week_timestamp = int(start_of_week_date.timestamp())

```

Calculating Start Of The Week Date

Calculating Start of the Week Date: It subtracts 7 days from the last recorded date (last_date) using the timedelta object timedelta(days=7). This operation results in obtaining the date exactly one week before the last recorded date. The result is stored in the variable start_of_week_date.

Converting Start of the Week Date to Timestamp: It converts the start of the week date (start_of_week_date) into a Unix timestamp using the timestamp() method of the datetime object. The Unix timestamp represents the number of seconds that have elapsed since the Unix epoch (January 1, 1970, 00:00:00 UTC). The result is stored in the variable start_of_week_timestamp.

Then we filter the DataFrame to include only the data recorded within the last week, based on the start of the week timestamp calculated earlier. this operation

restricts the data to only include records that occurred within the time frame of the last week, enabling analysis or processing of recent data.

```
  # Filter data for last week
last_week_Dataframe = Dataframe[Dataframe['click_timestamp']
    > start_of_week_timestamp]

```

Filtering The Data For Last Week

It does so by creating a new DataFrame called last_week_Dataframe using boolean indexing. Specifically, it selects rows from the original DataFrame where the value in the 'click_timestamp' column is greater than the calculated start_of_week_timestamp.

Then we calculate the number of clicks for each unique product ID recorded within the last week. The resulting click_counts_last_week variable contains a Series where the index represents the unique product IDs and the corresponding values represent the number of clicks recorded for each product ID during the last week.

```
# Calculate click counts for last week
click_counts_last_week = last_week_Dataframe[
    last_week_Dataframe['product_id'] != '-1']['product_id'].
    value_counts()

```

Calculating Click Counts For Last Week

last_week_Dataframe[last_week_Dataframe['product_id'] != '-1']: This filters last_week_Dataframe to exclude rows where the 'product_id' is −1. This condition ensures that only valid product IDs are considered for click counting.

['product_id']: This selects only the 'product_id' column from the filtered DataFrame.

.value_counts(): This method counts the occurrences of each unique value in the 'product_id' column, effectively calculating the number of clicks for each product ID.

Then we define a function fill_na_nb_click_1week to fill missing values in the 'nb_clicks_1week' column based on the corresponding 'product_id'

value.

```
1   # Define a function to fill missing values in '
      nb_clicks_1week' column based on 'product_id'
2 def fill_na_nb_click_1week(row):
3     if row['nb_clicks_1week'] == -1:
4         if row['product_id'] != '-1':
5             try:
6                 return click_counts_last_week.loc[row['
      product_id']]
7             except:
8                 return 0
9         else:
10            return 0
11    else:
12        return row['nb_clicks_1week']
13
14 # Apply the function to fill missing values in '
      nb_clicks_1week' column
15 Dataframe['nb_clicks_1week'] = Dataframe.apply(
      fill_na_nb_click_1week, axis=1)
16
```

Filling Missing Values In 'nb clicks 1week' Column

The function takes a row of the DataFrame as input. If the value in the 'nb_clicks_1week' column is -1 (indicating a missing value):

It checks if the 'product_id' is not −1 (indicating a valid product ID). If a valid product ID is found, it tries to retrieve the corresponding click count from the click_counts_last_week Series using the 'product_id'. If the 'product_id' is not found in click_counts_last_week, it returns 0. If the 'product_id' is −1, it also returns 0. If the value in the 'nb_clicks_1week' column is not −1, it returns the original value.

Finally, the function is applied to the DataFrame using the .apply() method along the rows (axis=1), filling missing values in the 'nb_clicks_1week' column.

3.4 Cleaning The Dataset

Lastly, before we to the training and testing of our models, we perform data cleaning and preparation tasks on the DataFrame. This process ensures that the DataFrame is cleaned of missing or irrelevant values and is ready for further analysis or modeling.

```
  # Drop rows with missing or irrelevant values
Dataframe = Dataframe[Dataframe['product_price'] != -1]
Dataframe = Dataframe.drop(['SalesAmountInEuro','
    time_delay_for_conversion','product_title','user_id'],
    axis=1)
Dataframe = Dataframe[Dataframe['device_type'] != '-1']
Dataframe = Dataframe.drop(['audience_id'], axis=1)

# Save cleaned dataset to CSV
Dataframe.to_csv('clean-dataset.csv')

```

Dropping Rows With missing Or Irrelevant Values

Removing Rows with Missing Product Prices: Rows with missing values in the 'product_price' column (where 'product_price' equals −1) are dropped from the DataFrame.

Removing Irrelevant Columns: Columns such as 'SalesAmountInEuro', 'time_delay_for_conversion', 'product_title', and 'user_id' are dropped from the DataFrame using the .drop() method along the columns axis (axis=1).

Removing Rows with Missing or Invalid Device Types: Rows with missing or invalid values (where 'device_type' equals -1) in the 'device_type' column are dropped from the DataFrame.

Dropping the 'audience_id' Column: The 'audience_id' column is dropped from the DataFrame using the .drop() method along the columns axis (axis=1).

Saving the Cleaned Dataset: The cleaned DataFrame is saved to a CSV file named 'clean-dataset.csv' using the .to_csv() method.

Then we drop rows from the DataFrame where the 'product_id' is missing or equals −1, ensuring that only rows with valid product IDs are retained in the dataset. This step helps in maintaining data integrity and ensures that each record in the DataFrame corresponds to a valid product. This operation ensures that the dataset contains only relevant and complete information regarding products, which

is essential for accurate analysis and modeling.

```
    # Drop rows with missing 'product_id'
Dataframe = Dataframe[Dataframe['product_id'] != '-1']

```

Dropping rows With Missing 'product id'

It filters the DataFrame to include only rows where the 'product_id' is not equal to '-1' using boolean indexing. The filtered DataFrame is assigned back to the variable Dataframe, effectively removing rows with missing or invalid product IDs.

4 Training The Models

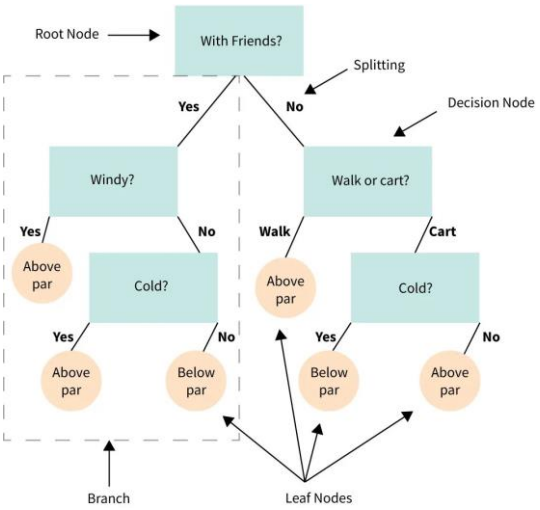
4.1 The Models We Used

4.1.1 Decision Tree

A decision tree is a type of supervised machine learning used to categorize or make predictions based on how a previous set of questions were answered. The model is a form of supervised learning, meaning that the model is trained and tested on a set of data that contains the desired categorization.

The decision tree may not always provide a clear-cut answer or decision. Instead, it may present options so the data scientist can make an informed decision on their own. Decision trees imitate human thinking, so it is generally easy for data scientists to understand and interpret the results.

A decision tree resembles, well, a tree. The base of the tree is the root node. From the root node flows a series of decision nodes that depict decisions to be made. From the decision nodes are leaf nodes that represent the consequences of those decisions. Each decision node represents a question or split point, and the leaf nodes that stem from a decision node represent the possible answers. Leaf nodes sprout from decision nodes similar to how a leaf sprouts on a tree branch. This is why we call each subsection of a decision tree a “branch”. Let’s take a look at an example for this. You are a golfer, and a consistent one at that. On any given day you want to predict where your score will be in two buckets: below par or over par.



4.1.2 Implementation

prepare the data, train a Decision Tree classifier, evaluate its performance, and visualize the results using various metrics, providing insights into the model's effectiveness in predicting the 'Sale' outcome.

```
  # Split dataset into features (X) and target (y)
y = Dataframe['Sale']
X = Dataframe[Dataframe.columns[1:]]

# Undersample the majority class to handle class imbalance
undersampler = RandomUnderSampler(random_state=42)
X, y = undersampler.fit_resample(X, y)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)
# %%
# Define preprocessing steps for numerical and categorical
    features
numerical_transformer = SimpleImputer(strategy='median')
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Preprocess features using ColumnTransformer
preprocessor = ColumnTransformer(
```

```
    transformers=[
        ('num', numerical_transformer, ['click_timestamp','nb_clicks_1week', 'product_price']),
        ('cat', categorical_transformer, ['device_type', 'product_country', 'product_id', 'partner_id'])
    ])

# Define a Decision Tree Classifier pipeline
model = Pipeline(steps=[('preprocessor', preprocessor),
                        ('classifier', DecisionTreeClassifier(random_state=42))])

# Train the model
model.fit(X_train, y_train)

# Make predictions on test data
y_pred = model.predict(X_test)

# Model Evaluation

# Calculate and print accuracy
print("Decision Tree Accuracy:", accuracy_score(y_test, y_pred))

# Print classification report
print("\nDecision Tree Classification Report:")
print(classification_report(y_test, y_pred))

# Print confusion matrix
print("\nDecision Tree  Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Calculate and plot ROC curve
y_probs = model.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_probs)
auc = roc_auc_score(y_test, y_probs)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC = %0.2f)' % auc)
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Decision Tree Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

64

Decision Tree

Splitting the dataset: It separates the target variable 'Sale' (y) from the feature variables (X). The feature variables (X) include all columns from the DataFrame except the first one, which typically contains row identifiers or indexes.

Handling class imbalance: It uses the RandomUnderSampler from the imblearn library to undersample the majority class (instances where 'Sale' equals 0) to handle class imbalance. This step ensures that both classes are represented equally in the dataset, which can improve the model's performance, especially in scenarios with imbalanced classes.

Splitting data into training and testing sets: It divides the feature variables (X) and target variable (y) into training and testing sets using the `train_test_split` function from `sklearn.model_selection`. The testing set comprises 20% of the data, while the training set comprises the remaining 80%.

Defining preprocessing steps: It defines preprocessing steps for numerical and categorical features separately using SimpleImputer for imputing missing values. For numerical features, missing values are imputed using the median, while for categorical features, the most frequent value is used. Categorical features are also one-hot encoded using OneHotEncoder to convert them into a numerical format suitable for modeling.

Defining a Decision Tree Classifier pipeline: It constructs a machine learning pipeline that consists of the defined preprocessing steps followed by a DecisionTreeClassifier. This pipeline encapsulates the entire data preprocessing and modeling process, making it easier to replicate and deploy.

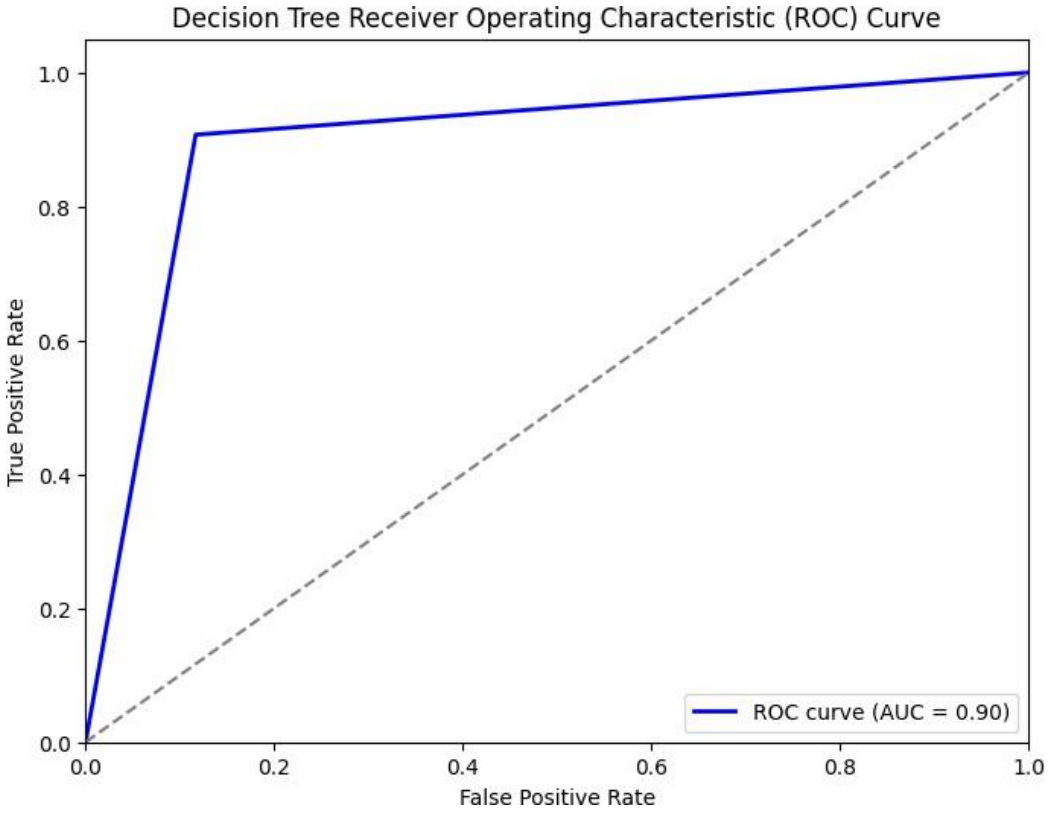
Training the model: It fits the pipeline model to the training data (`X_train` and `y_train`) using the fit method.

Making predictions: It predicts the target variable for the testing data (`X_test`) using the trained model and stores the predictions in `y_pred`.

Model evaluation: It evaluates the performance of the Decision Tree classifier by calculating and printing various metrics, including accuracy, classification report, confusion matrix, and the ROC curve with the area under the curve (AUC) score.

Visualization: It plots the Receiver Operating Characteristic (ROC) curve to visualize the performance of the classifier in distinguishing between the two classes (0 and 1). The AUC score quantifies the classifier's ability to discriminate between positive and negative classes.

```
Decision Tree Accuracy: 0.8951434878587197

Decision Tree Classification Report:
              precision    recall  f1-score   support

         0.0       0.91      0.88      0.89      1361
         1.0       0.89      0.91      0.90      1357

    accuracy                           0.90      2718
   macro avg       0.90      0.90      0.90      2718
weighted avg       0.90      0.90      0.90      2718


Decision Tree  Confusion Matrix:
[[1202  159]
 [ 126 1231]]
```

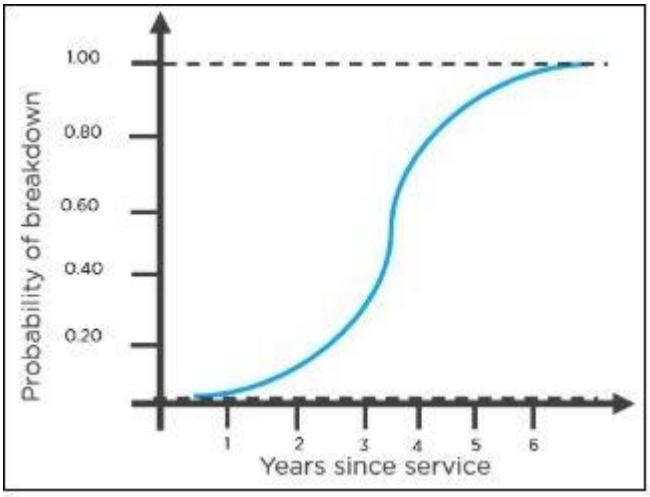

Decision Tree Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (AUC = 0.90)

4.1.3 Logistic Regression

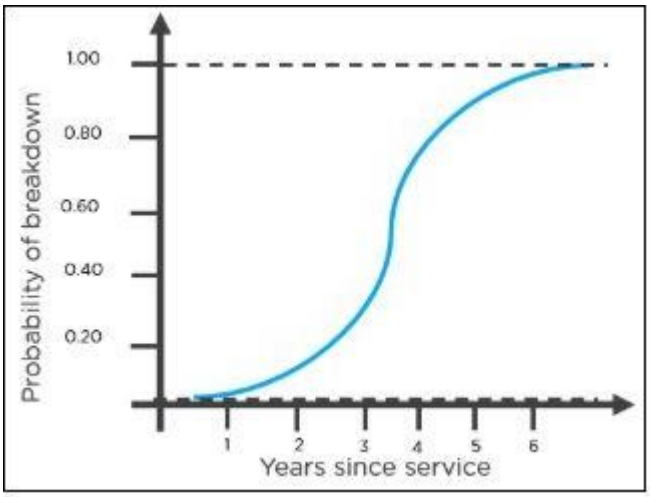
Logistic regression is a statistical method that is used for building machine learning models where the dependent variable is dichotomous: i.e. binary. Logistic regression is used to describe data and the relationship between one dependent variable and one or more independent variables. The independent variables can be nominal, ordinal, or of interval type.

The name "logistic regression" is derived from the concept of the logistic function that it uses. The logistic function is also known as the sigmoid function. The value of this logistic function lies between zero and one.

The following is an example of a logistic function we can use to find the probability of a vehicle breaking down, depending on how many years it has been since it was serviced last.



Here is how you can interpret the results from the graph to decide whether the vehicle will break down or not.



4.1.4 Implementation

Here we perform similar tasks as before, but now it's tailored for training and evaluating a Logistic Regression model. this section prepares the data, trains a logistic regression classifier, evaluates its performance, and visualizes the results using various metrics, providing insights into the model's effectiveness in predicting the 'Sale' outcome.

```
  # Define preprocessing steps for numerical and categorical features for Logistic Regression
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# Define a Logistic Regression pipeline
model = Pipeline(steps=[('preprocessor', preprocessor),
                        ('classifier', LogisticRegression(random_state=42))])

# Train the model
model.fit(X_train, y_train)

# Make predictions on test data
y_pred = model.predict(X_test)

# Model Evaluation for Logistic Regression

# Calculate and print accuracy
print("Logistic Regression Accuracy:", accuracy_score(y_test, y_pred))
```

```
21
22 # Print classification report
23 print("\nLogistic Regression Classification Report:")
24 print(classification_report(y_test, y_pred))
25
26 # Print confusion matrix
27 print("\nLogistic Regression Confusion Matrix:")
28 print(confusion_matrix(y_test, y_pred))
29
30 # Calculate AUC and plot ROC curve
31 y_probs = model.predict_proba(X_test)[:, 1]
32 auc = roc_auc_score(y_test, y_probs)
33
34 plt.figure(figsize=(8, 6))
35 plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC
      = %0.2f)' % auc)
36 plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
37 plt.xlim([0.0, 1.0])
38 plt. ylim ([0.0 ,     1.05])
39 plt.xlabel('False Positive Rate')
40 plt.ylabel('True Positive Rate')
41 plt.title('Logistic Regression Receiver Operating
      Characteristic (ROC) Curve')
42 plt.legend(loc='lower right')
43 plt.show()
44
```

Logistic Regression

Defining preprocessing steps for numerical features: It defines a pipeline for numerical features that first imputes missing values using the median strategy and then scales the features using StandardScaler. Scaling ensures that all numerical features have a similar scale, which can improve the performance of the logistic regression model.

Defining a Logistic Regression pipeline: It constructs a new pipeline for the logistic regression model. This pipeline includes the preprocessing steps defined earlier (preprocessor) and a LogisticRegression classifier.

Training the logistic regression model: It fits the logistic regression pipeline to the training data (X_train and y_train) using the fit method.

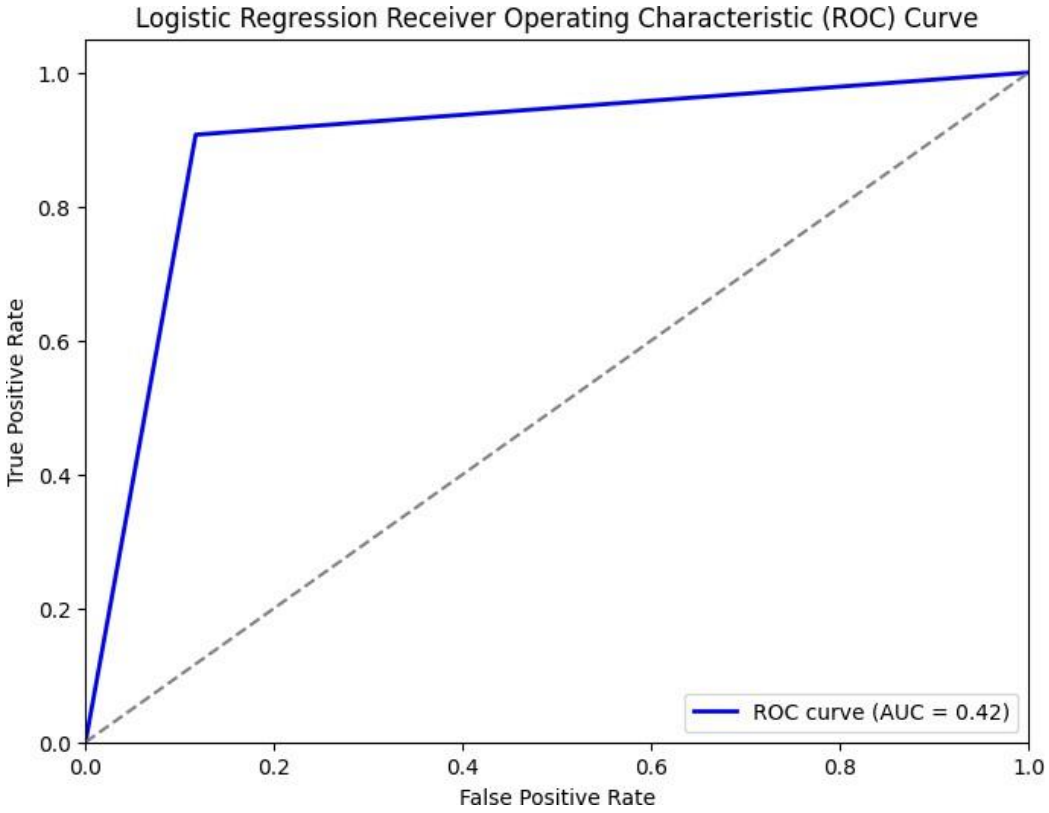
Making predictions: It predicts the target variable for the testing data (X_test) using the trained logistic regression model and stores the predictions in y_pred.

Model evaluation for logistic regression: It evaluates the performance of the

logistic regression model by calculating and printing various metrics, including accuracy, classification report, confusion matrix, and the ROC curve with the area under the curve (AUC) score.

Visualization: It plots the Receiver Operating Characteristic (ROC) curve to visualize the performance of the logistic regression classifier in distinguishing between the two classes (0 and 1). The AUC score quantifies the classifier's ability to discriminate between positive and negative classes.

```
Logistic Regression Accuracy: 0.42347314201618835

Logistic Regression Classification Report:
              precision    recall  f1-score   support

         0.0       0.25      0.07      0.12      1361
         1.0       0.45      0.77      0.57      1357

    accuracy                           0.42      2718
   macro avg       0.35      0.42      0.34      2718
weighted avg       0.35      0.42      0.34      2718


Logistic Regression Confusion Matrix:
[[ 102 1259]
 [ 308 1049]]
```


Logistic Regression Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (AUC = 0.42)

4.1.5 SVC (Support Vector Classifier)

SVC is a specific implementation of the Support Vector Machine algorithm that is designed specifically for classification tasks. In other words, SVC is an SVM used for classification. It seeks to find the hyperplane that best separates the data points into different classes. The terms "SVC" and "SVM" are sometimes used interchangeably, but when someone refers to an "SVC," they are usually referring to the classification variant of the algorithm.

The math behind Support Vector Classifier (SVC) is rooted in linear algebra and optimization. I'll provide a high-level overview of the mathematical concepts involved in SVC.

Hyperplane Equation: In a binary classification problem, the goal is to find a hyperplane that separates the data points of two classes. Mathematically, a hyperplane is defined by the equation: □ ∗ □ + □ = 0 Here, w is the weight vector perpendicular to the hyperplane, x represents a data point, and b is the bias term.

Margins: The margin is the distance between the hyperplane and the nearest data points from each class. The larger the margin, the more confident we are in the classification. The margin can be calculated as the distance between two parallel hyperplanes, one for each class. Mathematically, the margin is inversely proportional to the norm of the weight vector w.

Support Vectors: Support vectors are the data points that are closest to the hyperplane. These are the points that play a crucial role in determining the position of the hyperplane. The support vectors are the ones that contribute to the margin calculation.

Soft Margin: In real-world datasets, it's often not possible to have a perfect separation between classes. The concept of a soft margin SVM allows for some misclassification by allowing data points to be on the wrong side of the margin or even the wrong side of the hyperplane. This is controlled by introducing slack variables. Objective Function: The main objective in SVM is to maximize the margin while minimizing the classification error. This is achieved by solving an optimization problem.

Kernel Trick: For non-linearly separable data, SVM uses the kernel trick to map the data into a higher-dimensional space where a linear hyperplane can separate the data. Common kernel functions include polynomial kernels and radial basis function (RBF) kernels.

Solving the optimization problem yields the optimal values of w and b that

define the separating hyperplane. The optimization can be achieved using various optimization algorithms such as the Sequential Minimal Optimization (SMO) algorithm.

This is a simplified overview of the mathematics behind Support Vector Classifier. The actual implementations and optimizations might involve more complexities, especially when dealing with multi-class classification and non-linear problems using kernel functions.

4.1.6 Implementation

Here we define and evaluate a Support Vector Classifier (SVC) pipeline and provide a comprehensive assessment of the SVC classifier's performance, allowing for comparisons with other classifiers evaluated in previous sections.

Defining an SVC pipeline: It constructs a pipeline for the SVC classifier. The pipeline includes the preprocessing steps (preprocessor) defined earlier and an SVC classifier with a linear kernel, probability estimates enabled (probability=True), and a random state set to 42.

Training the SVC model: It fits the SVC pipeline to the training data (`X_train` and `y_train`) using the fit method.

Making predictions: It predicts the target variable for the testing data (`X_test`) using the trained SVC model and stores the predictions in `y_pred`.

Model evaluation for SVC: It evaluates the performance of the SVC classifier by calculating and printing various metrics, including accuracy, classification report, confusion matrix, and the ROC curve with the area under the curve (AUC) score.

Visualization: It plots the Receiver Operating Characteristic (ROC) curve to visualize the performance of the SVC classifier in distinguishing between the two classes (0 and 1). The AUC score quantifies the classifier's ability to discriminate between positive and negative classes.

```
# Define a Support Vector Classifier (SVC) pipeline
model = Pipeline(steps=[('preprocessor', preprocessor),
                        ('classifier', SVC(kernel='linear',
    probability=True, random_state=42))])

# Train the model
```

```
model.fit(X_train, y_train)

# Make predictions on test data
y_pred = model.predict(X_test)

# Model Evaluation for SVC

# Calculate and print accuracy
print("SVC Accuracy:", accuracy_score(y_test, y_pred))

# Print classification report
print("\nSVC Classification Report:")
print(classification_report(y_test, y_pred))

# Print confusion matrix
print("\nSVC Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Calculate AUC and plot ROC curve
y_probs = model.predict_proba(X_test)[:, 1]
auc = roc_auc_score(y_test, y_probs)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC
    = %0.2f)' % auc)
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('SVC Receiver Operating Characteristic (ROC) Curve'
    )
plt.legend(loc='lower right')
plt.show()

```

Support Vector Classifier (SVC)

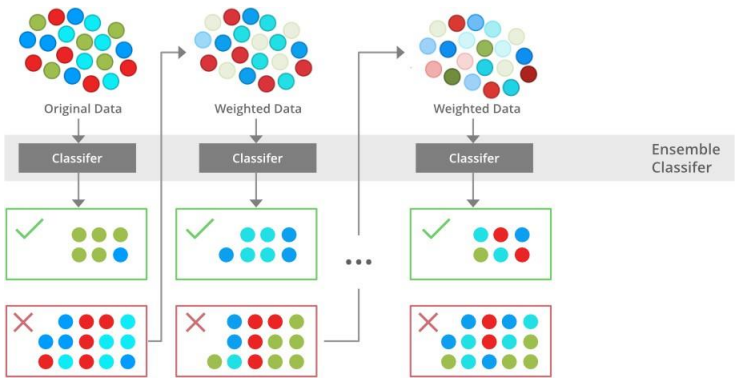
4.1.7 XGBoost

XGBoost is an optimized distributed gradient boosting library designed for efficient and scalable training of machine learning models. It is an ensemble learning method that combines the predictions of multiple weak models to produce a stronger prediction. XGBoost stands for “Extreme Gradient Boosting” and it has become one of the most popular and widely used machine learning algorithms due to its ability to handle large datasets and its ability to achieve state-of-the-art performance in many machine learning tasks such as classification and regression.

One of the key features of XGBoost is its efficient handling of missing values, which allows it to handle real-world data with missing values without requiring significant pre-processing. Additionally, XGBoost has built-in support for parallel processing, making it possible to train models on large datasets in a reasonable amount of time.

XGBoost can be used in a variety of applications, including Kaggle competitions, recommendation systems, and click-through rate prediction, among others. It is also highly customizable and allows for fine-tuning of various model parameters to optimize performance.

XgBoost stands for Extreme Gradient Boosting, which was proposed by the researchers at the University of Washington.



4.1.8 Implementation

In this section, we focuse on training and evaluating an XGBoost classifier. Preparing the data, training an XGBoost classifier, evaluating its performance, and visu-

alizing the results using various metrics, providing insights into the model's effectiveness in predicting the 'Sale' outcome.

```
  # Define preprocessing steps for numerical and categorical features for XGBoost
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# Define a XGBoost Classifier pipeline
model = Pipeline(steps=[('preprocessor', preprocessor),
                        ('classifier', xgb.XGBClassifier(random_state=42))])

# Train the model
model.fit(X_train, y_train)

# Make predictions on test data
y_pred = model.predict(X_test)

# Model Evaluation for XGBoost
# Calculate and print accuracy
print("XGBoost Accuracy:", accuracy_score(y_test, y_pred))

# Print classification report
print("\nXGBoost Classification Report:")
print(classification_report(y_test, y_pred))

# Print confusion matrix
print("\nXGBoost Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Calculate AUC and F1 Score, and plot ROC curve
y_probs = model.predict_proba(X_test)[:, 1]
auc = roc_auc_score(y_test, y_probs)
f1 = f1_score(y_test, y_pred)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC = %0.2f)' % auc)
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('XGBoost Receiver Operating Characteristic (ROC) Curve')
```

```
plt.legend(loc='lower right')
plt.show()

```

XGBoost

Defining preprocessing steps for numerical features: It defines a pipeline for numerical features that includes imputing missing values with the median and scaling the features using StandardScaler.

Defining an XGBoost Classifier pipeline: It constructs a pipeline for the XGBoost classifier. The pipeline comprises the preprocessing steps (preprocessor) defined earlier and an XGBClassifier.

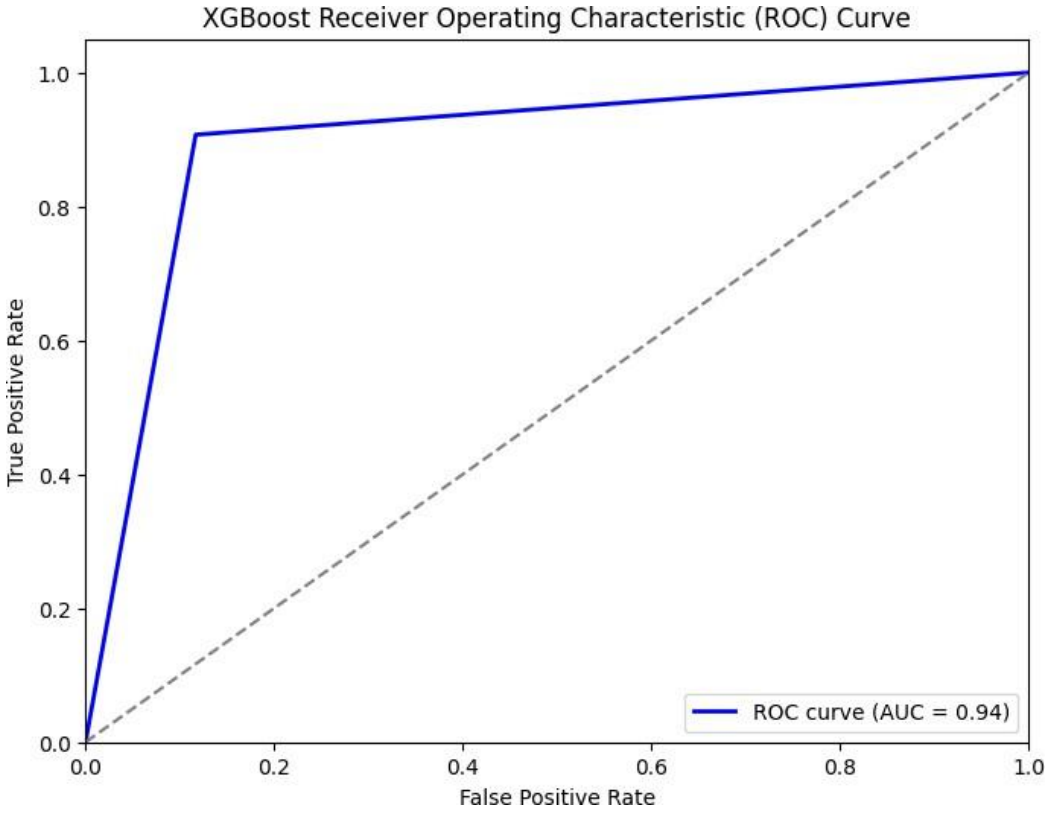
Training the XGBoost model: It fits the XGBoost pipeline to the training data (`X_train` and `y_train`) using the fit method.

Making predictions: It predicts the target variable for the testing data (`X_test`) using the trained XGBoost model and stores the predictions in `y_pred`.

Model evaluation for XGBoost: It evaluates the performance of the XGBoost classifier by calculating and printing various metrics, including accuracy, classification report, confusion matrix, and the ROC curve with the area under the curve (AUC) score and the F1 score.

Visualization: It plots the Receiver Operating Characteristic (ROC) curve to visualize the performance of the XGBoost classifier in distinguishing between the two classes (0 and 1). The AUC score quantifies the classifier's ability to discriminate between positive and negative classes.

```
XGBoost Accuracy: 0.9050772626931567

XGBoost Classification Report:
              precision    recall  f1-score   support

         0.0       0.96      0.84      0.90      1361
         1.0       0.86      0.97      0.91      1357

    accuracy                           0.91      2718
   macro avg       0.91      0.91      0.90      2718
weighted avg       0.91      0.91      0.90      2718


XGBoost Confusion Matrix:
[[1146  215]
 [  43 1314]]
```

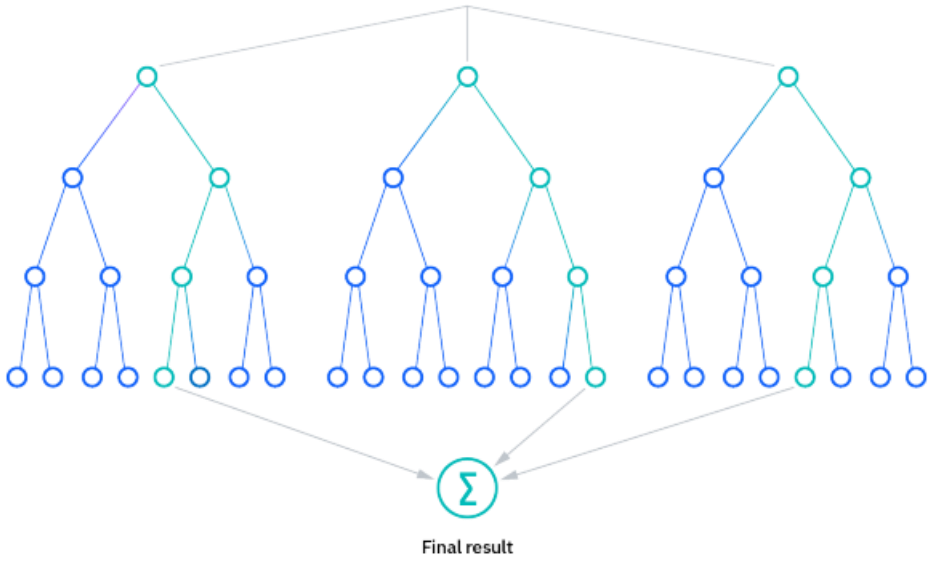


4.1.9 Random Forest

Random forest is a commonly-used machine learning algorithm, trademarked by Leo Breiman and Adele Cutler, that combines the output of multiple decision trees to reach a single result. Its ease of use and flexibility have fueled its adoption, as it handles both classification and regression problems.

Random forest algorithms have three main hyperparameters, which need to be set before training. These include node size, the number of trees, and the number of features sampled. From there, the random forest classifier can be used to solve for regression or classification problems.

The random forest algorithm is made up of a collection of decision trees, and each tree in the ensemble is comprised of a data sample drawn from a training set with replacement, called the bootstrap sample. Of that training sample, one-third of it is set aside as test data, known as the out-of-bag (oob) sample, which we'll come back to later. Another instance of randomness is then injected through feature bagging, adding more diversity to the dataset and reducing the correlation among decision trees. Depending on the type of problem, the determination of the prediction will vary. For a regression task, the individual decision trees will be averaged, and for a classification task, a majority vote—i.e. the most frequent categorical variable—will yield the predicted class. Finally, the oob sample is then used for cross-validation, finalizing that prediction.



4.1.10 Implementation

Here we define a pipeline for a Random Forest Classifier, train the model, evaluate its performance using various metrics, and plots the Receiver Operating Characteristic (ROC) curve.

```
  # Define a Random Forest Classifier pipeline
model = Pipeline(steps=[('preprocessor', preprocessor),
                        ('classifier', RandomForestClassifier
    (random_state=42))])

# Train the model
model.fit(X_train, y_train)

# Make predictions on test data
y_pred = model.predict(X_test)

# Model Evaluation

# Calculate and print accuracy
print("Random Forest Accuracy:", accuracy_score(y_test,
    y_pred))

# Print classification report
print("\nRandom Forest Classification Report:")
print(classification_report(y_test, y_pred))

# Print confusion matrix
print("\nRandom Forest Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Calculate and plot ROC curve
y_probs = model.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_probs)
auc = roc_auc_score(y_test, y_probs)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC
    = %0.2f)' % auc)
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Random Forest Receiver Operating Characteristic (
    ROC) Curve')
plt.legend(loc='lower right')
```

```
38
```

Random Forest Classifier

Pipeline: The 'preprocessor' step presumably handles any necessary preprocessing of the data, while the 'classifier' step utilizes a Random Forest Classifier with a specified random state for reproducibility.

Model Training: The pipeline is trained on the training data (X_train, y_train) using the fit() method.

Making Predictions: The trained model is then used to make predictions on the test data (X_test), using the predict() method.

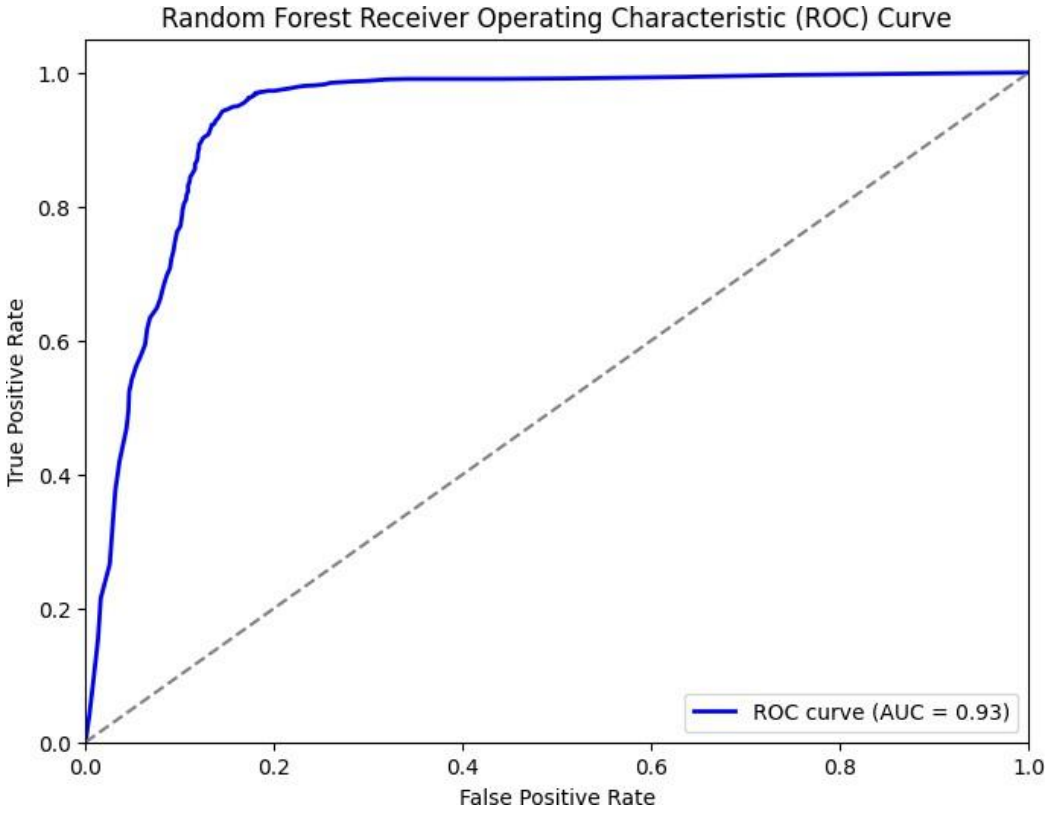
Model Evaluation: Accuracy: The accuracy of the model is calculated and printed using the accuracy_score() function.

Classification Report: The classification report, which includes precision, recall, F1-score, and support, is printed using the classification_report() function.

Confusion Matrix: The confusion matrix is printed using the confusion_matrix() function.

ROC Curve: The ROC curve is plotted using the true positive rate (TPR) against the false positive rate (FPR). The area under the ROC curve (AUC) is also calculated and displayed in the plot. This is done by first obtaining the predicted probabilities for the positive class using predict_proba(), then calculating the FPR, TPR, and thresholds using the roc_curve() function, and finally calculating the AUC score using the roc_auc_score() function.

```
Random Forest Accuracy: 0.8885209713024282

Random Forest Classification Report:
              precision    recall  f1-score   support

         0.0       0.90      0.87      0.89      1361
         1.0       0.87      0.91      0.89      1357

    accuracy                           0.89      2718
   macro avg       0.89      0.89      0.89      2718
weighted avg       0.89      0.89      0.89      2718


Random Forest Confusion Matrix:
[[1184  177]
 [ 126 1231]]
```



4.1.11 k-Nearest Neighbors

The k-nearest neighbors (KNN) algorithm is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point. It is one of the popular and simplest classification and regression classifiers used in machine learning today.

While the KNN algorithm can be used for either regression or classification problems, it is typically used as a classification algorithm, working off the assumption that similar points can be found near one another.

For classification problems, a class label is assigned on the basis of a majority vote—i.e. the label that is most frequently represented around a given data point is used. While this is technically considered "plurality voting", the term, "majority vote" is more commonly used in literature. The distinction between these terminologies is that "majority voting" technically requires a majority of greater than 50%, which primarily works when there are only two categories. When you have multiple classes—e.g. four categories, you don't necessarily need 50% of the vote to make a conclusion about a class; you could assign a class label with a vote of greater than 25%.

Regression problems use a similar concept as classification problem, but in this case, the average the k nearest neighbors is taken to make a prediction about a classification. The main distinction here is that classification is used for discrete

values, whereas regression is used with continuous ones. However, before a classification can be made, the distance must be defined. Euclidean distance is most commonly used, which we'll delve into more below. It's also worth noting that the KNN algorithm is also part of a family of "lazy learning" models, meaning that it only stores a training dataset versus undergoing a training stage. This also means that all the computation occurs when a classification or prediction is being made. Since it heavily relies on memory to store all its training data, it is also referred to as an instance-based or memory-based learning method.

Evelyn Fix and Joseph Hodges are credited with the initial ideas around the KNN model in this 1951 paper (link resides outside ibm.com) while Thomas Cover expands on their concept in his research (link resides outside ibm.com), "Nearest Neighbor Pattern Classification." While it's not as popular as it once was, it is still one of the first algorithms one learns in data science due to its simplicity and accuracy. However, as a dataset grows, KNN becomes increasingly inefficient, compromising overall model performance. It is commonly used for simple recommendation systems, pattern recognition, data mining, financial market predictions, intrusion detection, and more.

4.1.12 Implementation

And last but not least is the K-Nearest neighbors:

```
1
2 # Accuracy
3 print("K-Nearest Neighbors Accuracy:", accuracy_score(y_test,
      y_pred))
4
5 # Classification Report
6 print("\nK-Nearest Neighbors Classification Report:")
7 print(classification_report(y_test, y_pred))
8
9 # Confusion Matrix
10 print("\nK-Nearest Neighbors Confusion Matrix:")
11 print(confusion_matrix(y_test, y_pred))
12
13 # ROC Curve and AUC
14 y_probs = model.predict_proba(X_test)[:, 1]
15 fpr, tpr, thresholds = roc_curve(y_test, y_probs)
16 auc = roc_auc_score(y_test, y_probs)
17
18 #  Plot  ROC  curve
19 plt.figure(figsize=(8, 6))
```

```
20 plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (AUC
      = %0.2f)' % auc)
21 plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
22 plt.xlim([0.0, 1.0])
23 plt. ylim ([0.0 ,     1.05])
24 plt.xlabel('False Positive Rate')
25 plt.ylabel('True Positive Rate')
26 plt.title('K-Nearest Neighbors Receiver Operating
      Characteristic (ROC) Curve')
27 plt.legend(loc='lower right')
28 plt.show()
29
```

k-Nearest Neighbors

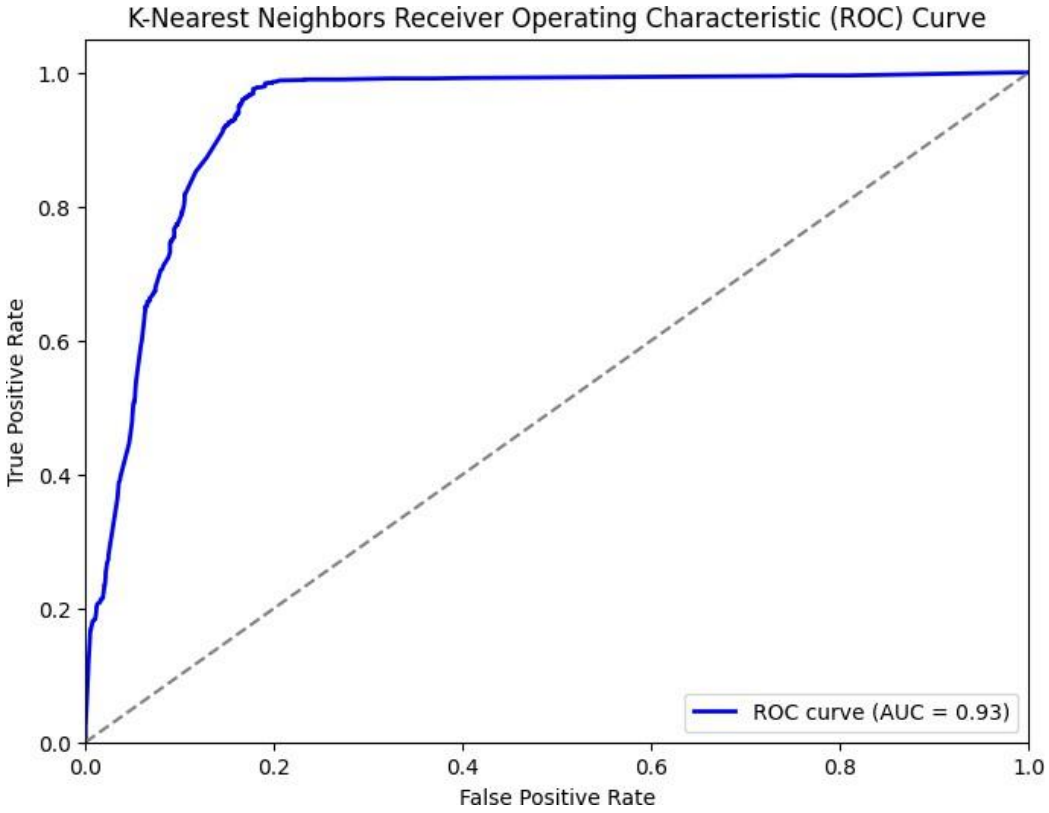
Accuracy: Calculates and prints the accuracy of the KNN model on the test data.

Classification Report: Prints the classification report for the KNN model, including precision, recall, F1-score, and support.

Confusion Matrix: Prints the confusion matrix for the KNN model.

ROC Curve and AUC: Computes the ROC curve and the Area Under the Curve (AUC) for the KNN model. It plots the ROC curve using matplotlib.

```
K-Nearest Neighbors Accuracy: 0.8958793230316409

K-Nearest Neighbors Classification Report:
              precision    recall  f1-score   support

         0.0       0.96      0.83      0.89      1361
         1.0       0.85      0.97      0.90      1357

    accuracy                           0.90      2718
   macro avg       0.90      0.90      0.90      2718
weighted avg       0.90      0.90      0.90      2718


K-Nearest Neighbors Confusion Matrix:
[[1124  237]
 [  46 1311]]
```


K-Nearest Neighbors Receiver Operating Characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (AUC = 0.93)

5 References

- 6 Step Field Guide for Building Machine Learning Projects
- EDA
- EDA Explanatory Data Analysis
- Exploratory Data Analysis with Pandas Python
- EDA
- logistic regression
- SVC
- XGBoost
- RFC
- KNN
- Decision Tree