Bank Marketing Campaign Analysis

Introduction

In the ever-evolving landscape of marketing, understanding the factors that drive customer behaviour is crucial for successful campaign strategies. This article delves into the pivotal role played by job type, education level, and marital status in shaping an individual's decision to subscribe to a term deposit. By analysing these key demographic variables, we can identify potential customer segments that show a higher propensity to say 'yes' to a term deposit. Furthermore, we explore the communication preferences of customers and consider whether alternative modes may prove more effective in improving conversion rates. Armed with these insights, marketers can tailor their efforts towards targeted segments and craft compelling campaigns that resonate with specific demographics, ultimately maximizing the potential for successful term deposit subscriptions.

Approach

In order to optimize marketing campaigns with the help of the dataset, we will have to take the following steps:

1. Import data from dataset and perform initial high-level analysis: look at the number of rows, look at the missing values, look at dataset columns and their values respective to the campaign outcome.

2. Cleaning the data

3. Model Selection and Optimization

4. Model Interpretability and Insights:

Importing the Libraries and the Data

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns



from sklearn.preprocessing import LabelEncoder

from sklearn.model_selection import train_test_split



from sklearn.tree import DecisionTreeClassifier

from sklearn.preprocessing import OneHotEncoder, StandardScaler

from sklearn.metrics import classification_report, accuracy_score

from sklearn.ensemble import RandomForestClassifier
```

```
from imblearn.over_sampling import SMOTE
```

```
import xgboost as xgb
```

```
from sklearn.utils.class_weight import compute_class_weight
```

```
from sklearn.model_selection import GridSearchCV
```

```
from sklearn.svm import SVC
```

```
all_datdata = pd.read_csv("bank-full.csv", delimiter=";")
```

Data Exploration

Let's take a look at the column types by using pandas shape and types:

```
print("The shape is:", data.shape)
```

```
print("The column types are:", data.dtypes)
```

```
The shape is: (45211, 17)
```

```
The column types are: age          int64
job          object
marital      object
```

education object

default object

balance int64

housing object

loan object

contact object

day int64

month object

duration int64

campaign int64

pdays int64

previous int64

poutcome object

y object

dtype: object

*Missing Attribute Values: None

We can use the columns to distinguish numerical and categorical columns:

```
cat = []

num=[]



for col in data.columns:

  if data[col].dtypes == "object":

     cat.append(col)

  else:

     num.append(col)



print("Categorical:",cat)

print("Numerical:", num)
```

Categorical: ['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'month', 'poutcome', 'y']

Numerical: ['age', 'balance', 'day', 'duration', 'campaign', 'pdays', 'previous']

Explore the Categorical Columns

In order to understand the distributions of the categorical columns in the dataset, we can iterate over each categorical feature. For every cat feature, a subplot as a countplot is added.

```
c = len(cat)

rows = (c // 3) + (c % 3 > 0)
```

```
print(rows)



plt.figure(figsize=(15,5*rows))



for i, cx in enumerate(cat, start=1):

  plt.subplot(rows, 3, i)

  sns.countplot(x=cx, data=data, palette="Set2")

  plt.title(f"Countplot of {cx}")

  plt.xticks(rotation=45)



plt.tight_layout()

plt.show()
```

Highlights from the plot:

Job: Most individuals belong to the ‘blue-collar’ and ‘management’ job categories, followed by ‘student’, ‘unemployed’, and ‘unknown’

Marital Status: A majority are married’, followed by ‘single’, and then a smaller portion of ‘divorced’.

Education: Those with ‘secondary’ education form the largest group, followed by ‘tertiary’. ‘

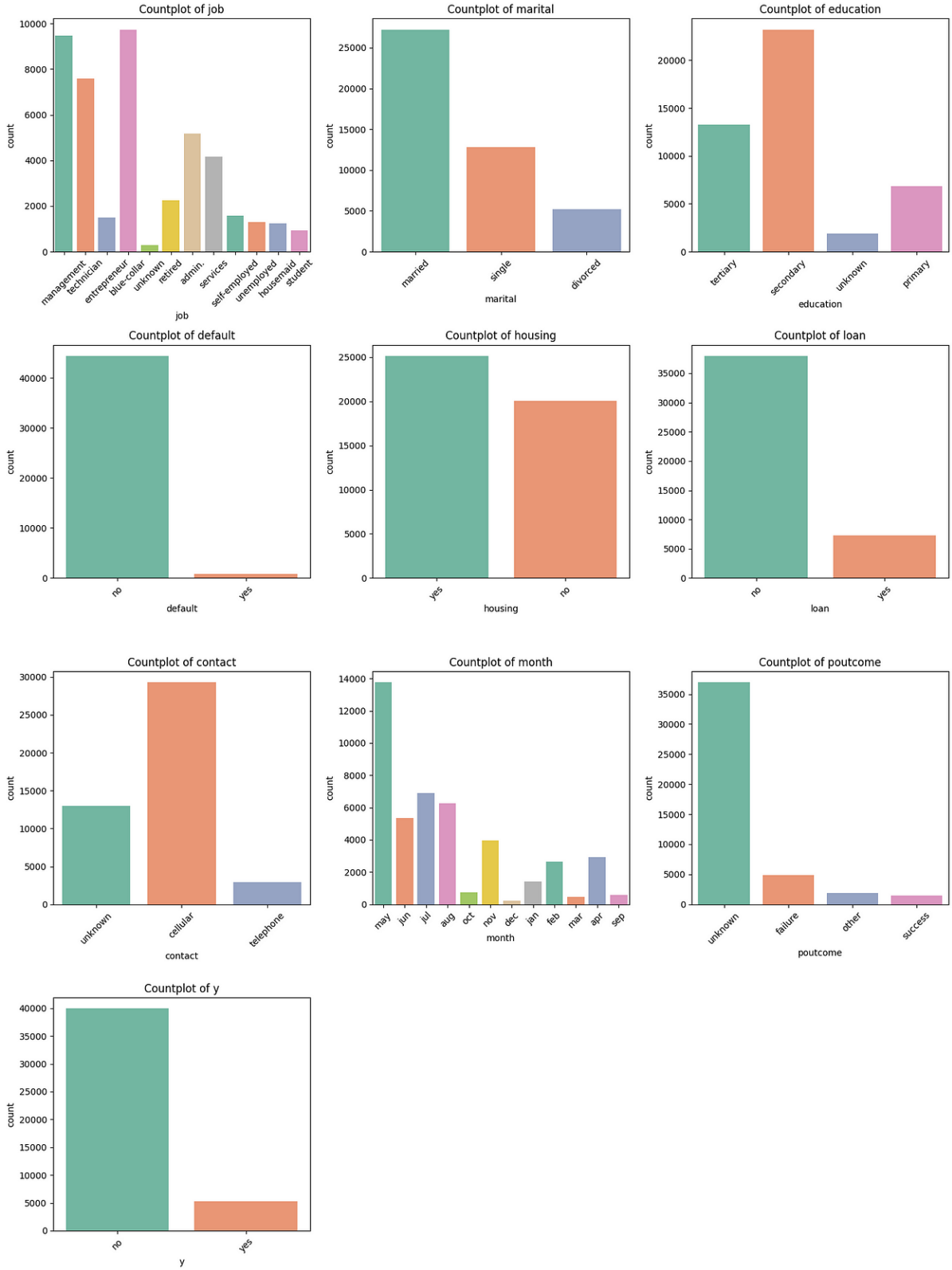
Default on Credit: A significant majority have no credit defaults, with a very small portion having defaulted.

Housing Loan: Most have a housing loan

Personal Loan: The majority do not have a personal loan

Previous Campaign Outcome: For most individuals, the outcome of the previous campaign remains ‘unknown’. Of those known, ‘failure’ is more common than ‘success’.

y: The client subscribed to a term deposit as the expected majority of the portion is No.



Explore the Numerical Columns

With the same logic/code let’s take a look at the numerical columns:

```
n = len(num)

nrows = (n // 3) + (n % 3 > 0)

print(rows)



plt.figure(figsize=(15,5*nrows))



for i,nx in enumerate(num, start=1):

  plt.subplot(nrows, 3, i)

  sns.histplot(x=nx, data=data, bins=15, color="green", kde=False)

  plt.title(f"Countplot of {nx}")

  plt.xticks(rotation=45)



plt.tight_layout()

plt.show()
```

Highlights from the plot:

Age: The majority of the customers fall within the age bracket of 20 to 60, with a pronounced peak around the late 20s to mid 30s.

Balance: Most customers have a balance in the lower range, with a few outliers possessing significantly higher balances.

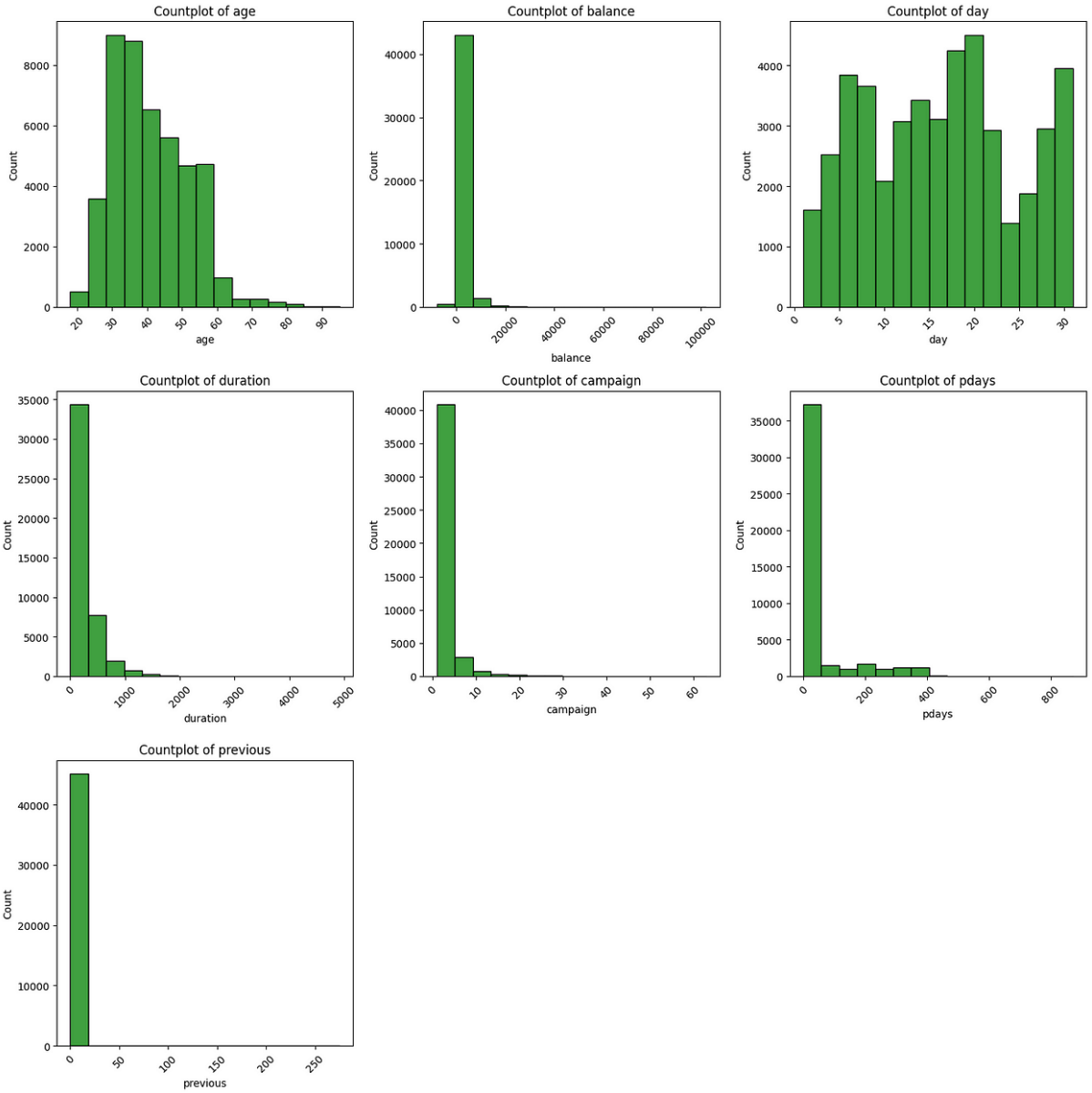
Day: The distribution shows a relatively even spread of days in the

month, with some noticeable peaks around specific days.

Duration: Most of the interactions lasted for a few minutes, with a small number lasting much longer.

previous: A significant portion of clients (at least 75%) were not contacted in previous campaigns.

pdays: This indicates the number of days since the customer was last contacted. The leftmost bar represents -1, which indicates that the customer was not contacted previously.



Final Observations upon examining the plots from the categorical and numerical distributions:

Customer Profiles:
There is a concentration of customers in the age group of late 20's and mid-30s mostly having a secondary education and being 'married'. These demographics seem to be the primary targets of marketing campaigns.

Economic Tokens:
A large proportion of customers do not have any default credit, have housing loans, and have low balances. This suggests that while many might have stable financial points, they also have financial commitments (housing loans).

Contact and Previous Campaigns:
The data shows a pattern of limited engagement in terms of the duration of contacts and the number of times a customer is contacted. That may indicate missed opportunities in engaging recurring customers.

Outliers and Data Cleaning

Taking a closer look at the data:

Last contact duration, attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call y is obviously

known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.

```
data = data.drop('duration', axis=1)
```

It is seen that even though I have raised the bins up to a decent number, some values are not represented in the histograms. So it is clear that we are observing outliers in the Balance, Duration, Campaign, Pdays, and Previous plots.

```
#Balance outliers holds a valueable info so I will not touch
```

```
data[["duration", "campaign", "pdays", "previous"]].describe()
```

```
       duration      campaign      pdays      previous
count 45211.000000  45211.000000 45211.000000 45211.000000
mean  258.163080    2.763841     40.197828    0.580323
std   257.527812    3.098021     100.128746   2.303441
min   0.000000      1.000000     -1.000000    0.000000
25%   103.000000    1.000000     -1.000000    0.000000
50%   180.000000    2.000000     -1.000000    0.000000
75%   319.000000    3.000000     -1.000000    0.000000
max   4918.000000   63.000000    871.000000   275.000000
```

To detect outliers I will use IQR:

```
Q1 = data['balance'].quantile(0.25)

Q3 = data['balance'].quantile(0.75)

IQR = Q3 - Q1



# Filtering the outliers

outliers = data[(data['balance'] < (Q1 - 1.5 * IQR)) | (data['balance'] > (Q3 + 1.5 * IQR))]

(len(outliers)/len(data))*100

%10.459843843312468 of the Balance column is outliers.
```

Instead of messing with the data, we will proceed with oversampling the minority groups.

Analysis of the response column

We can look at the y values among all the categorical columns. Unlike other subplots, we need to make sure the max values stays in the plot. For each plot, the y-axis is adjusted based on the maximum count of the current category.

```
ed_list = ["job", "education", "marital", "contact", "housing", "loan"]



n_rows = 4
```

```
n_cols = 3


plt.figure(figsize=(15, 6 * n_rows))


for index, column in enumerate(ed_list, start=1):
  plt.subplot(n_rows, n_cols, index)


  sns.countplot(data=data, x=column, hue='y', palette="Set2")


  # Get the maximum count for the current column
  max_count = data[column].value_counts().max()
  max_rounded = (max_count // 100) * 100


  plt.yticks(list(range(0, max_rounded + 1, max_rounded // 10)))


  plt.title(f'Distribution of y across "{column}"')
  plt.xticks(rotation=45)
```

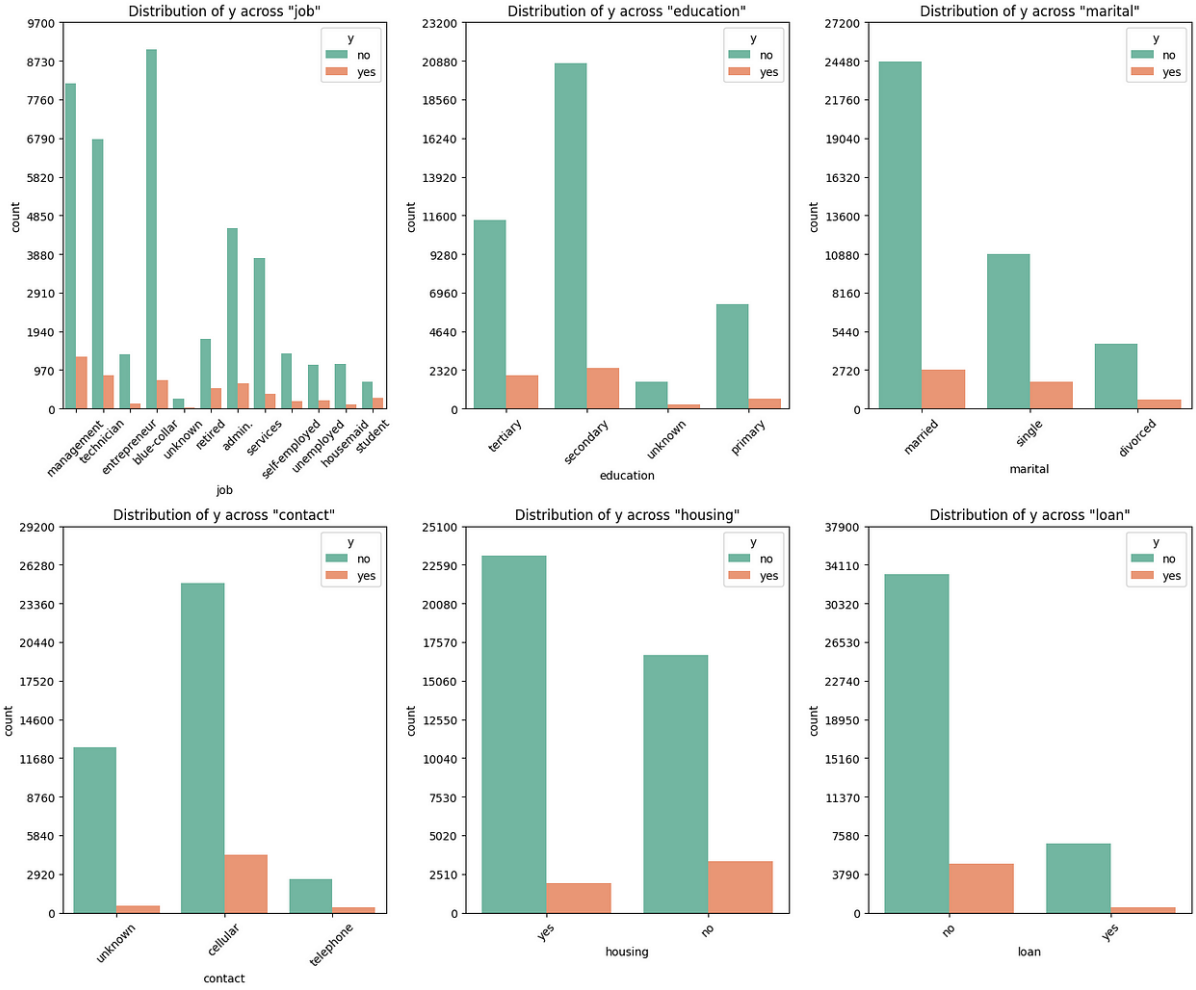
```
plt.tight_layout()

plt.show()
```

The job type plays a pivotal role in deciding whether an individual subscribes to a term deposit. While blue-collar, management, and technician categories house the most clients, the propensity to say ‘yes’ to a term deposit is noticeably higher among students and retired individuals. This suggests that marketing efforts might yield a higher conversion rate if tailored towards these groups.

The education level also correlates with decisions on term deposits. Those with tertiary education show a pronounced inclination towards subscribing, hinting at the potential value of crafting campaigns that resonate with educated demographics.

Marital status showcases that while a vast number of clients are married, it is the single clients who might have a relatively higher likelihood to subscribe to a term deposit.



Communication via cellular seems to be dominant, but it's important to consider if this mode is also the most effective in terms of conversion rates or if alternative modes like telephone yield better results, especially given the high 'no' rates seen in the cellular category.

Lastly, housing decisions also influence term deposit subscriptions. A significant chunk of clients with housing loans are less inclined to say 'yes', indicating that financial commitments like housing loans might deter clients from taking on additional financial products.

In sum, for better conversion rates, focusing on demographics like students, retired individuals, those with tertiary education, and possibly single clients could be fruitful. Moreover, evaluating the effectiveness of communication modes and understanding the financial commitments of clients can further refine the marketing approach.

Model Selection and Optimization

If you are feeling safe to make some predictions, follow along! It will be a long journey.

I have planned to implement:
1. Decision Trees and Random Forests
2. XGBoost for Classification
3. Support Vector Machines

1) Decision Trees and Random Forests

Decision Trees break down data by asking questions about specific features to make the data more organized.
Random Forest uses many of these trees and takes samples from the data to create them. The randomness comes from the method of picking different features unpredictably.

1.1. Pre-Processing:

Decision Trees & Random Forests can handle categorical variable but to be practical we will convert them with One-Hot Encoding.
We do not necessarily do feature scaling, at least for this part because trees are not sensivite.
We have a class imbalance and since we are dealing with the trees we are free to apply SMOTE.
We are already clean the missing data so we good to go!

1.1.1 Encoding:

I have used One-hot Encoding as a choice because in some of the categories we do not have an inherit order. But for the curious heads I have also applied Label Encoding which I will give the performance rates.
We will drop the first category in order to avoid duplication and transform the categorical column. Now they are represented as individual categories seen as below. Then remove the original categorical categories and replaced witht the one-hot encoded counterparts.

```
# 1. One-Hot-Encoding Categorical Variables


ohe = OneHotEncoder(drop='first', sparse=False)

encoded_features = ohe.fit_transform(data[cat])

encoded_df = pd.DataFrame(encoded_features, columns=ohe.get_feature_names_out(cat))
```

```
data = data.drop(columns=cat)

data = pd.concat([data, encoded_df], axis=1)
```

After One-hot Encoded, `data.columns` become as below. From now on our target variable is `y_yes`

```
Index(['age', 'balance', 'day', 'duration', 'campaign', 'pdays', 'previous',
       'job_blue-collar', 'job_entrepreneur', 'job_housemaid',
       'job_management', 'job_retired', 'job_self-employed', 'job_services',
       'job_student', 'job_technician', 'job_unemployed', 'job_unknown',
       'marital_married', 'marital_single', 'education_secondary',
       'education_tertiary', 'education_unknown', 'default_yes', 'housing_yes',
       'loan_yes', 'contact_telephone', 'contact_unknown', 'month_aug',
       'month_dec', 'month_feb', 'month_jan', 'month_jul', 'month_jun',
       'month_mar', 'month_may', 'month_nov', 'month_oct', 'month_sep',
       'poutcome_other', 'poutcome_success', 'poutcome_unknown', 'y_yes'],
      dtype='object')
```

1.1.2 Oversampling with SMOTE

As the target variable is imbalanced, we need to oversample the minority class with SMOTE.
Split the dataset: I prefer %80-%20

```
X1 = data.drop('y_yes', axis=1)

y1 = data['y_yes']



X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, y1, test_size=0.2, random_state=42)
```

And apply the SMOTE, to the training set:

```
smote = SMOTE(random_state=42)

X_resampled1, y_resampled1 = smote.fit_resample(X_train1, y_train1)
```

1.2. Training

We have clean the missing values, convert the categorical valuesfor better computation handled with the class imbalance for a better score, we can proceed with the training:

```
# Training Decision Tree

dt_model = DecisionTreeClassifier(random_state=42)

dt_model.fit(X_resampled1, y_resampled1)



dt_predictions1 = dt_model.predict(X_test1)

# Training Random Forest

rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
```

```
rf_model.fit(X_resampled1, y_resampled1)
```

```
rf_predictions = rf_model.predict(X_test1)
```

1.3. Scores

```
# Evaluation

print("Decision Tree Classifier:")

print("Accuracy:", accuracy_score(y_test1, dt_predictions1))

print(classification_report(y_test1, dt_predictions1))
```

1.3.1 The results for the One-Hot Encoded, Reduced Class Imbalance with SMOTE, Decision Tree Scores:

```
Decision Tree Classifier:

Accuracy: 0.8281543735486011

              precision    recall  f1-score   support

         0.0       0.91      0.90      0.90      7952

         1.0       0.31      0.33      0.32      1091
```

```
  accuracy                           0.83      9043
 macro avg       0.61      0.61      0.61      9043
weighted avg       0.83      0.83      0.83      9043
```

Not bad but we can raise the accuracy score with Random Forests. So training again:

```
# Evaluation
print("\nRandom Forest Classifier:")
print("Accuracy:", accuracy_score(y_test1, rf_predictions))
print(classification_report(y_test1, rf_predictions))
```

1.3.2 The results for the One-Hot Encoded, Reduced Class Imbalance with SMOTE, Random Forests Scores:

```
Random Forest Classifier:
Accuracy: 0.8920712153046555
              precision    recall  f1-score   support

         0.0       0.91      0.98      0.94      7952
```

1.0	0.63	0.25	0.36	1091
accuracy			0.89	9043
macro avg	0.77	0.62	0.65	9043
weighted avg	0.87	0.89	0.87	9043

For the curious talking heads, to get these results I have also applied feature selection and Label Encoding (instead of One-Hot), as they poorly scored compared to I haven't bother myself adding the code.

More interpretability: SHAP

SUMMARY

The Random Forest outperforms the Decision Tree in every metric for this dataset.

Both models have a challenge with Class 1, which is often the case when one class is underrepresented in the data (as is typical in binary classification problems with a class imbalance). However, the Random Forest does a better job than the Decision Tree.

It's worth noting that even though the accuracy of both models is quite high, it's important to also consider other metrics like precision, recall, and F1-score, especially when dealing with imbalanced datasets.

2. Extreme Gradient Boosting-XGBoost

2.1. Preprocessing

I have ensured that there are no missing data in the dataset and have already applied one-hot encoding. Moving forward, will focus on:
— Data Scaling
— Determining Class Weights
— Selecting Relevant Features

2.1.1. Scaling

We haven’t performed Feature Scaling due to unncecesesaties of the Decision Trees split on feature values using conditions and are not sensitive to the scale. However, as we further use on SVM as well, I have preferred to perform Scaling here (it is not a must for XGBoost but nice to have)

I will use Standart Scaling due to the performance metrics of the SVM.

```
# 3. Scaling

scaler = StandardScaler()

X_train3 = scaler.fit_transform(X_train3)

X_test3 = scaler.transform(X_test3)
```

```
X_train3 = pd.DataFrame(X_train3, columns=data.drop('y_1', axis=1).columns)

X_test3 = pd.DataFrame(X_test3, columns=data.drop('y_1', axis=1).columns)
```

2.1.2. Class Weights

Since XGBoost itself does not automatically compute class weights, it would be a We'll start with computing the class weight due:

```
class_weights = compute_class_weight('balanced', classes=[0,1], y=y_train3)

weights = {0: class_weights[0], 1: class_weights[1]}
```

The weights can be interpreted as follows:

1. The clients subscribed to a term deposit, has a weight of approximately 0.566
2. The clients did not subscribe a term deposit, has a weight of approximately 4.27

```
{0: 0.5662397845758838, 1: 4.27416686362562}
```

Class 0 way underrepresented so we will use the weights to balance them.

2.2. Traning w/ Weights

Let's use the weights in the model training.

```
# 5. Training with Class Weights
```

```
model = xgb.XGBClassifier()

model.fit(X_train3, y_train3, sample_weight=y_train3.map(weights))
```

Feature Selection

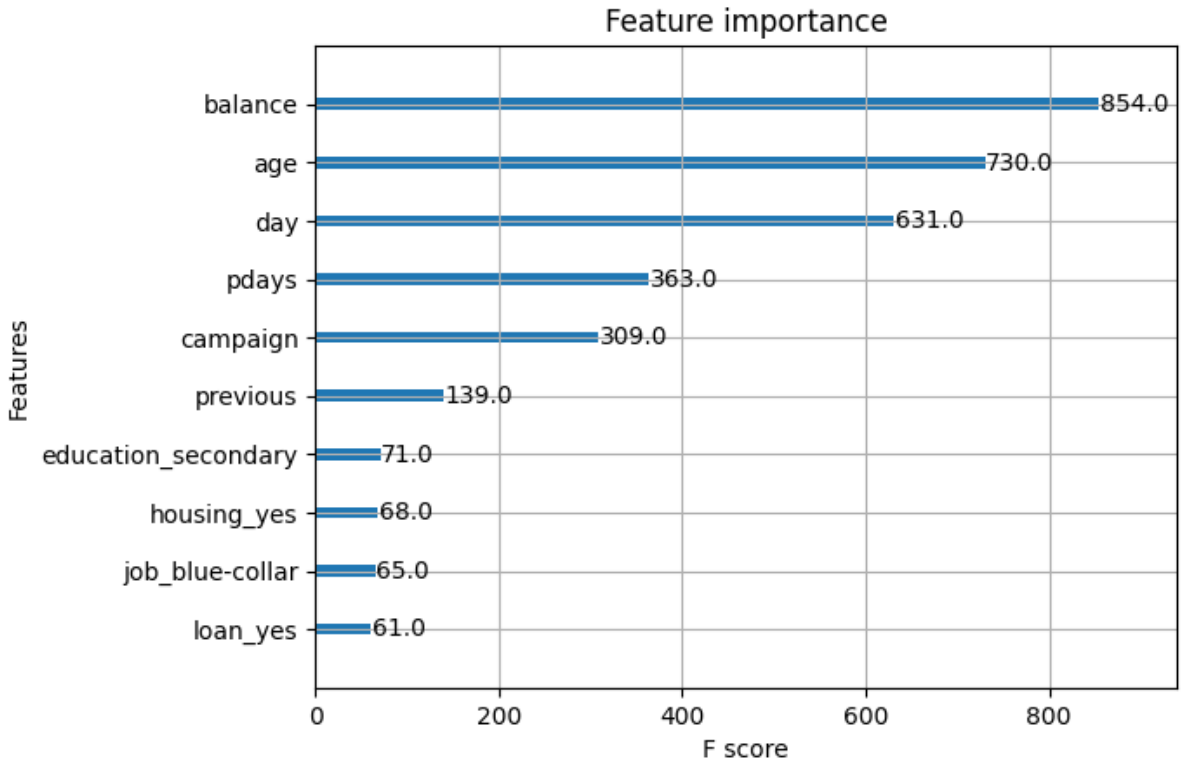
After training the model let’s check the feature importance based on the F Score. The importance is measured using the F score, which, in this context, indicates how often a feature is used to split the data.

— Balance:

With the highest F score of 854, the account balance of individuals is the most significant predictor in the model.

— Age:

The age of the individual is the second most influential feature with an F score of 730. This could imply that certain age groups might be more likely to respond in a particular way compared to others.



Features like education_secondary, housing_yes, job_blue-collar, and loan_yes have relatively lower importance, suggesting they play a less pivotal role in influencing the outcome. However, their presence still brings some value to the model so we will not drop any of the features.

2.3. Score

With the same code we will check the score

```
y_pred = model.predict(X_test3)

print(accuracy_score(y_test3, y_pred))

print(classification_report(y_test3, y_pred))
```

Score for the One-Hot Encoded, Scaled, Weight Trained XGBoost:

0.8247263076412695

	precision	recall	f1-score	support
0.0	0.94	0.86	0.90	7985
1.0	0.35	0.58	0.44	1058
accuracy			0.82	9043
macro avg	0.65	0.72	0.67	9043
weighted avg	0.87	0.82	0.84	9043

It performed poorly with the inital XGBoost. We can try with hyperparameter tunig:

```
# Define the hyperparameters and their possible values
param_grid = {
  'learning_rate': [0.01, 0.1, 0.5, 1],
  'max_depth': [3, 5, 7, 10],
  'min_child_weight': [1, 3, 5],
```

```
  'subsample': [0.5, 0.7, 1.0],
  'colsample_bytree': [0.5, 0.7, 1.0],
  'n_estimators': [100, 200, 500],
  'objective': ['binary:logistic']
}

# Instantiate the grid search model
grid_search = GridSearchCV(estimator=xgb.XGBClassifier(),
                           param_grid=param_grid,
                           scoring='accuracy',
                           cv=3,
                           verbose=1,
                           n_jobs=-1)

grid_search.fit(X_train3, y_train3)

print("Best hyperparameters:", grid_search.best_params_)
```

```
best_xgb_model = grid_search.best_estimator_
```

After hyperparameter tuning, the model has high precision and recall for the 0.0 class, which is expected since it's the majority class.

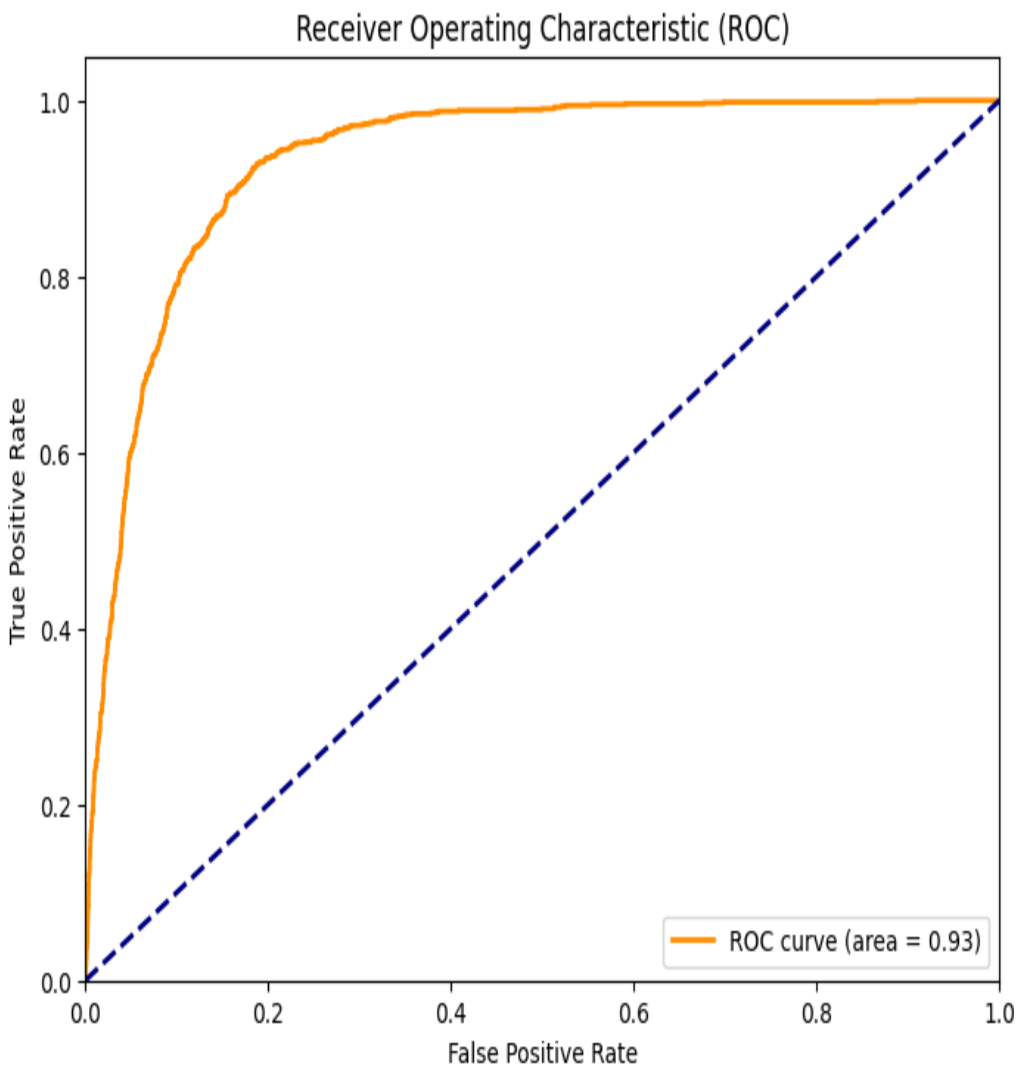
Score for the One-Hot Encoded, Scaled, Weight Trained XGBoost and Tuned:

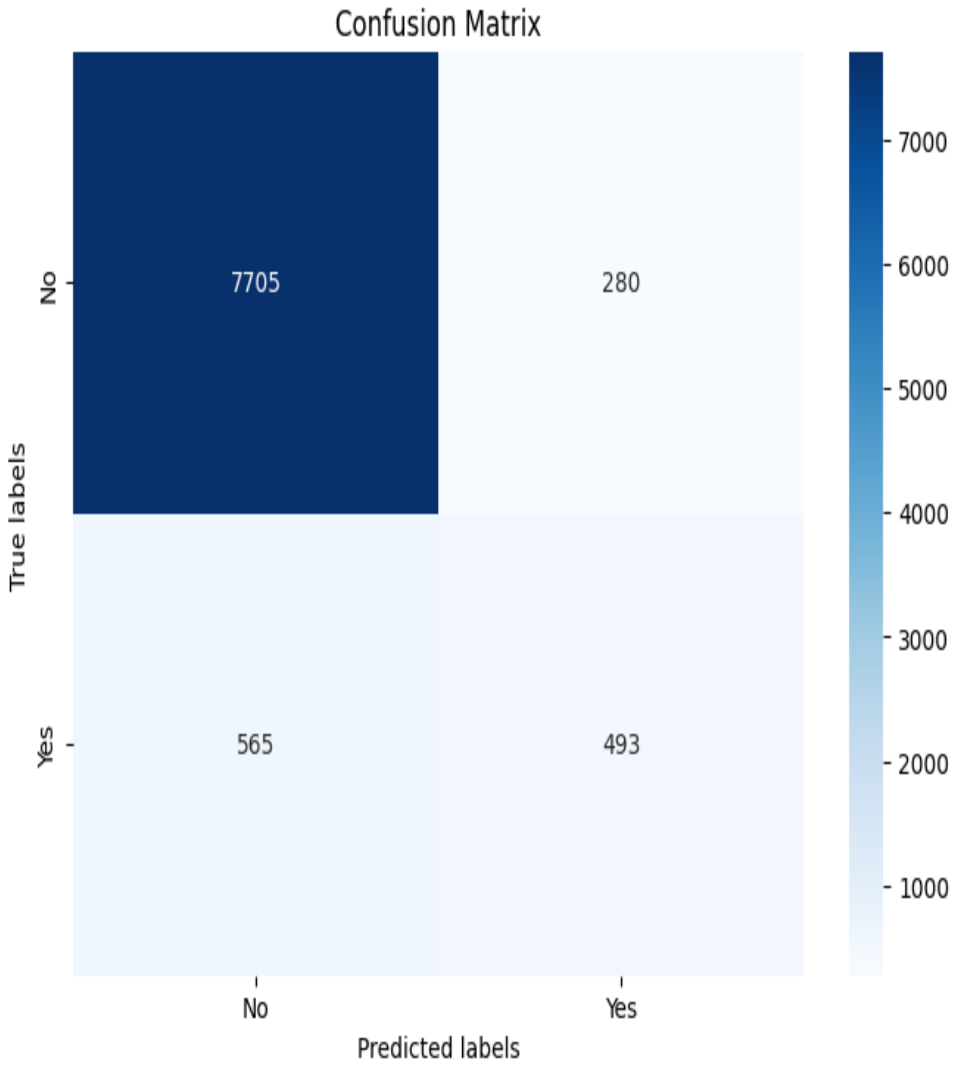
0.9065575583324118

	precision	recall	f1-score	support
0.0	0.93	0.96	0.95	7985
1.0	0.64	0.60	0.54	1058
accuracy			0.91	9043
macro avg	0.78	0.72	0.74	9043
weighted avg	0.90	0.91	0.90	9043

Next steps could involve evaluating the tuned model on the test set, checking confusion matrices, and ROC curves.

The AUC value is 0.93, which is close to 1. This is an excellent result.


Receiver Operating Characteristic (ROC)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 0.93)



3. Support Vector Machine

3.1. Pre-processing

In the earlier processes, we have already:

— Handled missing values

— Scaled the features

— Encoded Categorical Variables

— Checked the Feature Importance

— Handled Class Imbalance with SMOTE and Class Weights
— Checked the Outliers with IQR

There is only one more step for SVM, that is Kernel Choice. As we are dealing with non-linear data I will choose, Radial Basis Function (RBF) Kernel.

3.2 Training

When working with Support Vector Machines (SVM) on datasets, especially imbalanced ones, it's crucial to select the right hyperparameters for optimal performance

First, we define the range of values for each hyperparameter that we want to search. In this example, we'll search over:

— Regularization parameter
— Kernel type
— Gamma coefficient for the RBF kernel

We'll use GridSearchCV to automate the process of searching over the hyperparameters grid. It performs cross-validation for each combination of hyperparameters and selects the best one based on the cross-validation results.

```
# Define hyperparameters grid to search

param_grid = {
```

```
  'C': [0.01, 0.1, 1, 10],
  'kernel': ['linear', 'rbf'],
  'gamma': [0.01, 0.1, 1, 10]
}

# hyperparameters grid, cross-validation folds
grid_search = GridSearchCV(SVC(class_weight='balanced'), param_grid, cv=5, verbose=1, n_jobs=-1)

# Train GridSearchCV
grid_search.fit(X_train4, y_train4)

# Get best model
best_svm = grid_search.best_estimator_

# Predict using best model
y_pred_best_svm = best_svm.predict(X_test4)
```