BANK MARKETING

1. Importing Libraries

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import zscore
from sklearn.preprocessing import LabelEncoder, RobustScaler
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier,ExtraTreesClassifier
from sklearn.model_selection import train_test_split, RandomizedSearchCV,cross_
from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix, accuracy_s
from sklearn.metrics import roc_auc_score,roc_curve
import pickle
import warnings
warnings.filterwarnings('ignore')
```

2. Importing Data & Analyse

In [ ]:
```
train = pd.read_csv("C:\\Users\\Rima Das\\OneDrive\\Desktop\\Third-Phase Practi
train
```

In [ ]:
```
test= pd.read_csv("C:\\Users\\Rima Das\\OneDrive\\Desktop\\Third-Phase Practice
test
```

In [ ]:
```
#checking Dimension of Train & Test data

print("In Train dataset we have {} rows & {} columns".format(*train.shape))
print("In Test dataset we have {} rows & {} columns".format(*test.shape))
```

In [ ]:
```
# Column Names in Train & Test Dataset

print("These are the columns present in TRAIN dataset: \n",train.columns)
print("\nThese are the columns present in TEST dataset: \n",test.columns)
```

Observations:

- **Train DataFrame:**

- In Train dataset we can see that we have 31647 rows & 18 columns.

- **Test DataFrame:**
  - In test Dataset we can see that we have 13564 rows & 17 columns.
  - Here 1 column is missing because thats our target/label column

In [ ]:
```
# Checking data information
train.info()
```

In [ ]:
```
test.info()
```

Observations

- **Train Data**
  - We can see there are 18 columns including out Target column
  - Among 18 column, we have 8 columns contains integar values & 10 columns has object values
  - Our target column has object values
  - Memory Usage 4.3+ MB

- **Test Data**
  - There are 17 columns no target column is there
  - out of 17 columns, 8 columns has integar values and 9 columns has object values.
  - Memory Usage: 1.8+ MB

3. Missing Values

In [ ]:
```
train.isnull().sum()
```

In [ ]:
```
test.isnull().sum()
```

In [ ]:
```
#Visualizing it
sns.heatmap(train.isnull())
plt.show()
```

Observations:

- There are no null values in our datasets(train and test)

4. Duplicate Values:

In [ ]:
```
print('We have {} duplicated values in our TRAIN datasets'.format(train.duplica
print('We have {} duplicated values in our TEST datasets'.format(test.duplicate
```

Observations:

- We have no duplicated values in both datasets

5. Separating Categorical and numerical columns

In [ ]:
```
#Seaparating Categrocial columns in train Dataframe
cat_col = train.select_dtypes(include='object').columns
print("Categorial columns in our train dataframe:\n",cat_col)

#Seaparating Numerical columns in train Dataframe
num_col = train.select_dtypes(include='int64').columns
print("\nNumerical columns in our train dataframe:\n",num_col)
```

6. Statistical Summary

In [ ]:
```
#Statistical summary of Numerical colums
train.describe()
```

Observations:

1. **ID**:
    - The 'ID' column appears to represent unique identifier values.
    - The range of 'ID' values starts from 2 and goes up to 45,211, suggesting that there are a total of 31,647 unique records in the dataset.
2. **age**:
    - The 'age' column represents the age of individuals.
    - The mean age in the dataset is approximately 40.96 years, with a minimum age of 18 years and a maximum age of 95 years.
    - The majority of individuals (at least 25%) fall in the age range of 33 to 48 years.
3. **balance**:
    - The 'balance' column likely represents the financial balance of individuals.
    - The mean balance is approximately 1363.89 units, with a wide range of values.
    - The distribution of 'balance' appears to be positively skewed, as the maximum value is significantly higher than the mean, indicating the presence of outliers.
4. **day**:

- The 'day' column might represent the day of the month when the data was recorded.
- The values in this column range from 1 to 31, suggesting that it could be associated with calendar days.

5. **duration**:

   - The 'duration' column may represent the duration of a contact or interaction.
   - The mean duration is approximately 258.11 seconds (or about 4.3 minutes).
   - The maximum duration is considerably higher at 4918 seconds, indicating the presence of long-duration interactions.

6. **campaign**:

   - The 'campaign' column likely represents the number of contacts made during a marketing campaign.
   - On average, individuals were contacted approximately 2.77 times during a campaign.
   - The maximum number of campaign contacts for an individual is 63, suggesting some extreme values.

7. **pdays**:

   - The 'pdays' column may indicate the number of days since the last contact.
   - The majority of values in this column appear to be -1, which might represent that the client was not previously contacted.
   - The maximum value is 871, suggesting a significant time gap for some clients between contacts.

8. **previous**:

   - The 'previous' column may represent the number of previous marketing contacts.
   - On average, individuals have had approximately 0.57 previous contacts.
   - The maximum number of previous contacts is 275, indicating that some individuals have been contacted multiple times in previous campaigns.

These observations provide an initial understanding of the TRAIN dataset's numerical features, including central tendencies, spreads, and potential outliers. Further data analysis and preprocessing may be required to gain deeper insights and prepare the data for modeling or analysis.

7. Data Visualization:

Exploring Target/ Label

In [ ]:
```
# Calculate value counts for 'subscribed' column
value_counts = train['subscribed'].value_counts()

# Calculate percentages
percentages = (value_counts / len(train)) * 100

# Combine value counts and percentages into a DataFrame
result_df = pd.DataFrame({'Count': value_counts, 'Percentage': percentages})

# Print the result DataFrame
print(result_df)
```

In [ ]:
```
# Visualizing it
fig, axes = plt.subplots(1, 1, figsize=(8, 6))  # Use plt.subplots to create a

# Count plot: Value Count of Subscribed column
sns.countplot(x='subscribed', data=train, ax=axes)
axes.set_title('Value Count of Subscribed column')

# Show the plot
plt.tight_layout()
plt.show()
```

Observation:

- The 'subscribed' column has two unique values: 'no' and 'yes'.
- The total count of 'no' in the column is 27,932.
- The total count of 'yes' in the column is 3,715.
- The percentage of 'no' in the column is approximately 88.26%.
- The percentage of 'yes' in the column is approximately 11.74%.

This indicates that the majority of the data falls under the 'no' category in the 'subscribed' column, with 'yes' accounting for a smaller portion of the data. Thats means our data is imbalanced.

a. Univariate Analysis

i. Categorical columns of train dataframe

In [ ]:
```
# Checking Value Counts of Categorical Column(Train Data)
cat_col = cat_col[:-1]

for col in cat_col:
    print('*'*10,col,'*'*10)
    print(train[col].value_counts())
    print("\n")
```

In [ ]:
```
#Visualizing it

# Create subplots
fig, axes = plt.subplots(len(cat_col), 1, figsize=(6, 4*len(cat_col)))

# Iterate through categorical columns
for i, column in enumerate(cat_col):
    # Count plot
    sns.countplot(x=column, data=train, ax=axes[i])
    axes[i].set_title(f"Value count of {column}", fontsize=16)

plt.tight_layout()
plt.show()
```

Observations:

job:

- The 'job' column represents the occupation or job type of individuals.
- The most common job categories are 'blue-collar,' 'management,' 'technician,' and 'admin.'
- 'Blue-collar' is the most frequent job category, with 6,842 individuals having this occupation.

marital:

- The 'marital' column represents the marital status of individuals.
- The most common marital status categories are 'married' and 'single.'
- 'Married' is the most frequent marital status, with 19,095 individuals.

education:

- The 'education' column represents the education level of individuals.
- The majority of individuals have 'secondary' education, followed by 'tertiary' and 'primary' education.
- 'Secondary' education is the most common, with 16,224 individuals having this education level.

default:

- The 'default' column likely indicates whether individuals have credit in default.
- The majority of individuals do not have credit in default ('no'), while a smaller proportion have credit in default ('yes').

housing:

- The 'housing' column may indicate whether individuals have housing loans.
- More individuals have housing loans ('yes') compared to those without ('no').

loan:

- The 'loan' column represents whether individuals have personal loans.
- Most individuals do not have personal loans ('no').

contact:

- The 'contact' column may indicate the method of communication.
- 'Cellular' is the most common contact method, followed by 'unknown' and 'telephone.'

month:

- The 'month' column represents the month in which contacts were made.
- The most frequently used contact month is 'May,' followed by 'July' and 'August.'

poutcome:

- The 'poutcome' column might represent the outcome of a previous marketing campaign.
- The most common outcome is 'unknown,' followed by 'failure,' 'other,' and 'success.'

These observations provide insights into the distribution and frequencies of categories within each categorical variable, which is valuable for understanding the characteristics of the dataset.

ii. Numerical Column of Train Data

In [ ]:
```
#Creating for loop to iterate over Numerical columns

for col in num_col:
    print('*'*10,col,'*'*10)
    print(train[col].value_counts())
    print("\n")
```

In [ ]:
```
# Visualizing numerical columns
plt.figure(figsize=(20, 15), facecolor='white')
plotnumber = 1

for column in num_col:
    if plotnumber <= 8:
        ax = plt.subplot(4,2, plotnumber)
        sns.histplot(train[column], color='g')
        plt.xlabel(column, fontsize=16)
    plotnumber += 1

plt.tight_layout()
plt.show()
```

Observations:

ID:

- The ID column appears to be unique, with each value occurring only once. This suggests that it might be an identifier for each row.

Age:

- The age column shows a distribution of ages, with the most common ages being around 32, 31, and 33.
- Age varies from 18 to 95.

Balance:

- The balance column has a wide range of values, including both positive and negative balances.
- The most common balance value is 0, followed by 1 and 2.

Day:

- The day column represents the day of the month. It shows a distribution of days, with some days being more common than others.
- The most common days are around the 20th, 18th, and 21st of the month.

Duration:

- The duration column represents the duration of the contact in seconds. It varies widely, with many different values.
- There are multiple occurrences of similar durations, indicating that certain call durations are more common.

Campaign:

- The campaign column represents the number of contacts performed during the campaign. It shows a distribution with most values between 1 and 10 contacts.
- There are a few outliers with higher numbers of contacts.

Pdays:

- The pdays column represents the number of days since the client was last contacted. The majority of values are -1, indicating that most clients were not previously contacted.
- There are a few positive values indicating the number of days since the last contact for some clients.

Previous:

- The previous column represents the number of contacts performed before this campaign. It shows a distribution with most values being 0.
- There are some non-zero values, indicating that some clients were contacted in previous campaigns.

These observations provide insights into the distribution and characteristics of the numerical columns in the dataset.

b. Bivariate Analysis

i. Categorical Columns Vs Target

In [ ]:
```
# Create a contingency table for 'job' and 'subscribed'
job_crosstab = pd.crosstab(train['job'], train['subscribed'])

# Calculate the percentage
job_crosstab_norm = job_crosstab.div(job_crosstab.sum(1).astype(float), axis=0)

# Concatenate the contingency table and normalized contingency table side by si
concatenated_tables = pd.concat([job_crosstab, job_crosstab_norm], axis=1, keys
concatenated_tables
```

In [ ]:
```
# Visualizing it
# Visualizing it
fig, axes = plt.subplots(1, 1, figsize=(10, 4))  # Use plt.subplots to create a

# Count plot: Value count of Subscribed with Job Role
sns.countplot(x='job', hue='subscribed', data=train, palette={"yes": "g", "no":
axes.set_title('Value count of Subscribed with Job Role', fontsize=12)

# Show the plot
plt.tight_layout()
plt.show()
```

Observations:

1. **Job vs. Subscription Count:**

    - Among the different job categories, "retired" and "student" have the highest counts of "yes" subscriptions, with 362 and 182 respectively.

- "Entrepreneur" and "unknown" have the lowest counts of "yes" subscriptions, with 85 and 26 respectively.
- "Admin" and "blue-collar" jobs have relatively high counts of both "yes" and "no" subscriptions, with 452 and 489 "yes" subscriptions, and 3179 and 6353 "no" subscriptions respectively.

2. **Job vs. Subscription Percentage:**

   - When looking at the percentage of "yes" subscriptions within each job category, "student" stands out with the highest percentage at approximately 28.66%.
   - "Retired" individuals also have a significant percentage of "yes" subscriptions at around 23.00%.
   - "Blue-collar" workers have the lowest percentage of "yes" subscriptions, accounting for only about 7.15% of the total in their category.
   - "Management" professionals have a relatively high percentage of "yes" subscriptions, with approximately 13.90%.

3. **Overall:**

   - The percentage of "yes" subscriptions varies significantly across different job categories, indicating that job type may be a significant factor in predicting whether an individual subscribes to the service or not.
   - "Student" and "retired" individuals are more likely to subscribe, while "blue-collar" workers are less likely to subscribe.

These observations provide insights into how the "job" variable relates to the subscription outcome, which can be valuable for targeted marketing strategies or further analysis in the

In [ ]:
```
#Marital status vs subscribed
marital_crosstab= pd.crosstab(train['marital'], train['subscribed'])

#Calculating the percentage
marital_crosstab_norm = marital_crosstab.div(marital_crosstab.sum(1).astype(flo

#concatenate marital crosstab and marital crosstab norm side by side
marital_concat=pd.concat([marital_crosstab, marital_crosstab_norm], axis=1, key
marital_concat
```

In [ ]:
```
# Visualizing it
fig, axes = plt.subplots(1, 1, figsize=(6, 4))  # Use plt.subplots to create a

# Count plot: Value count of Subscribed with Marital Status
sns.countplot(x='marital', hue='subscribed', data=train, palette={"yes": "g", "
axes.set_title('Value count of Subscribed with Marital Status', fontsize=16)

# Show the plot
plt.tight_layout()
plt.show()
```

Observations:

1. **Marital Status vs. Subscription Count:**

- Among the different marital statuses, individuals who are "married" have the highest count of "no" subscriptions, with 17,176 individuals.
- On the other hand, individuals who are "single" have the highest count of "yes" subscriptions, with 1,351 individuals.
- "Divorced" individuals fall in between, with 3,185 "no" subscriptions and 445 "yes" subscriptions.

2. **Marital Status vs. Subscription Percentage:**

- When looking at the percentage of "yes" subscriptions within each marital status category:
  - "Single" individuals have the highest percentage of "yes" subscriptions at approximately 15.14%.
  - "Divorced" individuals have the lowest percentage of "yes" subscriptions, accounting for about 12.26% of their category.
  - "Married" individuals have a moderate percentage of "yes" subscriptions, approximately 10.05%.

3. **Overall:**

- The marital status appears to have an influence on the subscription outcome.
- "Single" individuals have a notably higher likelihood of subscribing compared to "married" and "divorced" individuals.
- These observations suggest that marital status may be a relevant factor in predicting whether an individual subscribes to the service or not.

These insights into the relationship between marital status and subscription status can be

In [ ]:
```
# Education Vs subscribed

edu_crosstab =pd.crosstab(train['education'],train['subscribed'])

# Calculate the percentage
edu_crosstab_norm = edu_crosstab.div(edu_crosstab.sum(1).astype(float),axis=0)

# Concatenated the contingency table and normalize contingency table side by si
edu_concat = pd.concat([edu_crosstab, edu_crosstab_norm], axis=1, keys=['Count'
edu_concat
```

In [ ]:
```
# Visualizing it
plt.figure(figsize=(8, 6))  # Set the figure size

# Count plot: Value count of Subscribed with Marital Status
sns.countplot(x='marital', hue='subscribed', data=train, palette={"yes": "g", '
plt.title('Value count of Subscribed with Marital Status', fontsize=16)

# Show the plot
plt.tight_layout()
plt.show()
```

Observations:

- **Primary Education:**

- Count of "No" Subscriptions: 4,381
- Count of "Yes" Subscriptions: 427
- Percentage of "No" Subscriptions: 91.12%
- Percentage of "Yes" Subscriptions: 8.88%

- **Secondary Education:**

  - Count of "No" Subscriptions: 14,527
  - Count of "Yes" Subscriptions: 1,697
  - Percentage of "No" Subscriptions: 89.54%
  - Percentage of "Yes" Subscriptions: 10.46%

- **Tertiary Education:**

  - Count of "No" Subscriptions: 7,886
  - Count of "Yes" Subscriptions: 1,415
  - Percentage of "No" Subscriptions: 84.79%
  - Percentage of "Yes" Subscriptions: 15.21%

- **Unknown Education:**

  - Count of "No" Subscriptions: 1,138
  - Count of "Yes" Subscriptions: 176
  - Percentage of "No" Subscriptions: 86.61%
  - Percentage of "Yes" Subscriptions: 13.39%

These observations provide insights into the distribution of subscriptions based on education levels. It appears that the percentage of "Yes" subscriptions is higher among individuals with tertiary education compared to other education categories, while the primary education category has the highest percentage of "No" subscriptions.

In [ ]:
```
# Default vs Subscribed

default_crosstab=pd.crosstab(train['default'],train['subscribed'])

#Percentage Calculate
default_crosstab_norm = default_crosstab.div(default_crosstab.sum(1).astype(flo

#concate both tables
default_concat=pd.concat([default_crosstab, default_crosstab_norm], axis=1, key
default_concat
```

In [ ]:
```
# Visualizing it: Value count of Default vs Subscribed
sns.countplot(x='default', hue='subscribed', data=train)
plt.title('Value count of Subscribed with Default', fontsize=16)
plt.show()
```

Observation :

- For customers with no previous default (default=no):

    - Count: 27,388 customers did not subscribe ('no'), and 3,674 customers subscribed ('yes').
    - Percentage: Approximately 88.17% of customers with no previous default did not subscribe, while around 11.83% of them subscribed.
- For customers with a previous default (default=yes):
    - Count: 544 customers did not subscribe ('no'), and 41 customers subscribed ('yes').
    - Percentage: Approximately 92.99% of customers with a previous default did not subscribe, while around 7.01% of them subscribed.

This suggests that customers with no previous default are more likely to subscribe compared to those with a previous default, as indicated by the higher subscription rate (11.83% vs. 7.01%)

In [ ]:
```
# Housing vs Subscribed

Housing = pd.crosstab(train['housing'], train['subscribed']) #Value Count
Percentage = Housing.div(Housing.sum(1).astype(float), axis= 0) # Percentage

housing_concat=pd.concat([Housing,Percentage], axis=1, keys=['Count','Percentag
housing_concat
```

In [ ]:
```
#visualizing it

# First plot: Value count of housing vs Subscribed
sns.countplot(x='housing', hue='subscribed', data=train)
plt.title('Value count of Subscribed with Housing', fontsize=16)
plt.show()
```

Observation:

- **Count of Subscribed Customers:**
    - For customers with "housing" status as "no," there are 11,698 subscribers who have not subscribed (no), and 2,365 subscribers who have subscribed (yes).
    - For customers with "housing" status as "yes," there are 16,234 subscribers who have not subscribed (no), and 1,350 subscribers who have subscribed (yes).
- **Percentage of Subscribed Customers:**
    - Among customers with "housing" status as "no," approximately 83.18% have not subscribed (no), while about 16.82% have subscribed (yes).
    - Among customers with "housing" status as "yes," approximately 92.32% have not subscribed (no), while about 7.68% have subscribed (yes).

These observations provide insights into the relationship between housing status and subscription to a term deposit. It appears that customers with no housing tend to have a higher subscription rate compared to those with housing.

In [ ]:
```
# loan vs subscribed

loan_crosstab = pd.crosstab(train['loan'], train['subscribed']) #Value Count
Percentage = loan_crosstab.div(loan_crosstab.sum(1).astype(float), axis= 0) # P

loan_concat=pd.concat([loan_crosstab,Percentage], axis=1, keys=['Count','Percer
loan_concat
```

In [ ]:
```
#visualizing it

# First plot: Value count of Loan vs Subscribed
sns.countplot(x='loan', hue='subscribed', data=train)
plt.title('Value count of Subscribed with Loan', fontsize=12)
plt.show()
```

Observation:

It is evident that the count and percentage of subscribers and non-subscribers categorized by whether or not they have a loan. The "no" category represents clients without a loan, while the "yes" category represents clients with a loan. It is evident that a higher percentage of clients without a loan subscribed to the term deposit compared to those with a loan.

In [ ]:
```
#contact vs subscribed
contact_crosstab = pd.crosstab(train['contact'], train['subscribed']) #Value Co
Percentage = contact_crosstab.div(contact_crosstab.sum(1).astype(float), axis=

contact_concat=pd.concat([contact_crosstab,Percentage], axis=1, keys=['Count','
contact_concat
```

In [ ]:
```
# Visualizing it
fig, axes = plt.subplots(1, 1, figsize=(6, 4))  # Use plt.subplots to create a

# Count plot: Value count of Subscribed with Contact
sns.countplot(x='contact', hue='subscribed', data=train, palette={"yes": "g", "
axes.set_title('Value count of Subscribed with Contact', fontsize=16)

# Show the plot
plt.tight_layout()
plt.show()
```

Observations:

The count and percentage of subscribers and non-subscribers based on the contact method used. The "cellular" category represents clients contacted via cellular phones, "telephone" represents clients contacted via landline telephones, and "unknown" represents clients with unknown contact methods. It is evident that a higher percentage of subscribers were reached through cellular phones compared to other contact methods.

In [ ]:
```
#month vs subscribed

month_crosstab = pd.crosstab(train['month'], train['subscribed']) #Value Count
Percentage = month_crosstab.div(month_crosstab.sum(1).astype(float), axis= 0) #

month_concat=pd.concat([month_crosstab,Percentage], axis=1, keys=['Count','Perc
month_concat
```

In [ ]:
```
# Visualizing it
fig, axes = plt.subplots(1, 1, figsize=(6, 4))  # Use plt.subplots to create a

# Count plot: Value count of Subscribed with month
sns.countplot(x='month', hue='subscribed', data=train, palette={"yes": "g", "no
axes.set_title('Value count of Subscribed with month', fontsize=12)

# Show the plot
plt.tight_layout()
plt.show()
```

Observations:

- In the month of April (apr), there were 1,671 subscriptions, with 384 (18.7%) being "yes."
- August (aug) had 3,813 subscriptions, with 520 (12%) of them being "yes."
- December (dec) had the smallest number of subscriptions, with 85 in total. However, a high percentage of them, 72 (45.9%), were "yes."
- February (feb) had 1,522 subscriptions, and 305 (16.7%) of them were "yes."
- January (jan) had 880 subscriptions, with 97 (9.9%) being "yes."
- July (jul) had 4,403 subscriptions, and 441 (9.1%) were "yes."
- June (jun) had 3,355 subscriptions, and 383 (10.2%) were "yes."
- March (mar) had 168 subscriptions, with 174 (50.9%) of them being "yes."
- May (may) had the highest number of subscriptions, with 9,020 in total. However, only 649 (6.7%) of them were "yes."
- November (nov) had 2,508 subscriptions, with 275 (9.9%) being "yes."
- October (oct) had 288 subscriptions, with 224 (43.8%) being "yes."
- September (sep) had 219 subscriptions, with 191 (46.6%) being "yes."

These observations provide insights into the distribution of subscriptions across different months, highlighting variations in both count and percentage of "yes" subscriptions.

In [ ]:
```
#poutcome vs subscribed

poutcome_crosstab = pd.crosstab(train['poutcome'], train['subscribed']) #Value
Percentage = poutcome_crosstab.div(poutcome_crosstab.sum(1).astype(float), axis

poutcome_concat=pd.concat([poutcome_crosstab,Percentage], axis=1, keys=['Count'
poutcome_concat
```

In [ ]:
```
# Visualizing it
fig, axes = plt.subplots(1, 1, figsize=(6, 4))  # Use plt.subplots to create a

# Count plot: Value count of Subscribed with poutcome
sns.countplot(x='poutcome', hue='subscribed', data=train, palette={"yes": "g",
axes.set_title('Value count of Subscribed with poutcome', fontsize=12)

# Show the plot
plt.tight_layout()
plt.show()
```

ii. Bivariate Analysis for Numeric columns with Target

In [ ]:
```
#campaign vs Subscribed
train.groupby(['campaign'])['subscribed'].count().reset_index().sort_values(by=
```

Observation:

- The majority of clients were contacted fewer times during the campaign, with the most common counts being 1, 2, and 3.
- As the number of campaign contacts increased beyond 3, the counts of clients contacted decreased significantly. This suggests that a large portion of clients were contacted relatively few times.
- The subscription rate tends to decrease as the number of campaign contacts increases. Clients who were contacted only once or twice had a higher subscription rate compared to those contacted more frequently.
- There is a noticeable drop in the subscription rate after the 6th campaign contact, with a steep decline in subscriptions as the number of contacts continues to increase.
- It's interesting to note that there are instances where clients were contacted up to 63 times during the campaign, but the subscription rate remains low for these cases.

In [ ]:
```
# Age vs Subscribed
pd.set_option('display.max_rows', None)
train.groupby(['age'])['subscribed'].count().reset_index().sort_values(by='subs
```

Observations:

- The age of 32 has the highest number of subscribers, with 1457 individuals.
- Ages 31, 33, 34, and 35 also have a substantial number of subscribers, ranging from 1314 to 1417.
- The number of subscribers generally decreases as age increases, with fewer subscribers in the older age groups.
- There are very few subscribers in the age groups beyond 60, with ages 61 and above having the lowest numbers of subscribers.
- Ages between 18 and 28 have a moderate number of subscribers.

Overall, the data suggests that the majority of subscribers are in their early thirties, and the number of subscribers tends to decrease as age increases beyond that point.

In [ ]:
```
#Balance vs Subscribed
train.groupby(['balance'])['subscribed'].count().reset_index().sort_values(by='
```

Observations:

It's evident that the balance variable doesn't have a significant impact on detecting whether someone has subscribed to the terms or not.

In [ ]:
```
# Day vs Subscribed
train.groupby(['day'])['subscribed'].count().reset_index().sort_values(by='subs
```

Observations:

1. The dataset contains two columns: "day" and "subscribed."
2. The "day" values range from 0 to 30, indicating a time period of one month.
3. The "subscribed" values vary, with the highest being 1909 and the lowest being 220.
4. There appears to be a general downward trend in the "subscribed" values as the "day" values increase, suggesting a potential decrease in subscriptions over time.
5. There are a few instances where the "subscribed" values remain relatively stable despite changes in the "day" values, such as around days 16-17, 20-21, and 27-28.
6. Day 19 has the highest number of subscriptions with 1909, indicating a potential peak in subscriptions during that day.
7. Days 0 and 30 have the lowest number of subscriptions with 220 and 460, respectively, suggesting lower activity or interest in subscriptions at the beginning and end of the month.

These observations provide an initial overview of the data and may serve as a basis for further analysis or exploration.

In [ ]:
```
# Pdays vs subscribed
train.groupby(['pdays'])['subscribed'].count().reset_index().sort_values(by='su
```

Observations:

The timing of client contacts, as represented by "pdays," can have an impact on subscription rates, with shorter time intervals since the last contact generally associated with higher subscription counts. However, this relationship is not strictly linear, and other factors may also play a role in determining subscription outcomes. Further analysis may be needed to explore these relationships in more detail.

In [ ]:
```
# duration vs subscribed
train.groupby(['duration'])['subscribed'].count().reset_index().sort_values(by=
```

Observations:

It suggest that contact duration isnot that imapact on detecting subscribe

In [ ]:
```
#Visulazing it
plt.figure(figsize=(20,25))
p=1
for i in num_col:
    if p<=8:
        plt.subplot(4,2,p)
        sns.distplot(train[train['subscribed'] == "yes"][i], label='Subscribed'
        sns.distplot(train[train['subscribed'] == "no"][i], label='not subscrib
        plt.title(f'Distribution of {i} vs Subscribed')
        plt.xlabel(i, fontsize=16)
        plt.legend()
    p+=1
plt.show()
```

Outlier Analysis & Treatment

a. TRAIN Data Outlier detection

In [ ]:
```
plt.figure(figsize=(20,6))
sns.boxplot(train)
plt.show()
```

Handling Outliers using Zscore

In [ ]:
```
z = np.abs(zscore(train[['age','balance','duration','campaign','pdays','previou

#dropping outliers
train_new=train[(z<3).all(axis=1)]

#Data loss percentage after using Zscore
print("Old DataFrame: ",train.shape[0])
print("New DataFrame: ",train_new.shape[0])
data_loss_percentage= ((train.shape[0]-train_new.shape[0])/train.shape[0])*100
print("Data loss Percentage: {:.2f}".format(data_loss_percentage))
```

In [ ]:
```
#Checking with IQR method
Q1=train.quantile(0.25)
Q3=train.quantile(0.75)

IQR = Q3-Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
train2=train[~((train<(Q1-1.5 *IQR))|(train>(Q3+1.5*IQR))).any(axis=1)]
print("Data Loss Percentage After removing Outliers with IQR method: ",((train.
```

Observation:

The Z-score method for outlier removal demonstrates a substantial reduction in data loss, with only 11.13% compared to the 37.44% observed with the IQR method. This pronounced disparity in data loss percentages strongly suggests that the Z-score method is the preferred approach for managing outliers in this dataset.

Despite our efforts to address outliers, it is apparent that some still persist within our dataframe. Consequently, we will employ the Robust Scaler, known for its capability to effectively centers the data around the median and scales it by the spread of the middle 50% of the data, making it less sensitive to extreme values (outliers) compared to other scaling methods.

b. TEST Data Outlier detection

In [ ]:
```
plt.figure(figsize=(20,6))
sns.boxplot(test)
plt.show()
```

In [ ]:
```
z = np.abs(zscore(test[['age','balance','duration','campaign','pdays','previous

#dropping outliers
test_new=test[(z<3).all(axis=1)]

#Data loss percentage after using Zscore
print("Old DataFrame: ",test.shape[0])
print("New DataFrame: ",test_new.shape[0])
data_loss_percentage= ((test.shape[0]-test_new.shape[0])/test.shape[0])*100
print("Data loss Percentage: {:.2f}".format(data_loss_percentage))
```

Observations:

The 'Data loss Percentage' after using the Z-score method for outlier removal is calculated to be approximately 11.28%. This percentage represents the proportion of data points that were identified as outliers and subsequently removed from the test dataset.

It's important to note that we performed outlier removal in the test dataset following the same approach used for the train dataset. This consistency in data preprocessing ensures that our test

Feature Engineering

a. TEST Data

Converting Categorical column to numeric column

In [ ]:
```
# Define a dictionary for replacing 'yes' and 'no'
yes_no_mapping = {'yes': 1, 'no': 0}

# Replace values in the specified columns
columns_to_replace = ['default', 'housing', 'loan', 'subscribed']
train[columns_to_replace] = train[columns_to_replace].replace(yes_no_mapping)

# Define a dictionary for replacing months
month_mapping = {
    'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
    'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
    'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12
}


# Replace month values
train['month'] = train['month'].replace(month_mapping)

# Replace 'unknown' job values with 'other'
train['job'].replace('unknown', 'other', inplace=True)

# Print the updated DataFrame
train.head()
```

In [ ]:
```
# Encoding other categorical columns using LabelEncoder
le=LabelEncoder()
train['job']=le.fit_transform(train['job'])
train['education']=le.fit_transform(train['education'])
train['poutcome']=le.fit_transform(train['poutcome'])
train['marital']=le.fit_transform(train['marital'])
train['contact']=le.fit_transform(train['contact'])
```

In [ ]:
```
train.head()
```

Observations:

We have enconded all the categorical column into numerical column

b. TEST Data

In [ ]:
```
# Define a dictionary for replacing 'yes' and 'no'
yes_no_mapping = {'yes': 1, 'no': 0}

# Replace values in the specified columns
columns_to_replace = ['default', 'housing', 'loan']
test[columns_to_replace] = test[columns_to_replace].replace(yes_no_mapping)

# Define a dictionary for replacing months
month_mapping = {
    'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
    'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
    'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12
}

# Replace month values
test['month'] = test['month'].replace(month_mapping)

# Replace 'unknown' job values with 'other'
test['job'].replace('unknown', 'other', inplace=True)

# Print the updated DataFrame
test.head()
```

In [ ]:
```
# Encoding other categorical columns using LabelEncoder
le=LabelEncoder()
test['job']=le.fit_transform(test['job'])
test['education']=le.fit_transform(test['education'])
test['poutcome']=le.fit_transform(test['poutcome'])
test['marital']=le.fit_transform(test['marital'])
test['contact']=le.fit_transform(test['contact'])
```

In [ ]:
```
test.head()
```

Observations:

It is evident that we have successfully transformed categorical columns into numerical format for both the training and testing datasets using label encoding.

Correlation Matrix

In [ ]:
```
#visulaizing it
plt.figure(figsize=(20, 15))
sns.heatmap(train.corr(), annot=True, fmt='.1F', cmap='coolwarm', annot_kws={"s
plt.show()
```

Observations:

Based on the correlation matrix, here are the observations:

1. **Correlations with the Target Variable (subscribed):**
    - The feature 'duration' has a relatively strong positive correlation with the target variable 'subscribed' (0.355888). This suggests that a longer duration of the last contact with the client is associated with a higher likelihood of subscription.
    - 'pdays' and 'previous' also show moderate positive correlations with 'subscribed,' indicating that previous contacts and the number of days since the last contact have some influence on subscription.
2. **Age and Marital Status:**
    - 'age' has a negative correlation with 'marital' (-0.411087), suggesting that younger clients are more likely to be married.
    - 'marital' and 'subscribed' have a positive correlation (0.056387), indicating that marital status may have some influence on subscription.
3. **Contact Method:**
    - 'contact' (contact method) has a relatively strong negative correlation with 'ID' (-0.734588) and 'default' (credit default), indicating that these features are somewhat related. This might suggest that certain contact methods are preferred or more effective for specific client profiles.
4. **Previous Marketing Campaigns:**
    - 'pdays' (number of days since the last contact) has strong positive correlations with 'previous' (number of previous contacts) and 'poutcome' (outcome of the previous marketing campaign). This is expected, as a longer time since the last contact would likely lead to more previous contacts and potentially different outcomes from previous campaigns.
5. **Default, Housing, and Loan:**
    - 'default' (credit default) has a positive correlation with 'balance' and 'previous,' but it doesn't show strong correlations with the other features or the target variable.
    - 'housing' (housing loan) has a negative correlation with 'duration' and a positive correlation with 'contact.' This suggests some relationships with call duration and the preferred contact method among those with housing loans.
    - 'loan' (personal loan) doesn't show strong correlations with other features or the target variable.
6. **Month and Day:**
    - 'month' and 'day' don't have strong correlations with most other features or the target variable, except for a moderate positive correlation between 'month' and 'poutcome' (0.040108). This might indicate some seasonality or temporal patterns in campaign outcomes.
7. **Campaign-Related Features:**
    - 'campaign' (number of contacts during this campaign) and 'poutcome' show some negative correlation (-0.086074), suggesting that more contacts in the current

campaign may lead to a less favorable outcome if the previous campaign didn't succeed.

8. **Balance:**

   - 'balance' doesn't show strong correlations with most other features or the target variable, except for a weak positive correlation with 'duration' (0.031060).

These observations provide insights into the relationships between different features in your dataset. They can be valuable for feature selection, understanding the data, and potentially building predictive models.

Data Preprocessing

In [ ]:
```
# Separating feature & Label

# Feature
x = train.drop(columns=["ID", "subscribed"])

# Target
y= train["subscribed"]
```

In [ ]:
```
x.shape
```

In [ ]:
```
y.shape
```

Observations:

We have effectively split the dataset into features and labels. Initially, our training dataset had dimensions (28124, 17). Now, with 'X' containing all the features, the shape of 'X' is (28124, 16), while 'y' contains our target variable, resulting in a shape of (28124,).

In [ ]:
```
#Dropping Column from test Dataset

test = test.drop("ID", axis=1)
test.shape
```

Feature Scaling/Normalization

Train Data

In [ ]:
```
# Instantiate the Robust Scaler
scaler = RobustScaler()

# Fit and transform the scaler on the features
x_scaled = scaler.fit_transform(x)

# Create a DataFrame view of the scaled features after preprocessing
scaled_df = pd.DataFrame(x_scaled, columns=x.columns)
scaled_df
```

b. Test Data

In [ ]:
```
# Instantiate the Robust Scaler
scaler = RobustScaler()

# Fit and transform the scaler on the features
test_scaled = scaler.fit_transform(test)

# Create a DataFrame view of the scaled features after preprocessing
scaled_test = pd.DataFrame(test_scaled, columns=test.columns)
scaled_test
```

Observations

It's clear that we've effectively applied the Robust Scaler to standardize all the features of train dataset as well as test dataset, benefiting from its robustness to outliers.

Handling Imbalanced Data

In [ ]:
```
smote = SMOTE()
x_resampled, y_resampled = smote.fit_resample(x, y)

# Previous vs New Class distribution
print('Before Sampling:\n', y.value_counts())
print('\nAfter Sampling:\n', y_resampled.value_counts())
```

Observations:

By applying sampling techniques, we have balanced the class distribution, ensuring that both classes now have an equal number of samples. This can help improve the performance and fairness of machine learning models, particularly in cases where class imbalance can impact

model predictions

Data Splitting

In [ ]:
```
x_train, x_test, y_train, y_test= train_test_split(x_resampled, y_resampled, ra
```

In [ ]:
```
print('Shape of x_train',x_train.shape)
print('Shape of x_test',x_test.shape)
print('Shape of y_train',y_train.shape)
print('Shape of y_test',y_test.shape)
```

Observations:

The dataset has been successfully split into training and testing sets.This separation of data into training and testing subsets allows you to develop and evaluate machine learning models using the specified dataset.

Model Selection

In [ ]:
```
# List of classifiers
classifiers = {
    'Decision Tree': DecisionTreeClassifier(),
    'Logistic Regression': LogisticRegression(),
    'Random Forest': RandomForestClassifier(),
    'Support Vector Machine': SVC(),
    'Extra Tree': ExtraTreesClassifier(),
}
```

Cross Validation

In [ ]:
```
# Dictionary to store results
results = {'Classifier': [], 'Accuracy': [], 'Precision': [], 'Recall': [], 'F1

# Perform 10-fold cross-validation for each classifier and store results
for clf_name, clf in classifiers.items():
    scores = cross_val_score(clf, x_resampled, y_resampled, cv=10, scoring='acc

    # Predict on cross validation
    y_pred = cross_val_predict(clf, x_resampled, y_resampled, cv=5)

    # Compute the metrics: Mean Accuracy, Precision, Recall, F1-Score
    mean_accuracy = np.mean(scores)
    precision = precision_score(y_resampled, y_pred)
    recall = recall_score(y_resampled, y_pred)
    f1_score_value = f1_score(y_resampled, y_pred)


    results['Classifier'].append(clf_name)
    results['Accuracy'].append(mean_accuracy)
    results['Precision'].append(precision)
    results['Recall'].append(recall)
    results['F1 Score'].append(f1_score_value)

# Create a DataFrame from the results dictionary
results_df = pd.DataFrame(results)
```

In [ ]:
```
# Print the results as a DataFrame
results_df.sort_values(by='F1 Score', ascending=False)
```

Observations:

Selection and Use of Models:

Based on the provided metrics, we would select the following three models:

1. **Extra Tree:** This model has a high accuracy, precision, recall, and F1 score. It should be selected when a high overall performance is crucial, such as in applications where both false positives and false negatives are costly.
2. **Random Forest:** Similar to Extra Tree, Random Forest also performs exceptionally well across all metrics. It is a robust choice for general classification tasks and can handle a variety of datasets effectively.

These three models can be considered as top candidates for further evaluation or deployment, depending on the specific requirements and constraints of the task at hand.

Hyperparameter Tuning

In [ ]:
```
# Instantiate Parameters for Exta Trees
param_ET = {
        'n_estimators': [50, 100, 200],
        'criterion': ['gini', 'entropy','log_loss'],
        'max_depth': [None, 5, 10, 15, 20],
        'min_samples_split': [2, 5, 10]
    }

# Instantiate Parameters for Random Forest

param_rf = {
        'n_estimators': [50, 100, 200],
        'criterion': ['gini', 'entropy'],
        'max_depth': [None, 5, 10, 15, 20],
        'min_samples_split': [2, 5, 10]
    }
```

a. Hyper Parameter Tuning for Extra Trees Classifier

In [ ]:
```
# Instantiate The Extra Trees Classifier object
clf_ET = ExtraTreesClassifier()

#Instantiate Randomized Search CV for Extra Trees Classifier
random_search_ET = RandomizedSearchCV(clf_ET, param_distributions= param_ET,
                                      n_iter=10, scoring='accuracy', cv=5, n_job

#Fit the Data
random_search_ET.fit(x_resampled, y_resampled)
# Print the Best parameters and best score
print(random_search_ET.best_params_)
print(random_search_ET.best_score_)
```

In [ ]:
```
# Instantiate the Extra Trees Classifier with hyperparameter tuning
clf_ET = ExtraTreesClassifier(n_estimators=200, min_samples_split=10, max_depth

# Fit the training data to the Extra Trees Classifier
clf_ET.fit(x_train, y_train)

# Make predictions on the test data using the same classifier
y_pred_ET = clf_ET.predict(x_test)

# Print the classification report for the Extra Trees Classifier
print("Extra Trees Classifier with Hyperparameter Tuning:\n")
print(classification_report(y_test, y_pred_ET))
```

b. Hyper Parameter Tuning for Random Forest Classifier

In [ ]:
```
# Instantiate the Random Forest Classifier Object
clf_rf =  RandomForestClassifier()

# Instantiate Randomized Search CV for Random Forest Classifier
random_search_rf = RandomizedSearchCV(
    clf_rf, param_distributions=param_rf,
    n_iter=10, scoring='accuracy', cv=5, n_jobs=-1, random_state=42
)

# Fit the Data with hyperparameter tuning
random_search_rf.fit(x_resampled, y_resampled)

# Print the Best parameters and best score from hyperparameter tuning
print("Best Parameters:", random_search_rf.best_params_)
print("Best Score:", random_search_rf.best_score_)
```

In [ ]:
```
# Instantiate the Random Forest Classifier with the best hyperparameters
clf_rf = RandomForestClassifier(**random_search_rf.best_params_)

# Fit the training data to the Random Forest Classifier
clf_rf.fit(x_train, y_train)

# Make predictions on the test data using the same classifier
y_pred_rf = clf_rf.predict(x_test)

# Print the classification report for the Random Forest Classifier with hyperpa
print("\nRandom Forest Classifier with Hyperparameter Tuning:\n")
print(classification_report(y_test, y_pred_rf))
```

Plotting Confusion Matrix

- Extra Trees Classifier
- Random Forest Classifier

In [ ]:
```
# Calculate confusion matrices for both models
cm_ET = confusion_matrix(y_test, y_pred_ET)
cm_rf = confusion_matrix(y_test, y_pred_rf)

# Create subplots for the confusion matrices
fig, axes = plt.subplots(1, 2, figsize=(8, 5))

# Plot Extra Trees Classifier Confusion Matrix
sns.heatmap(cm_ET, annot=True, fmt="d", cmap="Blues", ax=axes[0])
axes[0].set_title('Extra Trees Classifier Confusion Matrix')
axes[0].set_xlabel('Predicted Labels')
axes[0].set_ylabel('True Labels')

# Plot Random Forest Confusion Matrix
sns.heatmap(cm_rf, annot=True, fmt="d", cmap="Blues", ax=axes[1])
axes[1].set_title('Random Forest Confusion Matrix')
axes[1].set_xlabel('Predicted Labels')
axes[1].set_ylabel('True Labels')

plt.tight_layout()
plt.show()
```

Observations:

Both the Random Forest Classifier and the Extra Trees Classifier with hyperparameter tuning have very similar performance metrics, including accuracy, precision, recall, and F1-score.

ROC-AUC Curve:

- Extra Trees Classifier
- Random Forest Classifier

In [ ]:
```
# Get predicted probabilities for the positive class
y_prob = clf_ET.predict_proba(x_test)[:, 1]
# Calculate ROC AUC score
auc_score = roc_auc_score(y_test, y_prob)
print("ROC AUC Score Of Extra Tress Classifier:", auc_score)

# Get predicted probabilities for the positive class
y_prob = clf_rf.predict_proba(x_test)[:, 1]
# Calculate ROC AUC score
auc_score = roc_auc_score(y_test, y_prob)
print("ROC AUC Score Of Random Forest Classifier:", auc_score)
```

In [ ]:
```
# Get predicted probabilities for the positive class for both classifiers
y_prob_et = clf_ET.predict_proba(x_test)[:, 1]
y_prob_rf = clf_rf.predict_proba(x_test)[:, 1]

# Calculate ROC AUC scores for both classifiers
auc_score_et = roc_auc_score(y_test, y_prob_et)
auc_score_rf = roc_auc_score(y_test, y_prob_rf)

# Calculate ROC curves for both classifiers
fpr_et, tpr_et, _ = roc_curve(y_test, y_prob_et, pos_label=1)
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_prob_rf, pos_label=1)

# Create subplots
plt.figure(figsize=(8, 4))

# Plot ROC curve for Extra Trees Classifier
plt.subplot(1, 2, 1)
plt.plot(fpr_et, tpr_et, label='Extra Trees ROC Curve (AUC = {:.2f})'.format(au
plt.plot([0, 1], [0, 1], 'r--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve - Extra Trees')
plt.legend(loc='lower right')

# Plot ROC curve for Random Forest Classifier
plt.subplot(1, 2, 2)
plt.plot(fpr_rf, tpr_rf, label='Random Forest ROC Curve (AUC = {:.2f})'.format(
plt.plot([0, 1], [0, 1], 'r--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve - Random Forest')
plt.legend(loc='lower right')

# Adjust layout and show plots
plt.tight_layout()
plt.show()
```

Observations:

Both models have very high ROC AUC scores, suggesting that they are performing well in terms of distinguishing between the classes. However, the Random Forest Classifier has a slightly higher ROC AUC score, indicating slightly better performance in this specific metric.

Predicting test data based on training data using best model

In [ ]:
```
test_pred = clf_ET.predict(test)
test_pred
```

```
with open('best_model','wb') as f:
```

```
    pickle.dump(clf_ET,f)
```

```
pickle.dump(scaler, open('scaler.pkl','wb'))
```