Predicting Term Deposit Subscription

Objective

The objective of this dataset is to build a EDA with the purpose to improve the marketing strategy of a bank for term deposit subscription and further model building. We can resume our our workings steps as the following:

- Initial data analysis
- Data cleaning
- Exploratory data analysis
- Model Building
- Results

Dataset

This infomation was extracted from dataset kaggle page bellow:

https://www.kaggle.com/datasets/prakharrathi25/banking-dataset-marketing-targets

Detailed Column Descriptions bank client data:

- 1 - age (numeric)
- 2 - job : type of job (categorical: "admin.","unknown","unemployed","management","housemaid","entrepreneur","student",
- "blue-collar","self-employed","retired","technician","services")
- 3 - marital : marital status (categorical: "married","divorced","single"; note: "divorced" means divorced or widowed)
- 4 - education (categorical: "unknown","secondary","primary","tertiary")
- 5 - default: has credit in default? (binary: "yes","no")
- 6 - balance: average yearly balance, in euros (numeric)
- 7 - housing: has housing loan? (binary: "yes","no")
- 8 - loan: has personal loan? (binary: "yes","no")

  related with the last contact of the current campaign:
- 9 - contact: contact communication type (categorical: "unknown","telephone","cellular")
- 10 - day: last contact day of the month (numeric)
- 11 - month: last contact month of year (categorical: "jan", "feb", "mar", ..., "nov", "dec")
- 12 - duration: last contact duration, in seconds (numeric)

  other attributes:
- 13 - campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
- 14 - pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric, -1 means client was not previously contacted)
- 15 - previous: number of contacts performed before this campaign and for this client (numeric)
- 16 - poutcome: outcome of the previous marketing campaign (categorical: "unknown","other","failure","success")

Output variable (desired target):

- 17 - y - has the client subscribed a term deposit? (binary: "yes","no")

Missing Attribute Values: None

Citation This dataset is publicly available for research. It has been picked up from the UCI Machine Learning with random sampling and a few additional columns.

Please add this citation if you use this dataset for any further analysis.

S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, June 2014

Past Usage The full dataset was described and analyzed in:

S. Moro, R. Laureano and P. Cortez. Using Data Mining for Bank Direct Marketing: An Application of the CRISP-DM Methodology. In P. Novais et al. (Eds.), Proceedings of the European Simulation and Modelling Conference - ESM'2011, pp. 117-121, Guimarães, Portugal, October, 2011. EUROSIS.

OBS: We opt to not use test data since it has data leakage as the publiser stated: "test.csv: 4521 rows and 18 columns with 10% of the examples (4521), randomly selected from train.csv" . This is very harmfull for the models, since it gives known data during the model fitting process and gives unrealistic results as consequence.

IMPORTANT OBSERVATION : INTERACT GENERATED VISUALIZATIONS WON'T WORK UNLESS YOU RUN IT YOURSELF!!!!

Importing Libraries

In [5]:

```
# EDA Libraries
import matplotlib.pyplot as plt
import pandas as pd
import pickle
import pingouin
import seaborn as sns
from ipywidgets import Dropdown, interact, IntText
import plotly.express as px
import numpy as np
from datetime import datetime

# Machine Learning Libraries
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, AdaBoost
Classifier
from ipywidgets import Dropdown, interact
from category_encoders import OrdinalEncoder, OneHotEncoder
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.model_selection import GridSearchCV, cross_val_score, train_test_split, Rand
omizedSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, ConfusionMatri
xDisplay, make_scorer, classification_report
from sklearn.tree import DecisionTreeClassifier
from imblearn.over_sampling import ADASYN, SMOTE
pd.set_option("display.max_columns", None)
```

Class created for EDA

In [2]:

```
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import matplotlib.pyplot as plt
import seaborn as sns
from ipywidgets import Dropdown, interact



class Columnator:
```

```python
    def __init__(self, df, target, df_test=False, labels=None):
        """
        A class that creates bar plots for categorical variables in a Pandas DataFrame

        Parameters:
        df (Pandas DataFrame): The input DataFrame
        target (str): The name of the target variable in the DataFrame
        df_test (bool, optional): A flag indicating whether to use a separate test DataFrame, defaults to False
        labels (dict, optional): A dictionary that maps the target variable's values to more meaningful labels, defaults to None
        """
        self.df = df
        self.df_test = df_test
        self.target = target
        self.labels = labels

    def comparator_categoric(self, column, normalize=False):
        """
        A method that creates a bar plot of the number of survivors in each category of a categorical variable.

        Parameters:
        column (str): The name of the categorical variable
        normalize (bool, optional): A flag indicating whether to normalize the counts, defaults to False

        Returns:
        None
        """

        # Group the DataFrame by the specified column and count the number of survivors for each category
        df_columnator = self.df.groupby(column)[self.target].value_counts(normalize=normalize).to_frame().rename(columns={self.target: 'Number'}).reset_index()

        if self.labels != None:
            # Map the labels to the 'Survived' column
            df_columnator[self.target] = df_columnator[self.target].map(self.labels)

        # If normalize == True
        if normalize:
            df_columnator['Number'] = df_columnator['Number'] * 100

        # Create a bar plot using seaborn, with the specified column as the x-axis, number of survivors as the y-axis, and survival status as the hue
        plt.figure(figsize=(15,11))

        if self.df[column].nunique() < 7:
            ax = sns.barplot(y='Number', x=column, hue=self.target, data=df_columnator)

            # Add annotations to the bars showing the exact percentage or number of survivors depending on the "normalize" parameter
            for p in ax.patches:
                ax.annotate(format(p.get_height(), '.2f'),
                            (p.get_x() + p.get_width() / 2, p.get_height()),
                            ha='center', va='center', xytext=(0, 10), textcoords='offset points');

            # If normalize=True, convert number of survivors to a percentage and set the y-axis label accordingly
            if normalize:
                plt.yticks(range(0, 101, 10))
                plt.ylabel('Number [%]')


        else:
            ax = sns.barplot(x='Number', y=column, hue=self.target, data=df_columnator)

            # Add annotations to the bars showing the exact percentage or number of survivors depending on the "normalize" parameter
            for p in ax.patches:
```

```
                ax.annotate(format(p.get_width(), '.2f'),
                            (p.get_width(), p.get_y() + p.get_height() / 2),
                            xytext=(5, 0),
                            textcoords='offset points',
                            ha='left', va='center')

            # If normalize=True, convert number of survivors to a percentage and set the
y-axis label accordingly
            if normalize:
                plt.xticks(range(0, 101, 10))
                plt.xlabel('Number [%]')

        # Set the figure title and legend title
        plt.title(f"{column} x {self.target}", fontsize=18)
        plt.legend(title=f"{self.target}")


        # Remove the top and right spines of the plot
        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)



    def dashbordator_categoric(self):
        """
        A method that creates an interactive dashboard for categorical variables in a Pan
das DataFrame

        Parameters:
        None

        Returns:
        A panel object containing the interactive dashboard
        """
        panel1 = interact(
            self.comparator_categoric,
            column=Dropdown(options=self.df.dtypes[(self.df.dtypes == "object") | (self.
df.dtypes == 'bool')].index)
        );
        return panel1;


    def comparator_numeric(self, column):

        # Get the data for the 'column' feature for survivors and non-survivors
        survived_true = self.df[self.df[self.target] == True][column]
        survived_false = self.df[self.df[self.target] == False][column]

        # Create a figure with two histograms, one for survivors and one for non-survivor
s
        fig = make_subplots()

        fig.add_trace(
            go.Histogram(x = survived_true, nbinsx=20)
        )

        fig.add_trace(
            go.Histogram(x = survived_false, nbinsx=20)
        )

        # Set the figure layout
        fig.update_layout(
            title_text=f"{column} x {self.target}"
        )

        # Set the x-axis label
        fig.update_xaxes(title_text=column)

        # Set the y-axis label
        fig.update_yaxes(title_text=f"Frequency", secondary_y=False)

        # Show the figure
```

```
        fig.show()

    def dashbordator_numeric(self):
        panel1 = interact(
            self.comparator_numeric,
            column=Dropdown(options= self.df.dtypes[(self.df.dtypes != 'object') & (self
.df.dtypes != 'bool')].index)
        );
        return panel1;
```

Loading Data

In [3]:

```
df = pd.read_csv('bank.csv', sep=';')
```

Initial Analysis

Functions

In [4]:

```
def columnator_frequency(column, boxplot = False, normalize = False):
    if df[column].dtype == 'object' or df[column].dtype == 'bool':
        df[column].value_counts(normalize = normalize).plot(kind= 'barh')
        plt.xlabel('Frequency')
        plt.ylabel(f'{column.capitalize()}');
    else:
        if boxplot == False:
            df[column].hist()
            plt.ylabel('Frequency')
            plt.xlabel(f'{column.capitalize()}');
        else:
            plt.boxplot(df[column])

    plt.title(f'{column.capitalize()} Frequency')
```

Analysis

First look at our data.

In [5]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>

RangeIndex: 45211 entries, 0 to 45210

Data columns (total 17 columns):

 #   Column     Non-Null Count  Dtype

---  ------     --------------  -----

 0   age        45211 non-null  int64

 1   job        45211 non-null  object

 2   marital    45211 non-null  object

 3   education  45211 non-null  object

 4   default    45211 non-null  object
```

```
 5   balance    45211 non-null  int64
 6   housing    45211 non-null  object
 7   loan       45211 non-null  object
 8   contact    45211 non-null  object
 9   day        45211 non-null  int64
 10  month      45211 non-null  object
 11  duration   45211 non-null  int64
 12  campaign   45211 non-null  int64
 13  pdays      45211 non-null  int64
 14  previous   45211 non-null  int64
 15  poutcome   45211 non-null  object
 16  y          45211 non-null  object
dtypes: int64(7), object(10)
memory usage: 5.9+ MB
```

In [6]:

```
df.head()
```

Out[6]:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1

In [7]:

```
df.shape
```

Out[7]:

```
(45211, 17)
```

In [8]:

```
df.duplicated().sum()  #Checking if there was any duplicated row.
```

Out[8]:

```
0
```

As we can see above, no duplications were found.

In [9]:

```
df.dtypes # Show data types of our dataset.
```

Out[9]:

```
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
```

In [10]:

```
df.isna().sum()  #Check if there are any nan values in each column.
```

Out[10]:

```
age          0
job          0
marital      0
education    0
default      0
balance      0
housing      0
loan         0
contact      0
day          0
month        0
duration     0
campaign     0
pdays        0
previous     0
poutcome     0
y            0
dtype: int64
```

As seen above, there were no missing values on our dataframe

In [11]:

```
df.describe().T # Shows general statistics from numeric columns.
```

Out[11]:

	count	mean	std	min	25%	50%	75%	max
age	45211.0	40.936210	10.618762	18.0	33.0	39.0	48.0	95.0
balance	45211.0	1362.272058	3044.765829	-8019.0	72.0	448.0	1428.0	102127.0
day	45211.0	15.806419	8.322476	1.0	8.0	16.0	21.0	31.0
duration	45211.0	258.163080	257.527812	0.0	103.0	180.0	319.0	4918.0
campaign	45211.0	2.763841	3.098021	1.0	1.0	2.0	3.0	63.0
pdays	45211.0	40.197828	100.128746	-1.0	-1.0	-1.0	-1.0	871.0
previous	45211.0	0.580323	2.303441	0.0	0.0	0.0	0.0	275.0

In [12]:

```
df[df['previous'] == 0].shape
```

Out[12]:

```
(36954, 17)
```

Considering the table above we can jump to the following conclusions:

- **'balance' column has a very high standard deviation and huge outliers values, considering that 75% of the data has a value of bellow 1428.**
- **we can see that 50% of the bank clients are bellow 40 years.**
- **we can see that 'duration' column has outilers values since there is a jump from 319, that represents 75% of the data, to max value of 4918. Also the standard deviation is very high, surpassing the median value.**
- **As shown on 'pdays' column, at least 75% of bank costumers were never contacted before.**
- **'previous' column follows the same pattern from 'pdays' column, since clients were not contacted during the last campaingn.**

In [13]:

```
df.corr('pearson').style.background_gradient(axis=None)
```

Out[13]:

	age	balance	day	duration	campaign	pdays	previous
age	1.000000	0.097783	-0.009120	-0.004648	0.004760	-0.023758	0.001288
balance	0.097783	1.000000	0.004503	0.021560	-0.014578	0.003435	0.016674
day	-0.009120	0.004503	1.000000	-0.030206	0.162490	-0.093044	-0.051710
duration	-0.004648	0.021560	-0.030206	1.000000	-0.084570	-0.001565	0.001203
campaign	0.004760	-0.014578	0.162490	-0.084570	1.000000	-0.088628	-0.032855
pdays	-0.023758	0.003435	-0.093044	-0.001565	-0.088628	1.000000	0.454820
previous	0.001288	0.016674	-0.051710	0.001203	-0.032855	0.454820	1.000000

As shown on the correlation matrix above, there are no significant correlation between the numeric columns.

In [14]:

```
panel_columnator = interact(
     columnator_frequency,
     column=Dropdown(options= df.drop(columns='y')),
 );
```

- **'age' column looks like it follows a normal distribution. It will be checked bellow via Shapiro Wilk test.**
- **'blue-collar', 'management' and technicians represent more than 58% of clients occupation in our data.**
- **60% of our clients are married.**
- **50% of our clients have secondary degree and almost 30% have tertiary degree.**
- **more than 95% of our data reprensents people that did not have given default.**
- **75% of our data has less than 1428 euros in balance. 99% of our that has less than 13164 euros in balance. We can observe outliers with less than 0 euro in account. This will be treated by creating a new categoric column to simplify those informations and treat outliers.**
- **more than 50% of our data has housing loan.**
- **more than 80% of our that dosen't have any kind of loan.**
- **'contact column has many 'unknown' values. We decided to "leave them be".**
- **day is a problematic column and will be dropped.**
- **considering how the campaings were made, we can expeculate that on winter there were less contacts been made, with summer being the prefered season for campaings.**
- **there are many outliers in 'duration' column, and they will be threated on next steps.**
- **at least 75% of our data was contacted at least 3 times.**
- **at least 75% of our data was not previously contacted.**
- **this column will be dropped since most values are unknown.**

In [15]:

```
pingouin.normality(df['age'], method='shapiro', alpha=0.05) # After running Shapiro Wilk
```

```
normality test, 'age' column distribution seems not to be Normal.
```

```
C:\Users\Benito de la Torre\anaconda3\lib\site-packages\scipy\stats\_morestats.py:1816: U
serWarning: p-value may not be accurate for N > 5000.

  warnings.warn("p-value may not be accurate for N > 5000.")
```

Out[15]:

	W	pval	normal
age	0.960546	0.0	False

As seen above via Shapiro Wilk test, 'age' column is not normaly distributed.

Data Cleaning

In [16]:

```
# Function that will be used to generate a new categoric column from 'balance' values.

def balanceator(x):
    if x < 72:
        return 'Class E'
    elif x >= 72 and x < 448:
        return 'Class D'
    elif x >= 448 and x < 1428:
        return 'Class C'
    elif x >= 1428 and x < df['balance'].quantile(0.99):
        return 'Class B'
    else:
        return 'Class A'
```

In [17]:

```
def wrangle(path):
    df = pd.read_csv(path, sep=';') # Read CSV file
    df['y'] = df['y'].apply(lambda x: True if x == 'yes' else False) # Change object out
put to bool
    df['default'] = df['default'].apply(lambda x: True if x == 'yes' else False) # Chang
e object output to bool
    df['balance_class'] = df['balance'].apply(lambda x: balanceator(x)) # Creates a new
categoric column 'balance_class' using data from 'balance' column
    df['housing'] = df['housing'].apply(lambda x: True if x == 'yes' else False) # Chang
e object output to bool
    df['loan'] = df['loan'].apply(lambda x: True if x == 'yes' else False) # Change obje
ct output to bool
    df['previous_bool'] = df['previous'].apply(lambda x: True if x != 0 else False) # Ch
ange object output to bool for visualization and modeling purpuses

    #dealing with outliers by capping them with 3 times std + mean.
    outliers = ['duration']
    upper_limit = df[outliers].mean() + 3*df[outliers].std()
    for i in upper_limit.index:
        df[i] = df[i].apply(lambda x: upper_limit.loc[i] if x > upper_limit.loc[i] else
x)

    #drop columns:
    to_drop =['previous', 'day', 'poutcome', 'pdays']
    df.drop(columns= to_drop, inplace=True)


    return df
```

In [18]:

```
df_pos = wrangle('bank.csv')
```

Checking data post wrangling:

In [19]:

```
df_pos.head()
```

Out[19]:

	age	job	marital	education	default	balance	housing	loan	contact	month	duration	campaign	y	bal
0	58	management	married	tertiary	False	2143	True	False	unknown	may	261.0	1	False	
1	44	technician	single	secondary	False	29	True	False	unknown	may	151.0	1	False	
2	33	entrepreneur	married	secondary	False	2	True	True	unknown	may	76.0	1	False	
3	47	blue-collar	married	unknown	False	1506	True	False	unknown	may	92.0	1	False	
4	33	unknown	single	unknown	False	1	False	False	unknown	may	198.0	1	False	

In [20]:

```
df_pos.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45211 entries, 0 to 45210
Data columns (total 15 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   age            45211 non-null  int64
 1   job            45211 non-null  object
 2   marital        45211 non-null  object
 3   education      45211 non-null  object
 4   default        45211 non-null  bool
 5   balance        45211 non-null  int64
 6   housing        45211 non-null  bool
 7   loan           45211 non-null  bool
 8   contact        45211 non-null  object
 9   month          45211 non-null  object
 10  duration       45211 non-null  float64
 11  campaign       45211 non-null  int64
 12  y              45211 non-null  bool
 13  balance_class  45211 non-null  object
 14  previous_bool  45211 non-null  bool
dtypes: bool(5), float64(1), int64(3), object(6)
memory usage: 3.7+ MB
```

In [21]:

```
df_pos.describe()
```

Out[21]:

	age	balance	duration	campaign
count	45211.000000	45211.000000	45211.000000	45211.000000
mean	40.936210	1362.272058	250.772487	2.763841
std	10.618762	3044.765829	220.986371	3.098021
min	18.000000	-8019.000000	0.000000	1.000000
25%	33.000000	72.000000	103.000000	1.000000
50%	39.000000	448.000000	180.000000	2.000000
75%	48.000000	1428.000000	319.000000	3.000000
max	95.000000	102127.000000	1030.746517	63.000000

Data Analysis

Functions

In [22]:

```
columnator_instansator = Columnator(df_pos, 'y')
```

In [23]:

```
def columnator_frequency2(column, boxplot = False, normalize = False):
    if df_pos[column].dtype == 'object' or df_pos[column].dtype == 'bool':
        df_pos[column].value_counts(normalize = normalize).plot(kind= 'barh')
        plt.xlabel('Frequency')
        plt.ylabel(f'{column.capitalize()}');
    else:
        if boxplot == False:
            df_pos[column].hist()
            plt.ylabel('Frequency')
            plt.xlabel(f'{column.capitalize()}');
        else:
            plt.boxplot(df_pos[column])

    plt.title(f'{column.capitalize()} Frequency')
```

In [24]:

```
# Create a mask to select only the rows in the dataset where the 'y' column is True.
mask_target_true = df_pos['y'] == True

# Define a function called tabelator_crosstab that takes in a column name as a parameter.
def tabelator_crosstab(column):
    # Use pd.crosstab() to create a cross-tabulation table that shows the relationship be
tween 'balance_class' and the specified column.
    # Use the mask_target_true to only include rows where 'y' is True in the calculation.
    # Normalize the table and multiply by 100 to get percentages.
    table = pd.crosstab(df_pos[mask_target_true]['balance_class'], df_pos[mask_target_tr
ue][column], normalize=True) * 100

    # Create a heatmap of the resulting table using seaborn.
    plt.figure(figsize= (20,12))
    sns.heatmap(table, annot=True, cmap='YlGnBu')
```

Checking data after wrangling:

In [25]:

```
panel_columnator = interact(
```

```
    columnator_frequency2,
    column=Dropdown(options= df_pos.drop(columns='y')),
);
```

- **As seen above 'balance_class' and 'previous_bool' where generated.**
- **'balance_class' column is better for visualization and generates a good simplification for model purposes.**

In [26]:

```
columnator_instansator.dashbordator_categoric()
```

Out[26]:

```
<function ipywidgets.widgets.interaction._InteractFactory.__call__.<locals>.<lambda>(*arg
s, **kwargs)>
```

- **Job column: Elderly and student demographics have a higher response rate of over 20% through marketing campaigns, with students having the highest response rate of 28%. However, entrepreneurs, laborers, homemakers, and service workers have lower response rates of less than 10%.**
- **Marital column: Single individuals have a higher response rate compared to other marital statuses.**
- **Education column: The response rate increases with the level of education, with primary education having a response rate of 8.6%, secondary education with 10.5%, and tertiary education with 15%.**
- **Default column: Individuals who do not have pre-approved credit have a higher response rate to the services (11.8% compared to 6.38%).**
- **Housing column: Individuals who do not have a housing loan have a higher response rate to the services (17.7% compared to 7.7%).**
- **Loan column: Individuals who do not have any loans have a higher response rate to the services (12.7% compared to 6.7%).**
- **Contact column: Individuals who received phone calls (mobile or landline) have a higher chance of responding to the services. However, it is worth noting that this column is problematic because the bank does not have the crucial information on how the customer was contacted during the campaing.**
- **Month column: This column needs further analysis. The months with higher response rates had less contact from the company, and the months with lower response rates coincided with the European winter.**

In [27]:

```
columnator_instansator.dashbordator_numeric()
```

Out[27]:

```
<function ipywidgets.widgets.interaction._InteractFactory.__call__.<locals>.<lambda>(*arg
s, **kwargs)>
```

- **as seen above, clients on their 20's are more susceptible to marketing campaings.**
- **considering 'duration' column, the more the duration on contact, the more susceptible the client is to subscribe for a term deposit.**

In [28]:

```
df_tabelator = df_pos[mask_target_true].drop(columns=['balance_class', 'y']).dtypes[(df_
pos.dtypes == 'object') | (df_pos.dtypes == 'bool')].index

panel1 = interact(
    tabelator_crosstab,
    column = Dropdown(options= df_tabelator))
```

On the heatmap above, we fixed our clients that subscribed for a term deposit to improve the targets of our marketing campaing. We choose to fix 'balance_class' column because it has one of the most important economic aspect of bank clients: their average yearly balance. With this information, we can try to trace a profile of each kind of client, that contais were does he work, if he has any kind of loan etc.

Report considerations:

- **9.5% from clients are Class B management workers.**
- **19% from clients are Class B and married.**
- **14% from clients are Class B and have secondary and tertiary education levels.**
- **33% of our clients are Class B that don't have a given default.**

Data Splitting

In [29]:

```
X = df_pos.drop(columns = ["y", "balance", 'duration'])
y = df_pos['y']
```

In [30]:

```
oe = OrdinalEncoder()
X = oe.fit_transform(X)
```

In [31]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=4
2, stratify=y)
```

Data Balancing

Given the unbalanced nature of our target variable, we have opted to utilize both SMOTE and ADASYN techniques in an effort to enhance our model's performance.

SMOTE

In [32]:

```
smote = SMOTE(random_state=42)
```

In [33]:

```
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
```

In [34]:

```
print(f"""
Original X shape: {X_train.shape}
SMOTE X shape: {X_train_smote.shape}
""")
```

```
Original X shape: (38429, 12)

SMOTE X shape: (67866, 12)
```

ADASYN

In [35]:

```
adasyn = ADASYN(random_state=42)
```

In [36]:

```
X_train_adasyn, y_train_adasyn = adasyn.fit_resample(X_train, y_train)
```

In [37]:

```
print(f"""
Original X shape: {X_train.shape}
ADASYIN X shape: {X_train_adasyn.shape}
""")
```

```
Original X shape: (38429, 12)

ADASYIN X shape: (67621, 12)
```

Defining Baseline

In [38]:

```
acc_baseline = y_train.value_counts(normalize=True).max()
print("Baseline Accuracy:", round(acc_baseline*100, 2), "%")
```

```
Baseline Accuracy: 88.3 %
```

Creating Class for Results Analysis

In [39]:

```
class Resultator():
    """
    A class for collecting and storing the results of different models.

    Attributes:
        data (pd.DataFrame): a DataFrame to store the results of the models.

    Methods:
        add_results: Add the results of a model to the data DataFrame.
        results: Return the results DataFrame.
    """
    def __init__(self):
        """
        Initialize an empty DataFrame with columns for storing the results of the models.
        """
        self.data = pd.DataFrame(columns=['Model', 'Accuracy CV', 'Acc Std CV', 'Recall 
CV', 'Rec Std CV', 'Precision CV', 'Precision std CV' , 'Test Acc', 'Test Recall', 'Test 
Precision'])

    def add_results(self, X_test, y_test, model, model_name):
        """
        Add the results of a model to the data DataFrame.

        Args:
            X_test (pd.DataFrame): the test data.
            y_test (pd.Series): the true labels for the test data.
            model (sklearn estimator): the trained model.
            model_name (str): the name of the model to use as a label in the results.

        Returns:
            None
        """
        # Initialize an empty dictionary to store the results
        results = {
            "Model": [],
            "Accuracy CV": [],
            "Acc Std CV": [],
            "Recall CV": [],
            "Rec Std CV": [],
```

```
            'Precision CV': [],
            "Precision std CV" : [],
            "Test Acc": [],
            "Test Recall": [],
            'Test Precision': []
        }

        # Extract the cross-validation results of the model
        cv_results = pd.DataFrame(model.cv_results_)

        # Add the model name, cross-validation accuracy and recall, and cross-validation
standard deviation to the results dictionary
        results["Model"].append(model_name)
        results["Accuracy CV"].append(round(cv_results[cv_results["rank_test_recall"] ==
1]["mean_test_accuracy"].iloc[0] * 100, 2))
        results["Acc Std CV"].append(round(cv_results[cv_results["rank_test_recall"] ==
1]["std_test_accuracy"].iloc[0] * 100, 2))
        results["Recall CV"].append(round(cv_results[cv_results["rank_test_recall"] == 1
]["mean_test_recall"].iloc[0] * 100, 2))
        results["Rec Std CV"].append(round(cv_results[cv_results["rank_test_recall"] ==
1]["std_test_recall"].iloc[0] * 100, 2))
        results['Precision CV'].append(round(cv_results[cv_results['rank_test_recall'] =
=1]['mean_test_precision'].iloc[0] * 100, 2))
        results['Precision std CV'].append(round(cv_results[cv_results['rank_test_recall
'] ==1]['std_test_precision'].iloc[0] * 100, 2))

        # Add the test accuracy and test recall to the results dictionary
        results["Test Acc"].append(round(accuracy_score(y_test, model.predict(X_test)) *
100, 2))
        results["Test Recall"].append(round(recall_score(y_test, model.predict(X_test))
* 100, 2))
        results['Test Precision'].append(round(precision_score(y_test, model.predict(X_t
est)) * 100, 2))

        # Concatenate the results DataFrame with a new DataFrame containing the results d
ictionary
        self.data = pd.concat([self.data, pd.DataFrame(results)])
        print(f"The Data from model {model_name} was acquired and stored.")

    def results(self):
        """
        Return the results DataFrame.

        Args:
            None

        Returns:
            pd.DataFrame: the results DataFrame.
        """
        return self.data

    def plot_results(self, column):
        fig = px.bar(data_frame=self.data.sort_values("Test Recall", ascending=False).he
ad(5),
                     y="Model",
                     x=f"{column}",
                     color= self.data.sort_values("Test Recall").head(5)["Model"],
                    title=f"{column} comparison")
        fig.update_layout(yaxis={'categoryorder':'total descending'}, xaxis_title=f"{col
umn}", yaxis_title="Models")
        fig.show()

    def dashbordator(self):
        """
        A method that creates an interactive dashboard for categorical variables in a Pan
das DataFrame

        Parameters:
        None

        Returns:
        A panel object containing the interactive dashboard
```

```
        """
        panel1 = interact(
            self.plot_results,
            column=Dropdown(options=self.data.drop(columns="Model").columns)
        );
        return panel1;
```

In [40]:

```
resultator = Resultator()
```

Model Building

The group has determined that the most important metric for this situation is recall. This is because we want the marketing team to target the clients with the highest likelihood of accepting the term deposit subscription, and recall measures the proportion of true positives (i.e., clients who would subscribe for the term deposit) among all actual positive cases (i.e., clients who were interested in subscribing). By optimizing for recall, we can ensure that the marketing team reaches out to as many interested clients as possible and maximizes the potential for subscription success.

Defining Parameters and Models

In [41]:

```
# Decision Tree Parameters

params_dt = {
    "max_depth": [5, 10, 15, 20, 25, 30, None], # Maximum depth of the decision tree
    "criterion": ["gini","entropy"], # The quality criterion to measure the information g
ain when splitting nodes
    "min_samples_split": [2,3], # Minimum number of samples required to split an internal
node
    "min_samples_leaf": [1,2] # Minimum number of samples required to be at a leaf node
}

# Random Forest Parameters
params_rf = {
    "n_estimators": range(50,251,50), # Number of decision trees in the random forest
    "max_depth": range(5,31,5), # Maximum depth of the decision trees in the random fore
st
    "min_samples_split": [2,3], # Minimum number of samples required to split an internal
node
    "min_samples_leaf": [1,2] # Minimum number of samples required to be at a leaf node
}

# KNN Parameters
params_knn = {
    "n_neighbors": range(20, 151, 10), # Number of neighbors to consider for each data po
int
    "weights": ["uniform","distance"] # The weight function used in prediction (uniform w
eights or weights based on inverse distance)
}
```

In [42]:

```
# Decision Tree Model
# Define a Decision Tree model with hyperparameters to be tuned and set the number of ite
rations for randomized search
model_dt = RandomizedSearchCV(
    DecisionTreeClassifier(random_state=42), # Define the Decision Tree model
    params_dt, # Pass in the hyperparameters to be tuned from the dictionary we defined e
arlier
    n_jobs=-1, # Use all available CPU cores for parallel computation
    cv=10, # Set the number of folds for cross-validation
    n_iter= 10, # Set the number of iterations for randomized search
    scoring=["recall", "accuracy", 'precision'], # Set the evaluation metrics to be used
```

```
for scoring
    refit="recall" # Choose the metric to optimize during randomized search
)

# Random Forest Model
# Define a Random Forest model with hyperparameters to be tuned and set the number of ite
rations for randomized search
model_rf = RandomizedSearchCV(
    RandomForestClassifier(random_state=42), # Define the Random Forest model
    params_rf, # Pass in the hyperparameters to be tuned from the dictionary we defined e
arlier
    n_jobs=-1, # Use all available CPU cores for parallel computation
    cv=10, # Set the number of folds for cross-validation
    n_iter=35, # Set the number of iterations for randomized search
    scoring=["recall", "accuracy", 'precision'], # Set the evaluation metrics to be used
for scoring
    refit="recall" # Choose the metric to optimize during randomized search
)

# KNN Model
# Define a KNN model with hyperparameters to be tuned and set the number of iterations fo
r GridSearch
model_knn = GridSearchCV(
    KNeighborsClassifier(), # Define the KNN model
    params_knn, # Pass in the hyperparameters to be tuned from the dictionary we defined
earlier
    n_jobs=-1, # Use all available CPU cores for parallel computation
    cv=10, # Set the number of folds for cross-validation
    scoring=["recall", "accuracy", 'precision'], # Set the evaluation metrics to be used
for scoring
    refit="recall" # Choose the metric to optimize during randomized search
)
```

DecisionTree

First we will build a DecisionTree model without applying any data balancing strategy or hyperparameter tunning to observe its behavior, like accuracy, recall and max depth.

In [43]:

```
dt = GridSearchCV(DecisionTreeClassifier(random_state=42), {}, n_jobs=-1, cv=10, scoring
=["recall", "accuracy", 'precision'], refit="recall")
dt.fit(X_train, y_train)
```

Out[43]:

```
▸ GridSearchCV
▸ estimator: DecisionTreeClassifier
  ▸ DecisionTreeClassifier
```

In [44]:

```
resultator.add_results(X_test, y_test, dt, "DecisionTreeBasic")
```

```
The Data from model DecisionTreeBasic was acquired and stored.
```

In [45]:

```
resultator.results()
```

Out[45]:

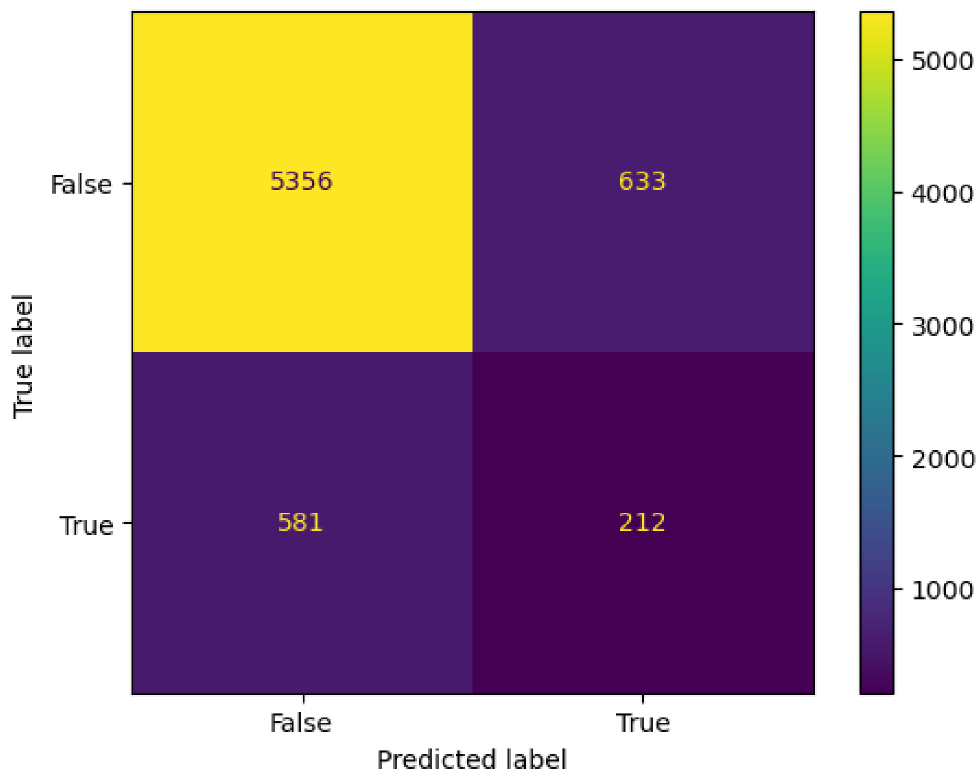
	Model	Accuracy CV	Acc Std CV	Recall CV	Rec Std CV	Precision CV	Precision std CV	Test Acc	Test Recall	Test Precision
0	DecisionTreeBasic	81.7	0.39	27.11	2.22	24.48	1.55	82.1	26.73	25.09

In [46]:

```
ConfusionMatrixDisplay.from_estimator(dt,X_test,y_test)
```

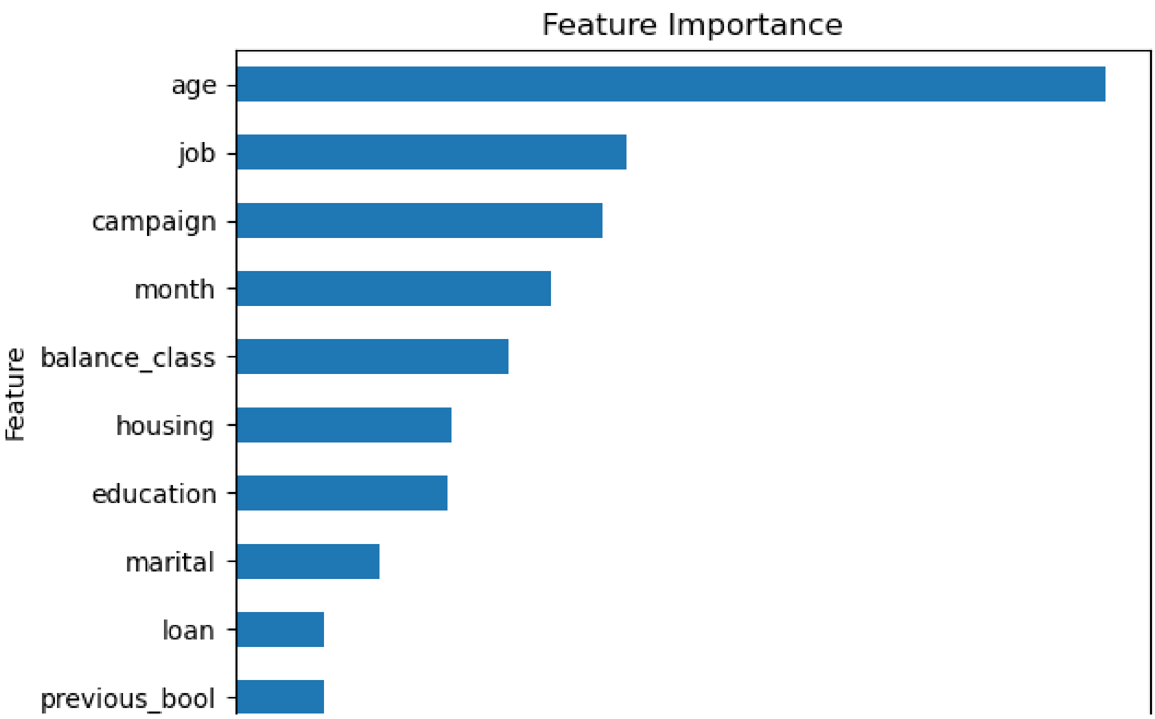
Out[46]:

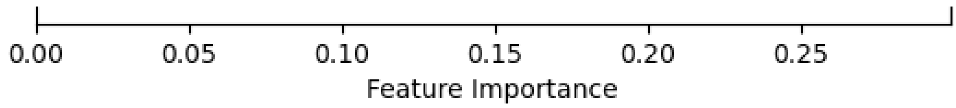
```
&lt;sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x205275a9b80&gt;
```



In [47]:

```
# Get feature names from training data
features = X_train.columns
# Extract importances from model
importances = dt.best_estimator_.feature_importances_
# Create a series with feature names and importances
feat_imp = pd.Series(importances, index=features)
# Plot 10 most important features
feat_imp.sort_values().tail(10).plot(kind="barh")
plt.xlabel("Feature Importance")
plt.ylabel("Feature")
plt.title("Feature Importance");
```





In [48]:

```
dt.best_estimator_.tree_.max_depth
```

Out[48]:

```
37
```

With the information acquired we can conclude that:

- **The model accuracy is inferior to the baseline in both cross-validation and test scores.**
- **The recall is bad and must be improved.**
- **The max_depth of a single tree without any hyperparameters tuning is 37.**
- **"age" column is by far the most "usefull" column, followed by 'job' and campaign.**

Models with original Data and Hyperparametrization

In [49]:

```
now = datetime.now()
model_dt.fit(X_train, y_train)
model_rf.fit(X_train, y_train)
model_knn.fit(X_train, y_train)
print(f"All the models fitted in: {datetime.now() - now} time")
```

```
All the models fitted in: 0:02:49.008094 time
```

In [50]:

```
resultator.add_results(X_test, y_test, model_dt, "DecisionTree")
resultator.add_results(X_test, y_test, model_rf, "RandomForest")
resultator.add_results(X_test, y_test, model_knn, "KNN")
```

```
The Data from model DecisionTree was acquired and stored.

The Data from model RandomForest was acquired and stored.

The Data from model KNN was acquired and stored.
```

In [51]:

```
resultator.results()
```

Out[51]:

	Model	Accuracy CV	Acc Std CV	Recall CV	Rec Std CV	Precision CV	Precision std CV	Test Acc	Test Recall	Test Precision
0	DecisionTreeBasic	81.7	0.39	27.11	2.22	24.48	1.55	82.1	26.73	25.09
0	DecisionTree	82.17	0.44	27.05	1.95	25.4	1.58	82.59	26.99	26.23
0	RandomForest	87.65	0.25	18.82	1.66	43.53	2.51	87.3	17.15	40.0
0	KNN	87.59	0.15	5.98	1.22	32.89	4.25	87.66	6.18	34.51

We can observe that recall remains too low without any data balancing strategy.

Models with SMOTE Data and Hyperparametrization

In [52]:

```
now = datetime.now()
```

```
now = datetime.now()
model_dt.fit(X_train_smote, y_train_smote)
model_rf.fit(X_train_smote, y_train_smote)
model_knn.fit(X_train_smote, y_train_smote)
print(f"All the models fitted in: {datetime.now() - now} time")
```

```
All the models fitted in: 0:05:22.603360 time
```

In [53]:

```
resultator.add_results(X_test, y_test, model_dt, "DecisionTreeSMOTE")
resultator.add_results(X_test, y_test, model_rf, "RandomForestSMOTE")
resultator.add_results(X_test, y_test, model_knn, "KNNSMOTE")
```

```
The Data from model DecisionTreeSMOTE was acquired and stored.

The Data from model RandomForestSMOTE was acquired and stored.

The Data from model KNNSMOTE was acquired and stored.
```

In [54]:

```
resultator.results()
```

Out[54]:

	Model	Accuracy CV	Acc Std CV	Recall CV	Rec Std CV	Precision CV	Precision std CV	Test Acc	Test Recall	Test Precision
0	DecisionTreeBasic	81.7	0.39	27.11	2.22	24.48	1.55	82.1	26.73	25.09
0	DecisionTree	82.17	0.44	27.05	1.95	25.4	1.58	82.59	26.99	26.23
0	RandomForest	87.65	0.25	18.82	1.66	43.53	2.51	87.3	17.15	40.0
0	KNN	87.59	0.15	5.98	1.22	32.89	4.25	87.66	6.18	34.51
0	DecisionTreeSMOTE	84.83	3.81	89.37	8.24	81.84	1.18	75.21	37.58	20.08
0	RandomForestSMOTE	88.05	4.41	90.58	9.18	86.12	1.24	80.37	40.23	27.13
0	KNNSMOTE	82.79	1.57	94.28	2.92	76.65	0.76	70.04	53.22	20.26

After implementing SMOTE data in our model, we have observed a significant improvement in its recall score. However, this improvement came at the cost of a reduction in accuracy.

Models with ADASYN Data and Hyperparametrization

In [55]:

```
now = datetime.now()
model_dt.fit(X_train_adasyn, y_train_adasyn)
model_rf.fit(X_train_adasyn, y_train_adasyn)
model_knn.fit(X_train_adasyn, y_train_adasyn)
print(f"All the models fitted in: {datetime.now() - now} time")
```

```
All the models fitted in: 0:04:10.131422 time
```

In [56]:

```
resultator.add_results(X_test, y_test, model_dt, "DecisionTreeADASYN")
resultator.add_results(X_test, y_test, model_rf, "RandomForestADASYN")
resultator.add_results(X_test, y_test, model_knn, "KNNADASYN")
```

```
The Data from model DecisionTreeADASYN was acquired and stored.

The Data from model RandomForestADASYN was acquired and stored.

The Data from model KNNADASYN was acquired and stored.
```

In [57]:

```
resultator.results().sort_values("Test Recall", ascending=False)
```

Out[57]:

	Model	Accuracy CV	Acc Std CV	Recall CV	Rec Std CV	Precision CV	Precision std CV	Test Acc	Test Recall	Test Precision
0	KNNADASYN	78.87	0.3	88.65	0.85	74.06	0.4	68.59	56.12	19.98
0	KNNSMOTE	82.79	1.57	94.28	2.92	76.65	0.76	70.04	53.22	20.26
0	RandomForestSMOTE	88.05	4.41	90.58	9.18	86.12	1.24	80.37	40.23	27.13
0	RandomForestADASYN	83.71	2.82	82.46	5.86	84.42	0.97	79.53	40.1	25.83
0	DecisionTreeADASYN	79.57	2.04	79.27	4.59	79.61	0.73	75.76	37.7	20.63
0	DecisionTreeSMOTE	84.83	3.81	89.37	8.24	81.84	1.18	75.21	37.58	20.08
0	DecisionTree	82.17	0.44	27.05	1.95	25.4	1.58	82.59	26.99	26.23
0	DecisionTreeBasic	81.7	0.39	27.11	2.22	24.48	1.55	82.1	26.73	25.09
0	RandomForest	87.65	0.25	18.82	1.66	43.53	2.51	87.3	17.15	40.0
0	KNN	87.59	0.15	5.98	1.22	32.89	4.25	87.66	6.18	34.51

After implementing ADASYN data in our model, we have observed a significant improvement in its recall score. However, this improvement came at the cost of even more reduction in accuracy.

Results Avaliation

Considering the tested models, the one that perfomed better for our goal was the KNN with ADASYN data. This is due it had the best Recall both in Cross Validation and in the Test Data.

In [58]:

```
pd.DataFrame(model_knn.cv_results_).sort_values("rank_test_recall").head(3)
```

Out[58]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_neighbors	param_weights	params	split0_te
1	0.367184	0.017396	0.747471	0.109388	20	distance	{'n_neighbors': 20, 'weights': 'distance'}	(
3	0.453403	0.072362	0.787380	0.069404	30	distance	{'n_neighbors': 30, 'weights': 'distance'}	(
5	0.428398	0.033495	0.865098	0.129896	40	distance	{'n_neighbors': 40, 'weights': 'distance'}	(

In [59]:

```
model_knn.best_params_
```

Out[59]:

```
{'n_neighbors': 20, 'weights': 'distance'}
```

As we can see, the lower the number of neighbors are, the better the model performs.

In [60]:

```
recall_score(y_test, model_dt.predict(X_test))
```

Out[60]:

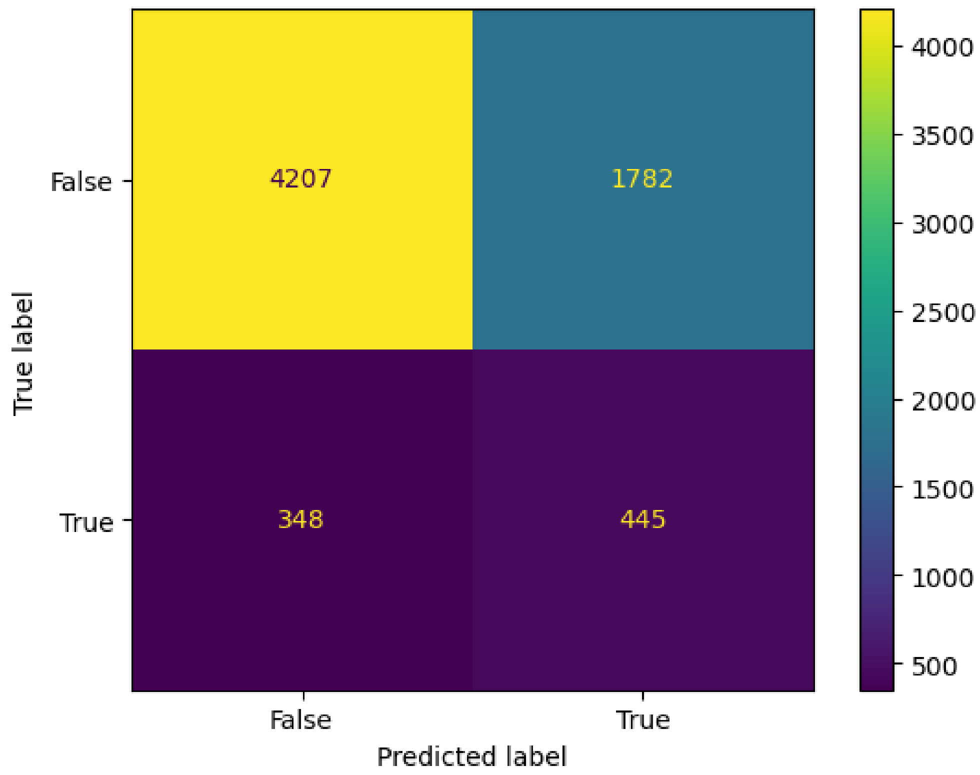
```
0.37704918032786885
```

In [61]:

```
ConfusionMatrixDisplay.from_estimator(model_knn, X_test,y_test)
```

Out[61]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x2052891a130>
```



As expected, since we were optimizing for high recall, we achieved the best performance for positive true cases, at the expense of accuracy due to the higher number of false positives. The confusion matrix provides a clear overview of the model's performance, with high true positive and false positive rates. Overall, the model's performance aligns with our objective of prioritizing the identification of true positive cases at the cost of an increased false positive rate.

In [62]:

```
resultator.dashbordator()
```

Out[62]:

```
<function ipywidgets.widgets.interaction._InteractFactory.__call__.<locals>.<lambda>(*arg
s, **kwargs)>
```

Model Deployment

Model refitting

After identifying the best model and its corresponding hyperparameters, we can train a new model using all available data to optimize its performance in a production setting.

In [63]:

```
X_adasyn, y_adasyn = adasyn.fit_resample(X, y)
```

In [64]:

```
model_knn = GridSearchCV(
    KNeighborsClassifier(weights="distance", n_neighbors= 20), # Define the KNN model
```

```
    {}, # Pass in the hyperparameters to be tuned from the dictionary we defined earlier
    n_jobs=-1, # Use all available CPU cores for parallel computation
    cv=10, # Set the number of folds for cross-validation
    scoring=["recall", "accuracy"], # Set the evaluation metrics to be used for scoring
    refit="recall" # Choose the metric to optimize during randomized search
)

model_knn.fit(X_adasyn, y_adasyn)
```

Out[64]:

```
▸        GridSearchCV
▸ estimator: KNeighborsClassifier
  ▸     KNeighborsClassifier
```

In [65]:

```
pd.DataFrame(model_knn.cv_results_)
```

Out[65]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	params	split0_test_recall	split1_test_recall	split2_test_recall
0	0.471066	0.02683	0.597615	0.06282	{}	0.858003	0.74285	0.710058

Dashboard

Now that we have trained the new model using all available data, we can develop a dashboard that interacts with the model. This dashboard can help the market team improve their results by providing a user-friendly interface for accessing and analyzing the model's output.

In [66]:

```
def make_prediction(age, job, marital, education, default, housing, loan,
                    contact, month,campaign, balance_class,
                    previous_bool):
    data = {
        "age": age,
        "job": job,
        "marital": marital,
        "education": education,
        "default": default,
        "housing": housing,
        "loan": loan,
        "contact": contact,
        "month": month,
        "campaign": campaign,
        "balance_class": balance_class,
        "previous_bool": previous_bool
    }
    df = pd.DataFrame(data, index=[0])
    prediction = model_knn.predict(oe.transform(df))[0]
    if prediction == 0:
        return "Probably will not convert into a client"
    else:
        return "Probably will convert into a client"
```

In [67]:

```
print("Will subscribe for a term deposit?")
s1 = interact(
    make_prediction,
    age=IntText(),

    job=Dropdown(
```

```
        options= df_pos["job"].unique()
    ),
    marital=Dropdown(
        options= df_pos["marital"].unique()
    ),
    education=Dropdown(
        options= df_pos["education"].unique()
    ),
    default=Dropdown(
        options= df_pos["default"].unique()
    ),
    housing=Dropdown(
        options= df_pos["housing"].unique()
    ),
    loan=Dropdown(
        options= df_pos["loan"].unique()
    ),
    contact=Dropdown(
        options= df_pos["contact"].unique()
    ),
    month=Dropdown(
        options= df_pos["month"].unique()
    ),
    campaign=IntText(),
    balance_class=Dropdown(
        options= df_pos["balance_class"].unique()
    ),
    previous_bool=Dropdown(
        options= df_pos["previous_bool"].unique()
    )
);
```

```
Will subscribe for a term deposit?
```