Practical Application III: Comparing Classifiers

Overview: In this practical application, your goal is to compare the performance of the classifiers we encountered in this section, namely K Nearest Neighbor, Logistic Regression, Decision Trees, and Support Vector Machines. We will utilize a dataset related to marketing bank products over the telephone.

Getting Started

Our dataset comes from the UCI Machine Learning repository link (https://archive.ics.uci.edu/ml/datasets/bank+marketing). The data is from a Portugese banking institution and is a collection of the results of multiple marketing campaigns. We will make use of the article accompanying the dataset here (CRISP-DM-BANK.pdf) for more information on the data and features.

Problem 1: Understanding the Data

To gain a better understanding of the data, please read the information provided in the UCI link above, and examine the **Materials and Methods** section of the paper. How many marketing campaigns does this data represent?

In total 17 campaigns were carried out between May 2008 and November 2010, corresponding to a total of 79354 contacts. During these phone campaigns, an attractive long-term deposit application, with good interest rates, was offered.

Problem 2: Read in the Data

Use pandas to read in the dataset `bank-additional-full.csv` and assign to a meaningful variable name.

In [81]:
```python
#pandas for data manipulation
import pandas as pd
#numpy for numerical computations
import numpy as np
#plotly.express and plotly.graph_objects for creating interactive plots and charts
import plotly.express as px
import plotly.graph_objects as go
from collections import defaultdict
import plotly.graph_objects as go
#matplotlib and seaborn for creating static plots and charts
import matplotlib.pyplot as plt
#sklearn for machine learning tasks such as preprocessing, model selection, and evaluation
from sklearn import svm
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split,cross_val_score
import seaborn as sns
import plotly.subplots as sp
#calendar for handling month names
from calendar import month_abbr
from sklearn.neighbors import KNeighborsRegressor,KNeighborsClassifier
from sklearn.impute import KNNImputer
#imblearn for oversampling techniques
from imblearn.over_sampling import SMOTE
# preprocessing
from sklearn.preprocessing import MinMaxScaler
# calculate the MSE score
from sklearn.metrics import mean_squared_error,confusion_matrix
from sklearn.metrics import roc_auc_score,precision_score,recall_score,accuracy_score,f1_score,balanced_accuracy_score
# cross validation
from sklearn.model_selection import GridSearchCV,StratifiedKFold,RandomizedSearchCV
# modeling
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression,RidgeClassifier
# model evaluation
from sklearn.metrics import classification_report, plot_roc_curve,roc_curve, RocCurveDisplay
from sklearn.metrics import classification_report,plot_confusion_matrix, ConfusionMatrixDisplay
from sklearn.dummy import DummyClassifier
from sklearn.inspection import permutation_importance
import random
#warnings for ignoring warnings
import warnings
import time
#The code also uses the "warnings" library to ignore any warnings that may be generated during the execution of the code.
warnings.filterwarnings("ignore")
#plotly.offline for creating offline plots
from plotly.offline import plot, iplot, init_notebook_mode
import plotly.graph_objs as go
init_notebook_mode(connected=True)
```

In [2]:
```python
df = pd.read_csv('data/bank-additional-full.csv', sep = ';')
```

In [3]:
```python
df.head()
```

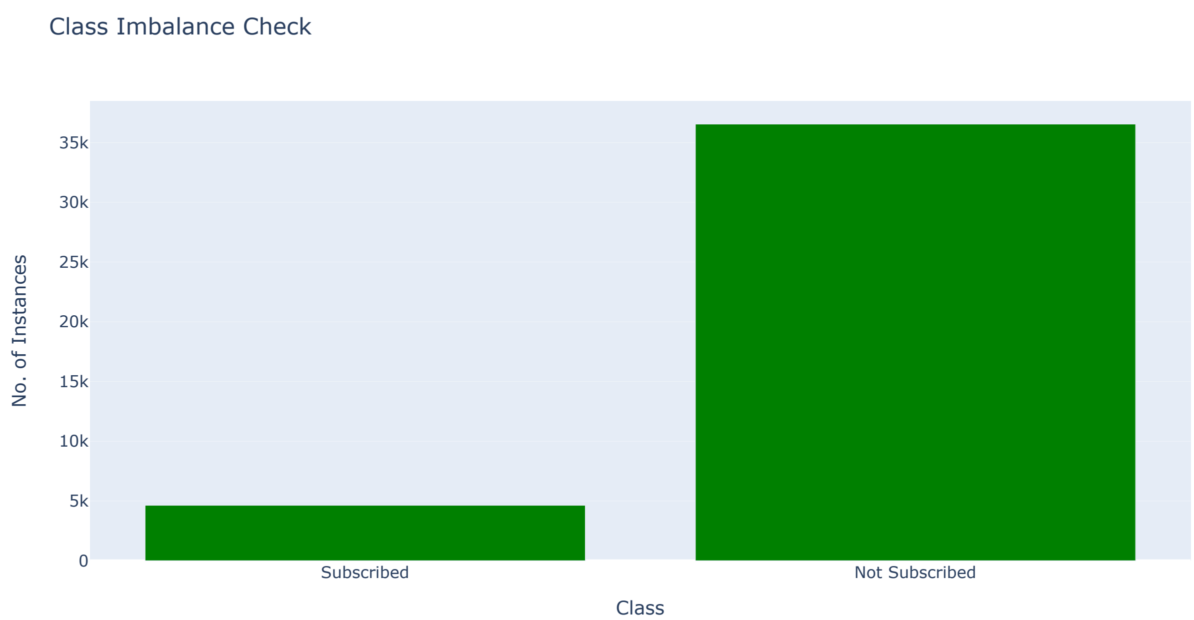
Out[3]:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcome	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	y
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
2	37	services	married	high.school	no	yes	no	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no
4	56	services	married	high.school	no	no	yes	telephone	may	mon	...	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	no

5 rows × 21 columns

Given our data is imbalanced with the majority not subscribing a term deposit, we might want to do re-sampling to adjust the proportion while training.

In [4]:
```
#This code is checking for class imbalance in the data by counting the number of instances for
#each class in the 'y' column of the dataframe 'df'.

#creates a variable 'y' that stores the values of the 'y' column of the dataframe 'df' as a list.
y = list(df['y'])
#creates a dictionary 'instances_per_class' with keys 'yes' and 'no' and values equal to the number of instances
#of the corresponding class in the 'y' column.
instances_per_class = {'yes' : y.count('yes'), 'no' : y.count('no')}
#creates a bar chart using the plotly library, with the x-axis representing the classes ('yes' and 'no') and
#the y-axis representing the number of instances of each class.
fig = go.Figure(data=[go.Bar(x=list(instances_per_class.keys()), y=list(instances_per_class.values()),
                      marker=dict(color='green'))])

fig.update_layout(title='Class Imbalance Check', xaxis_title='Class', yaxis_title='No. of Instances',
                  xaxis=dict(tickmode='array', tickvals=[0,1], ticktext=['Subscribed', 'Not Subscribed']))
fig.show()

print(f"Class Labels : {list(instances_per_class.keys())}\nNo. of Inst. : {list(instances_per_class.values())}\n\nTotal number of features : {len(df.columns)-1}")
```




```
Class Labels : ['yes', 'no']
No. of Inst. : [4640, 36548]

Total number of features : 20
```

Problem 3: Understanding the Features

Examine the data description below, and determine if any of the features are missing values or need to be coerced to a different data type.

```
Input variables:
# bank client data:
1 - age (numeric)
2 - job : type of job (categorical: 'admin.','blue-collar','entrepreneur','housemaid','management','retired','self-employed','services','student','technician','unemplo
yed','unknown')
3 - marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)
4 - education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown')
5 - default: has credit in default? (categorical: 'no','yes','unknown')
6 - housing: has housing loan? (categorical: 'no','yes','unknown')
7 - loan: has personal loan? (categorical: 'no','yes','unknown')
# related with the last contact of the current campaign:
8 - contact: contact communication type (categorical: 'cellular','telephone')
9 - month: last contact month of year (categorical: 'jan', 'feb', 'mar', ..., 'nov', 'dec')
10 - day_of_week: last contact day of the week (categorical: 'mon','tue','wed','thu','fri')
11 - duration: last contact duration, in seconds (numeric). Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the
duration is not known before a call is performed. Also, after the end of the call y is obviously known. Thus, this input should only be included for benchmark purposes
and should be discarded if the intention is to have a realistic predictive model.
# other attributes:
12 - campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
13 - pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)
14 - previous: number of contacts performed before this campaign and for this client (numeric)
15 - poutcome: outcome of the previous marketing campaign (categorical: 'failure','nonexistent','success')
# social and economic context attributes
16 - emp.var.rate: employment variation rate - quarterly indicator (numeric)
17 - cons.price.idx: consumer price index - monthly indicator (numeric)
18 - cons.conf.idx: consumer confidence index - monthly indicator (numeric)
19 - euribor3m: euribor 3 month rate - daily indicator (numeric)
20 - nr.employed: number of employees - quarterly indicator (numeric)

Output variable (desired target):
21 - y - has the client subscribed a term deposit? (binary: 'yes','no')
```

3.1) Describe Data

The describe() function in a pandas DataFrame is used to generate descriptive statistics of the data. It returns a summary of the central tendency, dispersion, and shape of the distribution, excluding missing values.

In [5]:
```
df.describe()
```

Out[5]:

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
count	41188.00000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000
mean	40.02406	258.285010	2.567593	962.475454	0.172963	0.081886	93.575664	-40.502600	3.621291	5167.035911
std	10.42125	259.279249	2.770014	186.910907	0.494901	1.570960	0.578840	4.628198	1.734447	72.251528
min	17.00000	0.000000	1.000000	0.000000	0.000000	-3.400000	92.201000	-50.800000	0.634000	4963.600000
25%	32.00000	102.000000	1.000000	999.000000	0.000000	-1.800000	93.075000	-42.700000	1.344000	5099.100000
50%	38.00000	180.000000	2.000000	999.000000	0.000000	1.100000	93.749000	-41.800000	4.857000	5191.000000
75%	47.00000	319.000000	3.000000	999.000000	0.000000	1.400000	93.994000	-36.400000	4.961000	5228.100000
max	98.00000	4918.000000	56.000000	999.000000	7.000000	1.400000	94.767000	-26.900000	5.045000	5228.100000

3.2) Data Types

The dtypes attribute in a pandas DataFrame returns the data types of each column in the DataFrame. The attribute returns a Series object where the index is the column name and the value is the data type.

In [6]:
```
df.dtypes
```

Out[6]:
```
age                 int64
job                object
marital            object
education          object
default            object
housing            object
loan               object
contact            object
month              object
day_of_week        object
duration            int64
campaign            int64
pdays               int64
previous            int64
poutcome           object
emp.var.rate      float64
cons.price.idx    float64
cons.conf.idx     float64
euribor3m         float64
nr.employed       float64
y                  object
dtype: object
```

3.3) Data Dimensions

The shape attribute in a pandas DataFrame returns the number of rows and columns in the DataFrame. The attribute returns a tuple, where the first element is the number of rows and the second element is the number of columns.

In [7]:
```
df.shape
```

Out[7]:
```
(41188, 21)
```

3.4) Data Collection - Check NA

There are several ways to check for missing values (also known as "null" values) in a pandas DataFrame.

One way is to use the isnull() function, which returns a DataFrame of the same shape as the original, but with True for missing values and False for non-missing values. We will then use the sum() function to count the number of missing values in each column:

In [8]:
```
df.isnull().sum()
```

Out[8]:
```
age               0
job               0
marital           0
education         0
default           0
housing           0
loan              0
contact           0
month             0
day_of_week       0
duration          0
campaign          0
pdays             0
previous          0
poutcome          0
emp.var.rate      0
cons.price.idx    0
cons.conf.idx     0
euribor3m         0
nr.employed       0
y                 0
dtype: int64
```

3.5) Numerical and Categorical Attributes

The select_dtypes() method is used to select columns in a DataFrame based on their data types. It takes as input the data types to be selected, and returns a new DataFrame containing only the columns that match the specified data types. We will use this to look at the list of numerical and categorical attributes.

In [9]:
```
num_attributes = df.select_dtypes(include=['int64', 'float64'] )
cat_attributes = df.select_dtypes(exclude=['int64', 'float64','datetime64[ns]'] )
num_attributes.sample()
```

Out[9]:

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
27264	37	196	1	999	0	-0.1	93.2	-42.0	4.021	5195.8

In [10]:
```
cat_attributes.sample()
```

Out[10]:

	job	marital	education	default	housing	loan	contact	month	day_of_week	poutcome	y
12642	services	married	high.school	unknown	yes	no	cellular	jul	mon	nonexistent	no

3.6) Clean Dataset by dropping columns

In [11]:
```
#Lets first map the no and yes values in the outcome to 0 and 1
mapping_yn = {'no': 0, 'yes': 1}
df['y'] = df['y'].map(mapping_yn)
df = df.drop(['emp.var.rate','cons.price.idx', 'cons.conf.idx', 'euribor3m', 'nr.employed'], axis = 1)
```

In [12]:
```
# We will makea copy of the dataframe to avoid running the notebook from the beginning.
df_raw = df.copy()
```

In [13]:

```python
# Define X and y variables
X = df_raw.drop('y', axis=1)
y = df_raw['y']

unique_val_count = {}

#Remove numberical variables since we will be exploring only the categorical variables in the beginning.
cat = list(X.columns)
cat.remove('age')
cat.remove('duration')
cat.remove('pdays')
cat.remove('campaign')
cat.remove('previous')

#Get the unique values in each of the categorical columns
for col in X.columns:
    if col not in cat:
        continue
    unique_values = np.unique(X[col])
    temp = defaultdict(int)
    for val in X[col]:
        temp[val] += 1
    unique_val_count[col] = temp

print(f"Number of Categorical columns : {len(unique_val_count.keys())}")

#Map those unique values to numbers.
for col, attr in unique_val_count.items():
    map_val = {}
    count = 0
    for key in attr.keys():
        map_val[key] = count
        count += 1
    X[col] = X[col].map(map_val)

X.head(10)
```

```
Number of Categorical columns : 10
```

Out[13]:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome
0	56	0	0	0	0	0	0	0	0	0	261	1	999	0	0
1	57	1	0	1	1	0	0	0	0	0	149	1	999	0	0
2	37	1	0	1	0	1	0	0	0	0	226	1	999	0	0
3	40	2	0	2	0	0	0	0	0	0	151	1	999	0	0
4	56	1	0	1	0	0	1	0	0	0	307	1	999	0	0
5	45	1	0	3	1	0	0	0	0	0	198	1	999	0	0
6	59	2	0	4	0	0	0	0	0	0	139	1	999	0	0
7	41	3	0	5	1	0	0	0	0	0	217	1	999	0	0
8	24	4	1	4	0	1	0	0	0	0	380	1	999	0	0
9	25	1	1	1	0	1	0	0	0	0	50	1	999	0	0

3.7) Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a critical step in the machine learning process that involves analyzing and understanding the data before building a model. The goal of EDA is to uncover patterns, relationships, and insights in the data that can inform the model building process.

EDA is an iterative process and it's essential to keep in mind the business objective and the problem statement.

In [14]:
```python
#Get the unique values for each column. We will use it for plotting purposes.
cross_tables = []
for i in unique_val_count.keys():
    cross_tables.append(pd.crosstab(X[i], y))
```

In [15]:
```python
#start by mapping the outcome values.
mapping_yn = {0:'no',1:'yes'}
for df1 in cross_tables:
    df1.rename(columns=mapping_yn, inplace=True)
```

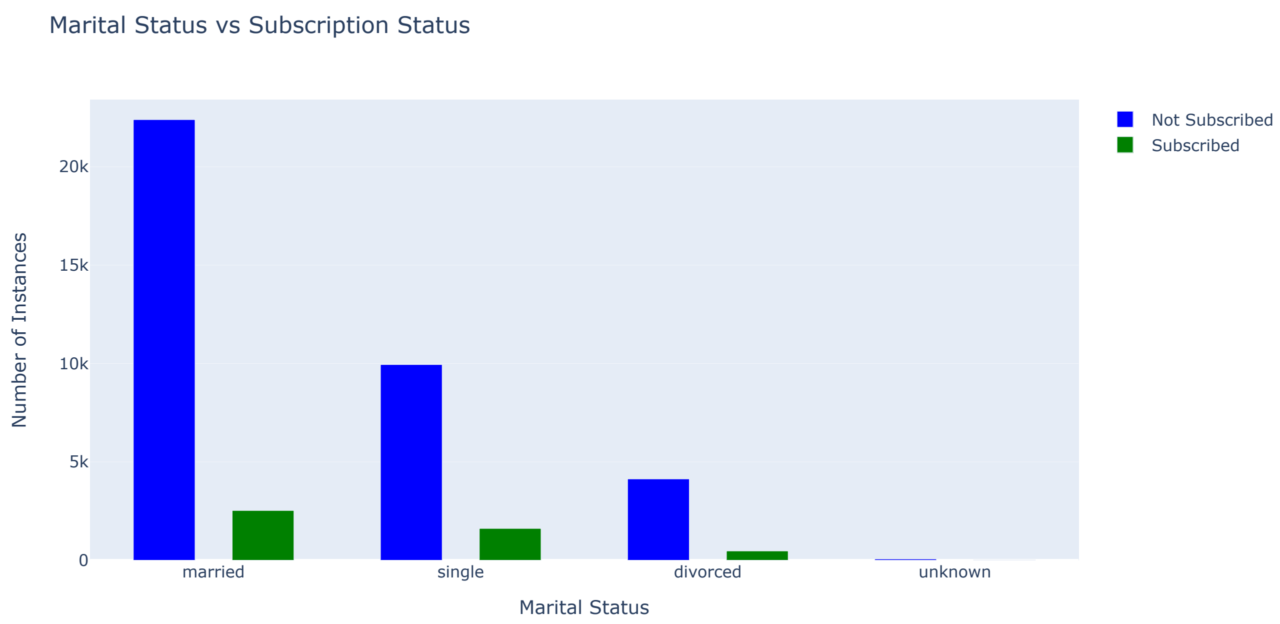
In [16]:
```python
unique_val_count
```

Out[16]:
```
{'job': defaultdict(int,
             {'housemaid': 1060,
              'services': 3969,
              'admin.': 10422,
              'blue-collar': 9254,
              'technician': 6743,
              'retired': 1720,
              'management': 2924,
              'unemployed': 1014,
              'self-employed': 1421,
              'unknown': 330,
              'entrepreneur': 1456,
              'student': 875}),
 'marital': defaultdict(int,
             {'married': 24928,
              'single': 11568,
              'divorced': 4612,
              'unknown': 80}),
 'education': defaultdict(int,
             {'basic.4y': 4176,
              'high.school': 9515,
              'basic.6y': 2292,
              'basic.9y': 6045,
              'professional.course': 5243,
              'unknown': 1731,
              'university.degree': 12168,
              'illiterate': 18}),
 'default': defaultdict(int, {'no': 32588, 'unknown': 8597, 'yes': 3}),
 'housing': defaultdict(int, {'no': 18622, 'yes': 21576, 'unknown': 990}),
 'loan': defaultdict(int, {'no': 33950, 'yes': 6248, 'unknown': 990}),
 'contact': defaultdict(int, {'telephone': 15044, 'cellular': 26144}),
 'month': defaultdict(int,
             {'may': 13769,
              'jun': 5318,
              'jul': 7174,
              'aug': 6178,
              'oct': 718,
              'nov': 4101,
              'dec': 182,
              'mar': 546,
              'apr': 2632,
              'sep': 570}),
 'day_of_week': defaultdict(int,
             {'mon': 8514,
              'tue': 8090,
              'wed': 8134,
              'thu': 8623,
              'fri': 7827}),
 'poutcome': defaultdict(int,
             {'nonexistent': 35563, 'failure': 4252, 'success': 1373})}
```

1) There're some binary variables such as 'default', 'housing', 'loan'. We might want to transform it for better predicting.
2) Although no NULL is detected, there are many 'unknown' values, which we should deal with when preprocessing.
3) There are a large percentage of unknown previous outcomes, which is not surprising because many customers don't have previous contacts (previous=0).

3.7.1) Comparing Marital Status with Subscription Status

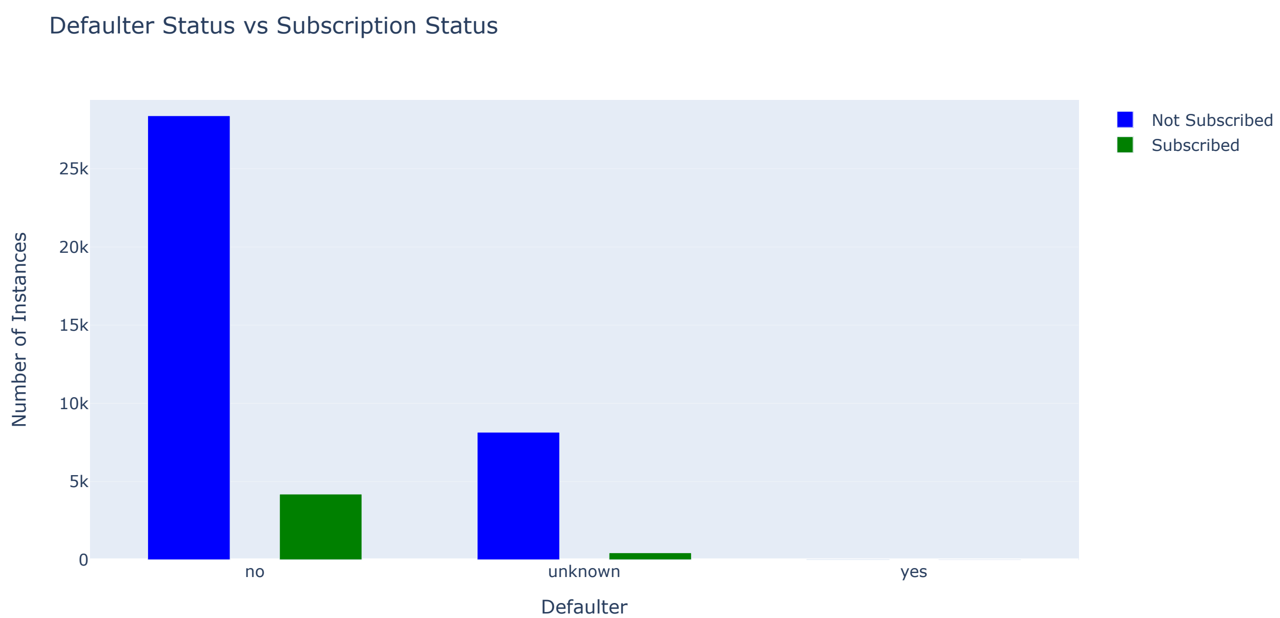
In [17]:
```
#We will be using the similar types of visualizations for all the categorical variables.

#This code is creating a stacked bar chart to compare the distribution of a categorical variable "marital"
#with respect to the target variable "subscription status".

width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['marital'].keys()), y=cross_tables[1]['no'],marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['marital'].keys()), y=cross_tables[1]['yes'],marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Marital Status vs Subscription Status', xaxis_title='Marital Status', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[1]['no'])) + list(cross_tables[1]['yes']))+1000]))
fig.show()
```



Contacts who are married are subscribed more compared to those that are single and divorced. The total number of those unsubscribed far exceeds those that are subscribed.

3.7.2) Comparing Defaulter with Subscription Status

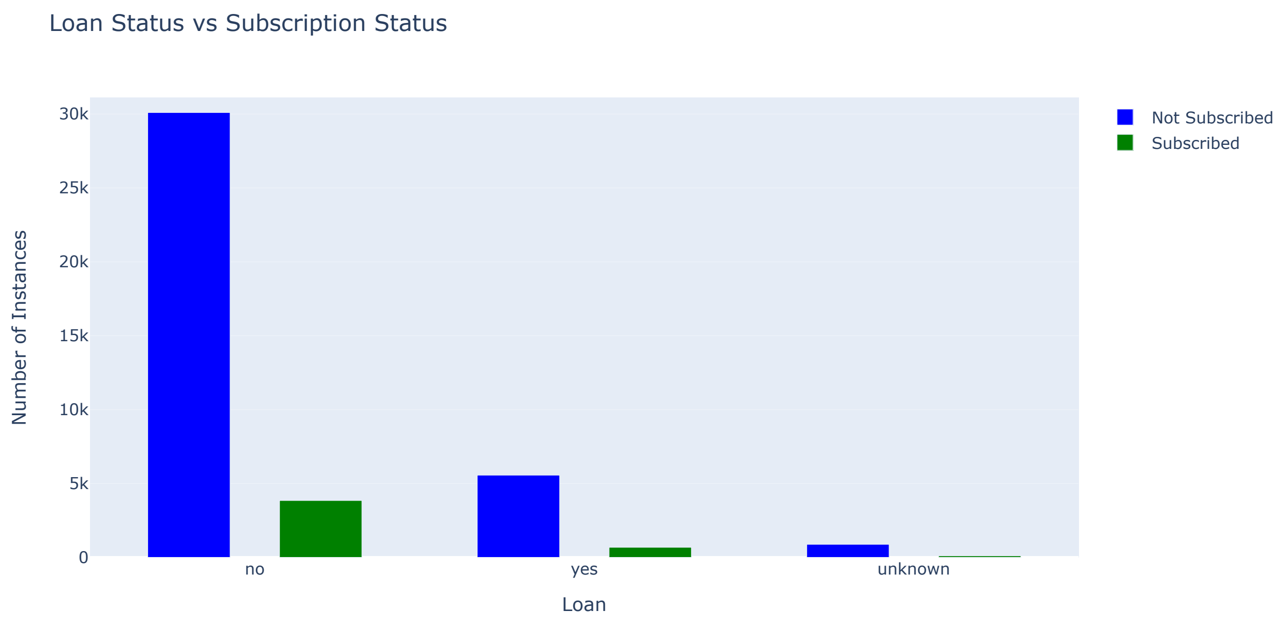
In [18]:
```
#Visualation of a categorical variable
#This code is creating a stacked bar chart to compare the distribution of a categorical variable
#"default" with respect to the target variable "subscription status".
width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['default'].keys()), y=cross_tables[3]['no'],
                    marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['default'].keys()), y=cross_tables[3]['yes'],
                    marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Defaulter Status vs Subscription Status', xaxis_title='Defaulter', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[3]['no'])) + list(cross_tables[3]['yes']))+1000]))
fig.show()
```



From the above, we see The number of defaulters is almost none compared to those unknown or with value yes. Those who have not defaulted, have subscribed more compared to those who have defaulted or to those who have no information.

3.7.3) Comparing Loan status with Subscription Status

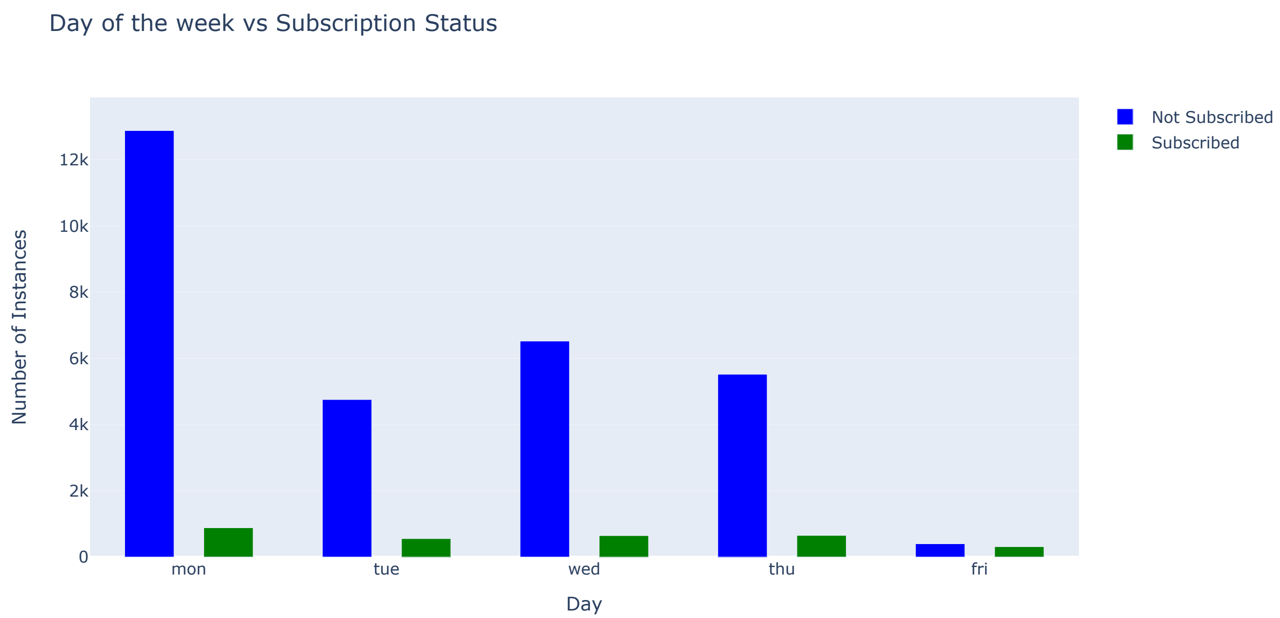
In [19]:
```
#Visualation of a categorical variable

#This code is creating a stacked bar chart to compare the distribution of a categorical variable "loan"
#with respect to the target variable "subscription status".
width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['loan'].keys()), y=cross_tables[5]['no'],
                    marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['loan'].keys()), y=cross_tables[5]['yes'],
                    marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Loan Status vs Subscription Status', xaxis_title='Loan', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[5]['no'])) + list(cross_tables[5]['yes']))+1000]))
fig.show()
```



From the above, we see that those without any loans are subscribed more compared to those who have loans.

3.7.4) Comparing Day of the week with Subscription Status

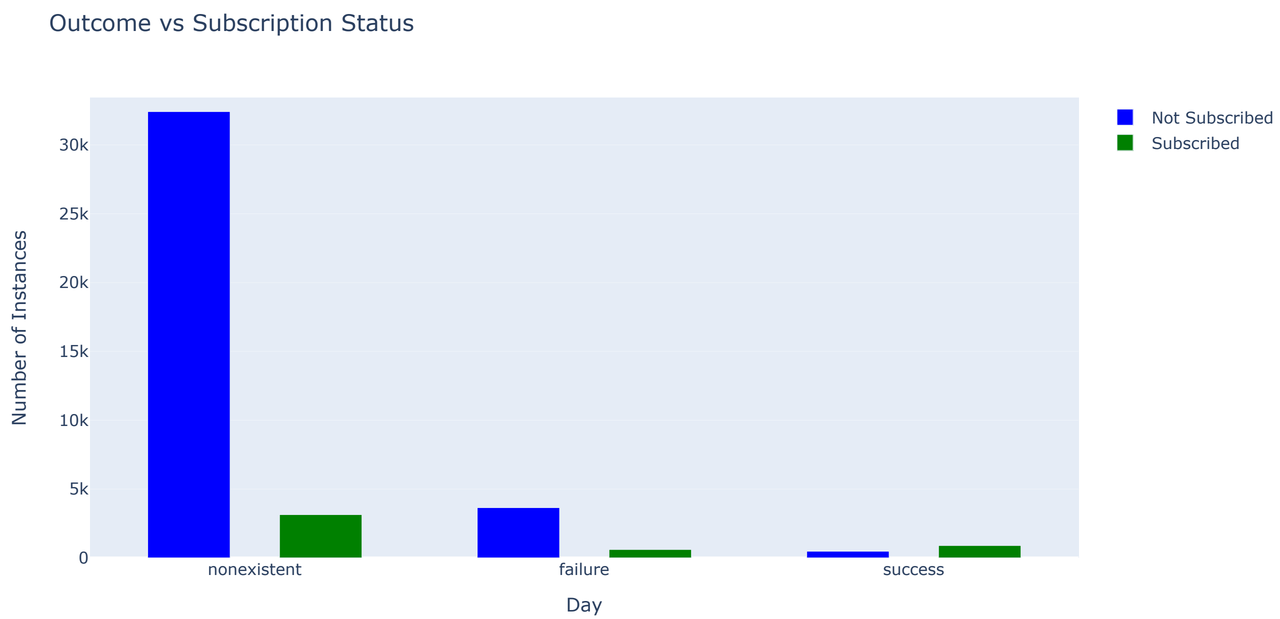
```
In [20]: #Visualation of a categorical variable

#This code is creating a stacked bar chart to compare the distribution of a categorical variable "day_of_week"
#with respect to the target variable "subscription status".

width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['day_of_week'].keys()), y=cross_tables[7]['no'],
                    marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['day_of_week'].keys()), y=cross_tables[7]['yes'],
                    marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Day of the week vs Subscription Status', xaxis_title='Day', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[7]['no'])) + list(cross_tables[7]['yes']))+1000]))
fig.show()
```



Monday is the day where most people subscribed and unsubscribed compared to the other days of the week.

3.7.5) Comparing Previous Campaign with Subscription Status

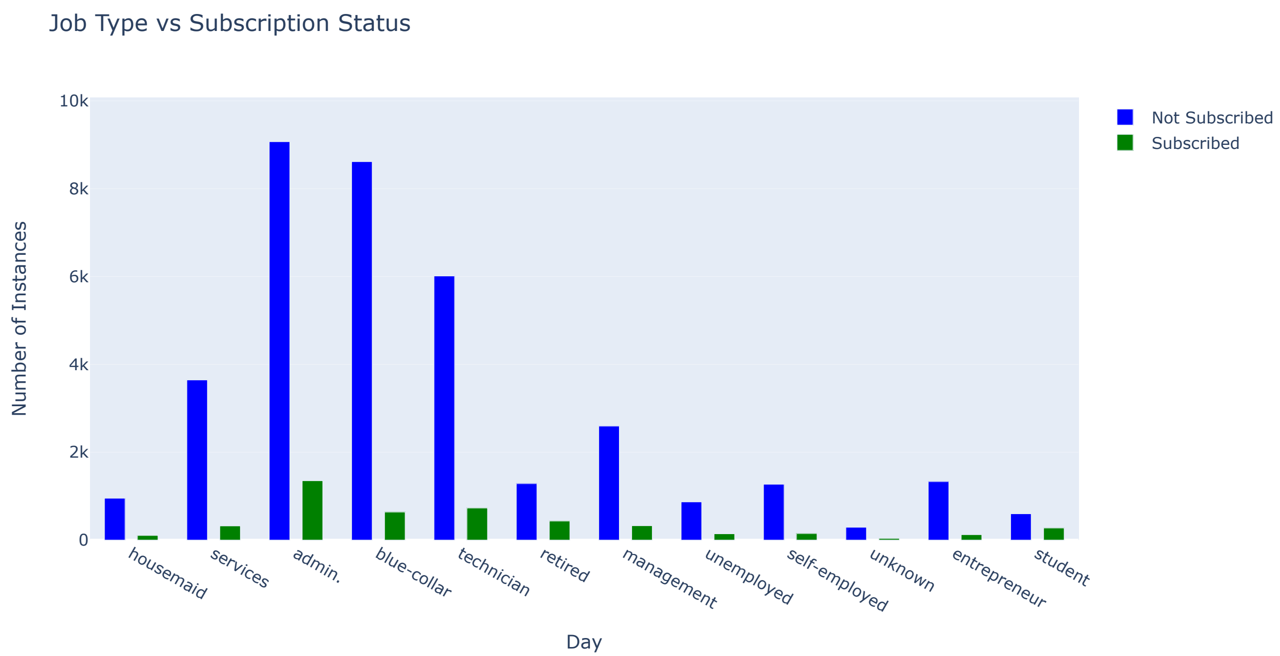
In [21]:
```
#Visualation of a categorical variable

#This code is creating a stacked bar chart to compare the distribution of a categorical variable
#"poutcome" with respect to the target variable "subscription status".
width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['poutcome'].keys()), y=cross_tables[9]['no'],
                    marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['poutcome'].keys()), y=cross_tables[9]['yes'],
                    marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Outcome vs Subscription Status', xaxis_title='Day', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[9]['no'])) + list(cross_tables[9]['yes']))+1000]))
fig.show()
```



From the above, we can see that those with non-existent outcomes have more subscriptions compared to those that are failure or success.

3.7.6) Comparing Job type with Subscription Status

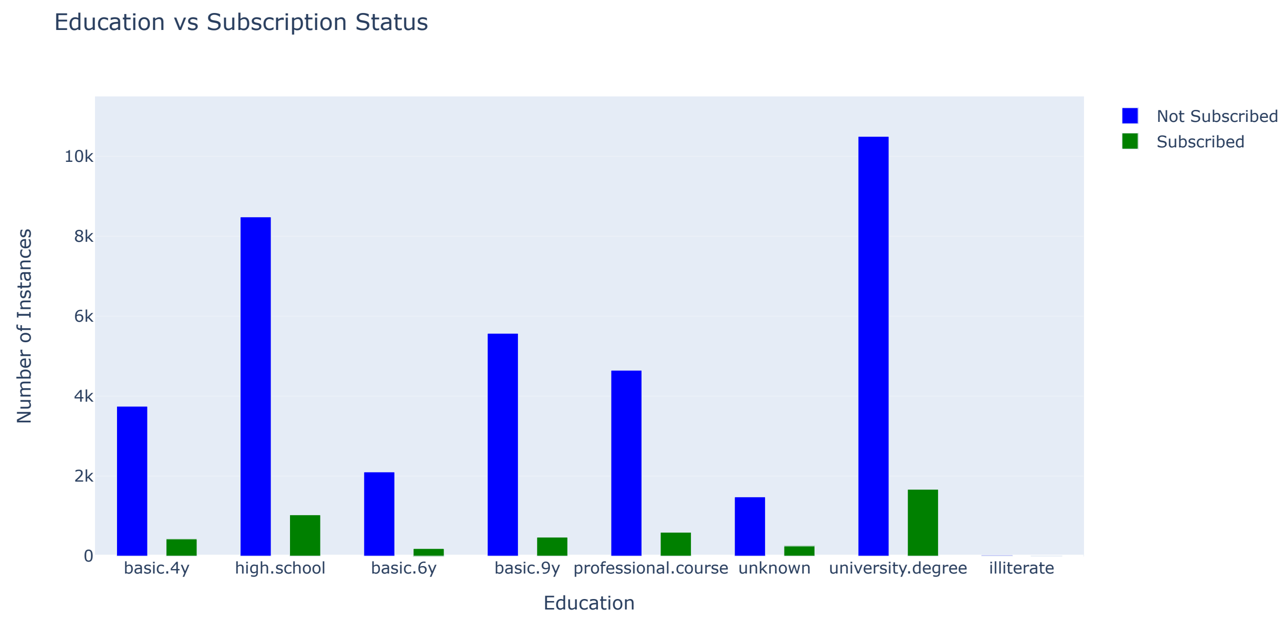
In [22]:
```
#Visualation of a categorical variable

#This code is creating a stacked bar chart to compare the distribution of a categorical variable
#"job type" with respect to the target variable "subscription status".
width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['job'].keys()), y=cross_tables[0]['no'],
                    marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['job'].keys()), y=cross_tables[0]['yes'],
                    marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Job Type vs Subscription Status', xaxis_title='Day', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[0]['no'])) + list(cross_tables[0]['yes']))+1000]))
fig.show()
```



Those with admin, blue-collar and technician job types subscribe more compared to other job types.

3.7.7) Comparing Education with Subscription Status

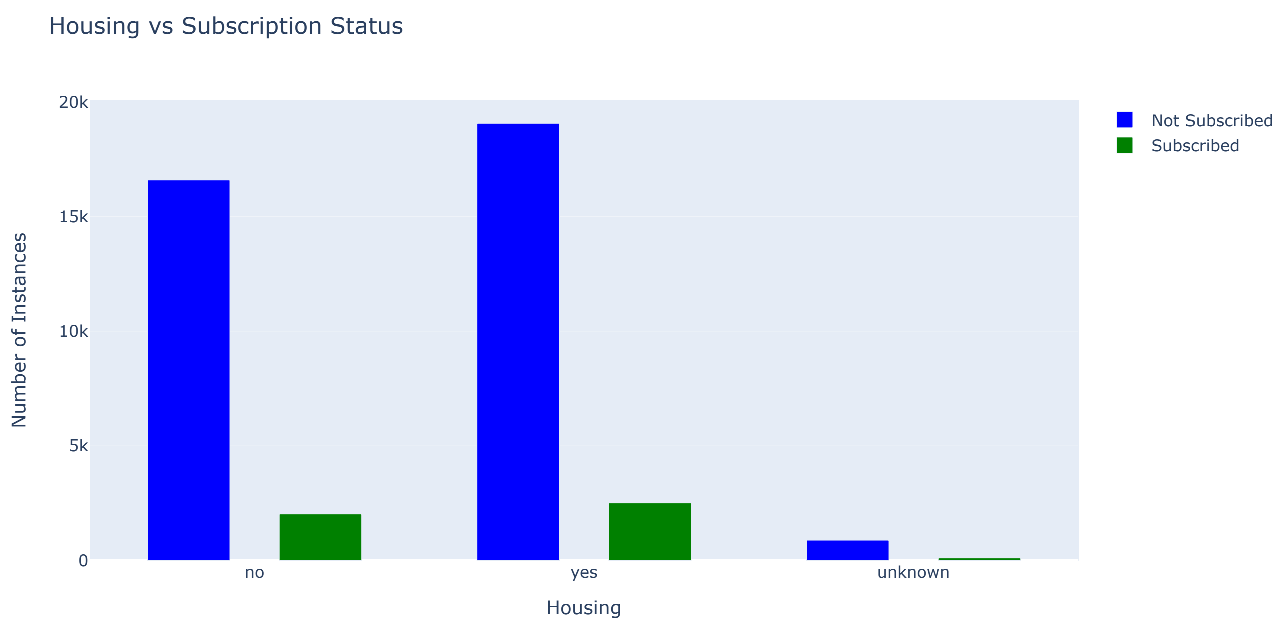
In [23]:
```
#Visualation of a categorical variable

#This code is creating a stacked bar chart to compare the distribution of a categorical variable
#"education" with respect to the target variable "subscription status".
width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['education'].keys()), y=cross_tables[2]['no'],marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['education'].keys()), y=cross_tables[2]['yes'],marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Education vs Subscription Status', xaxis_title='Education', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[2]['no'])) + list(cross_tables[2]['yes']))+1000]))
fig.show()
```



From the above, we can see that contacts with high school and university degree subscribe more compared to other education levels.

3.7.8) Comparing Housing Status with Subscription Status

```
In [24]: #Visualation of a categorical variable


#This code is creating a stacked bar chart to compare the distribution of a categorical variable
#"housing" with respect to the target variable "subscription status".
width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['housing'].keys()), y=cross_tables[4]['no'],marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['housing'].keys()), y=cross_tables[4]['yes'],marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Housing vs Subscription Status', xaxis_title='Housing', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[4]['no'])) + list(cross_tables[4]['yes']))+1000]))
fig.show()
```



As you can see from the above, we can see that there is not much difference between those with housing and those without housing. However, those with housing seem to have subscribed more compared to those without housing.

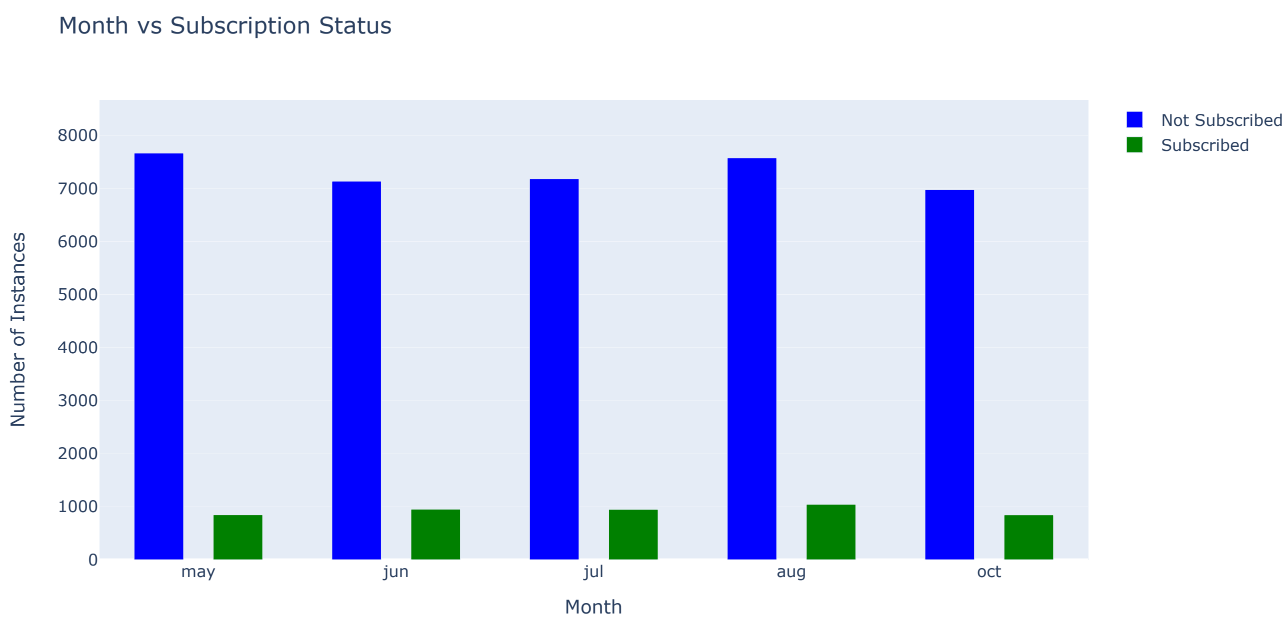
3.7.9) Comparing Contact Type with Subscription Status

In [25]:
```
#Visualation of a categorical variable

#
#This code is creating a stacked bar chart to compare the distribution of a categorical variable
#"contact" with respect to the target variable "subscription status".
width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['contact'].keys()), y=cross_tables[6]['no'],marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['contact'].keys()), y=cross_tables[6]['yes'],marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Contact Type vs Subscription Status', xaxis_title='Contact', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[6]['no'])) + list(cross_tables[6]['yes']))+1000]))
fig.show()
```

From the above, we see that those with cellular contact type subscribe more compared to those with telephone contact type.

3.7.10) Comparing Month with Subscription Status

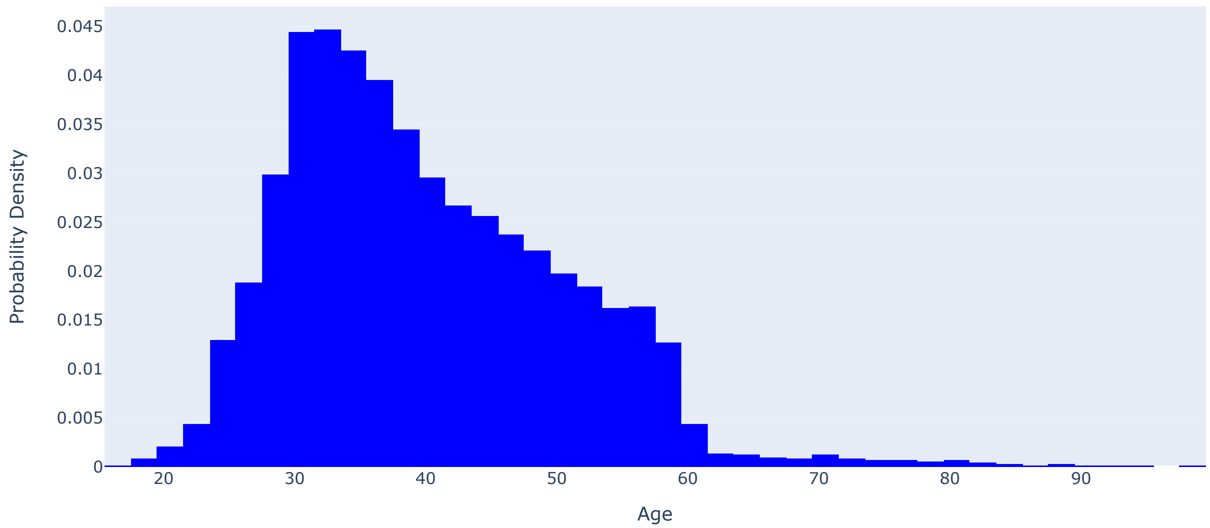
In [26]:
```
#Visualation of a categorical variable


#This code is creating a stacked bar chart to compare the distribution of a categorical variable
#"day of the week" with respect to the target variable "subscription status".
width = 0.25
fig = go.Figure()
fig.add_trace(go.Bar(x=list(unique_val_count['month'].keys()), y=cross_tables[8]['no'],marker=dict(color='blue'), width=width, name='Not Subscribed'))
fig.add_trace(go.Bar(x=list(unique_val_count['month'].keys()), y=cross_tables[8]['yes'],marker=dict(color='green'), width=width, name='Subscribed'))
fig.update_layout(title='Month vs Subscription Status', xaxis_title='Month', yaxis_title='Number of Instances',
                  yaxis=dict(range=[0, max((list(cross_tables[8]['no'])) + list(cross_tables[8]['yes']))+1000]))
fig.show()
```



From the above, we can see that August month had more number of subscriptions compared to other months.

3.7.11) Subscriptions by Age

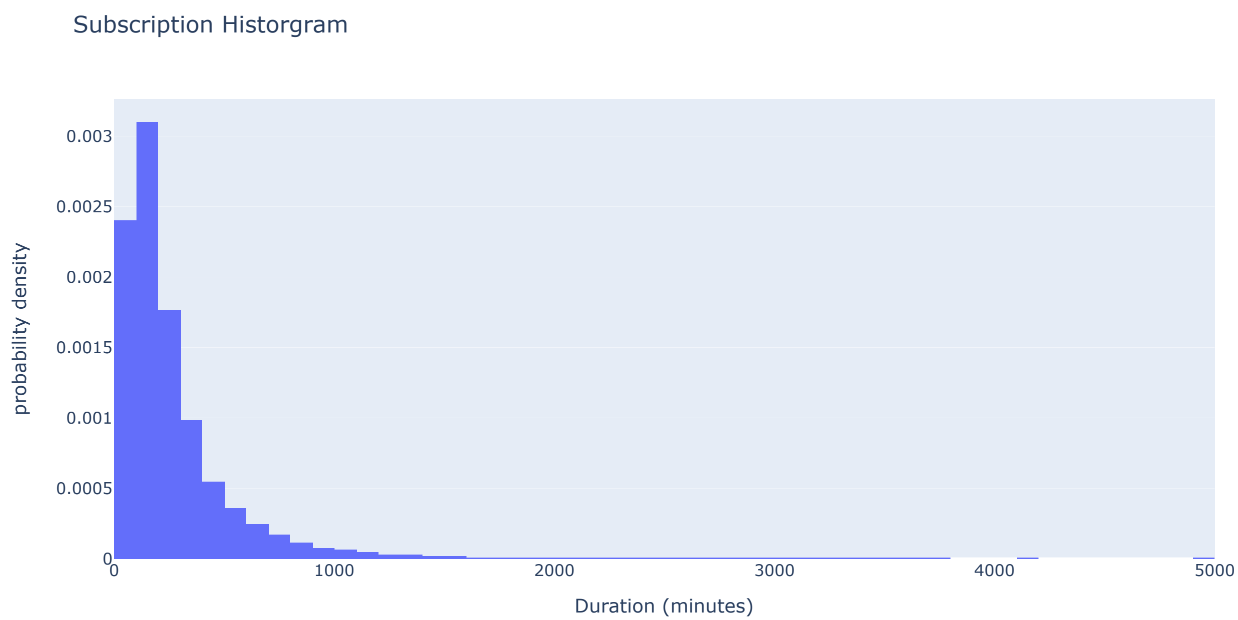
In [27]:
```
#Visualation of a numerica variable


#This code creates a histogram to visualize the distribution of the variable 'age' in the dataframe 'df'.
fig = go.Figure()
fig.add_trace(go.Histogram(x=df['age'],
                           nbinsx=len(df['age'].unique()),
                           histnorm='probability density',
                           name='Age Distribution',
                           marker_color='blue'))

fig.update_layout(xaxis_title='Age',
                  yaxis_title='Probability Density')
```



From the above, we can see that majority of contacts are <= 60 years. Among those with age <= 60, the younger the clients are most likely to subscribe.

3.7.12) Subscriptions by Duration

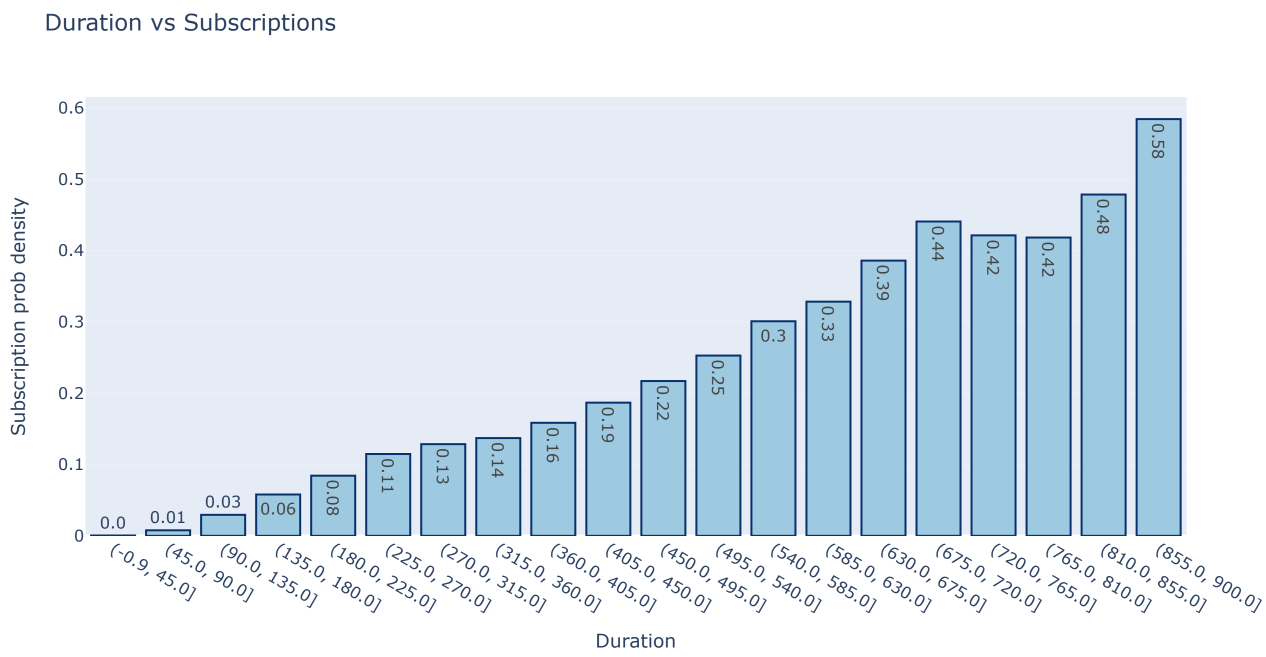
In [28]:
```
#Visualation of a numerica variable

#The second line adds a trace to the figure, using the histogram function of the plotly library.
#It sets the x-axis to the "age" column of the dataframe, the number of bins (nbinsx) to the number of unique values
#in the "age" column, the histogram normalization to "probability density",
#the name of the trace to "Age Distribution", and the color of the markers to blue.


px.histogram(df, x='duration', nbins=50, title='Subscription Historgram',
             labels={'duration':'Duration (minutes)'},
             histnorm='probability density',
             )
```



As we can see from the above plot, the graph contains many outliers. We will try to zoom into those with less than 1200.

In [29]:
```
df_duration_ranged = df[df['duration'] /60 <= 15]
```

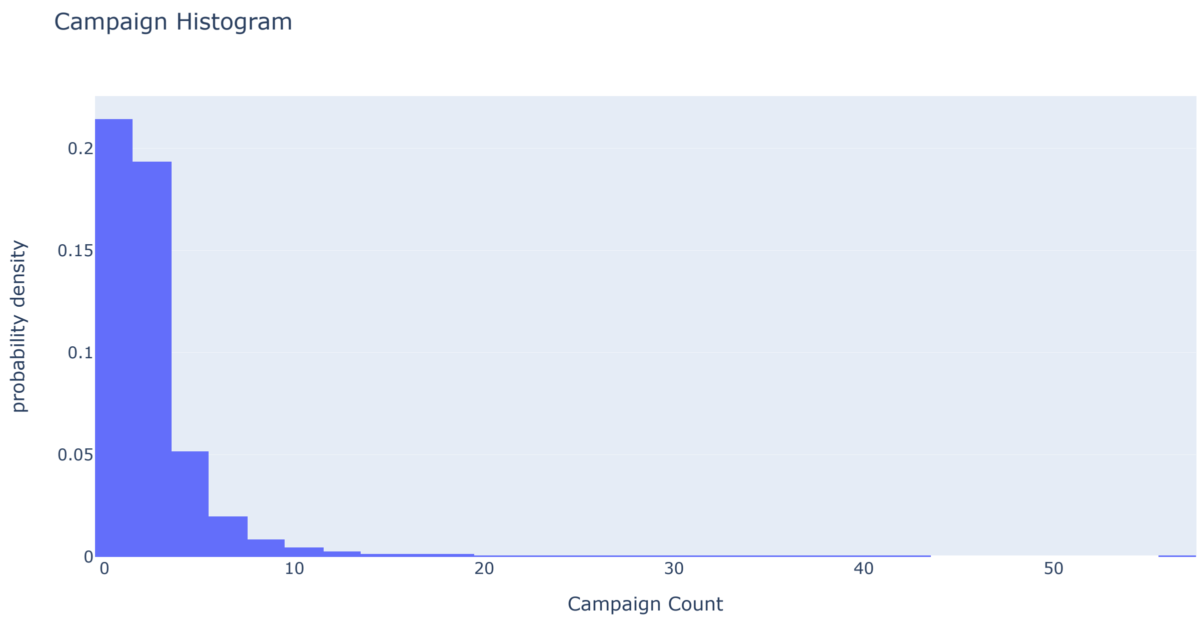
```
In [30]: #Visualization of numerical variable

#This code creates a bar chart to compare the distribution of the "duration" variable with the target
#variable "y" (subscription status) using the plotly library.

data = [go.Bar(
    x=df_duration_ranged.groupby(pd.cut(df_duration_ranged['duration'], bins=20)).mean().index.astype(str),
    y=df_duration_ranged.groupby(pd.cut(df_duration_ranged['duration'], bins=20)).mean()['y'],
    text=df_duration_ranged.groupby(pd.cut(df_duration_ranged['duration'], bins=20)).mean()['y'].round(2).astype(str),
    textposition='auto',
    marker=dict(color='rgb(158,202,225)', line=dict(color='rgb(8,48,107)',width=1.5))
)]

layout = go.Layout(
    title='Duration vs Subscriptions',
    xaxis=dict(title='Duration'),
    yaxis=dict(title='Subscription prob density')
)

fig = go.Figure(data=data, layout=layout)
fig.show()
```



From the above, Based on campaigns, clients being contacted more are less likely to subscribe.

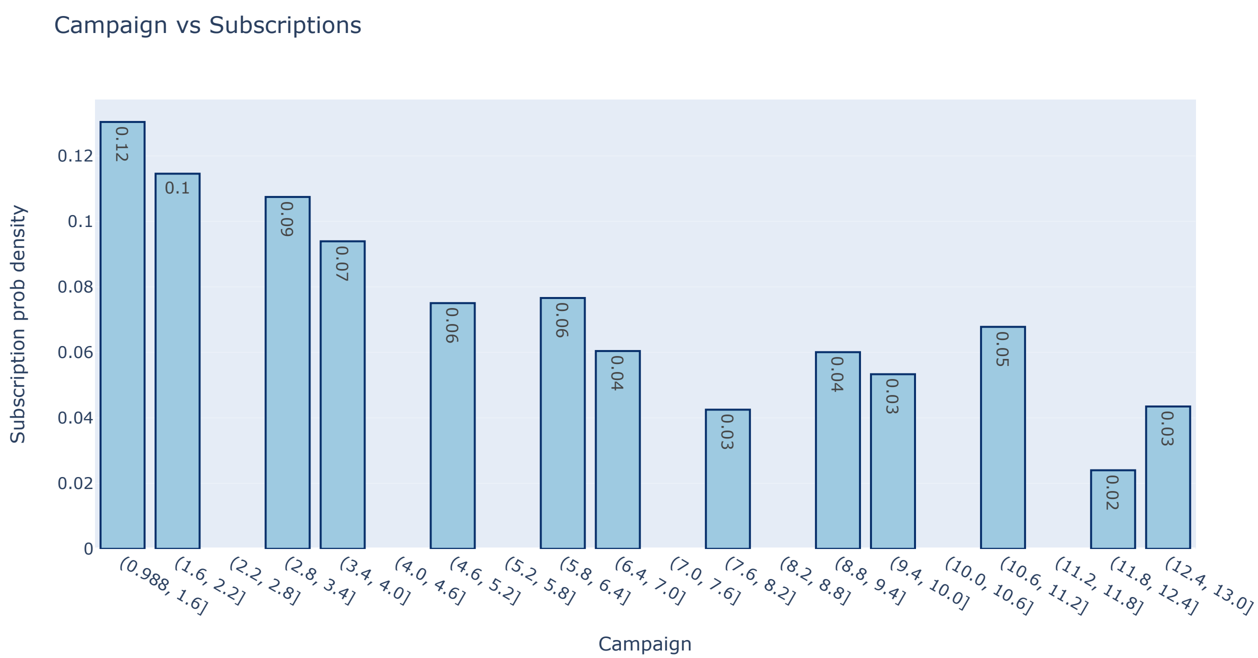
3.7.13) Subscriptions by Campaign

In [31]:
```
#Visualation of a numerica variable

#This code creates a histogram of the "campaign" variable of the dataframe "df" using the plotly express library.
px.histogram(df, x='campaign', nbins=50, title='Campaign Histogram',
             labels={'campaign':'Campaign Count'},
             histnorm='probability density',
            )
```



As 'campaign' > 13 don't have enough samples, we just take those <= 13 for deep diving

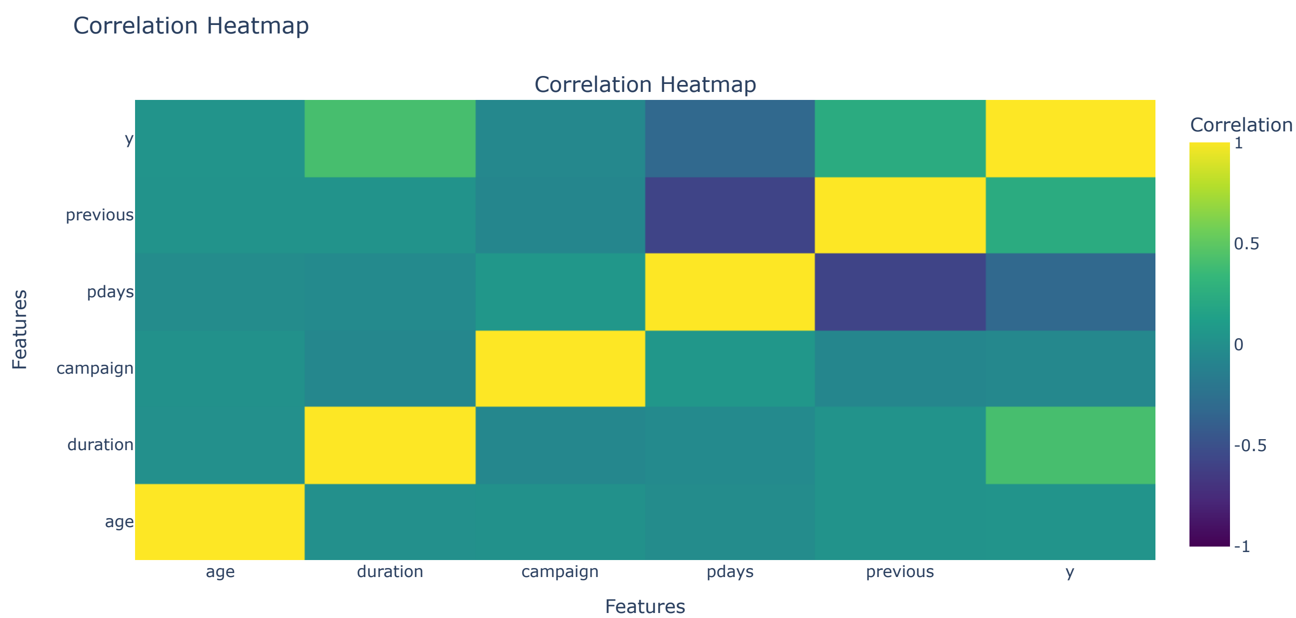
```
In [32]: #Visualization of numerical variable

#This code creates a bar chart to compare the distribution of the "campaign" variable with the target variable "y" (subscription status) using the plotly library.
df_campaign = df[df['campaign'] <= 13]
data = [go.Bar(
    x=df_campaign.groupby(pd.cut(df_campaign['campaign'], bins=20)).mean().index.astype(str),
    y=df_campaign.groupby(pd.cut(df_campaign['campaign'], bins=20)).mean()['y'],
    text=df_duration_ranged.groupby(pd.cut(df_campaign['campaign'], bins=20)).mean()['y'].round(2).astype(str),
    textposition='auto',
    marker=dict(color='rgb(158,202,225)', line=dict(color='rgb(8,48,107)',width=1.5))
)]

layout = go.Layout(
    title='Campaign vs Subscriptions',
    xaxis=dict(title='Campaign'),
    yaxis=dict(title='Subscription prob density')
)

fig = go.Figure(data=data, layout=layout)
fig.show()
```



From the above, we can see that the clients being contacted more are less likely to subscribe.

3.7.14) Correlation

In [33]:
```
# Create a subplot with 1 row and 1 column
fig = sp.make_subplots(rows=1, cols=1, specs=[[{}]],
                      subplot_titles=('Correlation Heatmap',))

# Compute the correlation matrix
corr = df.dropna().corr()
mask = np.triu(df.corr())

# Add heatmap trace
fig.add_trace(go.Heatmap(z=corr, x=corr.columns, y=corr.columns,
                         colorscale='Viridis', showscale=True,
                         colorbar=dict(title='Correlation', titleside='top', tickmode='array',
                                       tickvals=[-1, -0.5, 0, 0.5, 1], ticktext=['-1', '-0.5', '0', '0.5', '1']),
                         zmin=-1, zmax=1,
                         hoverongaps=False
                         ))

# Update layout
fig.update_layout(title='Correlation Heatmap',
                  xaxis=dict(title='Features'),
                  yaxis=dict(title='Features'))

fig.show()
```



From the above, we see that pdays & previous are correlated, while the rest are not considered correlated with each other.

Problem 4: Understanding the Task

After examining the description and data, your goal now is to clearly state the *Business Objective* of the task. State the objective below.

In [34]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 16 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   age          41188 non-null  int64 
 1   job          41188 non-null  object
 2   marital      41188 non-null  object
 3   education    41188 non-null  object
 4   default      41188 non-null  object
 5   housing      41188 non-null  object
 6   loan         41188 non-null  object
 7   contact      41188 non-null  object
 8   month        41188 non-null  object
 9   day_of_week  41188 non-null  object
 10  duration     41188 non-null  int64 
 11  campaign     41188 non-null  int64 
 12  pdays        41188 non-null  int64 
 13  previous     41188 non-null  int64 
 14  poutcome     41188 non-null  object
 15  y            41188 non-null  int64 
dtypes: int64(6), object(10)
memory usage: 5.0+ MB
```

Business Objective

The goal was to increase efficiency of directed campaigns for long-term deposit subscriptions by reducing the number of contacts to do.For this, we need to develop a predictive model using real-world data from a Portuguese marketing campaign for bank deposit subscriptions to increase campaign efficiency by identifying key factors that influence success and optimize resource allocation and target customer selection.

Problem 5: Engineering Features

Now that you understand your business objective, we will build a basic model to get started. Before we can do this, we must work to encode the data. Using just the bank information features (columns 1 - 7), prepare the features and target column for modeling with appropriate encoding and transformations.

5.1) Encoding

In [35]:
```
df_engg = df.copy()
```

In [36]:
```
df_engg
```

Out[36]:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome	y
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	261	1	999	0	nonexistent	0
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	149	1	999	0	nonexistent	0
2	37	services	married	high.school	no	yes	no	telephone	may	mon	226	1	999	0	nonexistent	0
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	151	1	999	0	nonexistent	0
4	56	services	married	high.school	no	no	yes	telephone	may	mon	307	1	999	0	nonexistent	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
41183	73	retired	married	professional.course	no	yes	no	cellular	nov	fri	334	1	999	0	nonexistent	1
41184	46	blue-collar	married	professional.course	no	no	no	cellular	nov	fri	383	1	999	0	nonexistent	0
41185	56	retired	married	university.degree	no	yes	no	cellular	nov	fri	189	2	999	0	nonexistent	0
41186	44	technician	married	professional.course	no	no	no	cellular	nov	fri	442	1	999	0	nonexistent	1
41187	74	retired	married	professional.course	no	yes	no	cellular	nov	fri	239	3	999	1	failure	0

41188 rows × 16 columns

In [37]:
```
#Encode job and marital columns
cols_dumm = ['job', 'marital']
df_engg = pd.get_dummies(df_engg, columns=cols_dumm)
```

In [38]:
```
# Remove the columns corresponding to unknown value
cols_onehot_unknown = [col for col in df_engg.columns if 'unknown' in col]
df_engg = df_engg.drop(cols_onehot_unknown, axis=1)
```

In [39]:
```
# Check current data set
pd.set_option('display.max_columns', None)
df_engg.head(10)
```

Out[39]:

	age	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome	y	job_admin.	job_blue-collar	job_entrepreneur	job_housemaid	job_management	job_retired	job_self-employed	j
0	56	basic.4y	no	no	no	telephone	may	mon	261	1	999	0	nonexistent	0	0	0	0	1	0	0	0	
1	57	high.school	unknown	no	no	telephone	may	mon	149	1	999	0	nonexistent	0	0	0	0	0	0	0	0	
2	37	high.school	no	yes	no	telephone	may	mon	226	1	999	0	nonexistent	0	0	0	0	0	0	0	0	
3	40	basic.6y	no	no	no	telephone	may	mon	151	1	999	0	nonexistent	0	1	0	0	0	0	0	0	
4	56	high.school	no	no	yes	telephone	may	mon	307	1	999	0	nonexistent	0	0	0	0	0	0	0	0	
5	45	basic.9y	unknown	no	no	telephone	may	mon	198	1	999	0	nonexistent	0	0	0	0	0	0	0	0	
6	59	professional.course	no	no	no	telephone	may	mon	139	1	999	0	nonexistent	0	1	0	0	0	0	0	0	
7	41	unknown	unknown	no	no	telephone	may	mon	217	1	999	0	nonexistent	0	0	1	0	0	0	0	0	
8	24	professional.course	no	yes	no	telephone	may	mon	380	1	999	0	nonexistent	0	0	0	0	0	0	0	0	
9	25	high.school	no	yes	no	telephone	may	mon	50	1	999	0	nonexistent	0	0	0	0	0	0	0	0	

5.2) Preprocessing

In [40]:
```
#map the months unsing the month package imported
mapping_month = dict((month.lower(), number) for number, month in enumerate(month_abbr))
df_engg['month'] = df_engg['month'].map(mapping_month)
```

In [41]:
```
#mao the education column column to numberical value
edu_map = {'unknown':0, 'basic.4y':1, 'basic.6y':2, 'basic.9y':3, 'high.school': 4, 'illiterate':5,'professional.course':6, 'university.degree':7}
df_engg['education'] = df_engg.education.map(edu_map).astype('int')
```

In [42]:
```
#map default, housing and loan values. We will deal with the unknown values later.
boolean_map = {'unknown':0, 'no':1, 'yes': 1}
df_engg['default'] = df_engg.default.map(boolean_map).astype('int')
df_engg['housing'] = df_engg.housing.map(boolean_map).astype('int')
df_engg['loan'] = df_engg.loan.map(boolean_map).astype('int')
```

In [43]:
```
#map the days of the week column to numberical value
day_map = {'mon': 0, 'tue': 1, 'wed':2, 'thu':3, 'fri':4}
df_engg['day_of_week'] = df_engg.day_of_week.map(day_map).astype('int')
```

In [44]:
```
#map contact types column to numberical value
contact_map = {'cellular': 0, 'telephone': 1}
df_engg['contact'] = df_engg.contact.map(contact_map).astype(int)
```

In [45]:
```
#map outcome column to numberical value
poutcome_map = {'failure': 0, 'success': 1, 'nonexistent':2}
df_engg['poutcome'] = df_engg.poutcome.map(poutcome_map).astype(int)
```

Problem 6: Train/Test Split

With your data prepared, split it into a train and test set.

6.1) Imputing Education using KNN

We will be imputing the unknown values instead of deleting the rows as the number of unknown values are pretty huge.

In [46]:
```
# Train KNN imputer using data with known education
knn_imputer = KNNImputer(n_neighbors=2,missing_values=0)

# fit the imputer on the dataset
knn_imputer.fit(df_engg[['education','default', 'housing','loan']])

# use the imputer to fill in missing values
imputed_df = knn_imputer.transform(df_engg[['education','default', 'housing','loan']])
```

In [47]:
```
# calculate the MSE score
mse = mean_squared_error(df_engg[['education','default', 'housing','loan']], imputed_df)
print("MSE score:", mse)
```

```
MSE score: 0.13149990522961696
```

In [48]:
```
# Transform the training set
imputed_data = knn_imputer.transform(df_engg[['education','default', 'housing','loan']])
```

In [49]:
```
df_engg[['education','default', 'housing','loan']] = imputed_data
```

In [50]:
```
df_encoded = df_engg.drop(['contact','month','day_of_week','duration','campaign','pdays','previous','poutcome'], axis=1)
```

In [51]:
```
X = df_encoded.drop('y', axis = 1)
y = df_encoded['y']
```

In [52]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,random_state = 52)
```

In [53]:
```
print(f"Train size : {len(X_train)}\tValidation size : {len(X_test)}")
```

```
Train size : 32950      Validation size : 8238
```

6.2) Standardization

In [54]:
```
scaler = MinMaxScaler()
# Standarding (for algorithms that can apply balanced class weights)
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

6.3) Oversampling

we will not be using this in our current analysis. However, we will be using it in our next steps and recommendations

In [55]:
```
sm = SMOTE(random_state=52)
# Only oversampling (for decision-tree-based algorithms and cannot apply balanced class weights)
X_train_sm, y_train_sm = sm.fit_resample(X_train, y_train)
# Stardardized + Oversampling (for algorithms cannot apply balanced class weights)
X_train_scaled_sm, y_train_scaled_sm = sm.fit_resample(X_train_scaled, y_train)
```

Problem 7: A Baseline Model

Before we build our first model, we want to establish a baseline. What is the baseline performance that our classifier should aim to beat?

Before we begin lets define some helper functions that we will use to output the scores. We will be using this to determine the best model.

Helper Functions

```
In [56]: r_seed = 52
def GenerateOutput(model_name,y_train,y_train_pred,x_test,y_test, y_pred,model):
    # Accuracy - Precision - Recall - F1 Score - Kappa Metrics - Confusion Matrix
    print(classification_report( y_test, y_pred, digits=2) )
    #confusion matrix display
    lr_matrix = ConfusionMatrixDisplay.from_predictions(y_test,y_pred, display_labels=['yes', 'no'])

    # =========== Balanced Dataframe Metrics =============
    #train Accuracy
    lr_train_acc = accuracy_score(y_train,y_train_pred)
    # Accuracy
    lr_acc = accuracy_score( y_test, y_pred)
    print( 'Accuracy: {}'.format(lr_acc))


    # =========== Unbalanced Dataframe Metrics =============
    # Weighted F1-Score
    f1score = f1_score( y_test, y_pred, average='weighted' )
    print( 'Weighted F1-Score: {}'.format( f1score ) )

    # Balanced Accuracy Score
    balanced_acc = balanced_accuracy_score( y_test, y_pred )
    print( 'Balanced Accuracy Score: {}'.format( balanced_acc))

    if(model_name == "Baseline model"):
        model = DummyClassifier(random_state=r_seed)
    start_time = time.time()
    model.fit(X_train, y_train)
    fit_time = time.time() - start_time

    y_pred_proba = model.predict_proba(x_test)[:, 1]
    # Calculate the AUC-ROC score
    auc = roc_auc_score(y_test, y_pred_proba)
    print("AUC-ROC:", auc)

    df_score = CalculateScores(model_name,lr_train_acc,lr_acc,f1score,balanced_acc,auc, fit_time)
    return df_score

#Return a dataframe with all the values
def CalculateScores(model_name, train_accuracy,accuracy, f1_score, balanced_accuracy, roc_auc,fit_time):
    return pd.DataFrame( { 'Model Name': model_name,
                           'Train Time': fit_time,
                           'Train Accuracy': train_accuracy,
                           'Test accuracy': accuracy,
                           'f1_score': f1_score,
                           'balanced_accuracy': balanced_accuracy,
                           'roc_auc score': roc_auc }, index=[0] )

def CrossVal_model(modelName, x_train, y_train):
    #As discussed above, we will be using stratifiedKfold to deal with imbalanced data.
    fold = 5
    kfold = StratifiedKFold(n_splits=fold, shuffle=True,random_state=r_seed)

    trainacc_list = []
    accuracy_list = []
    balanced_acc_list = []
    weighted_f1_score_list = []
    auc_score_list = []

    iter = 1
    for train_index, test_index in kfold.split(x_train, y_train):
        X_train_f, X_test_f = x_train.iloc[train_index], x_train.iloc[test_index]
        y_train_f, y_test_f = y_train.iloc[train_index], y_train.iloc[test_index]

        if(modelName == "Logistic Regression"):
            #define a Logistic Regression model
            model = LogisticRegression(random_state=r_seed)
        if(modelName == "SVM"):
            #define a SVM model
            model = svm.SVC(random_state=r_seed, probability=True)
        if(modelName == "DecisionTree"):
```

```python
            #define a decision tree model
            model = DecisionTreeClassifier(random_state=r_seed)
        if(modelName == "KNN"):
            #define a KNN model
            model = KNeighborsClassifier(random_state=r_seed)


        #fit the model
        start_time = time.time()
        model.fit(X_train_f, y_train_f)
        fit_time = time.time() - start_time
        #prediction
        y_f_pred = model.predict(X_test_f)
        y_t_pred = model.predict(X_train_f)
        #Train Accuracy,
        train_acc = accuracy_score(y_train_f, y_t_pred)
        trainacc_list.append(train_acc)
        #Accuracy
        acc = accuracy_score(y_test_f, y_f_pred)
        accuracy_list.append(acc)
        # Balanced Accuracy
        balanced_acc = balanced_accuracy_score( y_test_f, y_f_pred )
        balanced_acc_list.append( balanced_acc )
        # Weighted F1-Score
        weighted_f1_score = f1_score( y_test_f, y_f_pred, average='weighted')
        weighted_f1_score_list.append( weighted_f1_score )

        #auc score
        y_pred_proba = model.predict_proba(X_test_f)[:, 1]
        # Calculate the AUC-ROC score
        auc = roc_auc_score(y_test_f, y_pred_proba)
        auc_score_list.append(auc)
        iter += 1

    print( 'Avg Balanced Accuracy: {}'.format( np.mean( balanced_acc_list ) ) )
    print( 'Avg Weighted F1-Score: {}'.format( np.mean( weighted_f1_score_list ) ) )
    print( 'Avg AUC-ROC Score: {}'.format( np.mean( auc_score_list ) ) )

    modelName = modelName + " - Cross Validation"

    #calculate based on the mean values
    return CalculateScores(modelName,np.mean(trainacc_list),np.mean(accuracy_list),
                        np.mean( balanced_acc_list ),np.mean( weighted_f1_score_list),np.mean( auc_score_list), fit_time)
```

Baseline Model

For the baseline model, we will be using random choices to generate predictions. We will then use a dummy classifier in the above helper method to generate an output

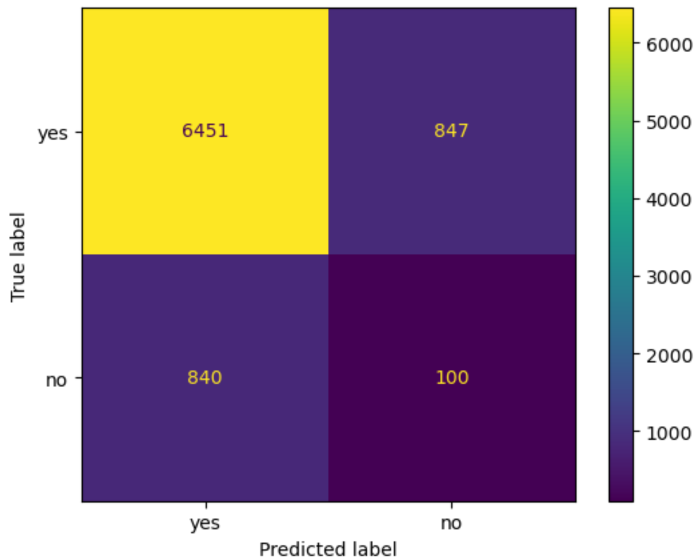
In [57]:
```python
train_list = y_train.drop_duplicates().sort_values().tolist()

target_list = y_test.drop_duplicates().sort_values().tolist()
target_weights = df_engg['y'].value_counts( normalize=True ).sort_index().tolist()
# prediction
y_train_pred = random.choices(train_list, k=X_train_scaled.shape[0],
                              weights=target_weights )
y_pred= random.choices(target_list, k=X_test_scaled.shape[0],
                              weights=target_weights )
```

In [58]:
```
df_score_base = GenerateOutput("Baseline model",y_train,y_train_pred,X_test_scaled,y_test, y_pred, DummyClassifier())
df_score_base
```

```
              precision    recall  f1-score   support

           0       0.88      0.88      0.88      7298
           1       0.11      0.11      0.11       940

    accuracy                           0.80      8238
   macro avg       0.50      0.50      0.50      8238
weighted avg       0.80      0.80      0.80      8238

Accuracy: 0.7952172857489682
Weighted F1-Score: 0.7955479871411996
Balanced Accuracy Score: 0.49516189221179807
AUC-ROC: 0.5
```

Out[58]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	Baseline model	0.000601	0.80003	0.795217	0.795548	0.495162	0.5



Problem 8: A Simple Model

Use Logistic Regression to build a basic model on your data.

Logistic Regression is a statistical method that we use to fit a regression model when the response variable is binary. It is a type of generalized linear model (GLM) that uses a logistic function to model a binary dependent variable.

In logistic regression, the goal is to find the best fitting model to describe the relationship between the independent variables (predictors) and the dependent binary variable.Logistic Regression is a simple yet powerful method for modeling binary data and is widely used in various fields, including medical research, economics, and social sciences. It has several advantages such as it's simple to implement, efficient to train and easy to interpret. However, it also has some limitations, such as it's assumption of linearity between the independent variables and the log-odds of the response, and it's inability to model complex non-linear relationships.

In [59]:
```
#model definition
mlm_lr = LogisticRegression(random_state=r_seed)
#fit the model
mlm_lr.fit(X_train_scaled, y_train)
```

Out[59]: LogisticRegression(random_state=52)

Problem 9: Score the Model

What is the accuracy of your model?

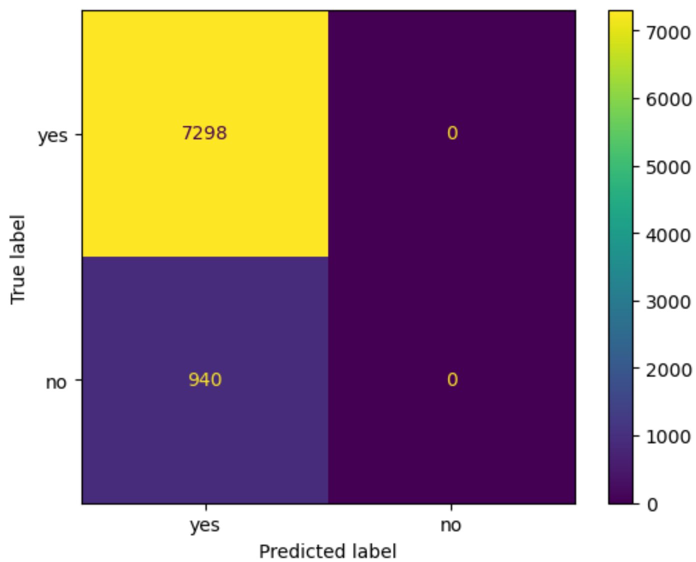
In [60]:
```python
y_pred = mlm_lr.predict(X_test_scaled)
y_train_pred = mlm_lr.predict(X_train_scaled)
```

In [61]:
```python
df_score = GenerateOutput("Logistic Regression",y_train,y_train_pred,X_test_scaled,y_test, y_pred, mlm_lr)
df_score
```

```
              precision    recall  f1-score   support

           0       0.89      1.00      0.94      7298
           1       0.00      0.00      0.00       940

    accuracy                           0.89      8238
   macro avg       0.44      0.50      0.47      8238
weighted avg       0.78      0.89      0.83      8238

Accuracy: 0.8858946346200534
Weighted F1-Score: 0.8322939036376351
Balanced Accuracy Score: 0.5
AUC-ROC: 0.6173354110423724
```

Out[61]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	Logistic Regression	0.215692	0.887709	0.885895	0.832294	0.5	0.617335



Problem 10: Model Comparisons

Now, we aim to compare the performance of the Logistic Regression model to our KNN algorithm, Decision Tree, and SVM models. Using the default settings for each of the models, fit and score each. Also, be sure to compare the fit time of each of the models. Present your findings in a `DataFrame` similar to that below:

Model	Train Time	Train Accuracy	Test Accuracy
		.	.

10.1) KNN

K-Nearest Neighbors (KNN) is a simple and popular machine learning algorithm that can be used for both classification and regression.

In the case of KNN classification, the algorithm works by finding the K nearest neighbors of a given data point and using their class labels to predict the class label of the data point. For example, if the K nearest neighbors of a data point are all labeled as "positive," the data point is predicted to be positive as well.

On the other hand, KNN regression works by finding the K nearest neighbors of a given data point and using their values to predict the value of the data point. For example, if the K nearest neighbors of a data point have values of 1, 2, and 3, the predicted value for the data point might be 2 (the average of the values of the nearest neighbors).

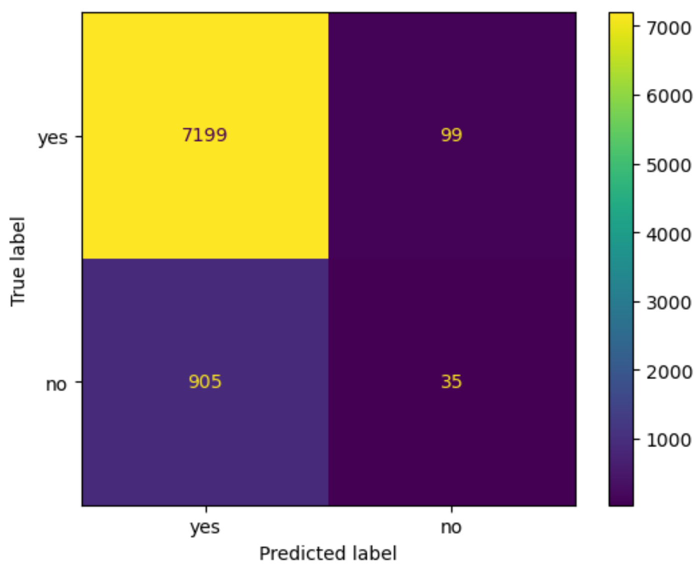
In [62]:

```
# model definition
model_knn = KNeighborsClassifier(n_neighbors=2)
# train model
model_knn.fit(X_train_scaled,y_train)
#predict the test data and train data
y_pred = model_knn.predict(X_test_scaled)
y_train_pred = model_knn.predict(X_train_scaled)
#Performance
df_score1 = GenerateOutput('KNN',y_train,y_train_pred,X_test_scaled,y_test, y_pred, model_knn)
df_score1
```

```
              precision    recall  f1-score   support

           0       0.89      0.99      0.93      7298
           1       0.26      0.04      0.07       940

    accuracy                           0.88      8238
   macro avg       0.57      0.51      0.50      8238
weighted avg       0.82      0.88      0.84      8238

Accuracy: 0.8781257586792911
Weighted F1-Score: 0.8355834412503123
Balanced Accuracy Score: 0.5118343410902433
AUC-ROC: 0.5
```

Out[62]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	KNN	0.001909	0.889014	0.878126	0.835583	0.511834	0.5



10.2) Support Vector Machines

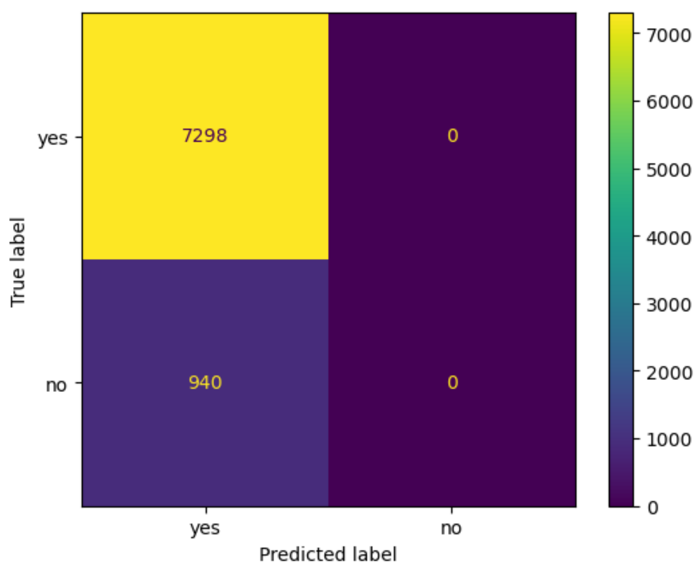
The main idea behind SVMs is to find the line (or hyperplane) that maximally separates the data points of different classes. This line is called the "maximum margin hyperplane." The points that lie closest to this line are called "support vectors." Once the support vectors are found, the algorithm uses them to construct the maximum margin hyperplane. However, SVMs can be sensitive to the choice of hyperparameters and can be computationally expensive to train, especially for large datasets. They also do not work well with noisy or highly imbalanced data, and they may not be suitable for tasks that require probability estimates.

In [63]:
```
# model definition
model_svm = svm.SVC(random_state=r_seed,probability=True)
# model training
model_svm.fit( X_train_scaled, y_train )
#predict the test data and train data
y_pred = model_svm.predict(X_test_scaled)
y_train_pred = model_svm.predict(X_train_scaled)
#Performance
df_score2 = GenerateOutput('SVM',y_train,y_train_pred,X_test_scaled,y_test, y_pred, model_svm)
df_score2
```

```
              precision    recall  f1-score   support

           0       0.89      1.00      0.94      7298
           1       0.00      0.00      0.00       940

    accuracy                           0.89      8238
   macro avg       0.44      0.50      0.47      8238
weighted avg       0.78      0.89      0.83      8238

Accuracy: 0.8858946346200534
Weighted F1-Score: 0.8322939036376351
Balanced Accuracy Score: 0.5
AUC-ROC: 0.470892928986665
```

Out[63]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	SVM	112.321783	0.887709	0.885895	0.832294	0.5	0.470893



10.3) Decision Trees

A decision tree works by recursively partitioning the input data into subsets, called branches, based on the values of the input features. Each internal node in the tree represents a feature, and each branch represents a possible value of that feature. The leaves of the tree represent the predicted class or output value.

The process of building a decision tree begins with selecting the feature that best splits the data into subsets that are as pure as possible. This feature is chosen by evaluating a metric such as information gain, Gini index or gain ratio. Once a feature is selected, the data is split according to the values of that feature, and the process is repeated for each subset of the data. This continues until the tree reaches a stopping criterion, such as a maximum depth or a minimum number of samples per leaf.

In [64]:
```
# model definition
model_clf = DecisionTreeClassifier()
# model training
model_clf.fit( X_train_scaled, y_train )
```

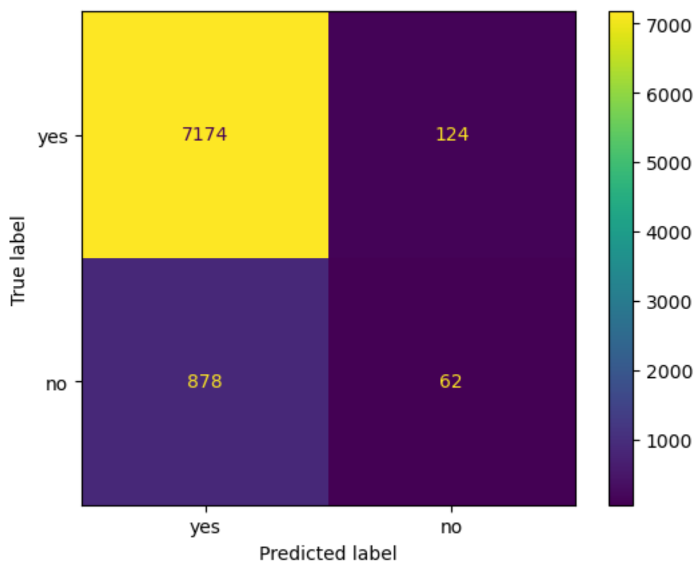
Out[64]:
```
DecisionTreeClassifier()
```

In [65]:
```
#predict the test data and train data
y_pred = model_clf.predict(X_test_scaled)
y_train_pred = model_clf.predict(X_train_scaled)
#performance
df_score3 = GenerateOutput('DecisionTree',y_train, y_train_pred, X_test_scaled, y_test, y_pred, model_clf)
df_score3
```

```
              precision    recall  f1-score   support

           0       0.89      0.98      0.93      7298
           1       0.33      0.07      0.11       940

    accuracy                           0.88      8238
   macro avg       0.61      0.52      0.52      8238
weighted avg       0.83      0.88      0.84      8238

Accuracy: 0.8783685360524399
Weighted F1-Score: 0.8406319804561019
Balanced Accuracy Score: 0.5244832451910462
AUC-ROC: 0.545889284735544
```

Out[65]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	DecisionTree	0.03925	0.898877	0.878369	0.840632	0.524483	0.545889



Problem 11: Improving the Model

Now that we have some basic models on the board, we want to try to improve these. Below, we list a few things to explore in this pursuit.

- More feature engineering and exploration. For example, should we keep the gender feature? Why or why not?
- Hyperparameter tuning and grid search. All of our models have additional hyperparameters to tune and explore. For example the number of neighbors in KNN or the maximum depth of a Decision Tree.
- Adjust your performance metric

11.1) Feature Importance

Permutation importance is a technique used to determine the importance of individual features in a machine learning model. It works by randomly shuffling the values of a single feature and evaluating the effect on the model's performance. The idea is that if a feature is important, then shuffling its values should result in a significant decrease in model performance.

In [66]:
```
#Let us perform permutation importance in order to determine the features.
X = df_engg.drop('y', axis = 1)
y = df_engg['y']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,random_state = 52)

print(f"Train size : {len(X_train)}\tValidation size : {len(X_test)}")

# Create a list of feature names
feature_names = ['Feature ' + str(i) for i in range(X.shape[1])]

# train a model
# model definition
model = DecisionTreeClassifier()
# model training
model.fit( X_train, y_train )

# calculate the permutation importance of each feature
result = permutation_importance(model, X_test, y_test, n_repeats=10, random_state=0)

# extract the importance scores
importance_scores = result.importances_mean

# create a dictionary mapping feature names to importance scores
feature_importance = dict(zip(X_test.columns, importance_scores))

# sort the features by importance
sorted_features = sorted(feature_importance.items(), key=lambda x: x[1], reverse=True)
```

```
Train size : 32950	Validation size : 8238
```

In [67]:
```
sorted_features
```

Out[67]:
```
[('duration', 0.043529983005583886),
 ('month', 0.0433600388443797),
 ('contact', 0.022250546249089554),
 ('pdays', 0.015683418305413930),
 ('age', 0.006797766448167031),
 ('poutcome', 0.006057295460063094),
 ('previous', 0.004819130857004128),
 ('education', 0.0039329934450109015),
 ('day_of_week', 0.0017965525613012456),
 ('job_student', 0.0012624423403738484),
 ('job_blue-collar', 0.0011410536537994399),
 ('job_housemaid', 0.0009468317552803862),
 ('campaign', 0.0007647487254187846),
 ('job_entrepreneur', 0.0004369992716678706),
 ('job_unemployed', 0.0004248604030104297),
 ('marital_married', 0.00038844379703810715),
 ('marital_single', 0.00037630492838066634),
 ('job_services', 0.0003034717164360212),
 ('job_admin.', 0.00012138868657439739),
 ('job_retired', 0.00010924981791696765),
 ('default', 0.0),
 ('housing', 0.0),
 ('loan', 0.0),
 ('job_management', -3.6416605972322544e-05),
 ('job_self-employed', -8.497208060208594e-05),
 ('marital_divorced', -0.0003277494537509029),
 ('job_technician', -0.0005341102209274085)]
```

Based on the feature importance, we see that the month and duration are two features that have a higher feature importance compared to other features. We will add these two features to the dataframe.

In [68]:
```
df_perm_imp = df_engg.drop(['contact','day_of_week','campaign','pdays','previous','poutcome'], axis=1)
```

In [69]:
```
X = df_perm_imp.drop('y', axis = 1)
y = df_perm_imp['y']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,random_state = 52)
```

11.1) Logistic Regression - Cross Validation

StratifiedKFold is a type of cross-validation technique used to evaluate the performance of machine learning models. It is a variant of KFold, which divides the data into a specified number of folds (or "splits") and iteratively trains and evaluates the model on each fold. The key difference between StratifiedKFold and KFold is that StratifiedKFold ensures that the proportion of samples belonging to each class is approximately the same across all the folds. This is particularly useful when the data is imbalanced, meaning that one class is significantly more prevalent than the others.

In [70]:
```
df_score4 = CrossVal_model("Logistic Regression", X_train, y_train)
df_score4
```

```
Avg Balanced Accuracy: 0.5836696696696697
Avg Weighted F1-Score: 0.8682089274899699
Avg AUC-ROC Score: 0.8429837144837145
```

Out[70]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	Logistic Regression - Cross Validation	0.178041	0.89409	0.893748	0.58367	0.868209	0.842984

11.2) K-neighbors fine tuning

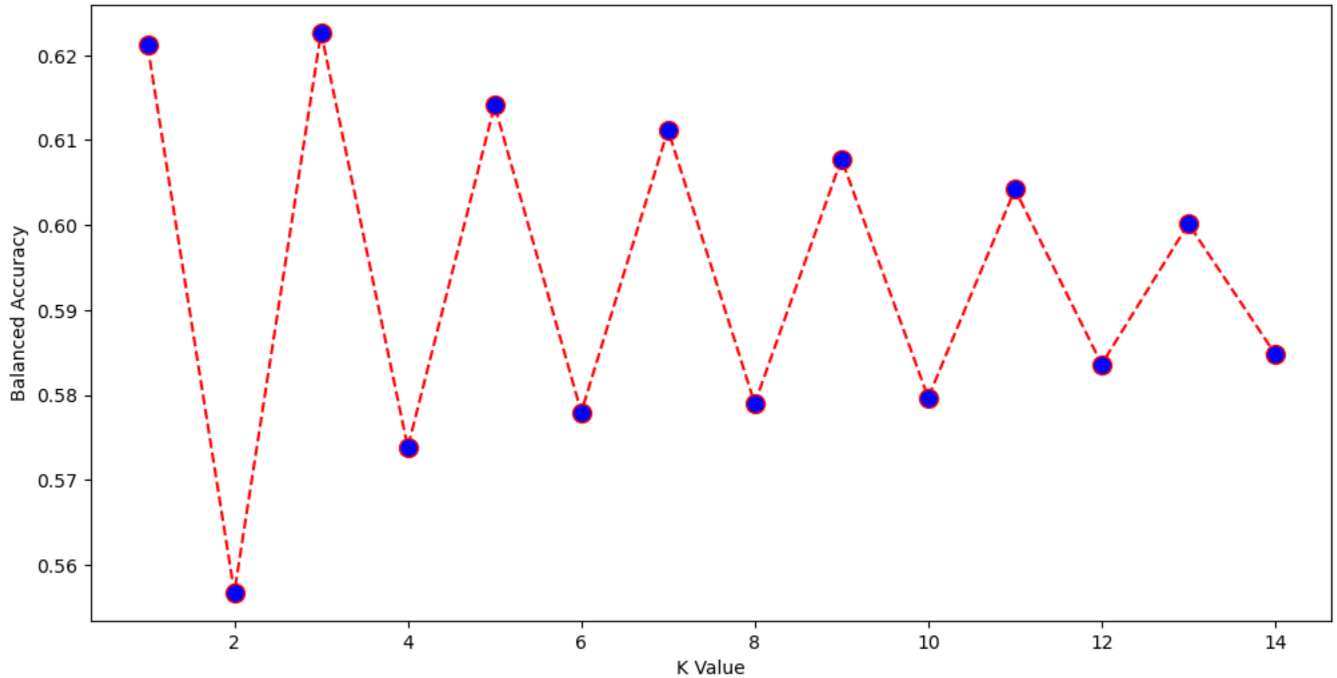
Fine-tuning a KNN model involves adjusting the parameters of the algorithm to optimize its performance. The main parameter that can be adjusted in KNN is the value of "K", which determines the number of nearest neighbors to consider when making predictions. A smaller value of K will make the model more sensitive to individual data points, while a larger value of K will make the model more robust to outliers.

In order to fine-tune a KNN model, you can use a technique called cross-validation. This involves dividing the data into a training set and a validation set, training the model on the training set, and evaluating its performance on the validation set. This process can be repeated multiple times, using different combinations of the parameters, and the model with the best performance on the validation set can be chosen as the final model.

In [71]:
```
balanced_acc_list = []
for i in range( 1, 15 ):
    # model definition
    model_knn = KNeighborsClassifier(n_neighbors=i, n_jobs=-1 )
    # train model
    model_knn.fit(X_train, y_train )
    # prediction
    y_pred = model_knn.predict( X_test )
    # Balanced Accuracy Score
    balanced_acc_list.append( balanced_accuracy_score( y_test, y_pred))
```

In [72]:
```
plt.figure( figsize=(12, 6) )
plt.plot( range( 1, 15 ), balanced_acc_list, color='red', linestyle='dashed', marker='o',
                                             markerfacecolor='blue', markersize=10 )
plt.xlabel('K Value' )
plt.ylabel('Balanced Accuracy')
```

Out[72]:
```
Text(0, 0.5, 'Balanced Accuracy')
```



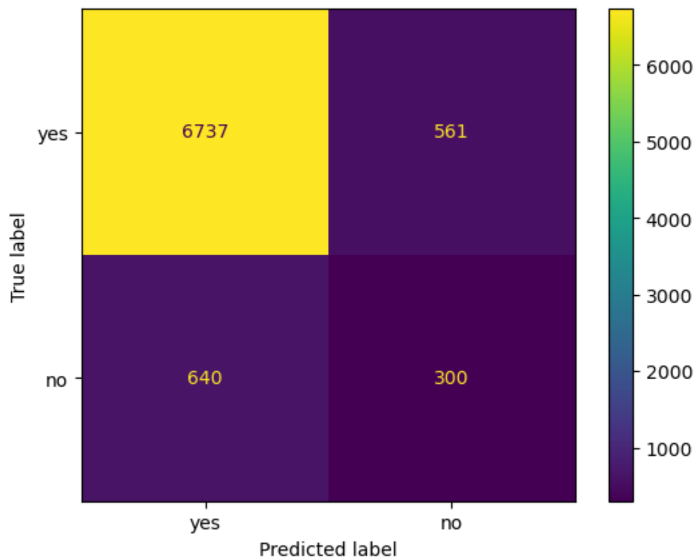
From the above, it is clear that balanced accuracy is greater for K values 1 and 3. We will be using K-Value of 1 for fine tuning the results

In [73]:
```
# model definition
model_knn = KNeighborsClassifier(n_neighbors=1, n_jobs=-1 )
# train model
model_knn.fit( X_train, y_train)
#predict the test data and train data
y_train_pred = model_knn.predict(X_train)
y_pred = model_knn.predict( X_test )
#Performance
df_score5 = GenerateOutput('KNN',y_train,y_train_pred,X_test,y_test, y_pred, model_knn)
df_score5
```

```
              precision    recall  f1-score   support

           0       0.91      0.92      0.92      7298
           1       0.35      0.32      0.33       940

    accuracy                           0.85      8238
   macro avg       0.63      0.62      0.63      8238
weighted avg       0.85      0.85      0.85      8238

Accuracy: 0.8542121874241321
Weighted F1-Score: 0.8514071413607764
Balanced Accuracy Score: 0.6211392803624427
AUC-ROC: 0.6211392803624426
```

Out[73]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	KNN	0.001961	0.999575	0.854212	0.851407	0.621139	0.621139



In this case, an ROC curve of 0.62 indicates that the model's performance is better than random chance but not perfect. The AUC of 0.62 is above 0.5, meaning that the model is able to discriminate to some extent between the positive and negative classes, but it's not a perfect model. The train time is pretty good compared to other models.

11.3) SVM - Cross Validation

Cross-validation is a technique that can be used to evaluate the performance of a SVM model and fine-tune its parameters. It involves dividing the data into multiple subsets, called "folds", and training the model on different combinations of the folds. The most commonly used techniques for cross-validation for SVM are k-fold cross-validation and Leave-One-Out cross-validation (LOOCV).

In [74]:
```
#Performance - the method is defined in the helper methods
df_score6 = CrossVal_model("SVM", X_train, y_train)
df_score6
```

```
Avg Balanced Accuracy: 0.5825592515592515
Avg Weighted F1-Score: 0.8680776556426398
Avg AUC-ROC Score: 0.6973198198198197
```

Out[74]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	SVM - Cross Validation	95.016824	0.894105	0.894082	0.582559	0.868078	0.69732

In this case, an ROC curve of 0.69 indicates that the model's performance is better than random chance but not perfect. The AUC of 0.69 is above 0.5, meaning that the model is able to discriminate to some extent between the positive and negative classes, but it's not a perfect model. The train accuracy and test accuracy are pretty good. However, the train time is pretty bad.

11.4) DecisionTreeClassifier - Cross Validation

The main parameters that can be adjusted in decision tree are the maximum depth of the tree, the minimum number of samples required to split an internal node, the minimum number of samples in a leaf node and the criterion used to split the nodes.

For decision trees, cross-validation is important because it allows to evaluate the performance of the model and select the best combination of parameters that minimize the overfitting problem. Overfitting occurs when a decision tree is too complex and is able to fit the noise in the data. This can be mitigated by techniques such as pruning or using ensembles of decision trees.

In [75]:
```
#Performance - the method is defined in the helper methods
df_score7 = CrossVal_model("DecisionTree", X_train, y_train)
df_score7
```

```
Avg Balanced Accuracy: 0.6576417186417186
Avg Weighted F1-Score: 0.8610096934307316
Avg AUC-ROC Score: 0.6577968583968585
```

Out[75]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	DecisionTree - Cross Validation	0.062328	0.999643	0.859727	0.657642	0.86101	0.657797

In this case, an ROC curve of 0.65 indicates that the model's performance is better than random chance but not perfect. The AUC of 0.65 is above 0.5, meaning that the model is able to discriminate to some extent between the positive and negative classes, but it's not a perfect model.

Hyperparameter Tuning

11.5) Logistic Regression - GridSearchCV

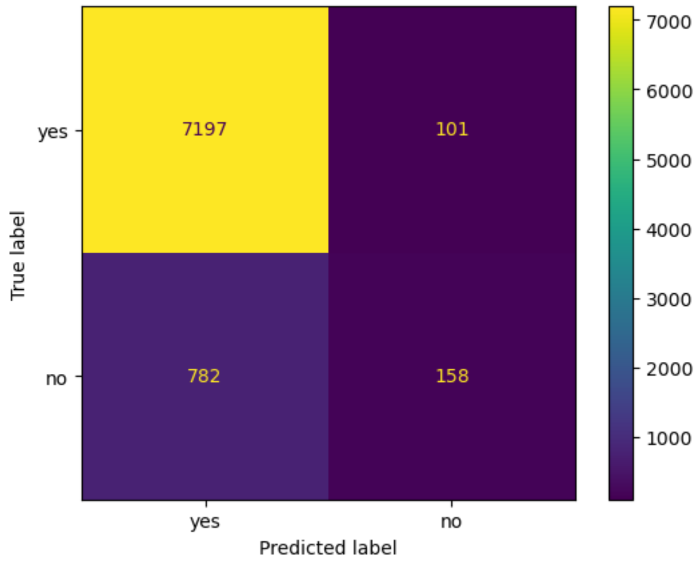
In logistic regression, the main parameters that can be fine-tuned are the regularization parameter (C) and the solver. The regularization parameter (C) controls the trade-off between maximizing the margin and minimizing the classification error. The solver determines the algorithm used to optimize the parameters.

In [76]:
```
# Define the logistic regression model
logreg = LogisticRegression()

# Define the parameter grid
param_grid = {
    'C': [0.1, 1, 10, 100],
    'penalty': ['l1', 'l2']
}

# Create the GridSearchCV object
grid_search = GridSearchCV(estimator=logreg, param_grid=param_grid, cv=5, scoring='accuracy')

# Fit the model
grid_search.fit(X_train, y_train)

#prediction
y_pred = grid_search.predict(X_test)

#Print the best params
best_params = grid_search.best_params_
best_score = grid_search.best_score_
print(best_params)
print(best_score)

# Use the best hyperparameters to fit the final model
logreg_best = LogisticRegression(**best_params)
logreg_best.fit(X_train, y_train)

# Make predictions on the test set
predictions = logreg_best.predict(X_test)
y_train_pred = logreg_best.predict(X_train)
df_score8 = GenerateOutput('LogisticRegression-GridSearchCV', y_train, y_train_pred,X_test,y_test, predictions, logreg_best)
df_score8
```

```
{'C': 100, 'penalty': 'l2'}
0.8938088012139606
              precision    recall  f1-score   support

           0       0.90      0.99      0.94      7298
           1       0.61      0.17      0.26       940

    accuracy                           0.89      8238
   macro avg       0.76      0.58      0.60      8238
weighted avg       0.87      0.89      0.86      8238

Accuracy: 0.8928137897547949
Weighted F1-Score: 0.8647633463092386
Balanced Accuracy Score: 0.5771228491629884
AUC-ROC: 0.8348812411444697
```

Out[76]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	LogisticRegression-GridSearchCV	0.205961	0.894021	0.892814	0.864763	0.577123	0.834881



An ROC curve of 0.52 indicates that the model's performance is not very good. The AUC of 0.52 is close to 0.5, meaning that the model does not discriminate well between the positive and negative classes, the model is not able to make a clear separation between the two classes.

11.6) KNN - RandomSearchCV

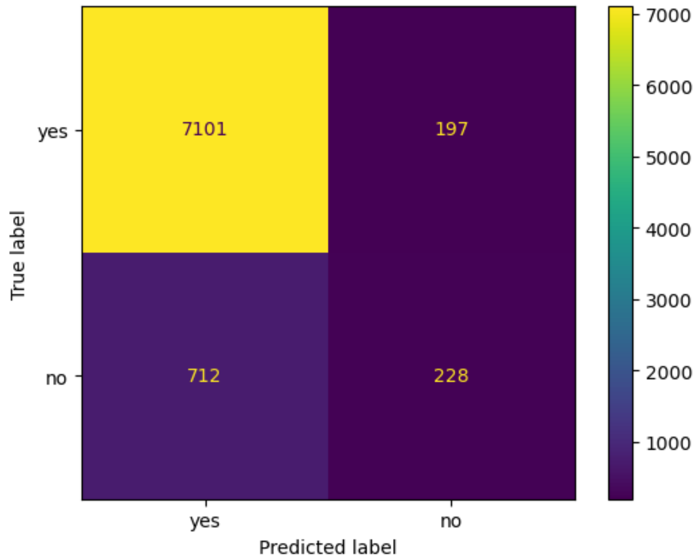
The param_dist dictionary specifies the hyperparameters to search over and the possible values for each. The n_iter parameter specifies the number of random combinations to try. The cv parameter specifies the number of folds to use in cross-validatio

In [77]:
```
# Set up the parameter distribution for the KNN hyperparameters
param_dist = {'n_neighbors': [1, 3, 5, 7, 9],
              'weights': ['uniform', 'distance'],
              'metric': ['euclidean', 'manhattan']}

# Initialize the KNN model
knn = KNeighborsClassifier()

# Initialize the randomized search
random_search = RandomizedSearchCV(estimator=knn, param_distributions=param_dist, cv=5, n_iter=10)

# Fit the randomized search to the data
random_search.fit(X_train, y_train)

# Print the best hyperparameters
print(random_search.best_params_)

# Print the best cross-validation score
print(random_search.best_score_)
# Predict on the test set
y_pred = random_search.predict(X_test)
y_train_pred = random_search.predict(X_train)
#performance
df_score9 = GenerateOutput('KNN-RandomSearchCV',y_train,y_train_pred,X_test,y_test, y_pred, random_search)
df_score9
```

```
{'weights': 'uniform', 'n_neighbors': 9, 'metric': 'euclidean'}
0.891350531107739
              precision    recall  f1-score   support

           0       0.91      0.97      0.94      7298
           1       0.54      0.24      0.33       940

    accuracy                           0.89      8238
   macro avg       0.72      0.61      0.64      8238
weighted avg       0.87      0.89      0.87      8238

Accuracy: 0.8896576839038601
Weighted F1-Score: 0.8707224876119846
Balanced Accuracy Score: 0.6077797472930503
AUC-ROC: 0.7942707270426758
```

Out[77]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	KNN-RandomSearchCV	109.194334	0.904036	0.889658	0.870722	0.60778	0.794271



In this case, an ROC curve of 0.79 indicates that the model's performance is good. The AUC of 0.79 is well above 0.5, meaning that the model is able to discriminate well between the positive and negative classes. The train time is not good compared to the other models.

11.7) DecisionTrees - RandomizedSearchCV

In decision trees, the main parameters that can be fine-tuned are the maximum depth of the tree, the minimum number of samples required to split an internal node, the minimum number of samples in a leaf node, the criterion used to split the nodes and the maximum number of features to consider when looking for the best split.

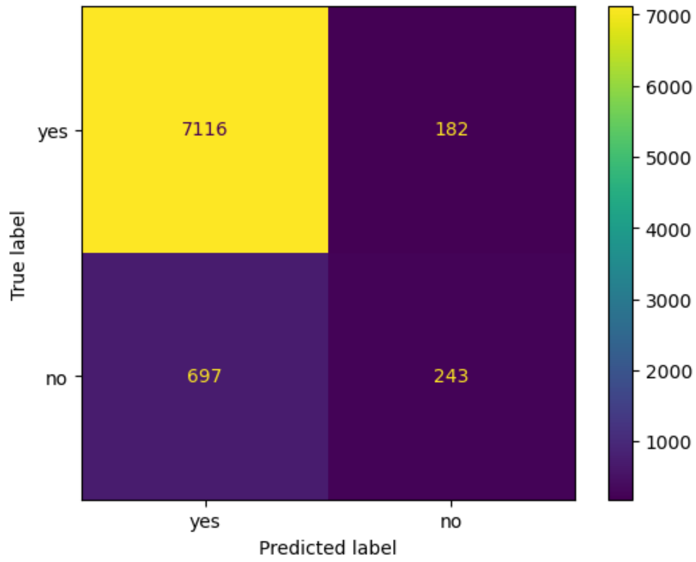
Random search can be more computationally efficient than grid search, especially when the number of hyperparameters and their possible values is large, but it doesn't guarantee to always find the best combination of parameters.

In [78]:
```
# Set up the parameter distribution for the decision tree hyperparameters
param_dist = {'max_depth': [1,3,5,7,9],
              'min_samples_split': [2,4,6,8,10,12,14,16],
              'min_samples_leaf': [1,3,5,7,9,13,15,17]}

# Initialize the decision tree model
tree = DecisionTreeClassifier()

# Initialize the randomized search
random_search = RandomizedSearchCV(estimator=tree, param_distributions=param_dist, cv=5, n_iter=10)

# Fit the randomized search to the data
random_search.fit(X_train, y_train)

# Print the best hyperparameters
print(random_search.best_params_)

# Print the best cross-validation score
print(random_search.best_score_)

# Predict on the test set
y_pred = random_search.predict(X_test)
y_train_pred = random_search.predict(X_train)

#performance
df_score10 = GenerateOutput('DecisonTree-RandomSearchCV',y_train, y_train_pred,X_test,y_test, y_pred,random_search)
df_score10
```

```
{'min_samples_split': 8, 'min_samples_leaf': 15, 'max_depth': 5}
0.897389984825493
              precision    recall  f1-score   support

           0       0.91      0.98      0.94      7298
           1       0.57      0.26      0.36       940

    accuracy                           0.89      8238
   macro avg       0.74      0.62      0.65      8238
weighted avg       0.87      0.89      0.87      8238

Accuracy: 0.8932993445010925
Weighted F1-Score: 0.8749890721792458
Balanced Accuracy Score: 0.6167861495134196
AUC-ROC: 0.8416034996472364
```

Out[78]:

	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	DecisonTree-RandomSearchCV	1.117782	0.90085	0.893299	0.874989	0.616786	0.841603



In this case, an ROC curve of 0.62 indicates that the model's performance is better than random chance but not perfect. The AUC of 0.62 is above 0.5, meaning that the model is able to discriminate to some extent between the positive and negative classes, but it's not a perfect model.

12) Evaluation

With some modeling accomplished, we aim to reflect on what we identify as a high quality model and what we are able to learn from this. We should review our business objective and explore how well we can provide meaningful insight on drivers of used car prices. Your goal now is to distill your findings and determine whether the earlier phases need revisitation and adjustment or if you have information of value to bring back to your client.

12.1) Metric Evaluation

12.1.1) Identification of Evaluation Metric

The following are evaluation metrics used for this particular binary classification.

Accuracy: This is the percentage of correct predictions made by the classifier. It is calculated as (true positives + true negatives) / total samples.

Precision: This is the percentage of positive predictions that were correct. It is calculated as true positives / (true positives + false positives).

F1 score: This is a weighted average of precision and recall, with a higher score indicating better performance. It is calculated as 2 * (precision * recall) / (precision + recall).

AUC (Area Under the Curve): This is a metric used to evaluate the performance of a binary classifier using an ROC (Receiver Operating Characteristic) curve. The ROC curve plots the true positive rate against the false positive rate at different classification thresholds, and the AUC is the area under this curve. AUC provides a single measure of the classifier's performance.

Balanced accuracy: This is a metric used to evaluate the performance of a classifier when the classes are imbalanced. It is defined as the average of the class-specific accuracies, where the class-specific accuracy for a class is the number of true positives for that class divided by the sum of the true positives and false negatives for that class.

Confusion matrix is a table that is used to evaluate the performance of a classification model. It is used to compare the predicted values with the actual values. The matrix is divided into four quadrants: true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Based on the confusion matrix values, several other evaluation metrics can be calculated such as precision, recall, F1-score, and accuracy. These metrics provide a more comprehensive understanding of the model performance, especially when the classes are imbalanced.

In [79]:
```
#Add all the scores of the models
modelling_result = pd.concat( [df_score_base,
                               df_score,
                               df_score2,
                               df_score3,
                               df_score4,
                               df_score5,
                               df_score6,
                               df_score7,
                               df_score8,
                               df_score9,
                               df_score10
                              ] )
modelling_result.sort_values( 'roc_auc score' )
```

Out[79]:

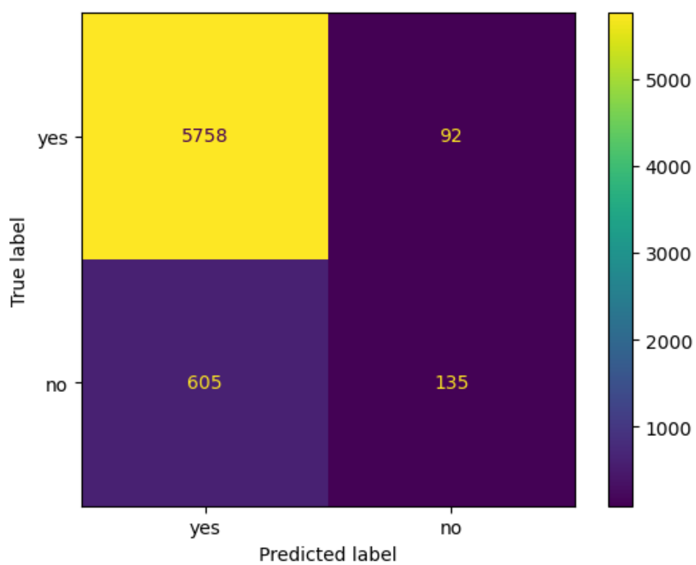
	Model Name	Train Time	Train Accuracy	Test accuracy	f1_score	balanced_accuracy	roc_auc score
0	SVM	112.321783	0.887709	0.885895	0.832294	0.500000	0.470893
0	Baseline model	0.000601	0.800030	0.795217	0.795548	0.495162	0.500000
0	DecisionTree	0.039250	0.898877	0.878369	0.840632	0.524483	0.545889
0	Logistic Regression	0.215692	0.887709	0.885895	0.832294	0.500000	0.617335
0	KNN	0.001961	0.999575	0.854212	0.851407	0.621139	0.621139
0	DecisionTree - Cross Validation	0.062328	0.999643	0.859727	0.657642	0.861010	0.657797
0	SVM - Cross Validation	95.016824	0.894105	0.894082	0.582559	0.868078	0.697320
0	KNN-RandomSearchCV	109.194334	0.904036	0.889658	0.870722	0.607780	0.794271
0	LogisticRegression-GridSearchCV	0.205961	0.894021	0.892814	0.864763	0.577123	0.834881
0	DecisonTree-RandomSearchCV	1.117782	0.900850	0.893299	0.874989	0.616786	0.841603
0	Logistic Regression - Cross Validation	0.178041	0.894090	0.893748	0.583670	0.868209	0.842984

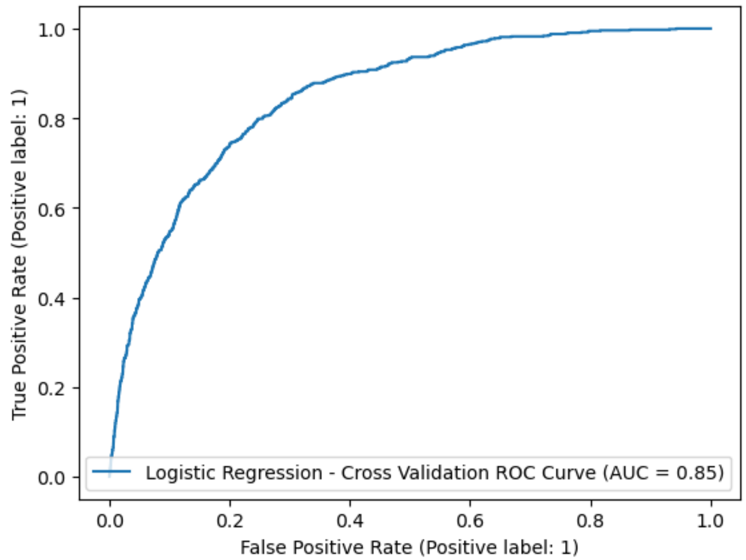
From the above, the train time and test accuracy are not the best. However, the roc_auc score is higher compared to other models for the Logistic Regression with Cross Validation. From this analysis, it is clear that Logistic Regression is the best model to predict the outcome of subscribing to the outcome.

Best Model - Logistic Regression with Cross Validation

In [80]:
```
#Lets get into details of Logistic Regression Cross Validation. We will print out the confusion Matrix and coef's
x_train = X_train
fold = 5
kfold = StratifiedKFold(n_splits=fold, shuffle=True,random_state=r_seed)


iter = 1
for train_index, test_index in kfold.split(x_train, y_train):
    X_train_f, X_test_f = x_train.iloc[train_index], x_train.iloc[test_index]
    y_train_f, y_test_f = y_train.iloc[train_index], y_train.iloc[test_index]

    model = LogisticRegression(random_state=r_seed)

    #fit the model
    start_time = time.time()
    model.fit(X_train_f, y_train_f)
    fit_time = time.time() - start_time
    #prediction
    y_f_pred = model.predict(X_test_f)
    y_t_pred = model.predict(X_train_f)
    #Train Accuracy,
    train_acc = accuracy_score(y_train_f, y_t_pred)
    #Accuracy
    acc = accuracy_score(y_test_f, y_f_pred)
    # Balanced Accuracy
    balanced_acc = balanced_accuracy_score( y_test_f, y_f_pred )
    # Weighted F1-Score
    weighted_f1_score = f1_score( y_test_f, y_f_pred, average='weighted')
    #coef
    coef = model.coef_
    #auc score
    y_pred_proba = model.predict_proba(X_test_f)[:, 1]
    # Calculate the AUC-ROC score
    auc = roc_auc_score(y_test_f, y_pred_proba)

    #I have ran this multiple times and came to a conclusion that iter 3 has the best ROC value.
    if(iter == 3):
        #confusion matrix display
        lr_matrix = ConfusionMatrixDisplay.from_predictions(y_test_f,y_f_pred, display_labels=['yes', 'no'])
        print('***********************************')
        print( 'Train Time: {}'.format(fit_time))
        print( 'Train Accuracy: {}'.format(train_acc))
        print( 'Test Accuracy: {}'.format(acc) )
        print( 'F1-Score: {}'.format( weighted_f1_score ) )
        print( 'AUC-ROC Score: {}'.format( auc ) )
        columns = X_train.columns
        print('***********************************')
        print( '*** Coefficients of Logistic Regression ***')
        for feature, coef_ in zip(columns, model.coef_[0]):
            print(f'{feature} coefficient: {coef_}')
        print('***********************************')
        RocCurveDisplay.from_predictions(y_true=y_test_f, y_pred=y_pred_proba, name="Logistic Regression - Cross Validation ROC Curve")
    iter += 1
```

```
**********************************
Train Time: 0.16910386085510254
Train Accuracy: 0.8939301972685888
Test Accuracy: 0.8942336874051593
F1-Score: 0.8684001439066389
AUC-ROC Score: 0.8501158466158466
**********************************
*** Coefficients of Logistic Regression ***
age coefficient: 0.010726302257827423
education coefficient: 0.06850397218006317
default coefficient: -0.9633732418676771
housing coefficient: -0.9633732418676771
loan coefficient: -0.9633732418676771
month coefficient: 0.03691696044280908
duration coefficient: 0.0038242986720552143
job_admin. coefficient: -0.13751858433791297
job_blue-collar coefficient: -0.7964306227936814
job_entrepreneur coefficient: -0.2903781378720096
job_housemaid coefficient: -0.0943807504632312
job_management coefficient: -0.07421008696189652
job_retired coefficient: 0.8233510651846832
job_self-employed coefficient: -0.18384534598588315
job_services coefficient: -0.6321812576319007
job_student coefficient: 0.6599592099282632
job_technician coefficient: -0.35773999064286394
job_unemployed coefficient: 0.11378381279040874
marital_divorced coefficient: -0.6535152775927779
marital_married coefficient: -0.42289591460452575
marital_single coefficient: 0.13621253517677157
**********************************
```





The train time is the time it took to train the model, in this case 0.18149399757385254 seconds. Train accuracy is the proportion of correctly classified samples in the training set, which is 89.39% in this case. Test accuracy is the proportion of correctly classified samples in the test set, which is 89.42% in this case. F1-Score is a measure of a test's accuracy. It considers both the precision and the recall of the test to compute the score. The F1 score for this model is 0.8684

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performance of a binary classification model. It plots the true positive rate (TPR) against the false positive rate (FPR) at different classification thresholds.

A ROC curve is generated by plotting the TPR and FPR values for a range of classification thresholds. The area under the ROC curve (AUC) is a measure of the model's overall performance. An AUC of 1 represents a perfect model and an AUC of 0.5 represents a model with no discrimination ability.

In this case, an ROC curve of 0.85 indicates that the model's performance is good. The AUC of 0.85 is well above 0.5, meaning that the model is able to discriminate well between the positive and negative classes.

13) Interpretation of Coefficients

The coefficients of a logistic regression model can be used to interpret the relationship between the predictor variables and the outcome variable. In this case, the outcome variable is the term deposit subscription by a member and the predictor variables are age, education, default, housing, loan, month, duration, job, marital status etc.

Here are some possible interpretations of the coefficients:

age coefficient: 0.010726302257827423 For a one-year increase in age, the log odds of a customer subscribing to a term deposit increase by 0.010726302257827423.

education coefficient: 0.06850397218006317 if contact has better education, the log odds of a customer subscribing to a term deposit increase by 0.06850397218006317.

default coefficient: -0.9633732418676771 For a contact who is goint to default, the log odds of a customer subscribing to a term deposit decrease by 0.9633732418676771.

housing coefficient: -0.9633732418676771 For a contact who has housing, the log odds of a customer subscribing to a term deposit decrease by 0.9633732418676771.

loan coefficient: -0.9633732418676771 For a contact who has loan, the log odds of a customer subscribing to a term deposit decrease by 0.9633732418676771.

month coefficient: 0.03691696044280908 For a unit increase in month, the log odds of a customer subscribing to a term deposit increase by 0.03691696044280908.

duration coefficient: 0.0038242986720552143 For a unit increase in duration, the log odds of a customer subscribing to a term deposit increase by 0.0038242986720552143.

14) Findings

Details findings are located [here (https://github.com/spalakollu/Bank_Products_Marketing/blob/main/BankMarketingFindings.pdf)](https://github.com/spalakollu/Bank_Products_Marketing/blob/main/BankMarketingFindings.pdf)

15) Next Steps and Recommendations

Further classification can be performed in a variety of ways.
For example, a bank can use a customer's past transaction history, demographics, and other data to make personalized product recommendations, such as credit cards or loans, that are likely to be of interest to the customer. These recommendations can be made through email campaigns, in-app notifications, or through the bank's website or mobile app. Additionally, AI/ML can also be used to predict which customers are most likely to churn, so the bank can proactively reach out to those customers to try to retain them.

1. Use the findings to target marketing efforts to customers who are more likely to open a term deposit account. This can be accomplished by building machine learning models that can predict the likelihood of a customer opening a term deposit account based on their characteristics.
2. Use the findings to identify the most effective marketing strategies and channels. For example, analysis shows that customers who are contacted by phone are more likely to open a term deposit account, then phone marketing should be prioritized.
3. Identify segments of customers that are most likely to open term deposit accounts and develop tailored marketing campaigns for those segments.
4. Use the findings to optimize the allocation of resources and effort in marketing campaigns. For example, if the analysis shows that certain demographics or geographic regions are more likely to open term deposit accounts, then marketing efforts should be focused on those demographics or regions.
5. Use the findings to improve the efficiency of the bank's marketing efforts by targeting the most promising customers. For example, if the analysis shows that customers who have a higher income are more likely to open a term deposit account, then marketing efforts should be focused on customers with higher incomes.
6. Use the findings to improve the customer experience by understanding their needs and preferences.
7. Continuously monitor the performance of the models and update the marketing strategies accordingly.

In [ ]:

In [ ]: