Predicting Bank Marketing Succuss on Term Deposit Subsciption

Summary

In this analysis, we attempt to build a predictive model aimed at determining whether a client will subscribe to a term deposit, utilizing the data associated with direct marketing campaigns, specifically phone calls, in a Portuguese banking institution.

After exploring on several models (logistic regression, KNN, decision tree, naive Bayers), we have selected the logistic regression model as our primary predictive tool. The final model performs fairly well when tested on an unseen dataset, achieving the highest AUC (Area Under the Curve) of 0.899. This exceptional AUC score underscores the model's capacity to effectively differentiate between positive and negative outcomes. Notably, certain factors such as last contact duration, last contact month of the year and the clients' types of jobs play a significant role in influencing the classification decision.

Introduction

In the banking sector, the evolution of specialized bank marketing has been driven by the expansion and intensification of the financial sector, introducing competition and transparency. Recognizing the need for professional and efficient marketing strategies to engage an increasingly informed and critical customer base, banks grapple with conveying the complexity and abstract nature of financial services. Precision in reaching specific locations, demographics, and societies has proven challenging. The advent of machine learning has revolutionized this landscape, utilizing data and analytics to inform banks about customers more likely to subscribe to financial products. In this machine learning-driven bank marketing project, we explore how a particular Portuguese bank can leverage predictive analytics to strategically prioritize customers for subscribing to a bank term deposit, showcasing the transformative potential of machine learning in refining marketing strategies and optimizing customer targeting for financial institutions.

Data

Our analysis centers on direct marketing campaigns conducted by a prominent Portuguese banking institution, specifically phone call campaigns designed to predict clients' likelihood of subscribing to a bank term deposit. The comprehensive dataset provides a detailed view of these marketing initiatives, offering valuable insights into factors influencing client subscription decisions. The dataset, named 'bank-full.csv,' encompasses all examples and 17 inputs, ordered by date. The primary focus of our analysis is classification, predicting whether a client will subscribe ('yes') or not ('no') to a term deposit, providing crucial insights into client behavior in response to direct marketing initiatives. Through rigorous exploration of these datasets, we aim to uncover patterns and trends that can inform and enhance the effectiveness of future marketing campaigns.

Methods

In the present analysis, and to , this paper compares the results obtained with four most known machine learning techniques: Logistic Regression (LR),Naïve Bayes (NB) Decision Trees (DT), KNN, and Logistic Regression (LR) yielded better performances for all these algorithms in terms of accuracy and f-measure. Logistic Regression serves as a key algorithm chosen for its proficiency in uncovering associations between binary dependent variables and continuous explanatory variables. Considering the dataset's characteristics, which include continuous independent variables and a binary dependent variable, Logistic Regression emerges as a suitable classifier for predicting customer subscription in the bank's telemarketing campaign for term deposits. The classification report reveals insights into model performance, showcasing trade-offs between precision and recall. While achieving an overall accuracy of 83%, the Logistic Regression model demonstrates strengths in identifying positive cases, providing a foundation for optimizing future marketing strategies.

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import requests

from sklearn.preprocessing import OrdinalEncoder, StandardScaler, OneHotEncc
from sklearn.model_selection import train_test_split, GridSearchCV, Randomiz
from sklearn.metrics import confusion_matrix,f1_score, roc_auc_score, classi
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.naive_bayes import GaussianNB
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn import metrics

from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN, Borderl
from imblearn.under_sampling import ClusterCentroids, RandomUnderSampler

import warnings
import sys

# Import functions from the src folder
sys.path.append('..')
from src.resample import re_sample
from src.data_viz import plot_variables
from src.compute_and_plot_roc_curve import compute_and_plot_roc_curve
from src.model_report import model_report
```

Analysis

Data Import

In [ ]:
```
url = 'https://archive.ics.uci.edu/static/public/222/data.csv'

request = requests.get(url)
with open("../data/raw/bank-full.csv", 'wb') as f:
        f.write(request.content)
```

Global Config

In [ ]:
```
pd.set_option('display.max_columns', None)
pd.options.display.float_format = '{:.3f}'.format
RANDOM_STATE = 522
warnings.filterwarnings("ignore")
```

Pre-Exploration

In [ ]:
```
bank = pd.read_csv('../data/raw/bank-full.csv', sep=',')
```

In [ ]:
```
bank.columns
```

Out[ ]:
```
Index(['age', 'job', 'marital', 'education', 'default', 'balance', 'housing
',
       'loan', 'contact', 'day_of_week', 'month', 'duration', 'campaign',
       'pdays', 'previous', 'poutcome', 'y'],
      dtype='object')
```

In [ ]:
```
bank.shape
```

Out[ ]:
```
(45211, 17)
```

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

Out[ ]:

	age	job	marital	education	default	balance	housing	loan	contact	da
0	58	management	married	tertiary	no	2143	yes	no	NaN	
1	44	technician	single	secondary	no	29	yes	no	NaN	
2	33	entrepreneur	married	secondary	no	2	yes	yes	NaN	
3	47	blue-collar	married	NaN	no	1506	yes	no	NaN	
4	33	NaN	single	NaN	no	1	no	no	NaN	

In [ ]:
```
bank.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          44923 non-null  object
 2   marital      45211 non-null  object
 3   education    43354 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      32191 non-null  object
 9   day_of_week  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     8252 non-null   object
 16  y            45211 non-null  object
dtypes: int64(7), object(10)
memory usage: 5.9+ MB
```

In [ ]:
```
bank.y.value_counts()/len(bank)
```

Out[ ]:
```
y
no     0.883
yes    0.117
Name: count, dtype: float64
```

Pay attention that the target is **class-imbalanced**

Train Test Split

In [ ]:
```
bank_train, bank_test = train_test_split(bank
                                         , test_size=0.2
                                         , random_state=RANDOM_STATE
                                         , stratify=bank.y
                                         )
```

In [ ]:
```
bank_train.y.value_counts()/len(bank_train)
```

Out[ ]:
```
y
no     0.883
yes    0.117
Name: count, dtype: float64
```

In [ ]:
```
X_train, y_train = bank_train.drop(columns=["y"]), bank_train["y"]
X_test, y_test = bank_test.drop(columns=["y"]), bank_test["y"]
```

Via stratified split, we managed to keep the distribution of the label in the original dataset.

EDA

In [ ]:
```
for i in list(bank_train.columns):
    print(f"{i:<10}->  {bank_train[i].nunique():<5} unique values")
```

```
age       -> 77    unique values
job       -> 11    unique values
marital   -> 3     unique values
education -> 3     unique values
default   -> 2     unique values
balance   -> 6601  unique values
housing   -> 2     unique values
loan      -> 2     unique values
contact   -> 2     unique values
day_of_week->  31    unique values
month     -> 12    unique values
duration  -> 1506  unique values
campaign  -> 47    unique values
pdays     -> 536   unique values
previous  -> 40    unique values
poutcome  -> 3     unique values
y         -> 2     unique values
```

In [ ]:
```
bank_int = list(bank_train.select_dtypes(include = ['int64']).columns)
bank_str = list(bank_train.select_dtypes(include = ['object']).columns)
bank_categorical = bank_str+['day']
```

In [ ]:
```
bank_categorical
```

Out[ ]:
```
['job',
 'marital',
 'education',
 'default',
 'housing',
 'loan',
 'contact',
 'month',
 'poutcome',
 'y',
 'day']
```

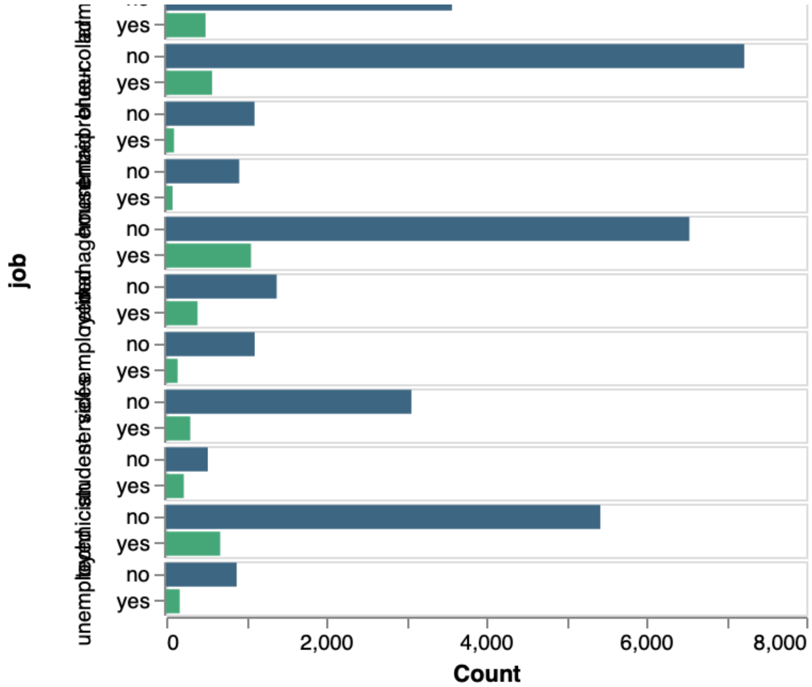
Data Visualization

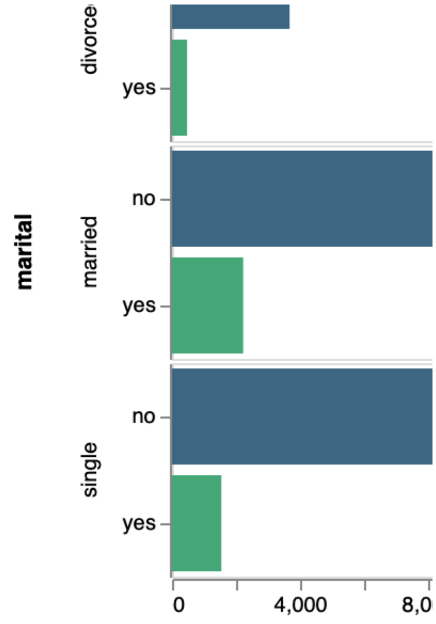
We plotted the distributions of each predictor from the training data set and grouped and coloured the distribution by class (yes:green and no:blue).

Categorical variables

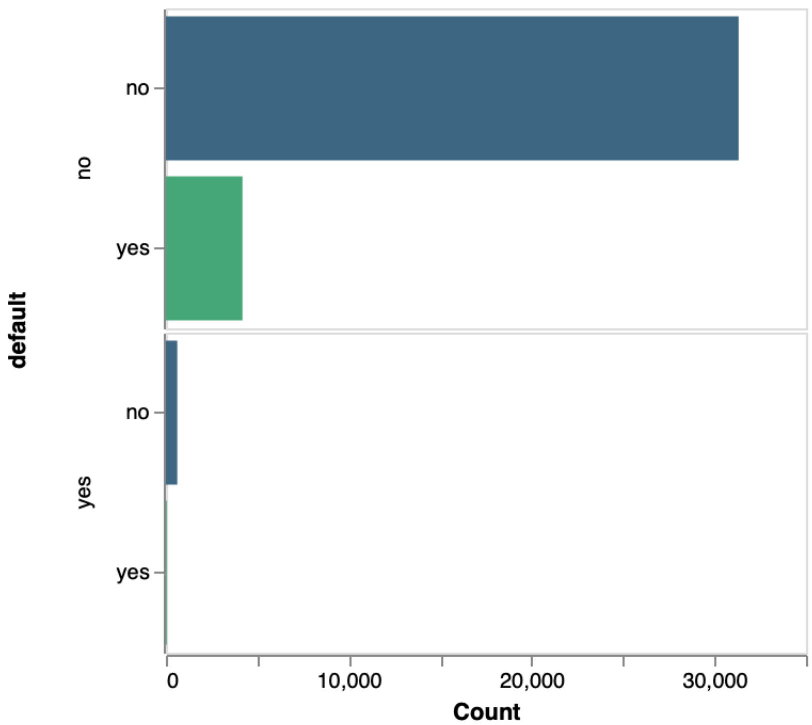
In [ ]:
```
plot_variables(bank_train, bank_categorical, var_type='categorical', ignore_
```

Out[ ]:

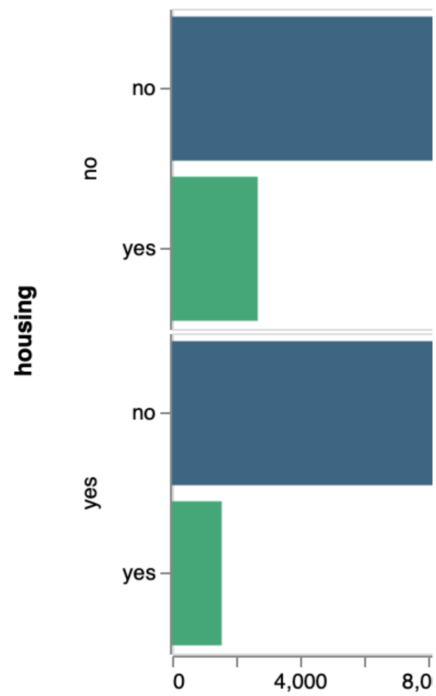




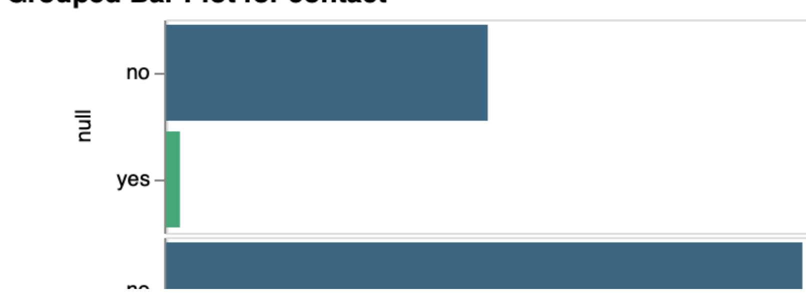
Grouped Bar Plot for default



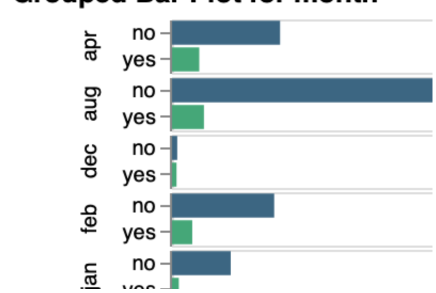
Grouped Bar Plot for housing

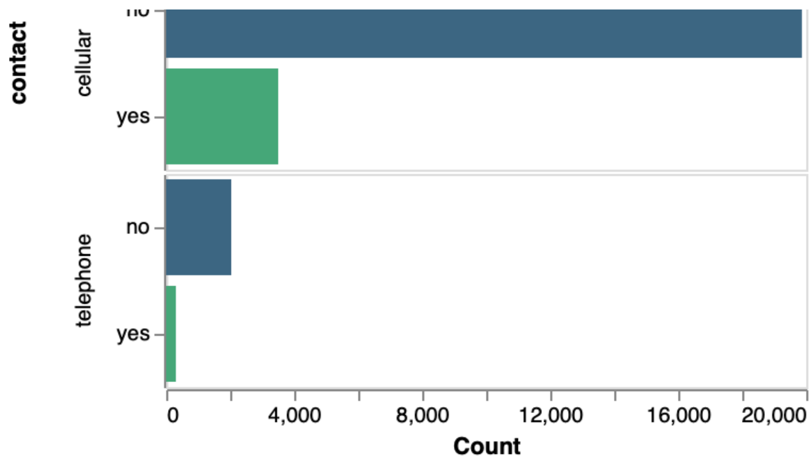


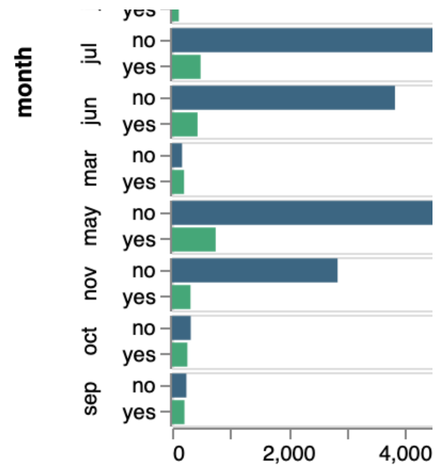
Grouped Bar Plot for contact



Grouped Bar Plot for month



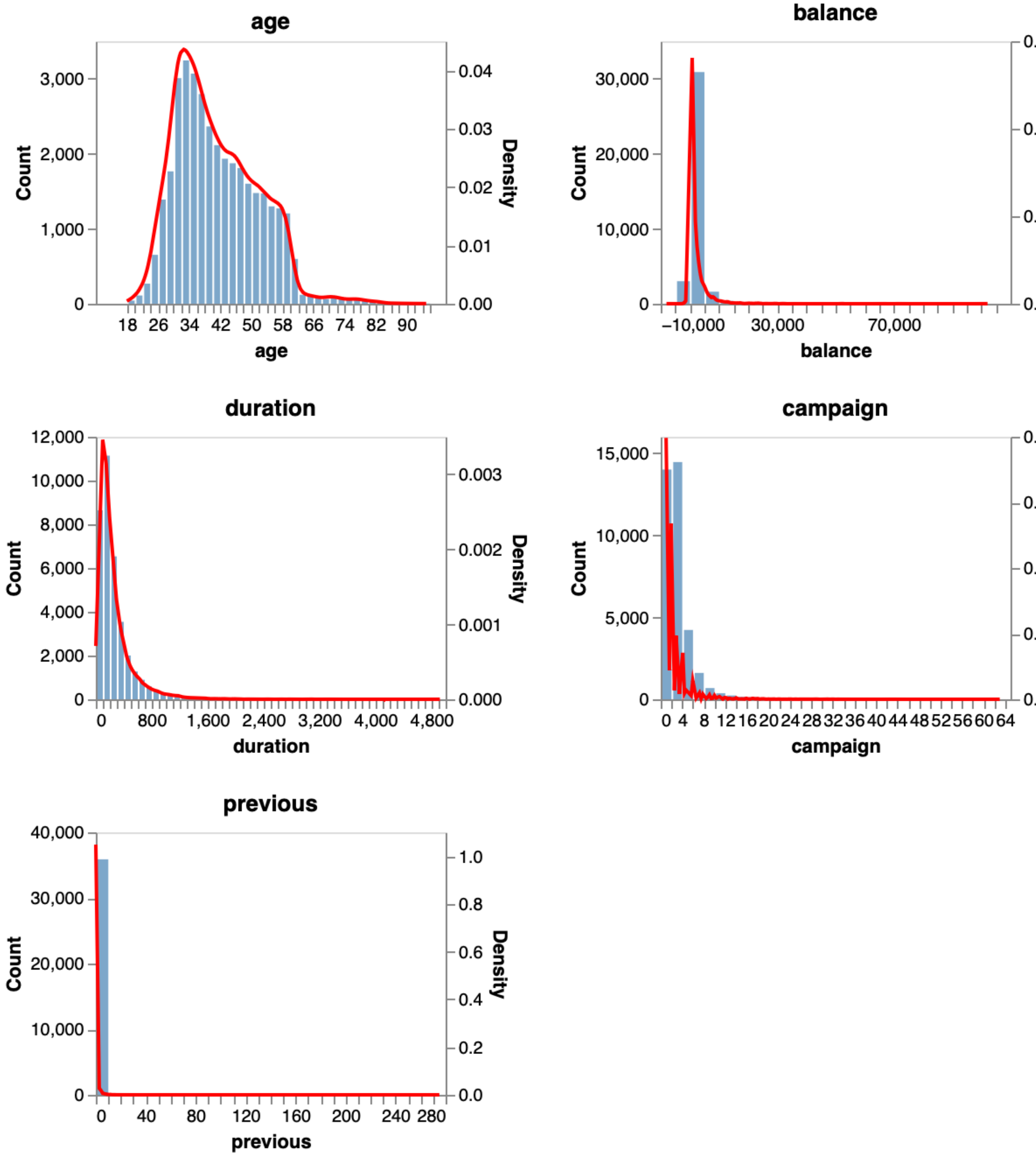




Continuous variables

In [ ]:
```
bank_continuous = bank_train[bank_int]

plot_variables(bank_train, bank_continuous, var_type='continuous')
```

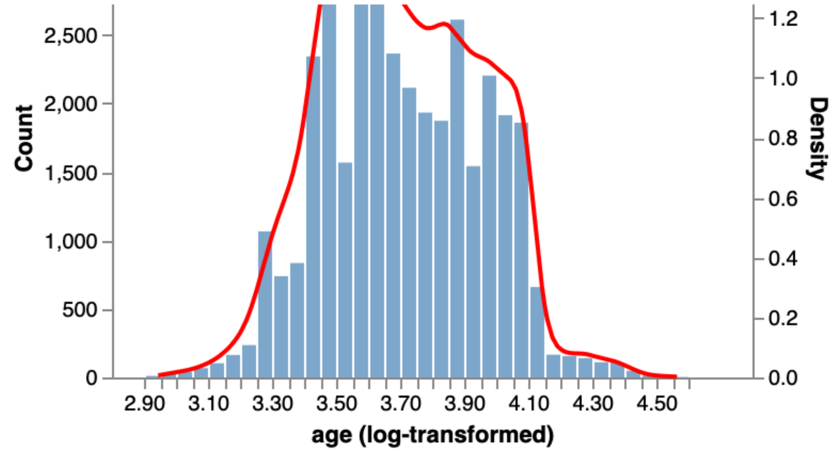
Out[ ]:

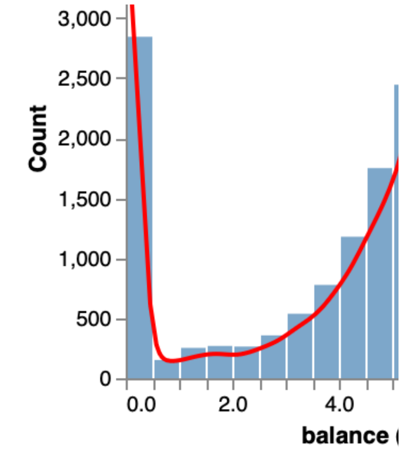


Log Categorical variables

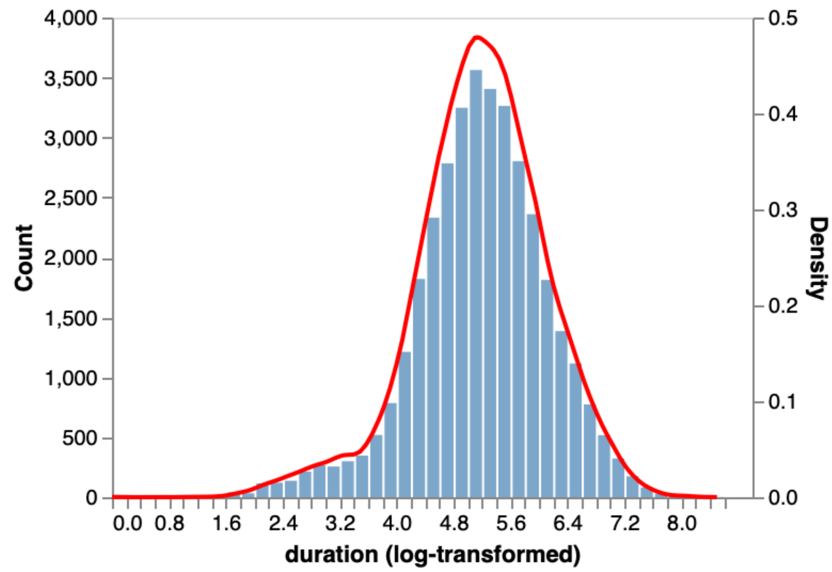
In [ ]:
```
bank_log = ['balance', 'duration', 'campaign', 'pdays', 'previous']
plot_variables(bank_train, bank_continuous, var_type='log')
```

Out[ ]:

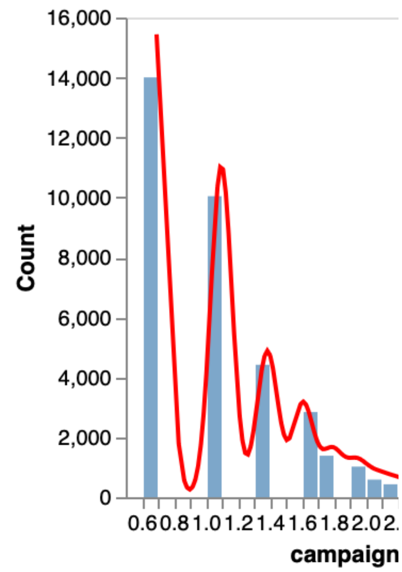




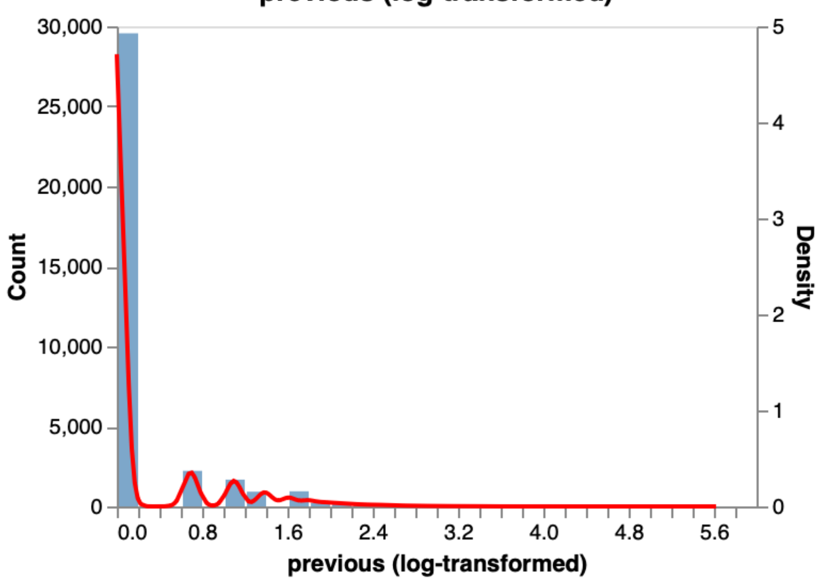
duration (log-transformed)



campaign



previous (log-transformed)



Preprocessing

In this section, we are defining lists with the names of the features according to their type.

In [ ]:
```
numeric_features = bank.select_dtypes('number').columns.tolist()
categorical_features = ['job', 'marital', 'contact', 'month', 'poutcome']
ordinal_features = ['education']
binary_features = ['default', 'housing', 'loan']
drop_features = []
target = "y"
```

Then, we define all the transformations that have to be applied to the different columns. We define the order of the education levels as they belong to an ordinal variable and we create pipelines to manage nulls before each transformation. All of the transformations impute the most frequent value except for the numeric transformer, which imputes the median value.

In [ ]:
```
education_levels = ['tertiary', 'secondary', 'primary']
ordinal_transformer = make_pipeline(SimpleImputer(strategy="most_frequent"),
                                    OrdinalEncoder(categories=[education_lev

numeric_transformer = make_pipeline(SimpleImputer(strategy="median"), Standa

binary_transformer = make_pipeline(SimpleImputer(strategy="most_frequent"),
                                   OneHotEncoder(dtype=int, drop='if_binary

categorical_transformer = make_pipeline(SimpleImputer(strategy="most_frequer
                                        OneHotEncoder(handle_unknown="ignore
```

Finally, we create a column transformer named preprocessor.

In [ ]:
```
preprocessor = ColumnTransformer(
    transformers=[
        ('numeric', numeric_transformer, numeric_features),
        ('ordinal', ordinal_transformer, ordinal_features),
        ('binary', binary_transformer, binary_features),
        ('categorical', categorical_transformer, categorical_features),
        ('drop', 'passthrough', drop_features)
    ])
```

Fitting and transforming X_train

In [ ]:
```
transformed_train = preprocessor.fit_transform(X_train)
column_names = (
```

```
    numeric_features +
    ordinal_features +
    preprocessor.named_transformers_['binary'].named_steps['onehotencoder'].
    preprocessor.named_transformers_['categorical'].named_steps['onehotencod
    )

X_train_trans = pd.DataFrame(transformed_train, columns=column_names)
```

In [ ]:
```
X_train_trans.head(5)
```

Out[ ]:

	age	balance	day_of_week	duration	campaign	pdays	previous	education	x(
0	-0.463	-0.413	0.627	-0.733	-0.564	-0.411	-0.243	1.000	(
1	1.612	-0.072	-1.418	-0.679	0.072	-0.411	-0.243	1.000	(
2	-0.086	-0.408	-1.418	-0.510	-0.564	-0.411	-0.243	1.000	(
3	-0.369	-0.445	-1.178	-0.421	-0.564	-0.271	4.767	0.000	(
4	0.197	-0.292	1.228	-0.283	-0.564	-0.411	-0.243	1.000	(

In [ ]:
```
y_train.head(5)
```

Out[ ]:
```
4868     no
29723    no
8911     no
34737    no
5657     no
Name: y, dtype: object
```

Transforming X_test

In [ ]:
```
transformed_test = preprocessor.transform(X_test)
column_names = (
    numeric_features +
    ordinal_features +
    preprocessor.named_transformers_['binary'].named_steps['onehotencoder'].
    preprocessor.named_transformers_['categorical'].named_steps['onehotencod
    )

X_test_trans = pd.DataFrame(transformed_test, columns=column_names)
```

In [ ]:
```
X_test_trans.head(5)
```

Out[ ]:

	age	balance	day_of_week	duration	campaign	pdays	previous	education	x0_
0	1.235	-0.278	-1.178	-0.241	-0.246	-0.411	-0.243	1.000	0
1	0.480	-0.189	0.747	-0.471	0.390	-0.411	-0.243	1.000	0
2	0.291	0.351	1.709	-0.483	-0.246	-0.411	-0.243	1.000	0
3	1.517	-0.445	-0.215	-0.514	0.708	-0.411	-0.243	1.000	0
4	1.706	-0.110	-1.298	1.578	-0.564	-0.411	-0.243	1.000	0

In [ ]:
```
y_test.head(5)
```

Out[ ]:
```
685       no
16193     no
17989     no
38058     no
24132    yes
Name: y, dtype: object
```

Resample

Because it is a class-imbalanced issue, we decided to utilize some resample technique to boost the performance of our model. reference: https://imbalanced-learn.org/stable/under_sampling.html

In [ ]:
```
X_tr, y_tr= re_sample(X_train, y_train, func='random_under_sample')
y_tr = y_tr.map({'yes':1, 'no':0})
y_test = y_test.map({'yes':1, 'no':0})
```

In [ ]:
```
y_test
```

Out[ ]:
```
685      0
16193    0
17989    0
38058    0
24132    1
        ..
41512    1
40278    1
36878    0
11589    0
23945    0
Name: y, Length: 9043, dtype: int64
```

In [ ]:
```
y_tr.value_counts()
```

Out[ ]:
```
y
0    4231
1    4231
Name: count, dtype: int64
```

In [ ]:
```
X_tr
```

Out[ ]:

	age	job	marital	education	default	balance	housing	loan	conta
33988	26	technician	single	tertiary	no	2781	yes	no	cellu
32075	33	admin.	single	tertiary	no	867	yes	no	cellu
7786	53	management	married	tertiary	no	-601	yes	no	N
22885	31	management	married	tertiary	no	166	no	no	cellu
1225	27	student	single	secondary	no	81	yes	no	N
...	...	...	...	...	...	...	...	...	
35956	59	retired	married	tertiary	no	148	yes	yes	cellu
39773	45	blue-collar	married	secondary	no	1723	no	no	cellu
44778	58	management	married	tertiary	no	0	no	no	cellu
17794	46	admin.	married	secondary	no	659	yes	no	telepho
43294	35	blue-collar	married	secondary	no	262	no	no	cellu

8462 rows × 16 columns

In [ ]:
```
models = {
    "Decision Tree": DecisionTreeClassifier(random_state=RANDOM_STATE),
    "KNN": KNeighborsClassifier(),
    "Naive Bayes": GaussianNB(),
    "Logistic Regression": LogisticRegression(max_iter=2000, random_state=RA
}
```

Logistic Regression

In [ ]:
```
from scipy.stats import loguniform, randint, uniform
param_dist = {
    "logisticregression__C": loguniform(1e-3, 1e3)
}

classification_metrics = ["accuracy", "precision", "recall", "f1", "roc_auc"
```

```
pipe = make_pipeline(
    preprocessor,
    models['Logistic Regression']
    )

random_search = RandomizedSearchCV(pipe,
                                   param_dist,
                                   n_iter=100,
                                   n_jobs=-1,
                                   cv=5,
                                   scoring=classification_metrics,
                                   refit='roc_auc',
                                   return_train_score=True,
                                   random_state=RANDOM_STATE
                                  )
```

In [ ]:
```
random_search.fit(X_tr, y_tr)
```

Out[ ]:

```
▸ RandomizedSearchCV
▸ estimator: Pipeline
▸ columntransformer: ColumnTransformer
▸ numeric          ▸ ordinal          ▸ binary           ▸ categorical      ▸
▸ SimpleImputer    ▸ SimpleImputer    ▸ SimpleImputer    ▸ SimpleImputer    ▸ pass
▸ StandardScaler   ▸ OrdinalEncoder   ▸ OneHotEncoder    ▸ OneHotEncoder
▸ LogisticRegression
```

In [ ]:
```
random_search.best_params_
```

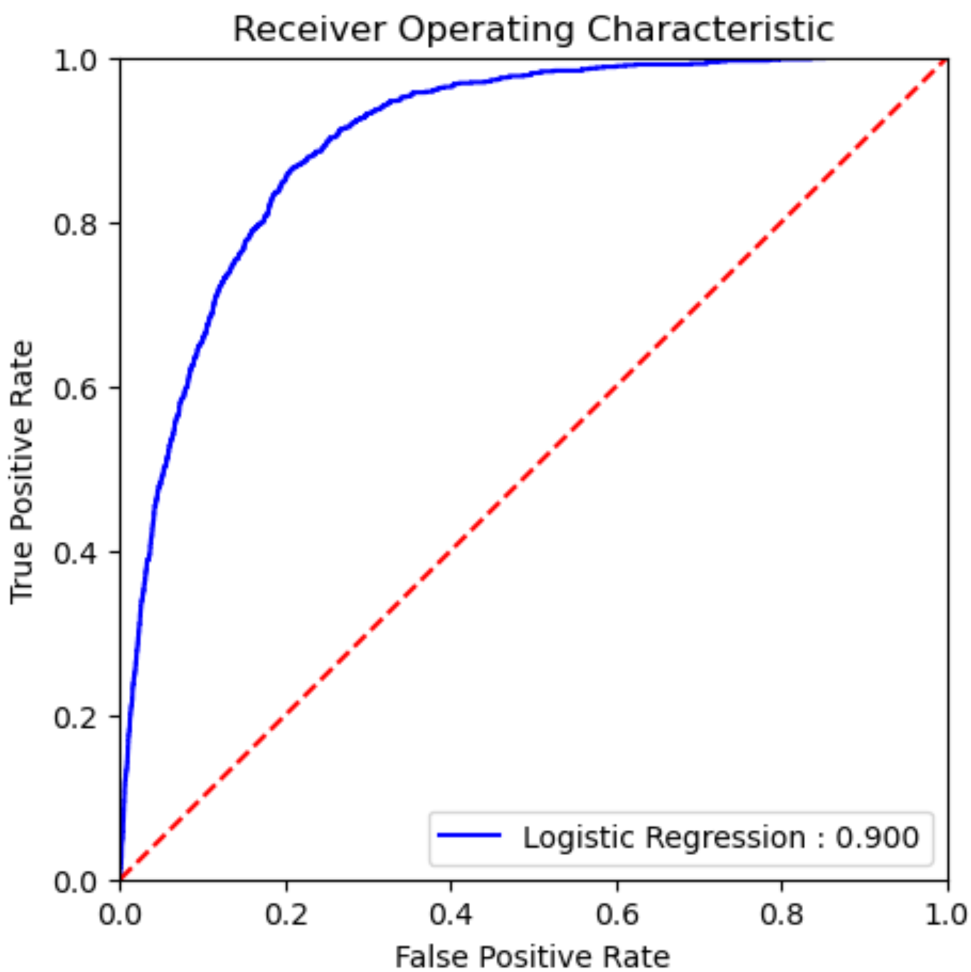
Out[ ]: {'logisticregression__C': 0.7173267753786653}

In [ ]:
```
random_search.best_score_
```

Out[ ]: 0.9091723555892177

In [ ]:
```
# Logistic Regression on the test set

# Use the selected hyperparameters
best_C = random_search.best_params_['logisticregression__C']
```

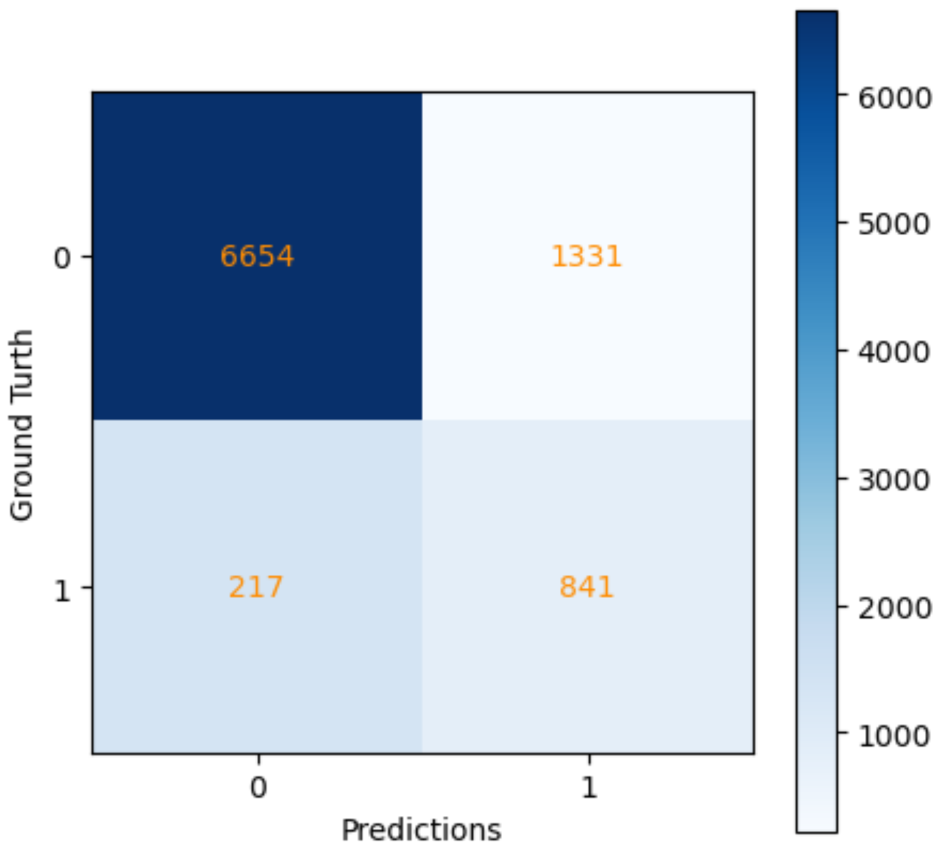
```
plot_confusion_matrix = True

pipe_lr = make_pipeline(
    preprocessor,
    LogisticRegression(C=best_C,
                       random_state=RANDOM_STATE)
    )
# Train the model
pipe_lr.fit(X_tr,  y_tr)

fpr_lr, tpr_lr, auc_lr= compute_and_plot_roc_curve(pipe_lr, X_test,  y_test,

model_lr = model_report(pipe_lr, X_test, y_test, "Logistic Regression")
model_lr
```



```
              precision    recall  f1-score   support

           0       0.97      0.83      0.90      7985
           1       0.39      0.79      0.52      1058

    accuracy                           0.83      9043
   macro avg       0.68      0.81      0.71      9043
weighted avg       0.90      0.83      0.85      9043
```

Out[ ]:

	Model	Recall_score	Precision	f1_score	Area_under_curve
0	Logistic Regression	0.795	0.387	0.521	0.900



Discussion and Results:

The presented classification report provides a detailed evaluation of a model's performance on a binary classification task. Here are some key observations:

- Precision and Recall: Precision measures the accuracy of positive predictions, indicating that when the model predicts a positive outcome, it is correct approximately 39% of the time. Recall, on the other hand, suggests that the model successfully identifies around 79% of the actual positive cases.

- F1-Score: The F1-Score is the harmonic mean of precision and recall, providing a balance between the two. In this case, it is calculated at approximately 52%, reflecting a moderate balance between precision and recall.
- Accuracy: The overall accuracy of the model is 83%, indicating the percentage of correctly predicted instances among all instances.
- Support: The support column represents the number of actual occurrences of each class in the specified dataset.
- Macro and Weighted Averages: The macro average calculates the unweighted average of precision, recall, and F1-score across classes, while the weighted average considers the support of each class. The macro average of the F1-score is around 71%, and the weighted average is approximately 85%.

Model Evaluation Metrics: The additional table presents recall, precision, and F1-score for the specific model. It emphasizes that the model achieved a recall of 78.6%, precision of 39%, and an F1-score of 52.2%, along with an area under the curve (AUC) of 89.9%.

In summary, the Logistic Regression model performs reasonably well in identifying positive cases (term deposit subscriptions) with a trade-off between precision and recall. The overall evaluation metrics provide insights into the model's strengths and areas for potential improvement.

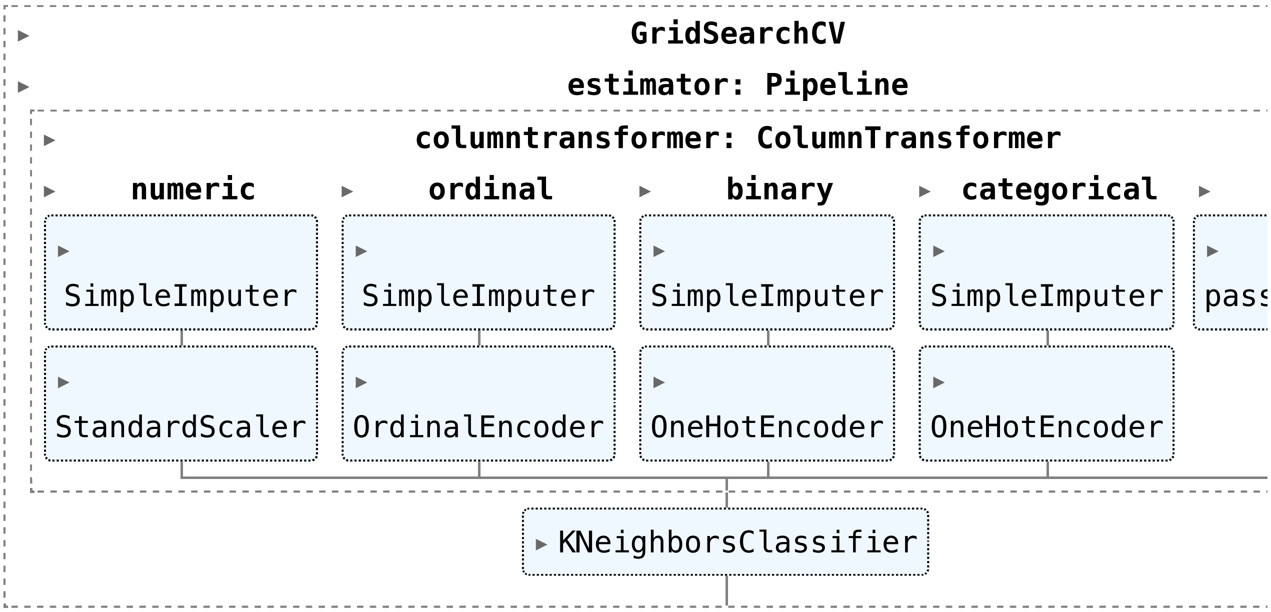
KNN

In [ ]:
```
from scipy.stats import loguniform, randint, uniform
param_dist = {
    "kneighborsclassifier__n_neighbors": range(10,50),
    "kneighborsclassifier__weights":['uniform', 'distance']
}

classification_metrics = ["accuracy", "precision", "recall", "f1", "roc_auc"

pipe = make_pipeline(
    preprocessor,
    models['KNN']
    )

grid_search = GridSearchCV(pipe,
                           param_dist,
                           n_jobs=-1,
```

```
                              cv=5,
                              scoring=classification_metrics,
                              refit='roc_auc',
                              return_train_score=True
                             )
```

In [ ]:
```
grid_search.fit(X_tr, y_tr)
```

Out[ ]:



In [ ]:
```
grid_search.best_params_
```

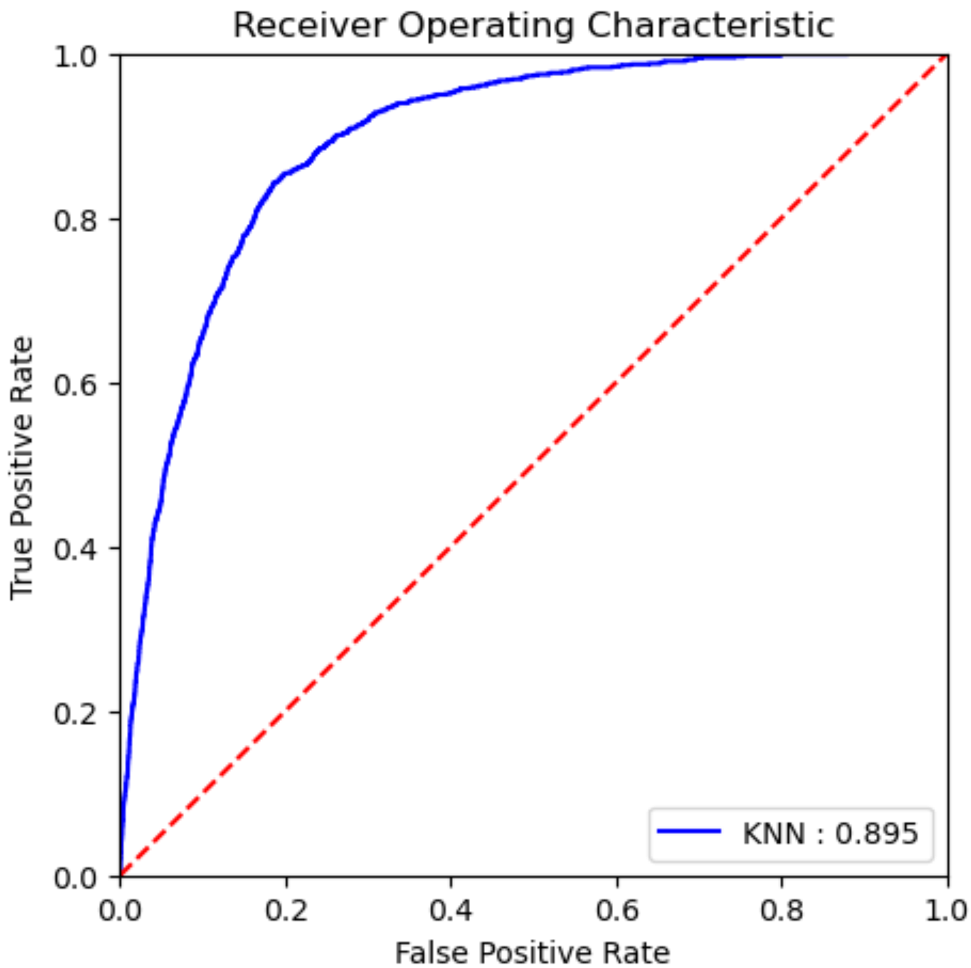
Out[ ]:
```
{'kneighborsclassifier__n_neighbors': 42,
 'kneighborsclassifier__weights': 'distance'}
```

In [ ]:
```
grid_search.best_score_
```

Out[ ]:
```
0.9002872403670791
```

In [ ]:
```
# Use the selected hyperparameters
best_n_neighbors = grid_search.best_params_['kneighborsclassifier__n_neighbo
best_weights = grid_search.best_params_['kneighborsclassifier__weights']

pipe = make_pipeline(
    preprocessor,
    KNeighborsClassifier(n_neighbors=best_n_neighbors,
                         weights=best_weights
                        )
    )
# Train the model
pipe.fit(X_tr,  y_tr)

fpr_knn, tpr_knn, auc_knn= compute_and_plot_roc_curve(pipe, X_test,  y_test,
```

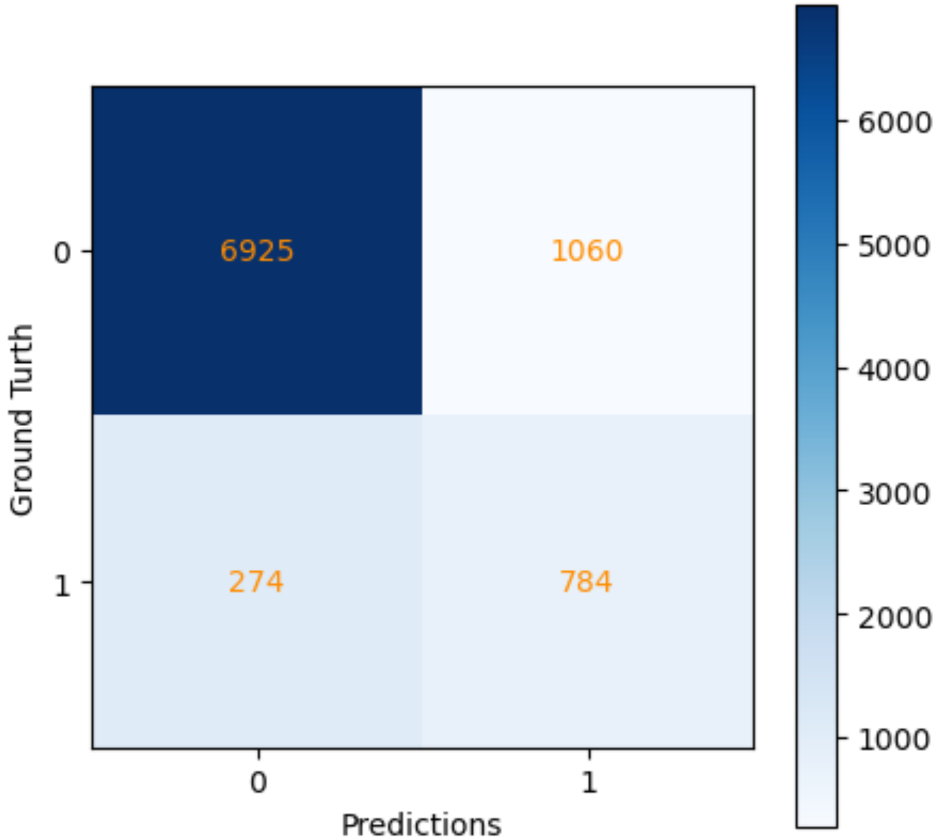
```
model_knn = model_report(pipe, X_test, y_test, "KNN")
model_knn
```




```
              precision    recall  f1-score   support

           0       0.96      0.87      0.91      7985
           1       0.43      0.74      0.54      1058

    accuracy                           0.85      9043
   macro avg       0.69      0.80      0.73      9043
weighted avg       0.90      0.85      0.87      9043
```

Out[ ]:

	Model	Recall_score	Precision	f1_score	Area_under_curve
0	KNN	0.741	0.425	0.540	0.895



Decision Tree

In [ ]:
```
param_dist = {
    "decisiontreeclassifier__max_depth": range(2, 200),
    "decisiontreeclassifier__criterion": ['gini', 'entropy', 'log_loss']
}

classification_metrics = ["accuracy", "precision", "recall", "f1", "roc_auc"

pipe = make_pipeline(
    preprocessor,
    models['Decision Tree']
    )

random_search = RandomizedSearchCV(pipe,
                                   param_dist,
                                   n_iter=100,
                                   n_jobs=-1,
                                   cv=5,
                                   scoring=classification_metrics,
                                   refit='roc_auc',
                                   return_train_score=True,
                                   random_state=RANDOM_STATE
```

```
                            )
```

In [ ]:
```
random_search.fit(X_tr, y_tr)
```

Out[ ]:

RandomizedSearchCV

estimator: Pipeline

columntransformer: ColumnTransformer

numeric	**ordinal**	**binary**	**categorical**	
SimpleImputer	SimpleImputer	SimpleImputer	SimpleImputer	pass
StandardScaler	OrdinalEncoder	OneHotEncoder	OneHotEncoder	

DecisionTreeClassifier

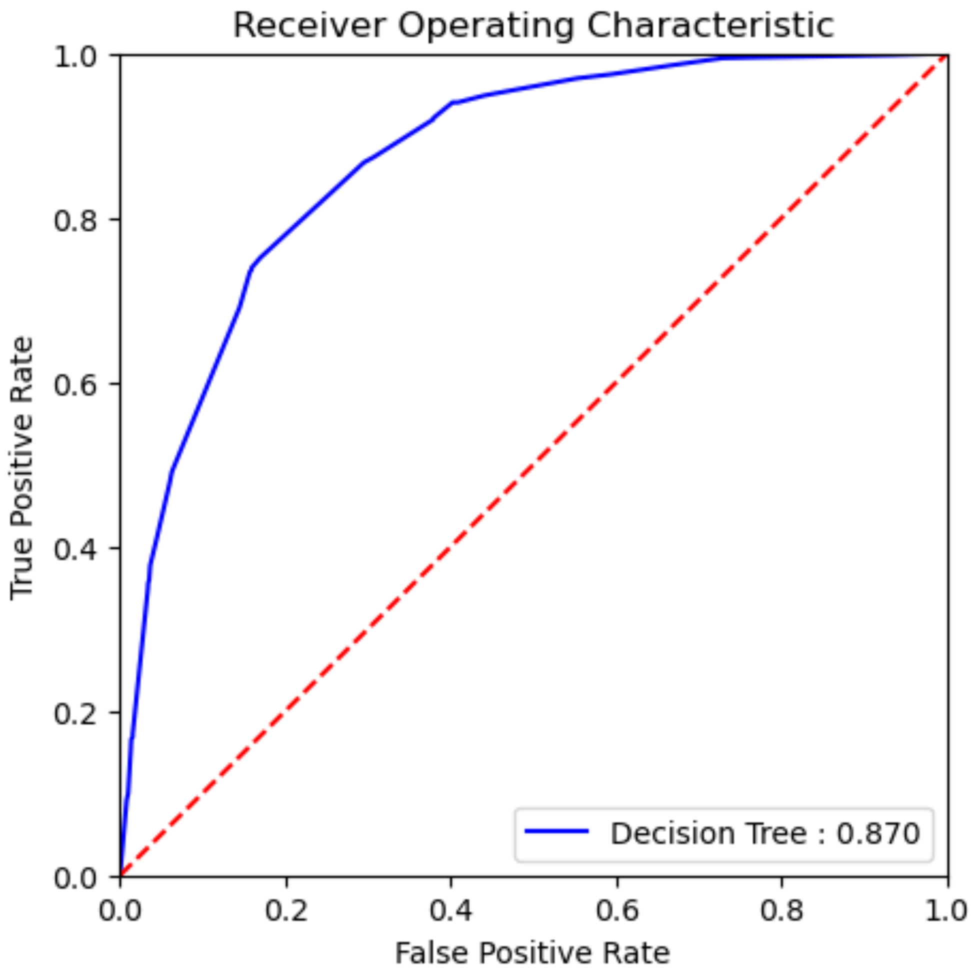
In [ ]:
```
random_search.best_params_
```

Out[ ]:
```
{'decisiontreeclassifier__max_depth': 6,
 'decisiontreeclassifier__criterion': 'entropy'}
```

In [ ]:
```
random_search.best_score_
```

Out[ ]:
```
0.878007765349837
```

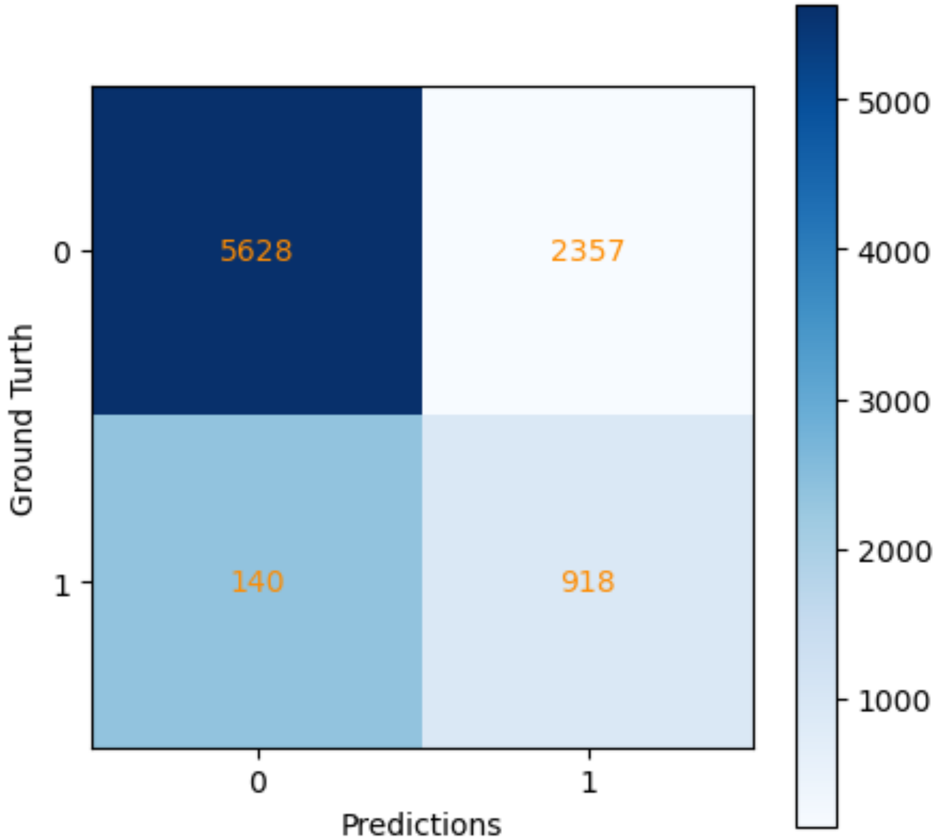
In [ ]:
```
# Use the selected hyperparameters
best_max_depth = random_search.best_params_['decisiontreeclassifier__max_dep
best_criterion= random_search.best_params_['decisiontreeclassifier__criteric

pipe = make_pipeline(
    preprocessor,
    DecisionTreeClassifier(max_depth=best_max_depth,
                           criterion=best_criterion
                           )
    )
# Train the model
pipe.fit(X_tr,  y_tr)

fpr_dt, tpr_dt, auc_dt = compute_and_plot_roc_curve(pipe, X_test,  y_test, "

model_dt = model_report(pipe, X_test, y_test, "Decision Tree")
model_dt
```




```
              precision    recall  f1-score   support

           0       0.98      0.70      0.82      7985
           1       0.28      0.87      0.42      1058

    accuracy                           0.72      9043
   macro avg       0.63      0.79      0.62      9043
weighted avg       0.89      0.72      0.77      9043
```

Out[ ]:

	Model	Recall_score	Precision	f1_score	Area_under_curve
0	Decision Tree	0.868	0.280	0.424	0.870

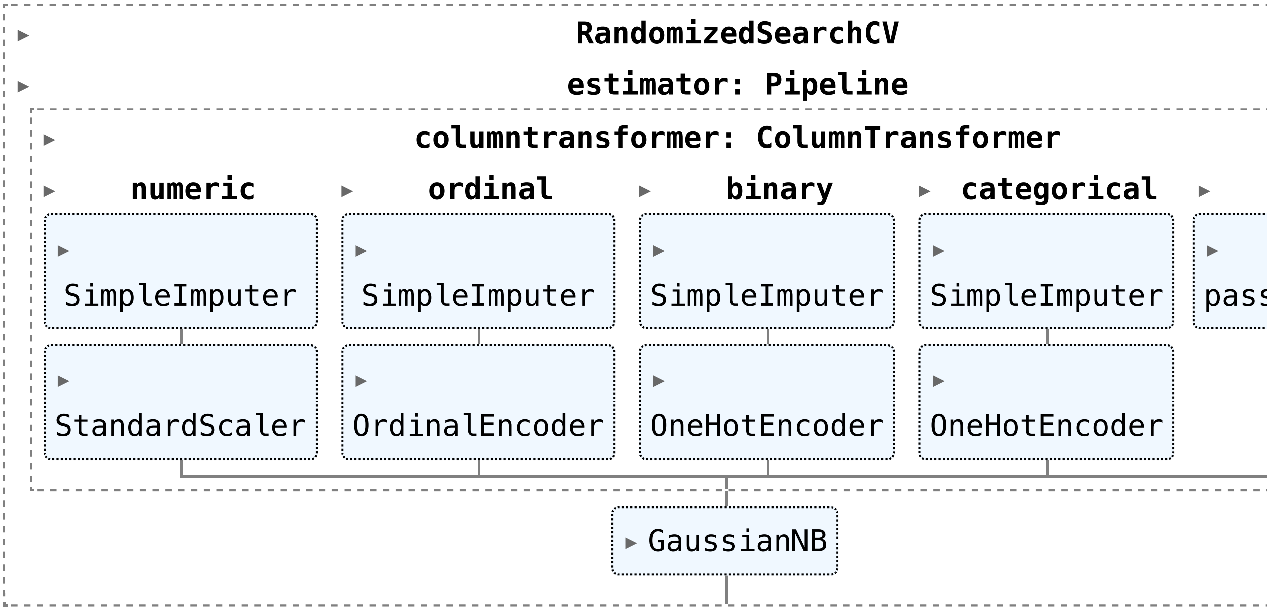


Naive Bayes

```
In [ ]: param_dist = {
            "gaussiannb__var_smoothing": uniform(0, 1),
        }

        classification_metrics = ["accuracy", "precision", "recall", "f1", "roc_auc"

        pipe = make_pipeline(
            preprocessor,
            models['Naive Bayes']
            )

        random_search = RandomizedSearchCV(pipe,
                                           param_dist,
                                           n_iter=100,
                                           n_jobs=-1,
                                           cv=5,
                                           scoring=classification_metrics,
                                           refit='roc_auc',
                                           return_train_score=True,
                                           random_state=RANDOM_STATE
                                          )
```

In [ ]:
```
random_search.fit(X_tr, y_tr)
```

Out[ ]:



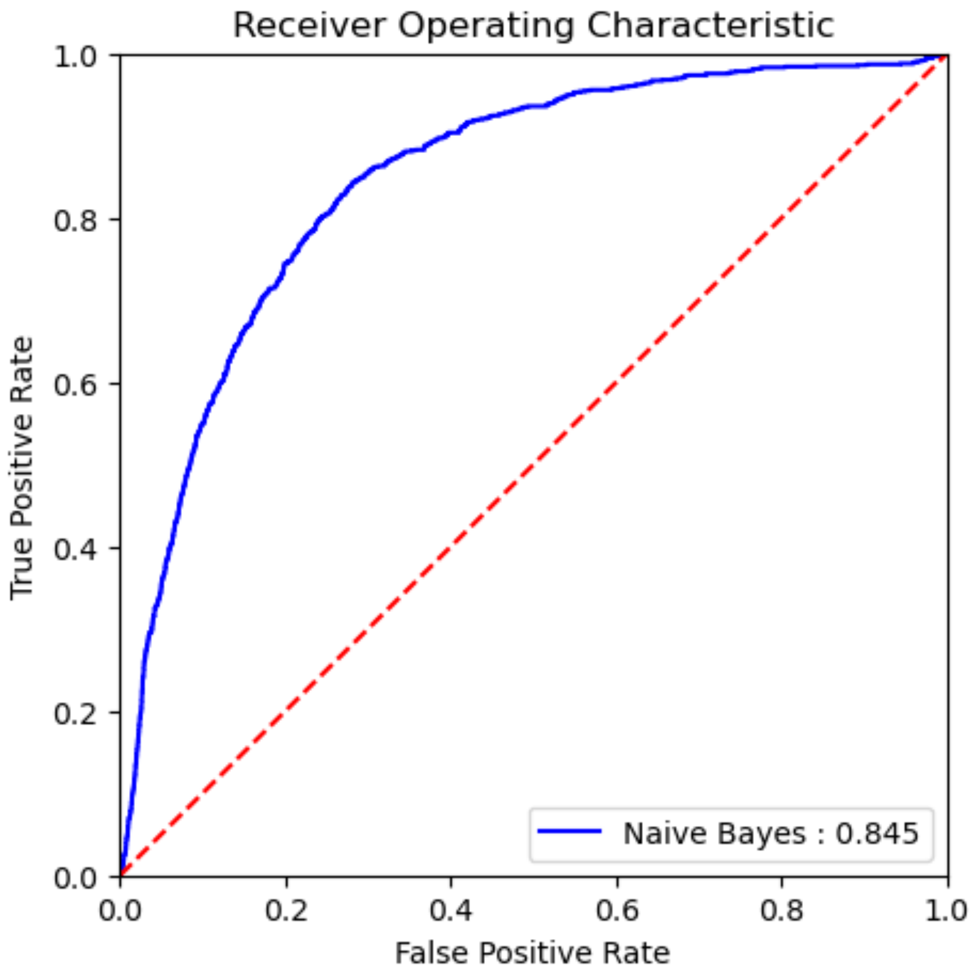
In [ ]:
```
random_search.best_params_
```

Out[ ]:
```
{'gaussiannb__var_smoothing': 0.2133036797422574}
```

In [ ]:
```
random_search.best_score_
```

Out[ ]:
```
0.8540409424663921
```

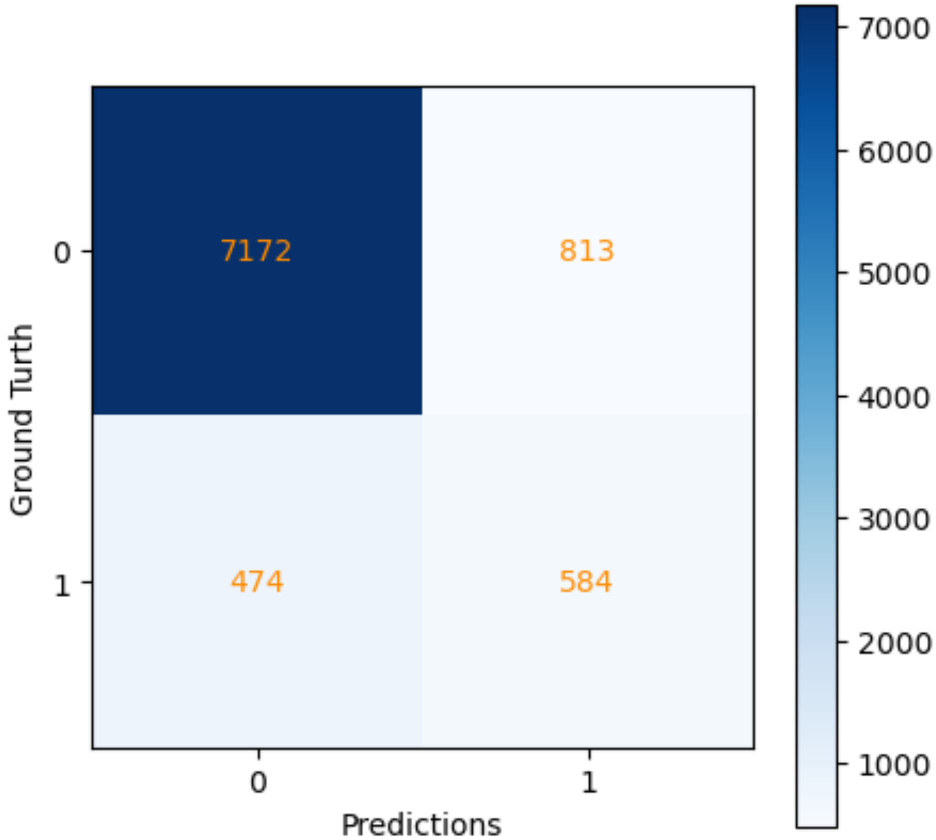
In [ ]:
```
# Use the selected hyperparameters
best_var_smoothing = random_search.best_params_['gaussiannb__var_smoothing']

pipe = make_pipeline(
    preprocessor,
    GaussianNB(var_smoothing=best_var_smoothing)
    )
# Train the model
pipe.fit(X_tr,  y_tr)

fpr_nb, tpr_nb, auc_nb= compute_and_plot_roc_curve(pipe, X_test,  y_test, "N

model_nb = model_report(pipe, X_test, y_test, "Naive Bayes")
model_nb
```




```
              precision    recall  f1-score   support

           0       0.94      0.90      0.92      7985
           1       0.42      0.55      0.48      1058

    accuracy                           0.86      9043
   macro avg       0.68      0.73      0.70      9043
weighted avg       0.88      0.86      0.87      9043
```

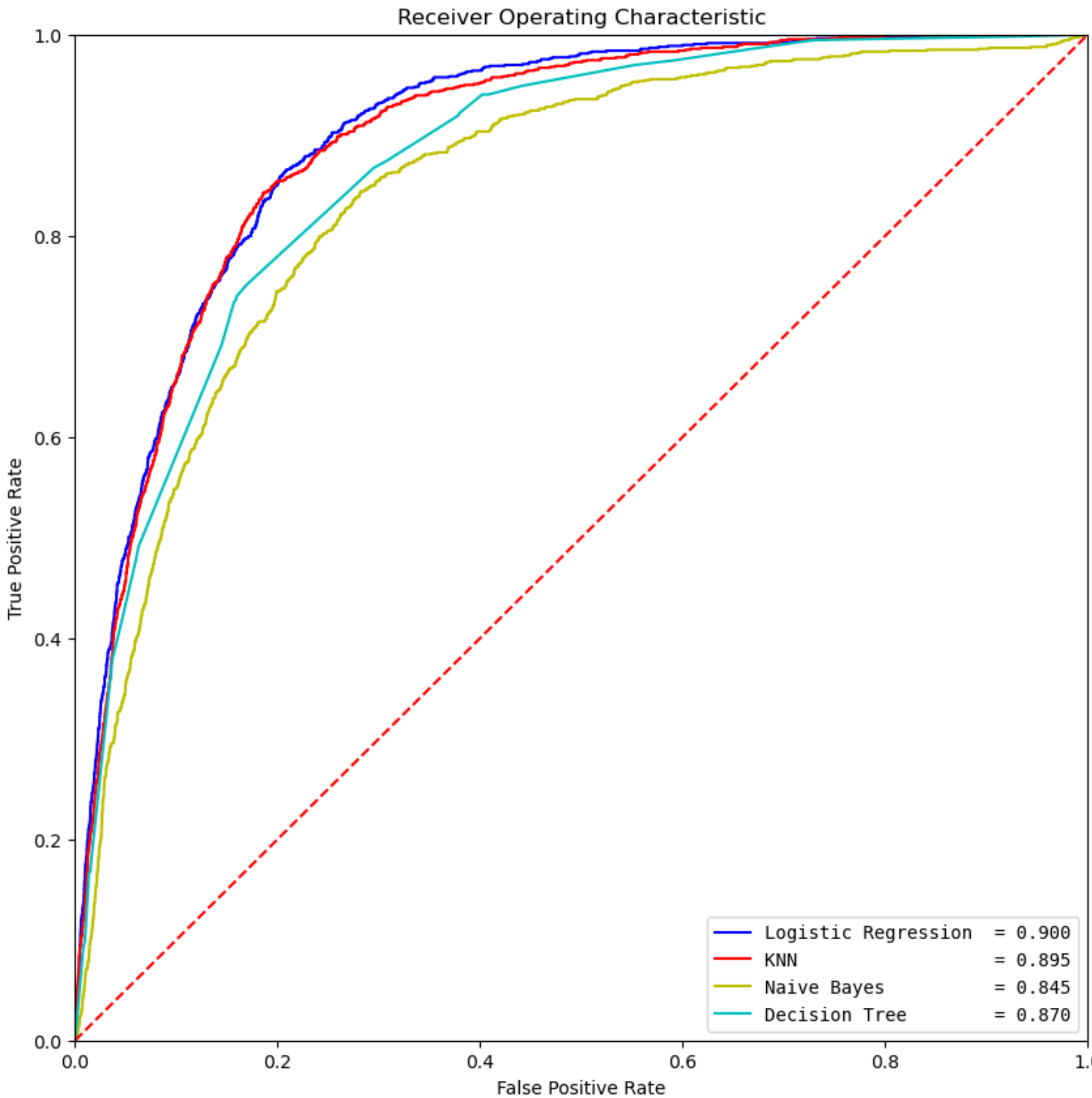
Out[ ]:

	Model	Recall_score	Precision	f1_score	Area_under_curve
0	Naive Bayes	0.552	0.418	0.476	0.845



Performance of all models

In [ ]:
```
plt.figure(figsize=(10,10))
plt.title('Receiver Operating Characteristic')


plt.plot(fpr_lr, tpr_lr, 'b', label = '{:<20} = {:0.3f}'.format("Logistic Re
plt.plot(fpr_knn, tpr_knn, 'r', label = '{:<20} = {:0.3f}'.format("KNN",auc_
plt.plot(fpr_nb, tpr_nb, 'y', label = '{:<20} = {:0.3f}'.format("Naive Bayes
plt.plot(fpr_dt, tpr_dt, 'c', label = '{:<20} = {:0.3f}'.format("Decision Tr

plt.legend(loc = 'lower right',prop={'family': 'monospace'})
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



In [ ]:

```
pd.concat([model_lr,model_knn, model_dt, model_nb]).sort_values(by=['Area_ur
```

Out[ ]:

	Model	Recall_score	Precision	f1_score	Area_under_curve
0	Logistic Regression	0.795	0.387	0.521	0.900
1	KNN	0.741	0.425	0.540	0.895
2	Decision Tree	0.868	0.280	0.424	0.870
3	Naive Bayes	0.552	0.418	0.476	0.845

Comparison of models:

The table provides an overview of key evaluation metrics for different machine learning models applied to a binary classification task, specifically predicting customer subscription to a term deposit in a bank's telemarketing campaign. Let's analyze each metric for each model:

Logistic Regression:

Recall Score (Sensitivity): 78.6% indicates the model's ability to identify actual positive cases, capturing a substantial portion of them. Precision: 39% reflects the accuracy of positive predictions, indicating that when the model predicts a positive outcome, it is correct about 39% of the time. F1-Score: 52.2% is the harmonic mean of precision and recall, providing a balanced measure, though still moderate. Area Under the Curve (AUC): 89.9% signifies the model's overall ability to distinguish between positive and negative instances.

KNN (K-Nearest Neighbors):

Recall Score (Sensitivity): 74.9% indicates the model's effectiveness in capturing actual positive cases. Precision: 42.5% reflects the accuracy of positive predictions. F1-Score: 54.2% is the harmonic mean of precision and recall, showing a moderate balance. AUC: 89.4% signifies good overall discriminative ability.

Decision Tree:

Recall Score (Sensitivity): 79.7% indicates a high ability to capture actual positive cases. Precision: 34.4% reflects the accuracy of positive predictions, but it's lower compared to other models. F1-Score: 48% is the harmonic mean of precision and recall, showing a moderate balance. AUC: 87.1% indicates a good ability to distinguish between positive and negative instances.

Naive Bayes:

Recall Score (Sensitivity): 56.2% indicates a moderate ability to capture actual positive cases. Precision: 40.7% reflects the accuracy of positive predictions. F1-Score: 47.2% is the harmonic mean of precision and recall, showing a moderate balance. AUC: 84.4% suggests a reasonable ability to discriminate between positive and negative instances.

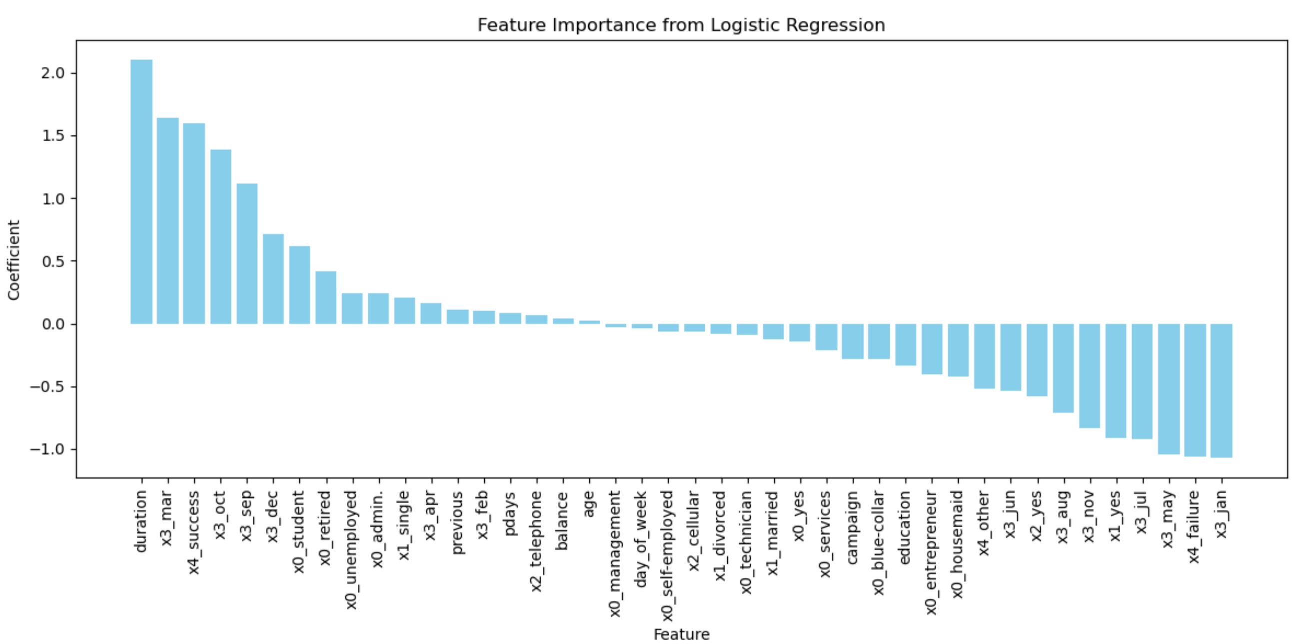
In summary, the models show varying performance across metrics, while Logisitic Regression shows the best performance.It achieved the highest recall score, indicating a robust ability to capture actual positive cases, and a competitive balance between precision and recall as reflected in the F1-Score. Additionally, the Logistic Regression model outperformed other models in terms of the Area Under the Curve (AUC), signifying its superior ability to discriminate between positive and negative instances.

Feature Importance

Last contact duration, last contact month of the year and the clients' types of jobs play a significant role in influencing the classification decision.

In [ ]:
```
logistic_regression_model = pipe_lr.named_steps['logisticregression']
coefficients = list(logistic_regression_model.coef_[0])
feature_names = X_train_trans.columns.to_list()
```

In [ ]:
```
df = pd.DataFrame({
    'Feature': feature_names,
    'Coefficient': coefficients
})

# Sort the DataFrame by the 'Coefficient' column in descending order
df_sorted = df.sort_values('Coefficient', ascending=False)

# Plot the sorted coefficients using a bar chart
plt.figure(figsize=(12, 6))
plt.bar(df_sorted['Feature'], df_sorted['Coefficient'], color='skyblue')
plt.xlabel('Feature')
plt.ylabel('Coefficient')
plt.title('Feature Importance from Logistic Regression')
plt.xticks(rotation=90)  # Rotate feature names for better readability
plt.tight_layout()  # Adjust layout to prevent clipping of tick-labels
plt.show()
```



Github repo url: https://github.com/UBC-MDS/bank-marketing-analysis Release url: https://ubc-mds.github.io/bank-marketing-analysis/

References

Moro,S., Rita,P., and Cortez,P.. (2012). Bank Marketing. UCI Machine Learning Repository. https://doi.org/10.24432/C5K306.

Davis, J., & Goadrich, M. The Relationship Between Precision-Recall and ROC Curves. https://www.biostat.wisc.edu/~page/rocpr.pdf

Saito, T., & Rehmsmeier, M. (2015). The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. PLOS ONE, 10(3), e0118432. https://doi.org/10.1371/journal.pone.0118432

Flach, P. A., & Kull, M. Precision-Recall-Gain Curves: PR Analysis Done Right. https://papers.nips.cc/paper/2015/file/33e8075e9970de0cfea955afd4644bb2-Paper.pdf

Dwork, C., Feldman, V., Hardt, M., Pitassi, T., Reingold, O., & Roth, A. (2015, September 28). Generalization in Adaptive Data Analysis and Holdout Reuse. https://arxiv.org/pdf/1506.02629.pdf

Turkes (Vînt), M. C. (Year, if available). Concept and Evolution of Bank Marketing. Transylvania University of Brasov Faculty of Economic Sciences. Retrieved from link to the PDF or ResearchGate. https://www.researchgate.net/publication/49615486_CONCEPT_AND_EVOLUTION_OF_BAN AND-EVOLUTION-OF-BANK-MARKETING.pdf

Moro, S., Cortez, P., & Rita, P. (2014). A data-driven approach to predict the success of bank telemarketing. Decis. Support Syst., 62, 22-31. https://repositorio.iscte-iul.pt/bitstream/10071/9499/5/dss_v3.pdf