1.0 Introduction

Term deposits are one of the most important sources of revenue for a bank. A term deposit is an investment in cash placed with a financial institution. The money is lent at an agreed rate of interest for a specific period. The bank has a number of strategies to sell term deposits to the client, including email marketing, print advertising, telephonic marketing and digital marketing. A telephonic marketing campaign is still quite effective, but it costs a lot of money, because telephone call center agents must be hired to actually carry out these campaigns. Therefore, it is important to know those customers who are most likely to convert and subscribe to the term deposit so when the telephone call comes, they can be targeted specifically from that phone call.

The data was collected from direct marketing campaigns of a Portuguese banking institution which consist of 21 columns and 4,119 rows. Each row of the dataset corresponds to one customer, with each column containing different attributes such as age, type of job, marital status, education level, credit default status, housing loan, personal loan, previous campaign outcome and so on. The target feature 'deposit' represents if the client subscribed to a term deposit after a marketing campaign and has a value of 'yes' or 'no'. The data provides a rich source of information which can be used to understand the behavior and preference of a client, thus becoming an excellent data source for machine learning models to improve marketing strategy.

For this binary classification of term deposit subscriptions, six classification models were compared such as Logistic Regression, Decision Tree, Random Forest, XGBoost, LightGBM and CatBoost. These models were selected because they encompass different algorithmic methodologies that are suitable for treating the categorical target variable and include a range of capability to handle imbalanced datasets, and to find complex relations among features. Logistic Regression is one of the simplest models available for classification, it is intuitive and easy to implement whereas Decision Trees are easy to interpret and simple model. Random Forest is a type of ensemble method which improves the predictive power of a classifier by combining multiple decision trees and increases the predictive accuracy by managing the overfitting. XGBoost and LightGBM are gradient boosting frameworks which are fast and achieve efficient performance when dealing with complicated datasets. CatBoost, which is an algorithm explicitly designed to take into account categorical values and works well without any pre-processing.

All six models were implemented using the default parameters, with a random seed of 42 to ensure reproducibility. A comprehensive evaluation was conducted to assess the performance of each model based on accuracy scores, allowing for a fair comparison of their effectiveness in predicting term deposit subscriptions. The model with the highest accuracy score will be selected for critique. Then, this model will go through another round of improvement by Hyperparameter Tuning, refining its parameters to increase predictive performance and decreasing overfit on unseen data.

This report investigates the application of classification on bank-additional dataset, which aims to develop a machine learning model that can accurately classify the client will subscribe to a term deposit in a marketing campaign.

2.0 Solution Architecture

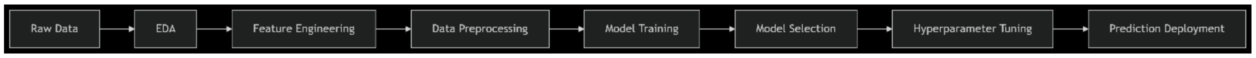
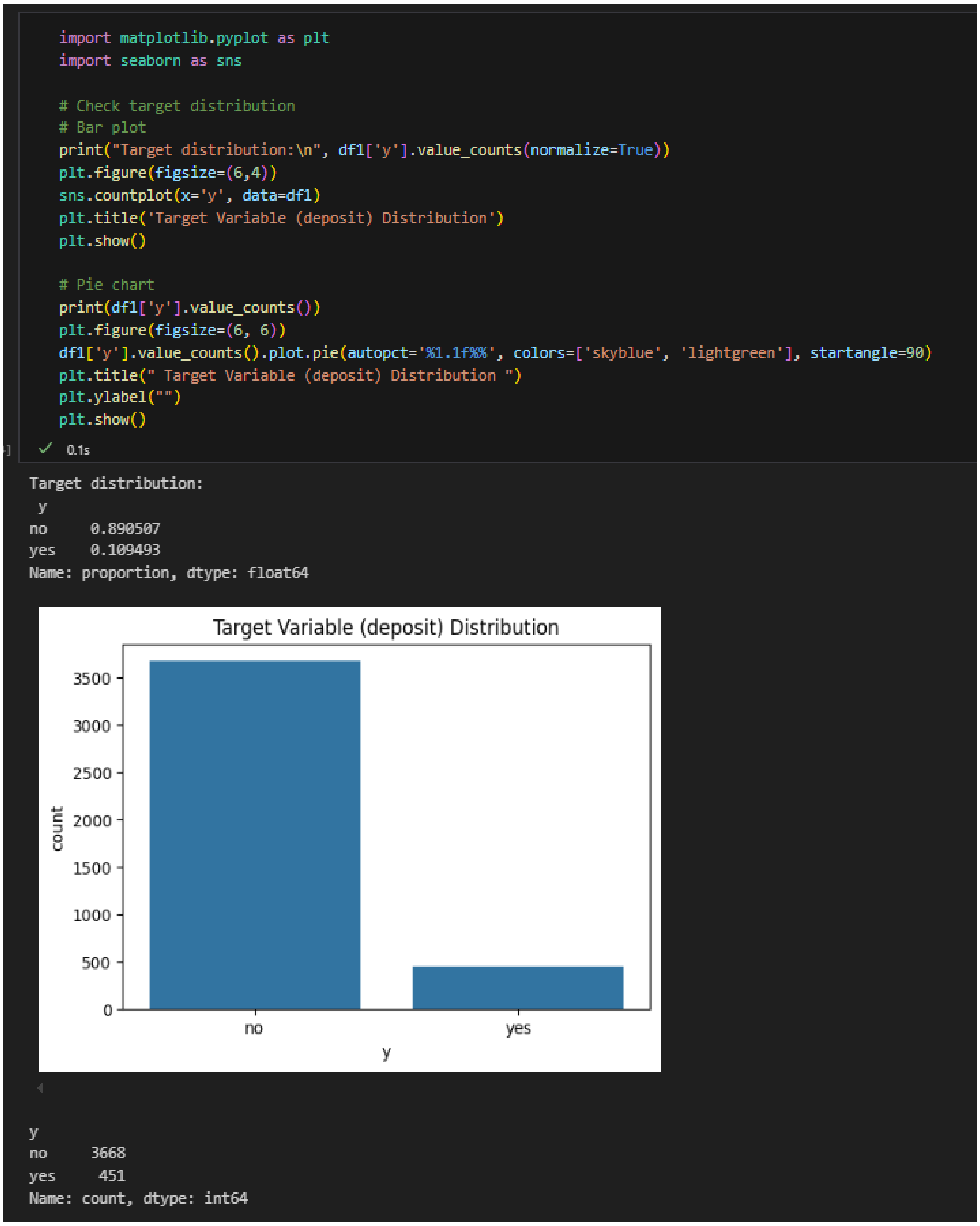


Figure 1: Workflow Diagram

The workflow of the predictive modeling methodology starts with importing the raw data which is the primary input in the entire analysis. Next is exploratory data analysis which is used to quick inspection of the data pattern and relation. Following is feature engineering. It is performed to generate new features or to remove a feature so that new information can be provided to the model for better predictions. Next is data preprocessing, where the raw data is preprocessed to make it fit for analysis and to improve the quality of the data. This includes missing value handling, categorical encoding and class imbalance handling. After the data is being preprocess, we proceed to the model training stage, where six different classification techniques are employed on the training data to learn patterns and relationships. After training, the best classification model is chosen and the hyperparameter tuning is executed to refine the parameters of the model for better classification performance. This is because we want to make the model generalize well on test data and not overfit. Finally, the optimal model is deployed for prediction which can help the bank to see the high-potential customers and allocate resources wisely in order to increase the term deposits subscription and refine their marketing strategy.

2.1 Exploratory Data Analysis

2.1.1 Distribution Plot of target variable

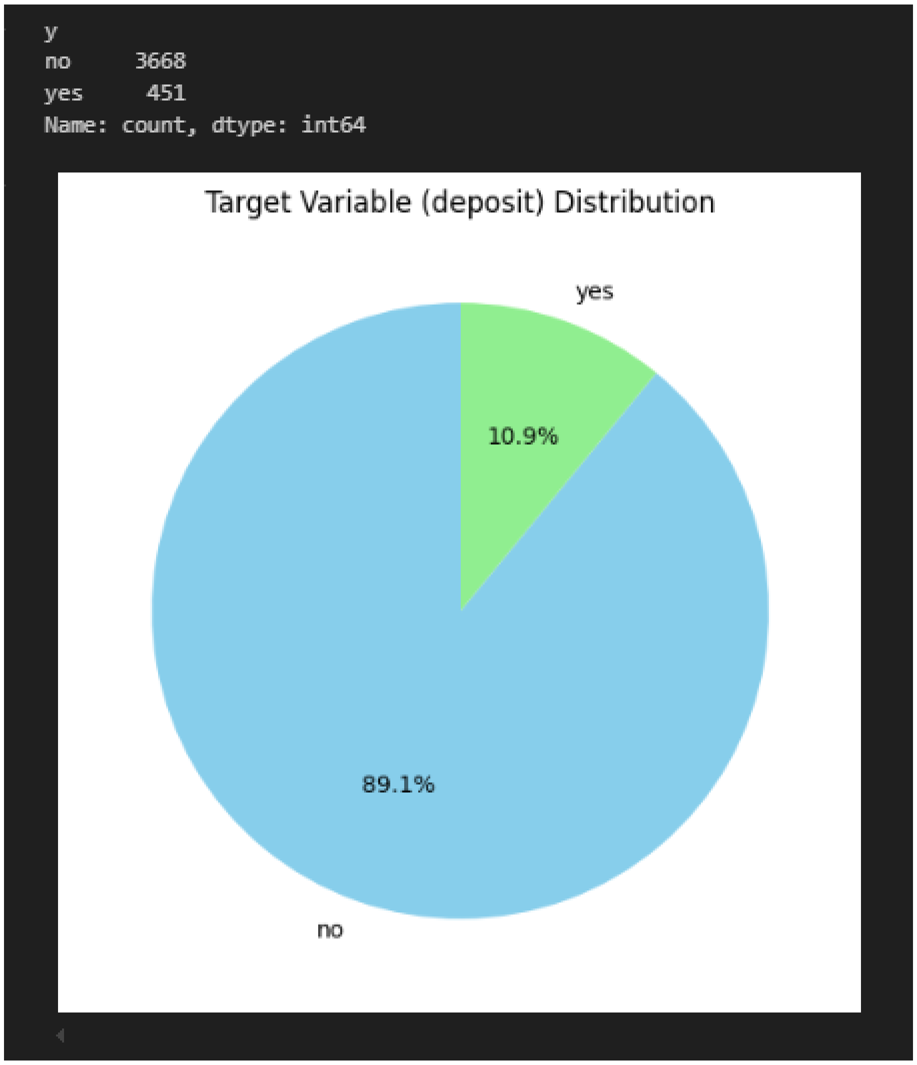


```
import matplotlib.pyplot as plt
import seaborn as sns

# Check target distribution
# Bar plot
print("Target distribution:\n", df1['y'].value_counts(normalize=True))
plt.figure(figsize=(6,4))
sns.countplot(x='y', data=df1)
plt.title('Target Variable (deposit) Distribution')
plt.show()

# Pie chart
print(df1['y'].value_counts())
plt.figure(figsize=(6, 6))
df1['y'].value_counts().plot.pie(autopct='%1.1f%%', colors=['skyblue', 'lightgreen'], startangle=90)
plt.title(" Target Variable (deposit) Distribution ")
plt.ylabel("")
plt.show()
```

✓ 0.1s

```
Target distribution:
 y
no     0.890507
yes    0.109493
Name: proportion, dtype: float64
```

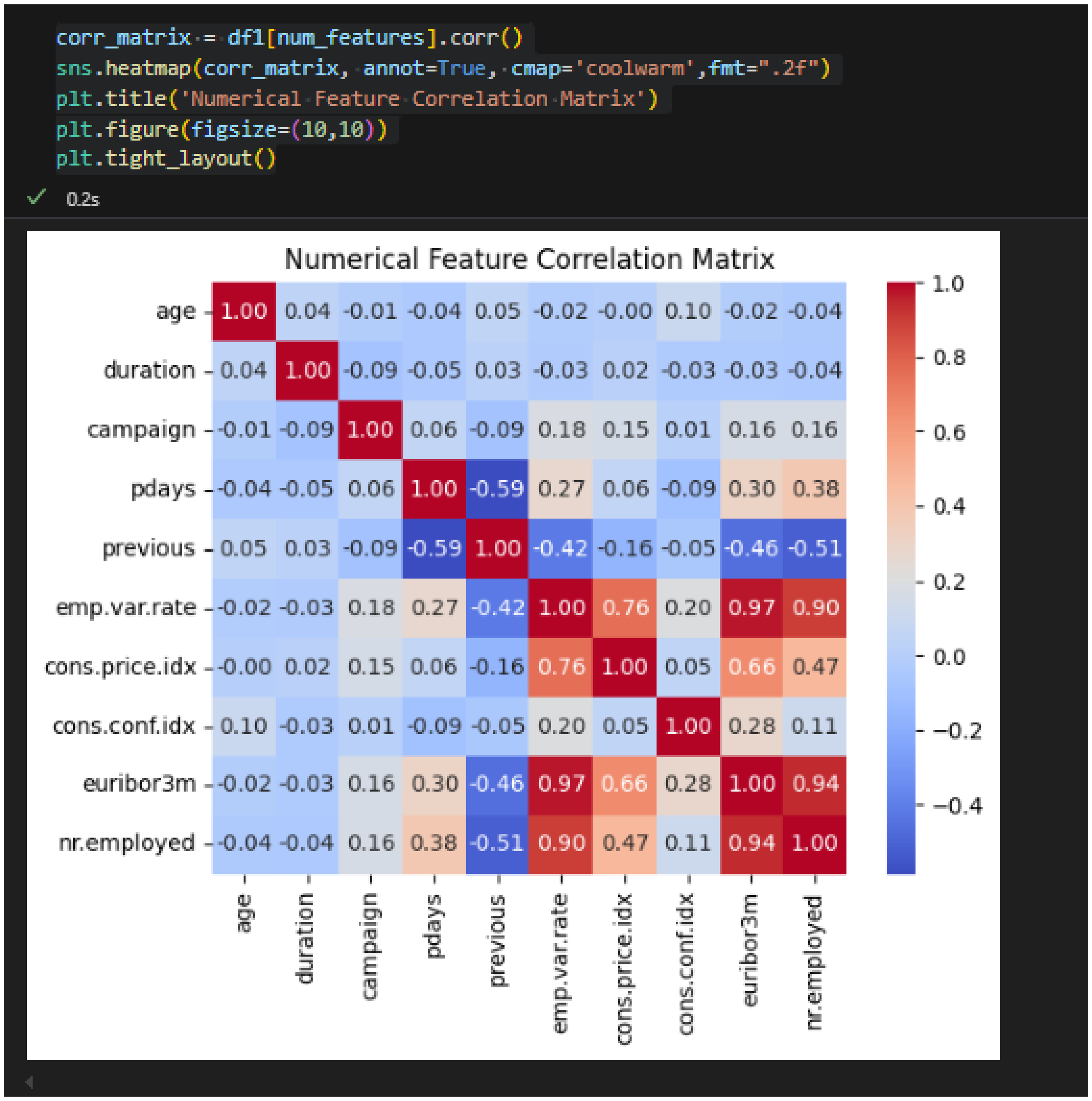
```
y
no     3668
yes     451
Name: count, dtype: int64
```



```
y
no     3668
yes     451
Name: count, dtype: int64
```

From the bar plot and pie chart, it shows the bank dataset for target variable is highly imbalance. We need to fix the imbalance because during model training, the distribution of the target variable will skewed towards the majority class which is the client that not subscribe to term deposit.

2.1.2 Correlation Analysis



```
corr_matrix = df1[num_features].corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm',fmt=".2f")
plt.title('Numerical Feature Correlation Matrix')
plt.figure(figsize=(10,10))
plt.tight_layout()
```

✓ 0.2s

Numerical Feature Correlation Matrix
age 1.00 0.04 -0.01 -0.04 0.05 -0.02 -0.00 0.10 -0.02 -0.04
duration 0.04 1.00 -0.09 -0.05 0.03 -0.03 0.02 -0.03 -0.03 -0.04
campaign -0.01 -0.09 1.00 0.06 -0.09 0.18 0.15 0.01 0.16 0.16
pdays -0.04 -0.05 0.06 1.00 -0.59 0.27 0.06 -0.09 0.30 0.38
previous 0.05 0.03 -0.09 -0.59 1.00 -0.42 -0.16 -0.05 -0.46 -0.51
emp.var.rate -0.02 -0.03 0.18 0.27 -0.42 1.00 0.76 0.20 0.97 0.90
cons.price.idx -0.00 0.02 0.15 0.06 -0.16 0.76 1.00 0.05 0.66 0.47
cons.conf.idx 0.10 -0.03 0.01 -0.09 -0.05 0.20 0.05 1.00 0.28 0.11
euribor3m -0.02 -0.03 0.16 0.30 -0.46 0.97 0.66 0.28 1.00 0.94
nr.employed -0.04 -0.04 0.16 0.38 -0.51 0.90 0.47 0.11 0.94 1.00
age
duration
campaign
pdays
previous
emp.var.rate
cons.price.idx
cons.conf.idx
euribor3m
nr.employed
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

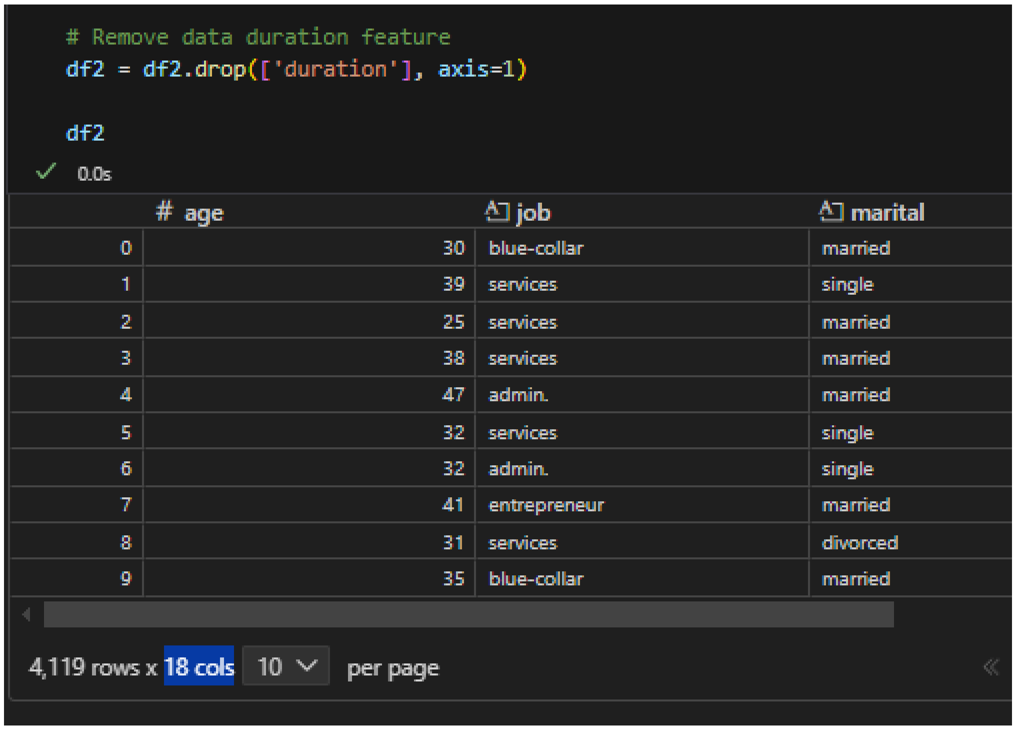
From the correlation matrix, we notice that (euribor3m and nr.employed= 0.94), (euribor3m and emp.var.rate= 0.97), (emp.var.rate and nr.employed= 0.90). This indicates that, it has multicollinearity among these feaure, so we need to drop some features to avoid redundancy. Let's decide which to drop first. We see that euribor3m is extremely highly correlated (0.97) with emp.var.rate and also highly (0.94) with nr.employed. So euribor3m is the most problematic, we should drop it first. After drop euribor3m, we still have a high correlation (0.90) between emp.var.rate and nr.employed. So, we need to drop emp.var.rate or nr.employed. If we choose to drop nr.employed and keep emp.var.rate,the emp.var.rate has a 0.76 correlation with cons.price.idx.

Alternatively, if we choose to drop emp.var.rate, the highest correlation is with cons.price.idx(0.47). So, we choose to drop emp.var.rate because it has slightly lower average correlations with other features.

2.2 Feature Engineering

2.2.1 Drop Irrelevant Feature

In this phase, the feature is either created or dropped. With a quick inspection in the dataset, we notice that 'duration' feature is contain post-call data. We need to drop this feature because this feature is only available after the call has been completed. This means we will not know the call duration for the future call and cannot make predictions on that. Models may learn the associate call duration with outcome, for instance longer call = higher subscription rate.



```
# Remove data duration feature
df2 = df2.drop(['duration'], axis=1)

df2
```

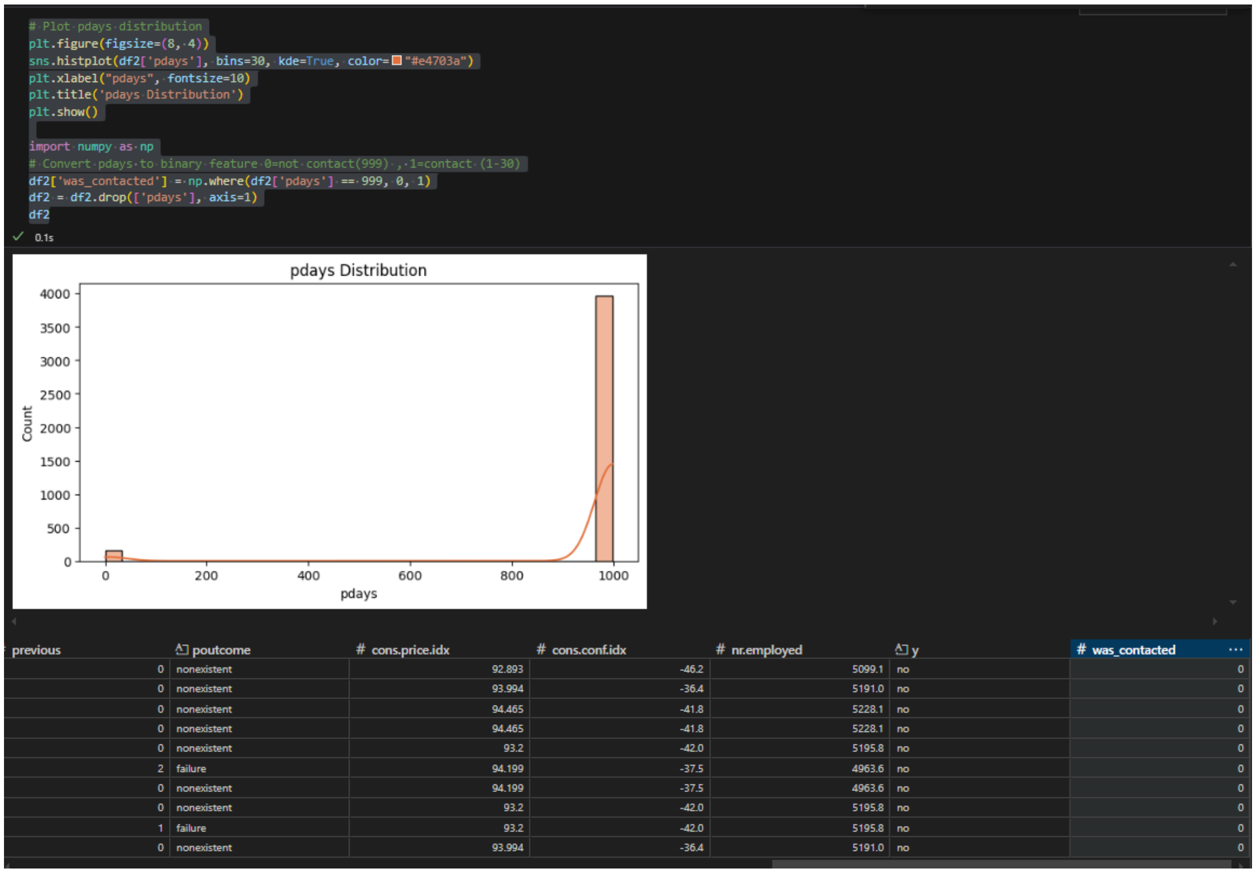
✓ 0.0s

	# age	job	marital
0	30	blue-collar	married
1	39	services	single
2	25	services	married
3	38	services	married
4	47	admin.	married
5	32	services	single
6	32	admin.	single
7	41	entrepreneur	married
8	31	services	divorced
9	35	blue-collar	married

4,119 rows x 18 cols 10 per page

2.2.2 Transformation of Feature

Next is transformation the variable 'p_day' to have a feature called 'was_contacted'. From the distribution plot of ‘pdays’, we can see that many client have not been contacted previously. Therefore we implement this binary transformation which the new feature is either call 1 if the client was contacted within the last 30 days and 0 otherwise (999). This transformation gives valuable information about customer recency and engagement. And this helps to predict the possibilities of these engagements towards the subscription of a term deposit.



```python
# Plot pdays distribution
plt.figure(figsize=(8, 4))
sns.histplot(df2['pdays'], bins=30, kde=True, color="#e4703a")
plt.xlabel("pdays", fontsize=10)
plt.title('pdays Distribution')
plt.show()

import numpy as np
# Convert pdays to binary feature 0=not contact(999) , 1=contact (1-30)
df2['was_contacted'] = np.where(df2['pdays'] == 999, 0, 1)
df2 = df2.drop(['pdays'], axis=1)
df2
```

```
✓ 0.1s
```

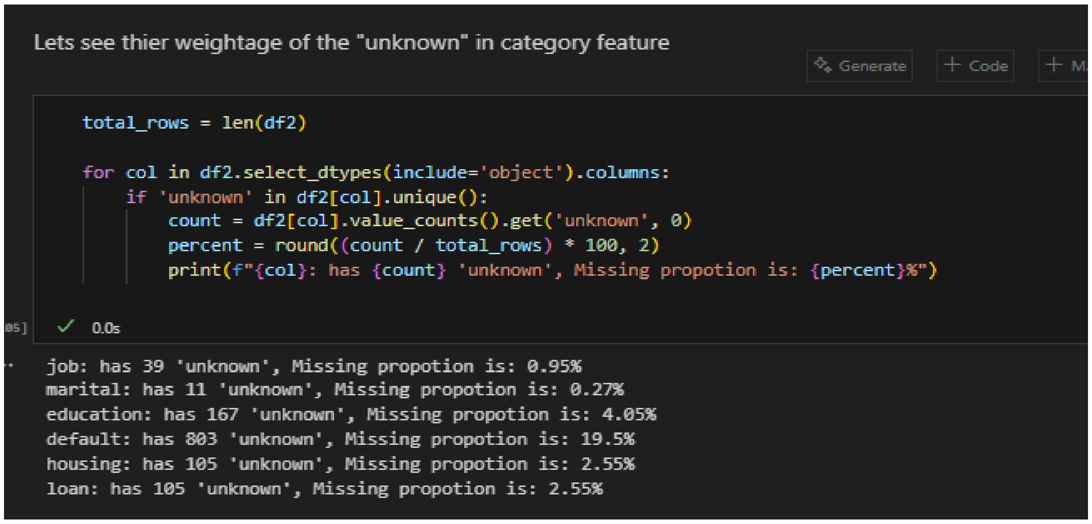
previous	poutcome	cons.price.idx	cons.conf.idx	nr.employed	y	was_contacted
0	nonexistent	92.893	-46.2	5099.1	no	0
0	nonexistent	93.994	-36.4	5191.0	no	0
0	nonexistent	94.465	-41.8	5228.1	no	0
0	nonexistent	94.465	-41.8	5228.1	no	0
0	nonexistent	93.2	-42.0	5195.8	no	0
2	failure	94.199	-37.5	4963.6	no	0
0	nonexistent	94.199	-37.5	4963.6	no	0
0	nonexistent	93.2	-42.0	5195.8	no	0
1	failure	93.2	-42.0	5195.8	no	0
0	nonexistent	93.994	-36.4	5191.0	no	0

2.3 Data Preprocessing

2.3.1 Handling Missing Value

In this data preprocessing phase, missing values and nonsignificant features should be carefully managed for dataset integrity. In the dataset, there is 'unknown' category in some feature. We need to make sure that their proportion is large or small. For less than <20% we can impute using mode.

Feature “poutcome” have 3352 value called “nonexistent”. This probably means the client has not been contacted before, therefore I find it significant and would not drop it.

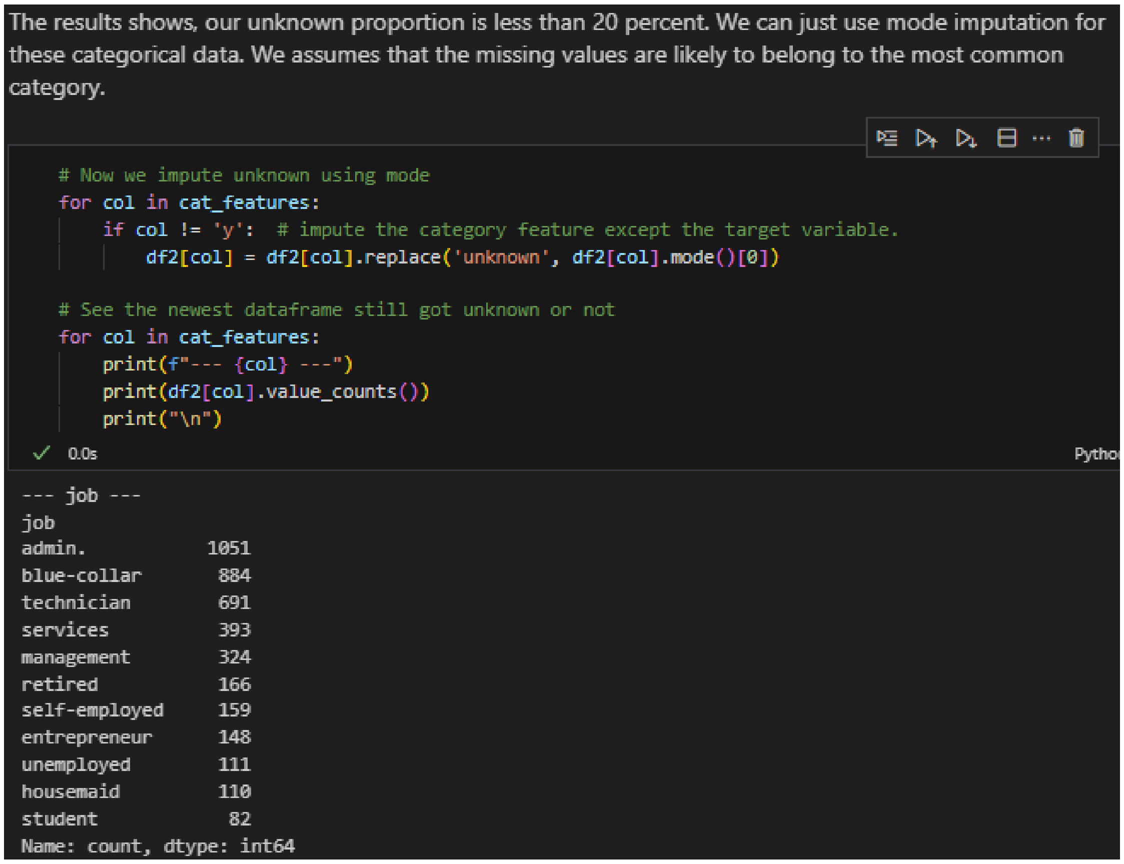


Lets see thier weightage of the "unknown" in category feature

```
total_rows = len(df2)

for col in df2.select_dtypes(include='object').columns:
    if 'unknown' in df2[col].unique():
        count = df2[col].value_counts().get('unknown', 0)
        percent = round((count / total_rows) * 100, 2)
        print(f"{col}: has {count} 'unknown', Missing propotion is: {percent}%")
```

```
job: has 39 'unknown', Missing propotion is: 0.95%
marital: has 11 'unknown', Missing propotion is: 0.27%
education: has 167 'unknown', Missing propotion is: 4.05%
default: has 803 'unknown', Missing propotion is: 19.5%
housing: has 105 'unknown', Missing propotion is: 2.55%
loan: has 105 'unknown', Missing propotion is: 2.55%
```

From the outcome, the maximum percentage of missing values is being 19.5 percent in the default category. Therefore, imputation using mode is workable in this scenario.

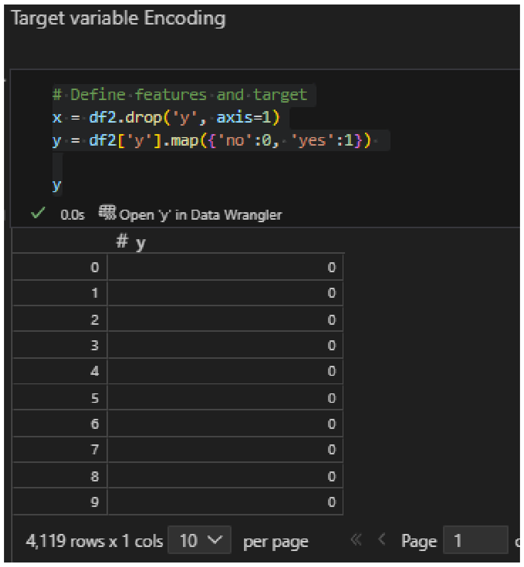


The results shows, our unknown proportion is less than 20 percent. We can just use mode imputation for these categorical data. We assumes that the missing values are likely to belong to the most common category.

```
# Now we impute unknown using mode
for col in cat_features:
    if col != 'y':  # impute the category feature except the target variable.
        df2[col] = df2[col].replace('unknown', df2[col].mode()[0])

# See the newest dataframe still got unknown or not
for col in cat_features:
    print(f"--- {col} ---")
    print(df2[col].value_counts())
    print("\n")
```

```
--- job ---
job
admin.           1051
blue-collar       884
technician        691
services          393
management        324
retired           166
self-employed     159
entrepreneur      148
unemployed        111
housemaid         110
student            82
Name: count, dtype: int64
```

2.3.2 Feature Encoding


Target variable Encoding

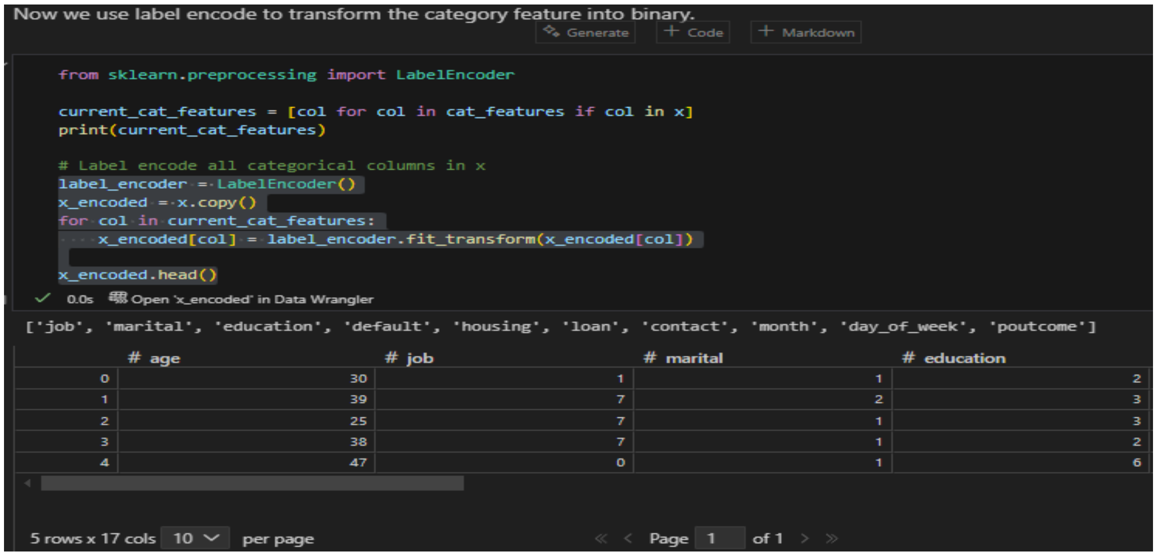
```
# Define features and target
x = df2.drop('y', axis=1)
y = df2['y'].map({'no':0, 'yes':1})

y
```

✓ 0.0s Open 'y' in Data Wrangler

	# y
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

4,119 rows x 1 cols 10 per page Page 1

Next, the variable x is created by dropping the 'y' column from the DataFrame df2, so it contains all the input features except the target. The variable y is defined by taking the 'y' column from df2 and mapping its categorical values where 'no' is converted to 0 and 'yes' to 1. This mapping transforms the target variable into a binary numeric format. This encoding step is essential, where we need clearly separated features (x) and target (y).


Now we use label encode to transform the category feature into binary.

```
from sklearn.preprocessing import LabelEncoder

current_cat_features = [col for col in cat_features if col in x]
print(current_cat_features)

# Label encode all categorical columns in x
label_encoder = LabelEncoder()
x_encoded = x.copy()
for col in current_cat_features:
    x_encoded[col] = label_encoder.fit_transform(x_encoded[col])

x_encoded.head()
```

✓ 0.0s Open 'x_encoded' in Data Wrangler

```
['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'month', 'day_of_week', 'poutcome']
```

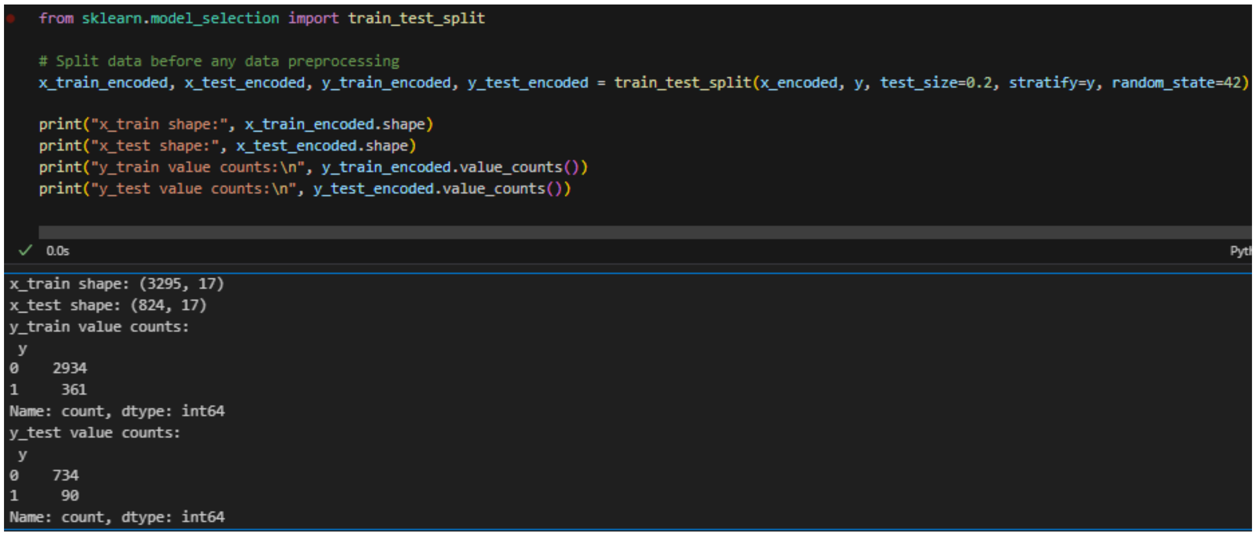
	# age	# job	# marital	# education
0	30	1	1	2
1	39	7	2	3
2	25	7	1	3
3	38	7	1	2
4	47	0	1	6

5 rows x 17 cols 10 per page Page 1 of 1

Next, we encode the categorical features in a DataFrame using scikit-learn's LabelEncoder. For each column, fit_transform is called, which assigns a unique integer to each category in that column and replaces the original values with these integers. This process converts text categories into numeric codes, making the data suitable for machine learning algorithms that require numerical input.

2.3.3 Data Splitting

In order to have a balanced training set with respect to the target feature, a stratified 80-20 train test split was performed which is to keep the original class distribution of the target feature.

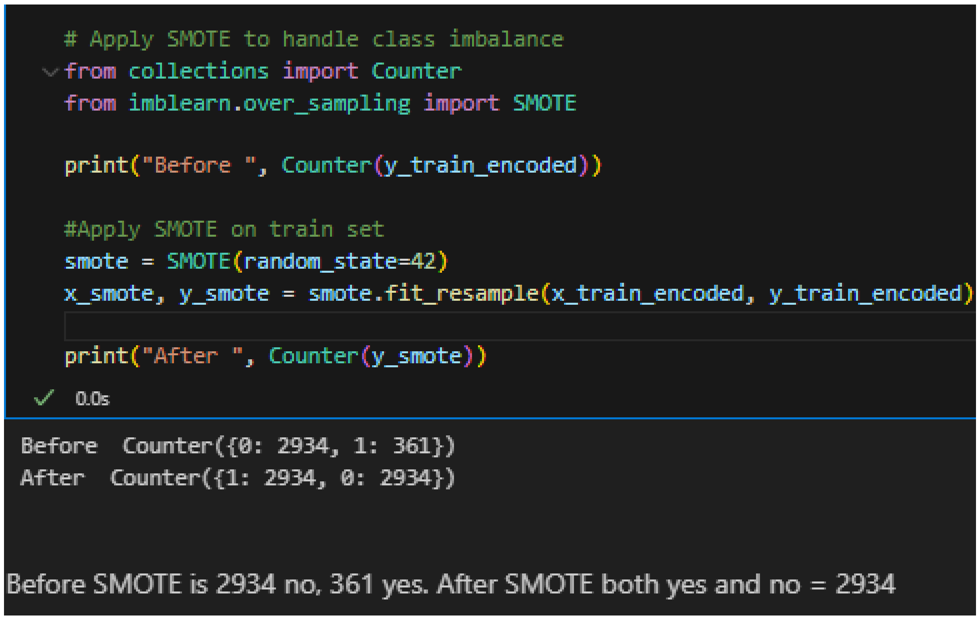


```
from sklearn.model_selection import train_test_split

# Split data before any data preprocessing
x_train_encoded, x_test_encoded, y_train_encoded, y_test_encoded = train_test_split(x_encoded, y, test_size=0.2, stratify=y, random_state=42)

print("x_train shape:", x_train_encoded.shape)
print("x_test shape:", x_test_encoded.shape)
print("y_train value counts:\n", y_train_encoded.value_counts())
print("y_test value counts:\n", y_test_encoded.value_counts())
```

```
x_train shape: (3295, 17)
x_test shape: (824, 17)
y_train value counts:
 y
0    2934
1     361
Name: count, dtype: int64
y_test value counts:
 y
0    734
1     90
Name: count, dtype: int64
```

2.4 Class Imbalance Handling

Synthetic Minority Over-sampling Technique (SMOTE) was used to balance the dataset and increased the samples of minority class, clients who subscribed to a term deposit to 2,934. This resampling method creates synthetic samples from the minority class samples and reduces the possibility of the model being bias to the majority class.



```
# Apply SMOTE to handle class imbalance
from collections import Counter
from imblearn.over_sampling import SMOTE

print("Before ", Counter(y_train_encoded))

#Apply SMOTE on train set
smote = SMOTE(random_state=42)
x_smote, y_smote = smote.fit_resample(x_train_encoded, y_train_encoded)

print("After ", Counter(y_smote))
```

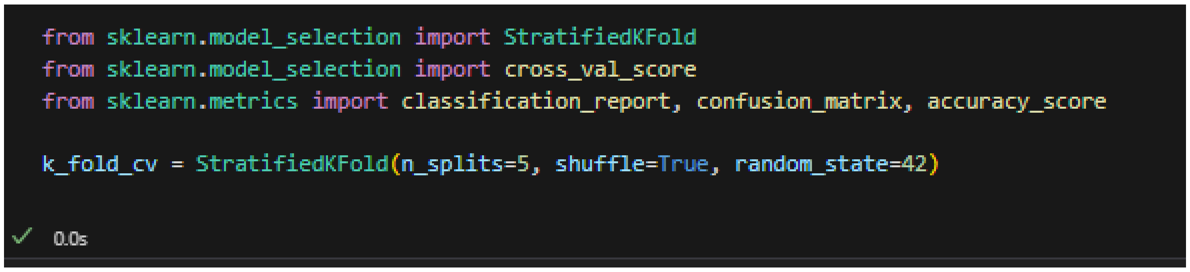
✓ 0.0s

```
Before  Counter({0: 2934, 1: 361})
After  Counter({1: 2934, 0: 2934})
```

Before SMOTE is 2934 no, 361 yes. After SMOTE both yes and no = 2934

2.5 Model Architecture and Development

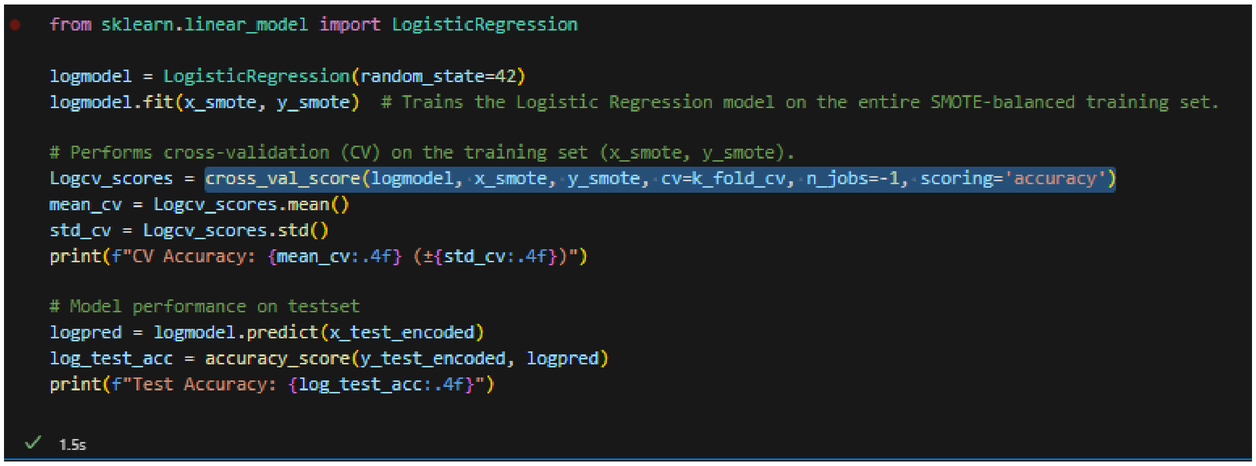
In the model architecture phase, a cross-validation technique was set up to guarantee a good performance on the predictive models. The Stratified K-Fold cross validation method was used with 5 splits (n_splits=5) to ensure a robust model performance evaluation. Also, “shuffle=True” is used to shuffle the data before splitting and also to avoid biases due to the ordered nature of the dataset.



```
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

k_fold_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

✓ 0.0s

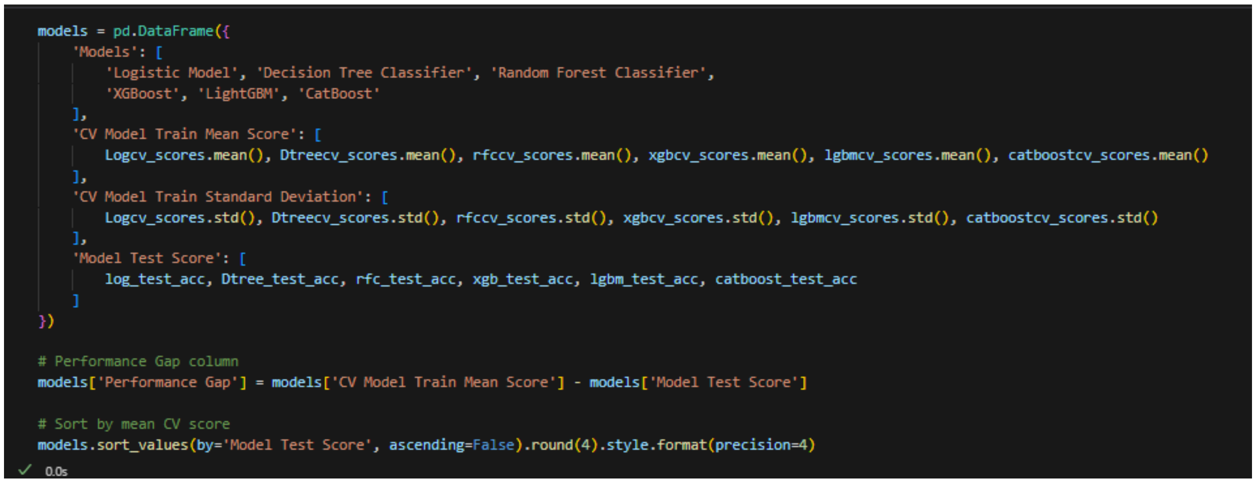
In order to execute this cross-validation framework, the ‘cross_val_score’ function was applied to compare multiple models within the stratified folds defined. The function was designed to evaluate the models and was provided with the SMOTE dataset (x_smote and y_smote). By specifying" n_jobs=-1", the implementation was parallelized, thus making the computation much faster.



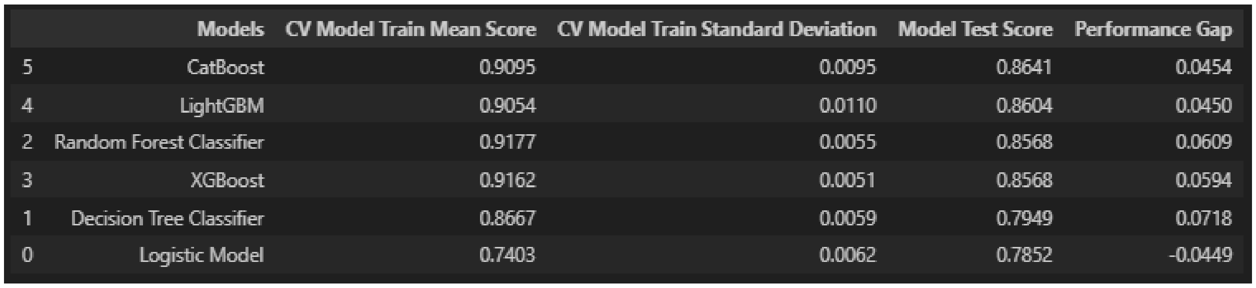
```
from sklearn.linear_model import LogisticRegression

logmodel = LogisticRegression(random_state=42)
logmodel.fit(x_smote, y_smote)  # Trains the Logistic Regression model on the entire SMOTE-balanced training set.

# Performs cross-validation (CV) on the training set (x_smote, y_smote).
Logcv_scores = cross_val_score(logmodel, x_smote, y_smote, cv=k_fold_cv, n_jobs=-1, scoring='accuracy')
mean_cv = Logcv_scores.mean()
std_cv = Logcv_scores.std()
print(f"CV Accuracy: {mean_cv:.4f} (±{std_cv:.4f})")

# Model performance on testset
logpred = logmodel.predict(x_test_encoded)
log_test_acc = accuracy_score(y_test_encoded, logpred)
print(f"Test Accuracy: {log_test_acc:.4f}")

✓ 1.5s
```

This model training process makes all six algorithms Logistic Regression, Decision Tree, Random Forest, XGBoost, LightGBM, and CatBoost will be properly benchmarked. We use standard library settings for the classifiers while designated "random_state=42" so that results are consistent. The results from the cross-validation process give us a valuable insights to the performance of the models. It will guide us to selecting the best algorithm to predict the term deposit subscription.

3.0 Model Evaluation and Selection



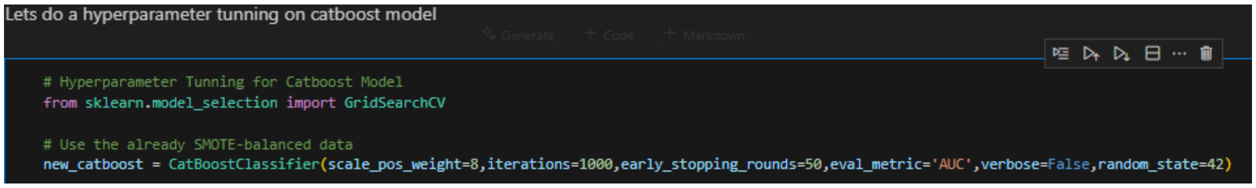
```
models = pd.DataFrame({
    'Models': [
        'Logistic Model', 'Decision Tree Classifier', 'Random Forest Classifier',
        'XGBoost', 'LightGBM', 'CatBoost'
    ],
    'CV Model Train Mean Score': [
        Logcv_scores.mean(), Dtreecv_scores.mean(), rfccv_scores.mean(), xgbcv_scores.mean(), lgbmcv_scores.mean(), catboostcv_scores.mean()
    ],
    'CV Model Train Standard Deviation': [
        Logcv_scores.std(), Dtreecv_scores.std(), rfccv_scores.std(), xgbcv_scores.std(), lgbmcv_scores.std(), catboostcv_scores.std()
    ],
    'Model Test Score': [
        log_test_acc, Dtree_test_acc, rfc_test_acc, xgb_test_acc, lgbm_test_acc, catboost_test_acc
    ]
})

# Performance Gap column
models['Performance Gap'] = models['CV Model Train Mean Score'] - models['Model Test Score']

# Sort by mean CV score
models.sort_values(by='Model Test Score', ascending=False).round(4).style.format(precision=4)
✓ 0.0s
```



	Models	CV Model Train Mean Score	CV Model Train Standard Deviation	Model Test Score	Performance Gap
5	CatBoost	0.9095	0.0095	0.8641	0.0454
4	LightGBM	0.9054	0.0110	0.8604	0.0450
2	Random Forest Classifier	0.9177	0.0055	0.8568	0.0609
3	XGBoost	0.9162	0.0051	0.8568	0.0594
1	Decision Tree Classifier	0.8667	0.0059	0.7949	0.0718
0	Logistic Model	0.7403	0.0062	0.7852	-0.0449

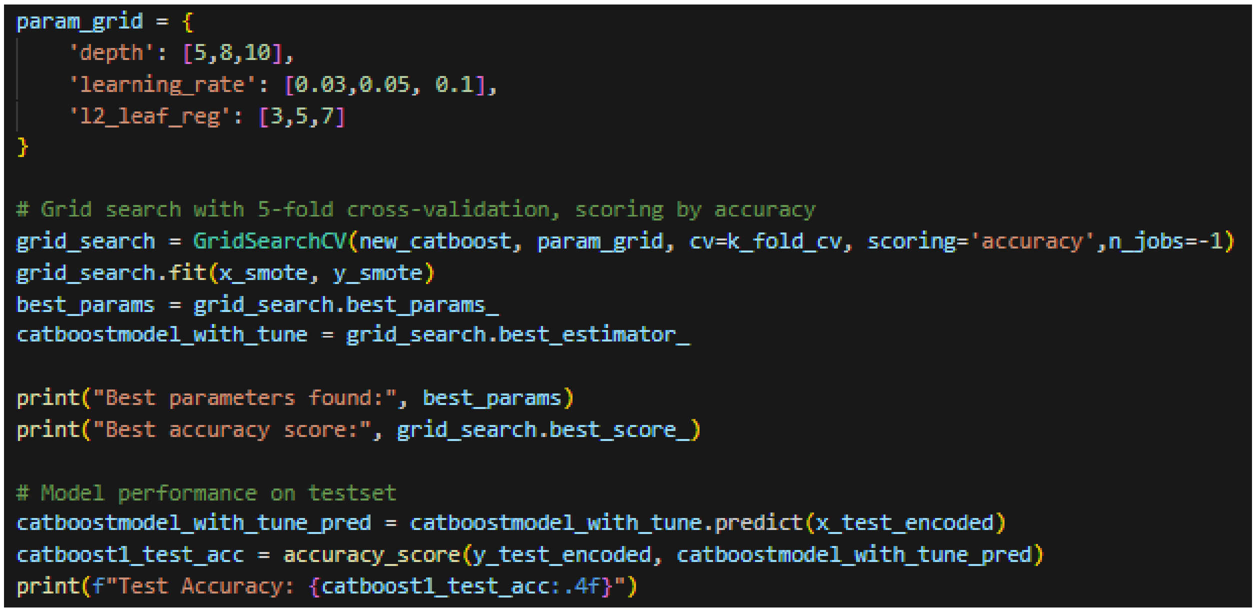
Evaluation of the six classification models shows that CatBoost model is the one that generalizes best with the unseen data since it has the highest test score of 0.8641 and lowest performance gaps 0.0454. This is followed by LightGBM with a test score of 0.8604 on train set and 0.0450 on performance gaps. On the other hand, Random Forest and XGBoost had a consistent overfitting tendency, with performance differences 0.0609 and 0.0594, respectively. This suggests that they memorized the training data too much, including noise in the data therefore leading to a bad generalization of the model. Decision Tree model had the largest performance difference of 0.0718, indicating the overfitting is related to its complexity. Interestingly, the Logistic Regression model was an outlier in terms of negative performance difference -0.0449 with the test score 0.7852 being superior to the training score 0.7403. Indicating potential underfitting or data leakage. Based on these findings, CatBoost models are suggested for deployed since it has better performance and generalization.

4.0 The proposed optimized parameters


Lets do a hyperparameter tunning on catboost model

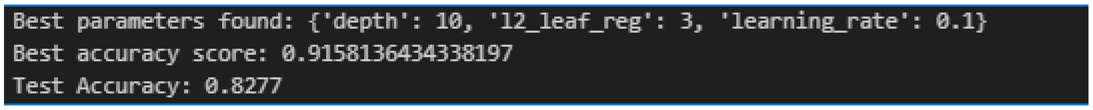
```
    # Hyperparameter Tunning for Catboost Model
    from sklearn.model_selection import GridSearchCV

    # Use the already SMOTE-balanced data
    new_catboost = CatBoostClassifier(scale_pos_weight=8,iterations=1000,early_stopping_rounds=50,eval_metric='AUC',verbose=False,random_state=42)
```

The tuning approach to optimize the CatBoost model was grid search with 5-fold cross-validation used to tune the model parameters on the SMOTE-balanced dataset. The original setting for the CatBoostClassifier contained parameters like scale_pos_weight=8 which adjusts the positive class imbalance ratio to the negative class and hence helps the model in controlling the imbalance by providing more weight to the minority class. The parameter iterations=1000 is to set the maximum number of boosting rounds, and early_stopping_rounds = 50 allowed the early stop of training as the model did not continue improving for 50 rounds, and thus kept the model from overfitting on the training set. In addition, eval_metric='AUC' told the model to use the Area Under the Curve to monitor model performance while it was training, and verbose=False prevented the model from being too chatty and cluttering up the console with logs. The random_state=42 made sure it was reproducible, by setting the seed for the random number generation.



```
param_grid = {
    'depth': [5,8,10],
    'learning_rate': [0.03,0.05, 0.1],
    'l2_leaf_reg': [3,5,7]
}

# Grid search with 5-fold cross-validation, scoring by accuracy
grid_search = GridSearchCV(new_catboost, param_grid, cv=k_fold_cv, scoring='accuracy',n_jobs=-1)
grid_search.fit(x_smote, y_smote)
best_params = grid_search.best_params_
catboostmodel_with_tune = grid_search.best_estimator_

print("Best parameters found:", best_params)
print("Best accuracy score:", grid_search.best_score_)

# Model performance on testset
catboostmodel_with_tune_pred = catboostmodel_with_tune.predict(x_test_encoded)
catboost1_test_acc = accuracy_score(y_test_encoded, catboostmodel_with_tune_pred)
print(f"Test Accuracy: {catboost1_test_acc:.4f}")
```

A dictionary param_grid was created to define the hyperparameters and their respective values for the grid search. This parameters were depth (5, 8, 10), which defines the maximum depth of splits that can be made, learning_rate (0.03, 0.05, 0.1), is a factor for weighing the tree loss when updating the model weights between iteration steps, and l2_leaf_reg (3, 5, 7), provided the regularization parameter used to penalize the leaf weights. The grid search was performed using GridSearchCV. This method trained and tested the model on each possible combination of hyperparameters, ultimately determining which was the best for improving performance.



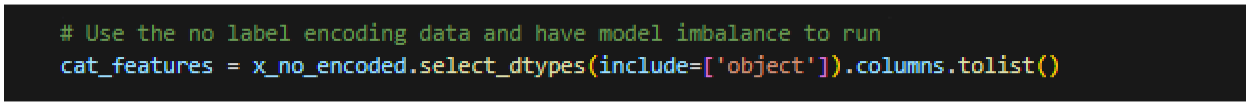
```
Best parameters found: {'depth': 10, 'l2_leaf_reg': 3, 'learning_rate': 0.1}
Best accuracy score: 0.9158136434338197
Test Accuracy: 0.8277
```

The results indicated the best parameters are 'depth': 10, 'l2_leaf_reg': 3, 'learning_rate': 0.1 corresponding to a best accuracy score around 0.9158 for the cross-validation. But the Model’s test accuracy was slightly lower, which is 0.8277.

Even though the tuned CatBoost model experienced a little increase in the training accuracy from 0.9095 to 0.9158, but its test accuracy has decreased from 0.864 to 0.8277. This difference indicates a potential overfitting problem of the model on the SMOTE training data. Furthermore, using a label encoding probably distorts the categories relations.

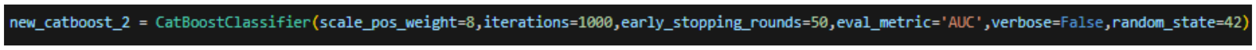
5.0 Implementation of Proposed Improvement and Enhancement

To resolve these problems, the SMOTE operation should not be used, and no encoding is performed on the training set. This is because CatBoost is naturally able to cope with categorical attributes well without preprocessing them, and it also has mechanisms for handling the class imbalance already built in as part of the parameter "scale_pos_weight". By exploiting these inherent properties, the model can directly learn the original data distribution.



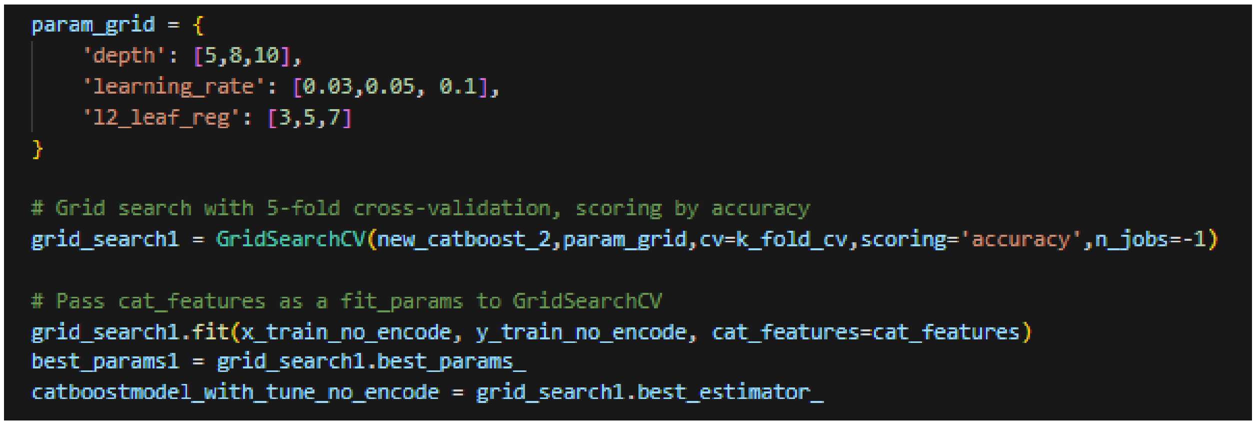
```
# Use the no label encoding data and have model imbalance to run
cat_features = x_no_encoded.select_dtypes(include=['object']).columns.tolist()
```

Next, the CatBoost model will be retrained with no SMOTE and encoding methods. This is to enabling the model to utilize CatBoost's native supports for identifying categorical features and imbalanced classes. The categorical features were chosen from the dataset as cat_features = x_no_encoded. select_dtypes(include=['object']) . columns. tolist() to ensure that these features were kept as strings for when the model was trained.



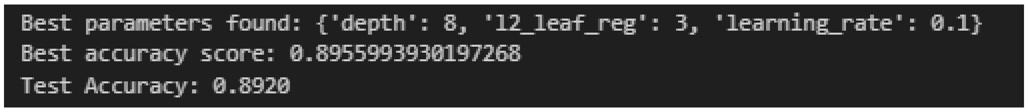
```
new_catboost_2 = CatBoostClassifier(scale_pos_weight=8,iterations=1000,early_stopping_rounds=50,eval_metric='AUC',verbose=False,random_state=42)
```

The model was implemented using CatBoostClassifier parameters and directed scale_pos_weight=8 to counterbalance the high class imbalance, iterations=1000 to indicate maximum boosting rounds, and early_stopping_rounds=50 to stop training if it does not improve further. The performance was measured by setting as the evaluation metric 'AUC' during training, the hyperparameter tuning was performed using a grid search with 5-fold cross-validation and attending to accuracy of scoring.



```
param_grid = {
    'depth': [5,8,10],
    'learning_rate': [0.03,0.05, 0.1],
    'l2_leaf_reg': [3,5,7]
}

# Grid search with 5-fold cross-validation, scoring by accuracy
grid_search1 = GridSearchCV(new_catboost_2,param_grid,cv=k_fold_cv,scoring='accuracy',n_jobs=-1)

# Pass cat_features as a fit_params to GridSearchCV
grid_search1.fit(x_train_no_encode, y_train_no_encode, cat_features=cat_features)
best_params1 = grid_search1.best_params_
catboostmodel_with_tune_no_encode = grid_search1.best_estimator_
```

The grid of parameters to be tuned was this, depth (5, 8, 10) , learning_rate (0.03, 0.05, 0.1) l2_leaf_reg (3, 5, 7). The grid search was performed treating the categorical as fit_params from GridSearchCV, so the model keeps the capability to understand the categorical properly.

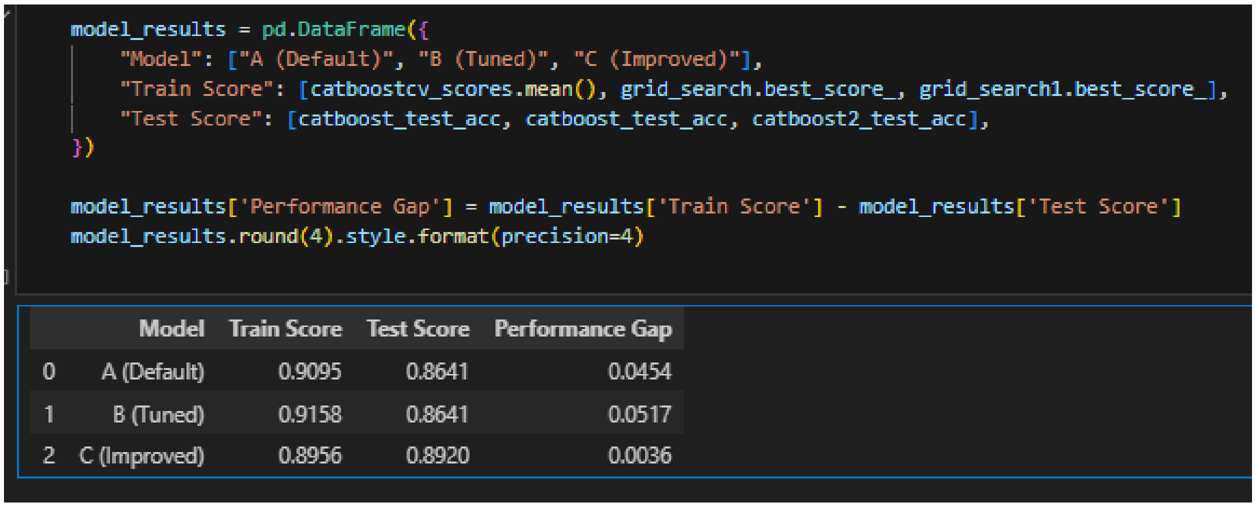


```
Best parameters found: {'depth': 8, 'l2_leaf_reg': 3, 'learning_rate': 0.1}
Best accuracy score: 0.8955993930197268
Test Accuracy: 0.8920
```

The best parameters obtained after grid search are {'depth':8, 'l2_leaf_reg':3, 'learning_rate': 0.1} with a best cross-validated accuracy score of around 0.8956. When tested on the test set, the model obtained 0.8920 as test accuracy, showing an important increase in generalization with respect to the previous one. This performance proves that the modified model pipeline and training strategy improve the model's classification power on term deposit subscription while keeping strong handling of categorical and class imbalanced data.

6.0 Result analysis and Discussion

The comparison of three CatBoost models, Model A (default model with SMOTE and encoding), Model B (hyperparameter tunning model with SMOTE and encoding), and Model C (hyperparameter tunning model without SMOTE and using native categorical handling). This provides a valuable insight into their performance and generalization.



```python
model_results = pd.DataFrame({
    "Model": ["A (Default)", "B (Tuned)", "C (Improved)"],
    "Train Score": [catboostcv_scores.mean(), grid_search.best_score_, grid_search1.best_score_],
    "Test Score": [catboost_test_acc, catboost_test_acc, catboost2_test_acc],
})

model_results['Performance Gap'] = model_results['Train Score'] - model_results['Test Score']
model_results.round(4).style.format(precision=4)
```

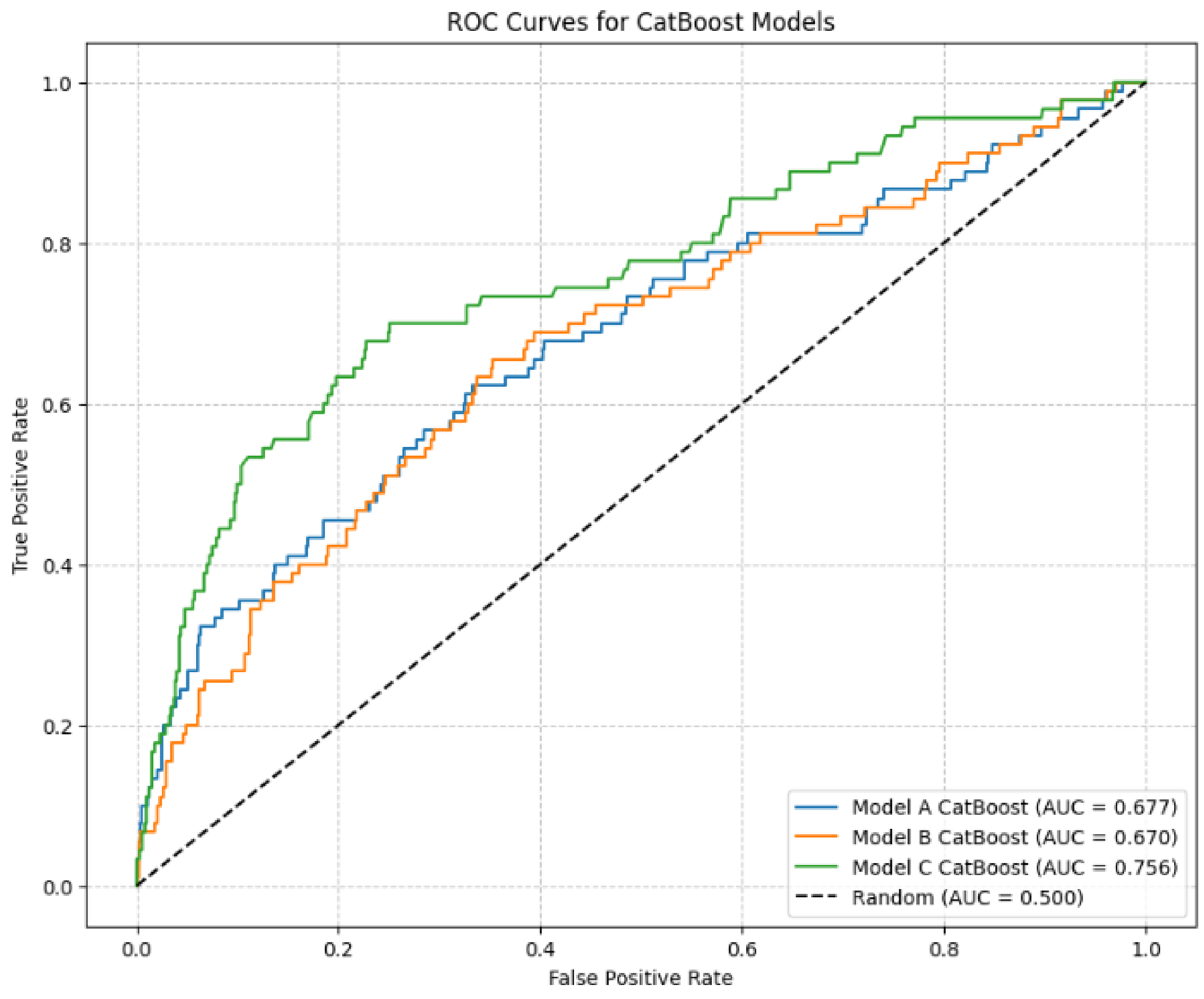
	Model	Train Score	Test Score	Performance Gap
0	A (Default)	0.9095	0.8641	0.0454
1	B (Tuned)	0.9158	0.8641	0.0517
2	C (Improved)	0.8956	0.8920	0.0036

From the result table, the result performance of Model C yielded the highest test accuracy of the three models (0.8920), which were also higher to Model A (0.8641) and Model B (0.8277). This suggests that Model C performs best on test data examples in terms of predicting the term deposit subscriptions.

In terms of generalization, Model C achieved the smallest performance gap between train data and test data of (0.0036) which implies very strong generalization. In contrast, Model A and B showed a larger gap of (0.0454) and 0.0881 respectively which indicates that both models might be overfitting.

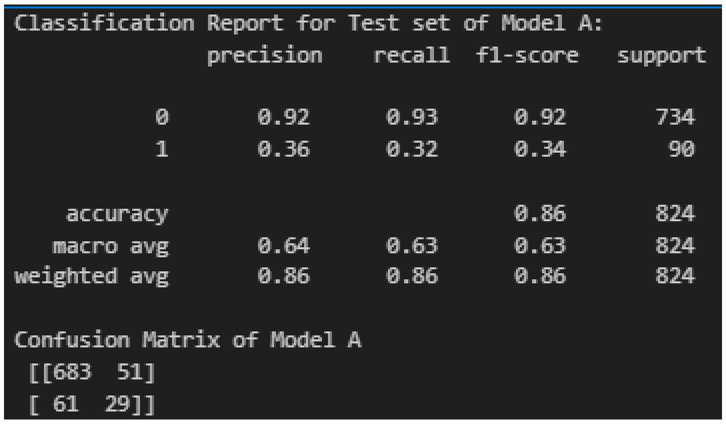
For training performance, although Model C has a train score of (0.8956) which is lower than both Model A (0.9095) and Model B (0.9158). However, this is a good sign, because this stands for less overfitting. Our main objective is to maximize the test performance, which is achieved by Model C even if its training score is lower.

6.1 ROC Analysis



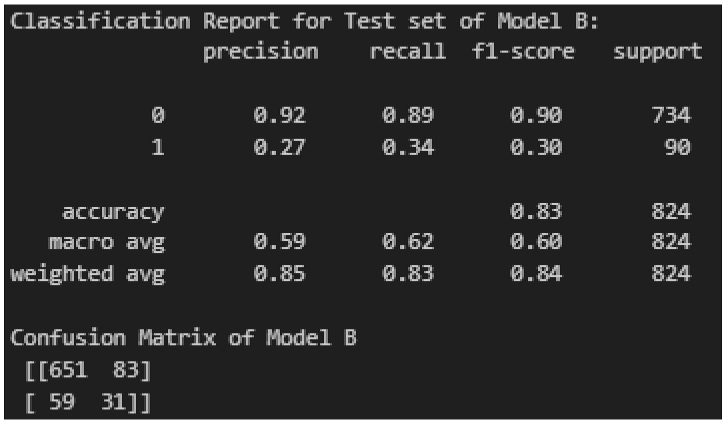
The Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC) are a visual representation for model performance at different classificatory thresholds. Model A has a moderate discriminatory ability with an AUC of (0.677). Although its performance is significantly better than random guess, it is not sufficient for the decision making. Model B has a little bit smaller AUC (0.670) than that of the model A, it means that the tuning of the CatBoost model and SMOTE added could not make the model discrimination ratio increase. Model C achieves the highest AUC of (0.756). Representing its better capability of identifying positive samples (subscribers) over negative samples (non-subscribers). This indicates that Model C provides better prediction results of term deposit subscriptions by capturing the imbalance structure of the data.

6.2 Classification Report and Confusion Matrix



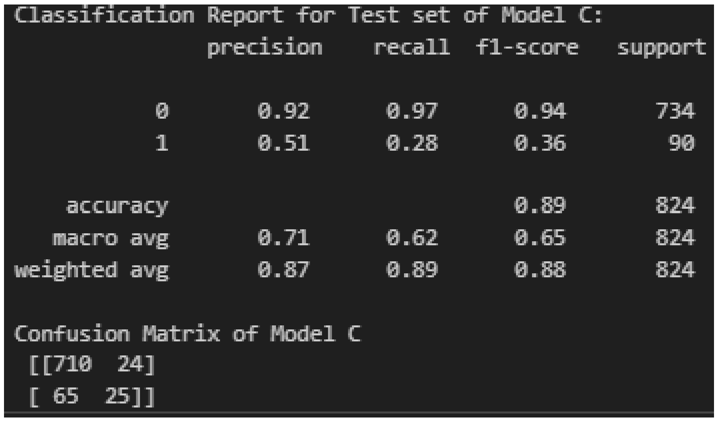
```
Classification Report for Test set of Model A:
              precision    recall  f1-score   support

           0       0.92      0.93      0.92       734
           1       0.36      0.32      0.34        90

    accuracy                           0.86       824
   macro avg       0.64      0.63      0.63       824
weighted avg       0.86      0.86      0.86       824

Confusion Matrix of Model A
 [[683  51]
 [ 61  29]]
```

Model A (Default CatBoost) achieves a moderate performance. The confusion matrix shows that model A only detected 29/90 actual subscribers with a recall sensitivity of 32%. Its precision is 36%, indicating that when predicting "yes"(subscribe client), only 36% were correct. The F1-score stands at 34%, which is interpreted as a trade-off between recall and precision. We observe that the false positive rate of Model A in a business context is very high, which is 68% of missing potential subscribed clients (61/90). This model might be acceptable if the cost of missing subscribers (FN- Marketing to non-subscribers) is low and the cost of falsely targeting non-subscribers (FP- Missing actual subscribers) is high. However, our goal is to identify subscribers, so this model is not effective.



```
Classification Report for Test set of Model B:
              precision    recall  f1-score   support

           0       0.92      0.89      0.90       734
           1       0.27      0.34      0.30        90

    accuracy                           0.83       824
   macro avg       0.59      0.62      0.60       824
weighted avg       0.85      0.83      0.84       824

Confusion Matrix of Model B
 [[651  83]
 [ 59  31]]
```

Model B (Tuned CatBoost) achieves a slight increase in recall up to 34% (detect 31/90 real subscriber). But it fails to maintain precision at 27%, leading to a larger false positive rate of 73% (73% of targeted "yes" are non-subscribers). Although model B manages to capture a few more true subscribers (31TP) relative to model A, the overall accuracy drops to 83% and the proportion of false subscriber is (83FP), leads to a significant amount of unnecessarily addressed marketing. The F1-score of 30% indicates that the model tuning does not give a better balance of precision and recall compared to Model A.



```
Classification Report for Test set of Model C:
              precision    recall  f1-score   support

           0       0.92      0.97      0.94       734
           1       0.51      0.28      0.36        90

    accuracy                           0.89       824
   macro avg       0.71      0.62      0.65       824
weighted avg       0.87      0.89      0.88       824

Confusion Matrix of Model C
 [[710  24]
 [ 65  25]]
```

Model C (Native CatBoost with Imbalance Handling) has the highest F1-score (36%) with lower recall (28%). However, it has the highest precision at 51% compared to Model A and B and it also has the least number of false positives at 24. Although model C fails to find more subscribers (65 FN), such trade-off is reasonable considering that less resources is wasted on client that would not subscribes.

Overall, in the business context, Model C is better to deploy because it is good in situations where marketing resources are expensive (eg. high expenses for the bank to recruit people to call the client). Although the model C subscriber recognition is slightly less but the precision advance decreases the resource overhead.

7.0 Conclusion

In summary, this project created an effective predictive model that helped the bank to target potential term deposit customers and therefore manage the telemarketing resources efficiently. Due to the strong class imbalance in the dataset, this project was developed the CatBoost native class imbalance for handling and improved interpretability. Model C (hyperparameter tunning Catboost model without SMOTE and using native categorical handling) was found to be the best model with the highest of 89.20% test accuracy, smallest performance gap and the best AUC (0.756), which means that Model C discriminates well between subscribers and non-subscribers. The model achieves 51% in precision making it suitable for saving significant amount of wasted marketing by minimizing false positives. For the future, the project result suggests the bank to employ Model C in the scenario of resource-constrained or budget constrained campaigns. Model C offers a tractable and efficient approach to term deposit subscription prediction, which can be taken as a baseline for data oriented optimization in marketing campaign.