Introduction to Data Analytics (31250) Assignment 3
Name: Aditya Narayan Chaudhuri
Student id: 25744309

Data mining Problem:

The object of this problem is to predict whether a customer will subscribe to a term deposit (subscribed = 1) or not (subscribed = 0) based on their demographic and financial attributes.
Business Impact:

- Helps the company target marketing campaigns more effectively.
- Reduces costs by focusing on high-probability customers.

Input:

The dataset includes:

- Demographic features: age, job, marital, education, etc.
- Financial features: balance, loan, housing, etc.
- Campaign-related features: contact, duration, campaign, etc.

Output:

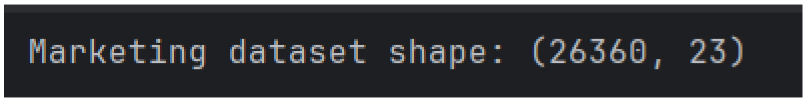
A predictive model that classifies customers as likely to subscribe (1) or not (0).

Data Preparation

Data Exploration

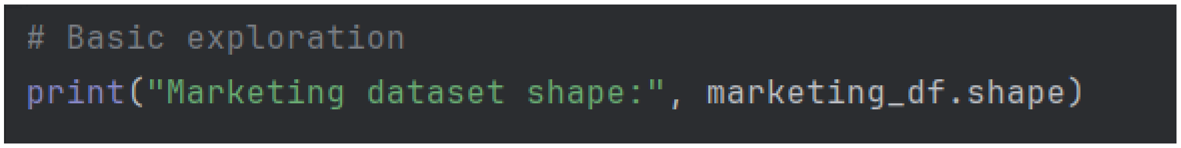
There were numerous values labelled as '?' which caused issues when it came to reading the data. To solve this issue we replaced all the values with NaN using the .replace() function.

The marketing dataset contains 2630 rows and 23 columns as shown by the output in Figure 1.



```
Marketing dataset shape: (26360, 23)
```

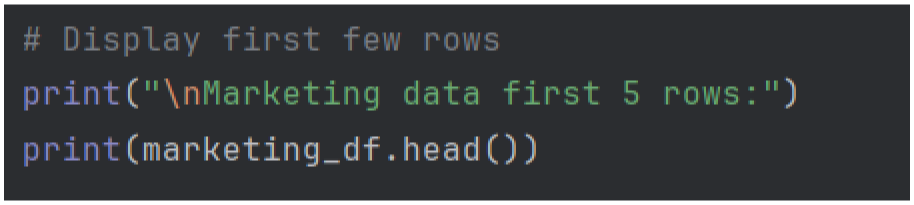
Figure 1 shows the output regarding the shape of the dataset.



```
# Basic exploration
print("Marketing dataset shape:", marketing_df.shape)
```

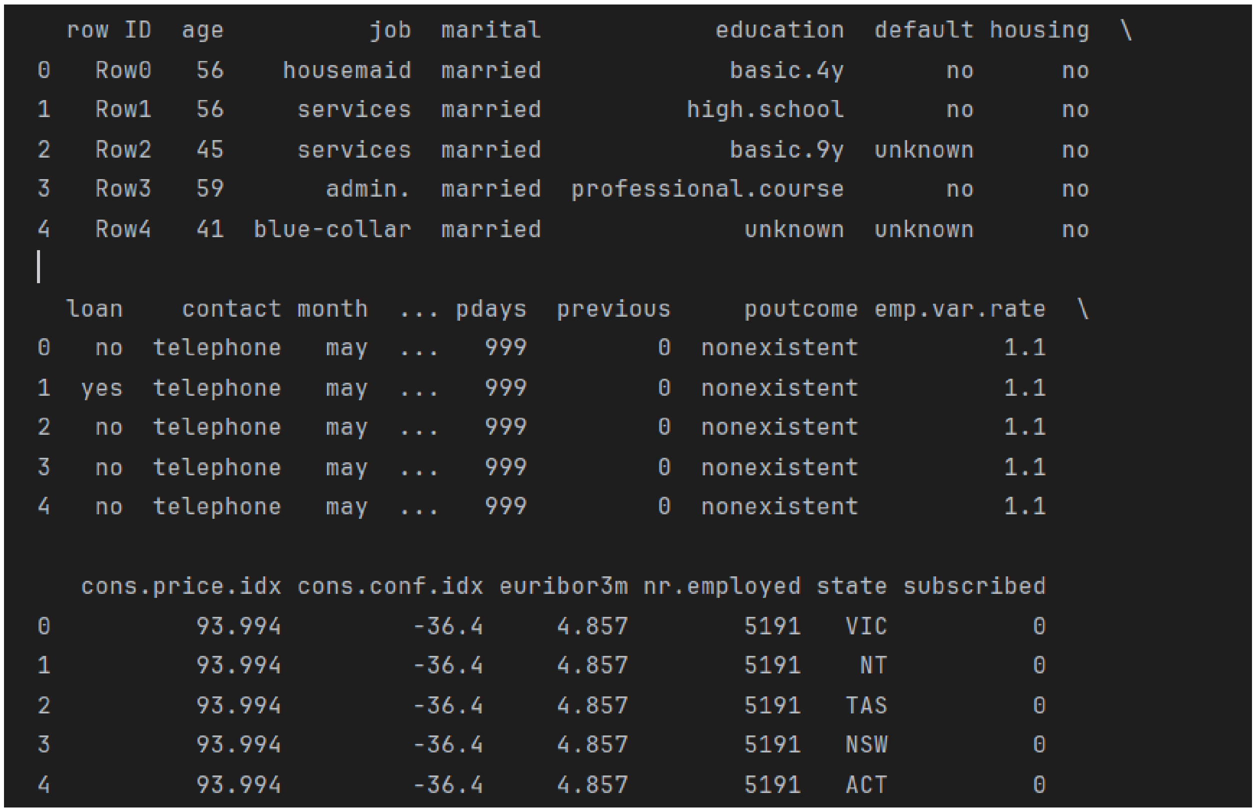
Figure 2 shows the code used to output the shape of the dataset.

Using the code in Figure 3 we checked some sample data to find out the structure of the dataset.



```python
# Display first few rows
print("\nMarketing data first 5 rows:")
print(marketing_df.head())
```

Figure 3 shows the code to find the first 5 rows.



```
  row ID  age          job  marital            education  default housing  \
0   Row0   56    housemaid  married             basic.4y       no      no
1   Row1   56     services  married          high.school       no      no
2   Row2   45     services  married             basic.9y  unknown      no
3   Row3   59       admin.  married  professional.course       no      no
4   Row4   41  blue-collar  married              unknown  unknown      no

  loan    contact month  ... pdays  previous     poutcome emp.var.rate  \
0   no  telephone   may  ...   999         0  nonexistent          1.1
1  yes  telephone   may  ...   999         0  nonexistent          1.1
2   no  telephone   may  ...   999         0  nonexistent          1.1
3   no  telephone   may  ...   999         0  nonexistent          1.1
4   no  telephone   may  ...   999         0  nonexistent          1.1

   cons.price.idx cons.conf.idx euribor3m nr.employed state subscribed
0          93.994         -36.4     4.857        5191   VIC          0
1          93.994         -36.4     4.857        5191    NT          0
2          93.994         -36.4     4.857        5191   TAS          0
3          93.994         -36.4     4.857        5191   NSW          0
4          93.994         -36.4     4.857        5191   ACT          0
```

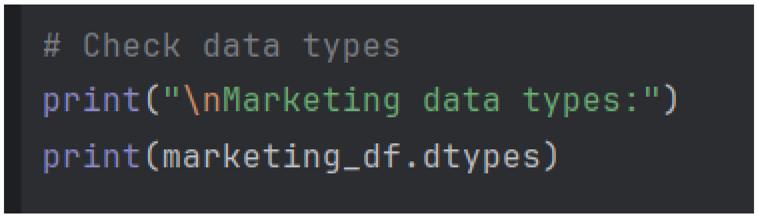
Figure 4 shows the output of the code in Figure 3.

Observations:-

- The dataset contains both numerical (e.g., age, balance) and categorical (e.g., job, education) features.
- The target variable subscribed is binary (0 = "No", 1 = "Yes").

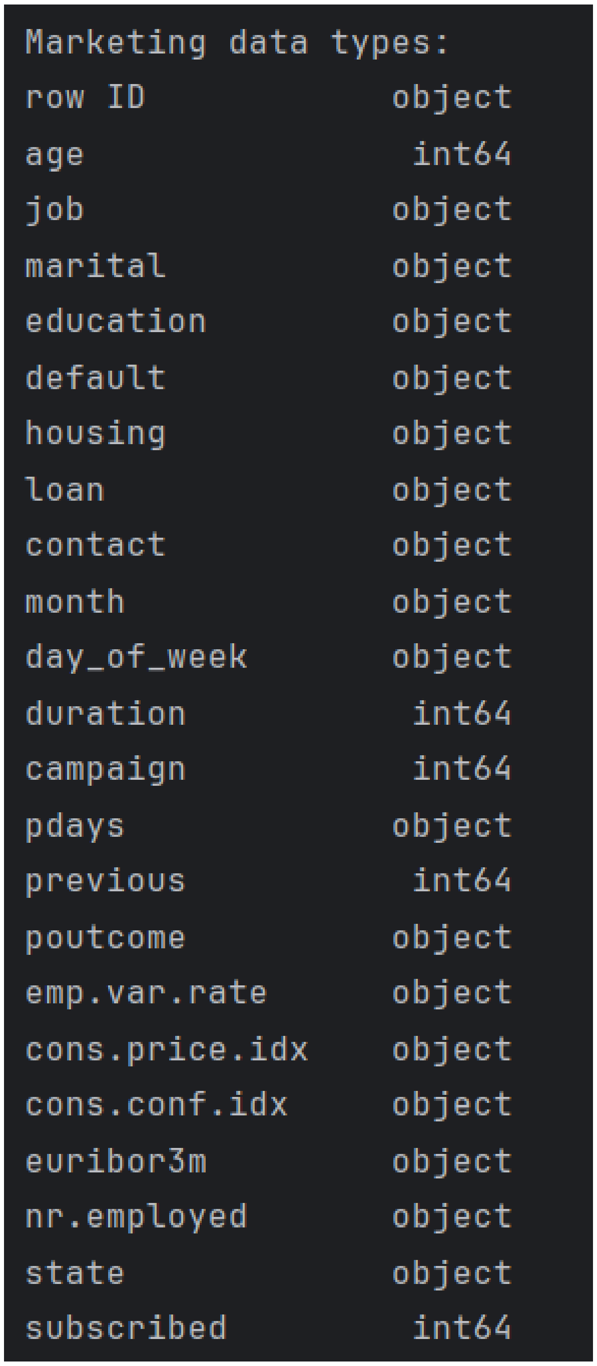
Data Types

Using the dtypes function we found the data types for each attribute in the dataset.



```python
# Check data types
print("\nMarketing data types:")
print(marketing_df.dtypes)
```

Figure 5 shows the code used to find the data types.



```
Marketing data types:
row ID             object
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              object
previous            int64
poutcome           object
emp.var.rate       object
cons.price.idx     object
cons.conf.idx      object
euribor3m          object
nr.employed        object
state              object
subscribed          int64
```

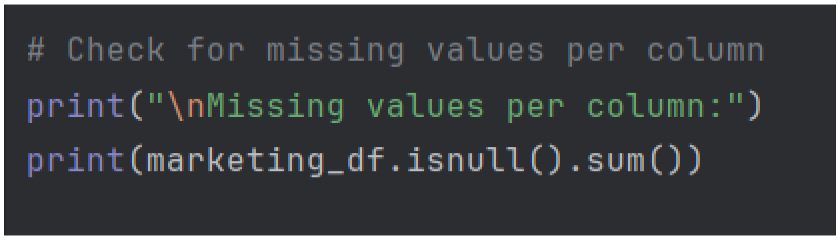
Figure 6 shows the output of the code in Figure 5.

Observations:

- Most attributes are of type object and there are a couple attributes of type integer
- Some “numerical” values such as euribor3m, nr. employed, cons.conf.idx etc are still labelled as an object data type.

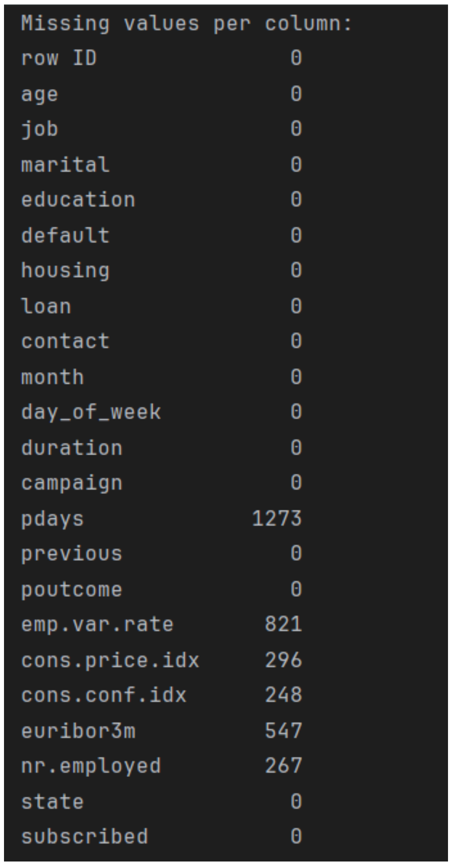
Missing values

Finally for the final part of the data exploration section we checked the number of missing values for every attribute.



```python
# Check for missing values per column
print("\nMissing values per column:")
print(marketing_df.isnull().sum())
```

Figure 7 shows the code used to find missing values.



```
Missing values per column:
row ID               0
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             1273
previous             0
poutcome             0
emp.var.rate       821
cons.price.idx     296
cons.conf.idx      248
euribor3m          547
nr.employed        267
state                0
subscribed           0
```

Figure 8 shows the output regarding the number of missing values.

Observations:-

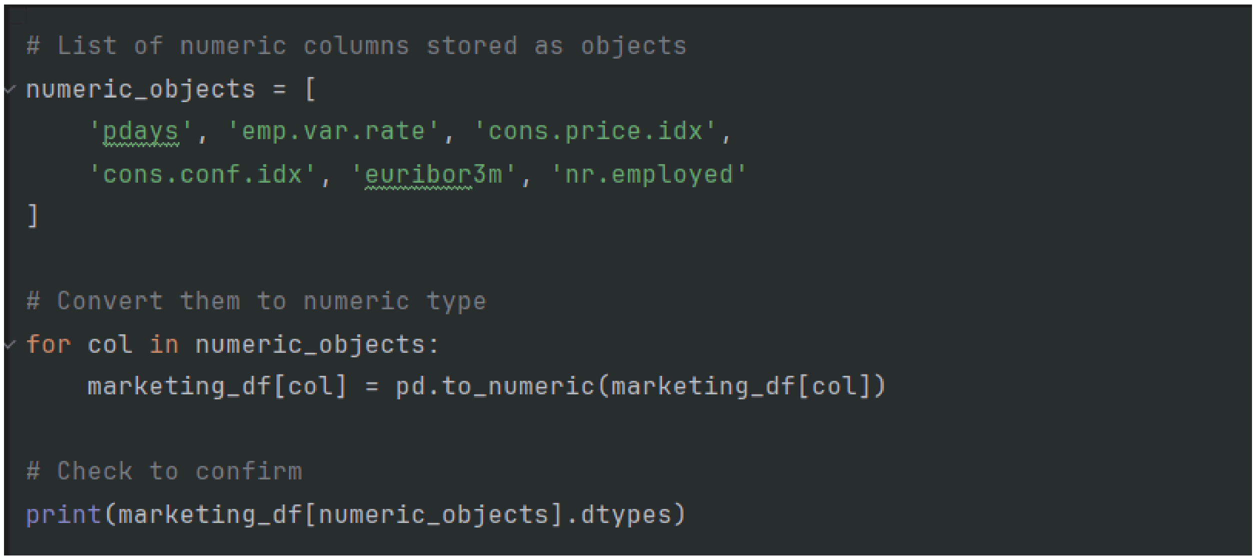
- Most attributes have no missing values and the attributes that do have missing values don’t have an alarming rate of missing values.

- We can see from our output that attributes like default have 0 missing values but when looking at Figure 4 we can see that it has values such as “unknown” which can classify as a missing value.

Data Preprocessing

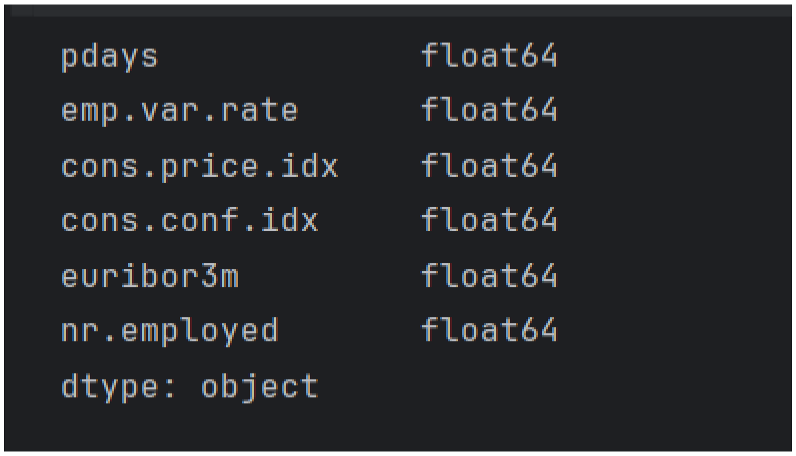
Converting object values to numerical values

Some columns, such as emp.var.rate, pdays, and euribor3m, represented numerical values but were stored as object (string) types. These were converted to numeric (float) using pd.to_numeric() to ensure they could be processed correctly during model training.



```
# List of numeric columns stored as objects
numeric_objects = [
    'pdays', 'emp.var.rate', 'cons.price.idx',
    'cons.conf.idx', 'euribor3m', 'nr.employed'
]

# Convert them to numeric type
for col in numeric_objects:
    marketing_df[col] = pd.to_numeric(marketing_df[col])

# Check to confirm
print(marketing_df[numeric_objects].dtypes)
```

Figure 9 shows the code used to convert attributes from object to float data types.

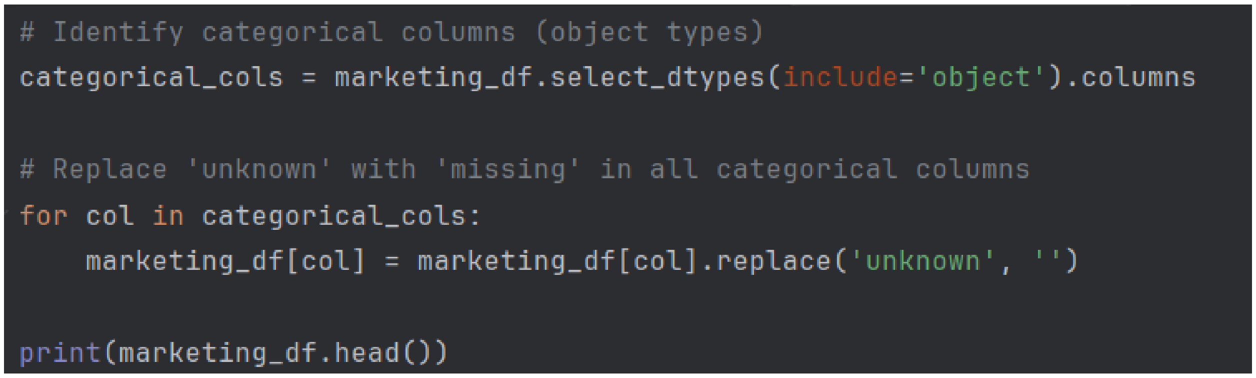


```
pdays             float64
emp.var.rate      float64
cons.price.idx    float64
cons.conf.idx     float64
euribor3m         float64
nr.employed       float64
dtype: object
```

Figure 10 shows the output of the code in Figure 9.

Replacing unknown values with null

As observed before there are some values in the categorical columns which are labelled as “unknown” but won’t be specified as a missing value. To convert this we used the replace() function as seen in Figure 11.



```python
# Identify categorical columns (object types)
categorical_cols = marketing_df.select_dtypes(include='object').columns

# Replace 'unknown' with 'missing' in all categorical columns
for col in categorical_cols:
    marketing_df[col] = marketing_df[col].replace('unknown', '')

print(marketing_df.head())
```

Figure 11 shows the code used to convert “unknown” values into null values.

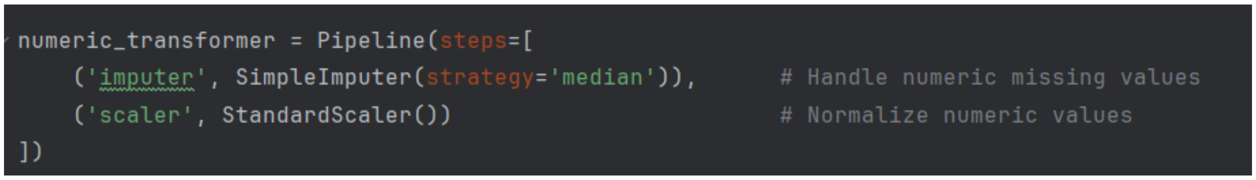
Data cleaning

Separating numerical and categorical features

The features were divided into two groups based on data type: numerical features (e.g., age, duration, euribor3m) and categorical features (e.g., job, education, month). This separation allowed for tailored preprocessing of each group.

Handling Numeric Features

Numerical features were preprocessed using a pipeline that first filled in any missing values with the median of the respective column. This was followed by normalization using StandardScaler, which standardizes each feature to have a mean of 0 and a standard deviation of 1. This step helps algorithms recognize patterns more easily.

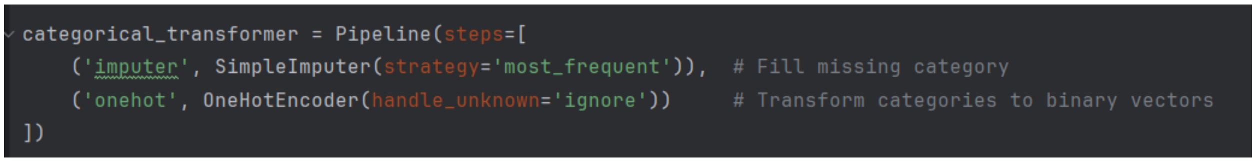


```python
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),      # Handle numeric missing values
    ('scaler', StandardScaler())                        # Normalize numeric values
])
```

Figure 12 shows the code used to handle missing values and normalize numbers.

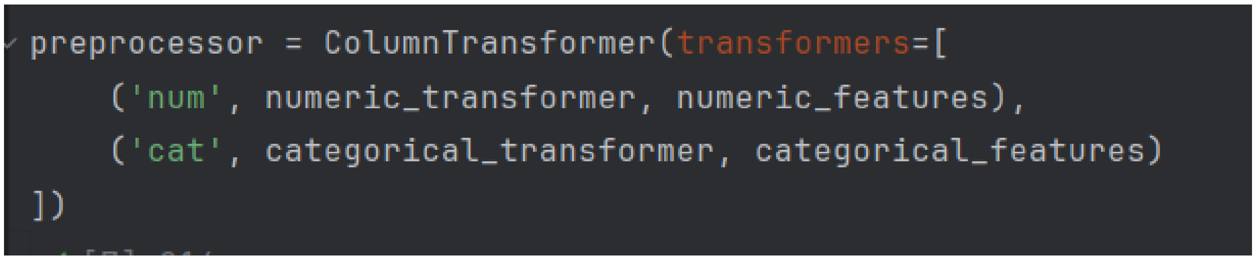
Handling Categorical Features

Categorical features were handled using a separate pipeline. Missing values were imputed using the most frequent value in each column. Then, one-hot encoding was applied using OneHotEncoder, converting each category into a set of binary features. This allows the models to interpret the categorical data numerically.



```
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),  # Fill missing category
    ('onehot', OneHotEncoder(handle_unknown='ignore'))     # Transform categories to binary vectors
])
```

Figure 13 shows the code used to handle categorical features.

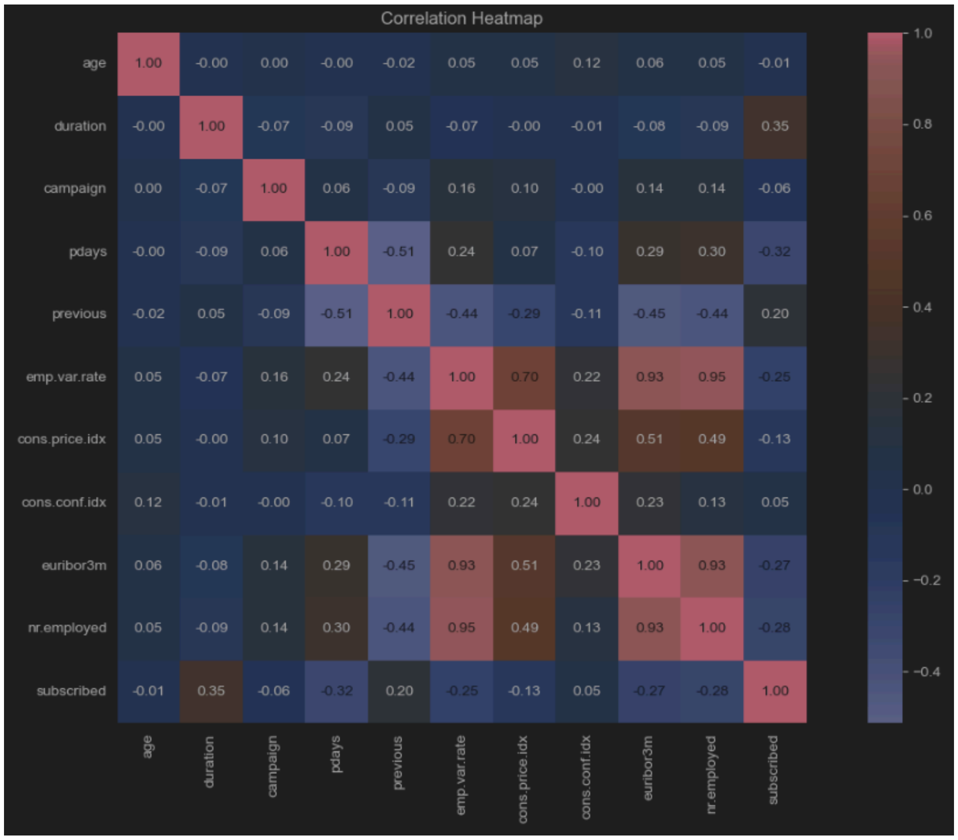


```
preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
])
```

Figure 14 shows the final preprocessor that will be used on the data.

Linear Rank/Correlation

To better understand the relationships between numerical features, a Spearman rank correlation heatmap was generated.



	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	subscribed
age	1.00	-0.00	0.00	-0.00	-0.02	0.05	0.05	0.12	0.06	0.05	-0.01
duration	-0.00	1.00	-0.07	-0.09	0.05	-0.07	-0.00	-0.01	-0.08	-0.09	0.35
campaign	0.00	-0.07	1.00	0.06	-0.09	0.16	0.10	-0.00	0.14	0.14	-0.06
pdays	-0.00	-0.09	0.06	1.00	-0.51	0.24	0.07	-0.10	0.29	0.30	-0.32
previous	-0.02	0.05	-0.09	-0.51	1.00	-0.44	-0.29	-0.11	-0.45	-0.44	0.20
emp.var.rate	0.05	-0.07	0.16	0.24	-0.44	1.00	0.70	0.22	0.93	0.95	-0.25
cons.price.idx	0.05	-0.00	0.10	0.07	-0.29	0.70	1.00	0.24	0.51	0.49	-0.13
cons.conf.idx	0.12	-0.01	-0.00	-0.10	-0.11	0.22	0.24	1.00	0.23	0.13	0.05
euribor3m	0.06	-0.08	0.14	0.29	-0.45	0.93	0.51	0.23	1.00	0.93	-0.27
nr.employed	0.05	-0.09	0.14	0.30	-0.44	0.95	0.49	0.13	0.93	1.00	-0.28
subscribed	-0.01	0.35	-0.06	-0.32	0.20	-0.25	-0.13	0.05	-0.27	-0.28	1.00

Figure 15 shows the correlation heatmap for numeric features.

From the heatmap, several observations were made:

- There is a strong positive correlation between emp.var.rate, euribor3m, and nr.employed, all with values above 0.90. This shows that these features are closely related and may be giving the model very similar information. Keeping all of them could lead to redundancy and affect the model's performance.

- The feature duration had a moderate positive correlation with the target variable subscribed (0.35), which means it could be a strong predictor. However, since duration is only known after the call takes place, it might not be useful in real-time prediction models.

- pdays showed a moderate negative correlation with subscribed (-0.32). This might suggest that customers who were contacted more recently are slightly less likely to subscribe.

- Most of the other features had weak correlations with the target.

How I Went About Solving the Problem

To solve the classification problem, I built five machine learning models Decision Tree, K-Nearest Neighbours, Random Forest, SVM, and Neural Network using pipelines in Python. The pipelines included preprocessing steps like imputation, scaling, and encoding, which ensured consistency across all models.

Parameter Optimization

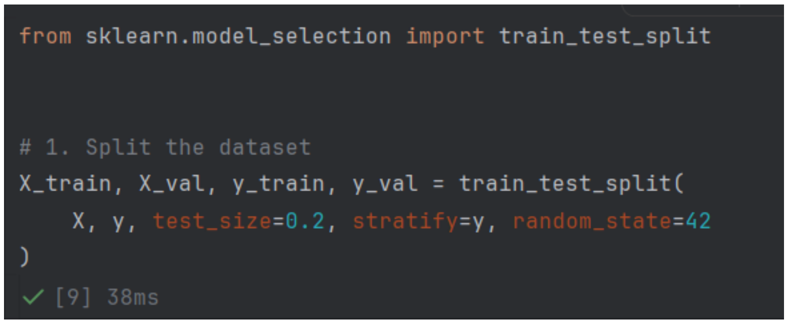
I used GridSearchCV with 5-fold cross-validation to tune each model's hyperparameters. For example, I tested different max_depth and min_samples_split values for Decision Trees, and adjusted C and gamma for SVM. This helped improve performance, especially in terms of F1 score and recall.

Handling Class Imbalance

I applied class_weight='balanced' for models that support it, like Random Forest and SVM. This helped the models treat both classes more equally and improved their ability to detect the minority class (subscribed = 1).

Data Splitting Strategy

I split the dataset into training and validation sets using a stratified 80-20 split to maintain class proportions. All parameter tuning was done using cross-validation within the training set, and final model evaluation was done on the validation set.



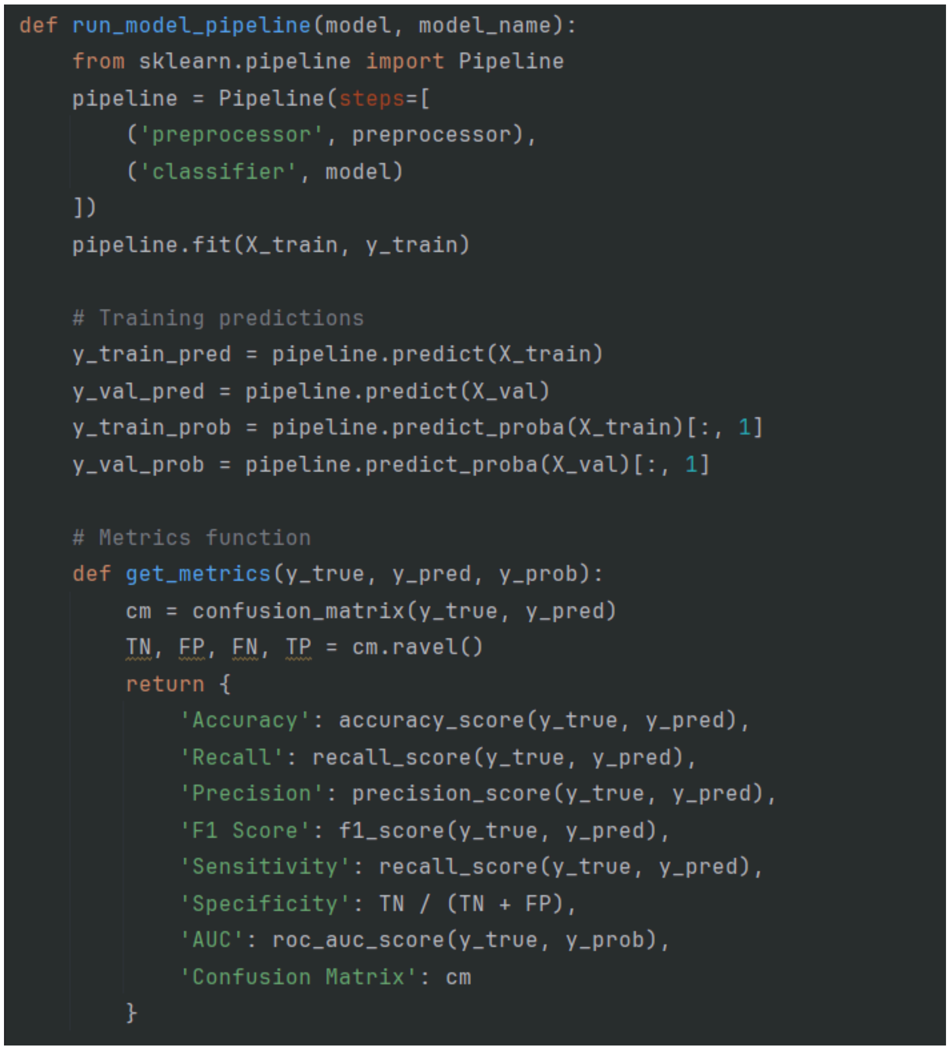
```python
from sklearn.model_selection import train_test_split


# 1. Split the dataset
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)
✓ [9] 38ms
```

Figure 16 shows the code regarding how the data was split into training and validation sets

Building Models

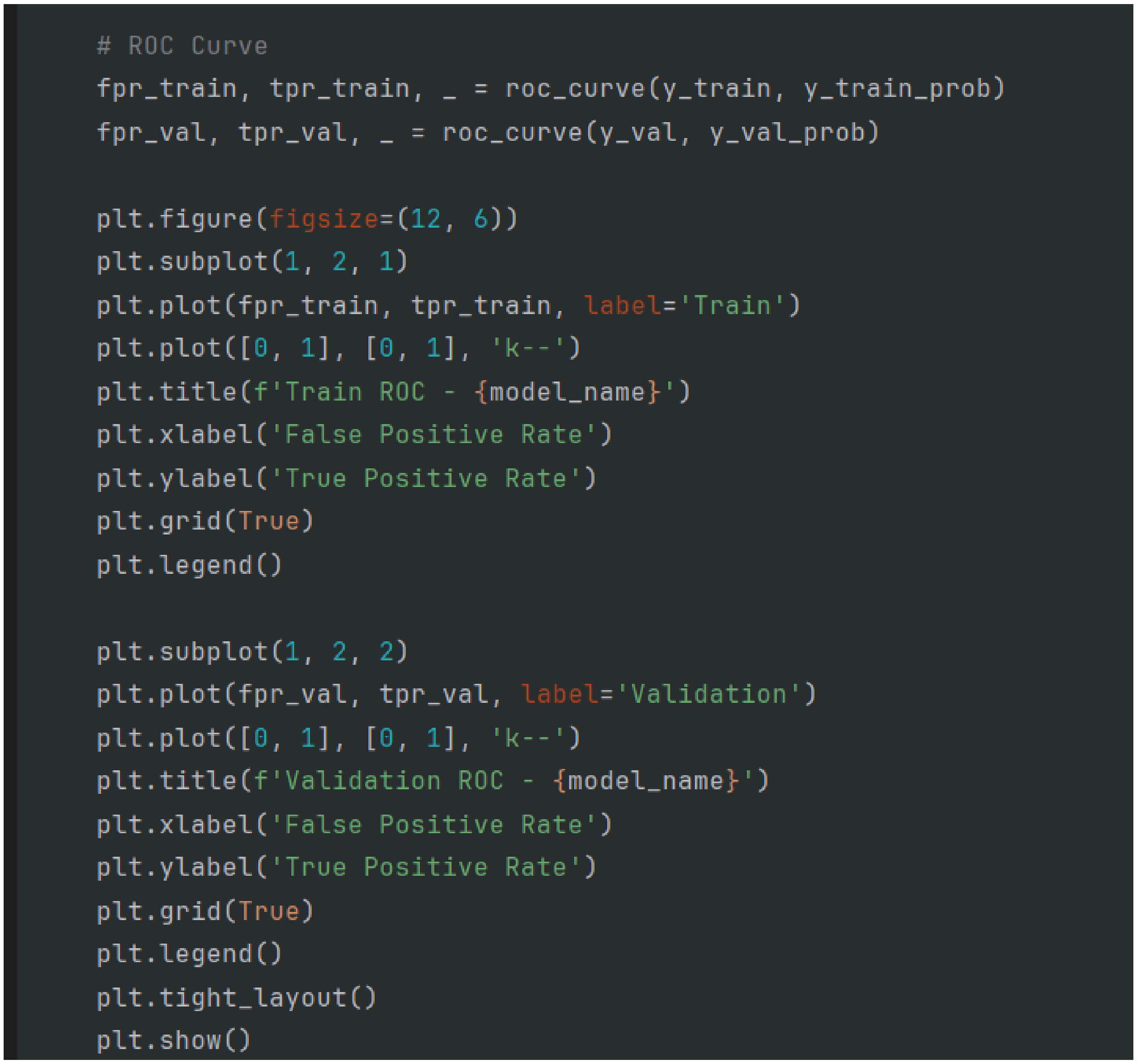
Before building specific machine learning models, I first developed a general function (run_model_pipeline) that outputs essential performance metrics and visualizations. This function prints accuracy, recall, precision, F1 score, sensitivity, specificity, AUC, and confusion matrices for both the training and validation sets. It also plots the ROC curves and generates the .csv file required for Kaggle submission.

Each model was trained using a pipeline that included preprocessing steps such as imputation for missing values, scaling for numeric features, and one-hot encoding for categorical features. For most classifiers, GridSearchCV was used for hyperparameter tuning with 5-fold cross-validation, and F1 score was selected as the evaluation metric to ensure a balance between precision and recall.



```python
def run_model_pipeline(model, model_name):
    from sklearn.pipeline import Pipeline
    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('classifier', model)
    ])
    pipeline.fit(X_train, y_train)

    # Training predictions
    y_train_pred = pipeline.predict(X_train)
    y_val_pred = pipeline.predict(X_val)
    y_train_prob = pipeline.predict_proba(X_train)[:, 1]
    y_val_prob = pipeline.predict_proba(X_val)[:, 1]

    # Metrics function
    def get_metrics(y_true, y_pred, y_prob):
        cm = confusion_matrix(y_true, y_pred)
        TN, FP, FN, TP = cm.ravel()
        return {
            'Accuracy': accuracy_score(y_true, y_pred),
            'Recall': recall_score(y_true, y_pred),
            'Precision': precision_score(y_true, y_pred),
            'F1 Score': f1_score(y_true, y_pred),
            'Sensitivity': recall_score(y_true, y_pred),
            'Specificity': TN / (TN + FP),
            'AUC': roc_auc_score(y_true, y_prob),
            'Confusion Matrix': cm
        }
```

Figure 17 shows the first part of the code for the run_model function which is used to get all the necessary metrics



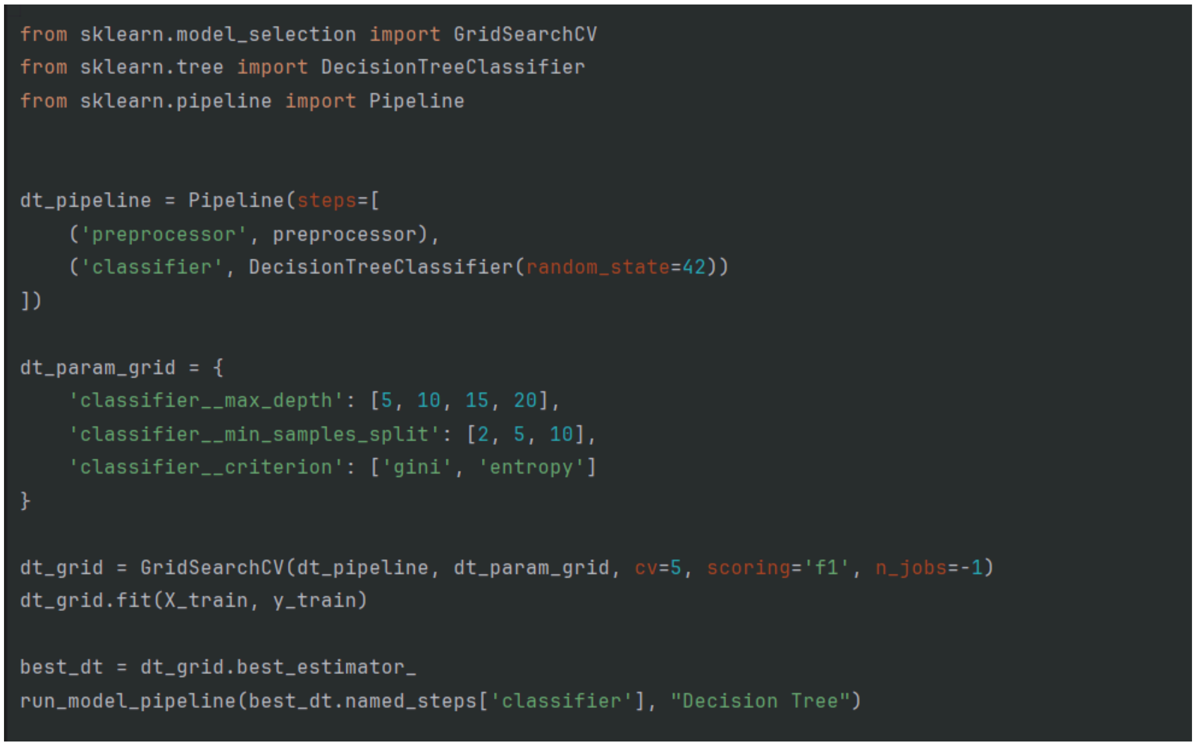
```
    # ROC Curve
    fpr_train, tpr_train, _ = roc_curve(y_train, y_train_prob)
    fpr_val, tpr_val, _ = roc_curve(y_val, y_val_prob)

    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(fpr_train, tpr_train, label='Train')
    plt.plot([0, 1], [0, 1], 'k--')
    plt.title(f'Train ROC - {model_name}')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.grid(True)
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(fpr_val, tpr_val, label='Validation')
    plt.plot([0, 1], [0, 1], 'k--')
    plt.title(f'Validation ROC - {model_name}')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()
```

Figure 18 shows the second part of the run_model function that is used to plot the ROC curve

Decision Tree

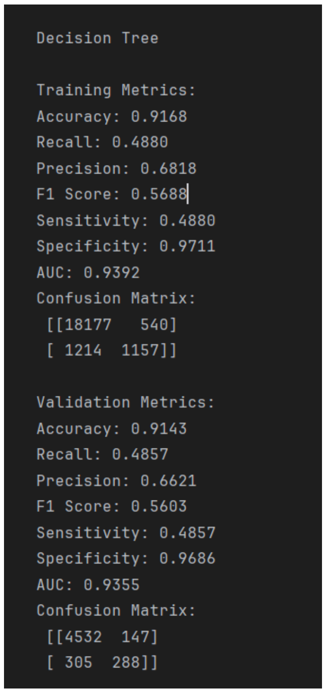
The Decision Tree classifier was built using Python's sklearn.tree.DecisionTreeClassifier. I used GridSearchCV to tune the model's key hyperparameters: max_depth, min_samples_split, and criterion, with F1 score as the evaluation metric and 5-fold cross-validation.

This setup helped find the best-performing version of the model while reducing overfitting. The final model was then evaluated on the validation set using metrics such as accuracy, recall, precision, and AUC.



```python
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline


dt_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', DecisionTreeClassifier(random_state=42))
])

dt_param_grid = {
    'classifier__max_depth': [5, 10, 15, 20],
    'classifier__min_samples_split': [2, 5, 10],
    'classifier__criterion': ['gini', 'entropy']
}

dt_grid = GridSearchCV(dt_pipeline, dt_param_grid, cv=5, scoring='f1', n_jobs=-1)
dt_grid.fit(X_train, y_train)

best_dt = dt_grid.best_estimator_
run_model_pipeline(best_dt.named_steps['classifier'], "Decision Tree")
```

Figure 19 shows the code for creating the decision tree classifier



```
Decision Tree

Training Metrics:
Accuracy: 0.9168
Recall: 0.4880
Precision: 0.6818
F1 Score: 0.5688
Sensitivity: 0.4880
Specificity: 0.9711
AUC: 0.9392
Confusion Matrix:
 [[18177   540]
 [ 1214  1157]]

Validation Metrics:
Accuracy: 0.9143
Recall: 0.4857
Precision: 0.6621
F1 Score: 0.5603
Sensitivity: 0.4857
Specificity: 0.9686
AUC: 0.9355
Confusion Matrix:
 [[4532  147]
 [ 305  288]]
```

Figure 20 shows the metrics and confusion matrix output for the decision tree classifier

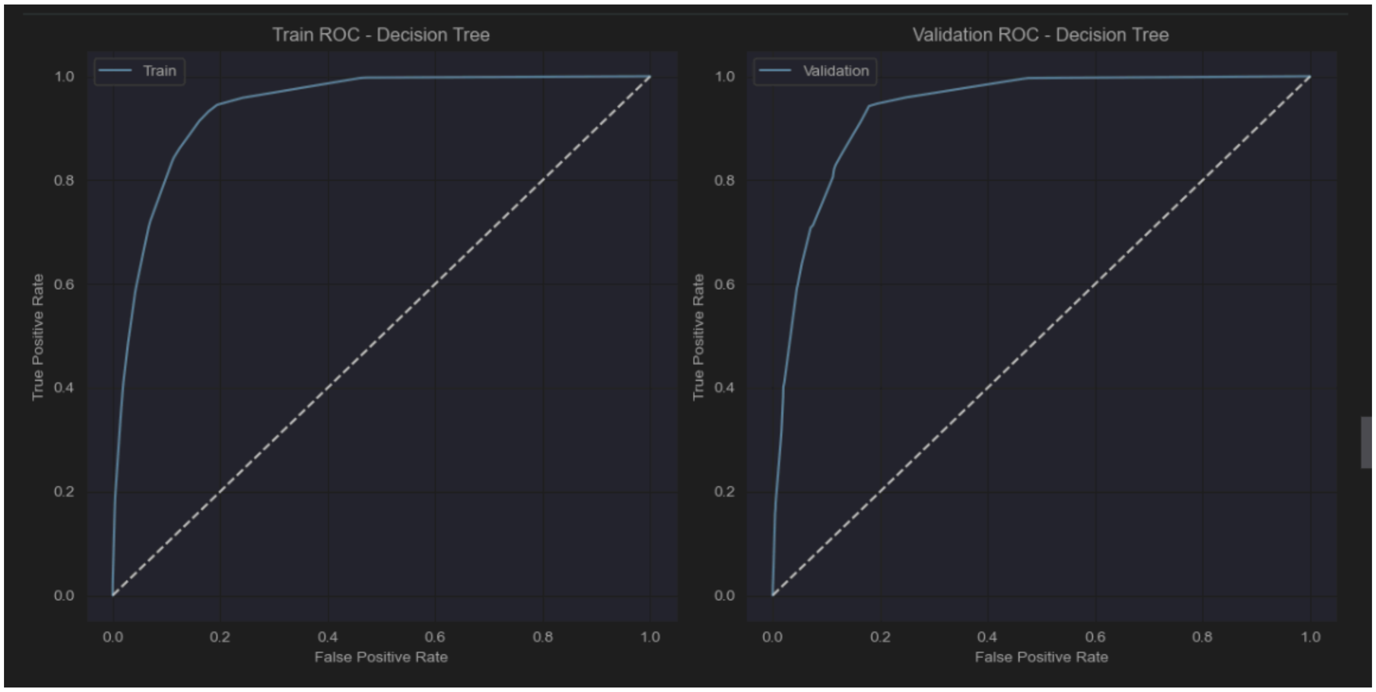


Figure 21 shows the ROC curve for both the training set and the validation set

	Accuracy	Recall	Precision	F1 Score	Sensitivity	Specificity	AUC
Training	0.9168	0.488	0.6818	0.5688	0.488	0.9711	0.9392
Validation	0.9143	0.4857	0.6621	0.5603	0.4857	0.9686	0.9355

The model achieved high accuracy and AUC on both sets, indicating good discriminative power. However, the relatively lower recall suggests the model is more conservative in predicting the positive class (i.e., fewer false positives, but more false negatives). Specificity is high, meaning the model is very accurate when predicting the negative class.

ROC Curve Analysis

The ROC curves for both training and validation indicate strong model performance, with both curves bending sharply toward the top-left corner a sign of good separation between classes. The small gap between training and validation AUC (0.9392 vs. 0.9355) suggests the model generalizes well and is not heavily overfitting.

Parameter Optimization

The Decision Tree model was optimized using GridSearchCV with 5-fold cross-validation. The tuning was based on F1 score to ensure a good balance between precision and recall. The following parameters were tested:

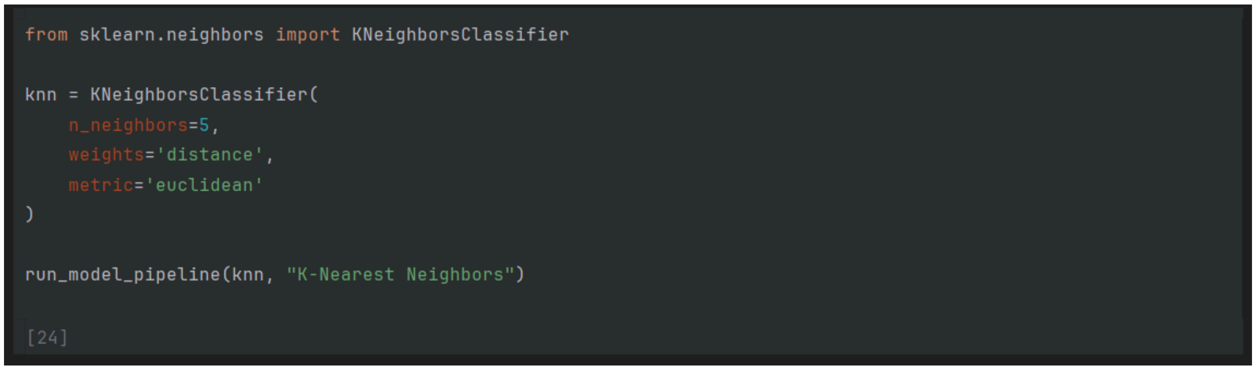
- **max_depth**: [5, 10, 15, 20]
  This controls the maximum depth of the tree. Limiting tree depth helps prevent overfitting by restricting how many times the tree can split.

- **min_samples_split**: [2, 5, 10]
  This sets the minimum number of samples required to split a node. Increasing this value can make the model more general by preventing it from creating splits based on very few samples.

- **criterion**: ['gini', 'entropy']
  This defines how the quality of a split is measured. gini uses Gini Impurity, and entropy uses Information Gain. Both were tested to determine which produced the best separation between classes

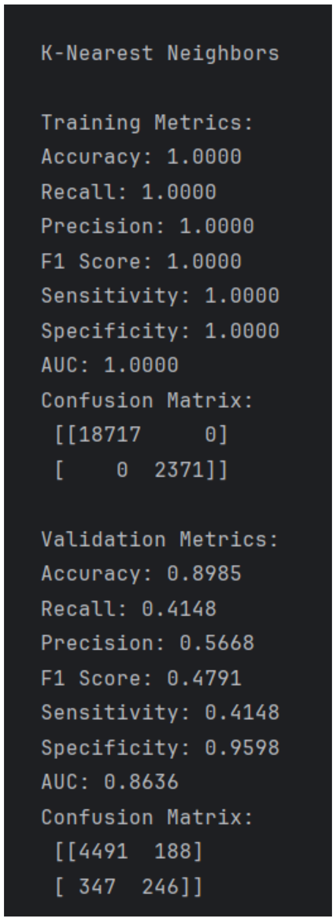
K-Nearest Neighbor (KNN)

The K-Nearest Neighbours classifier was implemented using sklearn.neighbors.KNeighborsClassifier. Initially, I attempted to optimize KNN using GridSearchCV with cross-validation to tune n_neighbors, weights, and distance metrics. However, the grid search process was computationally intensive and did not lead to significant performance improvements. As a result, I chose to proceed with a simpler configuration using n_neighbors=5, weights='distance', and metric='euclidean', which offered reasonable performance and reduced execution time.



```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(
    n_neighbors=5,
    weights='distance',
    metric='euclidean'
)

run_model_pipeline(knn, "K-Nearest Neighbors")

[24]
```

Figure 22 shows the code for K-nearest neighbor classifier



```
K-Nearest Neighbors

Training Metrics:
Accuracy: 1.0000
Recall: 1.0000
Precision: 1.0000
F1 Score: 1.0000
Sensitivity: 1.0000
Specificity: 1.0000
AUC: 1.0000
Confusion Matrix:
 [[18717     0]
 [    0  2371]]

Validation Metrics:
Accuracy: 0.8985
Recall: 0.4148
Precision: 0.5668
F1 Score: 0.4791
Sensitivity: 0.4148
Specificity: 0.9598
AUC: 0.8636
Confusion Matrix:
 [[4491  188]
 [ 347  246]]
```

Figure 23 shows the metrics and confusion matrix output for the K-Nearest Neighbour classifier

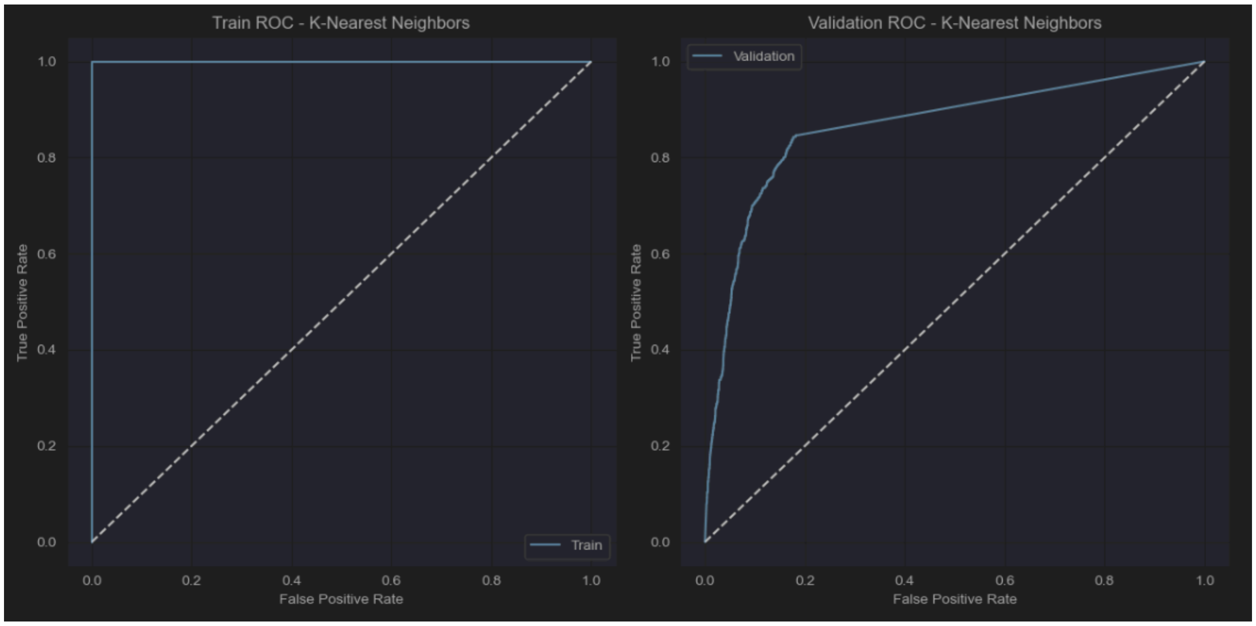


Figure 24 shows the ROC curve for both training and validation sets for the KNN classifier

	Accuracy	Recall	Precision	F1 Score	Sensitivity	Specificity	AUC
Training	1	1	1	1	1	1	1
Validation	0.8985	0.4148	0.5668	0.4791	0.4148	0.9598	0.8636

The model achieved perfect accuracy, recall, precision, and AUC on the training set, which strongly suggests overfitting. On the validation set, accuracy and AUC remained high, showing that the model still had good discriminative power overall. However, the relatively low recall indicates that the model was not very effective at correctly identifying positive cases. Specificity remained high, showing that it was very accurate at predicting the negative class.

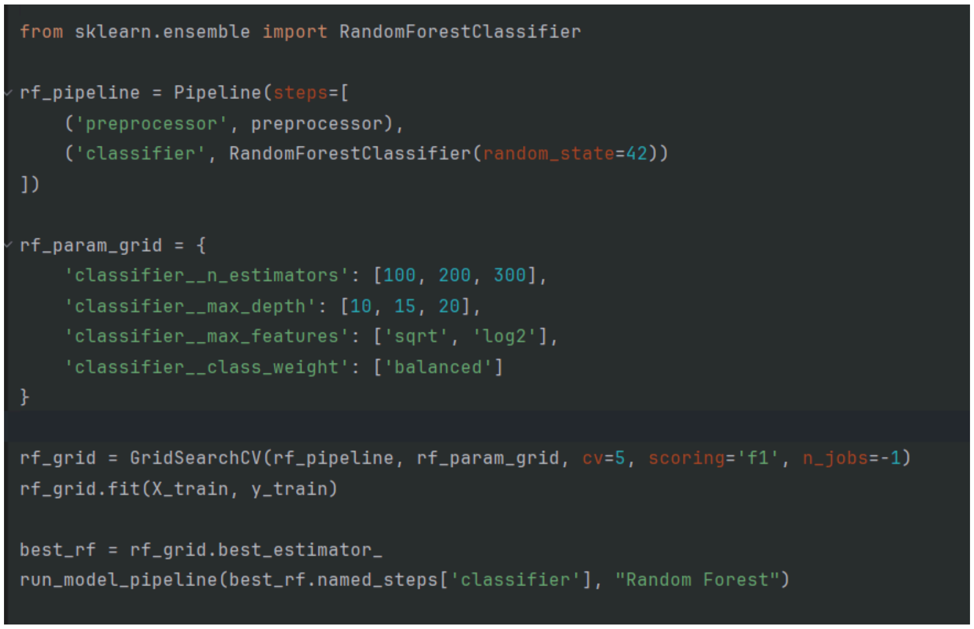
ROC Curve Analysis

The ROC curve for the training set shows a perfect separation of classes, which again points to overfitting. The validation ROC curve still curves well toward the top-left corner, indicating decent class separation. The gap between training and validation AUC (1.0000 vs. 0.8636) is large, confirming that the model does not generalize well to unseen data.

Random Forest (RF)

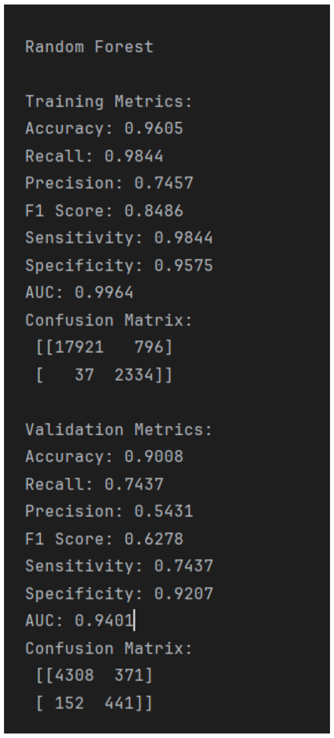
The Random Forest classifier was built using Python's sklearn.ensemble.RandomForestClassifier. I used GridSearchCV to tune the model's key hyperparameters: n_estimators, max_depth, max_features, and class_weight, with F1 score as the evaluation metric and 5-fold cross-validation.

This setup helped select the most effective model configuration for balancing bias and variance, especially with imbalanced class labels. The final model was then evaluated on the validation set using metrics such as accuracy, recall, precision, and AUC.



```python
from sklearn.ensemble import RandomForestClassifier

rf_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(random_state=42))
])

rf_param_grid = {
    'classifier__n_estimators': [100, 200, 300],
    'classifier__max_depth': [10, 15, 20],
    'classifier__max_features': ['sqrt', 'log2'],
    'classifier__class_weight': ['balanced']
}

rf_grid = GridSearchCV(rf_pipeline, rf_param_grid, cv=5, scoring='f1', n_jobs=-1)
rf_grid.fit(X_train, y_train)

best_rf = rf_grid.best_estimator_
run_model_pipeline(best_rf.named_steps['classifier'], "Random Forest")
```

Figure 25 shows the code for the RF classifier



```
Random Forest

Training Metrics:
Accuracy: 0.9605
Recall: 0.9844
Precision: 0.7457
F1 Score: 0.8486
Sensitivity: 0.9844
Specificity: 0.9575
AUC: 0.9964
Confusion Matrix:
 [[17921   796]
 [   37  2334]]

Validation Metrics:
Accuracy: 0.9008
Recall: 0.7437
Precision: 0.5431
F1 Score: 0.6278
Sensitivity: 0.7437
Specificity: 0.9207
AUC: 0.9401
Confusion Matrix:
 [[4308  371]
 [ 152  441]]
```

Figure 26 shows the metrics and confusion matrix for the RF classifier

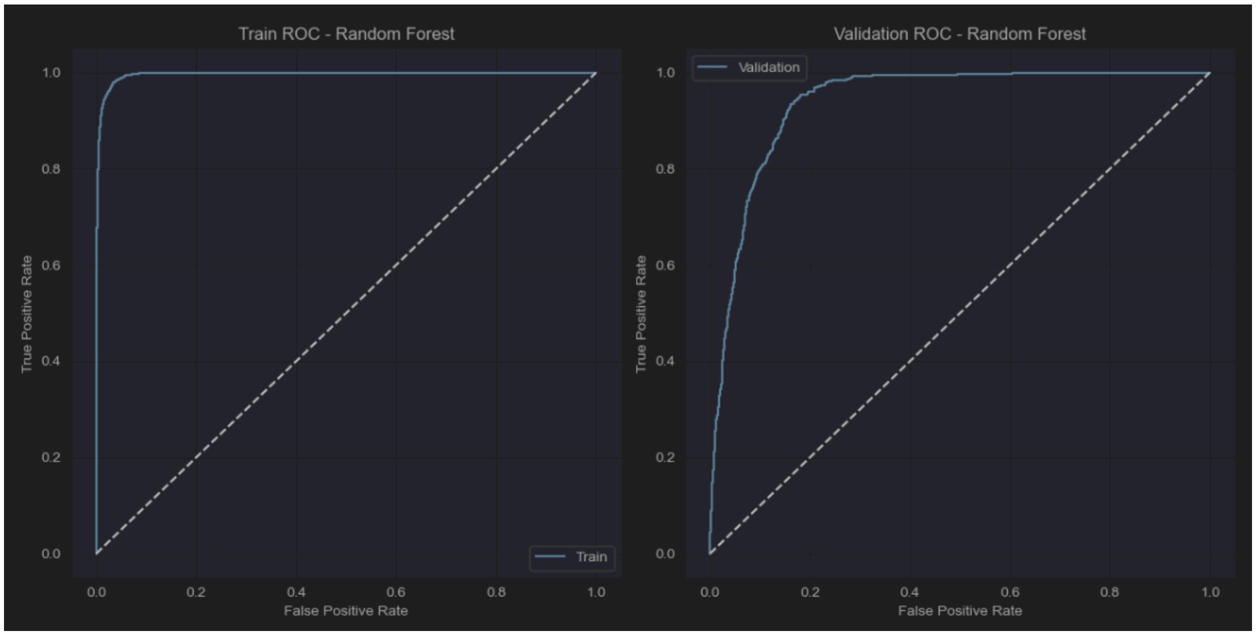


Figure 27 shows the ROC curve for both training and validation sets for the RF classifier

	Accuracy	Recall	Precision	F1 Score	Sensitivity	Specificity	AUC
Training	0.9605	0.9844	0.7457	0.8486	0.9844	0.9575	0.9964
Validation	0.9008	0.7437	0.5431	0.6278	0.7437	0.9207	0.9401

The model performed strongly on both training and validation sets, achieving high accuracy and AUC, which indicates solid discriminative ability. The recall was very high on the training set but noticeably lower on validation, suggesting the model slightly overfits by capturing patterns too closely in training. Specificity remained high across both sets, confirming the model's strength in correctly predicting the negative class.

ROC Curve Analysis

Both ROC curves curve sharply toward the top-left corner, showing clear class separation. The AUC dropped from 0.9964 (training) to 0.9401 (validation), which is still a strong result and shows that the model generalizes fairly well with only mild overfitting.

Parameter Optimization

The Random Forest model was optimized using GridSearchCV with 5-fold cross-validation and F1 score as the evaluation metric. The goal was to balance recall and precision while handling class imbalance. The following parameters were tuned:

- n_estimators: [100, 200, 300]
  Sets the number of trees in the forest. More trees generally improve accuracy but require more computation.

- max_depth: [10, 15, 20]
  Controls the maximum depth of each tree. Limiting this helps prevent overfitting by restricting how deep the model can grow.

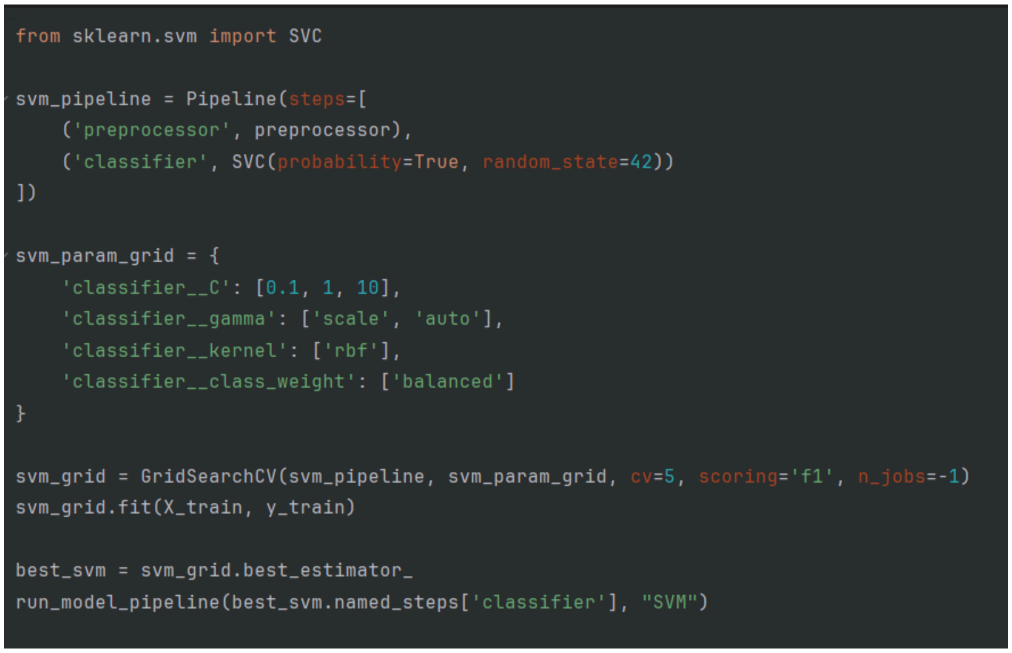
- max_features: ['sqrt', 'log2']
  Determines how many features to consider when looking for the best split, adding randomness to improve generalization.

- class_weight: ['balanced']
  Automatically adjusts weights inversely to class frequencies to address class imbalance during training.

Support Vector Machine (SVM)

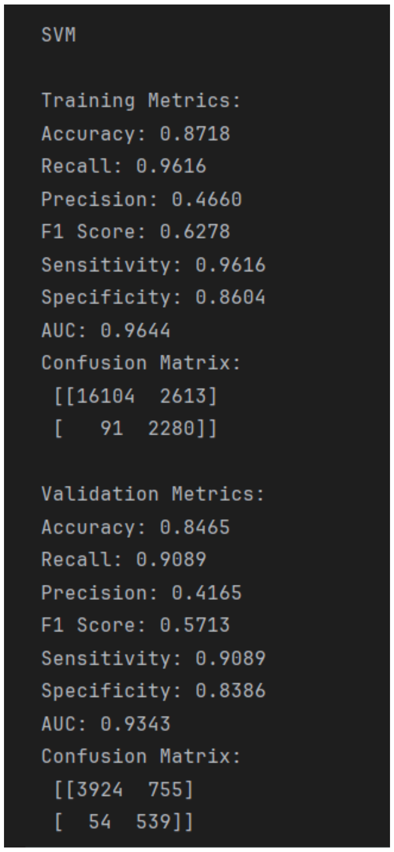
The Support Vector Machine classifier was built using Python's sklearn.svm.SVC. I used GridSearchCV to tune the model's key hyperparameters: C, gamma, kernel, and class_weight, with F1 score as the evaluation metric and 5-fold cross-validation.

This setup allowed the model to adjust its decision boundary for better class separation while handling class imbalance. The final model was then evaluated on the validation set using metrics such as accuracy, recall, precision, and AUC.



```python
from sklearn.svm import SVC

svm_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', SVC(probability=True, random_state=42))
])

svm_param_grid = {
    'classifier__C': [0.1, 1, 10],
    'classifier__gamma': ['scale', 'auto'],
    'classifier__kernel': ['rbf'],
    'classifier__class_weight': ['balanced']
}

svm_grid = GridSearchCV(svm_pipeline, svm_param_grid, cv=5, scoring='f1', n_jobs=-1)
svm_grid.fit(X_train, y_train)

best_svm = svm_grid.best_estimator_
run_model_pipeline(best_svm.named_steps['classifier'], "SVM")
```

Figure 28 shows the code for the SVM classifier



```
SVM

Training Metrics:
Accuracy: 0.8718
Recall: 0.9616
Precision: 0.4660
F1 Score: 0.6278
Sensitivity: 0.9616
Specificity: 0.8604
AUC: 0.9644
Confusion Matrix:
 [[16104  2613]
 [   91  2280]]

Validation Metrics:
Accuracy: 0.8465
Recall: 0.9089
Precision: 0.4165
F1 Score: 0.5713
Sensitivity: 0.9089
Specificity: 0.8386
AUC: 0.9343
Confusion Matrix:
 [[3924  755]
 [  54  539]]
```

Figure 29 shows the metrics and confusion matrix for the output the SVM classifier

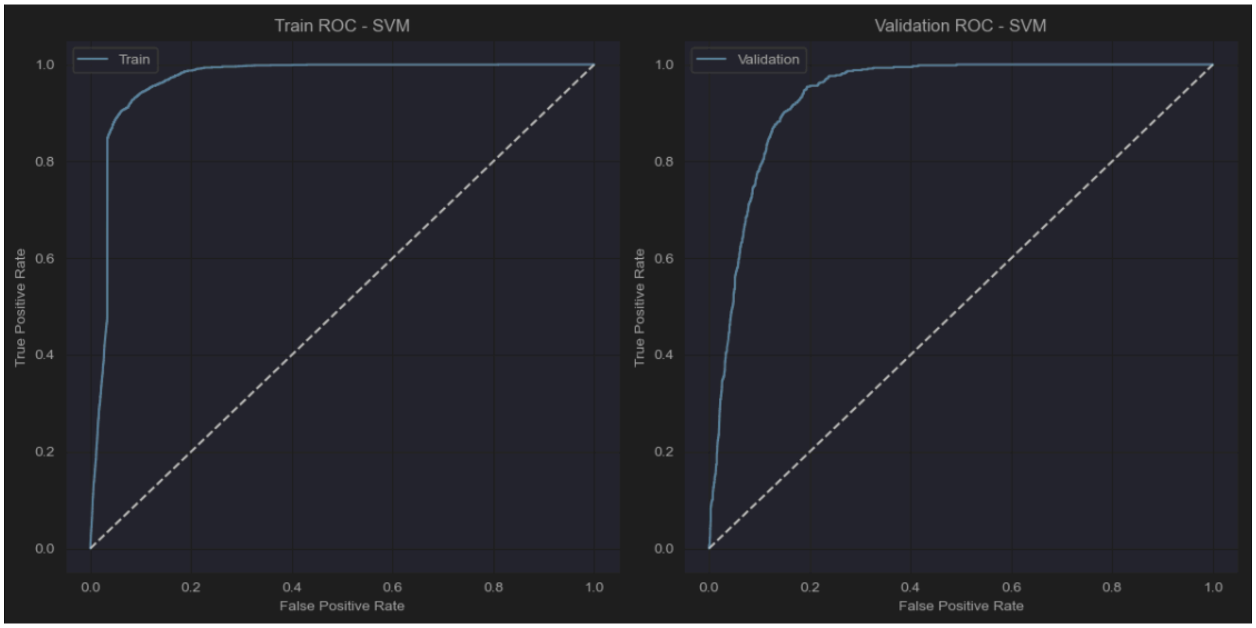


Figure 30 shows the ROC curve for the training and validation sets for the SVM classifier

	Accuracy	Recall	Precision	F1 Score	Sensitivity	Specificity	AUC
Training	0.8718	0.9616	0.4660	0.6278	0.9616	0.8604	0.9644
Validation	0.8465	0.9809	0.4163	0.5713	0.9809	0.8386	0.9343

The model showed strong performance across both datasets, with high AUC and recall, indicating good ability to detect positive cases. While the precision was lower, the high sensitivity suggests that the model prioritized capturing as many true positives as possible. Specificity remained solid, meaning it also handled negative predictions well. The validation accuracy was slightly lower than training, but still respectable.

ROC Curve Analysis

Both training and validation ROC curves show excellent separation, bending well toward the top-left. The AUC only dropped slightly from 0.9644 to 0.9343, showing that the model generalizes well with minimal overfitting.

Parameter Optimization

The SVM classifier was optimized using GridSearchCV with 5-fold cross-validation and F1 score as the selection metric. The following hyperparameters were explored:

- C: [0.1, 1, 10]
  Controls the regularization strength. Smaller values lead to simpler models, while larger ones aim for higher accuracy.

- gamma: ['scale', 'auto']
  Affects the shape of the decision boundary in the RBF kernel. scale adapts based on feature variance.

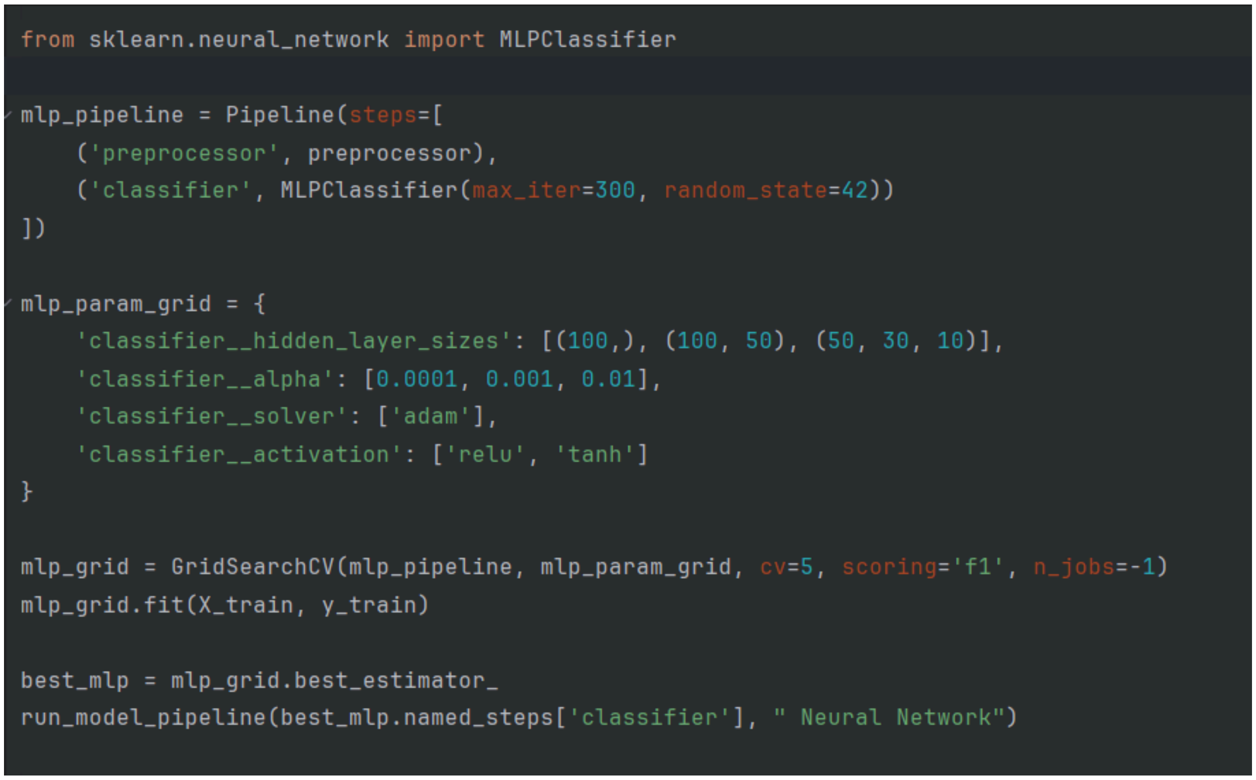
- kernel: ['rbf']
  The RBF kernel was chosen for its ability to model non-linear relationships between features.

- class_weight: ['balanced']
  Adjusts weights inversely proportional to class frequencies to handle imbalance in the dataset.

Neural Network (NN)

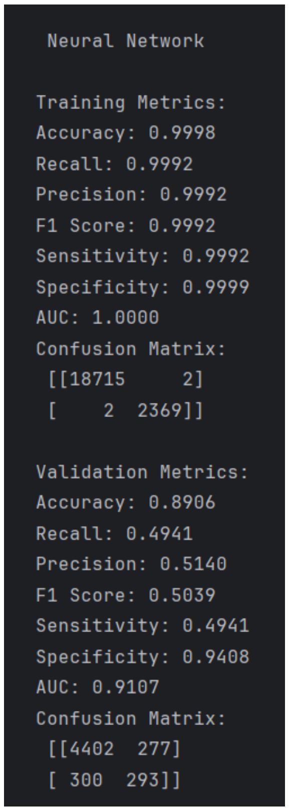
The Neural Network classifier was built using Python's sklearn.neural_network.MLPClassifier. I used GridSearchCV to tune the model's key hyperparameters: hidden_layer_sizes, alpha, solver, and activation, with F1 score as the evaluation metric and 5-fold cross-validation.

This approach helped optimize the network's architecture and regularization to improve learning without overfitting. The final model was then evaluated on the validation set using metrics such as accuracy, recall, precision, and AUC.



```python
from sklearn.neural_network import MLPClassifier

mlp_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', MLPClassifier(max_iter=300, random_state=42))
])

mlp_param_grid = {
    'classifier__hidden_layer_sizes': [(100,), (100, 50), (50, 30, 10)],
    'classifier__alpha': [0.0001, 0.001, 0.01],
    'classifier__solver': ['adam'],
    'classifier__activation': ['relu', 'tanh']
}

mlp_grid = GridSearchCV(mlp_pipeline, mlp_param_grid, cv=5, scoring='f1', n_jobs=-1)
mlp_grid.fit(X_train, y_train)

best_mlp = mlp_grid.best_estimator_
run_model_pipeline(best_mlp.named_steps['classifier'], " Neural Network")
```

Figure 31 shows the code for the neural network classifier



```
 Neural Network

Training Metrics:
Accuracy: 0.9998
Recall: 0.9992
Precision: 0.9992
F1 Score: 0.9992
Sensitivity: 0.9992
Specificity: 0.9999
AUC: 1.0000
Confusion Matrix:
 [[18715     2]
 [    2  2369]]

Validation Metrics:
Accuracy: 0.8906
Recall: 0.4941
Precision: 0.5140
F1 Score: 0.5039
Sensitivity: 0.4941
Specificity: 0.9408
AUC: 0.9107
Confusion Matrix:
 [[4402  277]
 [ 300  293]]
```

Figure 32 shows the output of the metrics and the confusion matrix for the Neural Network classifier

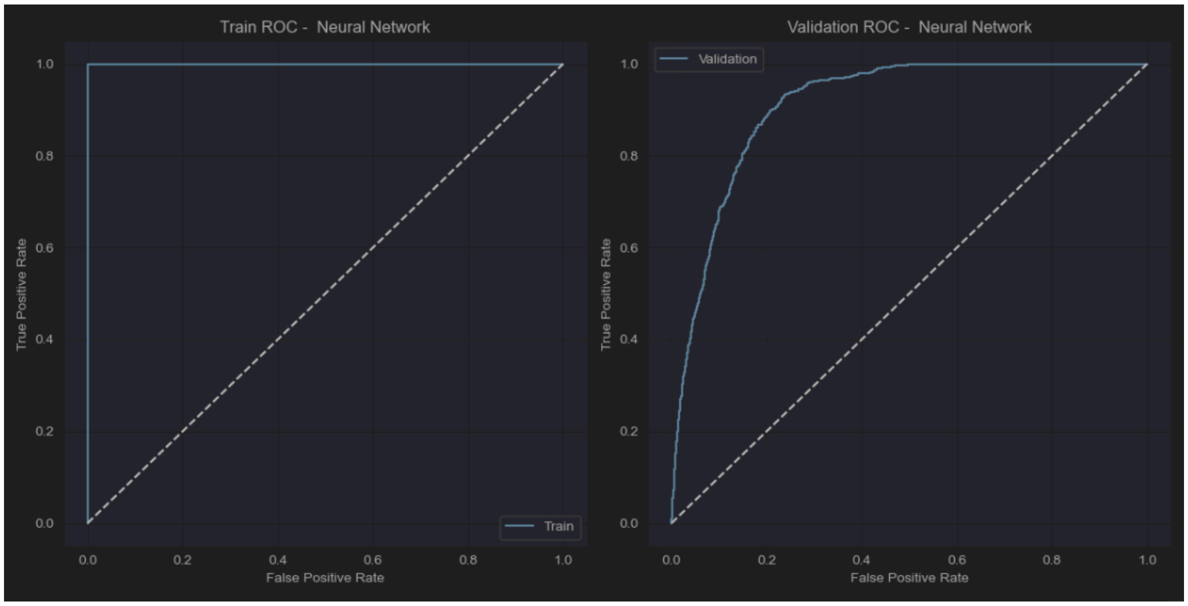


Figure 33 shows the ROC curve for both training and validation sets for the NN classifier

	Accuracy	Recall	Precision	F1 Score	Sensitivity	Specificity	AUC
Training	0.9998	0.9992	0.9992	0.9992	0.9992	0.9999	1
Validation	0.8901	0.4941	0.5140	0.5309	0.4941	0.9408	0.9107

The model performed very well on the training set, with extremely high accuracy, recall, and AUC. However, its performance dropped on the validation set, especially in recall and F1 score. This suggests the model overfitted the training data and didn't generalize as well. Specificity remained high on both sets, meaning it was still good at predicting the negative class.

ROC Curve Analysis

Both ROC curves curve sharply toward the top-left corner, showing strong class separation. The AUC dropped from 1.0000 (training) to 0.9107 (validation), which is still a good result. However, the drop shows the model doesn't perform as well on new data.

Parameter Optimization

The Neural Network was tuned using GridSearchCV with 5-fold cross-validation and F1 score as the main metric. The following parameters were tested:

- hidden_layer_sizes: [(100,), (100, 50), (50, 30, 10)]
  Controls the size and depth of the network. Deeper layers help learn complex patterns but can overfit.
- alpha: [0.0001, 0.001, 0.01]
  Used to reduce overfitting by adding regularization.
- solver: ['adam']
  An optimizer that works well with most datasets.
- activation: ['relu', 'tanh']
  Decides how the network learns patterns. ReLU is faster, while tanh can help in some cases.

Summary of All Model Results

Model	Accuracy	Recall	Precision	F1 Score	Sensitivity	Specificity	AUC
Decision Tree	0.9143	0.4857	0.6621	0.5603	0.4857	0.9686	0.9355
KNN	0.8985	0.4148	0.5668	0.4791	0.4148	0.9598	0.8636
Random Forest	0.9008	0.7437	0.5431	0.6278	0.7437	0.9207	0.9401
SVM	0.8465	0.9809	0.4163	0.5713	0.9809	0.8386	0.9343
Neural Network	0.8906	0.4941	0.5140	0.5039	0.4941	0.9408	0.9107

Best Classifier Selection

After testing all five classifiers, the Random Forest model was selected as the best-performing classifier. It achieved a strong balance between accuracy (0.9008), recall (0.7437), and F1 score (0.6278), while maintaining a high AUC of 0.9401. The recall was especially important in this task because the goal was to correctly identify customers likely to subscribe, and a higher recall means fewer false negatives.

The model also handled class imbalance effectively using `class_weight='balanced'` and benefited from hyperparameter tuning through `GridSearchCV`. These steps helped reduce overfitting and improve generalization.

Although the SVM had the highest recall (0.9809), its lower precision (0.4163) and F1 score (0.5713) meant it was too aggressive in predicting positive cases, leading to more false positives. On the other hand, Decision Tree and KNN showed lower recall and F1 scores, and the Neural Network overfitted the training data, which led to a drop in validation performance.

Therefore, Random Forest was the most reliable model overall, showing solid performance across all key metrics and generalizing well to unseen data.

Testing Score of All Classifiers on Kaggle Using Unknown Dataset

The following table shows the best F1 scores obtained on Kaggle for each classifier using the unknown test dataset:

Model	Kaggle F1 Score
Decision Tree	0.55172
K-Nearest Neighbors	0.45757
Random Forest	0.65000
SVM	0.58361
Neural Network	0.49734

Best Classifier Result on Kaggle

The best-performing classifier on Kaggle was the Random Forest, achieving a highest F1 score of 0.65000. This aligns with the validation results where Random Forest had the best overall balance between recall, precision, and AUC. It was able to consistently outperform other models.

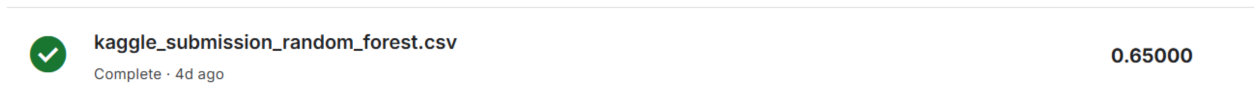
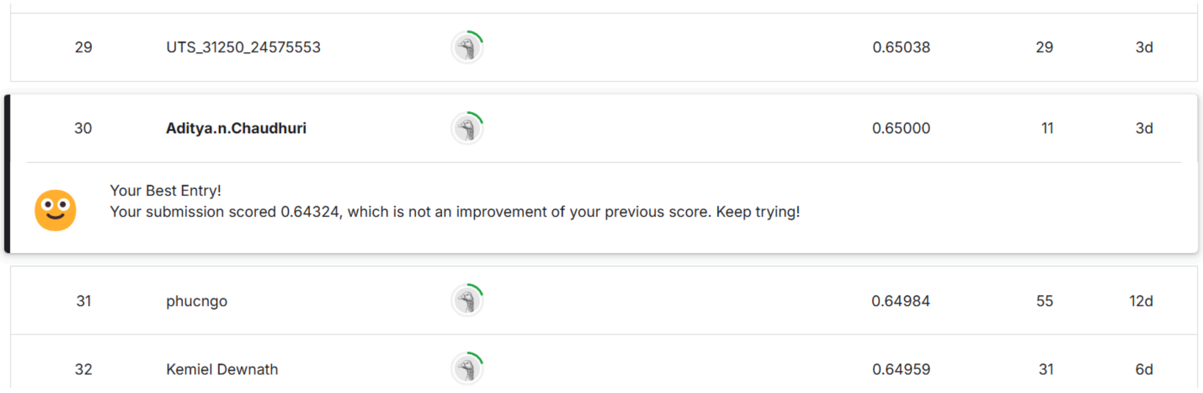

kaggle_submission_random_forest.csv
Complete · 4d ago
0.65000

Figure 34 shows the Kaggle submission for Random Forest and it's accuracy



29	UTS_31250_24575553	0.65038	29	3d
30	**Aditya.n.Chaudhuri**	0.65000	11	3d

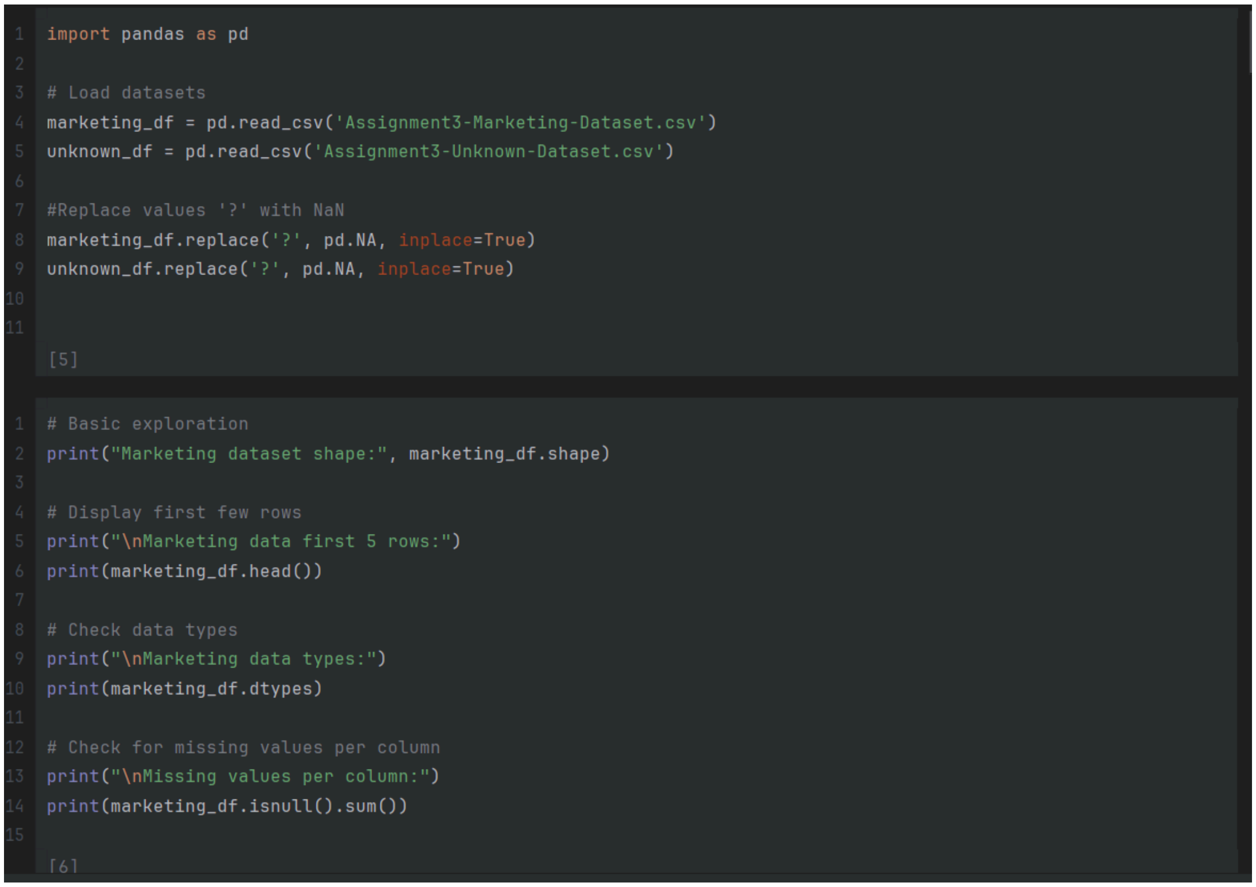
Your Best Entry!
Your submission scored 0.64324, which is not an improvement of your previous score. Keep trying!

31	phucngo	0.64984	55	12d
32	Kemiel Dewnath	0.64959	31	6d

Figure 35 shows my position on the leaderboard for the Kaggle competition.

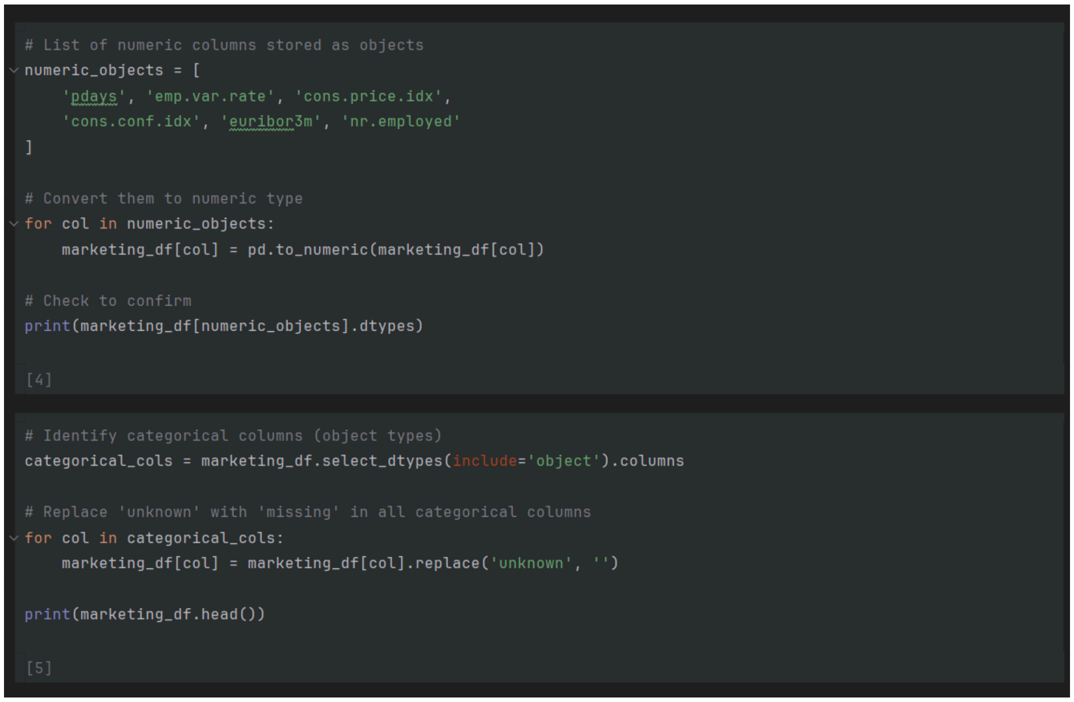
Appendix

Below is the screenshots for the python code used for this assignment.



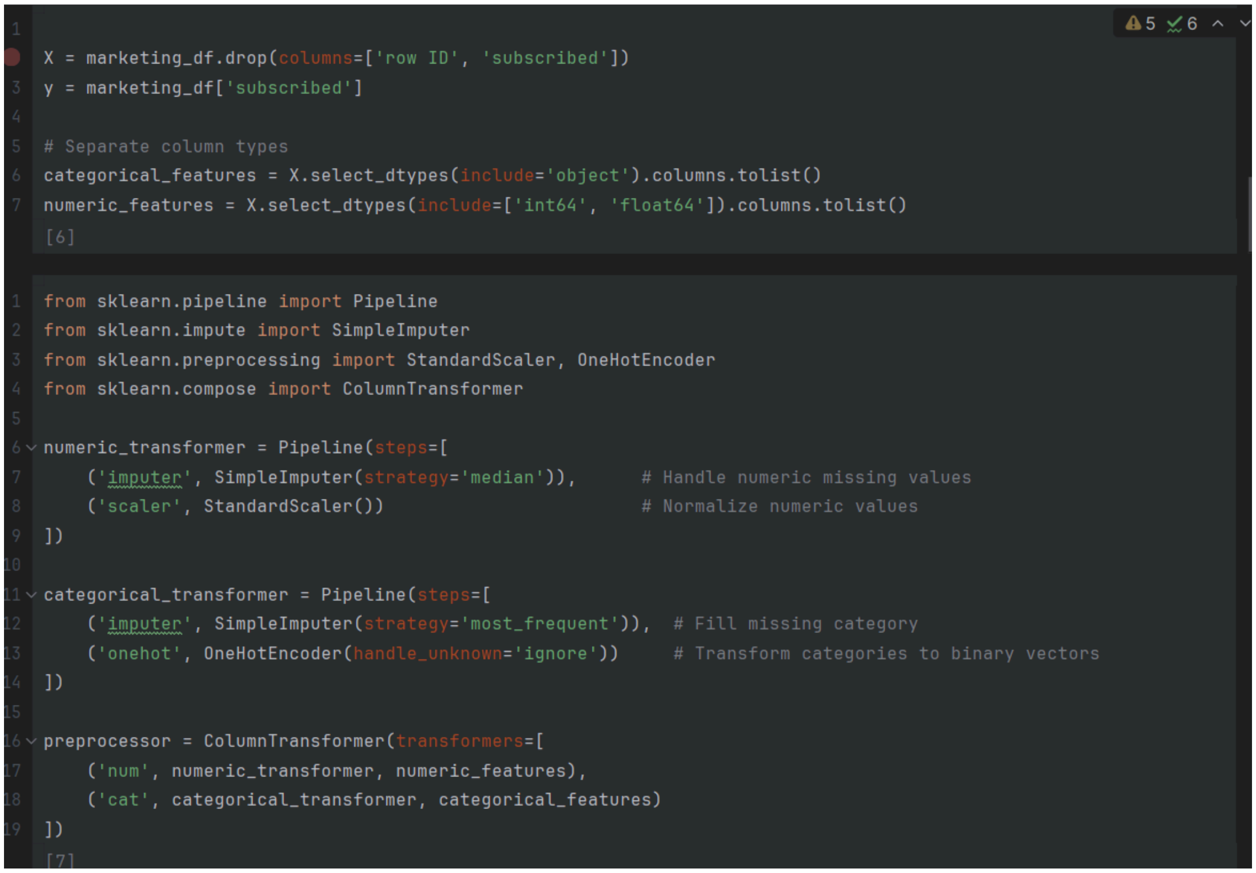
```python
import pandas as pd

# Load datasets
marketing_df = pd.read_csv('Assignment3-Marketing-Dataset.csv')
unknown_df = pd.read_csv('Assignment3-Unknown-Dataset.csv')

#Replace values '?' with NaN
marketing_df.replace('?', pd.NA, inplace=True)
unknown_df.replace('?', pd.NA, inplace=True)


[5]
```

```python
# Basic exploration
print("Marketing dataset shape:", marketing_df.shape)

# Display first few rows
print("\nMarketing data first 5 rows:")
print(marketing_df.head())

# Check data types
print("\nMarketing data types:")
print(marketing_df.dtypes)

# Check for missing values per column
print("\nMissing values per column:")
print(marketing_df.isnull().sum())


[6]
```



```python
# List of numeric columns stored as objects
numeric_objects = [
    'pdays', 'emp.var.rate', 'cons.price.idx',
    'cons.conf.idx', 'euribor3m', 'nr.employed'
]

# Convert them to numeric type
for col in numeric_objects:
    marketing_df[col] = pd.to_numeric(marketing_df[col])

# Check to confirm
print(marketing_df[numeric_objects].dtypes)

[4]
```

```python
# Identify categorical columns (object types)
categorical_cols = marketing_df.select_dtypes(include='object').columns

# Replace 'unknown' with 'missing' in all categorical columns
for col in categorical_cols:
    marketing_df[col] = marketing_df[col].replace('unknown', '')

print(marketing_df.head())

[5]
```

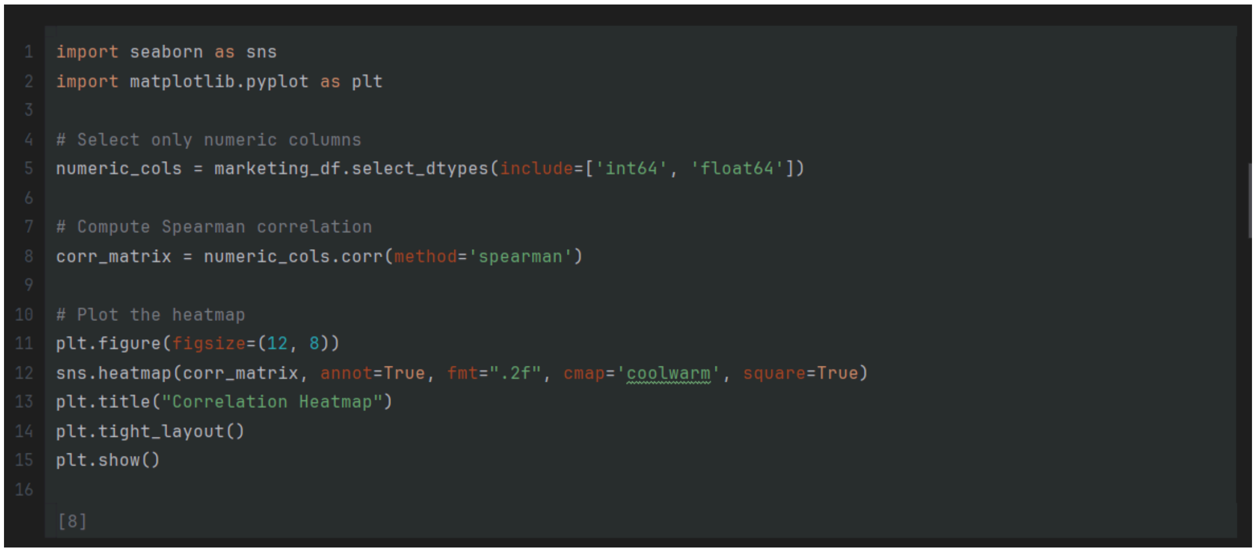


```python
X = marketing_df.drop(columns=['row ID', 'subscribed'])
y = marketing_df['subscribed']

# Separate column types
categorical_features = X.select_dtypes(include='object').columns.tolist()
numeric_features = X.select_dtypes(include=['int64', 'float64']).columns.tolist()
```

[6]

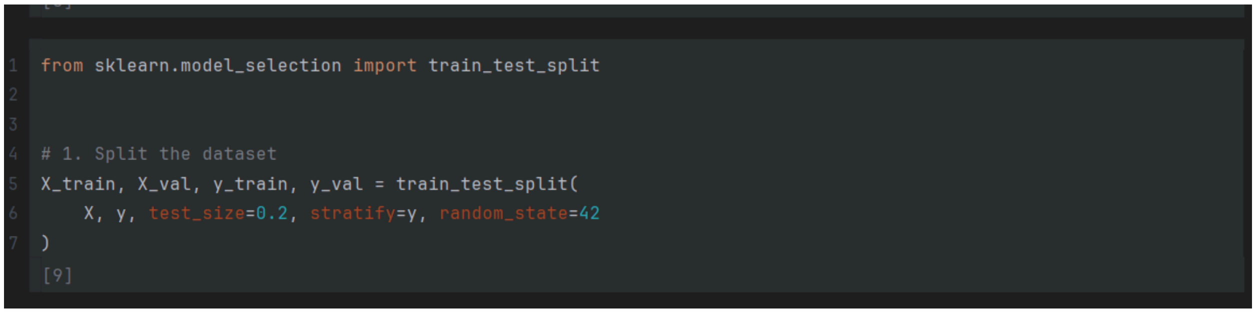
```python
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),      # Handle numeric missing values
    ('scaler', StandardScaler())                        # Normalize numeric values
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),  # Fill missing category
    ('onehot', OneHotEncoder(handle_unknown='ignore'))     # Transform categories to binary vectors
])

preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
])
```

[7]



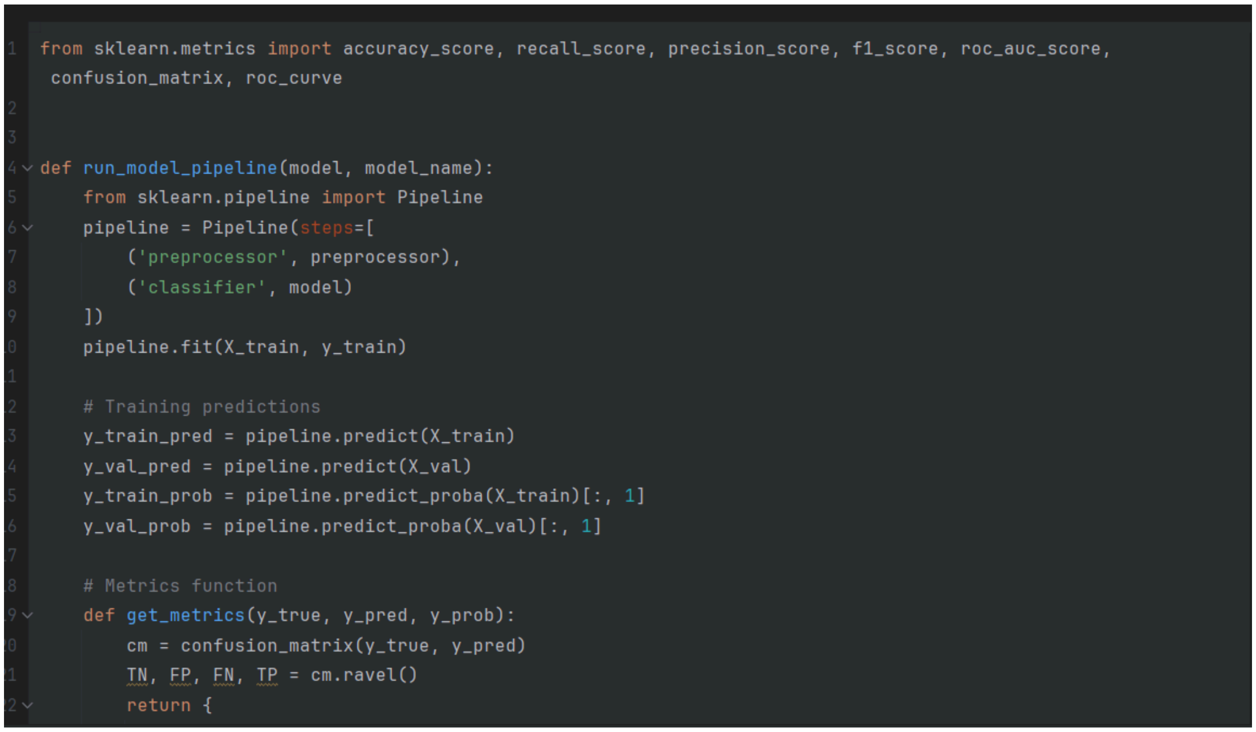
```python
import seaborn as sns
import matplotlib.pyplot as plt

# Select only numeric columns
numeric_cols = marketing_df.select_dtypes(include=['int64', 'float64'])

# Compute Spearman correlation
corr_matrix = numeric_cols.corr(method='spearman')

# Plot the heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm', square=True)
plt.title("Correlation Heatmap")
plt.tight_layout()
plt.show()
```

[8]

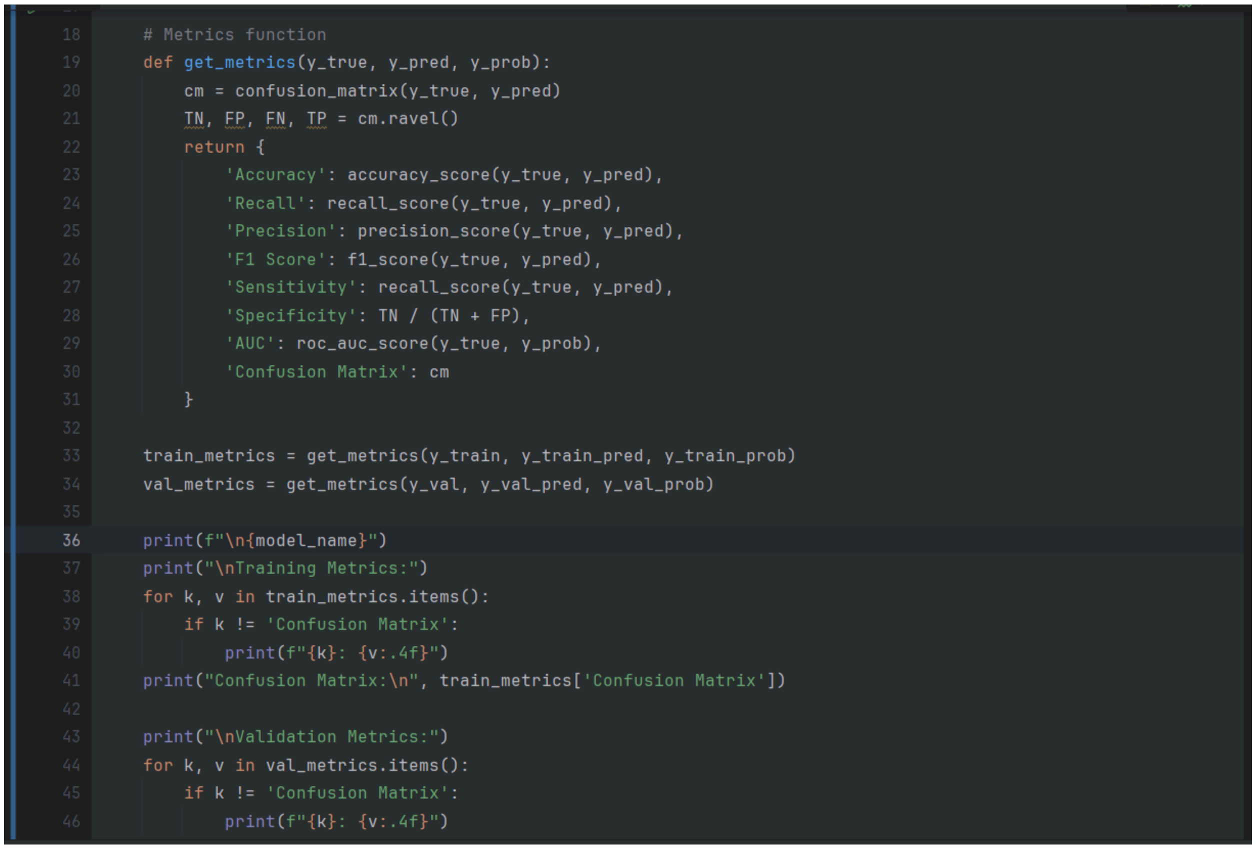


```python
from sklearn.model_selection import train_test_split


# 1. Split the dataset
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)
```

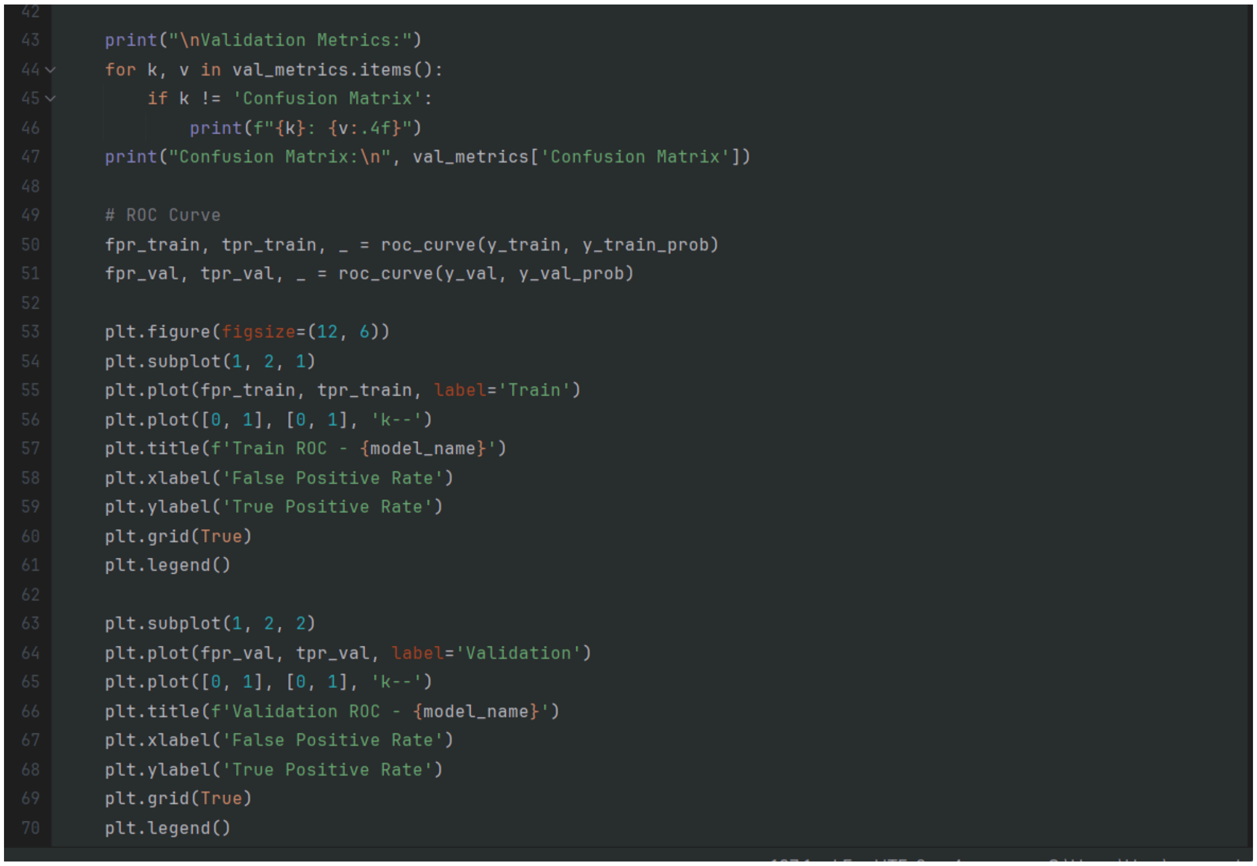
```
[9]
```



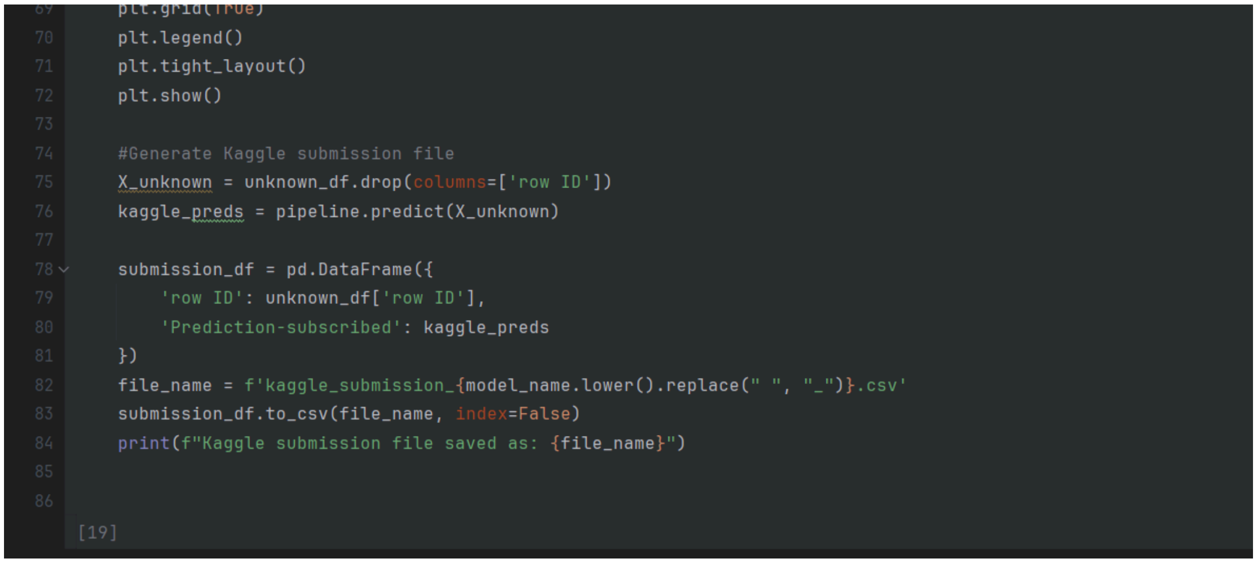
```python
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score, roc_auc_score, confusion_matrix, roc_curve


def run_model_pipeline(model, model_name):
    from sklearn.pipeline import Pipeline
    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('classifier', model)
    ])
    pipeline.fit(X_train, y_train)

    # Training predictions
    y_train_pred = pipeline.predict(X_train)
    y_val_pred = pipeline.predict(X_val)
    y_train_prob = pipeline.predict_proba(X_train)[:, 1]
    y_val_prob = pipeline.predict_proba(X_val)[:, 1]

    # Metrics function
    def get_metrics(y_true, y_pred, y_prob):
        cm = confusion_matrix(y_true, y_pred)
        TN, FP, FN, TP = cm.ravel()
        return {
```



```python
# Metrics function
def get_metrics(y_true, y_pred, y_prob):
    cm = confusion_matrix(y_true, y_pred)
    TN, FP, FN, TP = cm.ravel()
    return {
        'Accuracy': accuracy_score(y_true, y_pred),
        'Recall': recall_score(y_true, y_pred),
        'Precision': precision_score(y_true, y_pred),
        'F1 Score': f1_score(y_true, y_pred),
        'Sensitivity': recall_score(y_true, y_pred),
        'Specificity': TN / (TN + FP),
        'AUC': roc_auc_score(y_true, y_prob),
        'Confusion Matrix': cm
    }

train_metrics = get_metrics(y_train, y_train_pred, y_train_prob)
val_metrics = get_metrics(y_val, y_val_pred, y_val_prob)

print(f"\n{model_name}")
print("\nTraining Metrics:")
for k, v in train_metrics.items():
    if k != 'Confusion Matrix':
        print(f"{k}: {v:.4f}")
print("Confusion Matrix:\n", train_metrics['Confusion Matrix'])

print("\nValidation Metrics:")
for k, v in val_metrics.items():
    if k != 'Confusion Matrix':
        print(f"{k}: {v:.4f}")
```

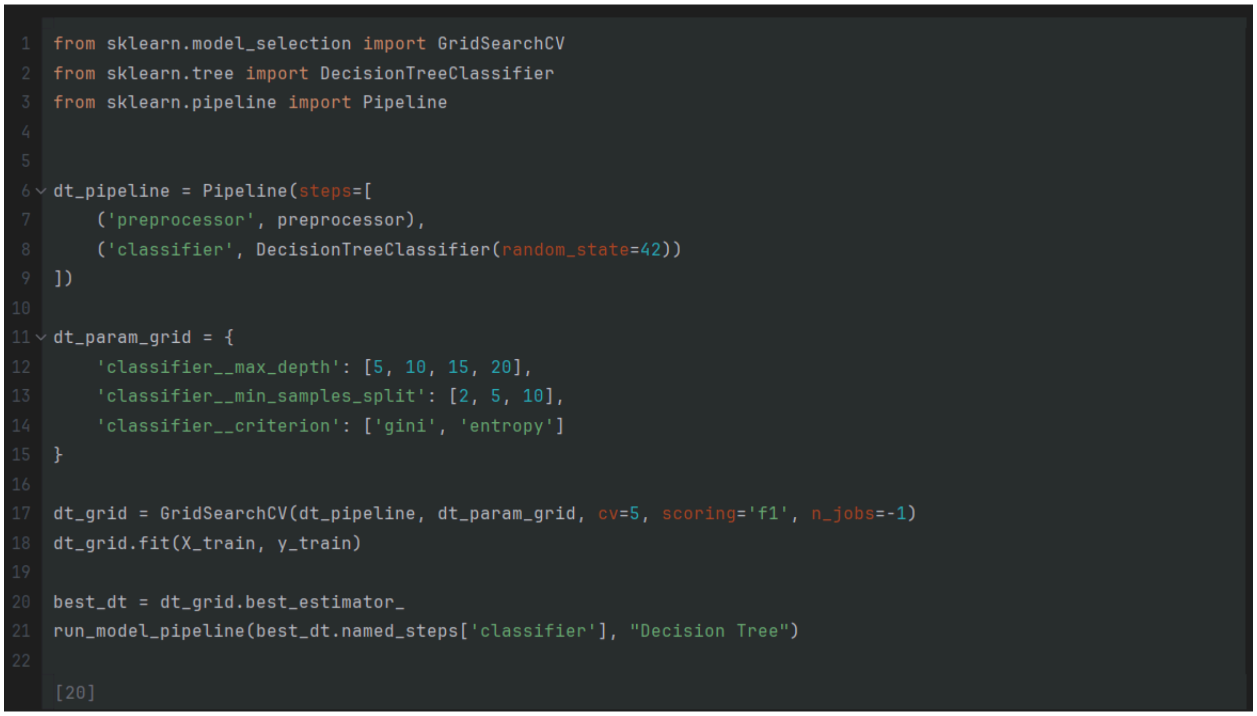


```python
print("\nValidation Metrics:")
for k, v in val_metrics.items():
    if k != 'Confusion Matrix':
        print(f"{k}: {v:.4f}")
print("Confusion Matrix:\n", val_metrics['Confusion Matrix'])

# ROC Curve
fpr_train, tpr_train, _ = roc_curve(y_train, y_train_prob)
fpr_val, tpr_val, _ = roc_curve(y_val, y_val_prob)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(fpr_train, tpr_train, label='Train')
plt.plot([0, 1], [0, 1], 'k--')
plt.title(f'Train ROC - {model_name}')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.grid(True)
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(fpr_val, tpr_val, label='Validation')
plt.plot([0, 1], [0, 1], 'k--')
plt.title(f'Validation ROC - {model_name}')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.grid(True)
plt.legend()
```



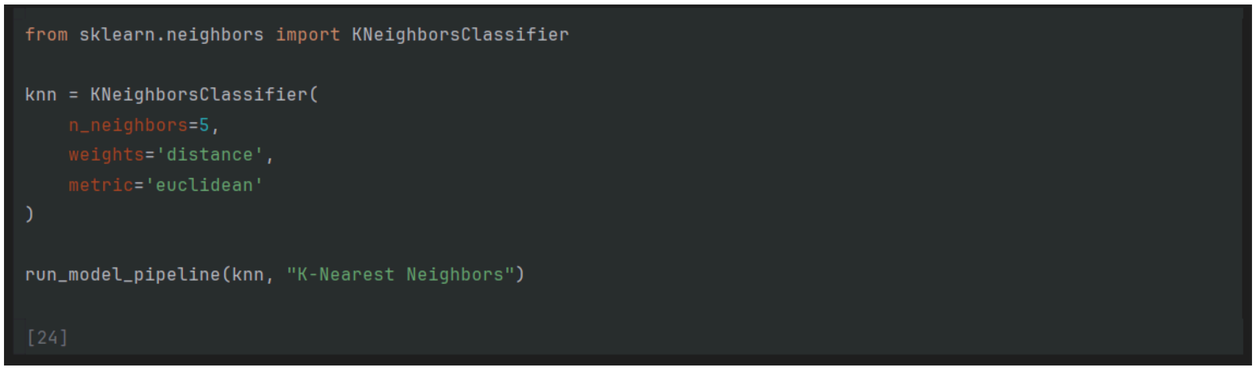
```python
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

    #Generate Kaggle submission file
    X_unknown = unknown_df.drop(columns=['row ID'])
    kaggle_preds = pipeline.predict(X_unknown)

    submission_df = pd.DataFrame({
        'row ID': unknown_df['row ID'],
        'Prediction-subscribed': kaggle_preds
    })
    file_name = f'kaggle_submission_{model_name.lower().replace(" ", "_")}.csv'
    submission_df.to_csv(file_name, index=False)
    print(f"Kaggle submission file saved as: {file_name}")


```

[19]



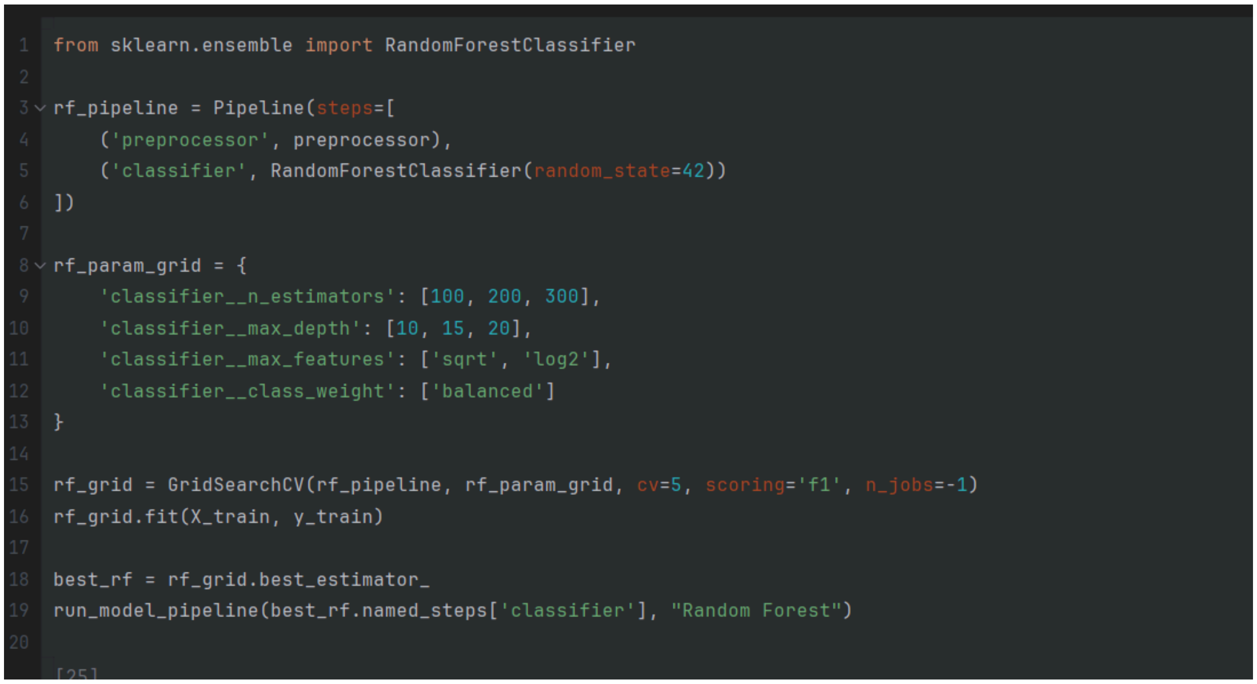
```python
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline


dt_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', DecisionTreeClassifier(random_state=42))
])

dt_param_grid = {
    'classifier__max_depth': [5, 10, 15, 20],
    'classifier__min_samples_split': [2, 5, 10],
    'classifier__criterion': ['gini', 'entropy']
}

dt_grid = GridSearchCV(dt_pipeline, dt_param_grid, cv=5, scoring='f1', n_jobs=-1)
dt_grid.fit(X_train, y_train)

best_dt = dt_grid.best_estimator_
run_model_pipeline(best_dt.named_steps['classifier'], "Decision Tree")

```

[20]

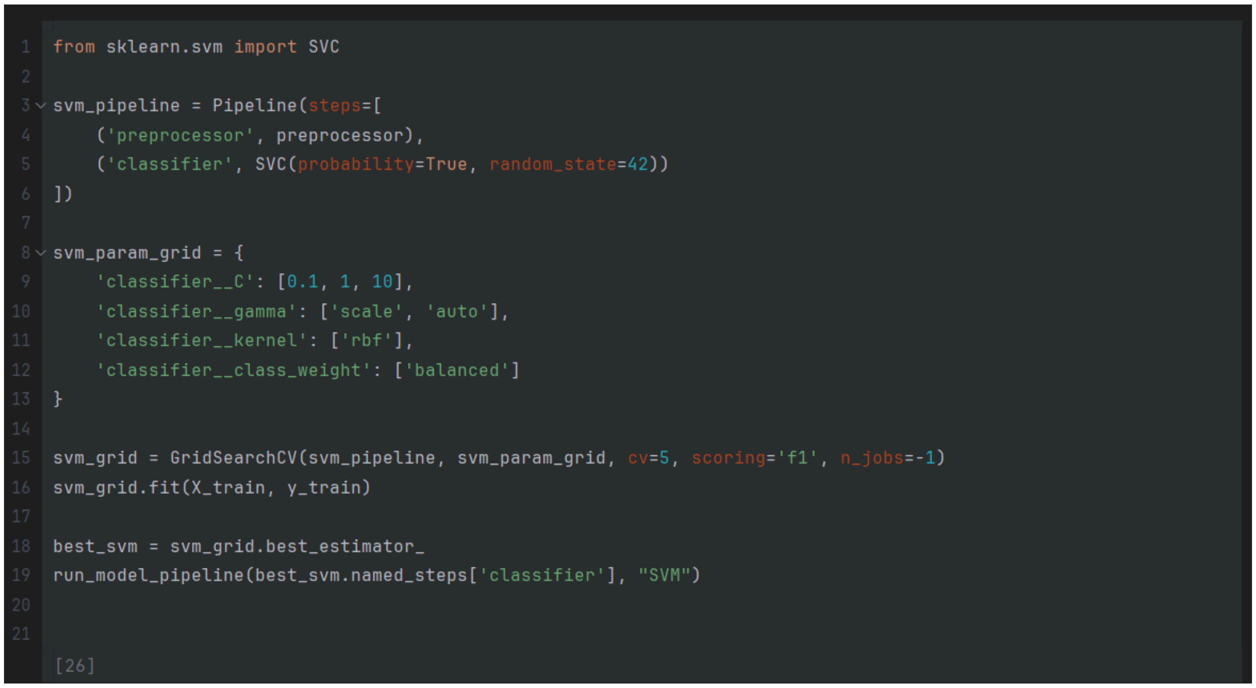


```python
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(
    n_neighbors=5,
    weights='distance',
    metric='euclidean'
)

run_model_pipeline(knn, "K-Nearest Neighbors")
```

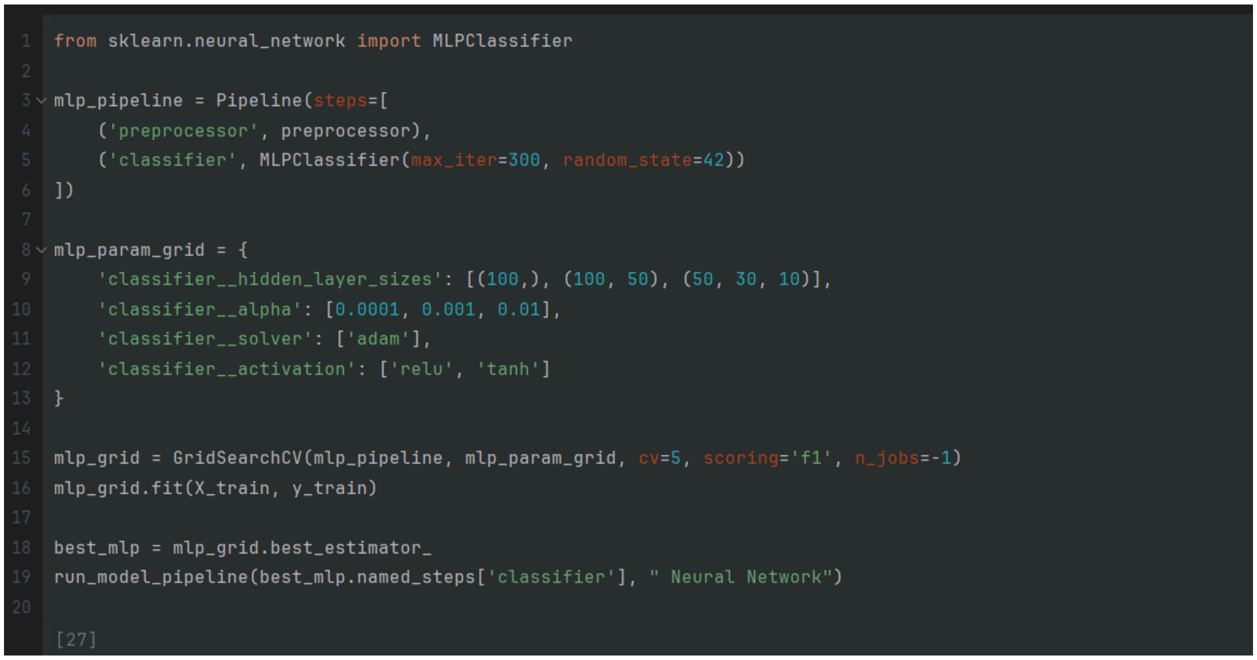
[24]



```python
from sklearn.ensemble import RandomForestClassifier

rf_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(random_state=42))
])

rf_param_grid = {
    'classifier__n_estimators': [100, 200, 300],
    'classifier__max_depth': [10, 15, 20],
    'classifier__max_features': ['sqrt', 'log2'],
    'classifier__class_weight': ['balanced']
}

rf_grid = GridSearchCV(rf_pipeline, rf_param_grid, cv=5, scoring='f1', n_jobs=-1)
rf_grid.fit(X_train, y_train)

best_rf = rf_grid.best_estimator_
run_model_pipeline(best_rf.named_steps['classifier'], "Random Forest")

[25]
```



```
from sklearn.svm import SVC

svm_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', SVC(probability=True, random_state=42))
])

svm_param_grid = {
    'classifier__C': [0.1, 1, 10],
    'classifier__gamma': ['scale', 'auto'],
    'classifier__kernel': ['rbf'],
    'classifier__class_weight': ['balanced']
}

svm_grid = GridSearchCV(svm_pipeline, svm_param_grid, cv=5, scoring='f1', n_jobs=-1)
svm_grid.fit(X_train, y_train)

best_svm = svm_grid.best_estimator_
run_model_pipeline(best_svm.named_steps['classifier'], "SVM")


[26]
```



```
from sklearn.neural_network import MLPClassifier

mlp_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', MLPClassifier(max_iter=300, random_state=42))
])

mlp_param_grid = {
    'classifier__hidden_layer_sizes': [(100,), (100, 50), (50, 30, 10)],
    'classifier__alpha': [0.0001, 0.001, 0.01],
    'classifier__solver': ['adam'],
    'classifier__activation': ['relu', 'tanh']
}

mlp_grid = GridSearchCV(mlp_pipeline, mlp_param_grid, cv=5, scoring='f1', n_jobs=-1)
mlp_grid.fit(X_train, y_train)

best_mlp = mlp_grid.best_estimator_
run_model_pipeline(best_mlp.named_steps['classifier'], " Neural Network")

[27]
```