SunbaseData

Machine Learning Intern Assessment Assignment

Customer Churn Prediction

Shivam Singh

Objective:

The objective of this customer churn prediction project is to leverage machine learning to forecast whether customers are inclined to leave the company's services or stay. This prediction relies on a diverse set of customer attributes, including age, gender, location, subscription duration, monthly billing amount, and total data usage. The core goal of the model is to proactively identify customers who exhibit a higher likelihood of churning, which empowers the business to take preemptive actions aimed at retaining them.

Table of contents:

Understanding The Data:

In the fast-paced world of business today, ensuring customer satisfaction is paramount to retain their loyalty and prevent them from discontinuing the use of our products or services. Our objective is to construct a predictive model capable of identifying customers who are at risk of churning, enabling us to proactively engage with them to prevent attrition.

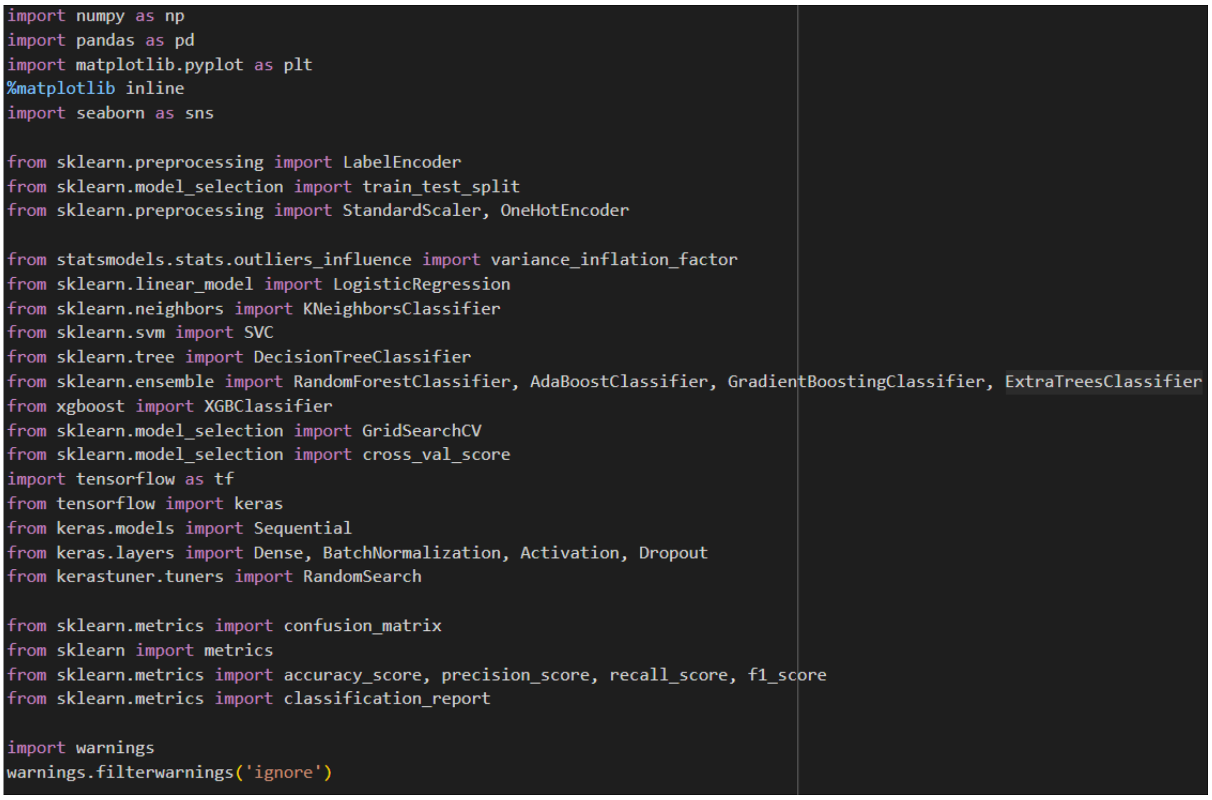
Customer churn, or the loss of customers, can have detrimental effects, including revenue loss and a decrease in the customer base. Leveraging machine learning, we aim to develop a model that can make precise predictions about which customers are likely to churn. These predictions will be based on various factors such as their historical interactions, demographic information, and subscription details.

Features:

1. Name: Name of the customer.
2. Age: Age of the customer.
3. Gender: Gender of the customer (Male or Female).
4. Location: Location where the customer is based, with options including Houston, Los
5. Angeles, Miami, Chicago, and New York.
6. Subscription_Length_Months: The number of months the customer has been subscribed.
7. Monthly_Bill: Monthly bill amount for the customer.
8. Total_Usage_GB: Total usage in gigabytes.
9. Churn: A binary indicator (1 or 0) representing whether the customer has churned (1) or not (0).

The core goal of the model is to proactively identify customers who exhibit a higher likelihood of churning, which empowers the business to take preemptive actions aimed at retaining them.

Libraries:

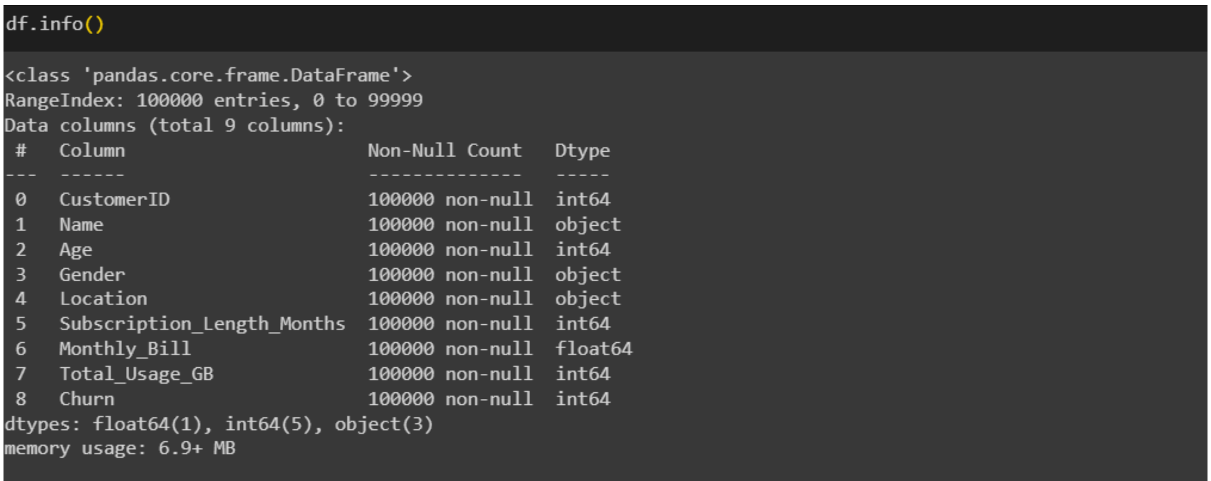


```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder

from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifier, ExtraTreesClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, BatchNormalization, Activation, Dropout
from kerastuner.tuners import RandomSearch

from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import classification_report

import warnings
warnings.filterwarnings('ignore')
```

The data set includes information about:

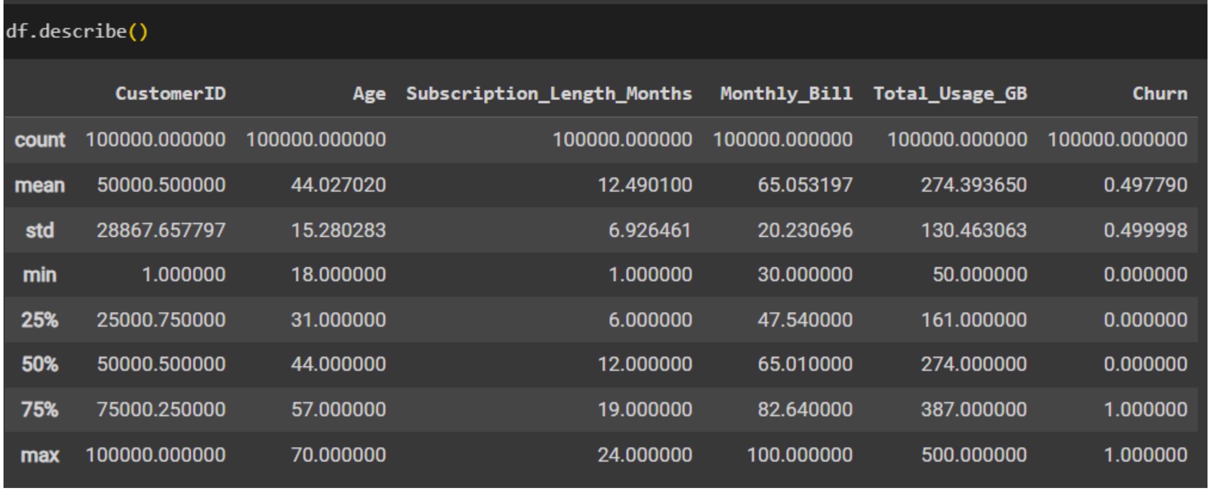
- Customers who left within the last month – the column is called Churn, values are binary (0 or 1), indicating whether a customer has churned or not.
- Services that each customer has signed up for – Subscription_Length_Months, Total_Usage_GB
- Customer account information - Monthly_Bill
- Demographic info about customers – gender, age

Data Visualization:



```
df.info()
```

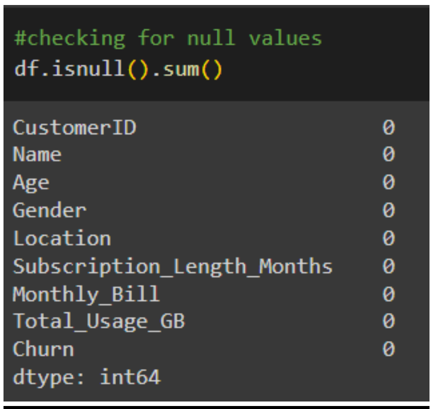
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 9 columns):
 #   Column                      Non-Null Count   Dtype
---  ------                      --------------   -----
 0   CustomerID                  100000 non-null  int64
 1   Name                        100000 non-null  object
 2   Age                         100000 non-null  int64
 3   Gender                      100000 non-null  object
 4   Location                    100000 non-null  object
 5   Subscription_Length_Months  100000 non-null  int64
 6   Monthly_Bill                100000 non-null  float64
 7   Total_Usage_GB              100000 non-null  int64
 8   Churn                       100000 non-null  int64
dtypes: float64(1), int64(5), object(3)
memory usage: 6.9+ MB
```



```
df.describe()
```

	CustomerID	Age	Subscription_Length_Months	Monthly_Bill	Total_Usage_GB	Churn
count	100000.000000	100000.000000	100000.000000	100000.000000	100000.000000	100000.000000
mean	50000.500000	44.027020	12.490100	65.053197	274.393650	0.497790
std	28867.657797	15.280283	6.926461	20.230696	130.463063	0.499998
min	1.000000	18.000000	1.000000	30.000000	50.000000	0.000000
25%	25000.750000	31.000000	6.000000	47.540000	161.000000	0.000000
50%	50000.500000	44.000000	12.000000	65.010000	274.000000	0.000000
75%	75000.250000	57.000000	19.000000	82.640000	387.000000	1.000000
max	100000.000000	70.000000	24.000000	100.000000	500.000000	1.000000

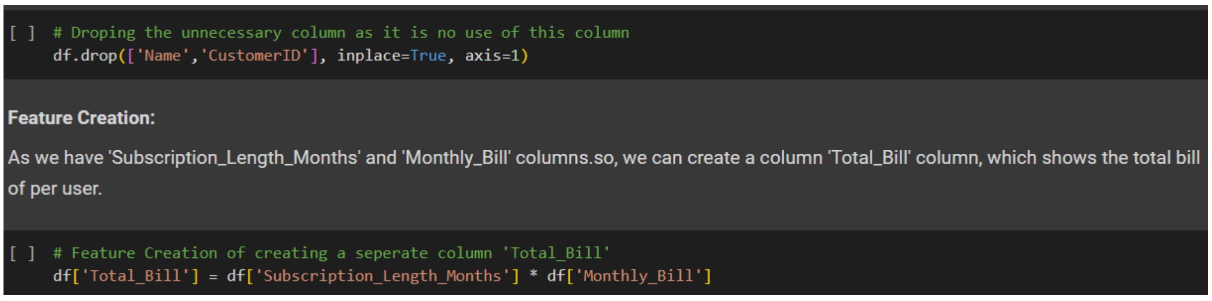
- The dataset contains information about 100,000 customers with 9 variables.
- All variables have the correct data type, and there are no missing values or duplicate records.



```
#checking for null values
df.isnull().sum()
```

```
CustomerID                    0
Name                          0
Age                           0
Gender                        0
Location                      0
Subscription_Length_Months    0
Monthly_Bill                  0
Total_Usage_GB                0
Churn                         0
dtype: int64
```

Data Pre-Processing:



```
[ ]  # Droping the unnecessary column as it is no use of this column
     df.drop(['Name','CustomerID'], inplace=True, axis=1)
```

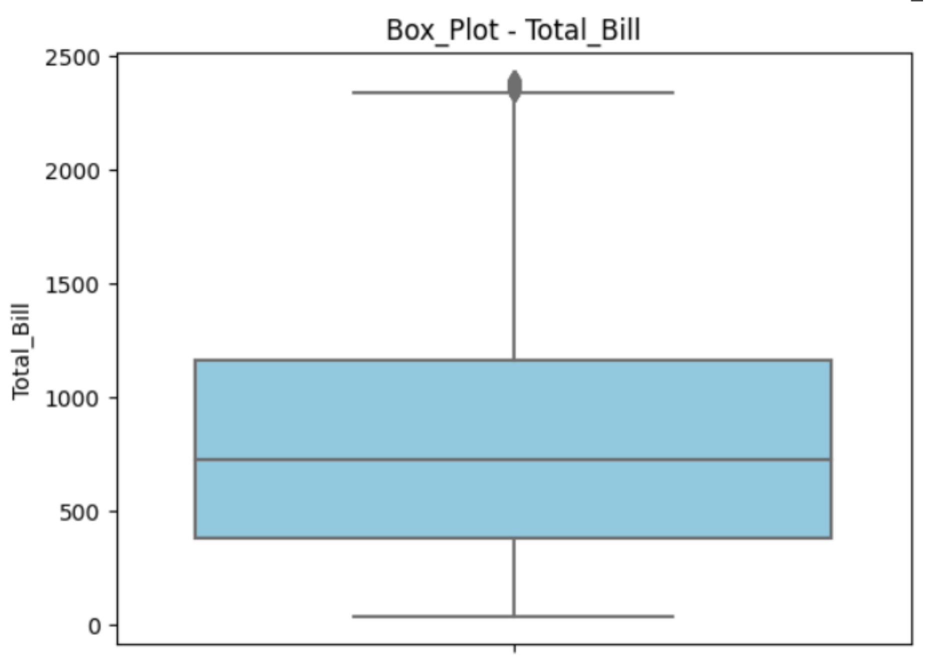
Feature Creation:

As we have 'Subscription_Length_Months' and 'Monthly_Bill' columns.so, we can create a column 'Total_Bill' column, which shows the total bill of per user.

```
[ ]  # Feature Creation of creating a seperate column 'Total_Bill'
     df['Total_Bill'] = df['Subscription_Length_Months'] * df['Monthly_Bill']
```

- The data contains useless columns like 'Name' and 'CustomerID'.
- Feature Creation: As we have 'Subscription_Length_Months' and 'Monthly_Bill' columns. so, we can create a column 'Total_Bill' column, which shows the total bill of per user.
- After the creation of Box-Plot of each columns, I got few outliers in 'Total_Bill' column.

Outliers:



Handle outliers (IQR):

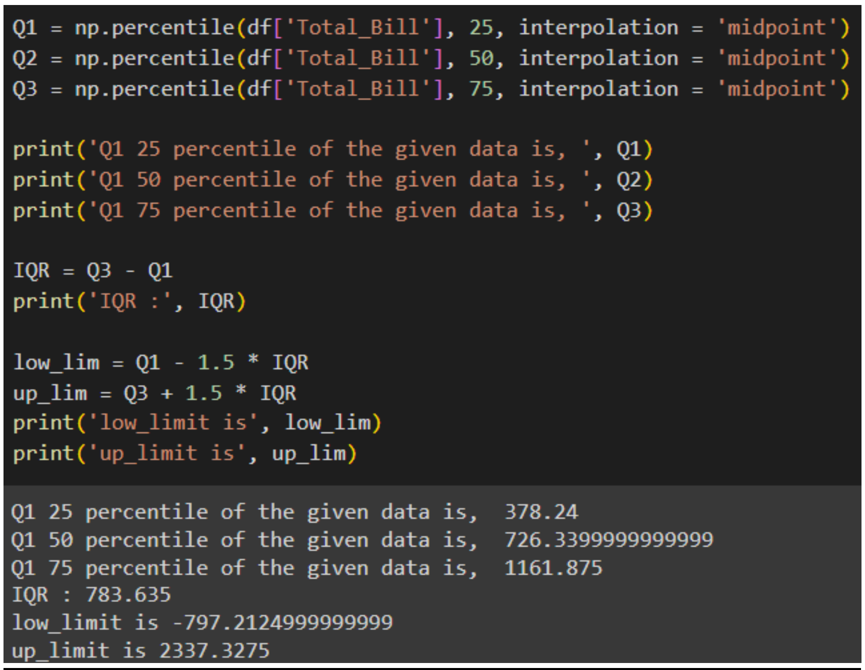
The IQR (Interquartile Range) method is a statistical technique used for identifying and dealing with outliers in a dataset. It involves calculating the spread or variability of the middle 50% of the data points in a dataset, specifically between the first quartile (25th percentile) and the third quartile (75th percentile). The IQR is defined as:

IQR=Q3−Q1

Where,

- IQR is the Interquartile Range.
- Q3 is the third quartile (75th percentile).
- Q1 is the first quartile (25th percentile).

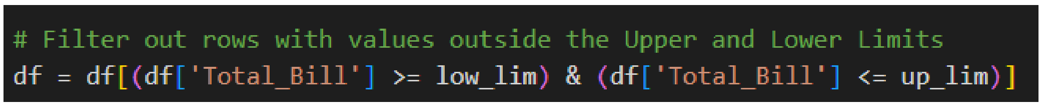
Steps:



```
Q1 = np.percentile(df['Total_Bill'], 25, interpolation = 'midpoint')
Q2 = np.percentile(df['Total_Bill'], 50, interpolation = 'midpoint')
Q3 = np.percentile(df['Total_Bill'], 75, interpolation = 'midpoint')

print('Q1 25 percentile of the given data is, ', Q1)
print('Q1 50 percentile of the given data is, ', Q2)
print('Q1 75 percentile of the given data is, ', Q3)

IQR = Q3 - Q1
print('IQR :', IQR)

low_lim = Q1 - 1.5 * IQR
up_lim = Q3 + 1.5 * IQR
print('low_limit is', low_lim)
print('up_limit is', up_lim)
```

```
Q1 25 percentile of the given data is,  378.24
Q1 50 percentile of the given data is,  726.3399999999999
Q1 75 percentile of the given data is,  1161.875
IQR : 783.635
low_limit is -797.2124999999999
up_limit is 2337.3275
```

As, I got total 139 outliers. Hers the code to remove the outliers.

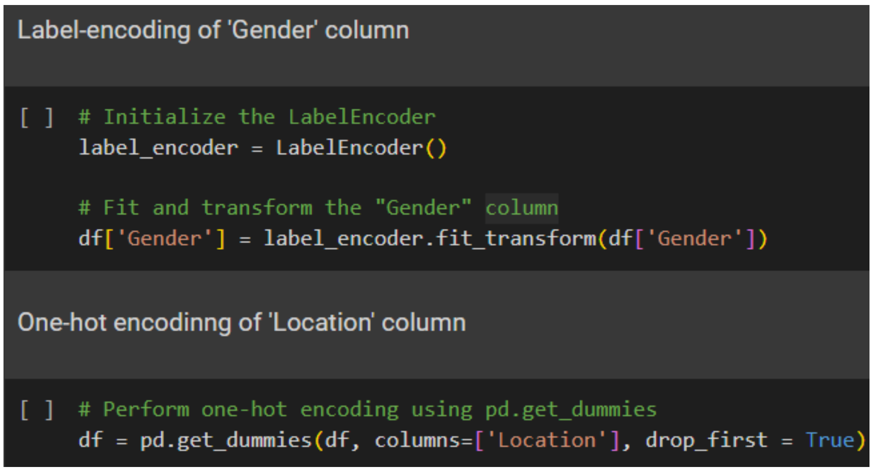


```
# Filter out rows with values outside the Upper and Lower Limits
df = df[(df['Total_Bill'] >= low_lim) & (df['Total_Bill'] <= up_lim)]
```

Feature Engineering:

Encoding:

Here, I used label encoding and one-hot encoding in ‘Gender’ and ‘Location’ columns.



Label-encoding of 'Gender' column

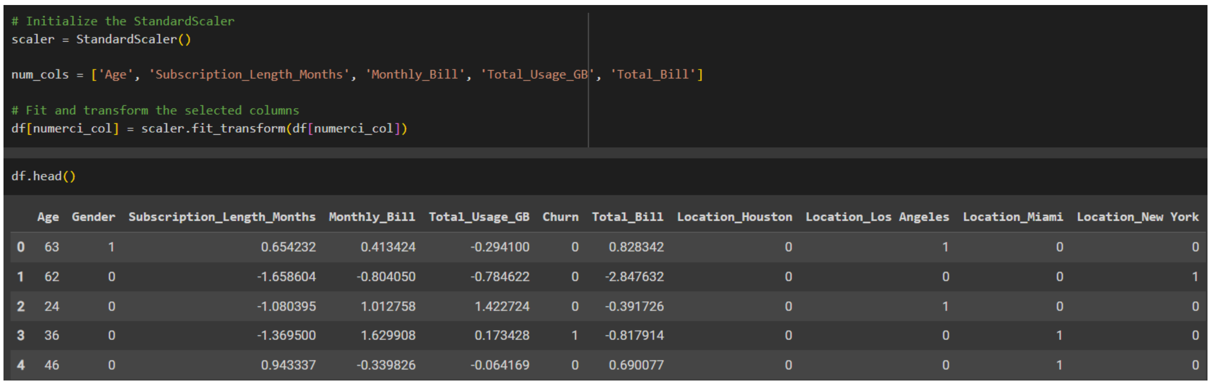
```
[ ]  # Initialize the LabelEncoder
     label_encoder = LabelEncoder()

     # Fit and transform the "Gender" column
     df['Gender'] = label_encoder.fit_transform(df['Gender'])
```

One-hot encodinng of 'Location' column

```
[ ]  # Perform one-hot encoding using pd.get_dummies
     df = pd.get_dummies(df, columns=['Location'], drop_first = True)
```

Standardization:

I performed the standard scaler.

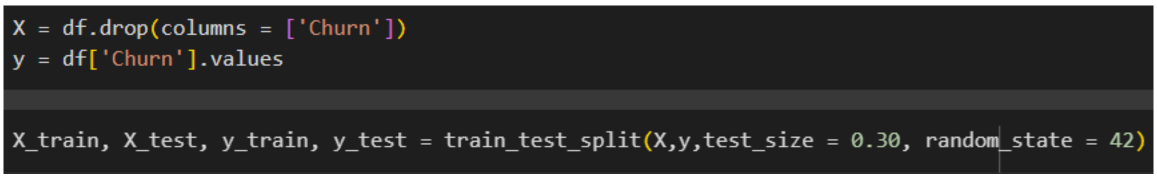


```
# Initialize the StandardScaler
scaler = StandardScaler()

num_cols = ['Age', 'Subscription_Length_Months', 'Monthly_Bill', 'Total_Usage_GB', 'Total_Bill']

# Fit and transform the selected columns
df[numerci_col] = scaler.fit_transform(df[numerci_col])
```

```
df.head()
```

	Age	Gender	Subscription_Length_Months	Monthly_Bill	Total_Usage_GB	Churn	Total_Bill	Location_Houston	Location_Los Angeles	Location_Miami	Location_New York
0	63	1	0.654232	0.413424	-0.294100	0	0.828342	0	1	0	0
1	62	0	-1.658604	-0.804050	-0.784622	0	-2.847632	0	0	0	1
2	24	0	-1.080395	1.012758	1.422724	0	-0.391726	0	1	0	0
3	36	0	-1.369500	1.629908	0.173428	1	-0.817914	0	0	1	0
4	46	0	0.943337	-0.339826	-0.064169	0	0.690077	0	0	1	0

Train-Test Split:

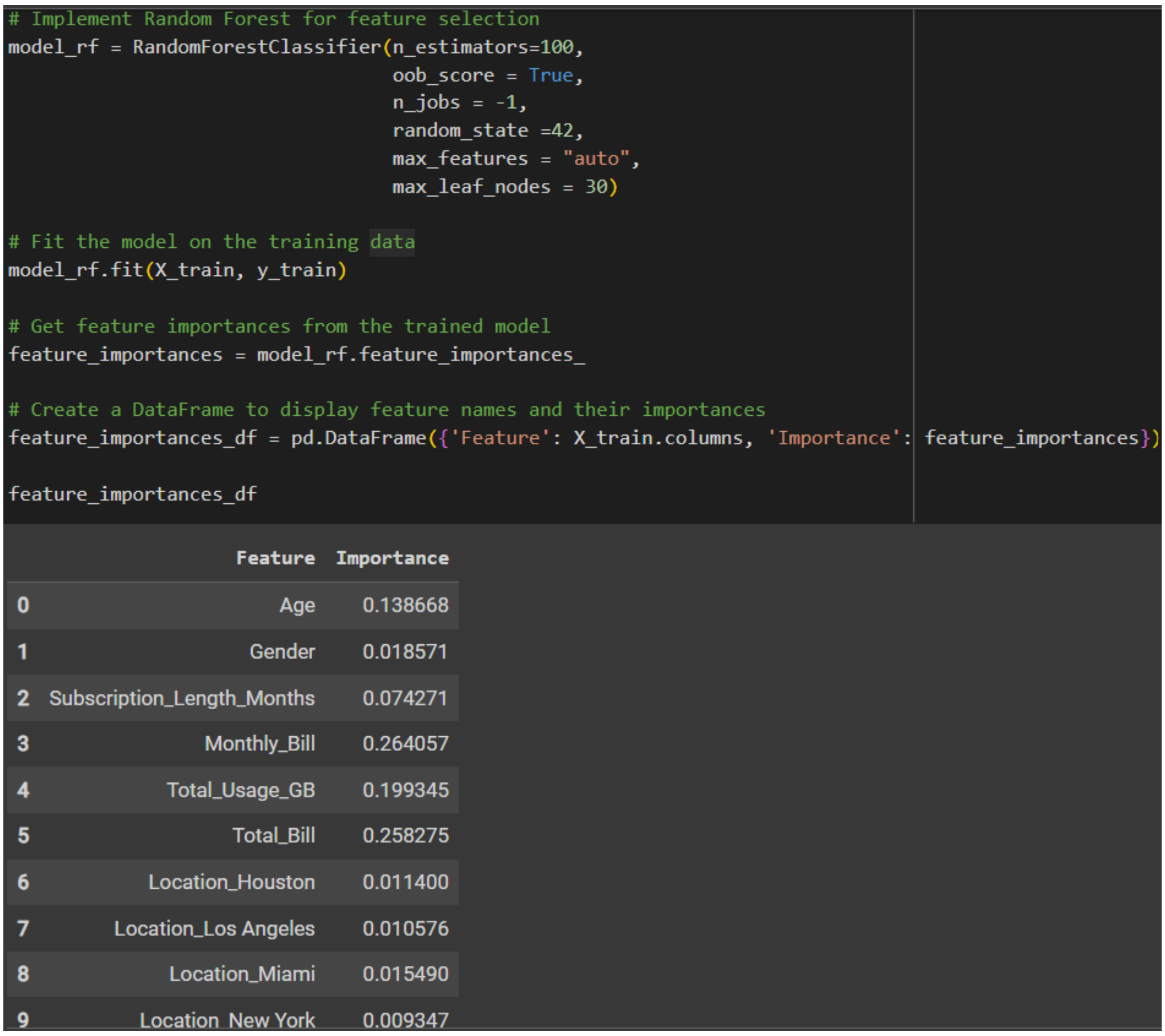


```
X = df.drop(columns = ['Churn'])
y = df['Churn'].values
```

```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.30, random_state = 42)
```

Feature Selection:

I utilized Random Forest. It is a popular and effective technique in machine learning. Random Forests can provide feature importance scores, which indicate the contribution of each feature to the model's predictive performance.



```
# Implement Random Forest for feature selection
model_rf = RandomForestClassifier(n_estimators=100,
                                  oob_score = True,
                                  n_jobs = -1,
                                  random_state =42,
                                  max_features = "auto",
                                  max_leaf_nodes = 30)

# Fit the model on the training data
model_rf.fit(X_train, y_train)

# Get feature importances from the trained model
feature_importances = model_rf.feature_importances_

# Create a DataFrame to display feature names and their importances
feature_importances_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})

feature_importances_df
```

	Feature	Importance
0	Age	0.138668
1	Gender	0.018571
2	Subscription_Length_Months	0.074271
3	Monthly_Bill	0.264057
4	Total_Usage_GB	0.199345
5	Total_Bill	0.258275
6	Location_Houston	0.011400
7	Location_Los Angeles	0.010576
8	Location_Miami	0.015490
9	Location_New York	0.009347

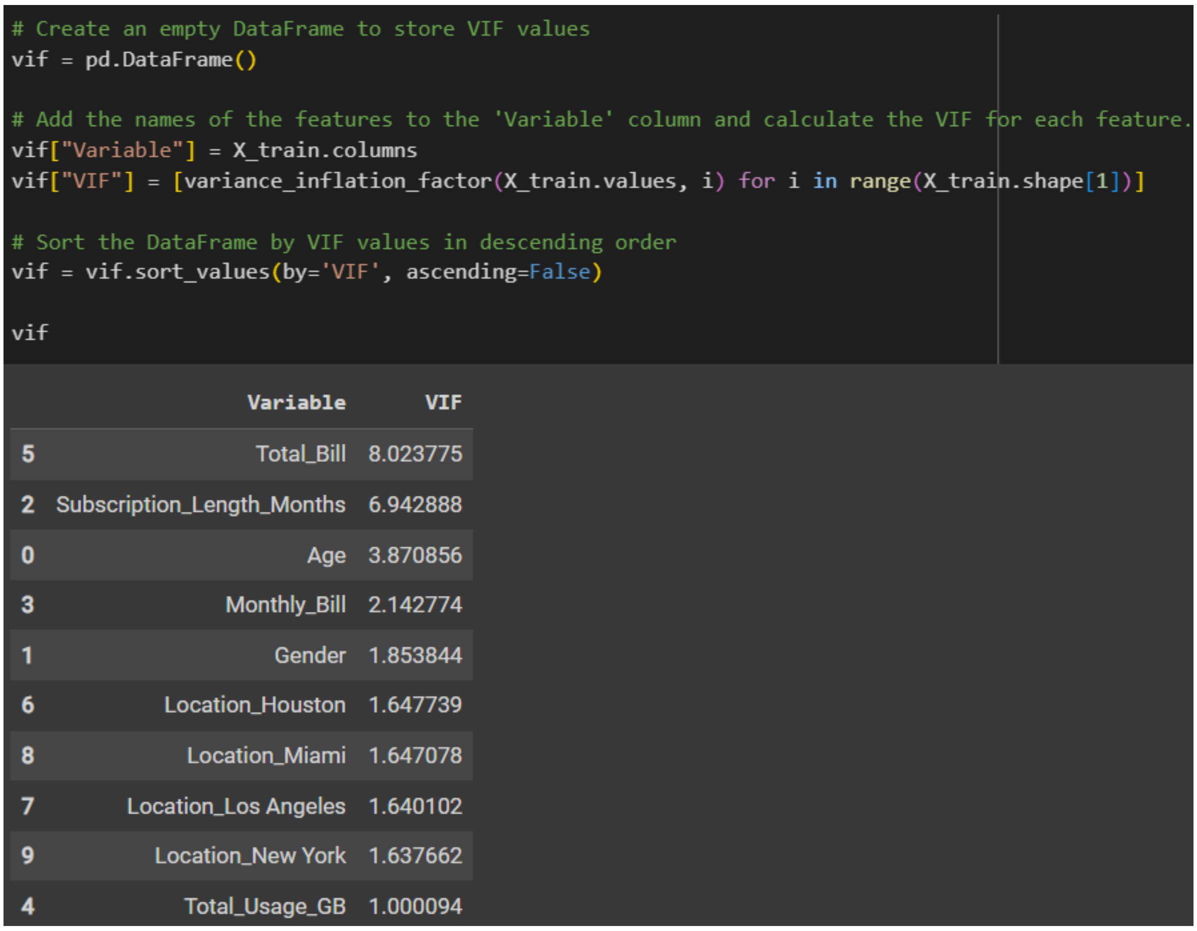
Here, "Monthly_Bill" and "Total_Bill" have relatively high importance scores, suggesting that these two features have a significant impact on your model's predictions and features with low importance scores (e.g., "Location_New York" and "Location_Los Angeles") are less influential.

In this case, it seems I have to use an ensemble method (e.g., Random Forest or XGBoost), which provides feature importances as part of its output.

Check for Multi-Collinearity:

To check the collinearity, I utilized Variation Inflation Factor (VIF).

The Variance Inflation Factor (VIF) is a measure used to assess multicollinearity in a dataset. Multicollinearity occurs when two or more independent variables in a regression model are highly correlated, which can lead to unstable coefficient estimates and difficulty in interpreting the model.



```
# Create an empty DataFrame to store VIF values
vif = pd.DataFrame()

# Add the names of the features to the 'Variable' column and calculate the VIF for each feature.
vif["Variable"] = X_train.columns
vif["VIF"] = [variance_inflation_factor(X_train.values, i) for i in range(X_train.shape[1])]

# Sort the DataFrame by VIF values in descending order
vif = vif.sort_values(by='VIF', ascending=False)

vif
```

	Variable	VIF
5	Total_Bill	8.023775
2	Subscription_Length_Months	6.942888
0	Age	3.870856
3	Monthly_Bill	2.142774
1	Gender	1.853844
6	Location_Houston	1.647739
8	Location_Miami	1.647078
7	Location_Los Angeles	1.640102
9	Location_New York	1.637662
4	Total_Usage_GB	1.000094

Here, I analyzed 2 points:

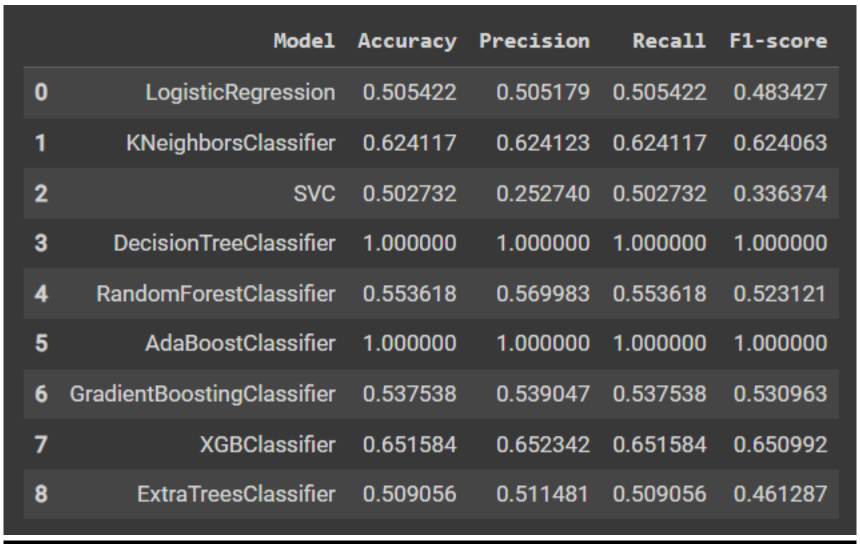
- Variables with high VIF values (e.g., Total_Bill and Subscription_Length_Months) are likely to be strongly correlated with other variables in the dataset.
- Variables with low VIF values (e.g., Total_Usage_GB and Location_New York) have lower multicollinearity and are less correlated with other variables.

Machine Learning Model Evaluation and Metrics:

I have implemented various different machine learning models such as:

Logistic Regression, Supprt Vector Classifier, Decision Tree, K-Nearest Neighbours, AdaBoost, Gradient Boosting, Random Forest, XGBoost, Support Vector Classifier (SVC), and Extree Tree Regressor.

Evaluation:



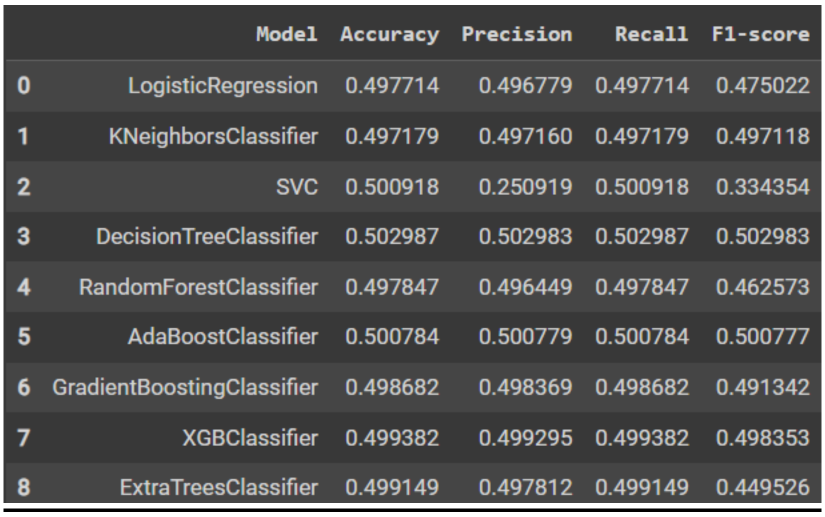
	Model	Accuracy	Precision	Recall	F1-score
0	LogisticRegression	0.505422	0.505179	0.505422	0.483427
1	KNeighborsClassifier	0.624117	0.624123	0.624117	0.624063
2	SVC	0.502732	0.252740	0.502732	0.336374
3	DecisionTreeClassifier	1.000000	1.000000	1.000000	1.000000
4	RandomForestClassifier	0.553618	0.569983	0.553618	0.523121
5	AdaBoostClassifier	1.000000	1.000000	1.000000	1.000000
6	GradientBoostingClassifier	0.537538	0.539047	0.537538	0.530963
7	XGBClassifier	0.651584	0.652342	0.651584	0.650992
8	ExtraTreesClassifier	0.509056	0.511481	0.509056	0.461287

Models like Decision Tree, AdaBoost, and SVC achieved perfect scores on the training data, which is often indicative of overfitting.

Some models, such as Decision Tree and Random Forest, offer more interpretability than others like XGBoost.

I think Hyperparameter tuning and feature engineering of XGBoost can often improve model performance.

Metrics:



	Model	Accuracy	Precision	Recall	F1-score
0	LogisticRegression	0.497714	0.496779	0.497714	0.475022
1	KNeighborsClassifier	0.497179	0.497160	0.497179	0.497118
2	SVC	0.500918	0.250919	0.500918	0.334354
3	DecisionTreeClassifier	0.502987	0.502983	0.502987	0.502983
4	RandomForestClassifier	0.497847	0.496449	0.497847	0.462573
5	AdaBoostClassifier	0.500784	0.500779	0.500784	0.500777
6	GradientBoostingClassifier	0.498682	0.498369	0.498682	0.491342
7	XGBClassifier	0.499382	0.499295	0.499382	0.498353
8	ExtraTreesClassifier	0.499149	0.497812	0.499149	0.449526

As, here models like AdaBoost and GradientBoostingClassifier may perform well due to their ensemble nature, which combines multiple weak learners.

I think, XGBoost appears to have relatively balanced performance across metrics, which can make it a good candidate for further tuning.

So, Let's try hyper-parameter tunning on XGBoost.

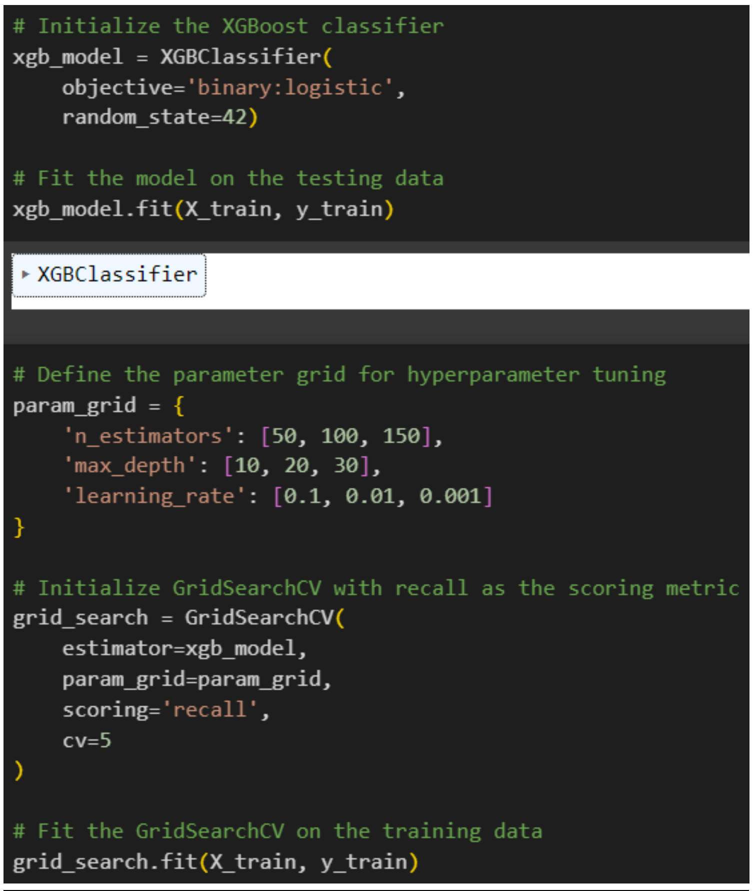
Hyper-Parameter Tunning:

and Grid Search Cross Validation:

Hyperparameter tuning is a critical process in machine learning model development that aims to find the best combination of hyperparameters to optimize model performance. Hyperparameters are settings that are predetermined before training and are not learned from the data. Examples include the minimum no of samples split, learning rate, tree depths, and the number of estimators in ensemble methods.

GridSearchCV is a technique used for hyperparameter tuning in machine learning. It exhaustively searches through a predefined hyperparameter grid to find the best combination of hyperparameters for a given model. It uses cross-validation to evaluate the model's performance for each set of hyperparameters and selects the best combination based on a specified scoring metric.

I have implemented Grid Search CV on XGBoost:

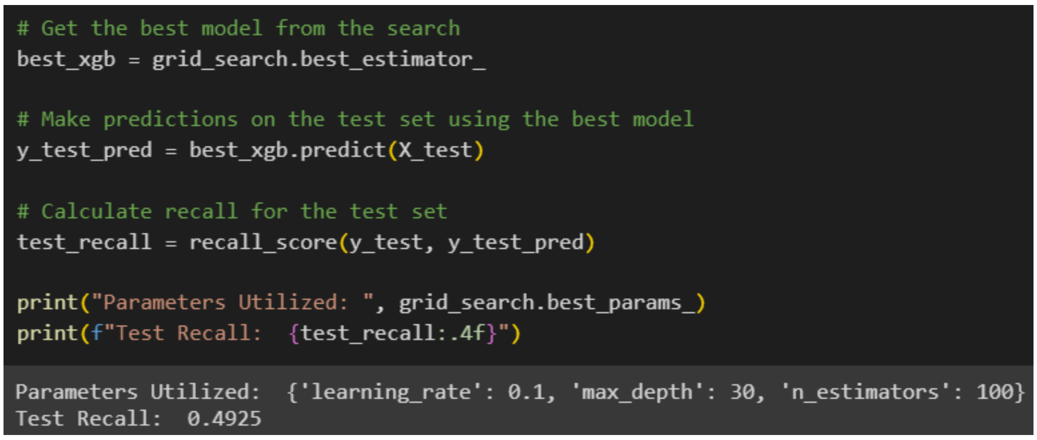


```
# Initialize the XGBoost classifier
xgb_model = XGBClassifier(
    objective='binary:logistic',
    random_state=42)

# Fit the model on the testing data
xgb_model.fit(X_train, y_train)
```

```
▸ XGBClassifier
```

```
# Define the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [10, 20, 30],
    'learning_rate': [0.1, 0.01, 0.001]
}

# Initialize GridSearchCV with recall as the scoring metric
grid_search = GridSearchCV(
    estimator=xgb_model,
    param_grid=param_grid,
    scoring='recall',
    cv=5
)

# Fit the GridSearchCV on the training data
grid_search.fit(X_train, y_train)
```

The parameters that I have used include the minimum no of samples split, learning rate, tree depths, and the number of estimators in ensemble methods.



```
# Get the best model from the search
best_xgb = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_test_pred = best_xgb.predict(X_test)

# Calculate recall for the test set
test_recall = recall_score(y_test, y_test_pred)

print("Parameters Utilized: ", grid_search.best_params_)
print(f"Test Recall:  {test_recall:.4f}")
```

```
Parameters Utilized:  {'learning_rate': 0.1, 'max_depth': 30, 'n_estimators': 100}
Test Recall:  0.4925
```

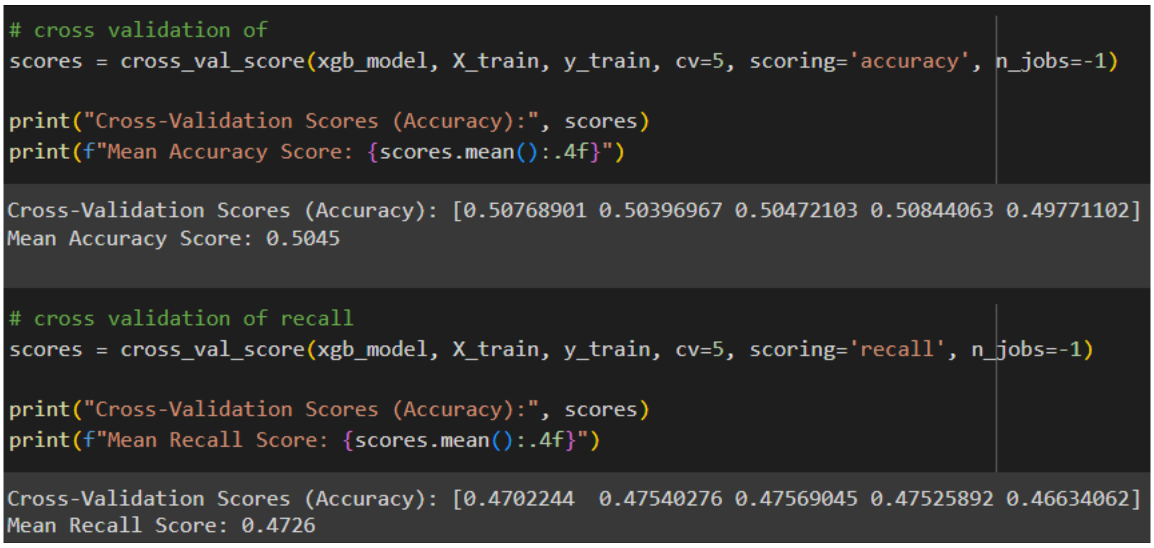
As, Here also I was getting the same result of accuracy.

Cross-Validation:

Cross-validation was performed to validate the model's performance and ensure it generalized well to new data.

K-Fold Cross-Validation: The dataset is divided into k equally sized folds, and the model is trained and tested k times.

I performed CV on accuracy as well as recall:



```
# cross validation of
scores = cross_val_score(xgb_model, X_train, y_train, cv=5, scoring='accuracy', n_jobs=-1)

print("Cross-Validation Scores (Accuracy):", scores)
print(f"Mean Accuracy Score: {scores.mean():.4f}")
```

```
Cross-Validation Scores (Accuracy): [0.50768901 0.50396967 0.50472103 0.50844063 0.49771102]
Mean Accuracy Score: 0.5045
```

```
# cross validation of recall
scores = cross_val_score(xgb_model, X_train, y_train, cv=5, scoring='recall', n_jobs=-1)

print("Cross-Validation Scores (Accuracy):", scores)
print(f"Mean Recall Score: {scores.mean():.4f}")
```

```
Cross-Validation Scores (Accuracy): [0.4702244  0.47540276 0.47569045 0.47525892 0.46634062]
Mean Recall Score: 0.4726
```

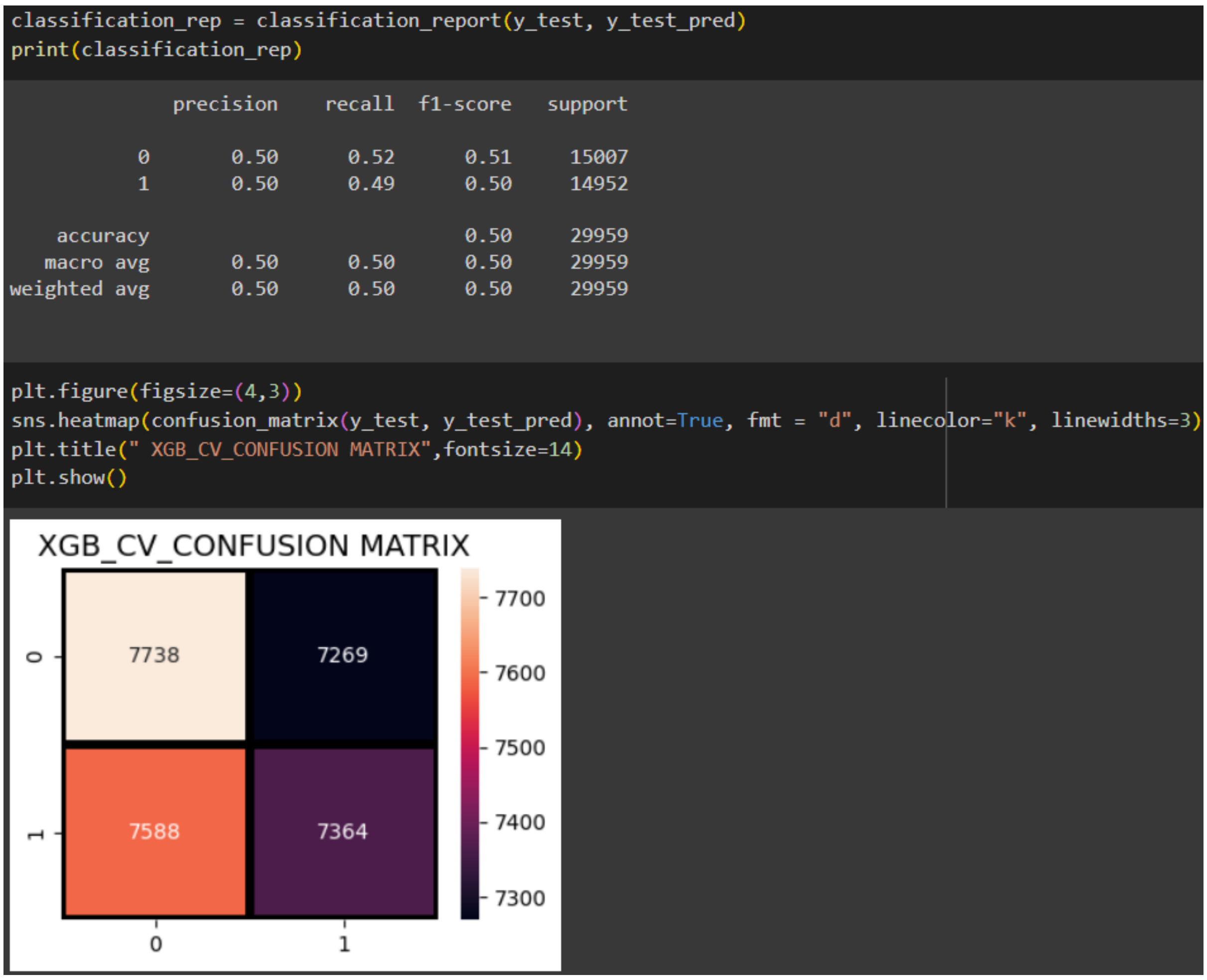
Confusion Matrix:

A confusion matrix is a table used in machine learning and statistics to evaluate the performance of a classification model.

A standard confusion matrix consists of four values:

- True Positives (TP): The number of instances correctly predicted as the positive class.
- True Negatives (TN): The number of instances correctly predicted as the negative class.
- False Positives (FP): The number of instances incorrectly predicted as the positive class (Type I error).
- False Negatives (FN): The number of instances incorrectly predicted as the negative class (Type II error).

XGBoost's model performance:



```
classification_rep = classification_report(y_test, y_test_pred)
print(classification_rep)
```

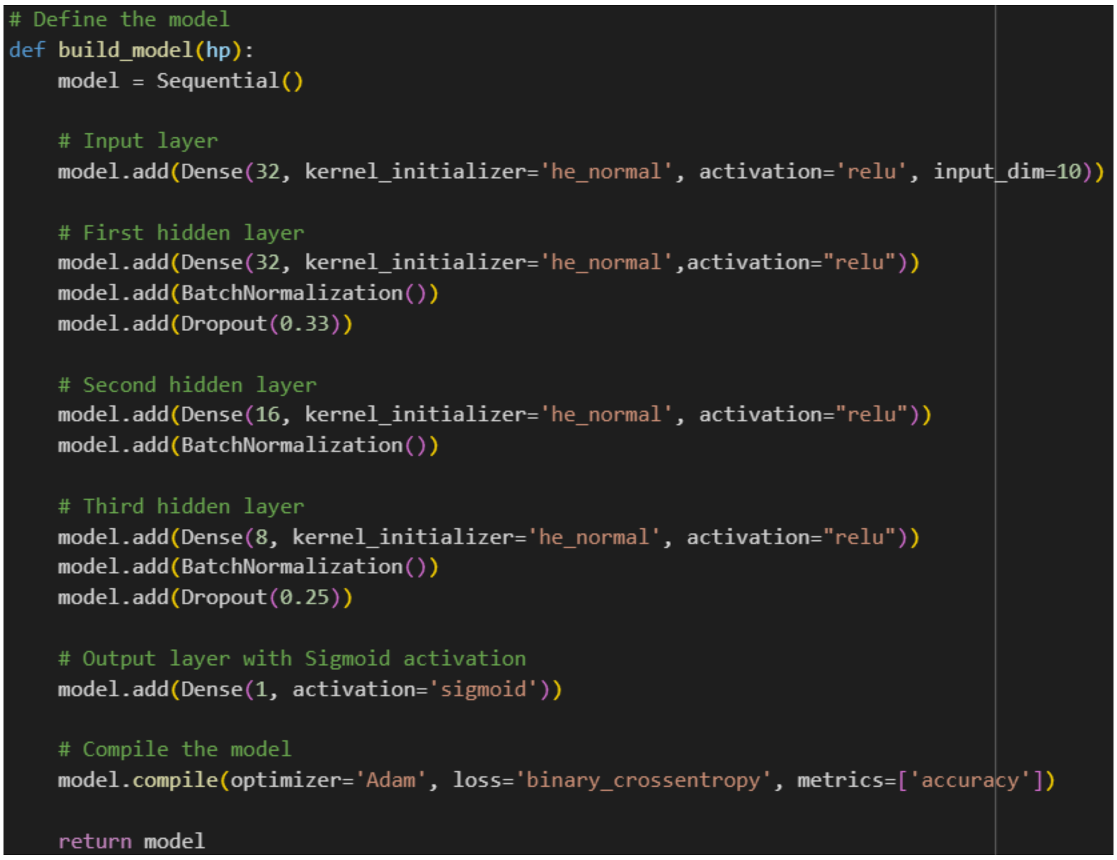
```
              precision    recall  f1-score   support

           0       0.50      0.52      0.51     15007
           1       0.50      0.49      0.50     14952

    accuracy                           0.50     29959
   macro avg       0.50      0.50      0.50     29959
weighted avg       0.50      0.50      0.50     29959
```

```
plt.figure(figsize=(4,3))
sns.heatmap(confusion_matrix(y_test, y_test_pred), annot=True, fmt = "d", linecolor="k", linewidths=3)
plt.title(" XGB_CV_CONFUSION MATRIX",fontsize=14)
plt.show()
```

Neural Network Architecture:

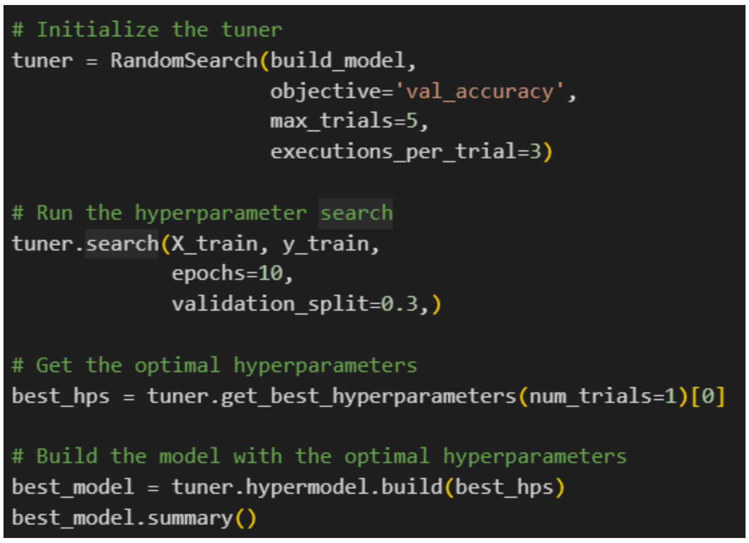
I designing a neural network architecture involves making decisions about various aspects of the network, including the number of layers, the number of neurons in each layer, the activation functions, and the architecture's overall structure. I will use Keras Tunner and Early Stopping

- Keras Tuner is an open-source library for hyperparameter tuning of deep learning models built using TensorFlow and Keras. It provides a high-level API to search for the best hyperparameters, such as learning rate, number of layers, number of neurons, and dropout rates, to optimize the performance of your neural networks.
- Early stopping is a technique that stops the training of a model when the validation loss stops improving. If you set restore_best_weights to False, the model will be saved at the end of the training, even if the validation loss has not improved. This could result in the model being saved at a point where it is overfitting the training data.



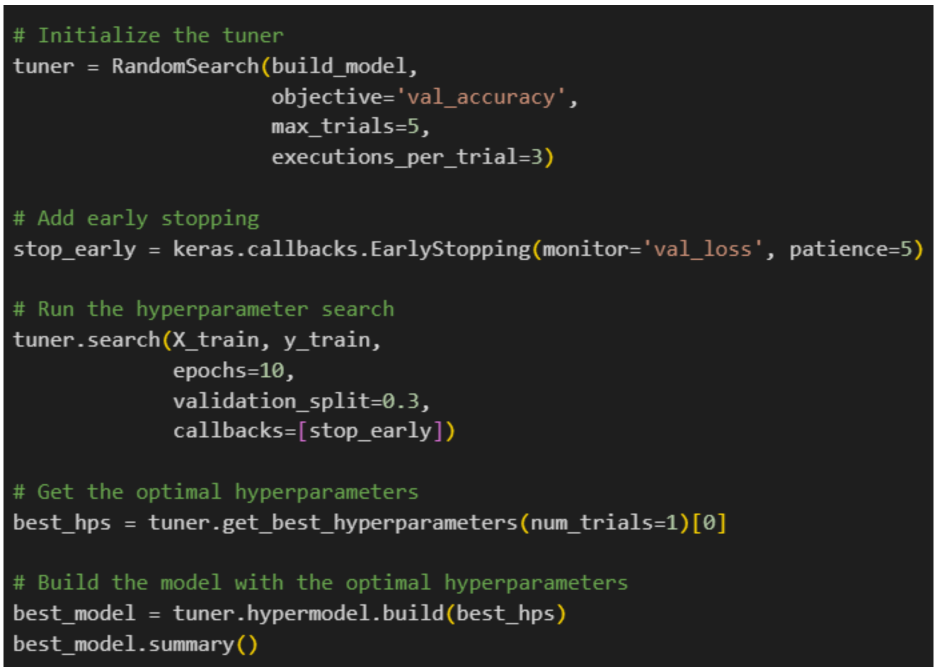
```python
# Define the model
def build_model(hp):
    model = Sequential()

    # Input layer
    model.add(Dense(32, kernel_initializer='he_normal', activation='relu', input_dim=10))

    # First hidden layer
    model.add(Dense(32, kernel_initializer='he_normal',activation="relu"))
    model.add(BatchNormalization())
    model.add(Dropout(0.33))

    # Second hidden layer
    model.add(Dense(16, kernel_initializer='he_normal', activation="relu"))
    model.add(BatchNormalization())

    # Third hidden layer
    model.add(Dense(8, kernel_initializer='he_normal', activation="relu"))
    model.add(BatchNormalization())
    model.add(Dropout(0.25))

    # Output layer with Sigmoid activation
    model.add(Dense(1, activation='sigmoid'))

    # Compile the model
    model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])

    return model
```

I have implemented with 1 input layer and 3 dense layer with kernel initializer as he_normal, Activation function as relu, Batchnormalization and with Drop out layer.

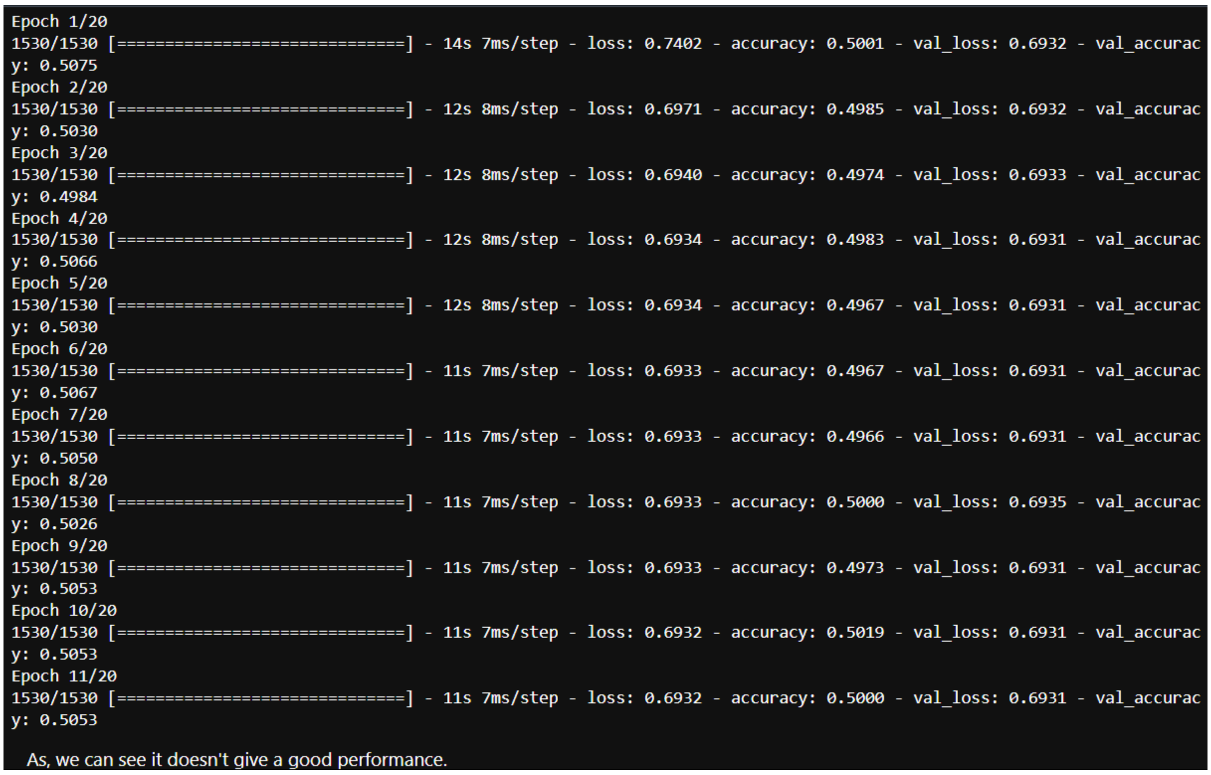


```
# Initialize the tuner
tuner = RandomSearch(build_model,
                     objective='val_accuracy',
                     max_trials=5,
                     executions_per_trial=3)

# Run the hyperparameter search
tuner.search(X_train, y_train,
             epochs=10,
             validation_split=0.3,)

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the optimal hyperparameters
best_model = tuner.hypermodel.build(best_hps)
best_model.summary()
```

Here, I had set the patience=5 i.e., the number of epochs with no improvement after which training will be stopped, and max_trials=3, which represents the number of hyperparameter combinations that will be tested by the tuner.



```
# Initialize the tuner
tuner = RandomSearch(build_model,
                     objective='val_accuracy',
                     max_trials=5,
                     executions_per_trial=3)

# Add early stopping
stop_early = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)

# Run the hyperparameter search
tuner.search(X_train, y_train,
             epochs=10,
             validation_split=0.3,
             callbacks=[stop_early])

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the optimal hyperparameters
best_model = tuner.hypermodel.build(best_hps)
best_model.summary()
```



```
Epoch 1/20
1530/1530 [==============================] - 14s 7ms/step - loss: 0.7402 - accuracy: 0.5001 - val_loss: 0.6932 - val_accurac
y: 0.5075
Epoch 2/20
1530/1530 [==============================] - 12s 8ms/step - loss: 0.6971 - accuracy: 0.4985 - val_loss: 0.6932 - val_accurac
y: 0.5030
Epoch 3/20
1530/1530 [==============================] - 12s 8ms/step - loss: 0.6940 - accuracy: 0.4974 - val_loss: 0.6933 - val_accurac
y: 0.4984
Epoch 4/20
1530/1530 [==============================] - 12s 8ms/step - loss: 0.6934 - accuracy: 0.4983 - val_loss: 0.6931 - val_accurac
y: 0.5066
Epoch 5/20
1530/1530 [==============================] - 12s 8ms/step - loss: 0.6934 - accuracy: 0.4967 - val_loss: 0.6931 - val_accurac
y: 0.5030
Epoch 6/20
1530/1530 [==============================] - 11s 7ms/step - loss: 0.6933 - accuracy: 0.4967 - val_loss: 0.6931 - val_accurac
y: 0.5067
Epoch 7/20
1530/1530 [==============================] - 11s 7ms/step - loss: 0.6933 - accuracy: 0.4966 - val_loss: 0.6931 - val_accurac
y: 0.5050
Epoch 8/20
1530/1530 [==============================] - 11s 7ms/step - loss: 0.6933 - accuracy: 0.5000 - val_loss: 0.6935 - val_accurac
y: 0.5026
Epoch 9/20
1530/1530 [==============================] - 11s 7ms/step - loss: 0.6933 - accuracy: 0.4973 - val_loss: 0.6931 - val_accurac
y: 0.5053
Epoch 10/20
1530/1530 [==============================] - 11s 7ms/step - loss: 0.6932 - accuracy: 0.5019 - val_loss: 0.6931 - val_accurac
y: 0.5053
Epoch 11/20
1530/1530 [==============================] - 11s 7ms/step - loss: 0.6932 - accuracy: 0.5000 - val_loss: 0.6931 - val_accurac
y: 0.5053
```

As, we can see it doesn't give a good performance.

Here, I ran the model over 20 epochs (Tried- 10, 30, 50 as well)

And the model automatically topes due to the early stopping technique.

Although, I have tried with different parameters, batch size, input dimension, and epochs, none of them able to give me best accuracy and minim loss.

Finally, Save The Model:

Lastly, I have tried with different ML models, tried Hyper parameter tunning with XGBoost, and ANN as well. So, Finally, decided to move on with the XGBoost model because ML as well as DL models both have the same accuracy accuracy and ML provides more parameters to tune the model for better accuracy in future, If we have more features.