Assignment 2 Task 1

Name: Elroy Chua Ming Xuan **UOW ID:** 7431673 **Data set:** https://www.kaggle.com/datasets/muhammadshahidazeem/customer-churn-dataset

Step 1: Import Necessary Libraries and Load Dataset

In [ ]:
```
import pandas as pd
import numpy as np
# Import data
train_df = pd.read_csv('customer_churn_dataset-training-master.csv')
test_df = pd.read_csv('customer_churn_dataset-testing-master.csv')

# Concatenate train and test data
df = pd.concat([train_df, test_df], axis=0, ignore_index=True)

# Get info on data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 505207 entries, 0 to 505206
Data columns (total 12 columns):
 #   Column             Non-Null Count   Dtype  
---  ------             --------------   -----  
 0   CustomerID         505206 non-null  float64
 1   Age                505206 non-null  float64
 2   Gender             505206 non-null  object 
 3   Tenure             505206 non-null  float64
 4   Usage Frequency    505206 non-null  float64
 5   Support Calls      505206 non-null  float64
 6   Payment Delay      505206 non-null  float64
 7   Subscription Type  505206 non-null  object 
 8   Contract Length    505206 non-null  object 
 9   Total Spend        505206 non-null  float64
 10  Last Interaction   505206 non-null  float64
 11  Churn              505206 non-null  float64
dtypes: float64(9), object(3)
memory usage: 46.3+ MB
```

Step 2: Explore and Preprocess the Data

In [ ]:
```
# Delete the single null row
df = df.dropna()

# Convert the feature types back to int64
for col in df.columns:
    if df[col].dtype == 'float64':
        df[col] = df[col].astype('int64')

# Check for missing values inside the data sets
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 505206 entries, 0 to 505206
Data columns (total 12 columns):
 #   Column             Non-Null Count   Dtype 
---  ------             --------------   ----- 
 0   CustomerID         505206 non-null  int64 
 1   Age                505206 non-null  int64 
 2   Gender             505206 non-null  object
 3   Tenure             505206 non-null  int64 
 4   Usage Frequency    505206 non-null  int64 
 5   Support Calls      505206 non-null  int64 
 6   Payment Delay      505206 non-null  int64 
 7   Subscription Type  505206 non-null  object
 8   Contract Length    505206 non-null  object
 9   Total Spend        505206 non-null  int64 
 10  Last Interaction   505206 non-null  int64 
 11  Churn              505206 non-null  int64 
dtypes: int64(9), object(3)
memory usage: 50.1+ MB
```

Step 3: One Hot Encoding nominal columns

In [ ]:
```
df_encoded = pd.get_dummies(
    df, columns=['Gender', 'Subscription Type', 'Contract Length'], dtype='int64')

print(df_encoded.head())
print()
df_encoded.info()
```

```
   CustomerID  Age  Tenure  Usage Frequency  Support Calls  Payment Delay  \
0           2   39      39               14              5             18   
1           3   65      49                1             10              8   
2           4   55      14                4              6             18   
3           5   58      38               21              7              7   
4           6   23      32               20              5              8   

   Total Spend  Last Interaction  Churn  Gender_Female  Gender_Male  \
0          932                17      1              1            0   
1          557                 6      1              1            0   
2          185                 3      1              1            0   
3          396                29      1              0            1   
4          617                20      1              0            1   

   Subscription Type_Basic  Subscription Type_Premium  \
0                        0                          0   
1                        1                          0   
2                        1                          0   
3                        0                          0   
4                        1                          0   

   Subscription Type_Standard  Contract Length_Annual  \
0                           1                       1   
1                           0                       0   
2                           0                       0   
3                           1                       0   
4                           0                       0   

   Contract Length_Monthly  Contract Length_Quarterly  
0                        0                          0  
1                        1                          0  
2                        0                          1  
3                        1                          0  
4                        1                          0  

<class 'pandas.core.frame.DataFrame'>
Index: 505206 entries, 0 to 505206
Data columns (total 17 columns):
 #   Column                       Non-Null Count   Dtype 
---  ------                       --------------   ----- 
 0   CustomerID                   505206 non-null  int64 
 1   Age                          505206 non-null  int64 
 2   Tenure                       505206 non-null  int64 
 3   Usage Frequency              505206 non-null  int64 
 4   Support Calls                505206 non-null  int64 
 5   Payment Delay                505206 non-null  int64 
 6   Total Spend                  505206 non-null  int64 
 7   Last Interaction             505206 non-null  int64 
 8   Churn                        505206 non-null  int64 
 9   Gender_Female                505206 non-null  int64 
 10  Gender_Male                  505206 non-null  int64 
 11  Subscription Type_Basic      505206 non-null  int64 
 12  Subscription Type_Premium    505206 non-null  int64 
 13  Subscription Type_Standard   505206 non-null  int64 
 14  Contract Length_Annual       505206 non-null  int64 
 15  Contract Length_Monthly      505206 non-null  int64 
 16  Contract Length_Quarterly    505206 non-null  int64 
dtypes: int64(17)
memory usage: 69.4 MB
```

Step 4: Implement Decision Tree Model

In [ ]:
```
# DEFINE NODE CLASS
class Node:
    def __init__(self, feature=None, value=None, left=None, right=None, info_gain=None, leaf_value=None):
        self.feature = feature
        self.value = value
        self.left = left
        self.right = right
        self.info_gain = info_gain

        # leaf nodes
        self.leaf_value = leaf_value

# DEFINE DECISION TREE CLASS


class DecisionTree():
    def __init__(self, min_samples_split=2, max_depth=None, criterion='info_gain'):
        self.root = None
        # stopping conditions
        # minimum number of samples required to split an internal node
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth # maximum depth of the tree
        # criterion to measure the quality of a split (gini_index, gain_ratio, info_gain)
        self.criterion = criterion

    # TYPES OF CRITERION:
    # FOR INFO GAIN
    def entropy(self, y):
        entropy = 0
        unique_values = set(y)

        # get the probability of each value
        for value in unique_values:
            p = sum(y == value) / len(y)
            entropy -= p * np.log2(p)
        return entropy

    def information_gain(self, y, y_left, y_right):
        entropy_parent = self.entropy(y)
        entropy_children = (len(y_left) / len(y)) * self.entropy(y_left) + \
            (len(y_right) / len(y)) * self.entropy(y_right)
        return entropy_parent - entropy_children

    # FOR GINI INDEX
    def gini_index(self, y):
        gini_index = 1
        unique_values = set(y)

        # get the probability of each value
        for value in unique_values:
            p = sum(y == value) / len(y)
            gini_index -= p ** 2
        return gini_index

    # FOR GAIN RATIO
    def gain_ratio(self, y, y_left, y_right):
        information_gain = self.information_gain(y, y_left, y_right)
        split_info = self.entropy(y)
        return information_gain / split_info

    def best_split(self, X, y):
        # best_split = (feature index, split value, information gain)
        best_split = (None, None, 0)

        for feature in range(X.shape[1]):
            X_feature = X[:, feature]
            unique_values = set(X_feature)
            for value in unique_values:
                y_left = y[X_feature <= value]
                y_right = y[X_feature > value]
                if self.criterion == 'gini_index':
                    info_gain = self.gini_index(y) - (len(y_left) / len(y)) * self.gini_index(
                        y_left) - (len(y_right) / len(y)) * self.gini_index(y_right)
                elif self.criterion == 'gain_ratio':
                    info_gain = self.gain_ratio(y, y_left, y_right)
                elif self.criterion == 'info_gain':
                    info_gain = self.information_gain(y, y_left, y_right)
                else:
                    raise ValueError('Invalid criterion')
                if info_gain > best_split[2]:
                    best_split = (feature, value, info_gain)
        return best_split

    def build_tree(self, X, y, depth=0):
        # Prevent further splitting if stopping conditions are met
        if (depth >= self.max_depth) or (len(y) < self.min_samples_split) or (len(np.unique(y)) == 1):
            # convert y to integers before using bincount
            leaf_value = np.bincount(y.astype(int)).argmax()
            return Node(leaf_value=leaf_value)

        # Continue splitting the data if stopping conditions are not met
        feature, value, info_gain = self.best_split(
            X, y) # find the best split point
        X_left, y_left = X[X[:, feature] <= value], y[X[:, feature] <= value]
        X_right, y_right = X[X[:, feature] > value], y[X[:, feature] > value]

        # Recursively build the subtrees
        left = self.build_tree(X_left, y_left, depth + 1)
        right = self.build_tree(X_right, y_right, depth + 1)
        return Node(feature=feature, value=value, left=left, right=right, info_gain=info_gain)

    def fit(self, X, y):
        self.root = self.build_tree(X, y)

    def predict(self, X):
        predictions = [self._predict(x) for x in X]
        return predictions

    def _predict(self, x):
        node = self.root
        while node.leaf_value is None:
            if x[node.feature] <= node.value:
                node = node.left
            else:
                node = node.right
        return node.leaf_value

    def print_tree(self, node=None, depth=0):
        if node is None:
            node = self.root
        if node.leaf_value is not None:
            print(depth * '  ' + 'Prediction', node.leaf_value)
            return
        print(depth * '  ' + 'Feature', node.feature, '<=',
              node.value, 'Info gain:', node.info_gain)
        self.print_tree(node.left, depth + 1)
        self.print_tree(node.right, depth + 1)
```

Step 5: Implement Random Forest Classifier

1. Bootstrapping
2. Feature Selection
3. Tree Construction
4. Prediction

In [ ]:
```
from collections import Counter


class RandomForest:
    def __init__(self, n_trees=10, max_depth=10, min_samples_split=2, n_features=None):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.n_features = n_features
        self.trees = []

    def fit(self, X, y):
        self.trees = []
        for _in range(self.n_trees):
            tree = DecisionTree(max_depth=self.max_depth,
                                min_samples_split=self.min_samples_split)
            X_sample, y_sample = self._bootstrap_samples(X, y)
            tree.fit(X_sample, y_sample)
            self.trees.append(tree)

    def _bootstrap_samples(self, X, y):
        n_samples = X.shape[0]
        idxs = np.random.choice(n_samples, n_samples, replace=True)
        return X[idxs], y[idxs]

    def _most_common_label(self, y):
        counter = Counter(y)
        most_common = counter.most_common(1)[0][0]
        return most_common

    def predict(self, X):
        predictions = np.array([tree.predict(X) for tree in self.trees])
        tree_preds = np.swapaxes(predictions, 0, 1)
        predictions = np.array([self._most_common_label(pred)
                                for pred in tree_preds])
        return predictions
```

Step 6: Run the models

In [ ]:
```
# sample 20% of the data
df = df.sample(frac=0.1, random_state=1)
```

Step 7: Splitting the Dataset into Training and Testing Sets

In [ ]:
```
def train_test_split(df, test_size=0.2):
    if isinstance(test_size, float):
        test_size = round(test_size * len(df))

    # get the indices of the test and train set
    test_indices = np.random.choice(len(df), test_size, replace=False)
    train_indices = np.array(list(set(range(len(df))) - set(test_indices)))

    # split the dataframe into train and test sets
    train_df = df.iloc[train_indices]
    test_df = df.iloc[test_indices]

    # convert the train and test sets into numpy arrays
    X_train = train_df.drop('Churn', axis=1).values
    y_train = train_df['Churn'].values
    X_test = test_df.drop('Churn', axis=1).values
    y_test = test_df['Churn'].values

    return X_train, X_test, y_train, y_test
```

In [ ]:
```
# Split df into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df, test_size=0.2)
```

Step 8: Train the Random Forest

In [ ]:
```
def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy


clf = RandomForest(n_trees=10)
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)

acc = accuracy(y_test, predictions)
print(acc)
```

```
0.9314139316627079
```

Result:

We can see that the accuracy from Random Forest is 0.931413304 with df.sample of 1, n_trees of 10, and runtime of 377m 15.9s