```
###Importing all necessary packages
import os
import matplotlib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import statsmodels.api as sm
from scipy import stats
from IPython.core.interactiveshell import InteractiveShell
from imblearn.over_sampling import RandomOverSampler
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier
from sklearn import preprocessing
from prettytable import PrettyTable
from sklearn.tree import DecisionTreeClassifier,plot_tree
from sklearn import tree
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix, roc_curve, auc
from sklearn.model_selection import StratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.decomposition import TruncatedSVD
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.metrics import mean_squared_error
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules




def specificity_score(y_true, y_pred):
    # Custom function to calculate specificity
    cm = confusion_matrix(y_true, y_pred)
    specificity = cm[0, 0] / (cm[0, 0] + cm[0, 1])
    return specificity


def evaluate_model(model, X_train, y_train, X_test, y_test, classifier_name):
    # Train the model
    model.fit(X_train, y_train)

    # Predictions on the training set
    train_predictions = model.predict(X_train)

    # Predictions on the test set
    test_predictions = model.predict(X_test)
    # Evaluate performance metrics
    train_accuracy = accuracy_score(y_train, train_predictions)
    test_accuracy = accuracy_score(y_test, test_predictions)

    precision = precision_score(y_test, test_predictions)
    recall = recall_score(y_test, test_predictions)
    f1 = f1_score(y_test, test_predictions)

    confusion_mat = confusion_matrix(y_test, test_predictions)

    # ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(y_test, model.predict_proba(X_test)[:, 1])
    roc_auc = auc(fpr, tpr)

    # Specificity
    specificity = specificity_score(y_test, test_predictions)

    # Plot Confusion Matrix
    plt.figure(figsize=(10, 6))
    plt.subplot(1, 2, 1)
    sns.heatmap(confusion_mat, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=['Predicted 0', 'Predicted 1'], yticklabels=['Actual
0', 'Actual 1'])
    plt.title('Confusion Matrix')

    # Plot ROC Curve
    plt.subplot(1, 2, 2)
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC =
{:.2f})'.format(roc_auc))
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc="lower right")

    plt.tight_layout()
    plt.show()

    # Display metrics
    print("Classifier:", classifier_name)
    print("Training Accuracy:", train_accuracy)
    print("Testing Accuracy:", test_accuracy)
    print("Precision:", precision)
    print("Recall:", recall)
    print("F1 Score:", f1)
    print("Specificity:", specificity)
    print("Confusion Matrix:\n", confusion_mat)
    print("ROC AUC:", roc_auc)

    # Create a PrettyTable to append the metrics
    table = PrettyTable()
    table.field_names = ["Classifier", "Training Accuracy", "Testing Accuracy",
"Precision", "Recall", "F1 Score", "Specificity", "ROC AUC"]
    table.add_row([classifier_name, train_accuracy, test_accuracy, precision,
recall, f1, specificity, roc_auc])
    print("\n" + str(table) + "\n")
```

```
# # Libraries Settings
warnings.simplefilter(action="ignore", category=FutureWarning)
warnings.simplefilter(action="ignore", category=Warning)
InteractiveShell.ast_node_interactivity = "all"

np.set_printoptions(suppress=True)
#
#
def set_seed(seed=5805):
    np.random.seed(seed)
    os.environ["PYTHONHASHSEED"] = str(seed)


# ### Reading the Training Dataset
url='https://raw.githubusercontent.com/Ashi911/ML_FTP/main/Data_set/Training
%20Data.csv'
df = pd.read_csv(url)
print(df.info())
print(df.describe())
### EDA  #####
### columsn and rows##
## converting column names to lower case
df.columns = df.columns.str.lower()
rows, columns = df.shape
print('Rows:', rows)
print('Columns:', columns)
print(df.columns)
df.drop('id',inplace=True,axis=1)
print(df.dtypes)
#### cleaning the city column as it has numbers, brackets in between
df.city = df.city.str.extract("([A-Za-z]+)")
df.rename(columns={"married/single":"married_single",},inplace=True)

print(df.isnull().sum())
df.hist(figsize=(20,20))
plt.show()

### Checking uniquness of categorical features
categorical_features=df.select_dtypes(['category','object']).columns
print('Uniquenes of Categorical features ')
for col in categorical_features:
    print(f"{col}:{df[col].nunique()} unique values")
### checking uniqueness of numerical features
numerical_features=df.select_dtypes(['int','float']).columns
print("Uniqueness of Numerical Features")
for col in numerical_features:
    print(f'{col}:{df[col].nunique()} unique values')


### rate of deafulters####
default = df.risk_flag.sum()
total = len(df.risk_flag)
rate_of_default = default / total
print(f"The rate of 'default-on-loan' is {rate_of_default * 100 }% and total number
of defaulter are", default)
```

```
# ### plotting paid vs defaulted ####
# f, ax = plt.subplots(1,2, figsize=(18,8))
# df.Risk_Flag.value_counts().plot.pie(explode=[0,0.1], autopct= "%1.f%%", shadow =
True, ax=ax[0])
# ax[0].set_title("Paid vs Defaulted")
# ax[0].set_ylabel("")
# sns.countplot(x="Risk_Flag", data=df, ax=ax[1])
# ax[1].set_title("Paid vs Defaulted")
# plt.show()



rows, columns = df.shape
print('Rows:', rows)
print('Columns:', columns)
print(df.columns)
### marital status vs riskflag ###
df.rename(columns={"married/single":"married_single",},inplace=True)
df.married_single.value_counts()
married_default_rate = 2636 / 25728
single_default_rate = 28360 / 226272
print(f"defaulters for married is {round(married_default_rate * 100)} %\n"
      f"defaulters for single is {round(single_default_rate * 100)}%")
f, ax = plt.subplots(1,2,figsize=(18,8))
df.married_single.value_counts().plot.bar(ax=ax[0]).set(title = "no. of singles and
married")
sns.countplot(x='married_single' ,hue='risk_flag',data=df, ax=ax[1])
ax[1].set(title="Paid and Default based on Marital status")
plt.show()

### profession ####
print(f"The number of profession in dataset is
{len(df.profession.value_counts())}")
plt.subplots(figsize=(18,8))
df.profession.value_counts().plot.bar()
plt.show()
crosstab_result = pd.crosstab(df['profession'], df['risk_flag'],
margins=True).sort_values(by=1, ascending=False).head()
print(crosstab_result)

######## house owenership ######
print(df.house_ownership.value_counts())
f, ax = plt.subplots(1,2,figsize=(18,8))
df["house_ownership"].value_counts().plot.bar(ax=ax[0])
sns.countplot(x='house_ownership' ,hue='risk_flag',data=df, ax=ax[1])
plt.show()

#### car Ownership #########
print(df.car_ownership.value_counts())
f, ax = plt.subplots(1,2,figsize=(18,8))
df["car_ownership"].value_counts().plot.bar(ax=ax[0])
sns.countplot(x='car_ownership' ,hue='risk_flag',data=df, ax=ax[1])
plt.show()

crosstab_result = pd.crosstab(df['car_ownership'], df['risk_flag'],
margins=True).sort_values(by=1, ascending=False)
print(crosstab_result)

print("defaulters with car:",round((8435/76000) * 100),"%")
print("defaulters with no car:",round((22561/176000) * 100),"%")

sns.catplot(x='house_ownership', y='risk_flag', hue='car_ownership', data=df,
kind='point')
plt.show()

###### state and city features #######

print(df.state.value_counts().shape), print(df.city.value_counts().shape)
print(pd.crosstab(df['state'], df['city'], margins=True))

print(pd.crosstab(df.state, df.risk_flag,
margins=True).style.background_gradient(cmap="summer_r"))
print((pd.crosstab(df.state, df.risk_flag, margins=True)[1] / pd.crosstab(df.state,
df.risk_flag, margins=True)["All"]).sort_values(ascending=False).head())

print((pd.crosstab(df['city'], df['risk_flag'], margins=True)[1] /
pd.crosstab(df['city'], df['risk_flag'], margins=True)
["All"]).sort_values(ascending=False).head())

####### Exploring Income ########
print('Highest Income is:',df.income.max())
print('Lowest Income is:',df.income.min())
print('Average Income is:',df.income.mean())

f , ax = plt.subplots(1,2,figsize=(18,8))
sns.distplot(df.income, ax=ax[0]);
sns.boxplot(df.income, ax=ax[1]);
plt.show()

### Exploring Age #####
print('Highest age is:',df.age.max())
print('Lowest age is:',df.age.min())
print('Average age is:',df.age.mean())

f, ax = plt.subplots(1,2, figsize=(18,8))
sns.distplot(df.age, ax=ax[0]);
sns.boxenplot(df.age, ax=ax[1]);
plt.show()

#### Exploring Experience ########
print(pd.crosstab(df['risk_flag'], df['experience'], margins=True))
pd.crosstab(df['experience'], df['risk_flag']).plot.bar(width=0.9)
plt.title("Number of loan takers according to Experience")
plt.show()

# Factor plot
sns.catplot(x='experience', y='risk_flag', data=df, kind='point')
plt.title("Default rate vs Experience")
plt.show()

#### Exploring Current job years #####
print(pd.crosstab(df['risk_flag'],df['current_job_yrs']))

pd.crosstab(df['current_job_yrs'] ,df['risk_flag'] ).plot.bar(width=0.9)
plt.title("Number of loan takers according to Current Job")
plt.show()
```

```
sns.catplot(x='current_job_yrs', y='risk_flag', data=df, kind='point')
plt.title("Default rate vs Current Job Years")
plt.show()

#### Exploring Housing years ####
print(pd.crosstab(df['risk_flag'],df['current_house_yrs']))

pd.crosstab(df['current_house_yrs'] ,df['risk_flag'] ).plot.bar(width=0.9)
plt.title("Number of loan takers according to Current House Years")
plt.show()


sns.catplot(x='current_house_yrs', y='risk_flag', data=df, kind='point')
plt.title("Default rate vs Current House Years")
plt.show()

##Correlation and Covariance  matrices
fig, ax = plt.subplots( figsize = (12,8) )
corr_matrix = df[numerical_features].corr()
corr_heatmap = sns.heatmap( corr_matrix, cmap = "YlGnBu", annot=True, ax=ax,
annot_kws={"size": 14})
plt.show()

fig, ax = plt.subplots( figsize = (12,8) )
cov_matrix=df[numerical_features].cov()
cov_heatmap=sns.heatmap(cov_matrix,cmap = "YlGnBu", annot=True, ax=ax,
annot_kws={"size": 14})
plt.show()
#
# #----->  if we observe there is a correlation between experience and
curr_job_years so we can drop any one feature
df.drop('experience',inplace=True,axis=1)

# ### balancing the data ####
df["risk_flag"].hist(figsize=(20,20))
plt.show()
df.drop_duplicates(inplace=True)
### as we can see the data is not balanced ####
### we are oversampling ###
X = df.drop("risk_flag",axis = 1)
y = df["risk_flag"]
rs=RandomOverSampler()
X,y=rs.fit_resample(X,y)
y = pd.Series(y)
value_counts = y.value_counts()
plt.bar(value_counts.index, value_counts.values)
plt.xlabel('Classes')
plt.ylabel('Count')
plt.title('Distribution of Resampled "y" Data')
plt.show()
df=pd.concat([X,y],axis=1)
df_bal=df
print(df.risk_flag.value_counts())


####now the data is balance
##### as we know the state and city and profession features have a lot of
cardinality so i will be using label encoder
label_encode_columns=['profession','city','state']
```

```
label_encoder = preprocessing.LabelEncoder()
for column in label_encode_columns:
    df[column] = label_encoder.fit_transform(df[column])

### one hot encoding for features with less cardinality #####
df_one_hot_encode=pd.get_dummies(df,columns=['married_single','house_ownership','ca
r_ownership'],drop_first=True)
### features values with True and False to 1 and 0 ###
features_to_convert = ['married_single_single','house_ownership_owned',
'house_ownership_rented', 'car_ownership_yes']
for feature in features_to_convert:
    df_one_hot_encode[feature] = df_one_hot_encode[feature].astype(int)
print(df_one_hot_encode.columns)
non_encoded_features=df_one_hot_encode.select_dtypes(['int','float']).columns
print(non_encoded_features)

## standardizing and splitting the dataset
scaler=StandardScaler()
df_standard=df_one_hot_encode;
df_standard[non_encoded_features]=scaler.fit_transform(df_one_hot_encode[non_encode
d_features])
df_stan=df_standard


df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805
)
print(df_train.head())
print(df_test.head())
dependent_variable='risk_flag'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]

print(df_standard.columns)
### Performing PCA
df_standard=df_stan
X_scaled = df_standard.drop(columns='risk_flag')
pca = PCA()
X_pca = pca.fit_transform(X_scaled)
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance_ratio = np.cumsum(explained_variance_ratio)
n_components_range = np.arange(1, 11)
n_components_needed = np.argmax(cumulative_variance_ratio >= 0.85) + 1
print("Number of components needed to explain 85% of the variance:",
n_components_needed)
plt.figure(figsize=(10,8))
x=np.arange(1,12,step=1)
plt.plot(x,cumulative_variance_ratio,marker='o', linestyle='--', color='b')
plt.xlabel('Number of Components')
plt.ylabel('Cummulative Variance %')
plt.xticks(np.arange(1, 12, step=1))
plt.axhline(y=0.90, color='b', linestyle='-')
plt.axvline(x=9,color='r',linestyle='-')
plt.grid(True)
plt.show()

## number of components needed are 4 to explain the 90% of the variace

### Performing the Random Forest Analysis for feature importance

X_rf=df_one_hot_encode.drop(columns=[dependent_variable])
y_rf=df[dependent_variable]
rf_model = RandomForestRegressor(n_estimators=100, random_state=5805)
rf_model.fit(X_rf, y_rf)
feature_importances = rf_model.feature_importances_
feature_labels = X_rf.columns
### sorting feature Importances ###
sorted_indices = np.argsort(feature_importances)[::-1]
sorted_importances = feature_importances[sorted_indices]
sorted_labels = feature_labels[sorted_indices]
### plot the feature importance ##
plt.figure(figsize=(18,10))
plt.barh(range(len(sorted_labels)),sorted_importances,align='center')
plt.yticks(range(len(sorted_labels)),sorted_labels)
plt.xlabel('Feature Importance')
plt.ylabel('Features')
plt.title('Feature Importances (Random Forest)')
plt.gca().invert_yaxis()
plt.show()

####### performing SVD ###############
df_standard=df_stan
svd = TruncatedSVD(n_components=len(df_standard.columns))
svd_model = svd.fit(df_standard)
i=0
for i, singular_value in enumerate(svd_model.singular_values_):
    variance_ratio = svd_model.explained_variance_ratio_[i]
    print(f"Feature {df_standard.columns[i]}:")
    print(f"\tSingular value: {singular_value:.2f}")
    print(f"\tVariance ratio: {variance_ratio:.2f}")

### Print the top 2 left singular vectors ####
print(f"Top 2 left singular vectors:\n{pd.DataFrame(svd_model.components_.T[:2],
columns=df_stan.columns)}")

######## VIF ################
# Create an empty list to store VIFs
vif_values = []
df_standard = df_stan

# Loop through each feature and calculate VIF
for i in range(len(df_standard.columns)):
    vif = variance_inflation_factor(df_standard.values, i)
    vif_values.append(vif)

# Create a DataFrame to store features and VIF values
vif_df = pd.DataFrame({"Feature": df_standard.columns, "VIF": vif_values})

# Print the VIF values
print(vif_df)


########### phase 2 ######################
```

```
### performing regression analysis###

# ##  splitting the dataset into train and test
df_standard=df_stan
print(df_standard.head())
print(df_standard.columns)
df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805)

#### Performing backward  Regression ###
dependent_variable='income'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]

### creating an array for droped features ###
dropped_features=[]

## adding constant
X_model_train=sm.add_constant(X_train)
X_test=sm.add_constant(X_test)

## training Model
model=sm.OLS(y_train,X_model_train).fit()
print(model.summary())

#### Creating PrettyTables ###
feature_names=X_model_train.columns
summary_table=PrettyTable()
summary_table.field_names=['process Update','AIC','BIC','Adj R^2','p-value']
compare_table=PrettyTable()
compare_table.field_names=['Model','R^2','Adj-R^2','AIC','BIC','MSE']

### adding initial summary to pretty table ###
p_value = model.pvalues['const']
summary_table.add_row(['const',round(model.aic,3),round(model.bic,3),round(model.rs
quared_adj,3),round(p_value, 3)])
#
### Removing const ###
X_model_train.drop(columns=['const'],inplace=True)
dropped_features.append('const')
model=sm.OLS(y_train,X_model_train).fit()
p_value = model.pvalues['house_ownership_owned']
summary_table.add_row(['house_ownership_owned',round(model.aic,3),round(model.bic,3
),round(model.rsquared_adj,3),round(p_value, 3)])
print(model.summary())
#
### Removing house_ownership_owned ###
X_model_train.drop(columns=['house_ownership_owned'],inplace=True)
dropped_features.append('house_ownership_owned')
model=sm.OLS(y_train,X_model_train).fit()
p_value = model.pvalues['city']
summary_table.add_row(['city',round(model.aic,3),round(model.bic,3),round(model.rsq
uared_adj,3),round(p_value, 3)])
```

```
print(model.summary())
#### Removing city ######
X_model_train.drop(columns=['city'],inplace=True)
dropped_features.append('city')
model=sm.OLS(y_train,X_model_train).fit()
p_value = model.pvalues['risk_flag']
summary_table.add_row(['risk_flag',round(model.aic,3),round(model.bic,3),round(mode
l.rsquared_adj,3),round(p_value, 3)])
print(model.summary())

#### Removing Risk_Flag ####
X_model_train.drop(columns=['risk_flag'],inplace=True)
dropped_features.append('risk_flag')
model=sm.OLS(y_train,X_model_train).fit()
p_value = model.pvalues['current_job_yrs']
summary_table.add_row(['current_job_yrs',round(model.aic,3),round(model.bic,3),roun
d(model.rsquared_adj,3),round(p_value, 3)])
print(model.summary())

#### Removing Current Job Years ####
X_model_train.drop(columns=['current_job_yrs'],inplace=True)
dropped_features.append('current_job_yrs')
model=sm.OLS(y_train,X_model_train).fit()
p_value = model.pvalues['current_house_yrs']
summary_table.add_row(['current_house_yrs',round(model.aic,3),round(model.bic,3),ro
und(model.rsquared_adj,3),round(p_value, 3)])
print(model.summary())
######## Removing current house years ####
X_model_train.drop(columns=['current_house_yrs'],inplace=True)
dropped_features.append('current_house_yrs')
model=sm.OLS(y_train,X_model_train).fit()
p_value = model.pvalues['profession']
summary_table.add_row(['profession',round(model.aic,3),round(model.bic,3),round(mod
el.rsquared_adj,3),round(p_value, 3)])
print(model.summary())
######## Removing Profession ####
X_model_train.drop(columns=['profession'],inplace=True)
dropped_features.append('prefession')
model=sm.OLS(y_train,X_model_train).fit()
p_value = model.pvalues['age']
summary_table.add_row(['age',round(model.aic,3),round(model.bic,3),round(model.rsqu
ared_adj,3),round(p_value, 3)])
print(model.summary())

#### Removing age ####
X_model_train.drop(columns=['age'],inplace=True)
dropped_features.append('age')
model=sm.OLS(y_train,X_model_train).fit()
p_value = model.pvalues['state']
summary_table.add_row(['state',round(model.aic,3),round(model.bic,3),round(model.rs
quared_adj,3),round(p_value, 3)])
print(model.summary())
#### removing state###
X_model_train.drop(columns=['state'],inplace=True)
dropped_features.append('state')
model=sm.OLS(y_train,X_model_train).fit()

print(model.summary())
```

```
final_features=X_model_train.columns
X_model_test=X_test[final_features]
y_pred=model.predict(X_model_test)
y_scaler=StandardScaler()
y_scaler.fit_transform(df_stan[['income']])
y_test_destand=y_scaler.inverse_transform(y_test.values.reshape(-1,1)).flatten()
y_pred_2d=y_pred.values.reshape(-1,1)
y_pred_destand=y_scaler.inverse_transform(y_pred_2d)
mse = mean_squared_error(y_test_destand, y_pred_destand)
y_pred_train=model.predict(X_model_train)
y_scaler=StandardScaler()
y_scaler.fit_transform(df_stan[['income']])
y_train_destand=y_scaler.inverse_transform(y_train.values.reshape(-1,1)).flatten()
y_pred_2d_train=y_pred_train.values.reshape(-1,1)
y_pred_destand_train=y_scaler.inverse_transform(y_pred_2d_train)
mse = mean_squared_error(y_train_destand, y_pred_destand_train)
# Print the result
print(f"Mean Squared Error: {mse}")
predictions = model.predict(X_model_test)
pred_int = model.get_prediction(X_model_test).summary_frame(alpha=0.05)
predictions_original=y_scaler.inverse_transform(predictions.values.reshape(-
1,1)).flatten()
# Extract the lower and upper prediction interval bounds
lower_pred = pred_int['obs_ci_lower']
upper_pred = pred_int['obs_ci_upper']
lower_pred_orig=y_scaler.inverse_transform(lower_pred.values.reshape(-
1,1)).flatten()
upper_pred_orig=y_scaler.inverse_transform(upper_pred.values.reshape(-
1,1)).flatten()

# Reverse any transformations on the predictions if needed
# For example, if you've transformed the target variable during modeling, you may
need to reverse the transformation here.

# Create a plot of predictions
plt.figure(figsize=(10, 6))
plt.plot(predictions_original, label="Predicted Sales", color='red')
plt.fill_between(np.arange(len(predictions)), lower_pred_orig, upper_pred_orig,
color='gray', alpha=0.5,
                 label="95% Prediction Interval")
plt.legend()
plt.xlabel("Observation")
plt.ylabel("Sales")
plt.title("Sales Prediction with 95% Prediction Interval")
plt.grid(True)
plt.show()
#
# ### plot graph between y_pred_destand and y_test_destand ######

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 8))

# Plot Test Sales Values
plt.plot(y_test_destand, label='Test Sales Values', color='blue')

# Plot Predicted Test Sales Values
plt.plot(y_pred_destand, label='Predicted Test Sales Values', linestyle='dashed',
color='orange')
```

```
# Plot Train Sales Values
plt.plot(y_train_destand, label='Train Sales Values', linestyle='dotted',
color='green')

# Plot Predicted Train Sales Values
plt.plot(y_pred_destand_train, label='Predicted Train Sales Values',
linestyle='dashdot', color='red')

plt.xlabel('Samples')
plt.ylabel('Sales')
plt.title('Test and Train Sales VS Predicted Sales Values')
plt.grid(True)
plt.legend()
plt.show()

# print(summary_table)

#
# # #
# # # ############## phase 3 ################
df_standard=df_stan
stratified_kfold=StratifiedKFold(n_splits=5,shuffle=True,random_state=5805)
df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805)
dependent_variable='risk_flag'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]
#
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

evaluate_model(model, X_train, y_train, X_test, y_test,'Decision Tree')

feature_importance = model.feature_importances_
feature_importance_dict = dict(zip(independent_variables, feature_importance))
sorted_feature_importance = sorted(feature_importance_dict.items(), key=lambda x:
x[1], reverse=True)
print("Feature Importance:")
for feature, importance in sorted_feature_importance:
    print(f"{feature}: {importance:.2f}")
plt.figure(figsize=(20,12))
tree.plot_tree(model,rounded=True,filled=True)
plt.show()

# # # #### gridsearch #####
tuned_parameters = {
    'max_depth': [7, 9, 11,13, None],
    'min_samples_split': [4,6,7, 9, 12],
    'min_samples_leaf': [1, 2, 4, 6, 8],
    'max_features': ['auto', 'sqrt', 'log2', None],
    'splitter': ['best', 'random'],
    'criterion': ['gini', 'entropy']
}
```

```
grid_model = GridSearchCV(
    DecisionTreeClassifier(),
    tuned_parameters,
    cv=stratified_kfold,
    scoring='accuracy'
)
grid_model.fit(X_train, y_train)


best_params = grid_model.best_params_
best_accuracy = grid_model.best_score_

print("Best Parameters:", best_params)
evaluate_model(grid_model, X_train, y_train, X_test, y_test,'Descion Tree
Prepruned')

best_model = grid_model.best_estimator_
plt.figure(figsize=(20, 12))
tree.plot_tree(best_model, rounded=True, filled=True)
plt.show()
accuracies = []


path = model.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

for ccp_alpha in ccp_alphas:
    pruned_model = DecisionTreeClassifier(ccp_alpha=ccp_alpha, random_state=5805)
    pruned_model.fit(X_train, y_train)
    y_pred = pruned_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)


plt.figure(figsize=(10, 6))
plt.title('Accuracy vs Effective Alpha for Decision Tree')
plt.xlabel('Effective Alpha')
plt.ylabel('Accuracy')
plt.plot(ccp_alphas, accuracies, marker='o', drawstyle="steps-post")
plt.show()


optimal_alpha = ccp_alphas[accuracies.index(max(accuracies))]


pruned_model = DecisionTreeClassifier(ccp_alpha=optimal_alpha, random_state=5805)
pruned_model.fit(X_train, y_train)

evaluate_model(pruned_model, X_train, y_train, X_test, y_test,'Decision Tree
Postpruned')
plt.figure(figsize=(15, 10))
tree.plot_tree(pruned_model, filled=True, rounded=True, class_names=True)
plt.show()



# #
# # # # ######### Logistic Regression #######
# #

df_standard=df_stan
df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805)
dependent_variable='risk_flag'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]
param_grid={
    "C": [0.001, 0.01, 0.1, 1, 10, 100],
    "solver": ["newton-cg", "lbfgs", "liblinear", "sag", "saga"],
    "penalty": ["none", "l1", "l2", "elasticnet"],
}
log_model=LogisticRegression(random_state=5805)
grid_model=GridSearchCV(log_model,param_grid,cv=stratified_kfold,scoring='accuracy'
)
grid_model.fit(X_train, y_train)
print("Best Hyperparameters:", grid_model.best_params_)


evaluate_model(grid_model, X_train, y_train, X_test, y_test,'Logistic Regression')
#
# # # ######## Random Forest ############
df_standard=df_stan
df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805)
dependent_variable='risk_flag'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]
param_grid={
    "n_estimators": [5,10,15,20],
    "max_features": ['auto', 'sqrt', 'log2'],
    "criterion": ['gini', 'entropy'],
    "max_depth":[None, 5, 10, 20, 50],
}
rf_model = RandomForestClassifier(class_weight="balanced", random_state=5805)
grid_model=GridSearchCV(rf_model,param_grid,cv=stratified_kfold,scoring='accuracy')
grid_model.fit(X_train, y_train)
print("Best Hyperparameters:", grid_model.best_params_)
evaluate_model(grid_model, X_train, y_train, X_test, y_test,'Random Forest')





# ### KNN algorithm ##########
df_standard=df_stan
df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805)
```

```
dependent_variable='risk_flag'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]


def train_knn_model(X_train, y_train, X_test, y_test, k):
    knn_model = KNeighborsClassifier(n_neighbors=k)
    knn_model.fit(X_train, y_train)
    y_pred = knn_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy
# Perform the elbow method to find optimal k
k_values = range(1, 21)  # Try different values of k
accuracies = []

for k in k_values:
    accuracy = train_knn_model(X_train, y_train, X_test, y_test, k)
    accuracies.append(accuracy)
# Plot the accuracy against different values of k
plt.plot(k_values, accuracies, marker='o')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.show()

# Find the optimal k (where accuracy plateaus)
optimal_k = k_values[np.argmax(accuracies)]
print(f'Optimal k: {optimal_k}')

# Train the final KNN model with the optimal k
final_knn_model = KNeighborsClassifier(n_neighbors=optimal_k)
final_knn_model.fit(X_train, y_train)

evaluate_model(final_knn_model, X_train, y_train, X_test, y_test,"KNN")


## Performing Naive Bayes #####
df_standard=df_stan
df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805)
dependent_variable='risk_flag'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]

NB_model = GaussianNB()

NB_model.fit(X_train, y_train)

evaluate_model(NB_model, X_train, y_train, X_test, y_test,'Naive Bayes')
```

```
#
#
#
# #
# #### Perfroming SVM ######
#
df_standard=df_stan
df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805)
dependent_variable='risk_flag'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]
param_grid={
    "C": [0.001, 0.01, 0.1, 1],
    "kernel": ["linear", "rbf", "poly"],
    "gamma": [ 0.01, 0.1, 1],
    "degree": [2, 3, 4],
}
svm_model = SVC(probability=True)
grid_model=GridSearchCV(svm_model,param_grid,cv=stratified_kfold,scoring='accuracy'
)
grid_model.fit(X_train, y_train)
print("Best Hyperparameters:", grid_model.best_params_)
evaluate_model(grid_model, X_train, y_train, X_test, y_test,'SVM')


#### Performing MLP classifier ######
df_standard=df_stan
df_train,df_test=train_test_split(
    df_standard,test_size=0.20,shuffle=True,random_state=5805)
dependent_variable='risk_flag'
independent_variables=[col for col in df_train.columns if col !=
dependent_variable]

X_train=df_train[independent_variables]
X_test=df_test[independent_variables]
y_train=df_train[dependent_variable]
y_test=df_test[dependent_variable]
param_grid={
    "hidden_layer_sizes":  [(100, ), (100, 50), (100, 50, 25)],
    "activation": ['relu', 'tanh', 'logistic'],
    "solver": ['adam', 'sgd'],
    "alpha": [ 0.001, 0.01],
}
mlp_model = MLPClassifier(random_state=5805)
grid_model=GridSearchCV(mlp_model,param_grid,cv=stratified_kfold,scoring='accuracy'
)
grid_model.fit(X_train, y_train)
print("Best Hyperparameters:", grid_model.best_params_)
evaluate_model(grid_model, X_train, y_train, X_test, y_test,' Neural
Networks(MLP)')


# # ######### K Means  Clustering ###
```

```
df_standard=df_stan

def get_silhouette_score(df_standard, k):
    kmeans = KMeans(n_clusters=k, random_state=5805)
    kmeans.fit(df_standard)
    return silhouette_score(df_standard, kmeans.labels_)

# Calculate silhouette score for a range of k values
k_range = range(2, 11)
silhouette_scores = [get_silhouette_score(df_standard, k) for k in k_range]

## plotting the silhouette score###
plt.plot(k_range, silhouette_scores)
plt.xlabel('Number of clusters (k)')
plt.ylabel('Silhouette score')
plt.show()

max_score_index = np.argmax(silhouette_scores)
best_k = k_range[max_score_index]
print(f"Best k based on silhouette score: {best_k}")
 # Replace with the determined optimal k
kmeans_final = KMeans(n_clusters=2, init='k-means++', random_state=5805)
kmeans_final.fit(df_standard)
clusters = kmeans_final.predict(df_standard)


# Count the occurrences of each cluster in the training and test sets
train_cluster_counts = np.bincount(clusters)


# Create a bar plot for the training set
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.barplot(x=np.arange(len(train_cluster_counts)), y=train_cluster_counts)
plt.title('Cluster Distribution in Training Set')
plt.xlabel('Cluster')
plt.ylabel('Count')


plt.show()
#
# # silhouette_score(df_standard, cluster_labels)
# # #
# # # ##### performing Apriori ###
df_balanced=df_bal
df_ap = df_balanced[['income', 'age', 'current_job_yrs', 'current_house_yrs']]
df_ap['income'] = pd.cut(df_ap['income'], bins=3, labels=['low', 'mid', 'high'])
df_ap['age'] = pd.cut(df_ap['age'], bins=3, labels=['young', 'mid_age', 'old'])
df_ap['current_job_yrs'] = pd.cut(df_ap['current_job_yrs'], bins=2,
labels=['fresher', 'experienced'])
df_ap['current_house_yrs'] = pd.cut(df_ap['current_house_yrs'], bins=2,
labels=['recent_move_in', 'stayed_long'])
te = TransactionEncoder()
te_ary = te.fit(df_ap).transform(df_ap)
df_ap = pd.DataFrame(te_ary, columns=te.columns_)
print(df_ap.head(5))
# convert the data into binary format
df_ap= df_ap.astype('int')
print(df_ap.head(5))
```

```
frequent_itemsets = apriori(df_ap, min_support=0.00001, use_colnames=True,
verbose=1)
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=0.00001)
rules = rules.sort_values(['confidence'], ascending=False)
print(rules.head(5).to_string())
```