Table of Contents:

Table of Figures and Table

Abstract

This project is focused on showing machine learning applications on IMDB Movie Rating dataset with multiple genres where regression, classification and clustering methods are applied to extract meaningful insights. The analysis has four phases: feature engineering and exploratory data analysis, regression analysis, classification analysis, and clustering with association rule mining.   
In Phase I: Following a rigorous treatment of the missing value and duplicate values, handle unwanted values, detection of outliers, and application of dimensional reduction to ensure strength in the dataset, the model went for serious data pre-processing. Feature transformations, like normalization and encoding, were done to help the model perform optimally.  
Phase II: Multiple linear regression to predict a continuous variable from the dataset was done, providing an insight into feature interactions and model performance metrics such as R-squared and AIC/BIC. In Phase III, advanced classification algorithms such as Decision Trees, Random Forests, Support Vector Machines, and Neural Networks were used to classify movie genres. Each model was hyperparameter-tuned using grid search, with performance evaluated through precision, recall, F-score, ROC curves, and AUC. Among these, the best performing model showed high accuracy in classification, making it suitable for the task at hand.  
Finally, the clustering algorithms-variety K-means and DBSCAN-extracted the hidden pattern in the data, while association rule mining identified relationships between variables that are significant. This provides insight into the factors governing genre classification and recommendations for any future improvement in the classes.

This project is one of the practical uses of machine learning on real-world data and could serve as a guide on how complex classification and clustering problems may be approached for the entertainment industry.

Introduction

The key purpose of the project is to illustrate an example of the application of Machine Learning techniques on a real-world dataset, which deals with feature engineering, classification, clustering, and association rule mining. A multi-genre movie dataset was selected containing attributes on production, popularity, financial metrics, among other target variables of movie genres. These attributes offer fertile ground for analytics exploration and predictive modeling.  
  
This report is structured to reflect the phases of the project. Phase I will cover feature engineering and EDA: data cleaning, dimensionality reduction, and outlier detection to prepare the dataset for analysis. Phase II includes regression analysis with evaluations, step wise elimination and prediction of a continuous variable. Phase III includes various classification models with proper evaluation matrices, hyperparameter tuning and predictions on multi class classification. Finally at phase IV, clustering and association rule mining were implemented.  
This project’s objective is meet all the academic requirements for CS 5805 as well as making an impact on entertainment industry using machine learning.

Description of the Dataset:  
The IMDB Rating dataset used in this project is consists of 16 different datasets, 1 for each genre. The new dataset was combined with a total observation of 368,300 observations and 14 features. The dataset includes a diverse range of attributes such as details of the production, popularity and financial metrics as well as audience reactions. For phase I the independent variable is “rating” and for phase II the target variable is “genre” which has multiple labels.

The combined dataset consists of both numerical and categorical features, making it suitable for various machine learning tasks such as classification and clustering. The features were:

* Numerical Features: rating, year, gross (in $), runtime, votes
* Categorical Features: directors, stars, movie name, description etc.

The dataset was pre-processed for handling missing values and duplicate values, encoded and transformed (standardized/normalized). The huge variation in the dataset provides an opportunity to analyze the data and use different kinds of machine learning models.  
The datasets were sourced from (https://www.kaggle.com/datasets/rajugc/imdb-movies-dataset-based-on-genre/data), ensuring transparency and reproducibility of results.

Phase I:  
Data Preprocessing:  
Data preprocessing was performed to ensure the quality of the data so that it is suitable for the machine learning models.

* Handling Duplicate Values: Duplicates were identified and removed to sustain redundancy of the data
* Handling Unwanted Values: “year”, “director”, “star” had unwanted values which was removed from the dataset
* Handling Null values: Features with more than 50% null values were removed from the dataset. Every null value was handled and dropped.
* Handling not Important Features: Dropped statistically not important values like “star\_id”, “director\_id”, “description” etc.
* Convert Categorical to Numerical: “year” and “runtime” were converted to numerical features.
* Balancing class: balanced the imbalance class using up sampling and down sampling methods.
* Outliers Detection and Removal: Used IQR, Z score, LOF and Isolation Forest to remove outliers.

Dimensionality Reduction and Feature Selection:

* Variance Inflation Factor: Using VIF the multicollinearity of the dataset was removed
* Principal Component Analysis: Used to reduce dimensionality to retain maximum variance
* Singular Values Decomposition: Used to analyze linear relationships between features
* Random Forest Analysis: Used to analyze top features with feature importance

Encoding and Transformation:

* Discretization and Binarization: Used label encoding to encode categorical features excluding “genre”. Used one hot encoding on “genre”
* Normalization/ Standardization: Used both normalization and standardization on numerical features

Exploratory Data Analysis:

* Heatmap: Heatmap to see the correlation and covariance matrix.
* Imbalanced Data: Bar graph to see the imbalance between the “genre”

Figure 1.1: Genre Distribution Before Balancing

A graph of a number of different colored bars

Description automatically generated with medium confidence

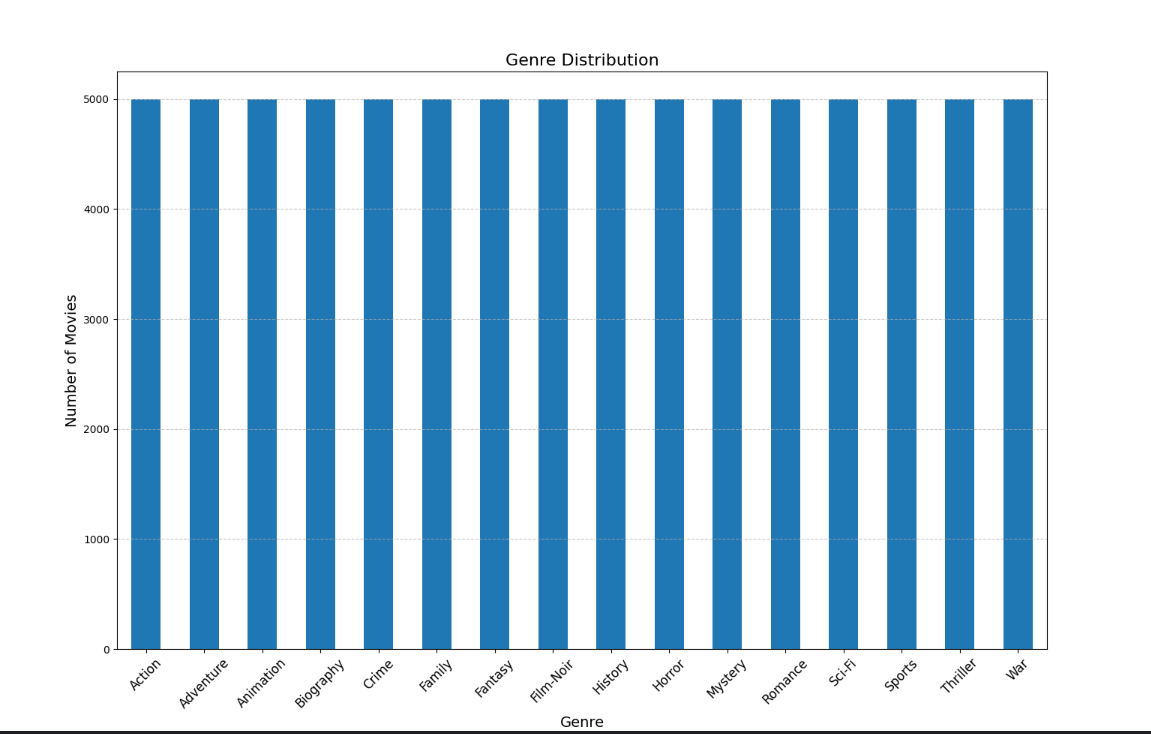
Figure 1.2: Genre Distribution after Balancing the Dataset

Figure 1.3: Covariance Matrix of Standardized Data

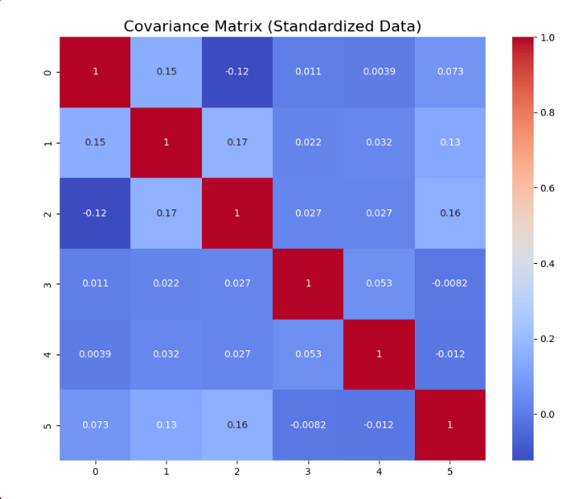
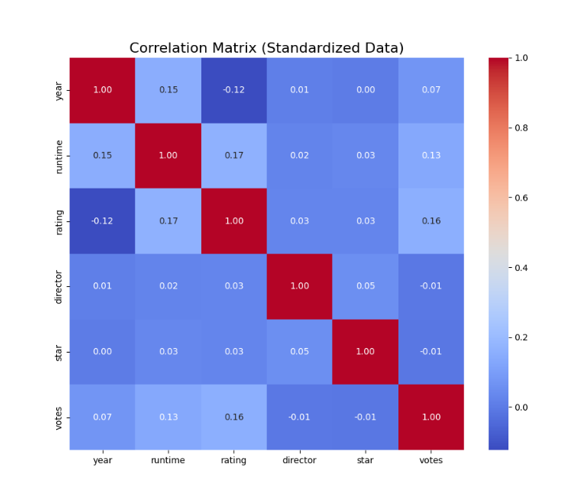


Figure 1.4: Correlation Matrix of Standardized Data



Phase II

Regression model was used to predict continuous value “rating” based on other features. The data was split to an 80:20 ratio with a training set of 80% of the data and testing set with 20% of the data.   
Statistical Analysis:

* T-Test: Done T test analysis
* F-Test: Found F test score to be high which proves these data is statistically important
* Confidence Interval: Found the confidence interval for each data

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Figure 2.1: T-test and Confidence Interval

Stepwise Regression was done with backward elimination to find the important features. After stepwise regression 11 features were selected.  
Evaluation Metrices:

* R Squared: 0.123
* Adjusted R Squared: 0.123
* AIC: 1.937e+05
* BIC: 1.938e+05

A screenshot of a computer program

Description automatically generated

Figure 2.2: Evaluation Metrics for Stepwise Regression

The model has the same value as R-squared and Adj. R-squared, which proves adding value does not affect the model significantly. Moreover, the low R-squared shows that the model cannot properly explain the variance of the model. While the model is statistically significant according to the F-test, it does not explain the variance of the dependent variable which may be because the predictor and dependent variables are not linear.

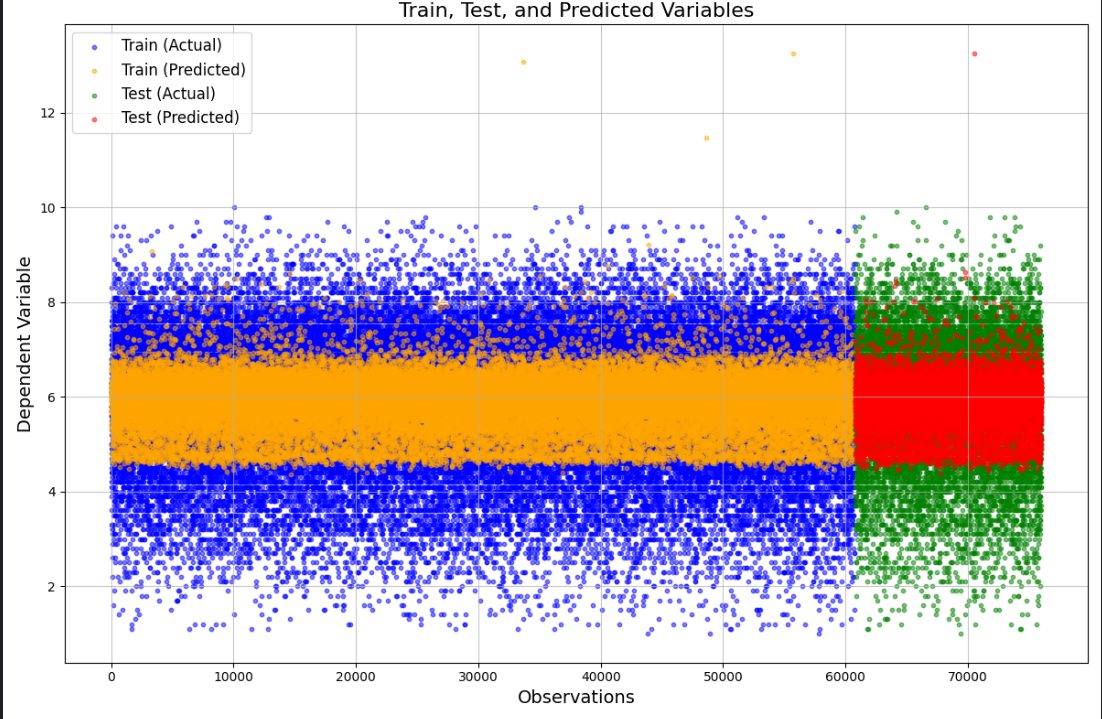
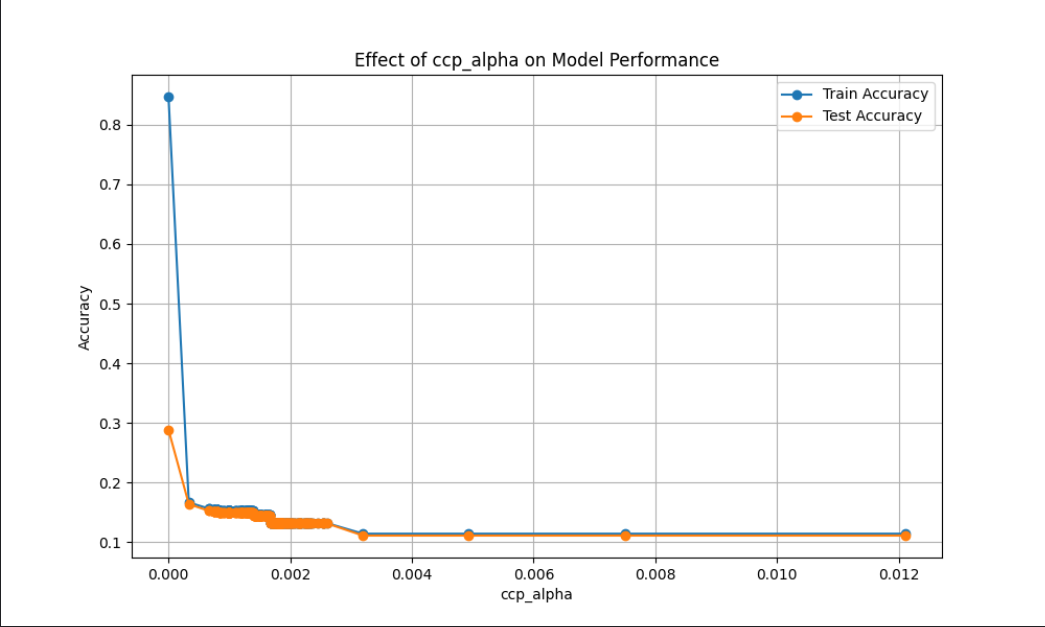


Figure 2.3: Final Look at the model

Phase III

Classification model was used to classify “genre”. “genre” has multiple classes and multiclass classifications was implemented. Training and testing set had 80:20 ratio of data.

* Decision Tree:
  + Base Model: base model had accuracy of 28.85%
  + Pre-pruning: pre-pruned tree had accuracy of 29%
  + Optimal Parameters: used grid search to find the best parameters to use on pre-pruned tree.
  + Post-pruning: first found the optimal ccp\_alpha =0, as a result have same result for both pre- and post-pruning. Also, found out how ccp\_alpha affects the train and test set (Figure 3.1.0).
  + Confusion Matrix (Figure 3.1.1): Displayed the confusion matrix as shown in the figure (Figure 3.1.1) below. The diagonal values show the true positive values.
  + Classification Report (Figure 3.1.2): For each class (class 0-15) classification report shows the precision, recall, F-1 score and support behind it.
  + ROC Curve and AUC (Figure 3.1.3): To assess the model’s discriminatory abilities, ROC Curve and AUC score were produced.
  + Stratified K-Fold Cross Validation (Figure 3.1.4): Chose k= 5, showed mean accuracy and standard deviation.

Figure 3.0.0: Effects of CCP\_alpha on the model

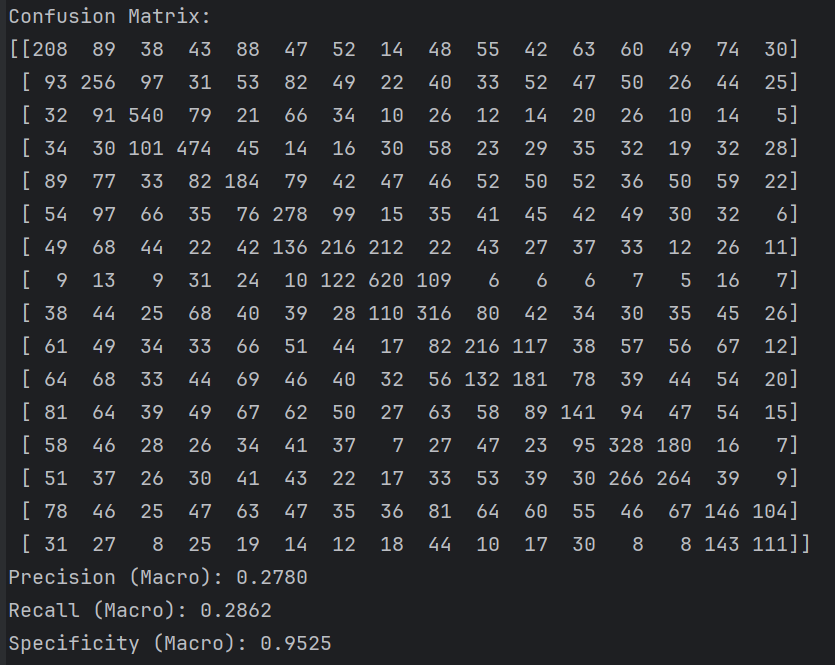


Figure 3.1.1: Confusion Matrix For both Pre and Post pruned model

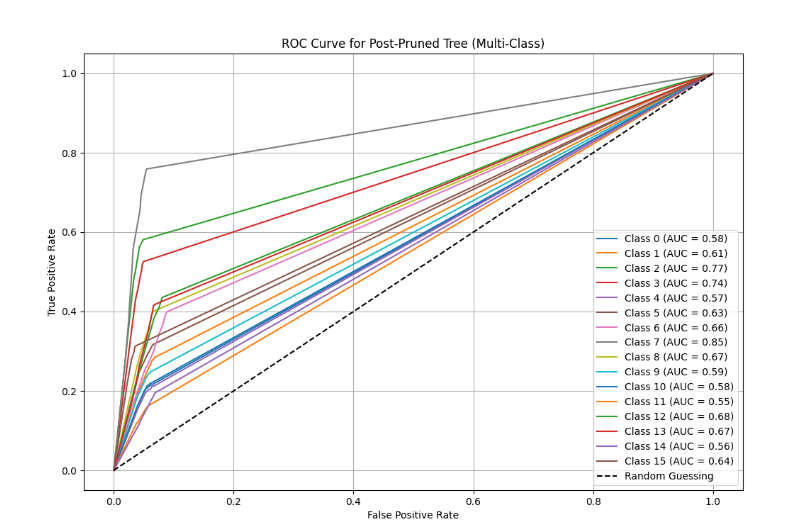


Figure 3.1.2.0: Pre-pruned model ROC AUC Curve

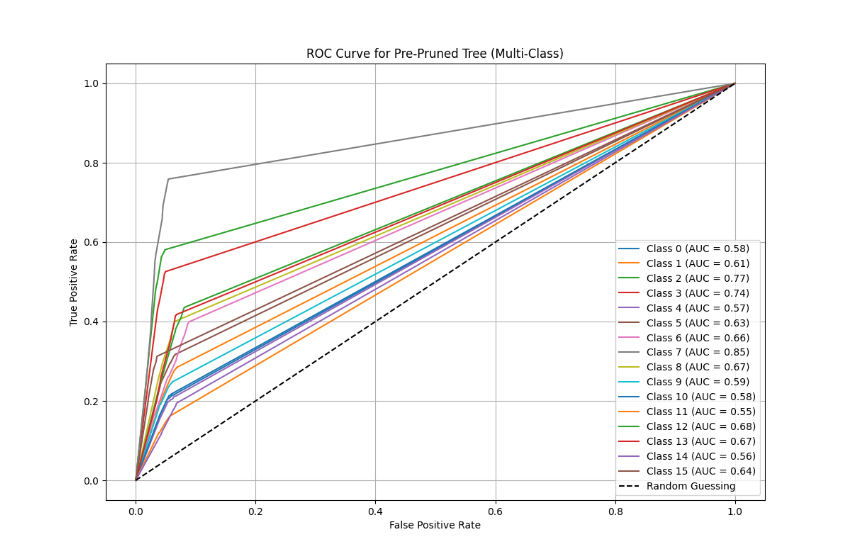


Figure 3.1.2.1: Post-pruned model ROC AUC Curve

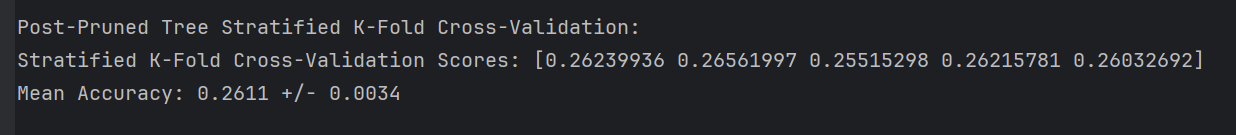
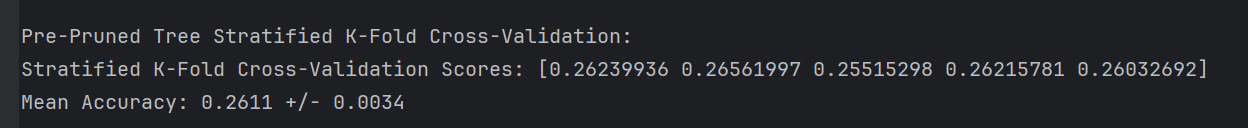


Figure 3.1.3: K-fold Cross Validation

For decision tree the pre and post pruned decision tree both have same values in the evaluation matrices because the optimum ccp\_alpha was 0. For ROC Curve every class has an AUC value greater than 0.50. As a result, the ROC Curve is above the 50% line. Class 7 has the highest AUC value and class 11 has the lowest AUC value.

The decision tree model overall has 29% accuracy.

* Logistic Regression
  + Confusion Matrix (Figure 3.2.1): Displayed the confusion matrix as shown in the figure (Figure 3.2.1) below. The diagonal values show the true positive values.
  + Classification Report (Figure 3.2.2): For each class (class 0-15) classification report shows the precision, recall, F-1 score and support behind it.
  + ROC Curve and AUC (Figure 3.2.3): To assess the model’s discriminatory abilities, ROC Curve and AUC score were produced.
  + Stratified K-Fold Cross Validation (Figure 3.2.4): Chose k= 5, showed mean accuracy and standard deviation.  
    A diagram of a confused matrix

    Description automatically generated

Figure 3.2.1: Logistic Regression Confusion Matrix

A screenshot of a computer screen

Description automatically generated

Figure 3.2.2: Logistic Regression Classification Report

A graph of different colored lines

Description automatically generated

Figure 3.2.3: Logistic Regression ROC Curve with AUC Values

A screenshot of a computer

Description automatically generated

Figure 3.2.4: Logistic Regression K-Fold Cross Validation

For logistic regression model, the accuracy of 16% was gained. The classification report shows precision, recall and F-1 score for each class. As a multi-label classification, the macro-AUC is 0.64 and in the ROC curve every class has an AUC over 0.50.

* KNN:
  + Used Elbow Method to find the best K which was 1, K was searched from 1 to 50 (Figure 3.3.0).
  + Confusion Matrix (Figure 3.3.1): Displayed the confusion matrix as shown in the figure (Figure 3.3.1) below. The diagonal values show the true positive values.
  + Classification Report (Figure 3.3.2): For each class (class 0-15) classification report shows the precision, recall, F-1 score and support behind it.
  + ROC Curve and AUC (Figure 3.3.3): To assess the model’s discriminatory abilities, ROC Curve and AUC score were produced.
  + Stratified K-Fold Cross Validation (Figure 3.3.4): Chose k= 5, showed mean accuracy and standard deviation.

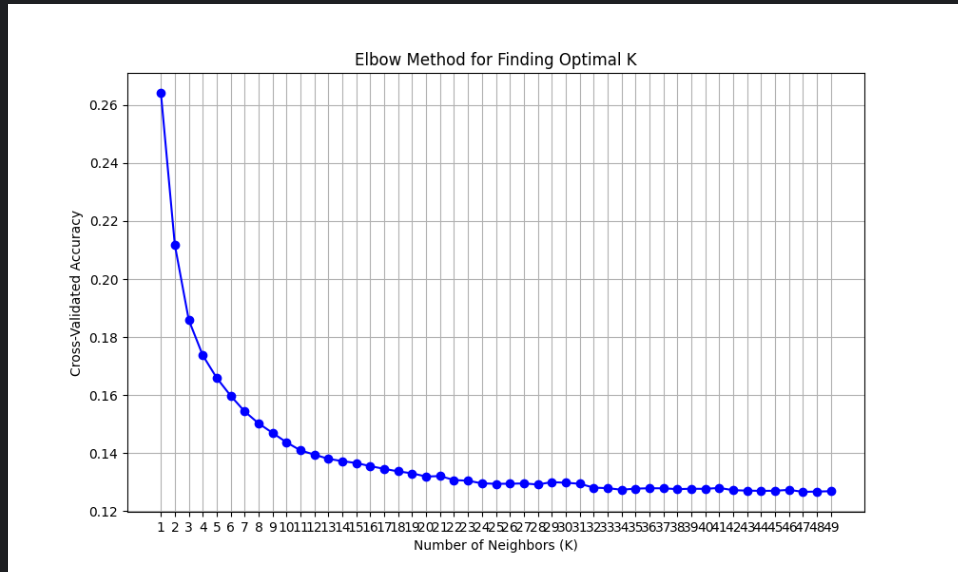
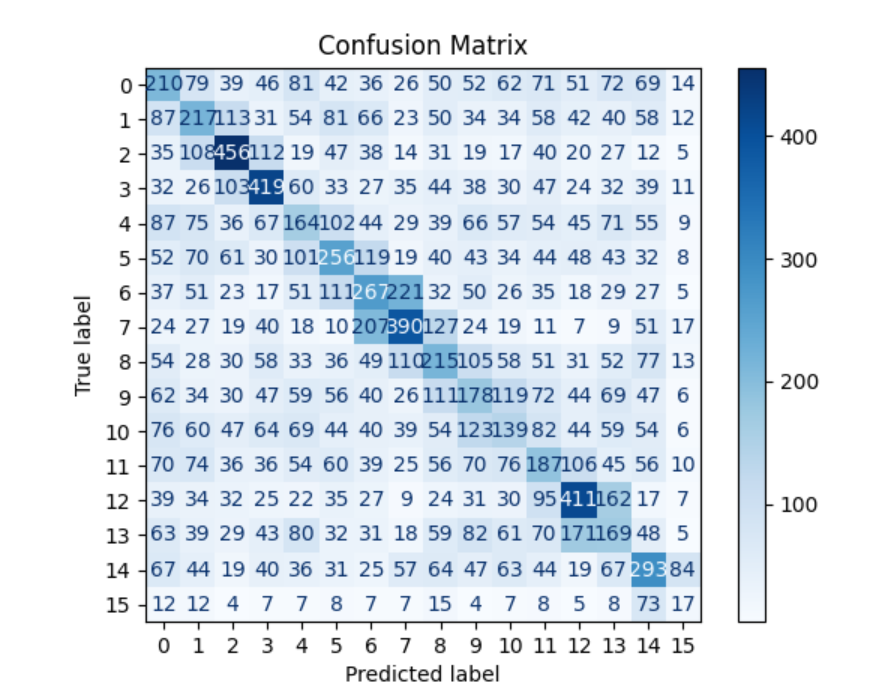
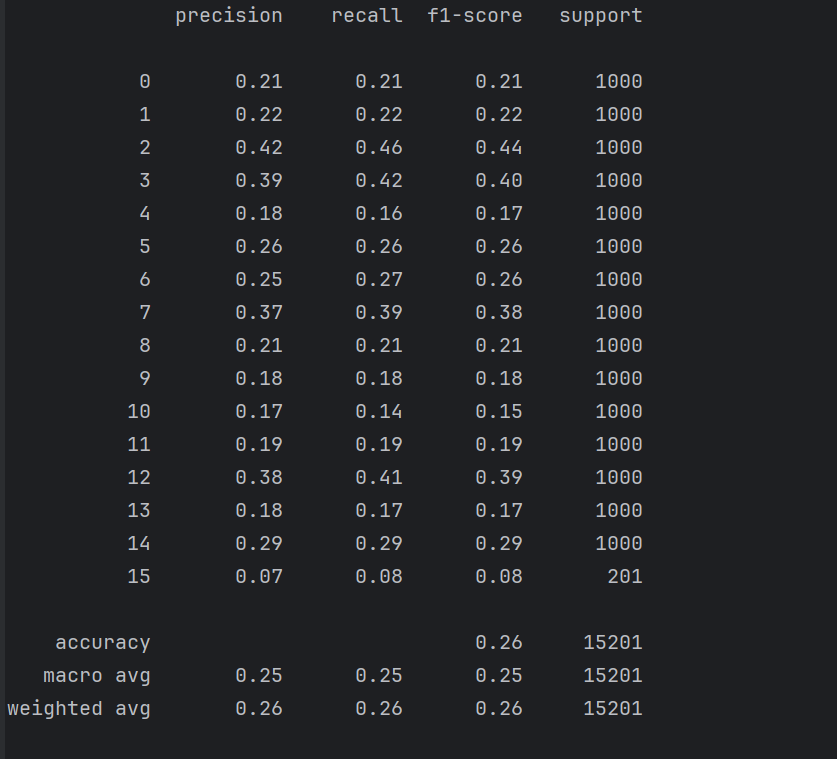


Figure 3.3.0: Finding optimum k

  
  
 Figure 3.3.1: KNN Confusion Matrix

 Figure 3.3.2: KNN Classification Report

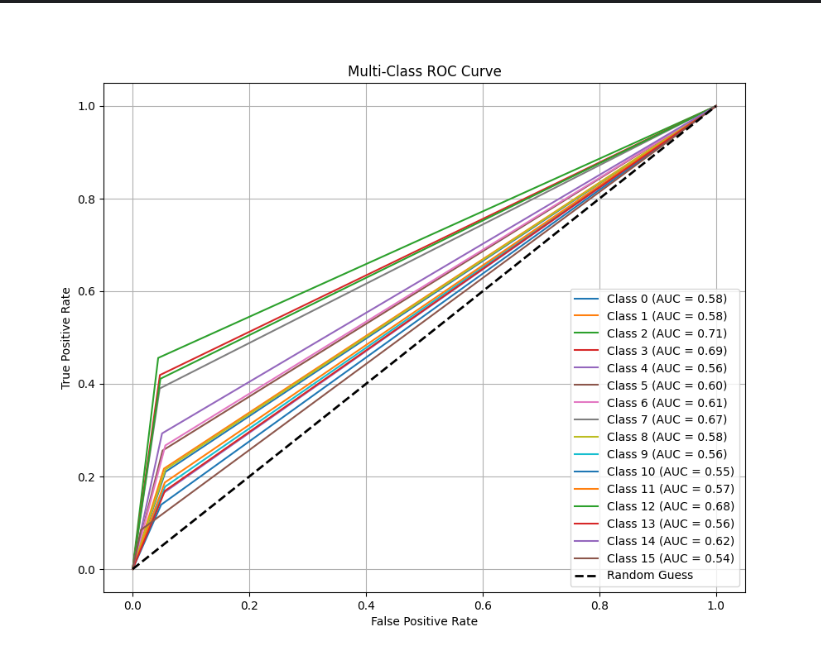


Figure 3.3.3: KNN ROC Curve

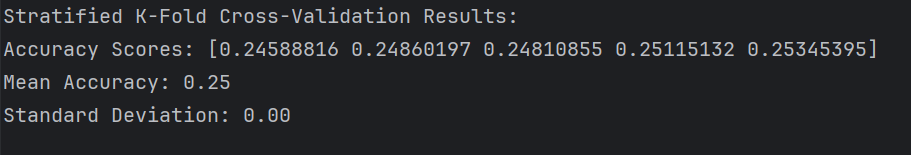


Figure 3.3.4: KNN K-Fold Cross Validation

For KNN, k is 1 after searching for k using elbow method. Using the optimum k, the accuracy of KNN is 26% and for each class the precision, recall and F-1 score is shown for every class. The mean accuracy after 5-fold is 25% with a standard deviation of 0.00. For each fold the accuracy score is also shown in the figure (3.3.4). The roc curve shows class-2 has the highest AUC value of 0.71 and class-15 has the lowest AUC value of 0.54.

* SVM
  + Linear Kernel: accuracy of 14%
    - Classification Report (Figure 3.4.1)
    - ROC Curve with AUC Score (Figure 3.4.2)
    - K-Fold Cross Validation (3.4.3)

A screenshot of a computer

Description automatically generated

Figure 3.4.1: SVM Linear Kernel Classification Report

A graph of multi-class roc curve

Description automatically generated

Figure 3.4.2: SVM Linear Kernel ROC Curve with AUC Score

A screenshot of a computer

Description automatically generated

Figure 3.4.3: SVM kernel K-Fold Cross Validation

* + Polynomial Kernel: accuracy of 14%
    - Classification Report (Figure 3.4.4)
    - ROC Curve with AUC Score (Figure 3.4.5)
    - K-Fold Cross Validation (3.4.6)

A screenshot of a computer screen

Description automatically generated  
 Figure 3.4.4: SVM Polynomial Kernel Classification Report

A graph of different colored lines

Description automatically generated

Figure 3.4.5: Polynomial Kernel ROC curve with AUC Score

A screenshot of a computer

Description automatically generated

Figure 3.4.6: Polynomial Kernel K-Fold Cross Validation

* + RBF Kernel:
    - Classification Report (Figure 3.4.7)
    - ROC Curve with AUC Score (Figure 3.4.8)
    - K-Fold Cross Validation (3.4.9)

A screenshot of a computer screen

Description automatically generated

Figure 3.4.7: RBF Classification Report

A graph of multi-class curve

Description automatically generated

Figure 3.4.8: RBF ROC Curve with AUC Score

A screenshot of a computer

Description automatically generated

Figure 3.4.9: RBF K-Fold Cross Validation

For SVM, linear kernel and polynomial kernel have the same accuracy of 14% whereas RBF kernel has 12% accuracy. In the ROC Curve for every kernel class 7 had the highest accuracy. Macro AUC for all the kernel is above 0.50 and as a result, the ROC curve is above 50% line.

5-fold cross validation shows and there are not that much improvement.

* Naïve Bayes
  + Classification Report (Figure 3.5.1): For each class (class 0-15) classification report shows the precision, recall, F-1 score and support behind it.
  + ROC Curve and AUC (Figure 3.5.2): To assess the model’s discriminatory abilities, ROC Curve and AUC score were produced.
  + Stratified K-Fold Cross Validation (Figure 3.5.3): Chose k= 5, showed mean accuracy and standard deviation.

A screenshot of a computer

Description automatically generated

A graph of different colored lines

Description automatically generated

1. K-Fold Accuracy Scores: [0.10209854614827972, 0.09769736842105263, 0.10026315789473685, 0.09907894736842106, 0.10236842105263158]
2. Mean Accuracy: 0.10,
3. K-Fold AUC Scores: [np.float64(0.5769371438766837), np.float64(0.5772191355633802), np.float64(0.5795489460093897), np.float64(0.5760854901701877), np.float64(0.5788754881161973)]
4. Mean AUC: 0.58
5. Standard Deviation: 0.00

So, overall, Naïve Bayes have the accuracy of 10% which basically proves the model performs very bad in this model.

Phase IV

In phase IV, clustering and association rule mining were implemented. As a result, there is no target variable, and model was used on the whole dataset.

Clustering:

For clustering,2 algorithms were used to uncover patterns in data:

* K-Means Clustering: For K-Means clustering, PCA was used to reduce the dimensionality of the data. After reducing the dimensionality there were only 2 principal components which explained more than 95% of the variance. silhouette analysis was done to find the optimum K. The optimum value of K was 2 (Figure 4.1.1, Figure 4.1.2). After that, the final figure (Figure 4.1.3) shows the final clusters for the model.  
  A graph with orange lines

  Description automatically generated

Figure 4.1.1: Silhouette Analysis

A graph with a line

Description automatically generated

Figure 4.1.2: Elbow Method to find K

A yellow and purple squares with red dots

Description automatically generated  
 Figure 4.1.2: Within Plot for K Means

* DBSCAN Algorithm: To use DBSCAN algorithm, first the number of eps needed to be determined. Initially, the min sample was chosen to be 2 and later it was updated to 5. As a result, K = 4 because K=min sample -1. According to the figure (Figure 4.2.1), by using K-distance graph the number of eps was found to be 800. Using these values, the final model had 3 clusters and 30 noises (Figure 4.2.2).

A graph with a line

Description automatically generated

Figure 4.2.1: K-distance graph to find eps

A blue graph with white text

Description automatically generated

Figure 4.2.2: Final Model DBSCAN

Association Rule Mining

The Apriori Algorithm was used to manifest meaningful rules between the features. First, SVD was used for dimensionality reduction technique as it works perfectly for this algorithm. After that, to binarize the value, values which are greater than 0 were replaced by 1 and values under 0 were replaced by 0 (Figure 4.3.1). After that, frequent item sets were found with 36 combinations (Figure 4.3.2) which were later used to get the association rules (Figure 4.3.3). According to the figure (Figure 4.3.3), the first few have high confidence and support. Lastly, the support vs confidence graph was plotted (Figure 4.3.4).  
A screen shot of a computer

Description automatically generated

Figure 4.3.1: Binarize the Data

A screenshot of a computer

Description automatically generated

Figure 4.3.2: frequent item set

A screenshot of a computer screen

Description automatically generated

Figure 4.3.3: Association Rules

A graph with different colored dots

Description automatically generated

Figure 4.3.4: Support vs Confidence graph

Recommendations

* Feature Engineering and Data pre-processing is an important part of any machine learning model and needs to be done more thoroughly.
* For future improvements, I would love to work with a proper machine which is computationally powerful for the models so that I can research more thoroughly.
* The classification models are computationally expensive and it was hard to run these in my machine which is why I could not research for further improvement for the predictions.

Appendix

Appendix A: Important Libraries

import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
import plotly.graph\_objects as go  
import plotly.express as px  
from statsmodels.tools import categorical  
from prettytable import PrettyTable  
import re  
import time  
from sklearn.decomposition import PCA  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
from sklearn.linear\_model import LinearRegression  
from sklearn.ensemble import RandomForestClassifier  
from scipy import stats

Appendix B: Python Code for Data Cleaning and Preprocessing

#%% Dataset  
df = pd.read\_csv('Machine Learning Term Project/combined\_genres.csv')  
print(df.head())  
#%% Duplicate Values  
print(f' Before Removing Duplicates {df.duplicated().sum()}')  
df = df.drop\_duplicates()  
print(f' After Removing Duplicates {df.duplicated().sum()}')  
#%% Null Values with dtypes  
null\_info = pd.DataFrame({  
 'Null Values': df.isnull().sum(),  
 'Null Percentage': (df.isnull().sum() / len(df)) \* 100,  
 'Data Type': df.dtypes  
})  
print(null\_info)  
#%% Handling Unwanted and NUll Values  
unwanted\_values = ['I', 'II', 'V', 'III', 'VII', 'IV', 'XXIII', 'IX', 'XV', 'VI', 'X', 'XIV', 'XIX', 'XXIX', 'XXI', 'VIII', 'XI', 'XVIII', 'XII', 'XIII', 'LXXI', 'XVI', 'XX', 'XXXIII', 'XXXII', 'XXXVI', 'XVII', 'LXIV', 'LXII', 'LXVIII', 'XL', 'XXXIV', 'XXXI', 'XLV', 'XLIV', 'XXIV', 'XXVII', 'LX', 'XXV', 'XXXIX', '2029', 'XXVIII', 'XXX', 'LXXII', '1909', 'XXXVIII', 'XXII', 'LVI', 'LVII' 'XLI', 'LII', 'XXXVII', 'LIX', 'LVIII', 'LXX', 'XLIII', 'XLIX', 'LXXIV', 'XXVI', 'C', 'XLI', 'LVII', 'LV','XLVI', 'LXXVII', 'XXXV', 'LIV', 'LI', 'LXXXII', 'XCIX', 'LXIII']  
  
df = df[~df['year'].astype(str).isin(unwanted\_values)]  
df['year'] = pd.to\_numeric(df['year'], errors='coerce')  
df = df.dropna(subset=['year'])  
df['year'] = df['year'].astype(int)  
#%% Clean runtime  
def clean\_runtime(runtime):  
 try:  
  
 return float(re.findall(r'\d+', str(runtime))[0])  
 except (IndexError, ValueError):  
 return None  
  
df['runtime'] = df['runtime'].apply(clean\_runtime)  
df = df.dropna(subset=['runtime'])  
df['runtime'] = df['runtime'].astype(float)  
#%% Clean director and stars  
def clean\_string(value):  
 if pd.isnull(value):  
 return value  
 return str(value).strip()  
df['director'] = df['director'].str.replace('\n', '')  
df['star'] = df['star'].str.replace('\n', '')  
df['director'] = df['director'].apply(clean\_string)  
df['star'] = df['star'].apply(clean\_string)  
  
print(df[['director', 'star']].head())  
#%%  
df =df.dropna(subset=['director','star'])  
#%%  
df = df.dropna(subset=['votes', 'rating'])  
print(f"New dataset shape: {df.shape}")  
  
  
#%% Dropping columns with no importance  
df = df.drop(columns=['movie\_id', 'director\_id', 'star\_id', 'description'])  
print(df.head())  
#%% Dropping columns with more than 50% null values  
df =df.drop(columns=['certificate','gross(in $)'])  
  
#%% Aggregation  
  
#%% Before Upsampling and Downsampling  
genre\_counts = df['genre'].value\_counts()  
  
plt.figure(figsize=(16, 10))  
genre\_counts.plot(kind='bar')  
plt.title('Genre Distribution', fontsize=16)  
plt.xlabel('Genre', fontsize=14)  
plt.ylabel('Number of Movies', fontsize=14)  
plt.xticks(rotation=45, fontsize=12)  
plt.grid(axis='y', linestyle='--', alpha=0.7)  
plt.show()  
#%% Upsample and Downsample  
film\_noir\_data = df[df['genre'] == 'Film-Noir']  
film\_noir\_upsampled = film\_noir\_data.sample(n=5000, random\_state=42, replace=True)  
  
# Downsample the top 15 genres to 5000 samples  
top\_15\_genres = df['genre'].value\_counts().index[:15]  
downsampled\_data = pd.concat([  
 x.sample(n=5000, random\_state=42) if len(x) >= 5000 else x  
 for \_, x in df[df['genre'].isin(top\_15\_genres)].groupby('genre')  
]).reset\_index(drop=True)  
balanced\_data = pd.concat([downsampled\_data, film\_noir\_upsampled], ignore\_index=True)  
print(f"Balanced dataset shape: {balanced\_data.shape}")  
print(balanced\_data['genre'].value\_counts())  
#%%  
balanced\_data = pd.concat([  
 x.sample(n=5000, random\_state=42, replace=True)  
 for \_, x in df.groupby('genre')  
]).reset\_index(drop=True)  
  
# Display the shape of the new balanced dataset  
print(f"Balanced dataset shape: {balanced\_data.shape}")  
print(balanced\_data['genre'].value\_counts())  
#%%  
genre\_counts = balanced\_data['genre'].value\_counts()  
  
plt.figure(figsize=(16, 10))  
genre\_counts.plot(kind='bar')  
plt.title('Genre Distribution', fontsize=16)  
plt.xlabel('Genre', fontsize=14)  
plt.ylabel('Number of Movies', fontsize=14)  
plt.xticks(rotation=45, fontsize=12)  
plt.grid(axis='y', linestyle='--', alpha=0.7)  
plt.show()  
#%%  
balanced\_data = balanced\_data.drop(columns =['movie\_name'])  
#%%  
df\_cleaned = balanced\_data.copy()  
  
#%% Encoding  
from sklearn.preprocessing import LabelEncoder  
  
y = df\_cleaned['genre']  
X = df\_cleaned.drop(columns=['genre'])  
  
label\_encoder = LabelEncoder()  
for col in X.select\_dtypes(include=['object']).columns:  
 X[col] = label\_encoder.fit\_transform(X[col].astype(str))  
y = label\_encoder.fit\_transform(y)  
  
#%%  
from sklearn.preprocessing import StandardScaler, MinMaxScaler  
  
scaler = StandardScaler()  
X\_standardized = scaler.fit\_transform(X)  
X\_standardized\_df = pd.DataFrame(X\_standardized, columns=X.columns)  
print("Standardized Features:")  
print(X\_standardized\_df.head())  
#%%  
cov\_matrix = np.cov(X\_standardized\_df, rowvar=False)  
plt.figure(figsize=(10, 8))  
sns.heatmap(cov\_matrix, annot=True, cmap='coolwarm', cbar=True)  
plt.title('Covariance Matrix (Standardized Data)', fontsize=16)  
plt.show()  
#%%  
corr\_matrix = X\_standardized\_df.corr()  
plt.figure(figsize=(10, 8))  
sns.heatmap(corr\_matrix, annot=True, fmt=".2f", cmap='coolwarm', cbar=True)  
plt.title('Correlation Matrix (Standardized Data)', fontsize=16)  
plt.show()  
#%%  
normalizer = MinMaxScaler()  
X\_normalized = normalizer.fit\_transform(X)  
X\_normalized\_df = pd.DataFrame(X\_normalized, columns=X.columns)  
print("\nNormalized Features:")  
print(X\_normalized\_df.head())  
  
  
  
#%%  
  
from statsmodels.stats.outliers\_influence import variance\_inflation\_factor  
from sklearn.decomposition import PCA, TruncatedSVD  
from sklearn.ensemble import RandomForestRegressor  
def calculate\_vif(X):  
 vif\_data = pd.DataFrame()  
 vif\_data['Feature'] = X.columns  
 vif\_data['VIF'] = [variance\_inflation\_factor(X.values, i) for i in range(X.shape[1])]  
 return vif\_data  
  
vif\_standardized = calculate\_vif(X\_standardized\_df)  
print("VIF Before Removal (Standardized):")  
print(vif\_standardized)  
  
high\_vif\_features = vif\_standardized[vif\_standardized['VIF'] > 10]['Feature']  
X\_standardized\_reduced = X\_standardized\_df.drop(columns=high\_vif\_features)  
  
# Recalculate VIF after removal  
vif\_standardized\_reduced = calculate\_vif(X\_standardized\_reduced)  
print("VIF After Removal (Standardized):")  
print(vif\_standardized\_reduced)  
  
pca = PCA(n\_components=0.95)  
X\_pca = pca.fit\_transform(X\_standardized\_reduced)  
explained\_variance = pca.explained\_variance\_ratio\_  
print("PCA Explained Variance Ratio:", explained\_variance)  
print("PCA Shape After Reduction:", X\_pca.shape)  
rf\_regressor = RandomForestRegressor(random\_state=42)  
rf\_regressor.fit(X\_standardized\_reduced, y)  
feature\_importances = pd.DataFrame({  
 'Feature': X\_standardized\_reduced.columns,  
 'Importance': rf\_regressor.feature\_importances\_  
}).sort\_values(by='Importance', ascending=False)  
  
print("Random Forest Feature Importance:")  
print(feature\_importances)  
  
top\_features = feature\_importances.head(10)['Feature']  
X\_rf\_selected = X\_standardized\_reduced[top\_features]  
  
svd = TruncatedSVD(n\_components=6, random\_state=42)  
X\_svd = svd.fit\_transform(X\_standardized\_reduced)  
print("SVD Shape After Reduction:", X\_svd.shape)  
  
print("\nFinal Selected Features (VIF and RF Importance):", X\_rf\_selected.columns.tolist())  
print("\nFinal PCA Components Shape:", X\_pca.shape)  
print("\nFinal SVD Components Shape:", X\_svd.shape)  
  
#%%  
vif\_normalized = calculate\_vif(X\_normalized\_df)  
print("\nVIF Before Removal (Normalized):")  
print(vif\_normalized)  
  
high\_vif\_features = vif\_normalized[vif\_normalized['VIF'] > 10]['Feature']  
X\_normalized\_reduced = X\_normalized\_df.drop(columns=high\_vif\_features)  
vif\_normalized\_reduced = calculate\_vif(X\_normalized\_reduced)  
print("\nVIF After Removal (Normalized):")  
print(vif\_normalized\_reduced)  
  
pca = PCA(n\_components=0.95)  
X\_pca\_normalized = pca.fit\_transform(X\_normalized\_reduced)  
explained\_variance\_normalized = pca.explained\_variance\_ratio\_  
print("\nPCA Explained Variance Ratio (Normalized):", explained\_variance\_normalized)  
print("PCA Shape After Reduction (Normalized):", X\_pca\_normalized.shape)  
  
rf\_regressor = RandomForestRegressor(random\_state=42)  
rf\_regressor.fit(X\_normalized\_reduced, y)  
feature\_importances\_normalized = pd.DataFrame({  
 'Feature': X\_normalized\_reduced.columns,  
 'Importance': rf\_regressor.feature\_importances\_  
}).sort\_values(by='Importance', ascending=False)  
  
print("\nRandom Forest Feature Importance (Normalized):")  
print(feature\_importances\_normalized)  
  
top\_features\_normalized = feature\_importances\_normalized.head(10)['Feature']  
X\_rf\_selected\_normalized = X\_normalized\_reduced[top\_features\_normalized]  
  
svd = TruncatedSVD(n\_components=4, random\_state=42)  
X\_svd\_normalized = svd.fit\_transform(X\_normalized\_reduced)  
print("\nSVD Shape After Reduction (Normalized):", X\_svd\_normalized.shape)  
print("\nFinal Selected Features (VIF and RF Importance, Normalized):", X\_rf\_selected\_normalized.columns.tolist())  
print("\nFinal PCA Components Shape (Normalized):", X\_pca\_normalized.shape)  
print("\nFinal SVD Components Shape (Normalized):", X\_svd\_normalized.shape)

#%%  
from sklearn.neighbors import LocalOutlierFactor  
from sklearn.ensemble import IsolationForest  
from sklearn.cluster import DBSCAN  
from scipy.stats import zscore  
  
  
def detect\_and\_remove\_outliers(method, data, \*\*kwargs):  
  
 if method == 'lof':  
 n\_neighbors = kwargs.get('n\_neighbors', 20)  
 contamination = kwargs.get('contamination', 0.05)  
 lof = LocalOutlierFactor(n\_neighbors=n\_neighbors, contamination=contamination)  
 outlier\_labels = lof.fit\_predict(data)  
 return data[outlier\_labels == 1]  
  
 elif method == 'isolation\_forest':  
 contamination = kwargs.get('contamination', 0.05)  
 random\_state = kwargs.get('random\_state', 42)  
 iso\_forest = IsolationForest(contamination=contamination, random\_state=random\_state)  
 outlier\_labels = iso\_forest.fit\_predict(data)  
 return data[outlier\_labels == 1]  
  
 elif method == 'dbscan':  
 eps = kwargs.get('eps', 0.5)  
 min\_samples = kwargs.get('min\_samples', 5)  
 dbscan = DBSCAN(eps=eps, min\_samples=min\_samples)  
 cluster\_labels = dbscan.fit\_predict(data)  
 return data[cluster\_labels != -1] # Keep only non-outlier points  
  
 elif method == 'zscore':  
 threshold = kwargs.get('threshold', 3)  
 z\_scores = np.abs(zscore(data))  
 outlier\_mask = (z\_scores < threshold).all(axis=1) # Identify rows where all features are below threshold  
 return data[outlier\_mask]  
  
 else:  
 raise ValueError("Invalid method. Choose 'lof', 'isolation\_forest', 'dbscan', or 'zscore'.")  
  
  
  
X\_lof\_cleaned = detect\_and\_remove\_outliers('lof', X\_rf\_selected, n\_neighbors=20, contamination=0.05)  
print(f"Shape after LOF outlier removal: {X\_lof\_cleaned.shape}")  
  
X\_iso\_cleaned = detect\_and\_remove\_outliers('isolation\_forest', X\_rf\_selected, contamination=0.05, random\_state=42)  
print(f"Shape after Isolation Forest outlier removal: {X\_iso\_cleaned.shape}")  
  
X\_dbscan\_cleaned = detect\_and\_remove\_outliers('dbscan', X\_rf\_selected, eps=0.5, min\_samples=5)  
print(f"Shape after DBSCAN outlier removal: {X\_dbscan\_cleaned.shape}")  
  
X\_zscore\_cleaned = detect\_and\_remove\_outliers('zscore', X\_rf\_selected, threshold=3)  
print(f"Shape after Z-score outlier removal: {X\_zscore\_cleaned.shape}")  
  
#%%  
  
X\_lof\_cleaned\_normalized = detect\_and\_remove\_outliers('lof', X\_rf\_selected\_normalized, n\_neighbors=20, contamination=0.05)  
print(f"Shape after LOF outlier removal (Normalized): {X\_lof\_cleaned\_normalized.shape}")  
  
X\_iso\_cleaned\_normalized = detect\_and\_remove\_outliers('isolation\_forest', X\_rf\_selected\_normalized, contamination=0.05, random\_state=42)  
print(f"Shape after Isolation Forest outlier removal (Normalized): {X\_iso\_cleaned\_normalized.shape}")  
  
X\_dbscan\_cleaned\_normalized = detect\_and\_remove\_outliers('dbscan', X\_rf\_selected\_normalized, eps=0.5, min\_samples=5)  
print(f"Shape after DBSCAN outlier removal (Normalized): {X\_dbscan\_cleaned\_normalized.shape}")  
  
X\_zscore\_cleaned\_normalized = detect\_and\_remove\_outliers('zscore', X\_rf\_selected\_normalized, threshold=3)  
print(f"Shape after Z-score outlier removal (Normalized): {X\_zscore\_cleaned\_normalized.shape}")

Appendix C: Regression Analysis

#%% Linear Regression Model  
from sklearn.linear\_model import LinearRegression  
from sklearn.metrics import mean\_squared\_error, r2\_score  
import statsmodels.api as sm  
  
X\_train\_sm = sm.add\_constant(X\_train)  
X\_test\_sm = sm.add\_constant(X\_test)  
  
model = sm.OLS(y\_train, X\_train\_sm).fit()  
  
y\_pred\_train = model.predict(X\_train\_sm)  
y\_pred\_test = model.predict(X\_test\_sm)  
  
mse = mean\_squared\_error(y\_test, y\_pred\_test)  
r\_squared = r2\_score(y\_test, y\_pred\_test)  
adjusted\_r\_squared = 1 - (1 - r\_squared) \* (len(y\_test) - 1) / (len(y\_test) - X\_test.shape[1] - 1)  
aic = model.aic  
bic = model.bic  
  
evaluation\_table = pd.DataFrame({  
 "Metric": ["R-squared", "Adjusted R-squared", "AIC", "BIC", "MSE"],  
 "Value": [r\_squared, adjusted\_r\_squared, aic, bic, mse]  
})  
print(evaluation\_table)  
  
#%% T-test and F-test  
print("T-Test Results:")  
print(model.t\_test(np.identity(len(model.params))))  
  
print("\nF-Test Results:")  
print(model.f\_test(np.identity(len(model.params))))  
  
#%% Confidence Interval Analysis  
confidence\_intervals = model.conf\_int()  
print("\nConfidence Intervals:")  
print(confidence\_intervals)  
  
#%% Stepwise Regression  
X\_train\_reset = pd.DataFrame(X\_train).reset\_index(drop=True)  
y\_train\_reset = pd.Series(y\_train).reset\_index(drop=True)  
  
def stepwise\_selection(X, y, initial\_list=[], threshold\_in=0.01, threshold\_out=0.05, verbose=True):  
 included = list(initial\_list)  
 while True:  
 changed = False  
 excluded = list(set(X.columns) - set(included))  
 new\_pval = pd.Series(index=excluded, dtype=float)  
 for new\_col in excluded:  
 model = sm.OLS(y, sm.add\_constant(X[included + [new\_col]])).fit()  
 new\_pval[new\_col] = model.pvalues[new\_col]  
 best\_pval = new\_pval.min()  
 if best\_pval < threshold\_in:  
 best\_feature = new\_pval.idxmin()  
 included.append(best\_feature)  
 changed = True  
 if verbose:  
 print(f"Add {best\_feature} with p-value {best\_pval}")  
  
 model = sm.OLS(y, sm.add\_constant(X[included])).fit()  
 pvalues = model.pvalues.iloc[1:]  
 worst\_pval = pvalues.max()  
 if worst\_pval > threshold\_out:  
 changed = True  
 worst\_feature = pvalues.idxmax()  
 included.remove(worst\_feature)  
 if verbose:  
 print(f"Drop {worst\_feature} with p-value {worst\_pval}")  
  
 if not changed:  
 break  
 return included  
selected\_features = stepwise\_selection(X\_train\_reset, y\_train\_reset)  
  
print("\nSelected Features via Stepwise Regression:")  
print(selected\_features)

Appendix D: Classification Analysis

#%% Baseline DT model  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.metrics import accuracy\_score  
  
baseline\_tree = DecisionTreeClassifier(random\_state=42)  
baseline\_tree.fit(X\_train, y\_train)  
  
y\_pred = baseline\_tree.predict(X\_test)  
print(f"Baseline Accuracy: {accuracy\_score(y\_test, y\_pred)}")  
  
#%% Pre-Pruning  
from sklearn.model\_selection import GridSearchCV  
  
param\_grid = {  
 'criterion': ['gini', 'entropy'],  
 'splitter': ['best', 'random'],  
 'max\_depth': [5, 10, 15, None],  
 'min\_samples\_split': [2, 5, 10],  
 'min\_samples\_leaf': [1, 2, 4],  
 'max\_features': [None, 'sqrt', 'log2'],  
}  
  
grid\_search = GridSearchCV(  
 estimator=DecisionTreeClassifier(random\_state=42),  
 param\_grid=param\_grid,  
 scoring='accuracy',  
 cv=5,  
 n\_jobs=-1  
)  
  
grid\_search.fit(X\_train, y\_train)  
  
best\_params = grid\_search.best\_params\_  
best\_tree = grid\_search.best\_estimator\_  
print(f"Best Parameters: {best\_params}")  
#%%  
from sklearn.metrics import classification\_report, accuracy\_score  
pre\_pruned\_tree = DecisionTreeClassifier(  
 random\_state=42,  
 criterion=best\_params['criterion'],  
 splitter=best\_params['splitter'],  
 max\_depth=best\_params['max\_depth'],  
 min\_samples\_split=best\_params['min\_samples\_split'],  
 min\_samples\_leaf=best\_params['min\_samples\_leaf'],  
 max\_features=best\_params['max\_features']  
)  
  
pre\_pruned\_tree.fit(X\_train, y\_train)  
  
y\_pred = pre\_pruned\_tree.predict(X\_test)  
  
print(f"Accuracy: {accuracy\_score(y\_test, y\_pred)}")  
print("\nClassification Report:")  
print(classification\_report(y\_test, y\_pred))  
#%%  
from sklearn.tree import DecisionTreeClassifier  
X\_t = X\_train[:1000]  
y\_t = y\_train[:1000]  
path = DecisionTreeClassifier(random\_state=42).cost\_complexity\_pruning\_path(X\_t,y\_t)  
ccp\_alphas = path.ccp\_alphas  
impurities = path.impurities  
ccp\_alphas = ccp\_alphas[ccp\_alphas >= 0]  
#%%  
trees = []  
for alpha in ccp\_alphas:  
 tree = DecisionTreeClassifier(random\_state=42, ccp\_alpha=alpha)  
 tree.fit(X\_train, y\_train)  
 trees.append(tree)  
  
  
#%%  
from sklearn.metrics import accuracy\_score  
  
train\_scores = [accuracy\_score(y\_train, tree.predict(X\_train)) for tree in trees]  
test\_scores = [accuracy\_score(y\_test, tree.predict(X\_test)) for tree in trees]  
  
import matplotlib.pyplot as plt  
  
plt.figure(figsize=(10, 6))  
plt.plot(ccp\_alphas, train\_scores, label="Train Accuracy", marker='o')  
plt.plot(ccp\_alphas, test\_scores, label="Test Accuracy", marker='o')  
plt.xlabel("ccp\_alpha")  
plt.ylabel("Accuracy")  
plt.title("Effect of ccp\_alpha on Model Performance")  
plt.legend()  
plt.grid(True)  
plt.show()  
  
#%%  
optimal\_ccp\_alpha = ccp\_alphas[np.argmax(test\_scores)]  
print(f"Optimal ccp\_alpha: {optimal\_ccp\_alpha}")  
  
#%%  
post\_pruned\_tree = DecisionTreeClassifier(random\_state=42, ccp\_alpha=optimal\_ccp\_alpha)  
post\_pruned\_tree.fit(X\_train, y\_train)  
#%%  
from sklearn.metrics import confusion\_matrix, precision\_score, recall\_score, f1\_score, roc\_curve, auc, roc\_auc\_score  
from sklearn.model\_selection import StratifiedKFold, cross\_val\_score  
import matplotlib.pyplot as plt  
import numpy as np  
  
#%%  
def calculate\_specificity\_multiclass(y\_true, y\_pred, average='macro'):  
 cm = confusion\_matrix(y\_true, y\_pred)  
 specificity\_per\_class = []  
 for i in range(len(cm)):  
 tn = np.sum(cm) - (np.sum(cm[i, :]) + np.sum(cm[:, i]) - cm[i, i])  
 fp = np.sum(cm[:, i]) - cm[i, i]  
 specificity\_per\_class.append(tn / (tn + fp) if (tn + fp) > 0 else 0)  
 if average == 'macro':  
 return np.mean(specificity\_per\_class)  
 elif average == 'weighted':  
 weights = np.sum(cm, axis=1) / np.sum(cm)  
 return np.sum(specificity\_per\_class \* weights)  
  
#%%  
def evaluate\_model(model, X\_test, y\_test):  
 y\_pred = model.predict(X\_test)  
 y\_prob = model.predict\_proba(X\_test)  
  
  
 cm = confusion\_matrix(y\_test, y\_pred)  
 print("Confusion Matrix:")  
 print(cm)  
  
  
 precision = precision\_score(y\_test, y\_pred, average='macro')  
 recall = recall\_score(y\_test, y\_pred, average='macro')  
 f1 = f1\_score(y\_test, y\_pred, average='macro')  
 specificity = calculate\_specificity\_multiclass(y\_test, y\_pred, average='macro')  
  
 print(f"Precision (Macro): {precision:.4f}")  
 print(f"Recall (Macro): {recall:.4f}")  
 print(f"Specificity (Macro): {specificity:.4f}")  
 print(f"F1-Score (Macro): {f1:.4f}")  
  
  
 roc\_auc = roc\_auc\_score(y\_test, y\_prob, multi\_class='ovr', average='macro')  
  
 print(f"AUC (Macro): {roc\_auc:.4f}")  
#%%  
evaluate\_model(pre\_pruned\_tree, X\_test, y\_test)  
#%%  
evaluate\_model(post\_pruned\_tree, X\_test, y\_test)  
  
#%%  
from sklearn.preprocessing import label\_binarize  
y\_prob\_pre\_pruned = pre\_pruned\_tree.predict\_proba(X\_test)  
y\_prob\_post\_pruned = post\_pruned\_tree.predict\_proba(X\_test)  
  
classes = np.unique(y\_test)  
y\_test\_binarized = label\_binarize(y\_test, classes=classes)  
  
fpr\_pre, tpr\_pre, roc\_auc\_pre = {}, {}, {}  
for i in range(len(classes)):  
 fpr\_pre[i], tpr\_pre[i], \_ = roc\_curve(y\_test\_binarized[:, i], y\_prob\_pre\_pruned[:, i])  
 roc\_auc\_pre[i] = auc(fpr\_pre[i], tpr\_pre[i])  
  
fpr\_post, tpr\_post, roc\_auc\_post = {}, {}, {}  
for i in range(len(classes)):  
 fpr\_post[i], tpr\_post[i], \_ = roc\_curve(y\_test\_binarized[:, i], y\_prob\_post\_pruned[:, i])  
 roc\_auc\_post[i] = auc(fpr\_post[i], tpr\_post[i])  
  
# Plot ROC-AUC Curve for Pre-Pruned Tree  
plt.figure(figsize=(12, 8))  
for i in range(len(classes)):  
 plt.plot(fpr\_pre[i], tpr\_pre[i], label=f'Class {classes[i]} (AUC = {roc\_auc\_pre[i]:.2f})')  
plt.plot([0, 1], [0, 1], 'k--', label='Random Guessing')  
plt.xlabel('False Positive Rate')  
plt.ylabel('True Positive Rate')  
plt.title('ROC Curve for Pre-Pruned Tree (Multi-Class)')  
plt.legend(loc='best')  
plt.grid()  
plt.show()  
  
# Plot ROC-AUC Curve for Post-Pruned Tree  
plt.figure(figsize=(12, 8))  
for i in range(len(classes)):  
 plt.plot(fpr\_post[i], tpr\_post[i], label=f'Class {classes[i]} (AUC = {roc\_auc\_post[i]:.2f})')  
plt.plot([0, 1], [0, 1], 'k--', label='Random Guessing')  
plt.xlabel('False Positive Rate')  
plt.ylabel('True Positive Rate')  
plt.title('ROC Curve for Post-Pruned Tree (Multi-Class)')  
plt.legend(loc='best')  
plt.grid()  
plt.show()  
#%%  
def perform\_stratified\_kfold(model, X, y):  
 skf = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)  
 scores = cross\_val\_score(model, X, y, cv=skf, scoring='accuracy')  
 print(f"Stratified K-Fold Cross-Validation Scores: {scores}")  
 print(f"Mean Accuracy: {scores.mean():.4f} +/- {scores.std():.4f}")  
  
print("\nPre-Pruned Tree Stratified K-Fold Cross-Validation:")  
perform\_stratified\_kfold(pre\_pruned\_tree, X\_train, y\_train)  
  
print("\nPost-Pruned Tree Stratified K-Fold Cross-Validation:")  
perform\_stratified\_kfold(post\_pruned\_tree, X\_train, y\_train)  
#%% Logistic Regression  
y\_zscore\_cleaned = y[X\_rf\_selected.index]  
y\_zscore\_cleaned = y\_zscore\_cleaned[:len(X\_zscore\_cleaned)]  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(  
 X\_zscore\_cleaned,  
 y\_zscore\_cleaned,  
 test\_size=0.2,  
 random\_state=42,  
 stratify=y\_zscore\_cleaned  
)  
  
print(f"X\_train shape: {X\_train.shape}, y\_train shape: {y\_train.shape}")  
print(f"X\_test shape: {X\_test.shape}, y\_test shape: {y\_test.shape}")  
  
#%%  
from sklearn.linear\_model import LogisticRegression  
from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay, classification\_report, roc\_curve, auc  
from sklearn.model\_selection import StratifiedKFold, cross\_val\_score  
import matplotlib.pyplot as plt  
import numpy as np  
  
log\_reg = LogisticRegression(random\_state=42, max\_iter=1000)  
log\_reg.fit(X\_train, y\_train)  
y\_pred = log\_reg.predict(X\_test)  
y\_pred\_prob = log\_reg.predict\_proba(X\_test)[:, 1]  
  
#%%  
print(y\_pred)  
  
#%% Confusion Matrix  
cm = confusion\_matrix(y\_test, y\_pred)  
  
if cm.shape == (2, 2):  
 TN, FP, FN, TP = cm.ravel()  
 precision = TP / (TP + FP) if (TP + FP) > 0 else 0  
 recall = TP / (TP + FN) if (TP + FN) > 0 else 0  
 specificity = TN / (TN + FP) if (TN + FP) > 0 else 0  
 f1\_score = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) > 0 else 0  
  
  
 print("Metrics:")  
 print(f"Precision: {precision:.2f}")  
 print(f"Recall (Sensitivity): {recall:.2f}")  
 print(f"Specificity: {specificity:.2f}")  
 print(f"F1-Score: {f1\_score:.2f}")  
  
else:  
 print("Classification Report:")  
 print(classification\_report(y\_test, y\_pred))  
  
#%%  
cm = confusion\_matrix(y\_test, y\_pred)  
disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=np.unique(y\_test))  
disp.plot(cmap=plt.cm.Blues)  
plt.title("Confusion Matrix")  
plt.show()  
  
#%%  
from sklearn.metrics import roc\_curve, auc, roc\_auc\_score  
from sklearn.preprocessing import label\_binarize  
classes = np.unique(y\_test) # Extract the unique classes  
y\_test\_binarized = label\_binarize(y\_test, classes=classes)  
y\_pred\_prob = log\_reg.predict\_proba(X\_test)  
classes = np.unique(y\_test) # Extract the unique classes  
y\_test\_binarized = label\_binarize(y\_test, classes=classes)  
y\_pred\_prob = log\_reg.predict\_proba(X\_test)  
fpr = {}  
tpr = {}  
roc\_auc = {}  
  
for i, class\_label in enumerate(classes):  
 fpr[class\_label], tpr[class\_label], \_ = roc\_curve(y\_test\_binarized[:, i], y\_pred\_prob[:, i])  
 roc\_auc[class\_label] = auc(fpr[class\_label], tpr[class\_label])  
  
fpr\_macro, tpr\_macro, \_ = roc\_curve(y\_test\_binarized.ravel(), y\_pred\_prob.ravel())  
roc\_auc\_macro = auc(fpr\_macro, tpr\_macro)  
plt.figure(figsize=(10, 8))  
for class\_label in classes:  
 plt.plot(fpr[class\_label], tpr[class\_label],  
 label=f"Class {class\_label} (AUC = {roc\_auc[class\_label]:.2f})")  
  
plt.plot(fpr\_macro, tpr\_macro, label=f"Macro-Average (AUC = {roc\_auc\_macro:.2f})", linestyle='--', color='black')  
  
plt.plot([0, 1], [0, 1], 'k--', label="Random Guess")  
  
plt.xlabel("False Positive Rate")  
plt.ylabel("True Positive Rate")  
plt.title("Multi-Class ROC-AUC Curve")  
plt.legend(loc="lower right")  
plt.grid(True)  
plt.show()  
roc\_auc\_macro = roc\_auc\_score(y\_test\_binarized, y\_pred\_prob, average="macro")  
roc\_auc\_weighted = roc\_auc\_score(y\_test\_binarized, y\_pred\_prob, average="weighted")  
  
print(f"Macro-Average AUC: {roc\_auc\_macro:.2f}")  
print(f"Weighted-Average AUC: {roc\_auc\_weighted:.2f}")  
  
#%%  
skf = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)  
#%%  
from sklearn.metrics import make\_scorer  
log\_reg = LogisticRegression(max\_iter=1000, random\_state=42, multi\_class='ovr')  
cv\_accuracy = cross\_val\_score(log\_reg, X\_train, y\_train, cv=skf, scoring='accuracy')  
  
cv\_f1\_macro = cross\_val\_score(log\_reg, X\_train, y\_train, cv=skf,  
 scoring=make\_scorer(f1\_score, average='macro'))  
  
  
cv\_roc\_auc\_weighted = cross\_val\_score(log\_reg, X\_train, y\_train, cv=skf,  
 scoring='roc\_auc\_ovr\_weighted')  
  
print("Stratified K-Fold Cross-Validation Results:")  
print(f"Accuracy Scores: {cv\_accuracy}")  
print(f"Mean Accuracy: {cv\_accuracy.mean():.2f}")  
print(f"Standard Deviation: {cv\_accuracy.std():.2f}\n")  
  
print(f"F1 Macro Scores: {cv\_f1\_macro}")  
print(f"Mean F1 Macro: {cv\_f1\_macro.mean():.2f}")  
print(f"Standard Deviation: {cv\_f1\_macro.std():.2f}\n")  
  
print(f"ROC-AUC Weighted Scores: {cv\_roc\_auc\_weighted}")  
print(f"Mean ROC-AUC Weighted: {cv\_roc\_auc\_weighted.mean():.2f}")  
print(f"Standard Deviation: {cv\_roc\_auc\_weighted.std():.2f}\n")  
  
#%% KNN  
y\_lof\_cleaned\_normalized = y[:len(X\_lof\_cleaned\_normalized)]  
from sklearn.model\_selection import train\_test\_split  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(  
 X\_lof\_cleaned\_normalized,  
 y\_lof\_cleaned\_normalized,  
 test\_size=0.2,  
 random\_state=42,  
 stratify=y\_lof\_cleaned\_normalized  
)  
  
print(f"X\_train shape: {X\_train.shape}, y\_train shape: {y\_train.shape}")  
print(f"X\_test shape: {X\_test.shape}, y\_test shape: {y\_test.shape}")  
  
#%% Elbow Method  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.model\_selection import cross\_val\_score  
import matplotlib.pyplot as plt  
  
k\_values = range(1, 50)  
cv\_scores = []  
  
for k in k\_values:  
 knn = KNeighborsClassifier(n\_neighbors=k)  
 scores = cross\_val\_score(knn, X\_train, y\_train, cv=10, scoring='accuracy')  
 cv\_scores.append(scores.mean())  
  
plt.figure(figsize=(10, 6))  
plt.plot(k\_values, cv\_scores, marker='o', linestyle='-', color='b')  
plt.xlabel('Number of Neighbors (K)')  
plt.ylabel('Cross-Validated Accuracy')  
plt.title('Elbow Method for Finding Optimal K')  
plt.grid(True)  
plt.xticks(k\_values)  
plt.show()  
  
optimal\_k = k\_values[cv\_scores.index(max(cv\_scores))]  
print(f"Optimal K: {optimal\_k}")  
  
#%% Model  
knn\_optimal = KNeighborsClassifier(n\_neighbors=optimal\_k)  
knn\_optimal.fit(X\_train, y\_train)  
  
accuracy = knn\_optimal.score(X\_test, y\_test)  
print(f"Test Set Accuracy with Optimal K ({optimal\_k}): {accuracy:.2f}")  
  
#%%  
y\_pred = knn\_optimal.predict(X\_test)  
y\_pred\_prob = knn\_optimal.predict\_proba(X\_test)[:, 1]  
  
#%% Confusion Matrix  
cm = confusion\_matrix(y\_test, y\_pred)  
disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=np.unique(y\_test))  
disp.plot(cmap=plt.cm.Blues)  
plt.title("Confusion Matrix")  
plt.show()  
  
if cm.shape == (2, 2): # Binary case  
 TN, FP, FN, TP = cm.ravel()  
  
 # Calculate metrics  
 precision = TP / (TP + FP) if (TP + FP) > 0 else 0  
 recall = TP / (TP + FN) if (TP + FN) > 0 else 0 # Sensitivity  
 specificity = TN / (TN + FP) if (TN + FP) > 0 else 0  
 f1\_score = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) > 0 else 0  
  
 print("Metrics:")  
 print(f"Precision: {precision:.2f}")  
 print(f"Recall (Sensitivity): {recall:.2f}")  
 print(f"Specificity: {specificity:.2f}")  
 print(f"F1-Score: {f1\_score:.2f}")  
  
else:  
 print("Classification Report:")  
 print(classification\_report(y\_test, y\_pred))  
  
#%%  
if len(np.unique(y\_test)) == 2: # Check for binary classification  
 fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred\_prob)  
 roc\_auc = auc(fpr, tpr)  
  
 # Plot ROC Curve  
 plt.figure(figsize=(8, 6))  
 plt.plot(fpr, tpr, label=f"AUC = {roc\_auc:.2f}", lw=2)  
 plt.plot([0, 1], [0, 1], 'k--', lw=2, label="Random Guess")  
 plt.xlabel("False Positive Rate")  
 plt.ylabel("True Positive Rate")  
 plt.title("ROC and AUC Curve")  
 plt.legend(loc="lower right")  
 plt.grid(True)  
 plt.show()  
  
 print(f"ROC-AUC: {roc\_auc:.2f}")  
else:  
 print("ROC-AUC is not available for multi-class classification without binarization.")  
  
if len(np.unique(y\_test)) > 2: # Multi-class case  
 from sklearn.preprocessing import label\_binarize  
  
 # Binarize the labels  
 y\_test\_binarized = label\_binarize(y\_test, classes=np.unique(y\_test))  
 y\_pred\_prob\_multi = knn\_optimal.predict\_proba(X\_test)  
  
 # Calculate AUC for each class  
 roc\_auc\_multi = roc\_auc\_score(y\_test\_binarized, y\_pred\_prob\_multi, average="macro")  
 print(f"Macro-Averaged ROC-AUC: {roc\_auc\_multi:.2f}")  
#%%  
classes = np.unique(y\_test) # Extract unique classes  
y\_test\_binarized = label\_binarize(y\_test, classes=classes)  
y\_pred\_prob = knn\_optimal.predict\_proba(X\_test) # Probabilities for each class  
  
# Compute ROC curve and AUC for each class  
fpr = {}  
tpr = {}  
roc\_auc = {}  
  
for i, class\_label in enumerate(classes):  
 fpr[class\_label], tpr[class\_label], \_ = roc\_curve(y\_test\_binarized[:, i], y\_pred\_prob[:, i])  
 roc\_auc[class\_label] = auc(fpr[class\_label], tpr[class\_label])  
  
# Plot ROC curves for all classes  
plt.figure(figsize=(10, 8))  
for class\_label in classes:  
 plt.plot(fpr[class\_label], tpr[class\_label],  
 label=f"Class {class\_label} (AUC = {roc\_auc[class\_label]:.2f})")  
  
# Plot random guess line  
plt.plot([0, 1], [0, 1], 'k--', label="Random Guess", lw=2, color='black')  
  
# Customize plot  
plt.xlabel("False Positive Rate")  
plt.ylabel("True Positive Rate")  
plt.title("Multi-Class ROC Curve")  
plt.legend(loc="lower right")  
plt.grid(True)  
plt.show()  
#%%  
skf = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)  
  
# Cross-validation for accuracy  
cv\_scores\_accuracy = cross\_val\_score(knn\_optimal, X\_train, y\_train, cv=skf, scoring='accuracy')  
  
# Print results  
print("Stratified K-Fold Cross-Validation Results:")  
print(f"Accuracy Scores: {cv\_scores\_accuracy}")  
print(f"Mean Accuracy: {cv\_scores\_accuracy.mean():.2f}")  
print(f"Standard Deviation: {cv\_scores\_accuracy.std():.2f}")  
  
#%%  
from sklearn.model\_selection import cross\_val\_score  
from sklearn.metrics import make\_scorer, f1\_score  
  
cv\_f1 = cross\_val\_score(knn, X\_train, y\_train, cv=5, scoring=make\_scorer(f1\_score, average='macro'))  
#%%  
k\_values = range(1, 50)  
#%%  
from sklearn.model\_selection import RepeatedStratifiedKFold  
  
rskf = RepeatedStratifiedKFold(n\_splits=5, n\_repeats=3, random\_state=42)  
cv\_scores = cross\_val\_score(knn, X\_train, y\_train, cv=rskf, scoring='accuracy')  
  
#%% SVM  
from sklearn.model\_selection import train\_test\_split  
  
# RBF Kernel: Normalized data  
y\_iso\_cleaned\_normalized = y[:len(X\_iso\_cleaned\_normalized)] # Ensure alignment  
X\_train\_rbf, X\_test\_rbf, y\_train\_rbf, y\_test\_rbf = train\_test\_split(  
 X\_iso\_cleaned\_normalized, y\_iso\_cleaned\_normalized, test\_size=0.2, random\_state=42, stratify=y\_iso\_cleaned\_normalized  
)  
  
# Polynomial and Linear Kernels: Original data  
y\_iso\_cleaned = y[:len(X\_iso\_cleaned)] # Ensure alignment  
X\_train\_poly, X\_test\_poly, y\_train\_poly, y\_test\_poly = train\_test\_split(  
 X\_iso\_cleaned, y\_iso\_cleaned, test\_size=0.2, random\_state=42, stratify=y\_iso\_cleaned  
)  
  
# Use the same split for Linear kernel as Polynomial kernel  
X\_train\_linear, X\_test\_linear, y\_train\_linear, y\_test\_linear = X\_train\_poly, X\_test\_poly, y\_train\_poly, y\_test\_poly  
  
#%% RBF Kernel  
from sklearn.svm import SVC  
  
svm\_rbf = SVC(kernel='rbf', random\_state=42, probability=True)  
svm\_rbf.fit(X\_train\_rbf, y\_train\_rbf)  
y\_pred\_rbf = svm\_rbf.predict(X\_test\_rbf)  
accuracy\_rbf = svm\_rbf.score(X\_test\_rbf, y\_test\_rbf)  
print(f"RBF Kernel Accuracy: {accuracy\_rbf:.2f}")  
  
#%% Polynomial Kernel  
svm\_poly = SVC(kernel='poly', degree=3, random\_state=42, probability=True)  
svm\_poly.fit(X\_train\_poly, y\_train\_poly)  
  
# Predictions and Accuracy  
y\_pred\_poly = svm\_poly.predict(X\_test\_poly)  
accuracy\_poly = svm\_poly.score(X\_test\_poly, y\_test\_poly)  
print(f"Polynomial Kernel Accuracy: {accuracy\_poly:.2f}")  
  
#%% Linear Kernel  
svm\_linear = SVC(kernel='linear', random\_state=42, probability=True)  
svm\_linear.fit(X\_train\_linear, y\_train\_linear)  
  
# Predictions and Accuracy  
y\_pred\_linear = svm\_linear.predict(X\_test\_linear)  
accuracy\_linear = svm\_linear.score(X\_test\_linear, y\_test\_linear)  
print(f"Linear Kernel Accuracy: {accuracy\_linear:.2f}")  
  
#%%  
from sklearn.metrics import (  
 confusion\_matrix, ConfusionMatrixDisplay, classification\_report, roc\_curve, auc  
)  
import matplotlib.pyplot as plt  
  
def evaluate\_model(y\_test, y\_pred, model\_name):  
 # Confusion Matrix  
 cm = confusion\_matrix(y\_test, y\_pred)  
 disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=np.unique(y\_test))  
 disp.plot(cmap=plt.cm.Blues)  
 plt.title(f"Confusion Matrix: {model\_name}")  
 plt.show()  
  
 # Classification Report  
 print(f"{model\_name} Classification Report:")  
 print(classification\_report(y\_test, y\_pred))  
  
evaluate\_model(y\_test\_rbf, y\_pred\_rbf, "RBF Kernel")  
  
evaluate\_model(y\_test\_poly, y\_pred\_poly, "Polynomial Kernel")  
  
evaluate\_model(y\_test\_linear, y\_pred\_linear, "Linear Kernel")  
  
#%%  
def plot\_roc\_auc\_multiclass(model, X\_test, y\_test, model\_name):  
 from sklearn.preprocessing import label\_binarize  
 from sklearn.metrics import roc\_curve, auc  
 import matplotlib.pyplot as plt  
 import numpy as np  
  
 # Binarize labels for multi-class  
 classes = np.unique(y\_test)  
 y\_test\_binarized = label\_binarize(y\_test, classes=classes)  
 y\_pred\_prob = model.predict\_proba(X\_test)  
  
 # Compute ROC curve and AUC for each class  
 fpr = {}  
 tpr = {}  
 roc\_auc = {}  
  
 for i, class\_label in enumerate(classes):  
 fpr[class\_label], tpr[class\_label], \_ = roc\_curve(y\_test\_binarized[:, i], y\_pred\_prob[:, i])  
 roc\_auc[class\_label] = auc(fpr[class\_label], tpr[class\_label])  
  
 # Plot ROC curves for all classes  
 plt.figure(figsize=(10, 8))  
 for class\_label in classes:  
 plt.plot(fpr[class\_label], tpr[class\_label],  
 label=f"Class {class\_label} (AUC = {roc\_auc[class\_label]:.2f})")  
  
 # Plot random guess line  
 plt.plot([0, 1], [0, 1], 'k--', lw=2, label="Random Guess")  
  
 # Customize the plot  
 plt.xlabel("False Positive Rate")  
 plt.ylabel("True Positive Rate")  
 plt.title(f"Multi-Class ROC Curve: {model\_name}")  
 plt.legend(loc="lower right")  
 plt.grid(True)  
 plt.show()  
  
 # Compute and print macro-averaged AUC  
 macro\_auc = roc\_auc\_score(y\_test\_binarized, y\_pred\_prob, average="macro")  
 print(f"{model\_name} Macro-Averaged ROC-AUC: {macro\_auc:.2f}")  
#%%  
plot\_roc\_auc\_multiclass(svm\_rbf, X\_test\_rbf, y\_test\_rbf, "RBF Kernel")  
plot\_roc\_auc\_multiclass(svm\_poly, X\_test\_poly, y\_test\_poly, "Polynomial Kernel")  
plot\_roc\_auc\_multiclass(svm\_linear, X\_test\_linear, y\_test\_linear, "Linear Kernel")  
  
#%%  
def stratified\_k\_fold\_validation(model, X, y, model\_name, n\_splits=5):  
 skf = StratifiedKFold(n\_splits=n\_splits, shuffle=True, random\_state=42)  
 cv\_scores = cross\_val\_score(model, X, y, cv=skf, scoring='accuracy')  
  
 print(f"{model\_name} Stratified K-Fold Results:")  
 print(f"Fold Scores: {cv\_scores}")  
 print(f"Mean Accuracy: {cv\_scores.mean():.2f}")  
 print(f"Standard Deviation: {cv\_scores.std():.2f}\n")  
#%%  
stratified\_k\_fold\_validation(svm\_rbf, X\_iso\_cleaned\_normalized, y[:len(X\_iso\_cleaned\_normalized)], "RBF Kernel")  
stratified\_k\_fold\_validation(svm\_poly, X\_iso\_cleaned, y[:len(X\_iso\_cleaned)], "Polynomial Kernel")  
stratified\_k\_fold\_validation(svm\_linear, X\_iso\_cleaned, y[:len(X\_iso\_cleaned)], "Linear Kernel")  
  
  
#%% Naive Bayes  
from sklearn.model\_selection import train\_test\_split  
  
y\_lof\_cleaned\_normalized = y[:len(X\_lof\_cleaned\_normalized)]  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(  
 X\_lof\_cleaned\_normalized,  
 y\_lof\_cleaned\_normalized,  
 test\_size=0.2,  
 random\_state=42,  
 stratify=y\_lof\_cleaned\_normalized  
)  
  
print(f"X\_train shape: {X\_train.shape}, y\_train shape: {y\_train.shape}")  
print(f"X\_test shape: {X\_test.shape}, y\_test shape: {y\_test.shape}")  
#%%  
from sklearn.naive\_bayes import GaussianNB  
  
nb\_model = GaussianNB()  
nb\_model.fit(X\_train, y\_train)  
y\_pred = nb\_model.predict(X\_test)  
accuracy = nb\_model.score(X\_test, y\_test)  
print(f"Naive Bayes Model Accuracy: {accuracy:.2f}")  
  
#%%  
from sklearn.metrics import (  
 confusion\_matrix, ConfusionMatrixDisplay, classification\_report, roc\_auc\_score  
)  
  
cm = confusion\_matrix(y\_test, y\_pred)  
disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=np.unique(y\_test))  
disp.plot(cmap=plt.cm.Blues)  
plt.title("Confusion Matrix")  
plt.show()  
  
print("Classification Report:")  
print(classification\_report(y\_test, y\_pred))  
  
#%%  
from sklearn.preprocessing import label\_binarize  
from sklearn.metrics import roc\_curve, auc  
  
# Binarize the labels for multi-class  
classes = np.unique(y\_test)  
y\_test\_binarized = label\_binarize(y\_test, classes=classes)  
y\_pred\_prob = nb\_model.predict\_proba(X\_test)  
fpr = {}  
tpr = {}  
roc\_auc = {}  
  
for i, class\_label in enumerate(classes):  
 fpr[class\_label], tpr[class\_label], \_ = roc\_curve(y\_test\_binarized[:, i], y\_pred\_prob[:, i])  
 roc\_auc[class\_label] = auc(fpr[class\_label], tpr[class\_label])  
plt.figure(figsize=(10, 8))  
for class\_label in classes:  
 plt.plot(fpr[class\_label], tpr[class\_label],  
 label=f"Class {class\_label} (AUC = {roc\_auc[class\_label]:.2f})")  
  
plt.plot([0, 1], [0, 1], 'k--', label="Random Guess", lw=2)  
plt.xlabel("False Positive Rate")  
plt.ylabel("True Positive Rate")  
plt.title("Multi-Class ROC Curve")  
plt.legend(loc="lower right")  
plt.grid(True)  
plt.show()  
#%%  
from sklearn.metrics import roc\_auc\_score  
macro\_auc = roc\_auc\_score(y\_test\_binarized, y\_pred\_prob, average="macro")  
print(f"Macro-Averaged ROC-AUC: {macro\_auc:.2f}")  
weighted\_auc = roc\_auc\_score(y\_test\_binarized, y\_pred\_prob, average="weighted")  
print(f"Weighted-Averaged ROC-AUC: {weighted\_auc:.2f}")  
#%%  
import numpy as np  
from sklearn.model\_selection import StratifiedKFold  
from sklearn.metrics import roc\_auc\_score  
from sklearn.preprocessing import label\_binarize  
from sklearn.naive\_bayes import GaussianNB  
  
def stratified\_k\_fold\_naive\_bayes(model, X, y, n\_splits=5):  
 skf = StratifiedKFold(n\_splits=n\_splits, shuffle=True, random\_state=42)  
  
 # Ensure X and y are NumPy arrays for proper indexing  
 if isinstance(X, pd.DataFrame) or isinstance(X, pd.Series):  
 X = X.values  
 if isinstance(y, pd.DataFrame) or isinstance(y, pd.Series):  
 y = y.values  
  
 accuracy\_scores = []  
 auc\_scores = []  
  
 for train\_idx, test\_idx in skf.split(X, y):  
  
 X\_train, X\_test = X[train\_idx], X[test\_idx]  
 y\_train, y\_test = y[train\_idx], y[test\_idx]  
  
 model.fit(X\_train, y\_train)  
  
 y\_pred\_prob = model.predict\_proba(X\_test)  
  
 accuracy = model.score(X\_test, y\_test)  
 accuracy\_scores.append(accuracy)  
  
 y\_test\_binarized = label\_binarize(y\_test, classes=np.unique(y))  
 auc = roc\_auc\_score(y\_test\_binarized, y\_pred\_prob, average="macro")  
 auc\_scores.append(auc)  
  
 print(f"K-Fold Accuracy Scores: {accuracy\_scores}")  
 print(f"Mean Accuracy: {np.mean(accuracy\_scores):.2f}, Standard Deviation: {np.std(accuracy\_scores):.2f}")  
 print(f"K-Fold AUC Scores: {auc\_scores}")  
 print(f"Mean AUC: {np.mean(auc\_scores):.2f}, Standard Deviation: {np.std(auc\_scores):.2f}")  
#%%  
nb\_model = GaussianNB()  
  
stratified\_k\_fold\_naive\_bayes(nb\_model, X\_lof\_cleaned\_normalized, y[:len(X\_lof\_cleaned\_normalized)], n\_splits=5)

Appendix E: Clustering and Association Rule Mining

# Dimensionality reduction  
#%% Variance Inflation Factor (VIF)  
from statsmodels.stats.outliers\_influence import variance\_inflation\_factor  
import pandas as pd  
X\_vif = balanced\_data.select\_dtypes(include=['float64', 'int64']) # Include only numeric columns  
vif\_data = pd.DataFrame()  
vif\_data['feature'] = X\_vif.columns  
vif\_data['VIF'] = [variance\_inflation\_factor(X\_vif.values, i) for i in range(X\_vif.shape[1])]  
while vif\_data['VIF'].max() > 10:  
 high\_vif\_feature = vif\_data.loc[vif\_data['VIF'].idxmax(), 'feature']  
 X\_vif = X\_vif.drop(columns=[high\_vif\_feature])  
 vif\_data = pd.DataFrame()  
 vif\_data['feature'] = X\_vif.columns  
 vif\_data['VIF'] = [variance\_inflation\_factor(X\_vif.values, i) for i in range(X\_vif.shape[1])]  
  
print("Final VIF Data:")  
print(vif\_data)  
  
#%% Principal Component Analysis (PCA)  
from sklearn.decomposition import PCA  
pca = PCA(n\_components=0.95)  
X\_pca = pca.fit\_transform(X\_vif)  
  
print(f"PCA Explained Variance Ratios: {pca.explained\_variance\_ratio\_}")  
print(f"Number of Components Retained: {pca.n\_components\_}")  
  
#%% Singular Value Decomposition (SVD)  
from sklearn.decomposition import TruncatedSVD  
  
svd = TruncatedSVD(n\_components=min(X\_vif.shape[1], 6)) # Retain top components  
X\_svd = svd.fit\_transform(X\_vif)  
  
print(f"SVD Explained Variance Ratios: {svd.explained\_variance\_ratio\_}")  
  
#%% Summary  
  
dimensionality\_reduction\_results = {  
 "VIF": vif\_data,  
 "PCA Explained Variance Ratio": pca.explained\_variance\_ratio\_,  
 "SVD Explained Variance Ratio": svd.explained\_variance\_ratio\_  
}  
  
print("Dimensionality Reduction Summary:")  
for key, value in dimensionality\_reduction\_results.items():  
 print(f"{key}:")  
 print(value)  
  
#%%  
pca = PCA(n\_components=2) # Reduce to 2 components for clustering  
X\_pca = pca.fit\_transform(balanced\_data)  
  
print(f"PCA Explained Variance Ratios: {pca.explained\_variance\_ratio\_}")  
  
#%% Step 2: Outlier Detection using IQR  
def detect\_outliers\_iqr(data):  
 *"""Detects outliers using the IQR method."""* Q1 = np.percentile(data, 25, axis=0)  
 Q3 = np.percentile(data, 75, axis=0)  
 IQR = Q3 - Q1  
 lower\_bound = Q1 - 1.5 \* IQR  
 upper\_bound = Q3 + 1.5 \* IQR  
 mask = np.all((data >= lower\_bound) & (data <= upper\_bound), axis=1)  
 return mask  
  
outlier\_mask = detect\_outliers\_iqr(X\_pca)  
X\_pca\_cleaned = X\_pca[outlier\_mask]  
  
print(f"Original data shape: {X\_pca.shape}")  
print(f"Cleaned data shape: {X\_pca\_cleaned.shape}")  
  
  
#%% Step 2: Outlier Detection using Z-Score  
from scipy.stats import zscore  
  
z\_scores = np.abs(zscore(X\_pca))  
  
threshold = 3  
outlier\_mask = (z\_scores < threshold).all(axis=1)  
  
X\_pca\_cleaned = X\_pca[outlier\_mask]  
  
print(f"Original data shape: {X\_pca.shape}")  
print(f"Cleaned data shape: {X\_pca\_cleaned.shape}")  
  
#%%  
from sklearn.cluster import KMeans  
from sklearn.metrics import silhouette\_score  
  
# %% Define range of K values  
range\_k = range(2, 11)  
inertia = []  
silhouette\_scores = []  
  
for k in range\_k:  
 kmeans = KMeans(n\_clusters=k, init='k-means++', random\_state=42)  
 kmeans.fit(X\_pca\_cleaned)  
 inertia.append(kmeans.inertia\_)  
 silhouette\_scores.append(silhouette\_score(X\_pca\_cleaned, kmeans.labels\_))  
  
# %% Plot Elbow Method (Within-Cluster Variation)  
plt.figure(figsize=(8, 5))  
plt.plot(range\_k, inertia, marker='o', linestyle='--')  
plt.title('Elbow Method: Within-Cluster Variation vs. K')  
plt.xlabel('Number of Clusters (K)')  
plt.ylabel('Within-Cluster Variation (Inertia)')  
plt.xticks(range\_k)  
plt.grid()  
plt.show()  
  
# %% Plot Silhouette Analysis  
plt.figure(figsize=(8, 5))  
plt.plot(range\_k, silhouette\_scores, marker='o', linestyle='--', color='orange')  
plt.title('Silhouette Analysis: Silhouette Score vs. K')  
plt.xlabel('Number of Clusters (K)')  
plt.ylabel('Silhouette Score')  
plt.xticks(range\_k)  
plt.grid()  
plt.show()  
  
# %% Optimal K and Final K-Means Clustering  
optimal\_k = range\_k[np.argmax(silhouette\_scores)]  
print(f"Optimal Number of Clusters (K): {optimal\_k}")  
final\_kmeans = KMeans(n\_clusters=optimal\_k, init='k-means++', random\_state=42)  
final\_kmeans.fit(X\_pca\_cleaned)  
cluster\_labels = final\_kmeans.labels\_  
  
# %% Visualize Final Clusters  
plt.figure(figsize=(8, 5))  
plt.scatter(X\_pca\_cleaned[:, 0], X\_pca\_cleaned[:, 1], c=cluster\_labels, cmap='viridis', s=30)  
plt.scatter(final\_kmeans.cluster\_centers\_[:, 0], final\_kmeans.cluster\_centers\_[:, 1], c='red', marker='X', s=200,  
 label='Centroids')  
plt.title(f"Final Clusters with K={optimal\_k}")  
plt.xlabel("Principal Component 1")  
plt.ylabel("Principal Component 2")  
plt.legend()  
plt.grid()  
plt.show()  
  
#%% DBSCAN  
from sklearn.cluster import DBSCAN  
from sklearn.neighbors import NearestNeighbors  
  
  
#%% k-Distance Plot for Optimal Eps  
  
k = 4  
nearest\_neighbors = NearestNeighbors(n\_neighbors=k)  
neighbors = nearest\_neighbors.fit(X\_pca\_cleaned)  
distances, indices = neighbors.kneighbors(X\_pca\_cleaned)  
  
distances = np.sort(distances[:, k - 1], axis=0)  
plt.figure(figsize=(8, 5))  
plt.plot(distances)  
plt.title("k-Distance Graph")  
plt.xlabel("Data Points (sorted)")  
plt.ylabel("k-Distance")  
plt.grid()  
plt.show()  
  
  
  
#%% Apply DBSCAN  
  
eps = 800  
min\_samples = 5  
  
dbscan = DBSCAN(eps=eps, min\_samples=min\_samples)  
dbscan.fit(X\_pca\_cleaned)  
  
dbscan\_labels = dbscan.labels\_  
  
#%% Visualize DBSCAN Clusters  
  
plt.figure(figsize=(8, 5))  
plt.scatter(X\_pca\_cleaned[:, 0], X\_pca\_cleaned[:, 1], c=dbscan\_labels, cmap='viridis', s=30)  
plt.title("DBSCAN Clustering")  
plt.xlabel("Principal Component 1")  
plt.ylabel("Principal Component 2")  
plt.grid()  
plt.show()  
  
#%% Summary of Clusters  
  
n\_clusters = len(set(dbscan\_labels)) - (1 if -1 in dbscan\_labels else 0)  
n\_noise = list(dbscan\_labels).count(-1)  
  
print(f"Number of Clusters: {n\_clusters}")  
print(f"Number of Noise Points: {n\_noise}")  
  
  
  
  
#%% Apriori  
from mlxtend.frequent\_patterns import apriori, association\_rules  
from sklearn.preprocessing import Binarizer  
svd\_columns = [f"component\_{i}" for i in range(X\_svd.shape[1])]  
svd\_df = pd.DataFrame(X\_svd, columns=svd\_columns)  
  
#%%  
binarizer = Binarizer(threshold=0.0)  
binary\_data = binarizer.fit\_transform(svd\_df)  
binary\_df = pd.DataFrame(binary\_data, columns=svd\_columns)  
  
print("Binarized Data (Ready for Apriori):")  
print(binary\_df.head())  
  
#%%  
frequent\_itemsets = apriori(binary\_df, min\_support=0.2, use\_colnames=True, verbose=1)  
  
print("Frequent Itemsets:")  
print(frequent\_itemsets)  
  
#%%  
rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=0.6)  
rules = rules.sort\_values(['confidence', 'lift'], ascending=[False, False])  
  
print("Association Rules:")  
print(rules.to\_string())  
  
#%%  
import matplotlib.pyplot as plt  
  
plt.figure(figsize=(8, 5))  
plt.scatter(rules['support'], rules['confidence'], alpha=0.6, c=rules['lift'], cmap='viridis')  
plt.colorbar(label='Lift')  
plt.title('Support vs Confidence')  
plt.xlabel('Support')  
plt.ylabel('Confidence')  
plt.grid()

Appendix F: Combine the Dataset

#%%READ THE URLS / I have done this with the actual file because the url was not working  
action= pd.read\_csv('Info Vis Term Project/action.csv')  
adventure = pd.read\_csv('Info Vis Term Project/adventure.csv')  
animation = pd.read\_csv('Info Vis Term Project/animation.csv')  
biography= pd.read\_csv('Info Vis Term Project/biography.csv')  
crime = pd.read\_csv('Info Vis Term Project/crime.csv')  
family = pd.read\_csv('Info Vis Term Project/family.csv')  
fantasy = pd.read\_csv('Info Vis Term Project/fantasy.csv')  
film\_noir = pd.read\_csv('Info Vis Term Project/film-noir.csv')  
history = pd.read\_csv('Info Vis Term Project/history.csv')  
horror = pd.read\_csv('Info Vis Term Project/horror.csv')  
mystery = pd.read\_csv('Info Vis Term Project/mystery.csv')  
romance = pd.read\_csv('Info Vis Term Project/romance.csv')  
scifi = pd.read\_csv('Info Vis Term Project/scifi.csv')  
sports = pd.read\_csv('Info Vis Term Project/sports.csv')  
thriller = pd.read\_csv('Info Vis Term Project/thriller.csv')  
war = pd.read\_csv('Info Vis Term Project/war.csv')  
  
#%% Making a different column with its own genre  
action['genre'] = 'Action'  
crime['genre'] = 'Crime'  
adventure['genre'] = 'Adventure'  
thriller['genre'] = 'Thriller'  
family['genre'] = 'Family'  
mystery['genre'] = 'Mystery'  
scifi['genre'] = 'Sci-Fi'  
history['genre'] = 'History'  
sports['genre'] = 'Sports'  
animation['genre'] = 'Animation'  
war['genre'] = 'War'  
biography['genre'] = 'Biography'  
horror['genre'] = 'Horror'  
fantasy['genre'] = 'Fantasy'  
romance['genre'] = 'Romance'  
film\_noir['genre'] = 'Film-Noir'  
#%% Making a dataframe with all of the above datas  
df = pd.concat([action, crime, adventure, thriller,  
 family, mystery, scifi, history,  
 sports, animation, war, biography,  
 horror, fantasy, romance, film\_noir])  
  
df =df.reset\_index(drop=True)  
print(df.head(10))  
  
#%%  
# Save the concatenated DataFrame 'df' to a CSV file  
output\_file\_path = 'combined\_genres.csv' # Replace with your desired file path and name  
df.to\_csv(output\_file\_path, index=False)  
  
print(f"CSV file has been generated and saved as '{output\_file\_path}'")

Appendix G: Instruction for the dataset

* I used the code in Appendix E to make one dataset
* GitHub of the datasets: <https://github.com/muhit009/Movie_Information-Visualization-and-Machine-Learning>
* You need to use this GitHub to get the dataset and replace the raw url inside the pd.read\_csv()

References:

1. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830. Retrieved from https://scikit-learn.org/

2. Waskom, M. L. (2021). Seaborn: Statistical data visualization. Journal of Open Source Software, 6(60), 3021. Retrieved from https://seaborn.pydata.org/

3. Raschka, S. (2018). MLxtend: Providing machine learning and data science utilities and extensions to Python’s scientific computing stack. Retrieved from http://rasbt.github.io/mlxtend/

4. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825), 357–362. Retrieved from https://numpy.org/

5.McKinney, W. (2010). Data structures for statistical computing in Python. Proceedings of the 9th Python in Science Conference, 51–56. Retrieved from <https://pandas.pydata.org/>

6. Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), 90–95. Retrieved from <https://matplotlib.org/>

7. Maryadi, R. (n.d.). *IMDB movie data*. Kaggle. Retrieved December 8, 2024, from <https://www.kaggle.com/code/rizkymaryadi/imdb-movie-data>

8. Rajugc. (n.d.). *IMDB movies dataset based on genre*. Kaggle. Retrieved December 8, 2024, from <https://www.kaggle.com/datasets/rajugc/imdb-movies-dataset-based-on-genre/data>