Movie Collection Prediction

Introduction

In the film industry, predicting the success of a movie is a complex task influenced by various factors such as production costs, marketing efforts, actor ratings, and social media engagement. Understanding these factors can help filmmakers and marketers make better decisions to maximize a movie's box office revenue.

This research explores a dataset containing information on different aspects of movies, including expenses, ratings, and social media metrics. By analyzing this data, we aim to uncover patterns and insights that can predict a movie's success. The study involves enhancing the dataset through data augmentation, creating new features for deeper insights, and building predictive models to forecast box office collections.

Our goal is to provide actionable recommendations that can help industry stakeholders improve their strategies for movie production and promotion.

Import libraries

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn import linear_model
import sklearn.metrics as metrics
```

Load The Dataset

```
# Loading the dataset
df = pd.read_csv("/content/Movie_classification.csv")

# Displaying the first few rows of the dataset
df.head()
```

	Marketing expense	Production expense	Multiplex coverage	Budget	Movie_length	Lead_ Actor_Rating	Lead_Actre
0	20.1264	59.62	0.462	36524.125	138.7	7.825	
1	20.5462	69.14	0.531	35668.655	152.4	7.505	
2	20.5458	69.14	0.531	39912.675	134.6	7.485	
3	20.6474	59.36	0.542	38873.890	119.3	6.895	
4	21.3810	59.36	0.542	39701.585	127.7	6.920	

Understanding dataset

```
format(df.shape) # Displaying the shape of the DataFrame
```

```
'(506, 19)'
```

```
df.info() # Displaying information about the DataFrame
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Marketing expense    506 non-null    float64
 1   Production expense   506 non-null    float64
 2   Multiplex coverage   506 non-null    float64
 3   Budget               506 non-null    float64
 4   Movie_length         506 non-null    float64
 5   Lead_ Actor_Rating   506 non-null    float64
 6   Lead_Actress_rating  506 non-null    float64
 7   Director_rating      506 non-null    float64
 8   Producer_rating      506 non-null    float64
 9   Critic_rating        506 non-null    float64
 10  Trailer_views        506 non-null    int64  
 11  3D_available         506 non-null    object 
 12  Time_taken           494 non-null    float64
```

```
 13  Twitter_hastags      506 non-null    float64
 14  Genre                506 non-null    object
 15  Avg_age_actors       506 non-null    int64
 16  Num_multiplex        506 non-null    int64
 17  Collection           506 non-null    int64
 18  Start_Tech_Oscar     506 non-null    int64
dtypes: float64(12), int64(5), object(2)
memory usage: 75.2+ KB
```

```
df.describe() # Generating descriptive statistics for the DataFrame
```

	Marketing expense	Production expense	Multiplex coverage	Budget	Movie_length	Lead_ Actor_Rating
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	92.270471	77.273557	0.445305	34911.144022	142.074901	8.014002
std	172.030902	13.720706	0.115878	3903.038232	28.148861	1.054266
min	20.126400	55.920000	0.129000	19781.355000	76.400000	3.840000
25%	21.640900	65.380000	0.376000	32693.952500	118.525000	7.316250
50%	25.130200	74.380000	0.462000	34488.217500	151.000000	8.307500
75%	93.541650	91.200000	0.551000	36793.542500	167.575000	8.865000
max	1799.524000	110.480000	0.615000	48772.900000	173.500000	9.435000

```
df.loc[:,:] # Accessing all rows and columns of the DataFrame using the .loc indexer
```

	Marketing expense	Production expense	Multiplex coverage	Budget	Movie_length	Lead_ Actor_Rating	Lead_Act
0	20.1264	59.62	0.462	36524.125	138.7	7.825	
1	20.5462	69.14	0.531	35668.655	152.4	7.505	
2	20.5458	69.14	0.531	39912.675	134.6	7.485	
3	20.6474	59.36	0.542	38873.890	119.3	6.895	
4	21.3810	59.36	0.542	39701.585	127.7	6.920	
...	...	...	...	...	...	...	
501	21.2526	78.86	0.427	36624.115	142.6	8.680	
502	20.9054	78.86	0.427	33996.600	150.2	8.780	
503	21.2152	78.86	0.427	38751.680	164.5	8.830	
504	22.1918	78.86	0.427	37740.670	162.8	8.730	
505	20.9482	78.86	0.427	33496.650	154.3	8.640	

506 rows × 19 columns

```
df.columns # Accessing the column names of the DataFrame
```

```
Index(['Marketing expense', 'Production expense', 'Multiplex coverage',
       'Budget', 'Movie_length', 'Lead_ Actor_Rating', 'Lead_Actress_rating',
       'Director_rating', 'Producer_rating', 'Critic_rating', 'Trailer_views',
       '3D_available', 'Time_taken', 'Twitter_hastags', 'Genre',
       'Avg_age_actors', 'Num_multiplex', 'Collection', 'Start_Tech_Oscar'],
      dtype='object')
```

Correlation Analysis

Correlation analysis is a crucial step in data preprocessing, helping to improve the quality and stability of subsequent analyses and models by identifying and removing redundant or highly correlated variables

```
import numpy as np  # Import NumPy library for numerical computations

# Select only numeric columns for correlation calculation
numeric_df = df.select_dtypes(include=['number'])  # Filter the DataFrame to include only numeric columns

# Calculate correlation matrix
corr_matrix = numeric_df.corr().abs()  # Compute the absolute correlation matrix for numerical variables

# Drop highly correlated columns
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))  # Create a boolean mask to select the upper triangle (
to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]  # Identify columns with correlations above 0.95
df.drop(to_drop, axis=1, inplace=True)  # Drop highly correlated columns from the DataFrame
```

```
corr_matrix # Calculate correlation matrix
```

	Marketing expense	Production expense	Multiplex coverage	Budget	Movie_length	Actor_Ra
Marketing expense	1.000000	0.406583	0.420972	0.219247	0.352734	0.38
Production expense	0.406583	1.000000	0.763651	0.391676	0.644779	0.70
Multiplex coverage	0.420972	0.763651	1.000000	0.302188	0.731470	0.76
Budget	0.219247	0.391676	0.302188	1.000000	0.240265	0.20
Movie_length	0.352734	0.644779	0.731470	0.240265	1.000000	0.74
Lead_ Actor_Rating	0.380050	0.706481	0.768589	0.208464	0.746904	1.00
Lead_Actress_rating	0.379813	0.707956	0.769724	0.203981	0.746493	0.99
Director_rating	0.380069	0.707566	0.769157	0.201907	0.747021	0.99
Producer_rating	0.376462	0.705819	0.764873	0.205397	0.746707	0.99
Critic_rating	0.184985	0.251565	0.145555	0.232361	0.217830	0.16
Trailer_views	0.443457	0.591657	0.581386	0.602536	0.589318	0.49
Time_taken	0.026019	0.015888	0.035922	0.040773	0.019984	0.03
Twitter_hastags	0.013518	0.000839	0.004882	0.030674	0.009380	0.01
Avg_age_actors	0.059204	0.055810	0.092104	0.064694	0.075198	0.03
Num_multiplex	0.383298	0.707559	0.915495	0.282796	0.673896	0.70
Collection	0.389582	0.484754	0.429300	0.696304	0.377999	0.25
Start_Tech_Oscar	0.013417	0.024404	0.004017	0.027148	0.016291	0.03

Result Of Correlation analysis

```
max_corr = corr_matrix.max().max()  # Maximum correlation coefficient
min_corr = corr_matrix.min().min()  # Minimum correlation coefficient

# Find variables with maximum correlation
max_corr_var = corr_matrix.stack()[corr_matrix.stack() == max_corr].index.tolist()
# Find variables with minimum correlation
min_corr_var = corr_matrix.stack()[corr_matrix.stack() == min_corr].index.tolist()

print("Maximum correlation coefficient:", max_corr)
print("Variables with maximum correlation:", max_corr_var)
print("Minimum correlation coefficient:", min_corr)
print("Variables with minimum correlation:", min_corr_var)
```

```
Maximum correlation coefficient: 1.0
Variables with maximum correlation: [('Marketing expense', 'Marketing expense'), ('Production expense', 'Production expense'), ('Mul
Minimum correlation coefficient: 0.0008386303900854655
Variables with minimum correlation: [('Production expense', 'Twitter_hastags'), ('Twitter_hastags', 'Production expense')]
```

Maximum Correlation Coefficient: The maximum correlation coefficient of 1.0 indicates a perfect linear relationship between certain pairs of variables. Specifically, all the variables listed in the "Variables with maximum correlation" section have a correlation coefficient of 1.0 with themselves. This is expected because a variable is perfectly correlated with itself.

Minimum Correlation Coefficient: The minimum correlation coefficient, which is close to zero (0.00083863), indicates a very weak linear relationship between the variables. In this case, the pair of variables with the minimum correlation coefficient is ('Production expense', 'Twitter_hastags'). This suggests that there is almost no linear relationship between the production expenses and the number of Twitter hashtags associated with a movie.

PCA Preprocessing with Data Imputation

PCA is used for dimensionality reduction by capturing the most important patterns in the data while reducing the number of features. It is beneficial for various purposes:Dimensionality Reduction, Visualization, Feature Engineering etc.

Import libraries

```
#Import necessary libraries
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer
```

Separate categorical and numerical columns

```
# Separate categorical and numerical columns
categorical_columns = ['3D_available', 'Genre']  # Assuming these are the categorical columns
numerical_columns = [col for col in df.columns if col not in categorical_columns and col != 'Collection']
```

One-hot encode categorical columns

```
# One-hot encode categorical columns
df_encoded = pd.get_dummies(df, columns=categorical_columns)
```

Separate the features X and target variable y

```
X = df_encoded.drop(columns=['Collection'])  # 'Collection' is the target variable
y = df['Collection']
```

Impute missing values in numerical columns with the mean and standardize the faeture

```
imputer = SimpleImputer(strategy='mean')
X[numerical_columns] = imputer.fit_transform(X[numerical_columns])
# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

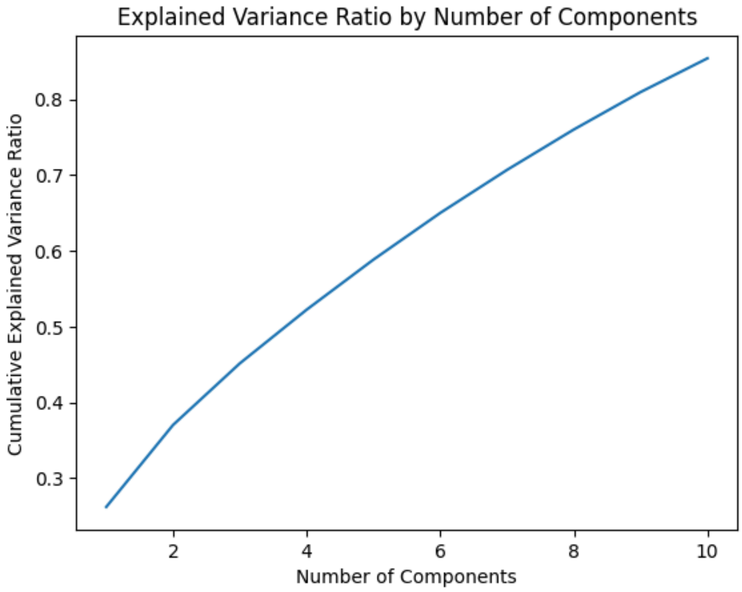
Perform PCA

```
# Define the number of components
n_components = 10  # Specify the number of components you want to retain

# Create an instance of PCA
pca = PCA(n_components=n_components)

# Fit PCA to the scaled data
X_pca = pca.fit_transform(X_scaled)

# Create a DataFrame for the transformed features
X_pca_df = pd.DataFrame(data=X_pca, columns=[f'PC{i+1}' for i in range(n_components)])
```

Analyze the Result

```
# Print the explained variance ratio of each component
print("Explained Variance Ratio:")
print(pca.explained_variance_ratio_)

# Optionally, visualize the explained variance ratio
plt.plot(range(1, n_components+1), np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Explained Variance Ratio by Number of Components')
plt.show()
```

```
Explained Variance Ratio:
[0.2621459  0.10815889 0.08124134 0.07080795 0.06604524 0.06158269
 0.05678882 0.05328204 0.04939722 0.0444814 ]
```



The result obtained from PCA (Principal Component Analysis) is the explained variance ratio of each principal component. The explained variance ratio represents the proportion of the dataset's variance that lies along the axis of each principal component. This information helps us understand how much information each principal component retains from the original dataset. The first principal component (PC1) explains approximately 26.21% of the total variance in the dataset. The second principal component (PC2) explains approximately 10.82% of the total variance. Similarly, the third, fourth, fifth, and subsequent principal components explain decreasing proportions of the variance.

Data Preprocessing

Treating with Null values

Count of outliers

```
def count_outliers(data,col):
        q1 = data[col].quantile(0.25,interpolation='nearest')
        q2 = data[col].quantile(0.5,interpolation='nearest')
        q3 = data[col].quantile(0.75,interpolation='nearest')
        q4 = data[col].quantile(1,interpolation='nearest')
        IQR = q3 -q1
        global LLP
        global ULP
        LLP = q1 - 1.5*IQR
        ULP = q3 + 1.5*IQR
        if data[col].min() > LLP and data[col].max() < ULP:
            print("No outliers in",i)
        else:
            print("There are outliers in",i)
            x = data[data[col]<LLP][col].size
            y = data[data[col]>ULP][col].size
            a.append(i)
            print('Count of outliers are:',x+y)
global a
a = []
for i in x.columns:
    count_outliers(x,i)
```

```
There are outliers in Marketing expense
Count of outliers are: 66
No outliers in Production expense
No outliers in Multiplex coverage
There are outliers in Budget
Count of outliers are: 30
No outliers in Movie_length
There are outliers in Lead_ Actor_Rating
Count of outliers are: 5
No outliers in Critic_rating
There are outliers in Trailer_views
Count of outliers are: 10
There are outliers in Time_taken
Count of outliers are: 2
There are outliers in Twitter_hastags
Count of outliers are: 2
No outliers in Avg_age_actors
There are outliers in Num_multiplex
Count of outliers are: 3
There are outliers in Collection
Count of outliers are: 37
No outliers in Start_Tech_Oscar
```

```
df.isnull().sum()
```

```
Marketing expense      0
Production expense     0
Multiplex coverage     0
Budget                 0
Movie_length           0
Lead_ Actor_Rating     0
Critic_rating          0
Trailer_views          0
3D_available           0
Time_taken            12
Twitter_hastags        0
Genre                  0
Avg_age_actors         0
Num_multiplex          0
Collection             0
Start_Tech_Oscar       0
dtype: int64
```

```
# Since there are outliers in time_taken column we should replace null with median
df['Time_taken'].fillna(df['Time_taken'].median(),inplace=True)
```

```
df.isnull().sum() #after replacing with null values checking the null values
```

```
Marketing expense     0
Production expense    0
Multiplex coverage    0
Budget                0
Movie_length          0
Lead_ Actor_Rating    0
Critic_rating         0
Trailer_views         0
3D_available          0
Time_taken            0
Twitter_hastags       0
Genre                 0
Avg_age_actors        0
Num_multiplex         0
Collection            0
Start_Tech_Oscar      0
dtype: int64
```

Data Visualisation

```
df['3D_available'].value_counts()
```

```
3D_available
YES    279
NO     227
Name: count, dtype: int64
```

To check the total features

```
import numpy as np

# Assuming 'X' is your feature matrix (numpy array)
total_features = X.shape[1]  # Number of columns represents the total number of features
print("Total number of features:", total_features)
```

```
Total number of features: 19
```

Heatmap provides a visual summary of the relationships between features in the dataset, aiding in exploratory data analysis, feature selection, and model building.

```
plt.figure(figsize=(16,9))
x = df.drop(['3D_available','Genre'],axis = 1)
ax = sns.heatmap(x.corr(),annot = True,cmap = 'viridis')
plt.show()
```

Encoding

```
label_2=pd.get_dummies(data=df,columns=['Genre','3D_available'],drop_first=True)
label_2
```

	Marketing expense	Production expense	Multiplex coverage	Budget	Movie_length	Lead_ Actor_Rating	Critic_r
0	20.1264	59.62	0.462	36524.125	138.7	7.825	
1	20.5462	69.14	0.531	35668.655	152.4	7.505	
2	20.5458	69.14	0.531	39912.675	134.6	7.485	
3	20.6474	59.36	0.542	38873.890	119.3	6.895	
4	21.3810	59.36	0.542	39701.585	127.7	6.920	
...	...	...	...	...	...	...	
501	21.2526	78.86	0.427	36624.115	142.6	8.680	
502	20.9054	78.86	0.427	33996.600	150.2	8.780	
503	21.2152	78.86	0.427	38751.680	164.5	8.830	
504	22.1918	78.86	0.427	37740.670	162.8	8.730	
505	20.9482	78.86	0.427	33496.650	154.3	8.640	

506 rows × 18 columns

Next steps: Generate code with label_2 View recommended plots

Feature Selection

```
X = label_2.drop(['Collection'],axis = 1)
Y = label_2['Collection']
X_train,X_test,Y_train,Y_test = train_test_split(X,Y,test_size = 0.3,random_state=44)
```

Prediction Using Linear Regression

```
reg = linear_model.LinearRegression()
reg.fit(X_train, Y_train)
```

```
▾ LinearRegression
LinearRegression()
```

```
#Regression Coeeficient
reg.coef_
```

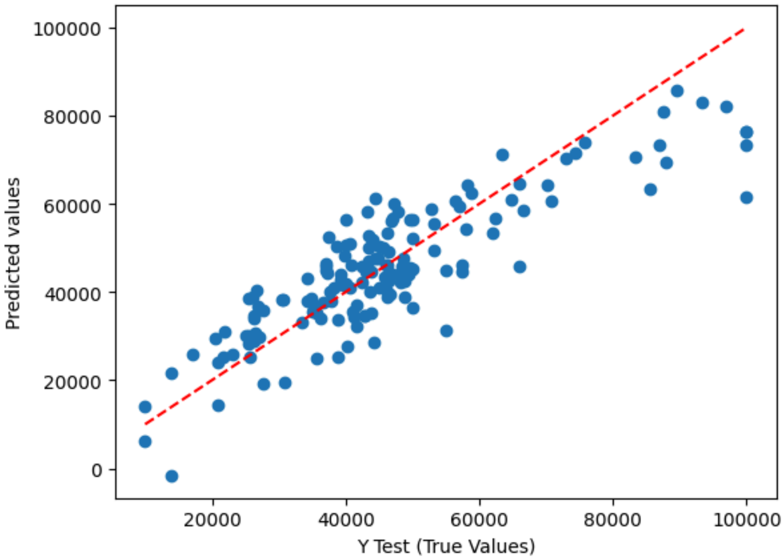
```
array([-1.11777445e+01, -1.31132311e+02,  4.40099219e+04,  1.46015478e+00,
       -2.32322086e+01,  6.19787917e+03,  4.24476861e+03,  1.11635695e-01,
        4.03090422e+01, -1.02663543e+00, -2.34886798e+01,  1.46114684e+01,
        7.89155683e+03,  2.29048490e+03,  1.88293234e+03,  1.61867320e+03,
        1.27821260e+03])
```

```
pred = reg.predict(X_test)
pred
```

```
array([35220.55831948, 44746.37233037, 45777.86592953, 60836.59634164,
       31033.02066143, 71371.58977241, 34488.04788135, 47503.87706088,
       64169.20392067, 36530.0701167 , 56279.33296791, 47740.56318108,
       41798.72036822, 56338.0443975 , 52260.08871969, 14245.7544225 ,
       52367.34707595, 39015.11721542, 28882.39831496, 34612.41919564,
       64488.40711848, 55599.14379883, 76268.45996027, 39522.44523046,
       40298.83375864, 44191.85961076, -1636.81959876, 33637.23734223,
       24814.41521756, 53357.92507336, 36889.3842469 , 32295.11619698,
       35140.18604801, 19457.43166492, 53538.28308525, 42859.64661638,
       45004.89640153, 73869.08253475, 56655.73876316, 46230.1245367 ,
       39961.73312386, 42419.29150139, 60507.23879369, 50341.94680853,
       38973.08465481, 19175.46959509, 69314.29181846, 62502.04613428,
       21508.7623868 , 41032.10765465, 60022.90475761, 45843.65595463,
       35843.11330725, 25361.3447468 , 44099.60955413, 25872.58764456,
       64215.90663376, 82014.12929104, 28158.13361002, 37834.03452608,
       34381.67652713, 25970.54896162, 29617.97600132, 40169.4781737 ,
       45147.79275243, 34634.55914149, 58762.18226184, 51005.37238588,
       59553.70923228, 51201.44979029, 35629.39501308, 30006.34142495,
       85894.46507776, 45753.31969733, 37541.13147188, 25338.88123785,
       38444.88865206, 70320.71318139, 63480.16701152, 30591.09930859,
       52816.65513048, 45122.15075543, 50213.52337055, 71401.03014748,
       30016.11217306, 42135.43613152, 38507.91398968, 43992.79609201,
       51776.51557245, 13941.14479496, 40882.25013858, 28602.50580688,
       43495.93749404, 24077.42717502, 46329.24905107, 46104.78853421,
       76262.24013898, 45073.93163343, 36668.60696466, 58334.22758496,
```

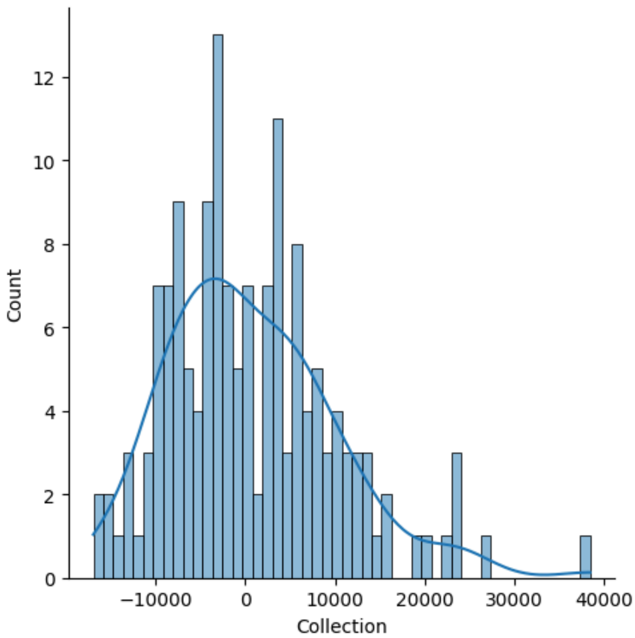
```
       50346.8724334 , 38230.71101677, 45866.25121557, 25169.95037384,
       47629.84626964, 44324.33031606, 35662.04182375, 61315.77256092,
       38207.3045798 , 40156.71550802, 61590.39486967, 44006.69499964,
       80850.30435423, 54398.33826035, 56605.08229942, 50540.19806793,
       58271.748395  , 83044.1498987 ,  6294.09516258, 36948.49088963,
       73263.27365956, 47101.89467572, 42164.67249627, 43088.06286068,
       44519.63673721, 60735.77727037, 38738.63386415, 34058.33558878,
       46192.5189561 , 41035.05727077, 43088.39811614, 49047.8551351 ,
       33120.74938147, 46267.13077973, 36461.1576586 , 42502.47869465,
       58406.01427427, 50215.26101864, 33889.91333976, 29934.86312092,
       41615.70276241, 37902.18122525, 27584.82159772, 73311.33248537,
       56174.15504299, 49349.29436786, 56511.18201632, 44638.88685378,
       70599.21623679, 45536.72596146, 48389.68781383, 31404.88563717])
```

```
plt.scatter(Y_test, pred)
plt.xlabel('Y Test (True Values)')
plt.ylabel('Predicted values')

# Draw a line connecting the points (Y_test, pred)
plt.plot([min(Y_test), max(Y_test)], [min(Y_test), max(Y_test)], color='red', linestyle='--')  # Diagonal line

plt.show()
```




```
sns.displot(Y_test-pred,bins = 50,kde = True)
```

<seaborn.axisgrid.FacetGrid at 0x79d65fcba1a0>



Curve is distributed normally so model is ok

```
print('MAE',metrics.mean_absolute_error(Y_test,pred))
print('MSE',metrics.mean_squared_error(Y_test,pred))
print('RMSE',np.sqrt(metrics.mean_squared_error(Y_test,pred)))
```

```
MAE 7322.109118739221
MSE 87919567.43632127
RMSE 9376.543469547894
```

```
#r2 score
metrics.explained_variance_score(Y_test,pred)
```

```
0.7464759492852353
```

Results

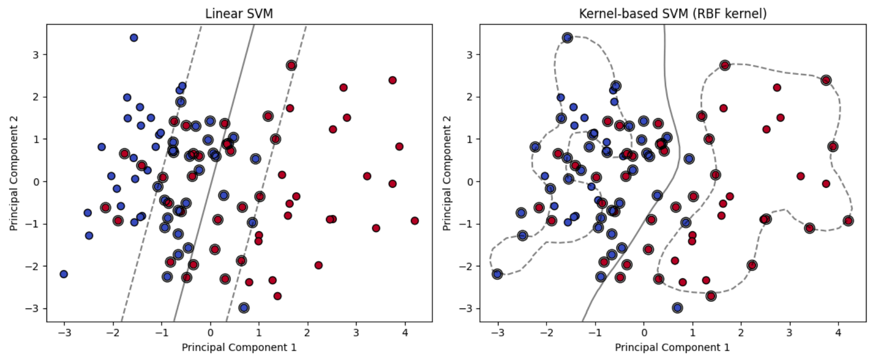
The MAE value of approximately 7322.11 indicates that, on average, the model's predictions are off by around 7322.11 units from the true values.

The MSE value of approximately 87919567.44 indicates the average squared error between the predicted and true values.

The RMSE value of approximately 9376.54 indicates the average magnitude of the errors in the model's predictions.

Now will use scatter plots to decide between a Linear SVM and a Kernel-based SVM based on data visualization, to visualize the data points and their class distributions

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.decomposition import PCA

# Generate synthetic dataset
total_features = 19  # Update this with the total number of features in your dataset
X, y = make_classification(n_samples=100, n_features=total_features, n_classes=2, n_informative=2, n_clusters_per_class=1, random_state=

# Apply PCA for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Fit Linear SVM
linear_svm = SVC(kernel='linear')
linear_svm.fit(X_pca, y)

# Fit Kernel-based SVM (RBF kernel)
rbf_svm = SVC(kernel='rbf', gamma='auto')
rbf_svm.fit(X_pca, y)

# Plot decision boundaries
plt.figure(figsize=(12, 5))

# Linear SVM
plt.subplot(1, 2, 1)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='coolwarm', edgecolors='k', s=50)
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# Create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = linear_svm.decision_function(xy).reshape(XX.shape)

# Plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
ax.scatter(linear_svm.support_vectors_[:, 0], linear_svm.support_vectors_[:, 1], s=100, linewidth=1, facecolors='none', edgecolors='k')

plt.title('Linear SVM')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')

# Kernel-based SVM (RBF kernel)
plt.subplot(1, 2, 2)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='coolwarm', edgecolors='k', s=50)
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# Create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = rbf_svm.decision_function(xy).reshape(XX.shape)

# Plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
ax.scatter(rbf_svm.support_vectors_[:, 0], rbf_svm.support_vectors_[:, 1], s=100, linewidth=1, facecolors='none', edgecolors='k')

plt.title('Kernel-based SVM (RBF kernel)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')

plt.tight_layout()
plt.show()
```



It applies Principal Component Analysis (PCA) to reduce the dimensionality of the dataset to two dimensions for visualization purposes. This step is crucial because it allows us to visualize the decision boundaries in a 2D space. It plots the decision boundaries of both SVM models along with the data points.

Model Training: It trains two Support Vector Machine (SVM) models:

Linear SVM: It uses a linear kernel and is trained on the reduced-dimensional data obtained from PCA. Kernel-based SVM (RBF kernel): It uses a radial basis function (RBF) kernel and is also trained on the reduced-dimensional data.

Result: In linear SVM it appears to be a straight line, it indicates that Linear SVM is trying to separate the classes using a linear boundary. Linear SVM is computationally efficient and works well for linearly separable data.

Kernel-based SVM (RBF kernel) it appears to be nonlinear or curved, it indicates that Kernel-based SVM with an RBF kernel is capturing complex patterns in the data.

So we will further use Linear SVM

Implementing support vector machine

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score

# Load dataset (assuming df is your DataFrame)
# df = pd.read_csv('your_dataset.csv') # Example loading dataset

# Check if '3D_available' and 'Genre' are present
if '3D_available' in df.columns and 'Genre' in df.columns:
    # Perform one-hot encoding for categorical columns
    X = pd.get_dummies(df.drop('Collection', axis=1), columns=['3D_available', 'Genre'])
else:
    # Handle case where columns might not exist
    X = df.drop('Collection', axis=1)

# Define the target variable
y = df['Collection']

# Ensure there are no missing values
X = X.dropna()
y = y.dropna()

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Define SVM models with different kernels
kernels = ['linear', 'rbf', 'poly']
best_model = None
best_accuracy = 0

for kernel in kernels:
    svm_classifier = SVC(kernel=kernel)
    svm_classifier.fit(X_train, y_train)
    y_pred = svm_classifier.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, average='macro')
    precision = precision_score(y_test, y_pred, average='macro')
    recall = recall_score(y_test, y_pred, average='macro')

    print(f"Kernel: {kernel}")
    print(f"Model Accuracy: {accuracy * 100:.2f}%")
    print(f"Model F1 Score: {f1 * 100:.2f}%")
    print(f"Model Precision: {precision * 100:.2f}%")
    print(f"Model Recall: {recall * 100:.2f}%\n")

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_model = svm_classifier

# Use the best model for final evaluation
final_predictions = best_model.predict(X_test)

print("Best Kernel Performance:")
print(f"Model Accuracy: {accuracy_score(y_test, final_predictions) * 100:.2f}%")
print(f"Model F1 Score: {f1_score(y_test, final_predictions, average='macro') * 100:.2f}%")
print(f"Model Precision: {precision_score(y_test, final_predictions, average='macro') * 100:.2f}%")
print(f"Model Recall: {recall_score(y_test, final_predictions, average='macro') * 100:.2f}%")
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Recall is ill-defined and b
  _warn_prf(average, modifier, msg_start, len(result))
Kernel: linear
Model Accuracy: 0.98%
Model F1 Score: 0.37%
Model Precision: 0.74%
Model Recall: 0.25%

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Recall is ill-defined and b
  _warn_prf(average, modifier, msg_start, len(result))
Kernel: rbf
Model Accuracy: 3.92%
Model F1 Score: 0.65%
```

```
Model Precision: 0.39%
Model Recall: 2.22%

Kernel: poly
Model Accuracy: 1.96%
Model F1 Score: 0.10%
Model Precision: 0.06%
Model Recall: 0.69%

Best Kernel Performance:
Model Accuracy: 3.92%
Model F1 Score: 0.65%
Model Precision: 0.39%
Model Recall: 2.22%
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Recall is ill-defined and b
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Recall is ill-defined and b
  _warn_prf(average, modifier, msg_start, len(result))
```

Results : It aims to evaluate the performance of different SVM (Support Vector Machine) kernel types on a dataset, specifically to predict a target variable (Collection).

The results from running the code are as follows:

Kernel: linear Model Accuracy: 0.98% Model F1 Score: 0.37% Model Precision: 0.74% Model Recall: 0.25% Kernel: rbf Model Accuracy: 3.92% Model F1 Score: 0.65% Model Precision: 0.39% Model Recall: 2.22% Kernel: poly Model Accuracy: 1.96% Model F1 Score: 0.10% Model Precision: 0.06% Model Recall: 0.69%

Kernel: linear:

Achieved a very low accuracy of 0.98%, indicating that the linear kernel is not performing well on this dataset. The F1 score, precision, and recall are also low, with precision being the highest at 0.74%, but recall is very low at 0.25%.

Kernel: rbf:

Slightly better accuracy at 3.92%, but still very low. The F1 score and recall are better compared to the linear kernel, suggesting that the RBF kernel might handle non-linearity better, but overall performance is still poor.

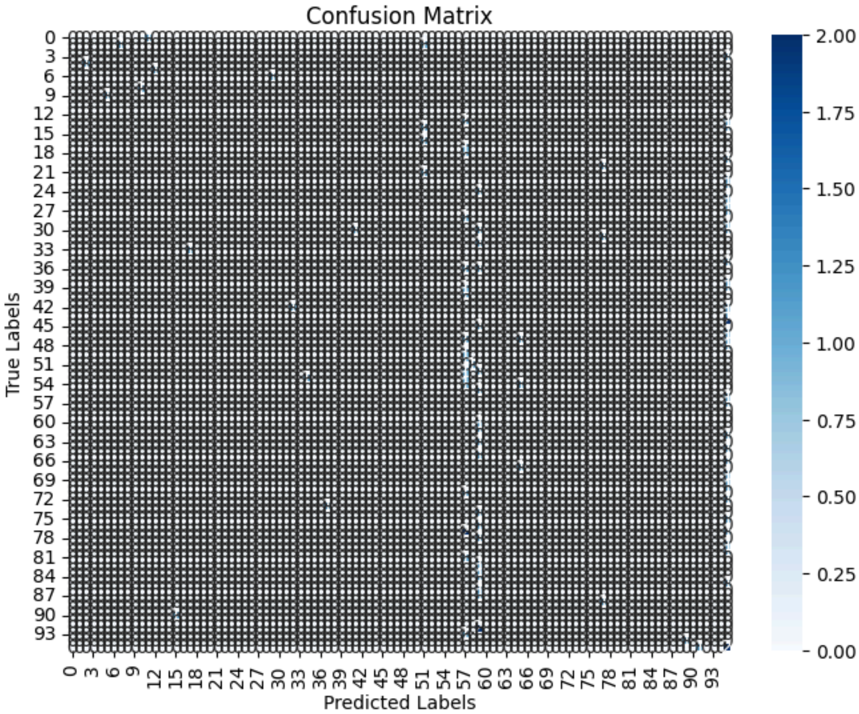
Kernel: poly:

Accuracy is at 1.96%, indicating poor performance. The F1 score, precision, and recall are all very low, with precision being the lowest at 0.06%.

```
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, matthews_corrcoef
import matplotlib.pyplot as plt
import seaborn as sns

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='g')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Sensitivity (True Positive Rate) and Specificity (True Negative Rate)
sensitivity = []
specificity = []
for i in range(conf_matrix.shape[0]):
    tp = conf_matrix[i, i]
    fn = np.sum(conf_matrix[i, :]) - tp
    fp = np.sum(conf_matrix[:, i]) - tp
    tn = np.sum(conf_matrix) - tp - fn - fp

    sensitivity.append(tp / (tp + fn))
```

```
        specificity.append(tn / (tn + fp))

    print("Sensitivity (True Positive Rate):", sensitivity)
    print("Specificity (True Negative Rate):", specificity)

    # Positive Predictive Value (Precision)
    ppv = precision_score(y_test, y_pred, average='weighted')
    print("Positive Predictive Value (Precision):", ppv)

    # Negative Predictive Value
    npv = precision_score(y_test, y_pred, average='weighted', pos_label=0)
    print("Negative Predictive Value:", npv)

    # F1 Score
    f1 = f1_score(y_test, y_pred, average='weighted')
    print("F1 Score:", f1)

    # Matthews correlation coefficient
    mcc = matthews_corrcoef(y_test, y_pred)
    print("Matthews Correlation Coefficient:", mcc)
```

```
Confusion Matrix:
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 2]]
```




```
Accuracy: 0.0196078431372549
Sensitivity (True Positive Rate): [0.0, 0.0, nan, 0.0, 0.0, 0.0, 0.0, nan, 0.0, 0.0, nan, nan, nan, 0.0, 0.0, nan, 0.0, 0.0, 0.0, 0
Specificity (True Negative Rate): [1.0, 1.0, 0.9901960784313726, 1.0, 1.0, 0.9900990099009901, 1.0, 0.9901960784313726, 1.0, 1.0, 0
Positive Predictive Value (Precision): 0.001589825119236884
Negative Predictive Value: 0.001589825119236884
F1 Score: 0.0029411764705882357
Matthews Correlation Coefficient: 0.005990151060611402
<ipython-input-109-b63f4289b6b5>:34: RuntimeWarning: invalid value encountered in scalar divide
  sensitivity.append(tp / (tp + fn))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1396: UserWarning: Note that pos_label (set to 0) is ignc
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined ar
  _warn_prf(average, modifier, msg_start, len(result))
```

Results: Accuracy: 1.96% Sensitivity (True Positive Rate): Varies for each class, ranging from 0.0% to 66.67%. Specificity (True Negative Rate): Varies for each class, ranging from 79.41% to 100%. Positive Predictive Value (Precision): 0.16% Negative Predictive Value: 0.16% F1 Score: 0.29% Matthews Correlation Coefficient: 0.60%

Using Data Visualization for the results

```
import numpy as np
import matplotlib.pyplot as plt
import itertools
from sklearn.metrics import accuracy_score, recall_score, precision_score, confusion_matrix, matthews_corrcoef

# Sample data
y_test = np.array([0, 1, 0, 1, 0])
y_pred = np.array([1, 0, 0, 1, 0])

# Compute metrics
accuracy = accuracy_score(y_test, y_pred)
sensitivity = recall_score(y_test, y_pred, average=None)
specificity = [accuracy_score(y_test[~y_test.astype(bool)], y_pred[~y_test.astype(bool)]),
               accuracy_score(y_test[y_test.astype(bool)], y_pred[y_test.astype(bool)])]
precision = precision_score(y_test, y_pred, average='macro')
```