Code:

import warnings

warnings.filterwarnings("ignore")

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt # corrected typo here

import seaborn as sns

%matplotlib inline

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

import pickle

from sklearn.metrics import mean\_squared\_error

from sklearn.ensemble import AdaBoostRegressor

from sklearn.impute import SimpleImputer

data = pd.read\_csv('box\_office.csv', encoding='latin-1')

data

data.head(10)

data.columns

# Display shape of dataset

print("Shape of dataset:", data.shape)

# Check for null values

print("Null values in dataset:")

print(data.isnull().sum())

import pandas as pd

# Assuming your dataset is stored in a DataFrame called 'data'

# Specify the column name for which you want to calculate the mean of null values

column\_name = 'director\_ig\_followers'

column\_name2 = 'actor\_2\_ig\_follow'

column\_name3 = 'box\_office'

column\_name4= 'actor\_1\_ig\_follow'

column\_name5='buget'

# Calculate the mean for the specified column where values are null

mean\_value = data[column\_name].mean()

mean\_value2 = data[column\_name2].mean()

mean\_value3 = data[column\_name3].mean()

mean\_value4 = data[column\_name4].mean()

mean\_value5 = data[column\_name5].mean()

# Fill null values in the specified column with the calculated mean

data[column\_name].fillna(mean\_value, inplace=True)

data[column\_name2].fillna(mean\_value2, inplace=True)

data[column\_name3].fillna(mean\_value3, inplace=True)

data[column\_name4].fillna(mean\_value4, inplace=True)

data[column\_name5].fillna(mean\_value5, inplace=True)

# Verify if nulls are filled

print("Null values after filling with mean:")

print(data.isnull().sum())

data.to\_csv('box\_office.csv', index=False)

# data.dropna(axis=0,inplace=True)

# data

data=data.reset\_index(drop=True)

data.to\_csv("test\_data.csv",index=False)

data.shape

data.info()

data['genres'] = data['genres'].astype('category')

data['director\_name'] = data['director\_name'].astype('category')

data['actor\_1\_name'] = data['actor\_1\_name'].astype('category')

data['actor\_2\_name'] = data['actor\_2\_name'].astype('category')

data

# Select columns with 'category' data type and apply .cat.codes

cat\_columns = data.select\_dtypes(['category']).columns

data[cat\_columns] = data[cat\_columns].apply(lambda x: x.cat.codes)

data['label'] = data['imdb\_socre'] # Corrected column name

data['label'] = data['label'].astype(int)

data['title\_year'] = data['title\_year'].astype(int)

data.head()

with plt.style.context("ggplot"):

plt.figure(figsize=(18, 8))

sns.countplot(x='title\_year', data=data) # Specify 'x' explicitly

plt.xticks(rotation=90)

plt.show()

X = data.drop(['movie\_title','title\_year','plot\_keywords','label','box\_office' ],axis=1)

y = data['box\_office']

feature\_list = X.columns

for column in X.columns:

unique\_values = X[column].unique()

print(f"Column '{column}' unique values: {unique\_values}")

print(X.columns)

import re

# Define a function to convert monetary strings to numeric values

def monetary\_to\_numeric(value):

if isinstance(value, str):

value = re.sub(r'[^\d.]', '', value) # Remove non-numeric characters

if value.isdigit():

return float(value)

elif 'million' in value:

return float(value.replace(' million', '')) \* 1e6

elif 'billion' in value:

return float(value.replace(' billion', '')) \* 1e9

return value

# Specify the columns containing monetary values

#monetary\_columns = ['buget', 'gross', 'box\_office']

monetary\_columns = ['buget', 'gross']

# Apply the function to the relevant columns

for column in monetary\_columns:

X[column] = X[column].apply(monetary\_to\_numeric)

print(X.columns)

print(X['duration'].unique())

# Replace commas in 'duration' column with an empty string

X['duration'] = X['duration'].replace(',', '', regex=True)

# Convert the 'duration' column to numeric, coercing errors to NaN

X['duration'] = pd.to\_numeric(X['duration'], errors='coerce')

# Check for any NaN values

nan\_values = X['duration'].isna().sum()

print("Number of NaN values in 'duration' column:", nan\_values)

unique\_durations = X['duration'].unique()

print("Unique values in 'duration' column:")

print(unique\_durations)

# Iterate over each value in the 'duration' column

for duration\_value in X['duration']:

try:

# Attempt to convert the value to an integer

int(duration\_value)

except ValueError:

# Print the value if it cannot be converted to an integer

print("Non-numeric value found:", duration\_value)

print("done")

X['duration'] = X['duration'].astype(str).str.replace(r'\D', '', regex=True)

# Convert the 'duration' column to numeric, coercing errors to NaN

X['duration'] = pd.to\_numeric(X['duration'], errors='coerce')

# Check for any NaN values

nan\_values = X['duration'].isna().sum()

print("Number of NaN values in 'duration' column:", nan\_values)

# Drop rows with NaN values in the 'duration' column

X = X.dropna(subset=['duration'])

# Check if there are still non-numeric values in the 'duration' column

if X['duration'].dtype != 'O':

# Check for negative or zero values in 'duration'

negative\_values = (X['duration'] <= 0).sum()

if negative\_values > 0:

print("Error: 'duration' column contains negative or zero values after cleaning.")

# else:

# # Now, you can proceed with scaling

# scaler\_feature = StandardScaler()

# X\_scaled = scaler\_feature.fit\_transform(X)

else:

print("Error: 'duration' column still contains non-numeric values after cleaning.")

# Remove commas from all numeric columns

numeric\_columns = ['facenumber\_in\_poster','director\_ig\_followers', 'actor\_1\_ig\_follow', 'actor\_2\_ig\_follow', 'duration', 'buget', 'gross', 'num\_critic\_for\_review', 'num\_user\_for\_review', 'imdb\_socre','num\_awards\_won']

X[numeric\_columns] = X[numeric\_columns].replace(',', '', regex=True)

# Convert the columns to numeric

X[numeric\_columns] = X[numeric\_columns].apply(pd.to\_numeric)

# # Now, you can proceed with scaling

# scaler\_feature = StandardScaler()

# X\_scaled = scaler\_feature.fit\_transform(X)

# # Save the preprocessed features

# with open(file="module/X\_scaled.pkl", mode="wb") as file:

# pickle.dump(obj=X\_scaled, file=file)

X\_train, X\_test, y\_train , y\_test = train\_test\_split(X,y, test\_size = 0.2, random\_state=42)

print(X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape)

# y\_train.fillna(y\_train.mean(), inplace=True)

# y\_test.fillna(y\_test.mean(), inplace=True)

# Check for missing values in y

print("Missing values in y\_train:", y\_train.isnull().sum())

print("Missing values in y\_test:", y\_test.isnull().sum())

# Verify indexing

print("Indices of y\_train:", y\_train.index)

print("Indices of y\_test:", y\_test.index)

print("y\_train:", y\_train)

print("y\_test:", y\_test)

y\_test.to\_csv('y\_test.csv',index=False)

from sklearn.ensemble import GradientBoostingRegressor

# Instantiate Gradient Boosting Regressor model

gb\_model = GradientBoostingRegressor()

# Fit the model on training data

gb\_model.fit(X\_train, y\_train)

# Make predictions on test data

y\_pred\_gb = gb\_model.predict(X\_test)

# Display the first 50 predicted and actual values

print("Actual\t\tPredicted")

for actual, predicted in zip(y\_test[:50], y\_pred\_gb[:50]):

print(f"{actual}\t{predicted}")

predictions\_df = pd.DataFrame({'Actual': y\_test[:50], 'Predicted': y\_pred\_gb[:50]})

# Save the DataFrame to a CSV file

predictions\_df.to\_csv('predictions.csv', index=False)

print(y\_test)

# Calculate Mean Squared Error

gb\_mse = mean\_squared\_error(y\_test, y\_pred\_gb)

print('Gradient Boosting Regressor Mean Squared Error:', gb\_mse)

gb\_r2\_score = gb\_model.score(X\_test, y\_test)

print('Gradient Boosting Regressor R^2 Score:', gb\_r2\_score)

# Plot graph of Actual and Predicted values for Gradient Boosting Regressor model

plt.figure(figsize=(18, 8))

plt.plot(range(len(y\_test[:50])), y\_test[:50], label='Actual', marker='o')

plt.plot(range(len(y\_pred\_gb[:50])), y\_pred\_gb[:50], label='Predicted', marker='x')

plt.title("Actual and Predicted Result Comparison for Gradient Boosting Regressor model")

plt.xlabel("Number of records")

plt.legend()

plt.grid(True)

plt.show()

X\_test

y\_test

# # Compute feature importances

# importances = gb\_model.feature\_importances\_

# # Plot feature importances

# with plt.style.context("fivethirtyeight"):

# plt.figure(figsize=(18,8))

# x\_values = list(range(len(importances)))

# plt.bar(x\_values, importances, orientation='vertical', color="crimson")

# plt.xticks(x\_values, feature\_list)

# plt.xlabel('Features')

# plt.ylabel('Importance')

# plt.title('Feature Importance')

# plt.xticks(rotation=90)

# plt.show()

import pickle

model = {'model': gb\_model}

with open('module/model1.pkl', 'wb' ) as f:

pickle.dump ( model , f )

frontend code:

from flask import Flask, render\_template, request

import pickle

import pandas as pd

from sklearn.impute import SimpleImputer

app = Flask(\_\_name\_\_)

# Load the trained Gradient Boosting Regressor model

with open(r"C:\\Users\\divya mulchandani\\OneDrive\\Documents\\AI-assignment\\box office\\box office\\module\\model1.pkl", "rb") as file:

    model = pickle.load(file)

# Load the dataset to fetch actual box office values

data = pd.read\_csv(r'C:\\Users\\divya mulchandani\\OneDrive\\Documents\\AI-assignment\\box office\\box office\\box\_office.csv', encoding='latin-1')

data.set\_index('movie\_title', inplace=True)

# Define function to preprocess input data

def preprocess\_input(movie\_name, data):

    # Check if the provided movie name exists in the dataset

    if movie\_name in data.index:

        # Extract features corresponding to the movie name from the dataset

        features = data.loc[movie\_name, ['facenumber\_in\_poster', 'director\_ig\_followers', 'actor\_1\_ig\_follow', 'actor\_2\_ig\_follow', 'duration', 'buget', 'gross', 'num\_critic\_for\_review', 'num\_user\_for\_review', 'imdb\_socre', 'num\_awards\_won']].tolist()

        # Adjust the number of features to match the model's expectations (if needed)

        while len(features) < 16:

            features.append(0)  # Add zeros to match the expected number of features

        return features

    else:

        # If the movie name is not found in the dataset, return None

        return None

@app.route('/')

def home():

    return render\_template('index.html')

@app.route('/predict', methods=['POST'])

def predict():

    if request.method == 'POST':

        movie\_name = request.form['movie\_name']

        # Preprocess input data

        input\_data = preprocess\_input(movie\_name, data)

        if input\_data is None:

            return render\_template('error.html', message=f"Movie '{movie\_name}' not found in the dataset.")

        # Handle missing values by imputing

        imputer = SimpleImputer(strategy='mean')

        input\_data\_imputed = imputer.fit\_transform([input\_data])

        # Make prediction

        predicted\_box\_office = model.predict(input\_data\_imputed)[0]

        # Fetch actual box office value for the movie

        actual\_box\_office = data.loc[movie\_name, 'box\_office']

        return render\_template('result.html', movie\_name=movie\_name, predicted=predicted\_box\_office, actual=actual\_box\_office)

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)





