**Netflix popular movies dataset**

1. **data cleaning and preprocessing**

involves correcting or removing inaccurate, corrupted, improperly formatted, duplicate, or incomplete data to ensure reliability. it is essential when merging multiple data sources, as errors can lead to misleading results, even if the output appears accurate

**actions I did to clean my dataset**

df\_og = pd.read\_csv("ai3.csv")  
df = df\_og.copy()  
  
print(df.info())  
print(df.head())  
  
missing\_values = df.isnull().sum()  
missing\_percentage = (missing\_values / len(df)) \* 100  
print(pd.DataFrame({"Missing Values": missing\_values, "Percentage": missing\_percentage}))

- reads the contents of a CSV file named ai3.csv into a data frame using Pandas

- creates a copy of the original data frame df\_og and stores it in a new variable df

- displays summary information about df

details shown: number of entries (rows) = 9958

number of columns = 9

column names and their data types

number of non-null values per column (helps spot missing data)

memory usage

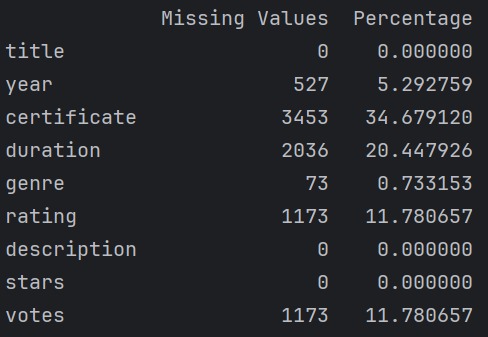
- displays the first 5 rows of the df

- missing\_values = df.isnull().sum() : finds out how many missing values exist in each column

- missing\_percentage = (missing\_values / len(df)) \* 100 : calculates the percentage of missing values for each column

- displays a readable summary of how much data is missing in each column, both as raw counts and percentages => helps in making informed decisions about cleaning like whether to drop, fill, or ignore missing data

**the results before cleaning**



**detailed actions per column**

1. **year** (5.29% missing)

year is useful for trend analysis, and prediction

=> fill with mode

def extract\_year(year\_str):  
 if pd.isna(year\_str):   
 return None  
 match = re.search(r'\d{4}', str(year\_str))   
 if match:  
 return int(match.group(0))  
 return None   
df['year'] = df['year'].apply(extract\_year)  
df['year'] = df['year'].fillna(df['year'].mode()[0])

df['year'] = df['year'].fillna(df['year'].mode()[0])

- checks if the input is NaN (missing value)

- if yes -> returns None, to be handled later

- converts input to string

- uses regular expression \d{4} to search for a 4-digit number

- if a 4-digit year is found -> return it as an integer

- if no match -> return None (considered missing later)

- applies the extract\_year function to each row in the year column

- now df['year'] only has integers or NaN

- calculates the mode = most frequent year in the column

=> fills all missing years (NaN) with that value

1. **certificate** (34.68% missing)

high missing, but useful for classification tasks

since used for modeling

=> fill with ‘Unknown’

df['certificate'] = df['certificate'].fillna('Unknown')

1. **duration** (20.44% missing)

useful for modeling

=> fill with mean duration per genre

df['duration'] = df['duration'].str.replace(' min', '').astype(float)  
genre\_duration\_mean = df.groupby('genre')['duration'].mean()  
def fill\_missing\_duration(row):  
 if pd.isnull(row['duration']):  
 genre\_mean = genre\_duration\_mean.get(row['genre'], df['duration'].mean())  
 return genre\_mean if not pd.isnull(genre\_mean) else df['duration'].mean()  
 return row['duration']  
df['duration'] = df.apply(fill\_missing\_duration, axis=1)

- removes ‘min’ from values

- converts the result to float

- groups the dataset by genre

- calculates average duration for each genre

- fill missing duration with genre mean

- if genre is also missing or no mean available, falls back to overall mean

- uses .apply(..., axis=1) to apply this logic row-by-row

1. **genre** (0.73% missing)

low missing

=> fill with most common genre

df['genre'] = df['genre'].fillna(df['genre'].mode()[0])

1. **rating** (11.78% missing)

key column for predictions

=> fill with mean

df['rating'] = df['rating'].fillna(df['rating'].mean())

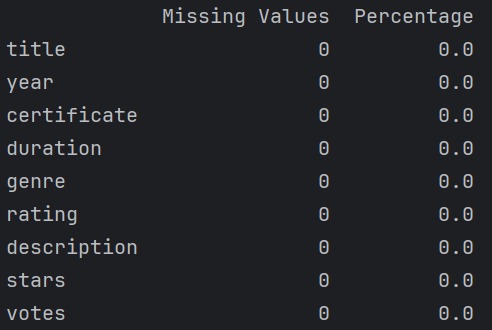
1. **votes** (11.78% missing)

useful indicator of popularity

=> convert to numeric (remove commas), then fill missing with 0 (no votes) or median if 0 skews results

df['votes'] = df['votes'].str.replace(',', '').astype(float)  
df['votes'] = df['votes'].fillna(0)

**the results after cleaning**



1. **data modeling**

supervised learning is when a machine is trained using labeled data => the data already includes the correct answers. the algorithm learns from this data and then uses it to make accurate predictions on new, unseen data

the choice of models depends on the nature of the dataset, the type of target variable, and the relationships between features and the target

**models I chose to perform data modeling**

1. **linear regression**

models the relationship between the independent variables (features) and the dependent variable (target) by fitting a linear equation to the observed data

**use case** : predict the rating of a movie based on features like year, duration, and votes

from sklearn.linear\_model import LinearRegression  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import mean\_squared\_error, r2\_score

- to create and train the regression model

- to split data into training and testing sets

- to evaluate model accuracy

X = df[['year', 'duration', 'votes']]   
y = df['rating']

- define features (X) and target (y)  
- X = input features (year, duration, votes) used to predict something (independent variables)

- y = what we want to predict (rating) (dependent variable)

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

- split data into training and test sets

- 80% of the data for training the model and 20% for testing the model on unseen data

- random\_state=42 : ensures reproducibility => it locks the randomness to a fixed pattern => every time we run the code, we get the same split of training and testing data

model = LinearRegression()  
model.fit(X\_train, y\_train)

- creates a linear regression model

- .fit() -> trains the model on my **training data**

**- the model learns relationships between year, duration, votes and the rating**

y\_pred = model.predict(X\_test)

- the model uses X\_test (unseen data) to predict ratings

- result: y\_pred = predicted ratings for movies in the test set

print("Mean Squared Error:", mean\_squared\_error(y\_test, y\_pred))  
print("R^2 Score:", r2\_score(y\_test, y\_pred))

- evaluate model performance

- Mean Squared Error (MSE) : average of the squared errors between real and predicted ratings (lower = better)

- R² Score : measures how well the model explains variance in the data (1.0 = perfect, 0 = poor)

**the results of the model’s performance**

Mean Squared Error: 1.2185589827444225

R^2 Score: 0.05904706361050771

**=> my model is underperforming (features like year, duration, votes alone don’t explain rating well)**

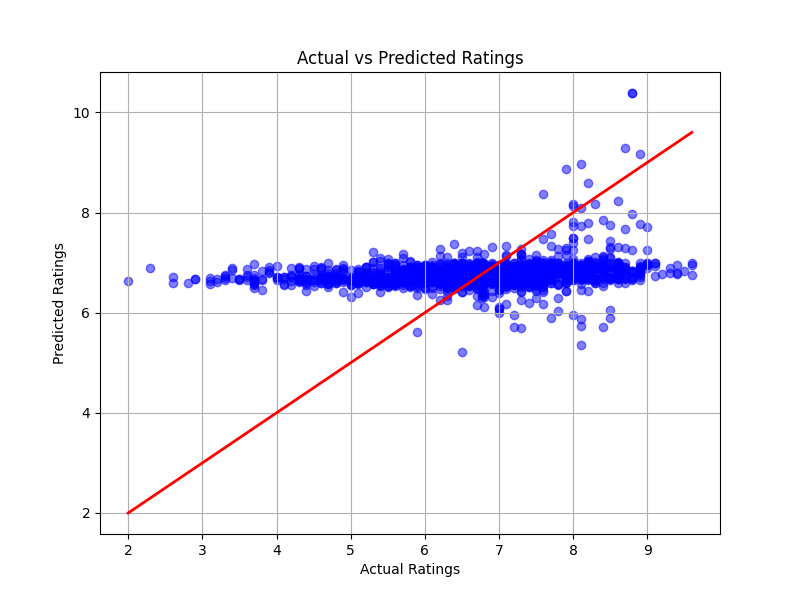
**possible reasons for poor performance**

1. **insufficient features**: the current features (year, duration, votes) may not be strong predictors of the target (rating)
2. **non-linear relationships**: the relationship between the features and the target may be non-linear, but I’m using a linear model
3. **feature scaling**: if the features are on different scales, it can affect the model's performance
4. **outliers**: can skew the model's predictions

**why is my model weak?**

- linear regression assumes all features are scaled and evenly distributed

- my data has extreme values (votes and duration), pulling the regression line in the wrong directions



- red line = ideal performance (line of perfect predictions)

- if my model were perfect, all blue dots would lie exactly on this line

- blue dots = my model’s predictions

- each dot is one movie’s rating => with actual rating on the x-axis, predicted rating on the y-axis

- ideally, they should be clustered around the red line

**conclusion = model underfitting**

- model lacks variation

- lacks accuracy

- fails to adjust predictions based on input variations

**experiments to improve performance**

**- feature scaling**

lower\_bound\_duration = df['duration'].quantile(0.01)  
upper\_bound\_duration = df['duration'].quantile(0.99)  
df['duration'] = df['duration'].clip(lower=lower\_bound\_duration, upper=upper\_bound\_duration)  
  
lower\_bound\_votes = df['votes'].quantile(0.01)  
upper\_bound\_votes = df['votes'].quantile(0.99)  
df['votes'] = df['votes'].clip(lower=lower\_bound\_votes, upper=upper\_bound\_votes)

- handles outliers in ‘duration’ and ‘votes’ columns

- quantile(0.01) and quantile(0.99) are used to determine the lower and upper bounds of the duration and votes columns => represent the 1st and 99th percentiles

- clip function limits the values of duration and votes within the specified bounds (any value below the 1st percentile is set to the 1st percentile value, and any value above the 99th percentile is set to the 99th percentile value)

- helps in handling outliers by capping extreme values

df['votes\_log'] = np.log1p(df['votes'])  
df.drop('votes', axis=1, inplace=True)  
df.rename(columns={'votes\_log': 'votes'}, inplace=True)

- log transformation on ‘votes’ column

- the np.log1p() function is used to apply a logarithmic transformation to the votes column

- log1p is equivalent to log(1 + x), which ensures that values of votes that are 0 or close to 0 don’t cause mathematical issues (as the logarithm of 0 is undefined)

- the original votes column is dropped, and the new votes\_log column is renamed back to votes for consistency

from sklearn.preprocessing import StandardScaler

- used to standardize features by removing the mean and scaling to unit variance

scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)

- standardizes the features

- StandardScaler standardizes the features by scaling them to have a mean of 0 and a standard deviation of 1

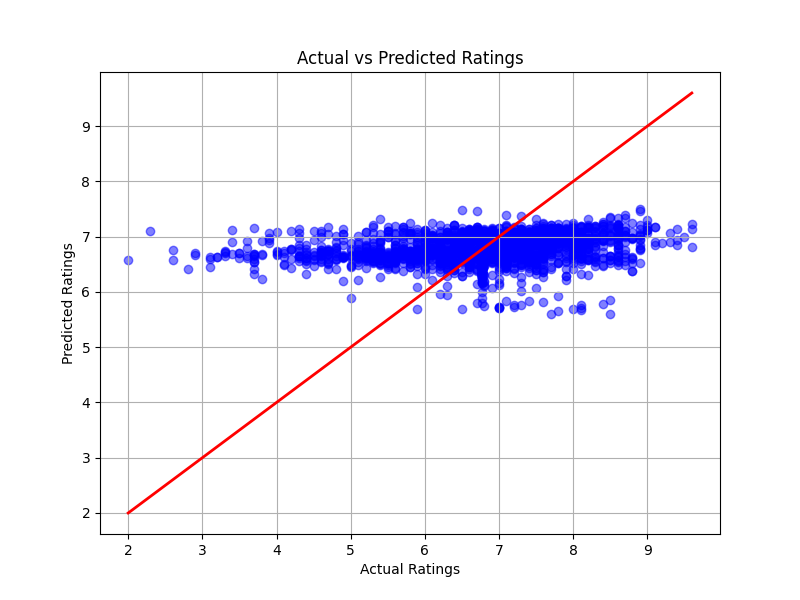
- this is important for models like linear regression, which are sensitive to the scale of the input features

**the results of the model’s performance**

Mean Squared Error: 1.2392807937887926

R^2 Score: 0.043045992488292106

**=> model’s performance is still poor**

****

- outliers don’t exist anymore

- more blue dots have come closer to the red line

**conclusion = model still needs improvement**

**- adding more features**

df['decade'] = (df['year'] // 10) \* 10

- decade feature is created by dividing the year by 10 (using integer division //), then multiplying by 10 to get the decade (if year is 1995, the decade would be 1990)

- this feature groups the movies into decades, which could be useful for capturing trends or patterns related to the era of the movie

df['age'] = 2025 - df['year']

- age feature calculates the age of the movie by subtracting its release year from the current year (2025)

- could help capture the effect of a movie’s age on its rating (older movies might have different ratings than newer ones)

df['lead\_actor'] = df['stars'].str.split(',').str[0]

- the stars column contains a list of actors in a movie. this code splits the string by commas and selects the first actor (assumed to be the lead actor) for the new lead\_actor feature

- this feature may help improve the model by providing more information about the actors associated with higher or lower ratings

X = df[['year', 'duration', 'votes', 'decade', 'age', 'genre', 'certificate', 'lead\_actor']]   
y = df['rating']

- defining features (X) and target (y)

categorical\_features = ['genre', 'certificate', 'lead\_actor']  
numeric\_features = ['year', 'duration', 'votes', 'decade', 'age']  
preprocessor = ColumnTransformer(  
 transformers=[  
 ('num', StandardScaler(), numeric\_features),  
 ('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_features)  
 ])

- preprocessing for categorical and numerical features

- ColumnTransformer : this allows us to apply different preprocessing steps to different types of features:

numeric features (numeric\_features): include year, duration, votes, decade, and age. they are standardized using StandardScaler, which scales the features to have a mean of 0 and a standard deviation of 1. this is important for models like linear regression, which can be sensitive to the scale of the input features

categorical features (categorical\_features): these include genre, certificate, and lead\_actor. they are transformed using OneHotEncoder, which creates binary columns for each category in the categorical features (1 column for each genre, each certificate, and each lead actor). the handle\_unknown='ignore' argument ensures that if a category appears in the test set but not in the training set, it is ignored rather than causing an error

model = Pipeline(steps=[  
 ('preprocessor', preprocessor),  
 ('regressor', LinearRegression())  
])

- creates a pipeline with preprocessing and model

- Pipeline : chains together multiple steps in a single workflow:

the first step ('preprocessor') applies the ColumnTransformer to preprocess the data (standardizing numeric features and one- hot encoding categorical features)

the second step ('regressor') trains a linear regression model on the preprocessed data

- using a pipeline ensures that all transformations (such as scaling and encoding) are consistently applied to both the training and test data, preventing data leakage and making the workflow cleaner

model.fit(X\_train, y\_train)

- trains the model

- the pipeline (which includes preprocessing and the linear regression model) is trained on the training data (X\_train, y\_train)

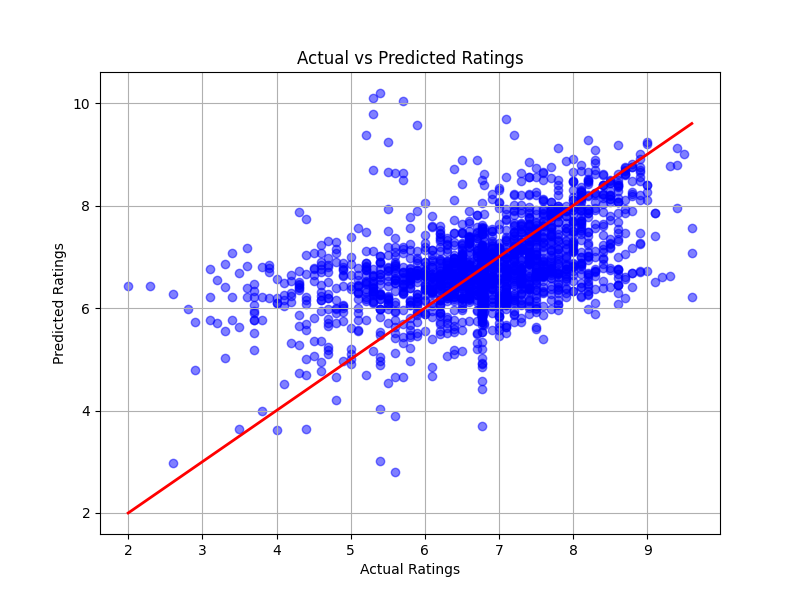
- this step applies both the preprocessing and training steps to the data

**the results of the model’s performance**

Mean Squared Error: 1.062034473256423

R^2 Score: 0.17991293789748808

**=> model still needs improvement**

****

- blue dots are no longer lined horizontally, they are now closer to the red line which shows obvious improvement

**conclusion = while the model’s performance has significantly enhanced, there still is plenty of room for more improvement thus more accuracy**

1. **random forest regressor**

handles non-linearity, outliers, and categorical data very well

it works by creating multiple decision trees during training and then averaging their outputs to produce a more accurate and robust prediction

**use case** : **predict movie ratings** based on features like year, duration, genre, certificate, votes…

preprocessor = ColumnTransformer([  
 ('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_features)  
], remainder='passthrough')

- I used ColumnTransformer to apply different preprocessing steps to different columns:

for categorical features, I used OneHotEncoder : converts text labels into binary vectors (‘Action’ -> [1,0,0], ‘Drama’ -> [0,1,0]...)

handle\_unknown='ignore' : if new unseen categories appear in test data, it won’t crash

remainder='passthrough' : all numeric features are left unchanged => no scaling, no encoding — just passed through

model = Pipeline([  
 ('preprocessor', preprocessor),  
 ('regressor', RandomForestRegressor(n\_estimators=100, random\_state=42))   
])

- build pipeline: preprocessing + model

- a pipeline ties together steps:

first, it preprocesses the data

then, it sends it to the random forest regressor for training or prediction

- benefits: clean code, avoids data leakage, easy cross-validation

- RandomForestRegressor

combines 100 decision trees (n\_estimators=100)

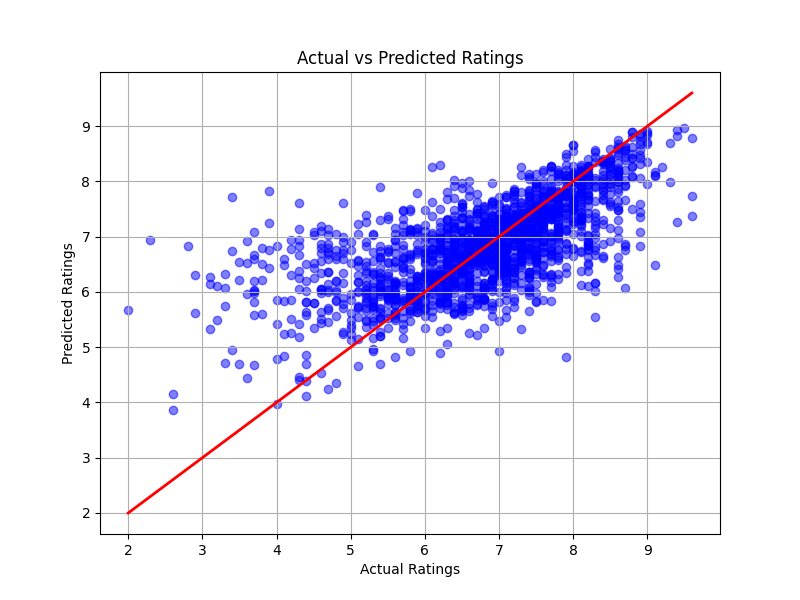
uses randomness in data and features to reduce overfitting and improve generalization

**the results of the model’s performance**

Mean Squared Error: 0.732435411852444

R^2 Score: 0.4344243805531467

**=> random forest regressor is performing much better than the linear regression model**



- the blue dots are still somehow scattered far away from the red line, but this result confirms the improvement random forest regressor did

**conclusion = there is still room for improvement**

**experiments to improve performance**

**- hyperparameter tuning**

one of the most effective ways to improve the performance of a random forest model. by finding the optimal combination of hyperparameters, we can significantly boost the model’s accuracy

param\_grid = {  
 'regressor\_\_n\_estimators': [50, 100, 200],   
 'regressor\_\_max\_depth': [None, 10, 20],   
 'regressor\_\_min\_samples\_split': [2, 5, 10],   
 'regressor\_\_min\_samples\_leaf': [1, 2, 4],   
 'regressor\_\_max\_features': ['sqrt', 'log2']   
}

- define the hyperparameter grid

- creating a dictionary that tells GridSearchCV which hyperparameters to try and which values for each

- the keys are hyperparameter names, and values are lists of possible options

- why regressor\_\_n\_estimators and not just n\_estimators?

because my model is inside a pipeline, we use stepname\_\_paramname format

- what each hyperparameter does:

**n\_estimators** : number of trees in the forest. more = better accuracy but slower

**max\_depth** : max depth of each tree. none = full depth. smaller = less overfitting

**min\_samples\_split** : minimum samples needed to split a node. higher = simpler trees

**min\_samples\_leaf** : minimum samples in a leaf. higher = smoother predictions

**max\_features** : how many features to consider at each split. smaller = more randomness

from sklearn.model\_selection import GridSearchCV  
   
grid\_search = GridSearchCV(  
 estimator=model,   
 param\_grid=param\_grid,   
 cv=5,   
 scoring='r2',   
 n\_jobs=-1

)

- GridSearchCV is a powerful tool in scikit-learn for hyperparameter tuning. it automates the process of searching for the best combination of hyperparameters for a machine learning model by evaluating all possible combinations using cross-validation

- **estimator=model** : the model (pipeline) we want to tune. it includes preprocessing + RF

- **param\_grid** : the dictionary of hyperparameter options defined above

- **cv=5** : splits data into 5 folds -> trains on 4, tests on 1, repeats 5x

- **scoring='r2'** : evaluation metric = R² score (explains variance). higher = better

- **n\_jobs=-1** : uses all CPU cores to run faster (parallel processing)

grid\_search.fit(X\_train, y\_train)

- fit grid search to the data

- for each combination of hyperparameters:

it applies preprocessing to X\_train (OneHotEncoding)

trains the Random Forest with those settings.

runs 5-fold cross-validation to get an average R² score for each combo

- at the end, it remembers the best combination of hyperparameters based on highest average R²

best\_model = grid\_search.best\_estimator\_

- retrieves the pipeline with the best hyperparameter settings found during the search

- we can now use best\_model to predict or evaluate as we would with any model

y\_pred = best\_model.predict(X\_test)

- applies same preprocessing to X\_test

- then predicts ratings using the best-tuned random forest

- results are stored in y\_pred

**the results of the model’s performance**

Best Parameters: {'regressor\_\_max\_depth': None, 'regressor\_\_max\_features': 'sqrt', 'regressor\_\_min\_samples\_leaf': 1, 'regressor\_\_min\_samples\_split': 2, 'regressor\_\_n\_estimators': 200}

Mean Squared Error: 0.7064820257718838

R^2 Score: 0.4544651953085824

**=> there appears to be clear improvement**

**best hyperparameters found**

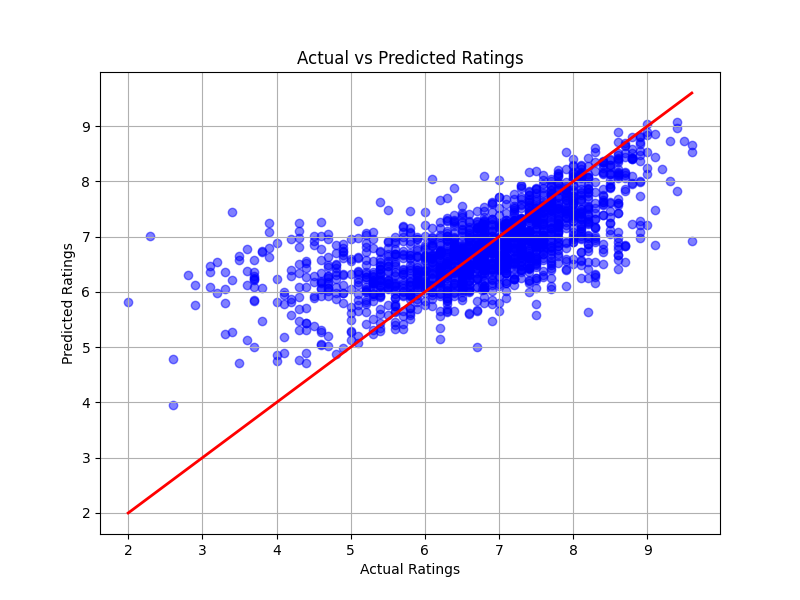
**n\_estimators** 200 model uses 200 trees for better prediction stability

**max\_depth** None trees grow fully (until other limits are reached) -> captures full patterns

**max\_features** 'sqrt' each split considers √number of features (reduces overfitting)

**min\_samples\_split** 2 nodes split if 2+ samples -> default, allows detailed splits

**min\_samples\_leaf** 1 leaves can have just 1 sample -> model captures fine details



- blue dots are more compact and closer to the red line

**conclusion = performance might still be improved**

**- gradient boosting regression**

builds trees sequentially, where each tree corrects the errors of the previous one, leading to better performance

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

- builds trees sequentially to correct errors, rather than in parallel like random forest

model = Pipeline([  
 ('preprocessor', preprocessor),  
 ('regressor', GradientBoostingRegressor(n\_estimators=100, learning\_rate=0.1, max\_depth=3, random\_state=42))  
])

- **GradientBoostingRegressor** : new model choice. it's better for precision and minimizing error

- **learning\_rate=0.1** : unique to gradient boosting — controls how much each tree corrects errors

- **max\_depth=3** : shallower trees by default (smaller than RF) — prevents overfitting in GBR

param\_grid = {  
 'regressor\_\_n\_estimators': [50, 100, 200],   
 'regressor\_\_max\_depth': [None, 10, 20],   
 'regressor\_\_min\_samples\_split': [2, 5, 10],   
 'regressor\_\_min\_samples\_leaf': [1, 2, 4],   
 'regressor\_\_max\_features': ['sqrt', 'log2']   
}

- **learning\_rate** : controls step size of each correction -> smaller = safer but slower

**the results of the model’s performance**

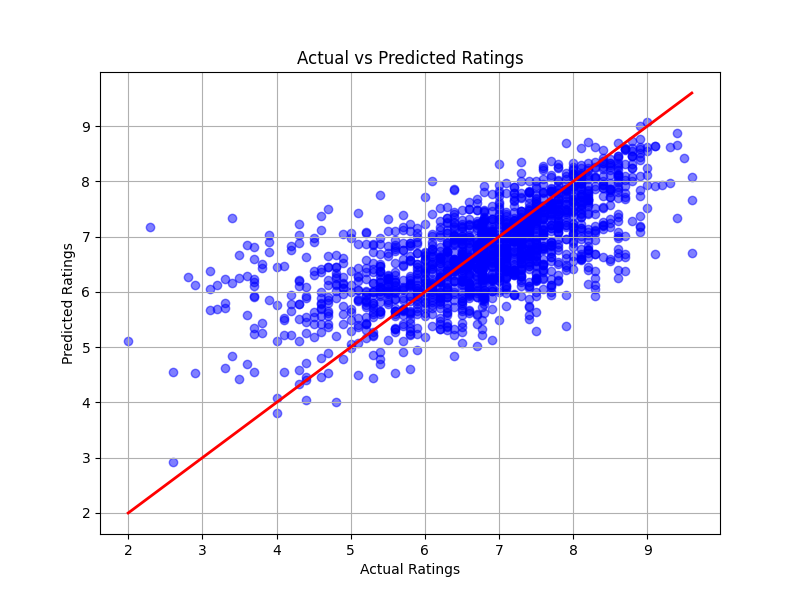
Best Parameters: {'regressor\_\_max\_depth': None, 'regressor\_\_max\_features': 'sqrt', 'regressor\_\_min\_samples\_leaf': 2, 'regressor\_\_min\_samples\_split': 5,

'regressor\_\_n\_estimators': 100}

Mean Squared Error: 0.695534209488002

R^2 Score: 0.46291893454094246

**=> there is still room for improvement**

****

- there is improvement

**conclusion = way better than before but still can improve**

1. **K-Nearest Neighbors (KNN) Regression**

preprocessor = ColumnTransformer(  
 transformers=[  
 ('num', StandardScaler(), numeric\_features),  
 ('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_features)  
])

- ColumnTransformer : this is used to apply different preprocessing steps to different columns in the dataset

transformers : a list of tuples specifying the preprocessing steps for different types of features

('num', StandardScaler(), numeric\_features):

'num' : a name for this transformation step

StandardScaler() : scales numeric features to have a mean of 0 and a standard deviation of 1

numeric\_features : a list of column names or indices for numeric features (['year', 'duration', 'votes', 'decade', 'age'])

('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_features):

'cat' : a name for this transformation step

OneHotEncoder() : converts categorical features into one-hot encoded vectors

handle\_unknown='ignore' : if a new category is encountered during testing, it will be ignored instead of throwing an error

categorical\_features : a list of column names or indices for categorical features (['genre', 'certificate', 'lead\_actor'])

pipeline = Pipeline(steps=[  
 ('preprocessor', preprocessor),  
 ('regressor', KNeighborsRegressor())  
])

- applies the ColumnTransformer defined

- initializes a KNN regressor

param\_grid = {  
 'regressor\_\_n\_neighbors': [3, 5, 7, 10, 15],   
 'regressor\_\_weights': ['uniform', 'distance'],   
 'regressor\_\_p': [1, 2]   
}

- param\_grid : a dictionary specifying the hyperparameters to tune and their possible values

regressor\_\_n\_neighbors : the number of neighbors to consider. KNN looks at the closest n\_neighbors points to make a prediction

regressor\_\_weights : how to weight the neighbors

'uniform' : all neighbors are weighted equally

'distance' : closer neighbors are given more weight

regressor\_\_p : the power parameter for the Minkowski distance metric

p=1: Manhattan distance

p=2: Euclidean distance (default)

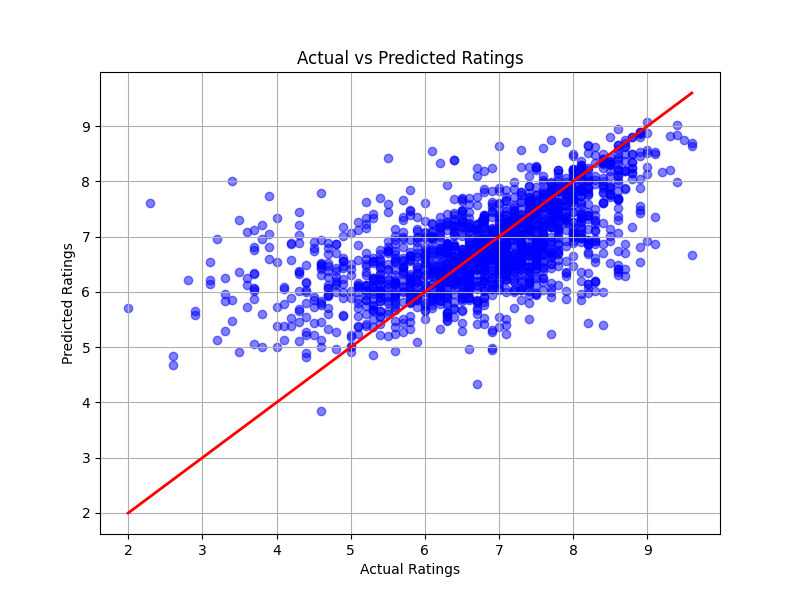
**the results of the model’s performance**

Best Parameters: {'regressor\_\_n\_neighbors': 15, 'regressor\_\_p': 1, 'regressor\_\_weights': 'distance'}

Mean Squared Error: 0.8240255364513894

R^2 Score: 0.3636998625724419

**=> the model’s performance receded from the last model**



**conclusion = GRADIENT BOOSTING REGRESSOR WAS THE MOST PROMISING MODEL WITH THE MOST ACCURATE RESULTS**

I don’t have a **confusion matrix**, which is a tool used to **evaluate the performance of a classification model** because the models I used are all related to regression (predicting ratings => continuous numbers)