Introduction

Table of Contents

1. Introduction
2. Data Wrangling
    - General Properties
    - Data Cleaning
3. Exploratory Data Analysis
    - Are there any obvious correlations?
    - Which actors, directors, and production companies produce the best movies by genre?
4. Modeling
    - Preprocessing
    - Fitting the Linear Regression Model
5. Explore Possible Improvements
    - Increase the number of actors, directors, and production companies used
    - Preprocessing Changes
6. New Models
    - Model 2
    - Model 3
    - Model 4
    - Model 5
7. Classification
    - Fitting the Classifier
    - CLF
8. Conclusions

For this project, I would like to predict the average rating of a movie before it has been released by building a machine learning model. The target audience for this project is the movie industry iteself. The goal here is to determine what combination of cast, director, production company, and budget will be suitable to produce a highly rated movie. In doing so, scriptwriters can choose the right cast, director, and production company to produce their movie; executives of a production company can choose the right cast and director given a specific budget; and a director can choose which actors to cast.

Using this notebook, I will detail each of the steps I take towards achieving this goal. I will begin by performing Data Wrangling, in which I will take a look at the data and see how I should properly clean and transform the data in order to proceed. Then, I will perform Exploratory Data Analysis to explore possible features to include in my model. I will then proceed to the Modeling phase where I will build and test an initial linear regression model. It is expected that the first model will have many issues, thus I will move onto the next phase where I will Explore Possible Improvements. Once a few ideas have been established, I will incorporate them by building several New Models and comparing their performances. The last phase of my project will involve Classification, in which I will find and convert models I believe will perform better as a classification model instead of a linear regression model.

By the end of the notebook, I will have built a finalized model capable of predicting a movie's average rating to a certain degree of accuracy.

Data Wrangling

General Properties

Description:

> Datasets are often riddled with various issues upon the first import, so I must examine the data and make necessary changes so my analysis can proceed as smooth as possible.

Procedure:

> Import the dataset and take a look at the data , and determine if necessary changes will be needed.
>
> 1. Make preliminary observations
>   A. Import dataset and look at the first 3 entries
>   B. Explore anything that stands out
> 2. Check for inconsistencies in data types, such as numerical values and dates being incorrectly represented as strings
> 3. Take a look at the columns and see which ones will be most important for my analysis

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.svm import SVC
```

In [2]:
```
# Import dataset and look at the first 3 entries
df = pd.read_csv('tmdb-movies.csv')
df.head(3)
```

Out[2]:

	id	imdb_id	popularity	budget	revenue	original_title	cast	
0	135397	tt0369610	32.985763	150000000	1513528810	Jurassic World	Chris Pratt|Bryce Dallas Howard|Irrfan Khan|Vi...	http://w
1	76341	tt1392190	28.419936	150000000	378436354	Mad Max: Fury Road	Tom Hardy|Charlize Theron|Hugh Keays-Byrne|Nic...	http://w
2	262500	tt2908446	13.112507	110000000	295238201	Insurgent	Shailene Woodley|Theo James|Kate Winslet|Ansel...	http://w

3 rows × 21 columns

In [3]:
```
# Check for multiple values
for x in df.columns:
    if type(df[x][0]) == str:
        if df[x].str.contains("\|").any():
            print(x)
```

```
cast
director
tagline
keywords
overview
genres
production_companies
```

> It seems that the **'cast'**, **'director'**, **'tagline'**, **'keywords'**, **'overview'**, **'genres'**, **'production_companies'** columns have multiple values per entry.
> I will have to keep this in mind and find a way to expand each entry for future analysis.

In [4]:
```
# Check for inconsistencies in data types
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10866 entries, 0 to 10865
Data columns (total 21 columns):
id                      10866 non-null int64
imdb_id                 10856 non-null object
popularity              10866 non-null float64
budget                  10866 non-null int64
revenue                 10866 non-null int64
original_title          10866 non-null object
cast                    10790 non-null object
homepage                2936 non-null object
director                10822 non-null object
tagline                 8042 non-null object
keywords                9373 non-null object
overview                10862 non-null object
runtime                 10866 non-null int64
genres                  10843 non-null object
production_companies    9836 non-null object
release_date            10866 non-null object
vote_count              10866 non-null int64
vote_average            10866 non-null float64
release_year            10866 non-null int64
budget_adj              10866 non-null float64
revenue_adj             10866 non-null float64
dtypes: float64(4), int64(6), object(11)
memory usage: 1.7+ MB
```

> The **release_date** column is of data type `str`, and will need to be parsed into a `datetime` object.

In [5]:
```
# Take a look at all columns
list(df.columns)
```

Out[5]:
```
['id',
 'imdb_id',
 'popularity',
 'budget',
 'revenue',
 'original_title',
 'cast',
 'homepage',
 'director',
 'tagline',
 'keywords',
 'overview',
 'runtime',
 'genres',
 'production_companies',
```

```
'release_date',
'vote_count',
'vote_average',
'release_year',
'budget_adj',
'revenue_adj']
```

> The dataset includes many columns, but the only columns I am interested in for my analysis are the following:
>
> - [**original_title**, **cast**, **director**, **genres**, **production_companies**, **release_date**, **vote_average**, **release_year**, **budget_adj**, **revenue_adj**]

Data Cleaning

Description:

> To clean the data, there are two main tasks I must complete: parsing the release_date column into `datetime` objects, and dropping several columns and rows.
> Firstly, I must parse the release_date column into `datetime` objects in order for python to recognize the values as dates.
> Secondly, I must drop all columns that are unnecessary for my analysis; I must drop all rows containing null values; and I must drop all rows containing zeros, specifically for columns containing numerical values (**vote_average**, **budget**, **revenue**).

Convert the 'release_date' column to datetime

> Parsing the column using the typical parse_dates argument of pd.read_csv or the pd.to_datetime function does not work properly. The reason being that the column contains years in the 1900s, and Python assumes all years are in the 21st century, resulting in incorrectly parsed future years. I will have to correctly identify and fix the problematic years before parsing the column.
>
> - Create a boolean mask to locate problematic years
> - Correctly label the dates
> - Parse the column to datetime

Columns and rows to drop

> I will drop the **budget** and **revenue** columns and use their adjusted values instead
>
> I will drop all columns except for the following, which I will be using for my analysis:

- [**original_title**, **cast**, **director**, **genres**, **production_companies**, **release_date**, **vote_average**, **release_year**, **budget**, **revenue**]

I will drop all null values

I will drop rows with zeros in relevant columns I will be analysing

- **vote_average**, **budget**, **revenue**

In [6]:
```
# Python does not properly parse certain years (ex: 1-1-66 gets parsed as
2066-1-1)
# Include a '19' in front of all years in the 1900's to clearly indicate t
he century
from datetime import datetime

df = pd.read_csv('tmdb-movies.csv')

# Boolean mask for years in the 1900's
incorrect_dates = df['release_year'] < 2000
# Correct the dates
df.loc[incorrect_dates,'release_date'] = df[incorrect_dates]['release_date
'].apply(lambda x: x[:-2] + '19' + x[-2:])
# Parse release_date column
df['release_date'] = pd.to_datetime(df['release_date'])
```

In [7]:
```
# Drop budget and revenue columns
df.drop(columns=['budget','revenue'], inplace=True)
# Rename budget_adj column to budget
df.rename(columns={'budget_adj':'budget','revenue_adj':'revenue'}, inplace
=True)

# List of columns to keep ['original_title','cast','director','genres','pr
oduction_companies','release_date','vote_average','release_year','budget']
keep_list = ['original_title','cast','director','genres','production_compa
nies','release_date','vote_average','budget','revenue']
# Update DataFrame
df = df[keep_list]

# Change all values of 'nan' to null values, then drop rows with null valu
es
df.dropna(subset=['original_title','cast','director','genres','production_
companies','release_date'], inplace=True)

# Drop rows with zeros in relevant columns
df = df.loc[df[['vote_average','budget','revenue']].ne(0).all(axis=1)]

df.head()
```

Out[7]:

	original_title	cast	director	genres	production_companies
0	Jurassic World	Chris Pratt|Bryce Dallas Howard|Irrfan	Colin Trevorrow	Action|Adventure|Science Fiction|Thriller	Universal Studios|Amblin Entertainment|Legenda...

		Khan|Vi...			
1	Mad Max: Fury Road	Tom Hardy|Charlize Theron|Hugh Keays-Byrne|Nic...	George Miller	Action|Adventure|Science Fiction|Thriller	Village Roadshow Pictures|Kennedy Miller Produ...
2	Insurgent	Shailene Woodley|Theo James|Kate Winslet|Ansel...	Robert Schwentke	Adventure|Science Fiction|Thriller	Summit Entertainment|Mandeville Films|Red Wago...
3	Star Wars: The Force Awakens	Harrison Ford|Mark Hamill|Carrie Fisher|Adam D...	J.J. Abrams	Action|Adventure|Science Fiction|Fantasy	Lucasfilm|Truenorth Productions|Bad Robot
4	Furious 7	Vin Diesel|Paul Walker|Jason Statham|Michelle ...	James Wan	Action|Crime|Thriller	Universal Pictures|Original Film|Media Rights ...

Exploratory Data Analysis

Are there any obvious correlations?

Description:

> I'd like to determine if there are any correlations in the data.
> I'll do this by comparing release month and budget to average rating.
> First, I'll determine which months have the highest average rating, and view the median number of movies released each month.
> Then I'll determine the average budget for movies by their release month.

Procedure:

> 1. Plot the average rating vs. release month
>   A. Create a copy of the DataFrame and name it **rating_by_month**
>   B. Convert the **release_date** column to numerical representations of the month

   C. Group by **release_date**, then find the mean of the **vote_average**, and plot the values
2. Plot the median number of movies each month
   A. Create a copy of the DataFrame with the index set to **release_date** and name it **monthly_count**
   B. Resample the **monthly_count** DataFrame by month, and count the number of movies that appear each month
   C. Drop the months with zero movies
   D. Reset the index
   E. Convert the **release_date** column to numerical representations of the month
   F. Group by **release_date**, then find the median number of movies for each month, and plot as a bar chart
3. Plot the average budget vs. release_month
   A. Create a copy of the DataFrame and name it **budget_info**
   B. Convert the **release_date** column to numerical representations of the month
   C. Group by **release_date**, then find the mean of the **budget**, and plot the values

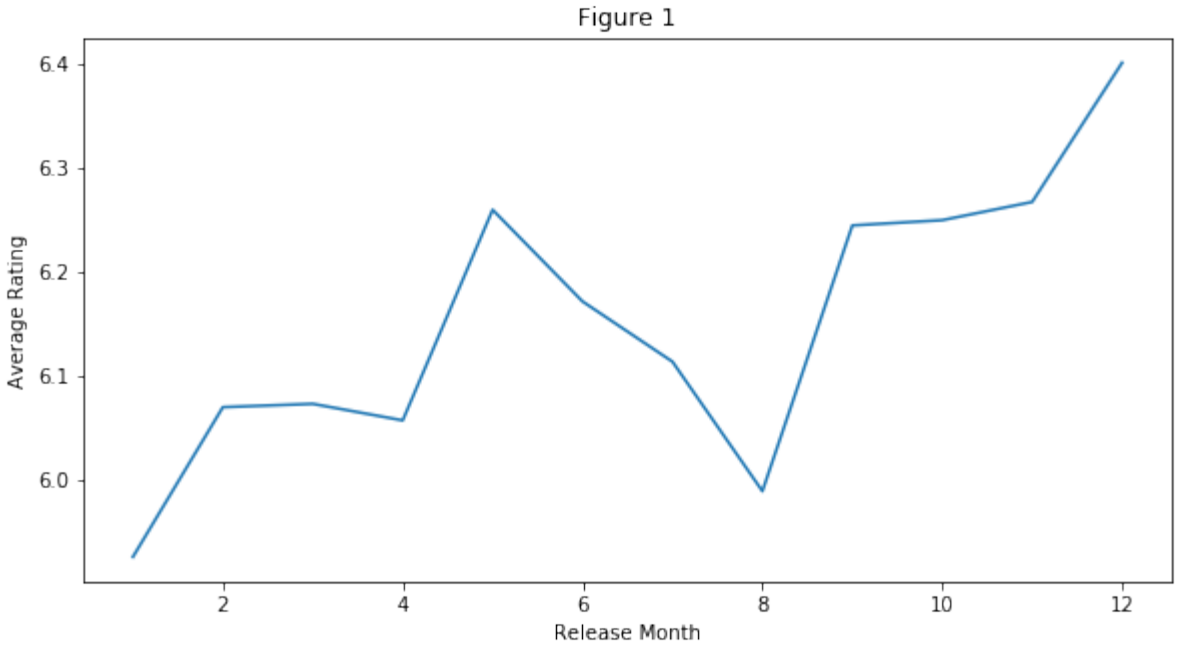
1. Plot the average rating vs. release month

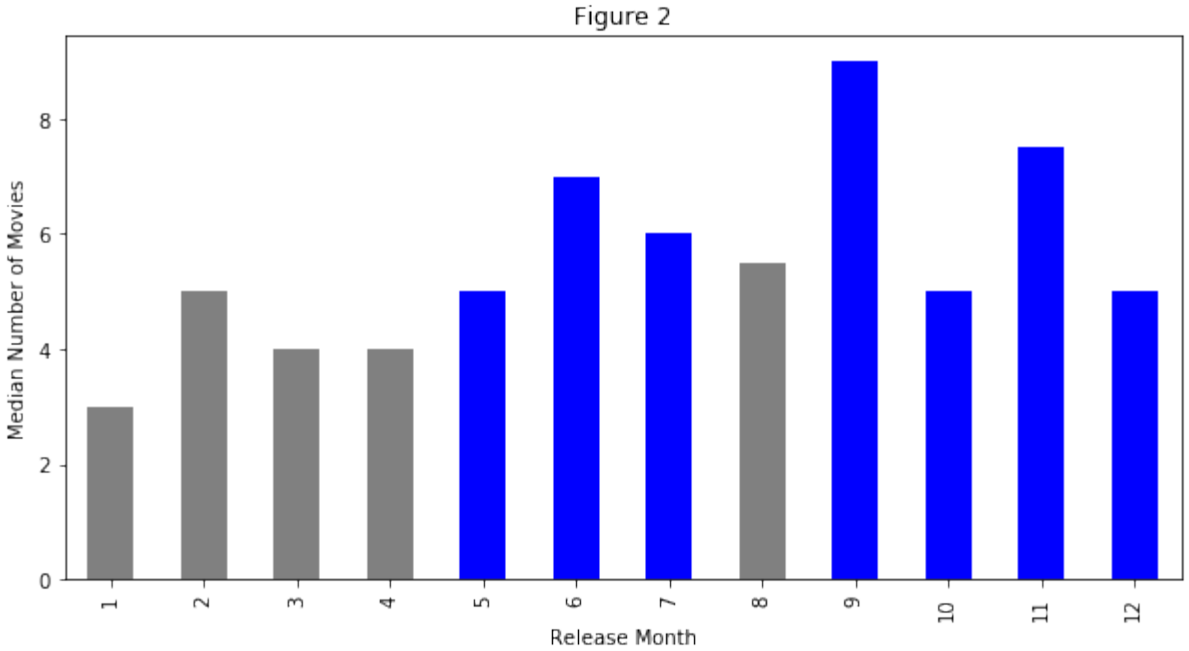
2. Plot the median number of movies each month

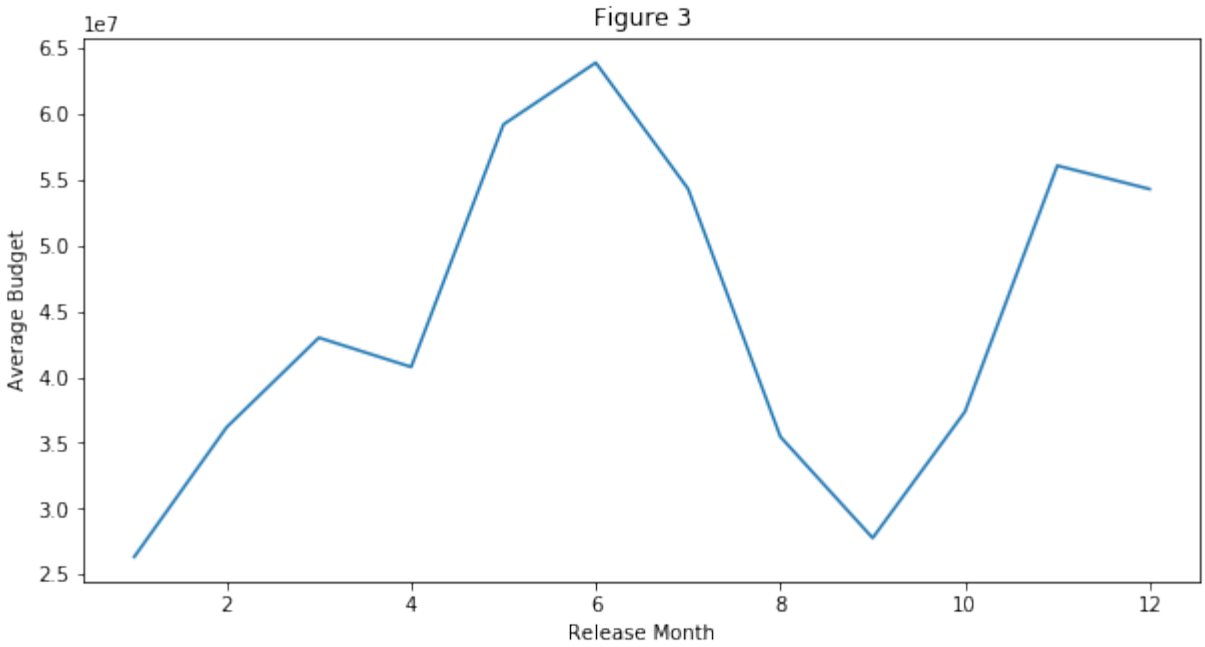
3. Plot the average budget vs. release_month

In [8]:
```
# Plot the average rating vs. release month
plt.figure
rating_by_month = df.copy()
rating_by_month['release_date'] = rating_by_month['release_date'].dt.month
rating_by_month.groupby('release_date')['vote_average'].mean().plot(figsiz
e=(10,5))
plt.subplots_adjust(left = 0.145)
plt.title('Figure 1')
plt.xlabel('Release Month')
plt.ylabel('Average Rating')
plt.show()

# Plot the median number of movies each month
plt.figure
monthly_count = df.set_index('release_date')
monthly_count = monthly_count.resample('M').count()
monthly_count = monthly_count.loc[monthly_count.ne(0).all(axis=1)]
monthly_count = monthly_count.reset_index()
monthly_count['release_date'] = monthly_count['release_date'].dt.month
standout = ['grey','grey','gray','grey','b','b','b','grey','b','b','b','b'
]
monthly_count.groupby('release_date')['original_title'].median().plot(kind
='bar', color=standout, figsize=(10,5))
plt.title('Figure 2')
```

```
plt.xlabel('Release Month')
plt.ylabel('Median Number of Movies')
plt.show()

# Plot the average budget vs. release_month
plt.figure
budget_info = df.copy()
budget_info['release_date'] = budget_info['release_date'].dt.month
budget_info.groupby('release_date')['budget'].mean().plot(figsize=(10,5))
plt.title('Figure 3')
plt.xlabel('Release Month')
plt.ylabel('Average Budget');
```







> By comparing Figures 1 and 2, it is quite clear that movies released on certain months have a higher average rating than others. This seems to be correlated with the observation that months with higher average ratings also have a higher number of newly released movies. These observations lead to a possible correlation between a movie's rating and the month it was released.
>
> By comparing Figures 1 and 3, an odd observation is discovered. The average ratings and the average budget for the months of May, June, July, November, and December seem to be correlated; but this is not true for the months of September, and October. Although the latter is observed, it seems that there may still be a correlation between a movie's budget and its rating.

Which actors, directors, and production companies produce the best movies by genre?

Description:

> I would like to find the top 10 actors/directors/production_companies by genre in order to use these values in the modeling phase to build my features. This will allow my model to predict a movie's ratings by learning who the best actors, directors, and production companies are.

Procedure:

> I will have to first write a function to expand each of these columns. Once the

columns are expanded, I will be able to group the data by the genres and the features, then find the 10 values with the highest rating.

1. Preliminary steps:
   A. Write a function to expand each of these columns, as each row contains multiple values
   B. Expand the columns
2. Find the top 10 actors by genre
   A. Group by the **genres** and **cast** column, then find the mean of the **vote_average** column
   B. Take the previous groupby object, groupby the **genres** colum, then find the top 10 values from the **vote_average** column
3. Find the top 10 directors by genre
   A. Group by the **genres** and **director** column, then find the mean of the **vote_average** column
   B. Take the previous groupby object, groupby the **genres** colum, then find the top 10 values from the **vote_average**
4. Find the top 10 production companies by genre
   A. Group by the **genres** and **production_companies** column, then find the mean of the **vote_average** column
   B. Take the previous groupby object, groupby the **genres** colum, then find the top 10 values from the **vote_average**

1. Preliminary steps:

In [9]:
```
# Function to expand columns with multiple values
def expand(df_in, column, sep='|'):
    """
    !!! IMPORTANT !!! :
        df_in MUST HAVE NO NULL VALUES
    !!! IMPORTANT !!!

    Split the values of a column and expand so the new
    DataFrame has one row per value.
    """
    indexes = list()
    new_values = list()
    # Find all rows, and column values to expand
    for i, presplit in enumerate(df_in[column]):
        # Expand the column values and store
        values = presplit.split(sep)
        new_values += values
        # Duplicate the row index and store
        indexes += [i] * len(values)
    # Copy of the original DataFrame with duplicate
     # rows to match expanded column
    df_out = df_in.iloc[indexes, :].copy()
    # Replace the column with the expanded values
    df_out[column] = new_values
    # Renumber the index
```

```
    df_out.index = list(range(len(df_out)))

    return df_out
```

In [10]:
```
# Test to see it can properly expand the columns

# Columns to expand
expand_columns = ['cast','director', 'genres','production_companies']

# Create a new dataframe
test = df.copy()
# Loop through all columns
count = 0
for column in expand_columns:
    test = expand(test,column)
    if not test[column].str.contains("\|").any():
        count += 1
    # Check to see it worked
    # Should print, "Properly expanded!"
    if count == 4:
        print('Properly expanded!')
```

```
Properly expanded!
```

In [11]:
```
# Expand the genres column, and store in a new DataFrame
expand_genres = expand(df.copy(), 'genres')

# Create a list of all the genres
genres_list = sorted(list(expand_genres['genres'].unique()))

# Create a new DataFrame for each attribute and expand the corresponding c
olumn
cast_genres = expand(expand_genres, 'cast')
dir_genres = expand(expand_genres, 'director')
prod_genres = expand(expand_genres, 'production_companies')
```

2. Find the top 100 actors by genre

In [12]:
```
# Find the top 10 actors by genre
cast_averages = pd.DataFrame(cast_genres.groupby(['genres', 'cast'])['vote
_average'].mean())
find_top_10 = cast_averages['vote_average'].groupby(level=0, group_keys=Fa
lse)
top_10_cast = pd.DataFrame(find_top_10.nlargest(10)).reset_index()
top_10_cast = top_10_cast.groupby('genres')['cast'].unique().to_dict()
```

3. Find the top 10 directors by genre

In [13]:
```
# Find the top 10 directors by genre
dir_averages = pd.DataFrame(dir_genres.groupby(['genres', 'director'])['vo
te_average'].mean())
find_top_10 = dir_averages['vote_average'].groupby(level=0, group_keys=Fal
se)
```

```
top_10_dir = pd.DataFrame(find_top_10.nlargest(10)).reset_index()
top_10_dir = top_10_dir.groupby('genres')['director'].unique().to_dict()
```

4. Find the top 10 production companies by genre

In [14]:
```
# Find the top 10 production companies by genre
prod_averages = pd.DataFrame(prod_genres.groupby(['genres', 'production_co
mpanies'])['vote_average'].mean())
find_top_10 = prod_averages['vote_average'].groupby(level=0, group_keys=Fa
lse)
top_10_prod = pd.DataFrame(find_top_10.nlargest(10)).reset_index()
top_10_prod = top_10_prod.groupby('genres')['production_companies'].unique
().to_dict()
```

Modeling

Preprocessing

Description:

> I must run my data through a preprocessing procedure in order to build the feature columns that my model will learn from.
>
> First, I must build the feature columns for the **cast**, **director**, and **production_companies**__ by counting the number of times each movie has a feature that appears at least once in the top 10 lists for its corresponding genres. To explain this, lets use a movie categorized under the genres of Action and Adventure as an example. I will then count the number of times the following are true: the movie has at least one actor from the top 10 action list, at least one actor from the top 10 adventure list, at least one director from the top 10 action list, etc.
>
> Next, I will build the feature columns for the **genres** by creating one column for each genre. Each column/genre will have a value of a 0 if the movie is not categorized under this genre, and a 1 if the movie is categorized under this genre.
>
> Lastly, I will build a feature column representing the month that each movie was released, by converting the **release_date** column into numerical representations of month.

Procedure:

> Build the feature columns for the **cast**, **director**, **production_companies**, **genres**,

and **release_date** in order to fit the data to a linear regression model.

1. Build the cast, director, and production_companies feature columns
    A. Loop through the features and their corresponding DataFrames
        a. Split the feature column into a list using the .split() method
    B. Loop through the genres
        a. Find all movies in correspondence with the top 10 of the current feature for the current genre
2. Build the genre feature columns
    A. Use pd.get_dummies() to create dummy variables
    B. Drop the original genres column
3. Collapse the DataFrame back to one movie per row
    A. Collapse genre features
        a. Groupby **original_title** and **release_date**, then sum the genre columns
    B. Collapse remaining features
        a. Groupby **original_title** and **release_date**, then find the unique values for the remaining columns
4. Build the release_date feature column
    A. Use the dt.month method to convert the column into integer values of month
5. Combine all steps into one function

1. Build the cast, director, and production_companies feature columns

In [15]:
```
# List of features
features = ['cast', 'director', 'production_companies']
# List of dataframes
dataframes = [top_10_cast, top_10_dir, top_10_prod]
# Copy the expand_genres DataFrame
model_df = expand_genres[expand_genres.columns]

# Loop through the features and their corresponding DataFrames
for i in range(3):
    feature = features[i]
    data =  dataframes[i]
    # Split the feature column
    model_df[feature] = model_df[feature].apply(lambda x: x.split('|'))

    # Loop through the genres
    for genre in genres_list:
        # Top 10 of the current feature for the current genre
        top_10 = data[genre]

        # Find all movies with a top_10 feature for the current genre
        temp = model_df.loc[model_df.genres == genre, feature].apply(lambd
a values: any(hit in values for hit in top_10)).astype(int)

        # Update the dataframe
        model_df.loc[model_df.genres == genre, feature] = temp
```

```
# Check the DataFrame to confirm changes
model_df.head(3)
```

Out[15]:

	original_title	cast	director	genres	production_companies	release_date	vote_average
0	Jurassic World	0	0	Action	0	2015-06-09	6.5
1	Jurassic World	0	0	Adventure	0	2015-06-09	6.5
2	Jurassic World	0	0	Science Fiction	0	2015-06-09	6.5

2. Build the genre feature columns

In [16]:
```
# Create dummy variables
model_df[genres_list] = pd.get_dummies(model_df['genres'])
# Drop the original 'genres' column
model_df.drop(columns='genres', inplace=True)

# Check the DataFrame to confirm changes
model_df.head(3)
```

Out[16]:

	original_title	cast	director	production_companies	release_date	vote_average	bu
0	Jurassic World	0	0	0	2015-06-09	6.5	1.379999e
1	Jurassic World	0	0	0	2015-06-09	6.5	1.379999e
2	Jurassic World	0	0	0	2015-06-09	6.5	1.379999e

3 rows × 28 columns

3. Collapse the DataFrame back to one movie per row

In [17]:
```
# Collapse the genre features back to one movie per row
genre_features = model_df.groupby(['original_title', 'release_date'])[genr
es_list].sum().reset_index()

# Collapse the remaining features back to one movie per row
agg_dict = {'cast':'sum', 'director':'sum', 'production_companies':'sum',
'vote_average':'mean', 'budget':'mean'}
remaining_features = model_df.groupby(['original_title', 'release_date']).
agg(agg_dict)
```

```
# Join the features
model_df = genre_features.join(remaining_features, on=['original_title', '
release_date'])

# Check the DataFrame to confirm changes
model_df.head(10)
```

Out[17]:

	original_title	release_date	Action	Adventure	Animation	Comedy	Crime	Documentary	I
0	(500) Days of Summer	2009-07-17	0	0	0	1	0	0	1
1	10 Things I Hate About You	1999-03-30	0	0	0	1	0	0	1
2	10,000 BC	2008-02-22	1	1	0	0	0	0	1
3	101 Dalmatians	1996-11-17	0	0	0	1	0	0	(
4	102 Dalmatians	2000-10-07	0	0	0	1	0	0	(
5	10th & Wolf	2006-02-19	1	0	0	0	1	0	1
6	12 Rounds	2009-03-19	1	1	0	0	0	0	1
7	12 Years a Slave	2013-10-18	0	0	0	0	0	0	1
8	127 Hours	2010-11-05	0	1	0	0	0	0	1
9	13 Going On 30	2004-04-13	0	0	0	1	0	0	(

10 rows × 27 columns

4. Build the release_date feature column

In [18]:
```
# Convert release dates to month
model_df['release_date'] = model_df['release_date'].dt.month

# Check the DataFrame to confirm changes
model_df.head(3)
```

Out[18]:

	original_title	release_date	Action	Adventure	Animation	Comedy	Crime	Documentary	I
0	(500) Days of Summer	7	0	0	0	1	0	0	1
	10 Things I								

1	Hate About You	3	0	0	0	1	0	0	1
2	10,000 BC	2	1	1	0	0	0	0	1

3 rows × 27 columns

5. Combine all steps into one function

In [19]:
```
def preprocessing(top_cast, top_dir, top_prod, model_df):
    '''1. Build the cast, director, and production_companies feature colum
ns'''
    # List of features
    features = ['cast', 'director', 'production_companies']
    # List of dataframes
    dataframes = [top_cast, top_dir, top_prod]

    # Loop through the features and their corresponding DataFrames
    for i in range(3):
        feature = features[i]
        data =  dataframes[i]
        # Split the feature column
        model_df[feature] = model_df[feature].apply(lambda x: x.split('|')
)

        # Loop through the genres
        for genre in genres_list:
            try:
                # Top n of the current feature for the current genre
                top_n = data[genre]

                # Find all movies with a top_n feature for the current gen
re
                temp = model_df.loc[model_df.genres == genre, feature].app
ly(lambda values: any(hit in values for hit in top_n)).astype(int)

                # Update the dataframe
                model_df.loc[model_df.genres == genre, feature] = temp
            except:
                model_df.loc[model_df.genres == genre, feature] = 0

    '''2. Build the genre feature columns'''
    # Create dummy variables for genres
    model_df[genres_list] = pd.get_dummies(model_df['genres'])
    # Drop the original 'genres' column
    model_df.drop(columns='genres', inplace=True)

    '''3. Collapse the DataFrame back to one movie per row'''
    # Collapse the genre features back to one movie per row
    genre_features = model_df.groupby(['original_title', 'release_date'])[
genres_list].sum().reset_index()

    # Collapse the remaining features back to one movie per row
    agg_dict = {'cast':'sum', 'director':'sum', 'production_companies':'su
```

```
m', 'vote_average':'mean', 'budget':'mean'}
    remaining_features = model_df.groupby(['original_title', 'release_date
']).agg(agg_dict)

    # Join the features
    model_df = genre_features.join(remaining_features, on=['original_title
', 'release_date'])

    '''4. Build the release_date feature column'''
    # Convert release dates to month
    model_df['release_date'] = model_df['release_date'].dt.month

    return model_df
```

Fitting the Linear Regression Model

Description:

> Fit the data to a linear regression model using OLS (Ordinary Least Squares)

Procedure:

> Using the **cast**, **director**, **production_companies**, **release_date**, **budget**, and the **genres** columns as the features and the **vote_average** column as the target, split the data into training and test sets. Fit the model to the training set, and use the fitted model to predict the average rating.
>
> 1. Split the data into training and test sets
>   A. Split the DataFrame into two variables, X for the features, and y for the target
>   B. Use the train_test_split function to split the data into a training set, and a test set
>   C. Standardize the features of the training and test sets
> 2. Fit the model to the training set using the LinearRegression class from sklearn.linear_model
> 3. Plot the ratings vs their predictions
> 4. Combine all steps into one function

1. Split the data into training and test sets

In [20]:
```
# Variable to hold feature data
X = model_df.drop(columns=['original_title', 'vote_average'])

# Variable to hold target data
y = model_df['vote_average']
```

```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Standardize the features
temp = StandardScaler().fit(X_train)
X_train = temp.transform(X_train)
X_test = temp.transform(X_test)
```

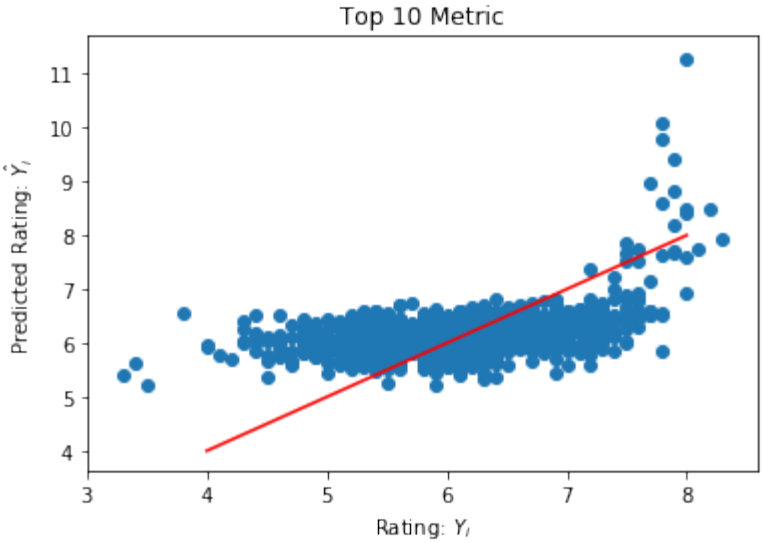
2. Fit the model to the training set

In [21]:
```
model = LinearRegression()
results = model.fit(X_train, y_train)
```

3. Evaluate model performance

In [22]:
```
# Results using sklearn
y_pred = results.predict(X_test)

# Print the R^2 score
print('R\u00b2 (Training): {}'.format(results.score(X_train, y_train)))
print('R\u00b2 (Test): {}'.format(results.score(X_test, y_test)))

# Print the mean squared error
print('MSE: {}'.format(mean_squared_error(y_test, y_pred)))

# Plot the Rating vs. Predicted Rating
plt.plot([4,8],[4,8], color='red')
plt.scatter(y_test, y_pred)
plt.xlabel("Rating: $Y_i$")
plt.ylabel("Predicted Rating: $\hat{Y}_i$")
plt.title("Top 10 Metric");
```

```
R² (Training): 0.2552991953573587
R² (Test): 0.21130813333466592
MSE: 0.4738747447476443
```



Top 10 metric results

> As you can see from the plot above, this model does a poor job of predicting the ratings.
> I think the main reason for its poor performance is the scarcity of the **cast**, **director**, and **production_companies** feature columns.
>
> I will now explore some ideas to alter those feature columns in order to improve the model.
>
> Comparisons:
>
> - top 500 metric
> - new preprocessing
> - model 2 (old preprocessing)

4. Combine all steps into one function

In [23]:
```
def fit_model(model_df, title):
    '''1. Split the data into training and test sets'''
    # Variable to hold feature data
    X = model_df.drop(columns=['original_title', 'vote_average'])

    # Variable to hold target data
    y = model_df['vote_average']

    # Split the data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state
=42)

    # Standardize the features
    temp = StandardScaler().fit(X_train)
    X_train = temp.transform(X_train)
    X_test = temp.transform(X_test)

    '''2. Fit the model to the training set'''
    model = LinearRegression()
    results = model.fit(X_train, y_train)

    '''3. Evaluate model performance'''
    # Results using sklearn
    y_pred = results.predict(X_test)

    # Print the R^2 score
    print('R\u00b2 (Training): {}'.format(results.score(X_train, y_train))
)
    print('R\u00b2 (Test): {}'.format(results.score(X_test, y_test)))
```

```
    # Print the mean squared error
    print('MSE: {}'.format(mean_squared_error(y_test, y_pred)))

    # Plot the Rating vs. Predicted Rating
    plt.plot([4,8],[4,8], color='red')
    plt.scatter(y_test, y_pred)
    plt.xlabel("Rating: $Y_i$")
    plt.ylabel("Predicted Rating: $\hat{Y}_i$")
    plt.title(title);

    return results
```

Explore Possible Improvements

Increase the number of actors, directors, and production companies used

Description:

> I will adjust the model to use the top 500 actors, directors, and production companies, instead of just the top 10.
> This will definitely increase the number of non-zero values in the feature columns.

Procedure:

> Adjust the model by finding the top 500 actors, directors, and production companies, then repeat the steps performed during the Modeling process.
>
> 1. Find the top 500 actors, directors, and production companies by genre
> 2. Preprocessing and Fitting the Linear Regression Model

1. Find the top 500 actors, directors, and production companies by genre

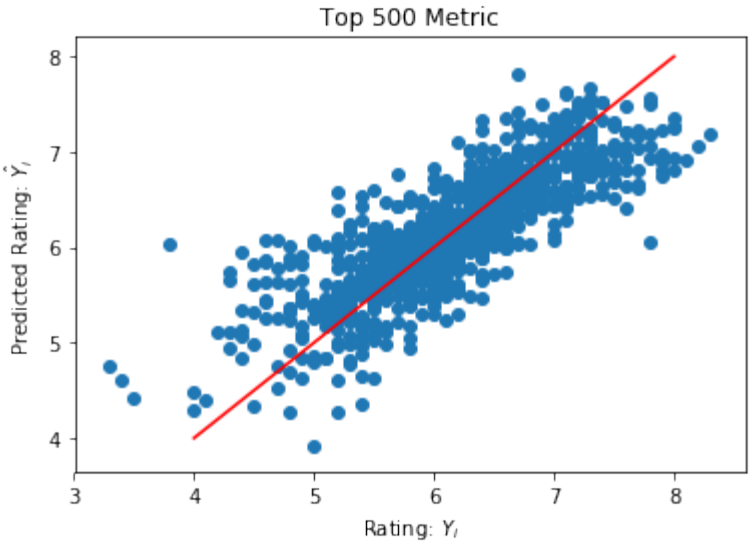
In [24]:
```
# Create a new DataFrame for each attribute and expand the corresponding c
olumn
cast_genres = expand(expand_genres, 'cast')
dir_genres = expand(expand_genres, 'director')
prod_genres = expand(expand_genres, 'production_companies')

# Find the top 500 actors by genre
cast_averages = pd.DataFrame(cast_genres.groupby(['genres', 'cast'])['vote
```

```
_average'].mean())
find_top_500 = cast_averages['vote_average'].groupby(level=0, group_keys=F
alse)
top_500_cast = pd.DataFrame(find_top_500.nlargest(500)).reset_index()
top_500_cast = top_500_cast.groupby('genres')['cast'].unique().to_dict()


# Find the top 500 directors by genre
dir_averages = pd.DataFrame(dir_genres.groupby(['genres', 'director'])['vo
te_average'].mean())
find_top_500 = dir_averages['vote_average'].groupby(level=0, group_keys=Fa
lse)
top_500_dir = pd.DataFrame(find_top_500.nlargest(500)).reset_index()
top_500_dir = top_500_dir.groupby('genres')['director'].unique().to_dict()

# Find the top 500 production companies by genre
prod_averages = pd.DataFrame(prod_genres.groupby(['genres', 'production_co
mpanies'])['vote_average'].mean())
find_top_500 = prod_averages['vote_average'].groupby(level=0, group_keys=F
alse)
top_500_prod = pd.DataFrame(find_top_500.nlargest(500)).reset_index()
top_500_prod = top_500_prod.groupby('genres')['production_companies'].uniq
ue().to_dict()
```

2. Preprocessing and Fitting the Linear Regression Model

In [25]:
```
# Prepare the DataFrame for modeling
model_df = expand_genres[expand_genres.columns]
model_df = preprocessing(top_500_cast, top_500_dir, top_500_prod, model_df
)

# Fit the model
title = 'Top 500 Metric'
results = fit_model(model_df, title)
```

```
R² (Training): 0.6601872747752968
R² (Test): 0.6617422074423811
MSE: 0.20323757842828108
```



Top 500 metric results

> These predicted values have an R^2 score of 66.17%, with a Mean Squared Error of 0.2. Compared to the results of the top 10 metric, this is a huge improvement!
>
> The problem with this metric, however, is that there is no justification for using the top ***500*** actors, directors, and production companies.
> 500 is just too large of a number, and defeats the entire purpose of the metric, which was to find the best actors, directors, and production companies to build the model's features.
>
> Despite the issues with this metric, it has shown that an increase in the number of actors, directors, and production companies greatly improved the model.
> Lets save this knowledge for later to create a new model implementing a new metric that includes a higher number of actors, directors, and production companies with a bit more control.
>
> Comparisons:
>
> - top 10
> - new preprocessing
> - model 2 (old preprocessing)
> - model 3 (old preprocessing)
> - model 4 (old preprocessing)
> - model 5

Preprocessing Changes

Description:

> In order to increase the numerical value of each feature column, I'm going to alter the preprocessing procedure to change how the **cast**, **director**, **production_companies** feature columns are built.
> I will now count the total number of the top actors, directors, and production companies that are listed for each movie by genre.

Procedure:

> Alter the preprocessing function to count the number of top actors, directors, and production companies listed for each movie.
>
> 1. Compare performance using the top 10 metric
> 2. Compare performance using the top 500 metric

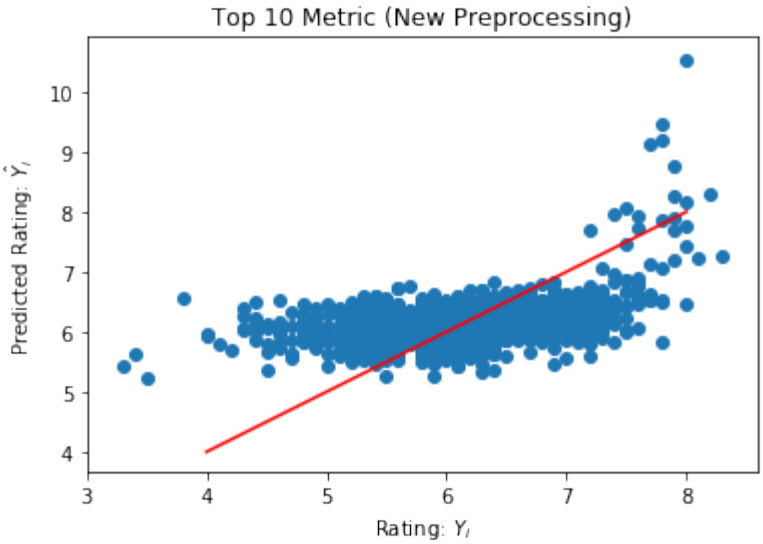
In [26]:
```
def new_preprocessing(top_cast, top_dir, top_prod, model_df):
    '''1. Build the cast, director, and production_companies feature colum
ns'''
    # List of features
    features = ['cast', 'director', 'production_companies']
    # List of dataframes
    dataframes = [top_cast, top_dir, top_prod]

    # Loop through the features and their corresponding DataFrames
    for i in range(3):
        feature = features[i]
        data =  dataframes[i]
        # Split the feature column
        model_df[feature] = model_df[feature].apply(lambda x: x.split('|')
)

        # Loop through the genres
        for genre in genres_list:
            try:
                # Top n of the current feature for the current genre
                top_n = data[genre]

                # Find all movies with a top_n feature for the current gen
re
                temp = model_df.loc[model_df.genres == genre, feature].app
ly(lambda values: sum(hit in values for hit in top_n))

                # Update the dataframe
                model_df.loc[model_df.genres == genre, feature] = temp
            except:
                model_df.loc[model_df.genres == genre, feature] = 0

    '''2. Build the genre feature columns'''
    # Create dummy variables for genres
    model_df[genres_list] = pd.get_dummies(model_df['genres'])
```

```
    # Drop the original 'genres' column
    model_df.drop(columns='genres', inplace=True)

    '''3. Collapse the DataFrame back to one movie per row'''
    # Collapse the genre features back to one movie per row
    genre_features = model_df.groupby(['original_title', 'release_date'])[
genres_list].sum().reset_index()

    # Collapse the remaining features back to one movie per row
    agg_dict = {'cast':'sum', 'director':'sum', 'production_companies':'su
m', 'vote_average':'mean', 'budget':'mean'}
    remaining_features = model_df.groupby(['original_title', 'release_date
']).agg(agg_dict)

    # Join the features
    model_df = genre_features.join(remaining_features, on=['original_title
', 'release_date'])

    '''4. Build the release_date feature column'''
    # Convert release dates to month
    model_df['release_date'] = model_df['release_date'].dt.month

    return model_df
```

1. Compare performance using the top 10 metric

In [27]:
```
# Prepare the DataFrame for modeling
model_df = expand_genres[expand_genres.columns]
model_df = new_preprocessing(top_10_cast, top_10_dir, top_10_prod, model_d
f)

# Fit the model
title = 'Top 10 Metric (New Preprocessing)'
results = fit_model(model_df, title)
```

```
R² (Training): 0.24792179100483147
R² (Test): 0.22042639919728424
MSE: 0.4683961591417616
```



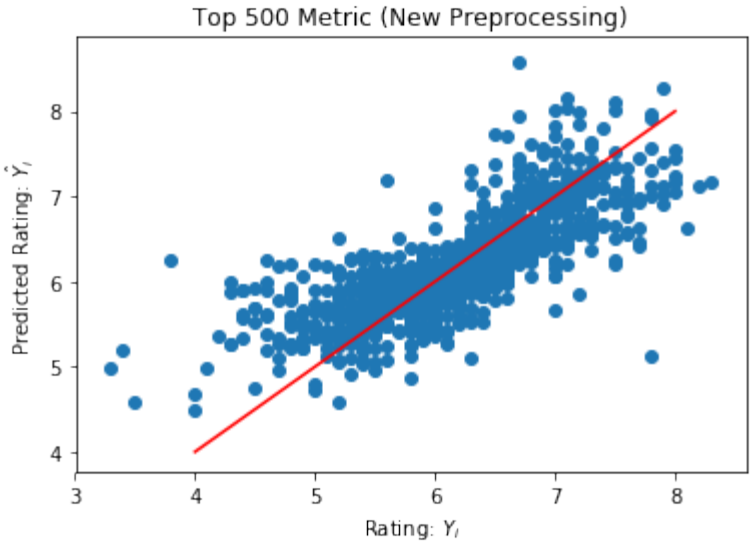
New preprocessing using top 10 metric results

> There is little to no difference in performance for the top 10 metric (old preprocessing).
>
> Comparisons:
>
> - old preprocessing
> - model 2 (new preprocessing)

2. Compare performance using the top 500 metric

In [28]:
```
# Prepare the DataFrame for modeling
model_df = expand_genres[expand_genres.columns]
model_df = new_preprocessing(top_500_cast, top_500_dir, top_500_prod, mode
l_df)

# Fit the model
title = 'Top 500 Metric (New Preprocessing)'
results = fit_model(model_df, title)
```

```
R² (Training): 0.6052314759134532
R² (Test): 0.587876427040588
MSE: 0.24761882453074427
```



New preprocessing using top 500 metric results

> The performance of the new preprocessing method performs slightly worse for the top 500 metric (old preprocessing).
>
> Comparisons:
>
> - old preprocessing
> - model 2 (new preprocessing)
> - model 3 (new preprocessing)
> - model 4 (new preprocessing)

New Models

Model 2

Description:

> I will now create a new model to find the *best* actors, directors, and production companies by genre instead of simply the top #.
>
> - My personal definition of a "good" movie has a cut-off of 7/10; as such, I

have chosen a minimum rating of 7.

Purpose:
I know that an increase in the number of actors, directors, and production companies has a positive effect on model performance. Therefore, this new metric allows me to increase the number of actors, directors, and production companies while granting me a level of control by being able to choose the minimum rating.

Procedure:

Find all movies with a minimum rating of 7; find all actors, directors, and production companies by genre; then build the model.

1. Find the top actors, directors, and production companies by genre
   A. Find all movies with a minimum rating of 7
   B. Find the top actors, directors, and production companies by genre
2. Evaluate performance using the old preprocessing method
3. Evaluate performance using the new preprocessing method

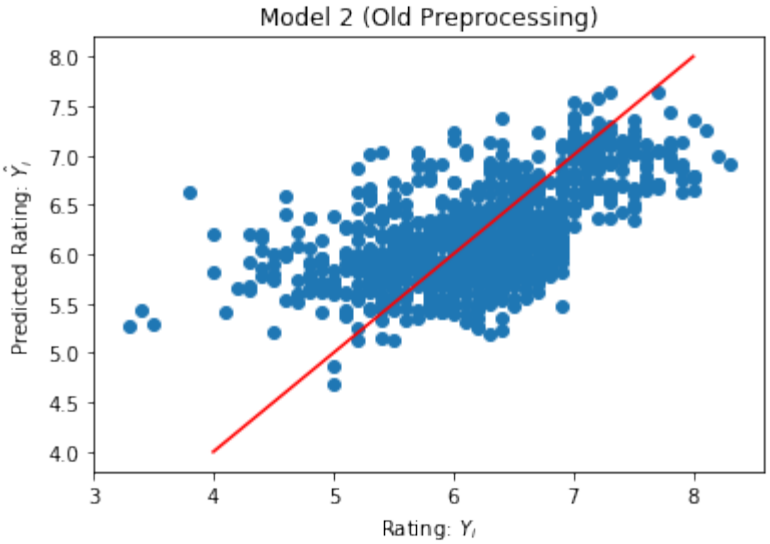
1. Find the top actors, directors, and production companies by genre

In [29]:
```
# Find all movies with a minimum rating of 7
good_movies = expand_genres.loc[expand_genres.vote_average >= 7]

# Create a new DataFrame for each attribute and expand the corresponding c
olumn
cast_genres = expand(good_movies, 'cast')
dir_genres = expand(good_movies, 'director')
prod_genres = expand(good_movies, 'production_companies')

# Find the top actors
top_cast = cast_genres.groupby('genres')['cast'].unique()
top_cast = top_cast.to_dict()

# Find the top directors
top_dir = dir_genres.groupby('genres')['director'].unique()
top_dir = top_dir.to_dict()

# Find the top production companies
top_prod = prod_genres.groupby('genres')['production_companies'].unique()
top_prod = top_prod.to_dict()
```

2. Evaluate performance using the old preprocessing method

In [30]:
```
# Prepare the DataFrame for modeling
model_df = expand_genres[expand_genres.columns]
model_df = preprocessing(top_cast, top_dir, top_prod, model_df)

# Fit the model
title = 'Model 2 (Old Preprocessing)'
results = fit_model(model_df, title)
```

```
R² (Training): 0.3836005424729261
R² (Test): 0.355842205758068
MSE: 0.3870334197995831
```



Model 2 using old preprocessing results

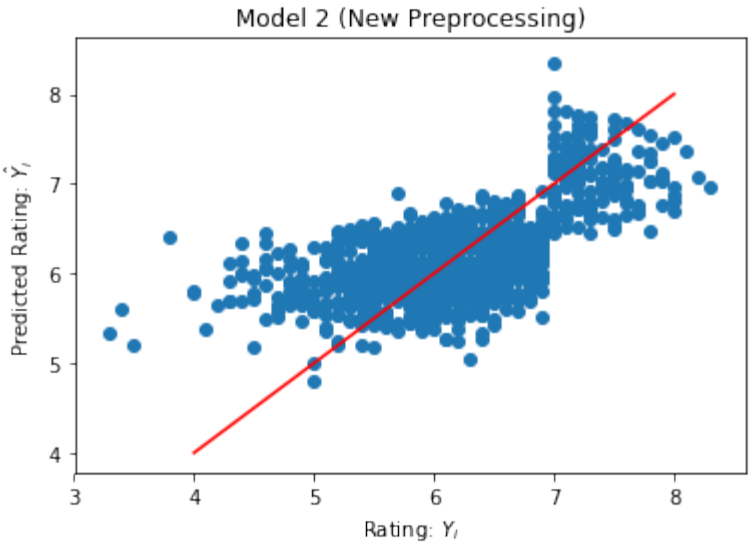
> Using the old preprocessing method, this model performs better than the previous model when comparing by the top 10 metric, but is outperformed by the top 500 metric.
>
> Comparisons:
>
> - model 1 top 500 metric (old preprocessing)
> - model 3 (old preprocessing)

3. Evaluate performance using the new preprocessing method

In [31]:
```
# Prepare the DataFrame for modeling
model_df = expand_genres[expand_genres.columns]
model_df = new_preprocessing(top_cast, top_dir, top_prod, model_df)

# Fit the model
```

```
title = 'Model 2 (New Preprocessing)'
results = fit_model(model_df, title)
```

```
R² (Training): 0.42407397272083824
R² (Test): 0.40357773139743336
MSE: 0.3583521806695396
```



Model 2 using new preprocessing results

> It is clear that compared to itself, this model performs better using the new preprocessing method. However, when compared to the previous model, it is still outperformed by the top 500 metric.
>
> Regardless of which preprocessing model is used, this new model it outperformed by the top 500 metric.
>
> Comparisons:
>
> - model 1 top 500 metric (new preprocessing)
> - model 3 (new preprocessing)

Model 3

Description:

> Adjust Model 2 to round ratings to a full number, or to a half number. For example, ratings will be rounded as such:
>
> - (5.8 - 6.2 => 6.0), (6.3 - 6.7 => 6.5), (6.8 - 7.2 => 7.0)

> **_Purpose:_**
> Reduce the number of possible ratings in the hopes of creating a better model.

Procedure:

> Round all ratings; find all actors, directors, and production companies by genre; then build the model.
>
> 1. Round all ratings
>   A. Define function to round the ratings
>   B. Initialize the DataFrame
>   C. Use pandas .apply() method and round all ratings
> 2. Find the top actors, directors, and production companies by genre
>   A. Find all movies with a minimum rating of 7
>   B. Find the top actors, directors, and production companies by genre
> 3. Evaluate performance using the old preprocessing method
> 4. Evaluate performance using the new preprocessing method

1. Round all ratings

In [32]:
```
# Import Decimal object
from decimal import Decimal

# Define function to round the ratings
def round_ratings(rating):
    # Find the rating's whole value
    whole_value = rating//1
    # Find the rating's decimal value
    dec_value = float(Decimal(str(rating)) % 1)

    # Round to whole number
    if dec_value in [0.8, 0.9, 0.0, 0.1, 0.2]:
        return round(rating)
    # Round to half number
    else:
        return whole_value + 0.5

# Initialize DataFrame
rounded_ratings = expand_genres[expand_genres.columns]

# Round ratings
rounded_ratings['vote_average'] = rounded_ratings['vote_average'].apply(ro
und_ratings)
```

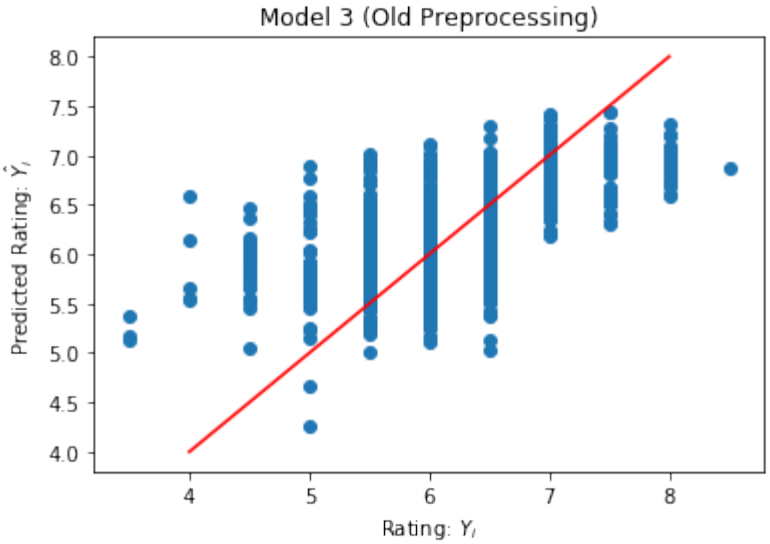
2. Find the top actors, directors, and production companies by genre

In [33]:
```
# Find all movies with a minimum rating of 7
good_movies = rounded_ratings.loc[rounded_ratings.vote_average >= 7]

# Create a new DataFrame for each attribute and expand the corresponding c
olumn
cast_genres = expand(good_movies, 'cast')
dir_genres = expand(good_movies, 'director')
prod_genres = expand(good_movies, 'production_companies')

# Find the top actors
top_cast = cast_genres.groupby('genres')['cast'].unique()
top_cast = top_cast.to_dict()

# Find the top directors
top_dir = dir_genres.groupby('genres')['director'].unique()
top_dir = top_dir.to_dict()

# Find the top production companies
top_prod = prod_genres.groupby('genres')['production_companies'].unique()
top_prod = top_prod.to_dict()
```

3. Evaluate performance using the old preprocessing method

In [34]:
```
'''Old Preprocessing Method'''
# Prepare the DataFrame for modeling
model_df = rounded_ratings[rounded_ratings.columns]
model_df = preprocessing(top_cast, top_dir, top_prod, model_df)

# Fit the model
title = 'Model 3 (Old Preprocessing)'
results = fit_model(model_df, title)
```

```
R² (Training): 0.41453363441676483
R² (Test): 0.37215905878236744
MSE: 0.3823527875543526
```



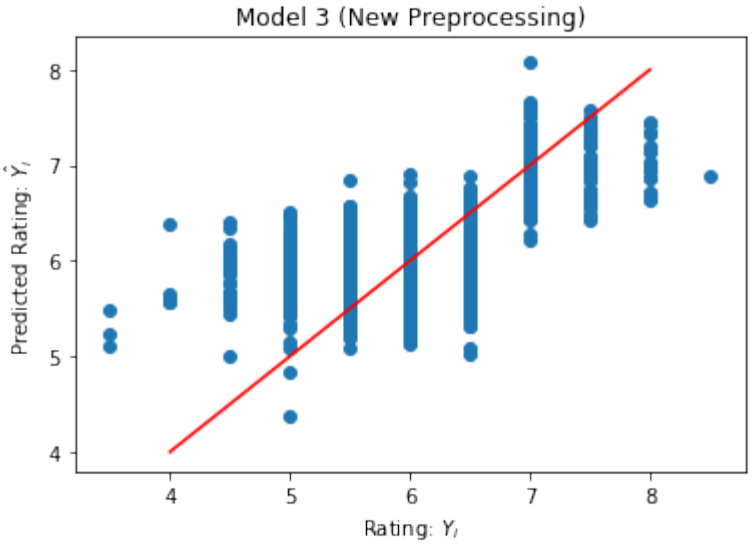
Model 3 using old preprocessing results

> Rounding the ratings actually improved the performance of the second model using the old preprocessing method! Unfortunately, it is still outperformed by the first model using the top 500 metric and the old preprocessing method.
>
> Comparisons:
>
> - model 1 top 500 metric (old preprocessing)
> - model 2 (old preprocessing)
> - model 4 (old preprocessing)

4. Evaluate performance using the new preprocessing method

In [35]:
```
'''New Preprocessing Method'''
# Prepare the DataFrame for modeling
model_df = rounded_ratings[rounded_ratings.columns]
model_df = new_preprocessing(top_cast, top_dir, top_prod, model_df)

# Fit the model
title = 'Model 3 (New Preprocessing)'
results = fit_model(model_df, title)
```

```
R² (Training): 0.46484232997218944
R² (Test): 0.4313814685423739
MSE: 0.3462865612685848
```



Model 3 using new preprocessing results

> Rounding the ratings actually improved the performance of the second model using the new preprocessing method, but is met with the same setback of being outperformed by the first model using the top 500 metric and the new preprocessing method.
>
> Comparisons:
>
> - model 1 top 500 metric (new preprocessing)
> - model 2 (new preprocessing)
> - model 4 (new preprocessing)

Model 4

Description:

> I will adjust Model 3 to bin the ratings for movies so that movies with a rating less than 5 will be rated as a 4, movies with a rating between 5 and 5.9 will be rated as a 5, movies with a rating between 6 and 6.9 will be rated as a 7, and so on.
>
> **_Purpose:_**
> Improve on Model 3 by further reducing the number of possible ratings.

Procedure:

> Bin all ratings; find all actors, directors, and production companies by genre; then build the model.
>
> 1. Bin all ratings
>   A. Initialize the DataFrame
>   B. Bin all ratings
> 2. Find the top actors, directors, and production companies by genre
>   A. Find all movies with a minimum rating of 7
>   B. Find the top actors, directors, and production companies by genre
> 3. Evaluate performance using the old preprocessing method
> 4. Evaluate performance using the new preprocessing method

1. Bin all ratings

In [36]:
```
# Initialize DataFrame
```

```
bin_ratings = expand_genres[expand_genres.columns]

# Bin ratings
bin_ratings.loc[bin_ratings['vote_average'] < 5, 'vote_average'] = 4
for rating in range(5, 10):
    bin_ratings.loc[bin_ratings['vote_average'].between(rating, rating+0.9
), 'vote_average'] = rating
```

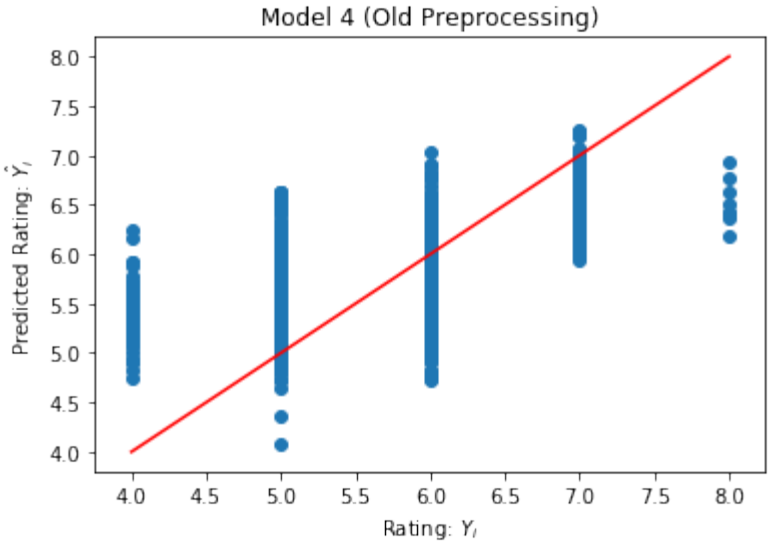
2. Find the top actors, directors, and production companies by genre

In [37]:
```
# Find all movies with a minimum rating of 7
good_movies = bin_ratings.loc[bin_ratings.vote_average >= 7]

# Create a new DataFrame for each attribute and expand the corresponding c
olumn
cast_genres = expand(good_movies, 'cast')
dir_genres = expand(good_movies, 'director')
prod_genres = expand(good_movies, 'production_companies')

# Find the top actors
top_cast = cast_genres.groupby('genres')['cast'].unique()
top_cast = top_cast.to_dict()

# Find the top directors
top_dir = dir_genres.groupby('genres')['director'].unique()
top_dir = top_dir.to_dict()

# Find the top production companies
top_prod = prod_genres.groupby('genres')['production_companies'].unique()
top_prod = top_prod.to_dict()
```

3. Evaluate performance using the old preprocessing method

In [38]:
```
'''Old Preprocessing Method'''
# Prepare the DataFrame for modeling
model_df = bin_ratings[bin_ratings.columns]
model_df = preprocessing(top_cast, top_dir, top_prod, model_df)

# Fit the model
title = 'Model 4 (Old Preprocessing)'
results = fit_model(model_df, title)
```

```
R² (Training): 0.3901675188144301
R² (Test): 0.35126825509888027
MSE: 0.43199846183066487
```



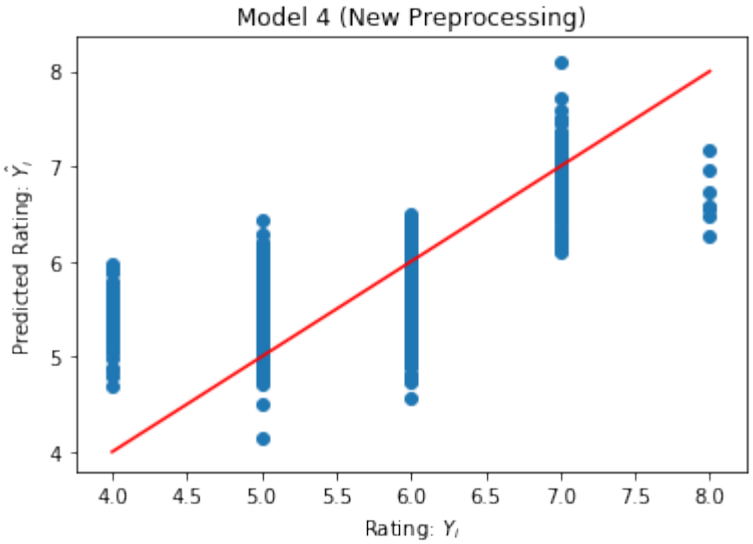
Model 4 using old preprocessing results

> Binning the ratings slightly decreased the performance of the third model using the old preprocessing method, meaning that this model is also outperformed by the first model using the top 500 metric and the old preprocessing method.
>
> Comparisons:
>
> - model 1 top 500 metric (old preprocessing)
> - model 3 (old preprocessing)

4. Evaluate performance using the new preprocessing method

In [39]:
```
'''New Preprocessing Method'''
# Prepare the DataFrame for modeling
model_df = bin_ratings[bin_ratings.columns]
model_df = new_preprocessing(top_cast, top_dir, top_prod, model_df)

# Fit the model
title = 'Model 4 (New Preprocessing)'
results = fit_model(model_df, title)
```

```
R² (Training): 0.45101181710641347
R² (Test): 0.41665319122511096
MSE: 0.3884578272687888
```



Model 4 using new preprocessing results

> Once again, binning the ratings slightly decreased the performance of the third model using the new preprocessing method, meaning that this model is also outperformed by the first model using the top 500 metric and the new preprocessing method.
>
> Comparisons:
>
> - model 1 top 500 metric (new preprocessing)
> - model 3 (new preprocessing)

Model 5

Description:

> All the models so far have been built using the old and new preprocessing procedures; both of which are very similar, and involves a list of actors, directors, and production companies for each genre to build the models' features.
>
> I will take this idea one step further by creating one final preprocessing procedure that takes the idea from Model 4 to bin the ratings, in order to create a list of actors, directors, and production companies for each genre and bin combination.
>
> For each bin, I will create separate feature columns for the actors, directors, and production companies.
> This means that for n number of bins there will be n number of columns of actors, and an equivalent number of columns for directors and production companies. Because

> there are three features (actors, directors, and production companies), I will have n*3 total feature columns.
>
> Each column will count the number of actors, directors, and production companies listed per movie similar to before.
>
> This may be a bit confusing, so I'll try to explain using an example.
> Lets say a movie is categorized under the Action and Adventure genres, and I am using bins between 4 and 8.
>
> Because I have 5 bins, I will have 5 cast feature columns, 5 director feature columns, and 5 production_companies feature columns where each of those 5 columns corresponds to bins 4-8.
>
> If this movie has 2 actors in the Action genre under bin 5, 1 actor in the Action genre under bin 6, and 3 actors in the Adventure genre under bin 6, then its cast feature columns will look like this:
>
> - 4.0 cast: 0
> - 5.0 cast: 2
> - 6.0 cast: 4
> - 7.0 cast: 0
> - 8.0 cast: 0 </br>

Procedure:

> Build the preprocessing function; find all actors, directors, and production companies by genres and bins; then build the model.
>
> 1. Build preprocessing function
> 2. Find all actors, directors, and production companies by genres and bins
>   A. Bin all ratings
>   B. Group the top actors, directors, and production companies by genres and bins
>   C. Unstack the bins, then turn to dictionary
> 3. Evaluate performance

1. Build preprocessing function

In [40]:
```
def final_preprocessing(cast_bins, dir_bins, prod_bins, model_df):
    '''1. Replace the cast, director, and production_companies columns wit
h dummy variables'''
    # List of features
    features = ['cast', 'director', 'production_companies']
    # List of dataframes
    dataframes = [cast_bins, dir_bins, prod_bins]
    # List of bins
    bins = list(cast_bins['Action'].keys())
```

```
    # Loop through the features and their corresponding DataFrames
    for i in range(3):
        feature = features[i]
        data =  dataframes[i]
        # Split the feature column
        model_df[feature] = model_df[feature].apply(lambda x: x.split('|')
)

        # Loop through the genres
        for genre in genres_list:
            genre_data = data[genre]
            # Loop through the bins
            for bin_n in bins:
                # List of values for the current feature, genre, and bin
                list_n = genre_data[bin_n]

                if list_n is not None:
                    # Count the number of values from list_n that appear i
n each movie
                    temp = model_df.loc[model_df.genres == genre, feature]
.apply(lambda values: sum(hit in values for hit in list_n))

                    # Update the dataframe
                    model_df.loc[model_df.genres == genre, str(bin_n) + '
' + feature] += temp
                else:
                    # Update the dataframe
                    model_df.loc[model_df.genres == genre, str(bin_n) + '
' + feature] = 0

    '''2. Create dummy variables for each genre in the genres column'''
    # Create dummy variables for genres
    model_df[genres_list] = pd.get_dummies(model_df['genres'])
    # Drop the original 'genres' column
    model_df.drop(columns='genres', inplace=True)

    '''3. Collapse the DataFrame back to one movie per row'''
    # Collapse the genre features back to one movie per row
    genre_features = model_df.groupby(['original_title', 'release_date'])[
genres_list].sum().reset_index()

    # Collapse the remaining features back to one movie per row
    agg_dict = {'vote_average':'mean', 'budget':'mean'}
    for bin_n in bins:
        agg_dict[str(bin_n)+' cast'] = 'mean'
        agg_dict[str(bin_n)+' director'] = 'mean'
        agg_dict[str(bin_n)+' production_companies'] = 'mean'
    remaining_features = model_df.groupby(['original_title', 'release_date
']).agg(agg_dict)

    # Join the features
    model_df = genre_features.join(remaining_features, on=['original_title
', 'release_date'])

    '''4. Convert the release_dates into months'''
    # Convert release dates to month
```

```
    model_df['release_date'] = model_df['release_date'].dt.month

    return model_df
```

2. Find all actors, directors, and production companies by genres and bins

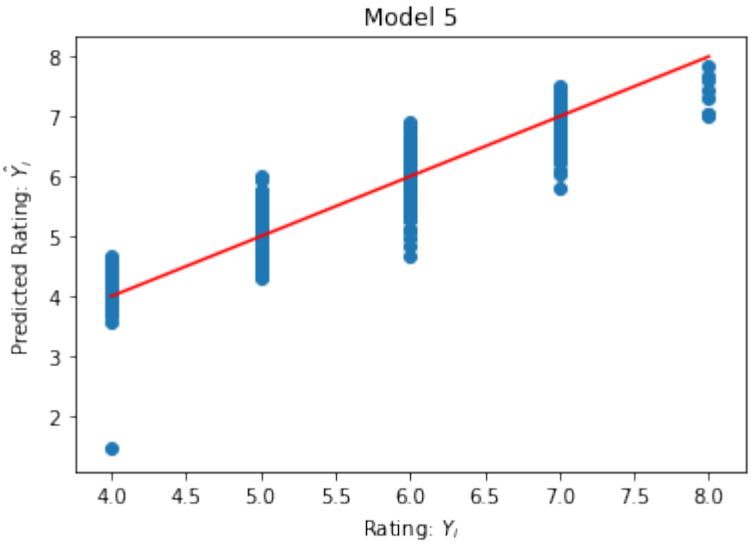
In [41]:
```
# Initialize DataFrame
bin_ratings = expand_genres[expand_genres.columns]
original_ratings = list(bin_ratings.vote_average)

# Bin ratings
bin_ratings.loc[bin_ratings['vote_average'] < 5, 'vote_average'] = 4
for rating in range(5, 10):
    bin_ratings.loc[bin_ratings['vote_average'].between(rating, rating+0.9
), 'vote_average'] = rating

# Create a new DataFrame for each attribute and expand the corresponding c
olumn
cast_genres = expand(bin_ratings, 'cast')
dir_genres = expand(bin_ratings, 'director')
prod_genres = expand(bin_ratings, 'production_companies')

# Groupby genres and bins, and find the list of cast
cast_bins = cast_genres.groupby(['genres', 'vote_average'])['cast'].unique
()
cast_bins = cast_bins.unstack().to_dict(orient='index')

# Groupby genres and vote_average, and find the list of directors
dir_bins = dir_genres.groupby(['genres', 'vote_average'])['director'].uniq
ue()
dir_bins = dir_bins.unstack().to_dict(orient='index')

# Groupby genres and vote_average, and find the list of production compani
es
prod_bins = prod_genres.groupby(['genres', 'vote_average'])['production_co
mpanies'].unique()
prod_bins = prod_bins.unstack().to_dict(orient='index')
```

3. Evaluate performance

In [42]:
```
# Prepare the DataFrame for modeling
model_df = bin_ratings[bin_ratings.columns]

# Initialize bin columns
bins = sorted(list(model_df.vote_average.unique()))
for bin_n in bins:
    model_df[str(bin_n) + ' ' + 'cast'] = 0
    model_df[str(bin_n) + ' ' + 'director'] = 0
    model_df[str(bin_n) + ' ' + 'production_companies'] = 0

# Run the preprocessing procedure
model_df = final_preprocessing(cast_bins, dir_bins, prod_bins, model_df)
```

```
# Fit the model
title = 'Model 5'
results = fit_model(model_df, title)
```

```
R² (Training): 0.8629151408785175
R² (Test): 0.8423997321430676
MSE: 0.10494796012899413
```



Model 5 results

> This model performs extremely well!
> With an R_2 of 84.2%, it is the first model to exceed the performance of the first model using the top 500 metric and the old preprocessing method.
>
> The one downfall, is that it falls short for movies with a rating of 8.0. The reason seems to be associated with the sparse amount of data for movies of that rating. I'm confident that if the dataset was updated to provide movies with a uniform amount of ratings, this model would perform even better.
>
> With this model, I have finally created a model with great performance and explainable features, unlike the first model.
>
> Although this model is great as is, it may perform even better as a classifier. Lets try to convert it into one.
>
> Comparisons:
>
> - model 1 top 500 metric (old preprocessing)
> - clf

Classification

Fitting the Classifier

Description:

> I will build a function to fit the data to a classifier using SVC (Support Vector Classification).
> This will not be able to predict a movie's rating using a continuous range, instead, it will classify movies into groups.

Procedure:

> Using the **cast**, **director**, **production_companies**, **release_date**, **budget**, and the **genres** columns as the features and the **vote_average** column as the target, split the data into training and test sets. Fit the model to the training set, and use the fitted model to predict the average rating.
>
> 1. Split the data into training and test sets
>   A. Split the DataFrame into two variables, X for the features, and y for the target
>   B. Use the train_test_split function to split the data into a training set, and a test set
>   C. Standardize the features of the training and test sets
> 2. Fit the model to the training set using the SVC class from sklearn.svm
> 3. Plot the ratings vs their predictions

In [43]:
```
def fit_clf(model_df, title):
    '''1. Split the data into training and test sets'''
    # Variable to hold feature data
    X = model_df.drop(columns=['original_title', 'vote_average'])

    # Variable to hold target data
    y = model_df['vote_average']

    # Split the data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state
=42)

    # Standardize the features
    temp = StandardScaler().fit(X_train)
    X_train = temp.transform(X_train)
    X_test = temp.transform(X_test)
```

```
    '''2. Fit the model to the training set'''
    model = SVC(kernel='linear')
    results = model.fit(X_train, y_train)

    '''3. Evaluate model performance'''
    # Results using sklearn
    y_pred = results.predict(X_test)

    # Print the accuracy
    print('Accuracy (Training): {}'.format(results.score(X_train, y_train)
))
    print('Accuracy (Test): {}'.format(results.score(X_test, y_test)))

    # Plot the Rating vs. Predicted Rating
    temp = pd.DataFrame({'$Y_i$':y_test.values.astype(float), '$\hat{Y}_i$
':y_pred.astype(float)})
    sns.swarmplot(x='$Y_i$', y='$\hat{Y}_i$', data=temp)
    plt.title(title);

    return results
```

CLF

Description:

> Using the fit_clf() function I have written above, I will not turn Model 5 into a classifier.
>
> > **_Purpose:_**
> > Because Model 5 has binned ratings, it may perform better as a classifier. Instead of making predictions within a linear range, the model will make strict predictions using the rounded ratings as classes.

Procedure:

> Follow the same procedure as Model 5, using the fit_clf() function to build a classifier.
>
> 1. Find all actors, directors, and production companies by genres and bins
>   A. Bin all ratings
>   B. Group the top actors, directors, and production companies by genres and bins
>   C. Unstack the bins, then turn to dictionary
> 2. Evaluate performance

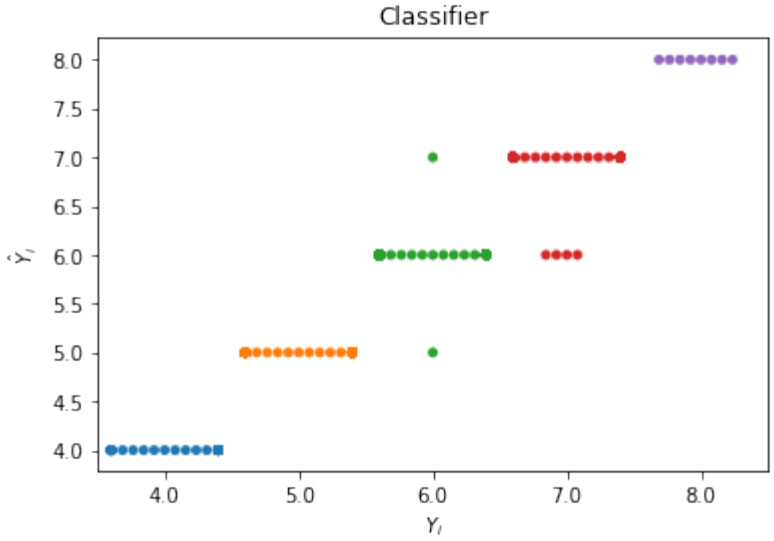
1. Find all actors, directors, and production companies by genres and bins

In [44]:
```
# Initialize DataFrame
bin_ratings = expand_genres[expand_genres.columns]

# Bin ratings
bin_ratings.loc[bin_ratings['vote_average'] < 5, 'vote_average'] = 4
for rating in range(5, 10):
    bin_ratings.loc[bin_ratings['vote_average'].between(rating, rating+0.9
), 'vote_average'] = rating

# Create a new DataFrame for each attribute and expand the corresponding c
olumn
cast_genres = expand(bin_ratings, 'cast')
dir_genres = expand(bin_ratings, 'director')
prod_genres = expand(bin_ratings, 'production_companies')

# Groupby genres and bins, and find the list of cast
cast_bins = cast_genres.groupby(['genres', 'vote_average'])['cast'].unique
()
cast_bins = cast_bins.unstack().to_dict(orient='index')

# Groupby genres and vote_average, and find the list of directors
dir_bins = dir_genres.groupby(['genres', 'vote_average'])['director'].uniq
ue()
dir_bins = dir_bins.unstack().to_dict(orient='index')

# Groupby genres and vote_average, and find the list of production compani
es
prod_bins = prod_genres.groupby(['genres', 'vote_average'])['production_co
mpanies'].unique()
prod_bins = prod_bins.unstack().to_dict(orient='index')
```

2. Evaluate performance

In [45]:
```
# Prepare the DataFrame for modeling
model_df = bin_ratings[bin_ratings.columns]

# Initialize bin columns
bins = sorted(list(model_df.vote_average.unique()))
for bin_n in bins:
    model_df[str(bin_n) + ' ' + 'cast'] = 0
    model_df[str(bin_n) + ' ' + 'director'] = 0
    model_df[str(bin_n) + ' ' + 'production_companies'] = 0

# Run the preprocessing procedure
model_df = final_preprocessing(cast_bins, dir_bins, prod_bins, model_df)
model_df['vote_average'] = model_df['vote_average'].astype(str)

# Fit the model
title = 'Classifier'
results = fit_clf(model_df, title)
```

```
Accuracy (Training): 0.9996494917630564
Accuracy (Test): 0.9936974789915967
```



Model 5 results

> This classifier, as expected, improves the performance of the Model 5, with a shockingly high accuracy score of 99.4%!
>
> The classifier can now properly classify movies with a rating of 8.
> The regression model on the other hand, incorrectly predicted all such movies, but managed to get relatively close.
>
> Its hard to say with certainty, if the model is better as a classifier or as a regression model.
> On one hand, the classifier is capable of classifying nearly all movies correctly, but is unable to predict a movie's rating numerically.
>
> On the other hand, the regression model is far less accurate with each rating fluctating between an error of (+/-)1.0 on average, but it offers more flexibility since its predictions follow a linear range.
>
> Comparisons:
>
> - model 5

Conclusions

In conclusion, I have built two models that can predict a movie's rating within reasonable accuracy, thus achieving my goal. These two models use the exact same features to learn from, but the

difference is that one is a linear regression model while the other is a classifier. The linear regression model is far less accurate, with an R^2 score of 84.2%, while the classifier has an accuracy score of 99.4%. They are both good models, but have different uses. The linear regression model will be good for predicting a rating within a continuous range if you are less concerned with accuracy. The classifier, on the other hand, is only capable of creating discrete predictions which is great for predicting ratings using a grading scale, offering a high level of accuracy for general grades.

To reach my goal, I have built several models. In total, I have built 5 linear regression models and 1 classifier model. The first regression model was the base for the next 4 models, in which each succeeding model is built upon the model that preceded it. The classification model was adapted from the fifth linear regression model and was built using the same features, with the only difference being that it was turned from a regression model into a classification model. The fifth linear regression model and the classifier, respectively, are the two final models that I mentioned in the previous paragraph.

With these two models, a client working in the movie industry will be able to test different combinations of actors, directors, and production companies with varying budgets to create a highly rated movie. A production company will typically offer a specified budget that the client can use to calculate the cost of production, and determine which actors and directors will be feasible. The client can then hold auditions and decide which actors they like the most, but it may be hard to choose which actors will play their respective roles the best. To remedy this problem, my models will be able to present insight on the combinations of actors and directors by using the predicted ratings. This is just one specific problem that my project can resolve, thus demonstrating its utility.

There are many aspects that come into play when trying to produce a great movie. My project does not offer a solution to every single problem, but it does a tremendous job of predicting movie ratings using a combination of actors, directors, production companies, genres, and budget. With this knowledge, clients will have greater confidence in producing their best movies.