[redacted]

Big Data Coursework - Netflix Dataset - Individual Part

In the group part, we loaded, cleaned and preprocessed the data.

In my individual part, I will keep on cleaning and transforming the data. Then, I will create several models to predict the average Netflix rating based on independent variables, I will have selected in the meantime.

Table of Contents

0.0 Importing Libraries and Preparing Environment

In [2]:
```
#usual library imports
import time
import numpy as np
import pandas as pd

#Library for Plotting
import seaborn as sns
sns.set(style="darkgrid")
import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:
```
# We will monitor the time it takes to run the notebook
start = time.time()
```

1.0 Business Objective and Context

Licensing agreements are made between Netflix and the bodies that produce TV shows and movies. These agreements give Netflix the right to stream the content. In 2019, Netflix spent 15.3 billion dollars on these agreements and this spending is forecast to reach 17 billion dollars in 2020.

We want to build a model that predicts the Netflix rating of a movie/TV show. This would allow Netflix to determine which new content to obtain the license for and how much they should spend on it. If predicted ratings for a new movie are high enough, then it is worth Netflix paying for the license. If not, it is not worth it. These decisions are even more important, when they relate to *exclusive content* as obtaining the license for these is more expensive than non-exclusive content.

The predictive model will use average Netflix rating as the dependent variable and IMDB data about the specific TV show/movie for the independent variables.

Thus, my individual model will predict the average rating of a new movie/TV show.

2.0 Data Loading and Selection

At the end of the group part, we created four dataframes, one for the train datasets and another one for the test dataset. Each set was split in two dataframes, one for the dependent variable y and one for the independent variables x.

2.1 Data Loading

In [4]:
```
# Load the four files in dataframes.
x_train= pd.read_csv("x_train.csv", index_col=0 )
y_train= pd.read_csv("y_train.csv",index_col=0 )
x_test= pd.read_csv("x_test.csv", index_col=0)
y_test= pd.read_csv("y_test.csv", index_col=0)
```

Let's have a look at the train datasets, their shape and the types of their variables.

In [5]:
```
x_train.head()
```

Out[5]:

	Movie_Id	Year_of_release	color	director_name	num_critic_for_reviews	duration	director_facebook_likes	actor_1_name
1612	11969	1980.0	Color	John Carpenter	212.0	89.0	0.0	Jamie Lee Curtis
1463	10846	2000.0	Color	Spike Lee	57.0	135.0	0.0	Gillian White
2268	17491	2002.0	Color	John Schultz	39.0	99.0	13.0	Brenda Song
550	3881	2004.0	Black and White	Larry Blamire	88.0	90.0	56.0	Fay Masterson
552	3890	2004.0	Color	Sara Sugarman	74.0	89.0	10.0	Adam Garcia

5 rows × 37 columns

In [6]:
```
x_train.dtypes
```

Out[6]:
```
Movie_Id                       int64
Year_of_release              float64
color                         object
director_name                 object
num_critic_for_reviews       float64
duration                     float64
director_facebook_likes      float64
actor_1_name                  object
actor_2_name                  object
actor_3_name                  object
actor_1_facebook_likes       float64
actor_2_facebook_likes       float64
actor_3_facebook_likes       float64
cast_total_facebook_likes      int64
movie_facebook_likes           int64
gross                        float64
num_voted_users                int64
facenumber_in_poster         float64
num_user_for_reviews         float64
language                      object
country                       object
content_rating                object
budget                       float64
imdb_score                   float64
genres_1                      object
genres_2                      object
genres_3                      object
genres_4                      object
genres_5                      object
genres_6                      object
genres_7                      object
genres_8                      object
plot_keywords_1               object
plot_keywords_2               object
plot_keywords_3               object
plot_keywords_4               object
plot_keywords_5               object
dtype: object
```

In [7]:
```
y_train.head()
```

Out[7]:

	avg_rating
1612	3.308080
1463	3.078547
2268	3.357953
550	2.927778
552	2.942760

In [8]:
```
print(x_train.shape)

print(y_train.shape)
```

```
(1609, 37)
(1609, 1)
```

2.2 Data Selection

In the dataframes with the independent variables, we have many categorical and numerical values, to be more precise 37 variables. Before moving forward with the feature engineering, correlation and the other steps, I would like to select the independent variables I will use.

Thanks to the several insights I had with the descriptive statistics, I will create a new dataframe with the columns I want from the x_train dataframe.

In the new dataframe, I will only use the facebook likes instead of the actors' and directors'names. Assuming that Netflix wants to target diverse customers, Netflix would like to have movies that represent as many genres as posible. That is why, I will not use the genres and thus the plot keywords.

```
In [9]: # I select the columns I want
        x_trainset = x_train.loc[:,['Movie_Id','Year_of_release','color','num_critic_for_reviews',
                                    'duration','director_facebook_likes','actor_1_facebook_likes',
                                    'actor_2_facebook_likes','actor_3_facebook_likes',
                                    'cast_total_facebook_likes','movie_facebook_likes','gross',
                                    'num_voted_users','facenumber_in_poster','num_user_for_reviews',
                                    'language','country','content_rating','budget','imdb_score']]
        x_trainset.shape

Out[9]: (1609, 20)
```

Create a new dataframe for the **test set**:

```
In [10]: x_testset = x_test.loc[:,['Movie_Id','Year_of_release','color','num_critic_for_reviews','duration',
                                   'director_facebook_likes','actor_1_facebook_likes','actor_2_facebook_likes',
                                   'actor_3_facebook_likes','cast_total_facebook_likes','movie_facebook_likes',
                                   'gross','num_voted_users','facenumber_in_poster','num_user_for_reviews',
                                   'language','country','content_rating','budget','imdb_score']]
         x_testset.shape

Out[10]: (690, 20)
```

3.0 Data Cleaning and Transformation

We can move forward with the data cleaning and transformation.

3.1 Feature Engineering

I will create a few new parameters.

In [11]:
```
#Calculate the ROI to see whether the movie was bankable, by dividing the gross revenue by the budget
x_trainset['ROI'] = x_trainset['gross']/x_trainset['budget']

# Calculate a famous actor/actress radar by averaging the three main actors facebook likes
# mean is calculated row-wise and for the selected columns, while skipping the null and
# missing values.
x_trainset['celebrity_radar'] = x_trainset[['actor_1_facebook_likes','actor_2_facebook_likes',
                                            'actor_3_facebook_likes']].mean(axis = 1, skipna=True )

# If the country is USA, use 1 and if it is not the case, use 0
x_trainset['USA_Country'] = [1 if x == 'USA' else 0 for x in x_trainset['country']]

# If the content category is not R, PG, PG-13, then put it in a other category
x_trainset['Content_Rating_Cat'] = [ x if x=='R' or x=='PG' or x=='PG-13' else 'Other'
                                     for x in x_trainset['content_rating']]

# We want to delete the columns, hence axis = 1 and directly in the dataframe, hence inplace=True
x_trainset.drop(["country", "language", "content_rating"], axis=1, inplace=True)

#Have a look at the dataframe
x_trainset.head()
```

Out[11]:

	Movie_Id	Year_of_release	color	num_critic_for_reviews	duration	director_facebook_likes	actor_1_facebook_likes	actor_
1612	11969	1980.0	Color	212.0	89.0	0.0	2000.0	
1463	10846	2000.0	Color	57.0	135.0	0.0	1000.0	
2268	17491	2002.0	Color	39.0	99.0	13.0	1000.0	
550	3881	2004.0	Black and White	88.0	90.0	56.0	126.0	
552	3890	2004.0	Color	74.0	89.0	10.0	811.0	

5 rows × 21 columns

In [12]:
```
x_trainset['USA_Country'].value_counts()
```

Out[12]:
```
1    1273
0     336
Name: USA_Country, dtype: int64
```

Most movies are not from the USA.

In [13]:
```
x_trainset.shape
```

Out[13]:
```
(1609, 21)
```

I will merge the x_trainset datafram with the y_trainset dataframe because later, I will drop the duplicated rows and do the correlation analysis.

In [14]:
```
train_joined_df = x_trainset.merge(y_train, left_index=True, right_index=True)
train_joined_df.head()
```

Out[14]:

	Movie_Id	Year_of_release	color	num_critic_for_reviews	duration	director_facebook_likes	actor_1_facebook_likes	actor_
1612	11969	1980.0	Color	212.0	89.0	0.0	2000.0	
1463	10846	2000.0	Color	57.0	135.0	0.0	1000.0	
2268	17491	2002.0	Color	39.0	99.0	13.0	1000.0	
550	3881	2004.0	Black and White	88.0	90.0	56.0	126.0	
552	3890	2004.0	Color	74.0	89.0	10.0	811.0	

5 rows × 22 columns

Let's do the same for the test dataset.

In [15]:
```
#Calculate the ROI to see whether the movie was bankable, by dividing the gross revenue by the budget
x_testset['ROI'] = x_testset['gross']/x_testset['budget']

# Calculate a famous actor/actress radar by averaging the three main actors facebook likes
# mean is calculated row-wise and for the selected columns, while skipping the null and
# missing values.
x_testset['celebrity_radar'] = x_testset[['actor_1_facebook_likes','actor_2_facebook_likes',
                                          'actor_3_facebook_likes']].mean(axis = 1, skipna=True )

# If the country is USA, use 1 and if it is not the case, use 0
x_testset['USA_Country'] = [1 if x == 'USA' else 0 for x in x_testset['country']]

# If the content category is not R, PG, PG-13, then put it in a other category
x_testset['Content_Rating_Cat'] = [ x if x=='R' or x=='PG' or x=='PG-13' else 'Other' for x
                                   in x_testset['content_rating']]

#We want to delete the columns, hence axis = 1 and directly in the dataframe, hence inplace=True
x_testset.drop(["country", "language", "content_rating"], axis=1, inplace=True)

test_joined_df = x_testset.merge(y_test, left_index=True, right_index=True)
```

3.2 Handling Outliers

In the group part, we identified several variables with outliers thanks to the descriptive statistics.

I will use the code created in the group part to remove them in the training and test datasets, in order to reduce the bias in my prediction models.

In [16]:
```
from sklearn.base import TransformerMixin, BaseEstimator

class OutlierTransformer(TransformerMixin, BaseEstimator):
  """
  This class transforms all outliers into np.NaN, then the NaNs will be imputed later
  The definition of an outlier is a value smaller than quantile1-(1.5*IQR_Value) or
  larger than quantile3+(1.5*IQR_Value)
  Note : the fit_transform method is included in the TransformerMixin super class
  """
  def __init__(self):
    self.fitted = False
    return None

  def fit(self, X):
    """
    Compute the upper and lower bound to be used later.
    Parameters
    ----------
    X : {array-like, sparse matrix}, shape [n_samples, n_features]
        The data used to compute the upper and lower bound
        used for later scaling along the features axis.
    """
    #Select and handle outliers for only the columns containing numerical variables
    numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
    numeric_X = X.select_dtypes(include=numerics)
    # Get Quantiles and IQR
    self.quantiles = pd.DataFrame(numeric_X.quantile([0.25, 0.75]))
    self.quantiles.loc['IQR',:] = self.quantiles.loc[0.75,:] - self.quantiles.loc[0.25,:]
    #Calculate Upper and Lower Bounds
    #These bounds are calculated on the Train Dataset, then applied to the train and test datasets
    self.quantiles.loc['Lower_Bound',:] = self.quantiles.loc[0.25,:] - 1.5 * self.quantiles.loc['IQR'
,:]
    self.quantiles.loc['Upper_Bound',:] = self.quantiles.loc[0.75,:] + 1.5 * self.quantiles.loc['IQR'
,:]

    #Save fitting status
    self.fitted = True

    return self

  def transform(self, X):
      """
      Replaces Outliers with NaNs
      Parameters
      ----------
      X : array-like, shape [n_samples, n_features]
          The data used to scale along the features axis.
      """
      if self.fitted == False:
        print('The transformer must be fitted before transformation.')
        return None

      #Make a copy of X to avoid mutating X
      X_copy = X.copy()

      #Old number of NaNs to count number of outliers
      old_num_na = sum(X_copy.isna().sum())

      for col in self.quantiles.columns:
        #Fill Beyond Lower Bound
        X_copy.loc[X_copy.loc[:,col] < self.quantiles.loc['Lower_Bound',col],col] = np.NaN

        #Fill Beyond Upper Bound
        X_copy.loc[X_copy.loc[:,col] > self.quantiles.loc['Upper_Bound',col],col] = np.NaN

      #Print Number of Outliers
      print(str(sum(X_copy.isna().sum()) - old_num_na) + ' Outliers are Identified.')

      return X_copy
```

```
In [17]: # the above function is called then fitted on the train dataframe
         # and transforms the train dataframe

         outlier_transformer = OutlierTransformer()
         train_new = outlier_transformer.fit_transform(train_joined_df)

         # All the outliers were transformed as NaN
```

```
2178 Outliers are Identified.
```

```
In [18]: # check the number of NaN for each column
         train_new.isnull().sum()
```

```
Out[18]: Movie_Id                    0
         Year_of_release           137
         color                       0
         num_critic_for_reviews     74
         duration                   67
         director_facebook_likes   155
         actor_1_facebook_likes     52
         actor_2_facebook_likes    194
         actor_3_facebook_likes     51
         cast_total_facebook_likes  98
         movie_facebook_likes      264
         gross                     264
         num_voted_users           137
         facenumber_in_poster       44
         num_user_for_reviews      132
         budget                    141
         imdb_score                 32
         ROI                       359
         celebrity_radar           101
         USA_Country               336
         Content_Rating_Cat          0
         avg_rating                 19
         dtype: int64
```

We will use the imputer fitted on the training dataset to transform the outliers in the test dataset into NaNs.

```
In [19]: #Transform the outliers in the test set into NaNs

         test_new = outlier_transformer.transform(test_joined_df)
```

```
947 Outliers are Identified.
```

```
In [20]: # Check the number of NaNs in the test set per column
         test_new.isnull().sum()
```

```
Out[20]: Movie_Id                    0
         Year_of_release            51
         color                       0
         num_critic_for_reviews     23
         duration                   24
         director_facebook_likes    81
         actor_1_facebook_likes     21
         actor_2_facebook_likes     84
         actor_3_facebook_likes     21
         cast_total_facebook_likes  40
         movie_facebook_likes      123
         gross                     110
         num_voted_users            63
         facenumber_in_poster       20
         num_user_for_reviews       51
         budget                     59
         imdb_score                 13
         ROI                       156
         celebrity_radar            42
         USA_Country               153
         Content_Rating_Cat          0
         avg_rating                 10
         dtype: int64
```

3.3 Handling Missing Values

Let's impute the missing values. First, I will check again the variables types.

```
In [21]: train_new.dtypes
```

```
Out[21]: Movie_Id                     float64
         Year_of_release              float64
         color                         object
         num_critic_for_reviews       float64
         duration                     float64
         director_facebook_likes      float64
         actor_1_facebook_likes       float64
         actor_2_facebook_likes       float64
         actor_3_facebook_likes       float64
         cast_total_facebook_likes    float64
         movie_facebook_likes         float64
         gross                        float64
         num_voted_users              float64
         facenumber_in_poster         float64
         num_user_for_reviews         float64
         budget                       float64
         imdb_score                   float64
         ROI                          float64
         celebrity_radar              float64
         USA_Country                  float64
         Content_Rating_Cat            object
         avg_rating                   float64
         dtype: object
```

Since we have categorical and numerical values, we need to split them into two dataframes and then, impute them separately. Because we have only two categorical values, we can put them into a separate dataframe and remove them from the numerical dataframe.

```
In [22]: from sklearn.impute import SimpleImputer

         imputer = SimpleImputer(strategy="median")

         # we create a temporary dataframe with only numerical columns,
         # but keep the categorical columns in a separate dataframe
         trainset_categorical = train_new.loc[:,["Movie_Id","color","Content_Rating_Cat"]]
         trainset_numerical = train_new.drop(["color","Content_Rating_Cat"], axis=1)
```

```
In [23]: # Let's verify if the transformation has worked
         print(train_new.shape)
         print(trainset_categorical.shape)
         print(trainset_numerical.shape)
```

```
         (1609, 22)
         (1609, 3)
         (1609, 20)
```

```
In [24]: #fit and transform on the train numerical dataset
         # we create a temporary dataframe with only numerical columns,
         # but keep the categorical columns in a separate dataframe
         trainset_categorical = train_new.loc[:,["Movie_Id","color","Content_Rating_Cat"]]
         trainset_numerical = train_new.drop(["color","Content_Rating_Cat"], axis=1)
         x_train_numerical=pd.DataFrame(imputer.fit_transform(trainset_numerical),
                                        columns = trainset_numerical.columns)
```

In [25]:
```
# Check the number of NaN value
x_train_numerical.isnull().sum()
```

Out[25]:
```
Movie_Id                    0
Year_of_release             0
num_critic_for_reviews      0
duration                    0
director_facebook_likes     0
actor_1_facebook_likes      0
actor_2_facebook_likes      0
actor_3_facebook_likes      0
cast_total_facebook_likes   0
movie_facebook_likes        0
gross                       0
num_voted_users             0
facenumber_in_poster        0
num_user_for_reviews        0
budget                      0
imdb_score                  0
ROI                         0
celebrity_radar             0
USA_Country                 0
avg_rating                  0
dtype: int64
```

In [26]:
```
# I fill the missing values with the most frequent values for each columns
x_train_categorical = trainset_categorical.apply(lambda x:x.fillna(x.value_counts().index[0]))

# I verify the number of NaN
x_train_categorical.isnull().sum()
```

Out[26]:
```
Movie_Id              0
color                 0
Content_Rating_Cat    0
dtype: int64
```

I will do the same thing for the test dataset.

In [27]:
```
# we create a temporary dataframe with only numerical columns,
# but keep the categorical columns in a separate variable
testset_categorical = test_new.loc[:,["Movie_Id","color","Content_Rating_Cat"]]
testset_numerical = test_new.drop(["color","Content_Rating_Cat"], axis=1)

# We transform the numerical test dataset using the transformer fitted on the train dataset
x_test_numerical=pd.DataFrame(imputer.transform(testset_numerical),
                              columns = testset_numerical.columns)

# I filled the missing values with the most frequent values for each columns
x_test_categorical = testset_categorical.apply(lambda x:x.fillna(x.value_counts().index[0]))

# I will verify if the imputation of missing values worked on the categorical dataset.
x_test_categorical.isnull().sum()
```

Out[27]:
```
Movie_Id              0
color                 0
Content_Rating_Cat    0
dtype: int64
```

In [28]:
```
# Check the missing values for the numerical dataset.
x_test_numerical.isnull().sum()
```

Out[28]:
```
Movie_Id                    0
Year_of_release             0
num_critic_for_reviews      0
duration                    0
director_facebook_likes     0
actor_1_facebook_likes      0
actor_2_facebook_likes      0
actor_3_facebook_likes      0
cast_total_facebook_likes   0
movie_facebook_likes        0
gross                       0
num_voted_users             0
facenumber_in_poster        0
num_user_for_reviews        0
budget                      0
imdb_score                  0
ROI                         0
celebrity_radar             0
USA_Country                 0
avg_rating                  0
dtype: int64
```

3.4 Dummy Variables

Before looking at the correlations between the variables, we have first to create dummy variables for the categorical values. I will use the OneHotEncoder method from scikit_learn.

In [29]:
```
from sklearn.preprocessing import OneHotEncoder

# create two variables OneHotEncoder for each categorical variable
color_ohe = OneHotEncoder(drop="first", sparse=False)
content_rating_ohe = OneHotEncoder(drop="first", sparse=False)

# the input to the encoder must be a 2-d numpy array,
# so we take the column, extract their values and reshape the array to be 2-d
color_transf=color_ohe.fit_transform(x_train_categorical['color'].values.reshape(-1,1))
content_r_transf= content_rating_ohe.fit_transform(x_train_categorical['Content_Rating_Cat'].
                                                   values.reshape(-1,1))

# put the transformed data as columns in the dataframe
col_names = color_ohe.categories_[0].tolist()[1:]
content_rating_names = content_rating_ohe.categories_[0].tolist()[1:]

for i, col_name in enumerate(col_names):
    x_train_categorical[col_name] = color_transf[:,i]

for i, content_rating_name in enumerate(content_rating_names):
    x_train_categorical[content_rating_name] = content_r_transf[:,i]

# delete the categorical columns
x_train_categorical.drop(['color','Content_Rating_Cat'], axis=1, inplace=True)

# check if the dummies were correctly created
x_train_categorical.head()
```

Out[29]:

	Movie_Id	Color	PG	PG-13	R
1612	11969.0	1.0	0.0	0.0	1.0
1463	10846.0	1.0	0.0	0.0	1.0
2268	17491.0	1.0	1.0	0.0	0.0
550	3881.0	0.0	1.0	0.0	0.0
552	3890.0	1.0	1.0	0.0	0.0

Now, we will create dummies in the test set using the fitted encoders we built above.

In [30]:
```
color_transf_test= x_test_categorical['color'].values.reshape(-1,1)
content_r_transf_test= x_test_categorical['Content_Rating_Cat'].values.reshape(-1,1)

transf_color=color_ohe.transform(color_transf_test)
transf_content=content_rating_ohe.transform(content_r_transf_test)
```

In [31]:
```
# put the transformed data as columns in the dataframe

for i, col_name in enumerate(col_names):
    x_test_categorical[col_name] = transf_color[:,i]

for i, content_rating_name in enumerate(content_rating_names):
    x_test_categorical[content_rating_name] = transf_content[:,i]

# delete the categorical columns
x_test_categorical.drop(['color','Content_Rating_Cat'], axis=1, inplace=True)

# check if the creation of the dummies worked
x_test_categorical.shape
```

Out[31]:
```
(690, 5)
```

3.5 Merging the numerical and categorical Dataframes

In [32]:
```
train_merged = x_train_numerical.merge(x_train_categorical, on='Movie_Id',how='inner' )
train_merged.head()
```

Out[32]:

	Movie_Id	Year_of_release	num_critic_for_reviews	duration	director_facebook_likes	actor_1_facebook_likes	actor_2_faceboo
0	11969.0	1980.0	212.0	89.0	0.0	2000.0	
1	11969.0	1980.0	212.0	89.0	0.0	2000.0	
2	11969.0	1980.0	212.0	89.0	0.0	2000.0	
3	11969.0	1980.0	212.0	89.0	0.0	2000.0	
4	10846.0	2000.0	57.0	135.0	0.0	1000.0	

5 rows × 24 columns

In [33]:
```
print(x_train_categorical.shape)
print(x_train_numerical.shape)
print(train_merged.shape)
```

```
(1609, 5)
(1609, 20)
(1669, 24)
```

I can see above that I have duplicated rows in my final merged dataset. I need to drop the duplicated rows.

In [34]:
```
train_merged['Movie_Id'].duplicated().sum()
```

Out[34]:
```
89
```

In [35]:
```
#dropping rows
train_merged.drop_duplicates(keep = 'first', inplace = True)
```

In [36]:
```
# Have a look the datset
train_merged.head(10)
```

Out[36]:

	Movie_Id	Year_of_release	num_critic_for_reviews	duration	director_facebook_likes	actor_1_facebook_likes	actor_2_faceboo
0	11969.0	1980.0	212.0	89.0	0.0	2000.0	
2	11969.0	1980.0	212.0	89.0	0.0	2000.0	
4	10846.0	2000.0	57.0	135.0	0.0	1000.0	
5	17491.0	2002.0	39.0	99.0	13.0	1000.0	
6	3881.0	2004.0	88.0	90.0	56.0	126.0	
7	3890.0	2004.0	74.0	89.0	10.0	811.0	
8	5055.0	1998.0	65.0	114.0	0.0	970.0	
9	385.0	2002.0	80.0	104.0	54.0	901.0	
10	6844.0	2001.0	187.0	106.0	357.0	23000.0	
14	9863.0	2000.0	170.0	123.0	55.0	3000.0	

10 rows × 24 columns

I still have duplicated movies. In my opinion, there were no duplicated movies in the Netflix dataset because we verified this. But the imdb database may not be as complete as the Netflix dataset. For example, for two different movies with the same name, we may have attributed the same data, even though they had different release years. I will drop duplicated rows if they have the same movie id, the same release year and the same duration. Thus, I assume that there cannot be two or more movies released the same year with the same name and the same duration.

In [37]:
```
#dropping rows
train_merged.drop_duplicates(subset = ["Movie_Id", "Year_of_release","duration"], inplace = True)
print(train_merged.shape)
```

```
(1580, 24)
```

We do the same thing for the test dataset.

In [38]:
```
#we merge the numerical and categorical dataframes
test_merged= x_test_numerical.merge(x_test_categorical, on='Movie_Id',how='inner' )

#dropping rows
test_merged.drop_duplicates(keep = 'first', inplace = True)
#dropping rows
test_merged.drop_duplicates(subset = ["Movie_Id", "Year_of_release","duration"], inplace = True)

print(test_merged.shape)
```

```
(687, 24)
```

3.6 Variable Correlations

We would like to know which variables have a correlation with our target variable "avg_rating". We will use Pearson's r correlation and sort the correlation coefficients from the strongest one to the weakest.

```
In [39]: corr_matrix=train_merged.corr(method='pearson')
         corr_matrix["avg_rating"].sort_values(ascending=False)
```

```
Out[39]: avg_rating                  1.000000
         imdb_score                  0.609796
         num_voted_users             0.283427
         duration                    0.272726
         num_user_for_reviews        0.238131
         gross                       0.235021
         ROI                         0.232137
         num_critic_for_reviews      0.215250
         actor_1_facebook_likes      0.116838
         PG                          0.095737
         cast_total_facebook_likes   0.086092
         celebrity_radar             0.084381
         director_facebook_likes     0.071425
         Movie_Id                    0.063990
         actor_3_facebook_likes      0.033522
         budget                      0.006502
         actor_2_facebook_likes     -0.005851
         facenumber_in_poster       -0.030576
         PG-13                      -0.049197
         R                          -0.086117
         Color                      -0.101330
         Year_of_release            -0.141006
         movie_facebook_likes       -0.214785
         USA_Country                      NaN
         Name: avg_rating, dtype: float64
```

The variable with the strongest correlation is "imdb_Score". The higher the imdb score of the movie is, the higher its average rating will be. Another correlated variable is the number of facebook likes. But this one has a negative correlation. The more movie facebook likes the movie will have, the lower its average score will be.

I have NaN values for USA_country. Let's see why.

```
In [40]: train_merged['USA_Country'].value_counts()
```

```
Out[40]: 1.0    1580
         Name: USA_Country, dtype: int64
```

When we deleted the duplicates, we also got rid of all movies which were not from the USA. Thus, we can remove this variable.

```
In [41]: train_merged.drop(['USA_Country'], axis=1, inplace=True)
```

Let's have a look again at the correlation between our independent variables and the target variable.

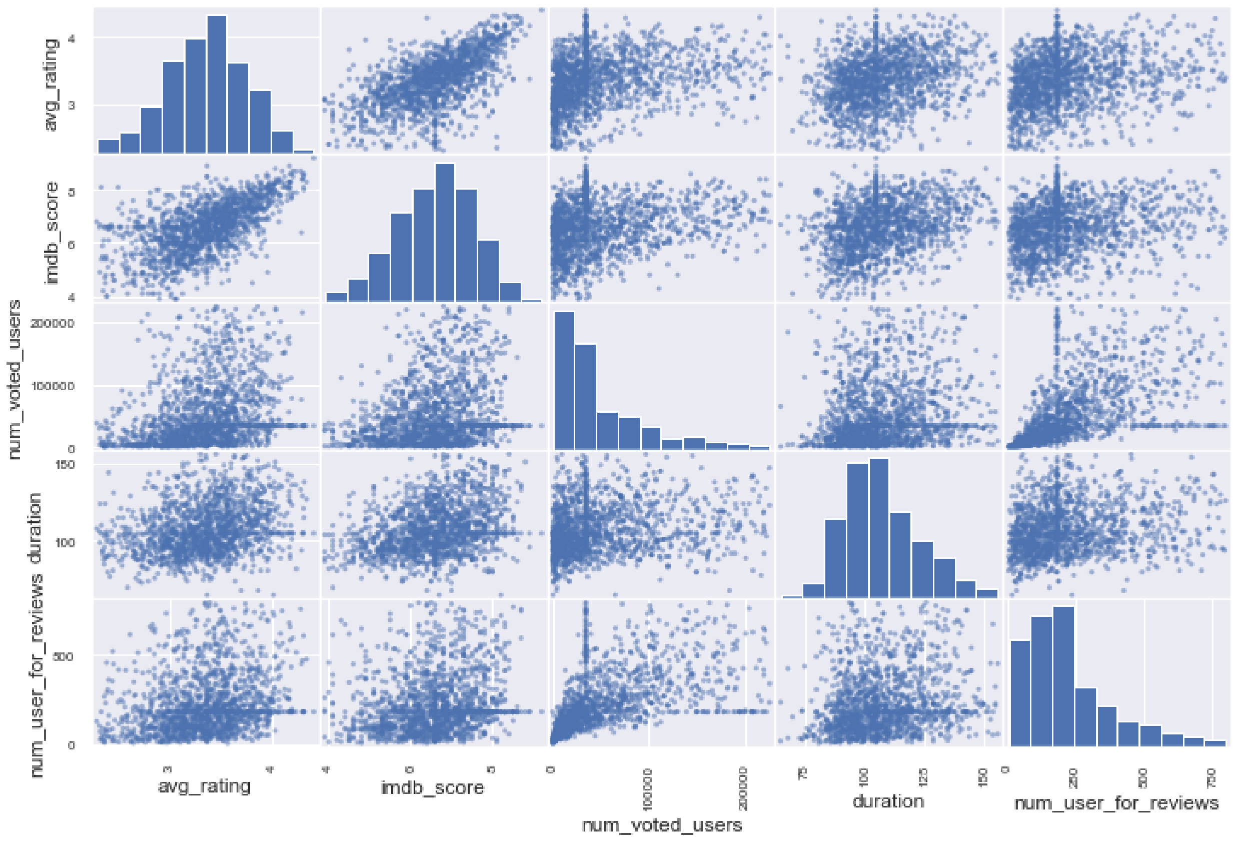
In [42]:
```
corr_matrix2=train_merged.corr(method='pearson')
corr_matrix2["avg_rating"].sort_values(ascending=False)
```

Out[42]:
```
avg_rating                 1.000000
imdb_score                 0.609796
num_voted_users            0.283427
duration                   0.272726
num_user_for_reviews       0.238131
gross                      0.235021
ROI                        0.232137
num_critic_for_reviews     0.215250
actor_1_facebook_likes     0.116838
PG                         0.095737
cast_total_facebook_likes  0.086092
celebrity_radar            0.084381
director_facebook_likes    0.071425
Movie_Id                   0.063990
actor_3_facebook_likes     0.033522
budget                     0.006502
actor_2_facebook_likes    -0.005851
facenumber_in_poster      -0.030576
PG-13                     -0.049197
R                         -0.086117
Color                     -0.101330
Year_of_release           -0.141006
movie_facebook_likes      -0.214785
Name: avg_rating, dtype: float64
```

Scatter matrix

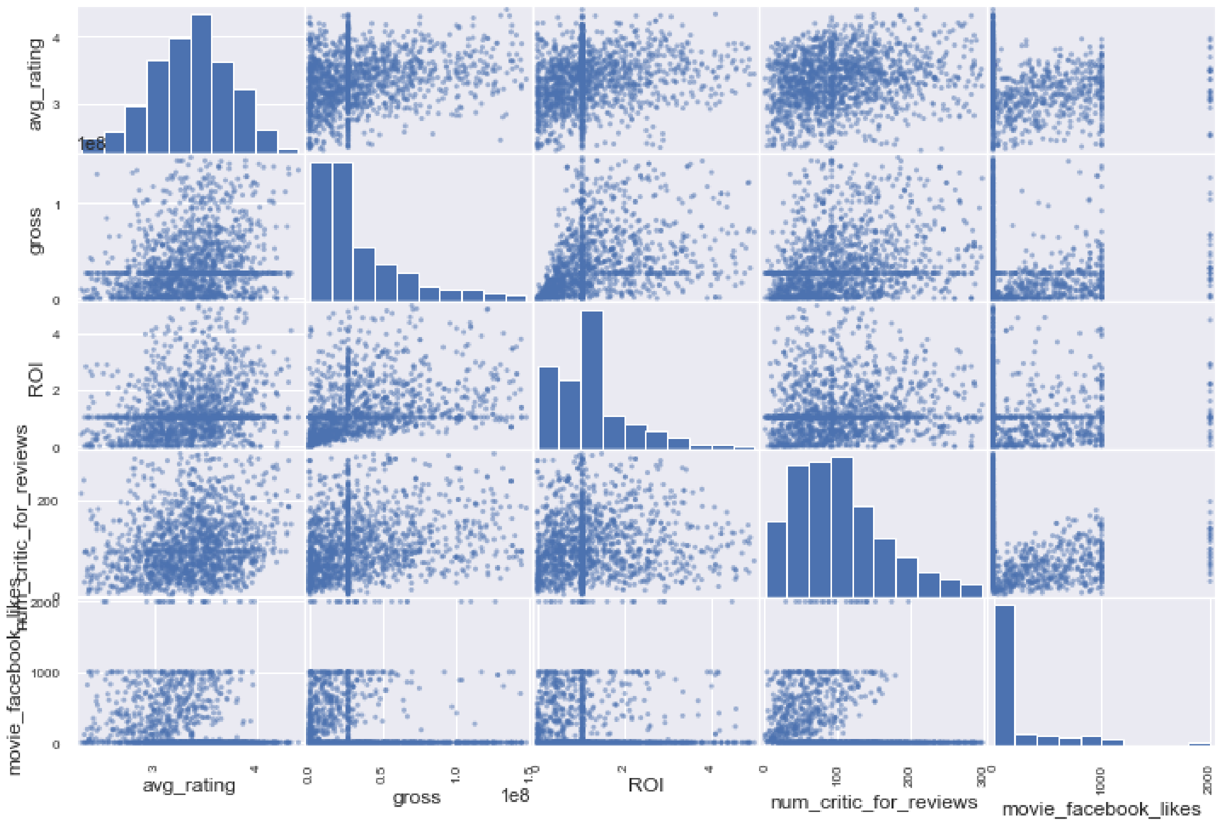
I will create a scatter matrix to visualise the correlations between the target variable and the main independent variables, that the target variable is correlated with.

In [43]:
```
from pandas.plotting import scatter_matrix

# select only the variables that are most likely to be correlated
attributes = ["avg_rating", "imdb_score", "num_voted_users", "duration","num_user_for_reviews"]
dummy = scatter_matrix(train_merged[attributes], figsize=(12, 8))
```



We can see there is a strong positive correlation between the imdb_score and the avergae rating.There may be also relationships between the average rating and the number of voted users and users for reviews. We can also see there is a correlation between the number of voted users and the number of users for reviews. It implies that the users who reviewed the movies also voted for the movies.

In [44]:
```
attributes2 = ["avg_rating", "gross", "ROI","num_critic_for_reviews", "movie_facebook_likes"]
# select only the variables that are most likely to be correlated
dummy = scatter_matrix(train_merged[attributes2], figsize=(12, 8))
```



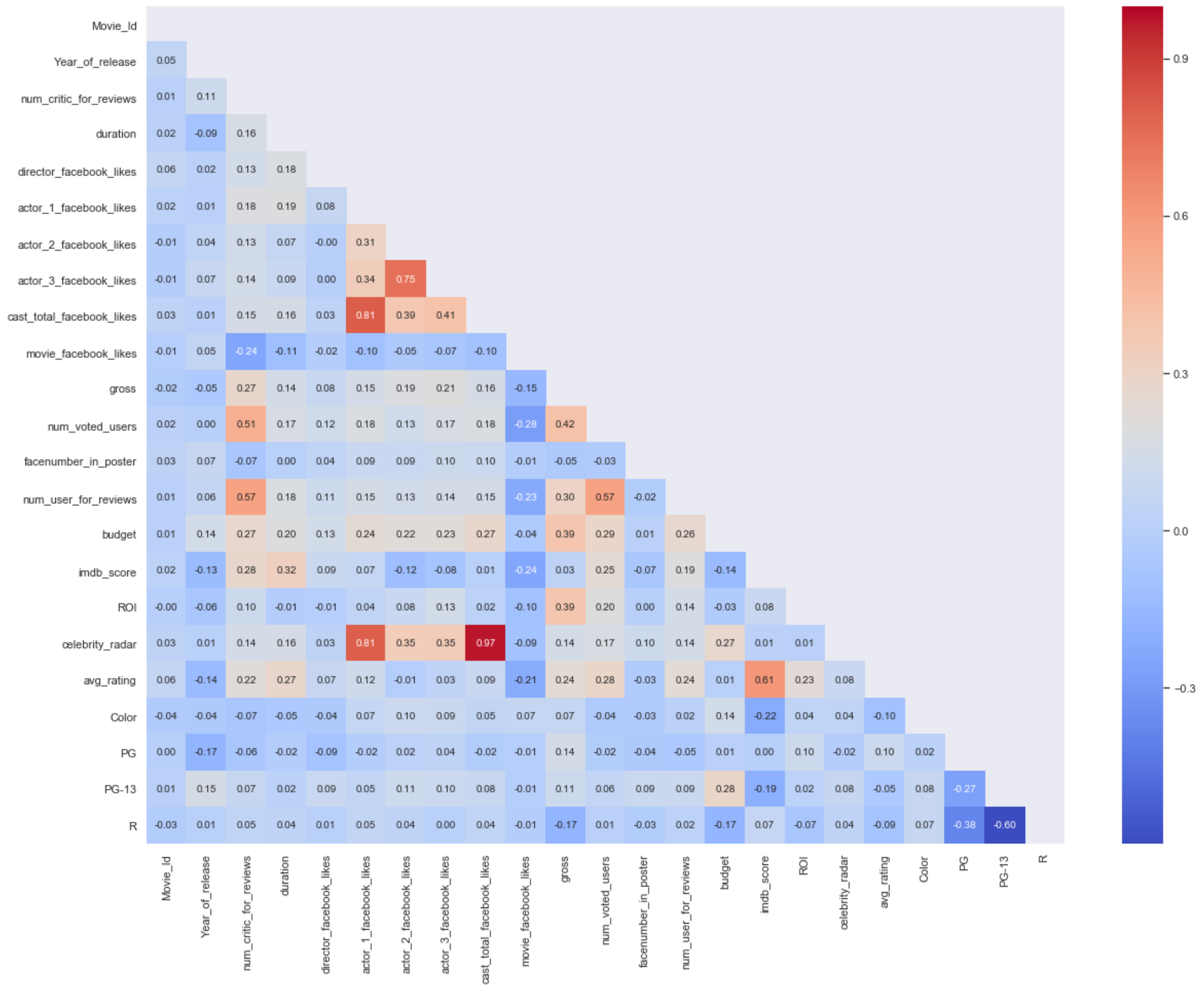
There seems to be a relationship between the ROI and the average rating. The ROI is correlated to the gross revenue because the ROI was calculated using it.

Correlation Matrix

We will create a correlation matrix to view the possible correlations between each variable.

In [45]:
```python
plt.figure(figsize = (20,15))
matrix2= np.triu(corr_matrix2)
sns.heatmap(corr_matrix2, annot = True,cmap= 'coolwarm', fmt='.2f', mask=matrix2)
```

Out[45]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x254cae01940>
```



From the correlation matrix above, we can see the biggest correlations between the following variables:

- celebrity radar and *cast total's Facebook likes* (0.97),
- number of actor 1's likes and cast total's likes (0.81), because the lead actor is more well-known, its Facebook likes account for most of the cast total's Facebook likes.
- number of actor 1's likes and celebrity radar (0.81). That is also due to the fact that the celebrity radar and the cast's likes are correlated.
- actor 3's likes and *actor's 2 likes* (0.75), which implies that movie directors and casting directors try to choose the second and third lead actors with the same level of prestige to avoid ego and money conflicts.
- between the number of critic for reviews and number of users for reviews (0.57).
- number of users for reviews and number of voted users (0.57). That is logical because if someone takes the time to write a review, this person will also have the time to rate the movie.
- the number of voted users and number of critic for reviews (0.51)
- number of voted users and gross (0.42)
- budget and gross (0.39)
- ROI and gross (0.39)
- imdb score and rating (0.61), as seen above

We can also see that the following variables have a correlation coefficient superior or equal to 0.10 in absolute value with avg_rating:

- year of release
- num critic for reviews
- duration
- actor 1 likes
- cast total likes
- movie fb likes
- gross
- num voted users
- num user for reviews
- imdb score
- ROI
- color
- PG

Since the actor's likes has a higher correlation with the average rating. I will keep it, unlike the cast's number of likes.

I will create a new dataframe with only the columns I need.

In [46]:
```
train_merged.dtypes
```

Out[46]:
```
Movie_Id                    float64
Year_of_release             float64
num_critic_for_reviews      float64
duration                    float64
director_facebook_likes     float64
actor_1_facebook_likes      float64
actor_2_facebook_likes      float64
actor_3_facebook_likes      float64
cast_total_facebook_likes   float64
movie_facebook_likes        float64
gross                       float64
num_voted_users             float64
facenumber_in_poster        float64
num_user_for_reviews        float64
budget                      float64
imdb_score                  float64
ROI                         float64
celebrity_radar             float64
avg_rating                  float64
Color                       float64
PG                          float64
PG-13                       float64
R                           float64
dtype: object
```

In [47]:
```
train_merged_new=train_merged.loc[:,['Movie_Id','avg_rating','Year_of_release','num_critic_for_reviews'
,
                                     'num_voted_users','num_user_for_reviews', 'duration',
                                     'actor_1_facebook_likes', 'cast_total_facebook_likes',
                                     'movie_facebook_likes', 'imdb_score', 'gross', 'ROI',
                                     'Color', 'PG']]
```

In [48]:
```
train_merged_new.head()
```

Out[48]:

	Movie_Id	avg_rating	Year_of_release	num_critic_for_reviews	num_voted_users	num_user_for_reviews	duration	actor_1_fac
0	11969.0	3.308080	1980.0	212.0	46492.0	335.0	89.0	
4	10846.0	3.078547	2000.0	57.0	8720.0	186.0	135.0	
5	17491.0	3.357953	2002.0	39.0	15074.0	72.0	99.0	
6	3881.0	2.927778	2004.0	88.0	4117.0	118.0	90.0	
7	3890.0	2.942760	2004.0	74.0	23408.0	105.0	89.0	

Let us do the same for the test set.

In [49]:
```
test_merged_new=test_merged.loc[:,['Movie_Id','avg_rating','Year_of_release','num_critic_for_reviews',
                                   'num_voted_users','num_user_for_reviews', 'duration',
                                   'actor_1_facebook_likes', 'cast_total_facebook_likes',
                                  'movie_facebook_likes', 'imdb_score', 'gross', 'ROI',
                                   'Color', 'PG']]
```

In [50]:
```
test_merged_new.shape
```

Out[50]:
```
(687, 15)
```

3.7 Scaling

Since our independent variables have different scales, as seen in the descriptive statistics in the group part, I will **standardize** them except for the movie_id, the year and the duration. It may improve the performances of my models.

I will also keep the original values for the target variable.

We will fit and transform the scaler on the training set. Then, use it to transform the test dataset.

In [51]:
```
# import the standard scaler
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# we remove the columns that we do not want to standardize
trainset_others= train_merged_new.loc[:,['Movie_Id', 'duration', 'Year_of_release',"avg_rating"]]
trainset_predictors = train_merged_new.drop(["avg_rating",'Movie_Id', 'duration', 'Year_of_release'], a
xis=1)

# fit_transform returns a NumPy array, so we need to put it back
# into a Pandas dataframe
scaled_vals = scaler.fit_transform(trainset_predictors)
train_merged_new = pd.DataFrame(scaled_vals, columns=trainset_predictors.columns)

# put the non-scaled columns back in
train_merged_df= train_merged_new.merge(trainset_others, left_index=True, right_index=True)

# inspect the data
train_merged_df.head()
```

Out[51]:

	num_critic_for_reviews	num_voted_users	num_user_for_reviews	actor_1_facebook_likes	cast_total_facebook_likes	movie_fac
0	1.703648	-0.086487	0.674234	-0.443974	-0.228683	
4	-0.460616	-0.543828	-0.700642	-0.620243	-0.592960	
5	-0.601763	-0.631694	-0.359912	-0.596672	-0.499807	
6	-0.366517	-0.322865	-0.634887	-0.606901	-0.176528	
7	1.311571	-0.325440	-0.246335	2.669280	-0.469428	

Now, we will use the fitted scaler on the test data.

In [52]:
```
testset_others = test_merged_new.loc[:,['Movie_Id', 'duration', 'Year_of_release', 'avg_rating']]
testset_predictors = test_merged_new.drop(['Movie_Id', 'duration', 'Year_of_release', 'avg_rating'], ax
is=1)

# we use the transformer fitted on the train dataset
scaled_vals = scaler.transform(testset_predictors)
test_merged_new = pd.DataFrame(scaled_vals, columns=testset_predictors.columns)

# put the non-scaled columns back in
test_merged_df= test_merged_new.merge(testset_others, left_index=True, right_index=True)

# inspect the data
test_merged_df.head()
```

Out[52]:

	num_critic_for_reviews	num_voted_users	num_user_for_reviews	actor_1_facebook_likes	cast_total_facebook_likes	movie_fac
0	-0.350834	-0.762592	-0.425667	-0.647966	-0.721734	
1	-0.366517	1.031722	0.046573	-0.443974	-0.233387	
2	-1.009523	-0.955363	-1.113105	-0.687697	-0.772544	
3	-0.680179	-0.516448	-0.294157	0.149027	0.414515	
4	1.217473	-0.325440	2.120843	-0.592224	-0.469428	

4.0 Model Building

In this part, I will create five models, train them on the training set and compare their results before picking the best ones to evaluate on the test set. I will use the following algorithms:

1. Linear Regression
2. Decision Trees
3. Random Forest
4. Support Vector Regression
5. AdaBoost

I picked those models because I was curious about their differences in terms of performances and complexity.

First, we need to create two dataframes, one for the predictors (Xtrain) and another one for the target variable (ytrain)

In [53]:
```
# drop the predictor column for the training set, but keep the other columns
Xtrain = train_merged_df.drop("avg_rating", axis=1)

ytrain = train_merged_df["avg_rating"].copy()
```

In [54]:
```
print (Xtrain.shape)
print (ytrain.shape)
```

```
(1491, 14)
(1491,)
```

We will do the same for the test set.

In [55]:
```
# drop the predictor column for the test set, but keep the other columns
Xtest = test_merged_df.drop("avg_rating", axis=1)

ytest = test_merged_df["avg_rating"].copy()
```

In [56]:
```
print (Xtest.shape)
print (ytest.shape)
```

```
(678, 14)
(678,)
```

4.1 Baseline

We will use as a baseline the median value in the training dataset.

In [57]:
```
median_rating = ytrain.median()
median_rating
```

Out[57]:
```
3.3657956572116365
```

To measure the regression accuracy of each model and compare each model against each other, we will use the Root Mean Squared Error (RMSE).

In [58]:
```
from sklearn.metrics import mean_squared_error

# each row of yhat will contain the median rating
yhat = np.full((ytrain.shape[0], 1), median_rating)

baseline_mse = mean_squared_error(ytrain, yhat)

# take square root
baseline_rmse = np.sqrt(baseline_mse)

baseline_rmse
```

Out[58]:
```
0.39529919697271415
```

In [59]:
```
#creation of a function to display the RMSE scores
def display_scores(scores):
    print("Scores:", scores)
    print("Accuracy:", scores.mean())
    print("Standard deviation:", scores.std())
```

4.2 Training and Evaluating on the Training data

4.2.1 Linear Regression

In [60]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV


#call the linear regression function
lin_reg = LinearRegression()

# cross validation with 10 parts to have the average accuracy score
lin_reg_scores = cross_val_score(lin_reg, Xtrain, ytrain,
                         scoring="neg_mean_squared_error", cv=10)

# With cross-validation, we have to take the opposite of the MSE to calculate the RMSE.
rmse_lin_scores = np.sqrt(-lin_reg_scores)

display_scores(rmse_lin_scores)
```

```
Scores: [0.36532339 0.38827114 0.38816454 0.33848736 0.38884871 0.38132385
 0.3500855  0.42528398 0.39013122 0.3496958 ]
Accuracy: 0.3765615481424965
Standard deviation: 0.024499181317022143
```

The RMSE is on average lower than the baseline, which is encouraging.

In [61]:
```
#fit the linear regression to the training set
lin_reg.fit(Xtrain, ytrain)

#print the R2 score of the linear regression model
lin_reg.score(Xtrain, ytrain)
```

Out[61]:
```
0.10331776148259442
```

The R2 score for the linear regression is low and thus bad. It means that the model has a bad accuracy.

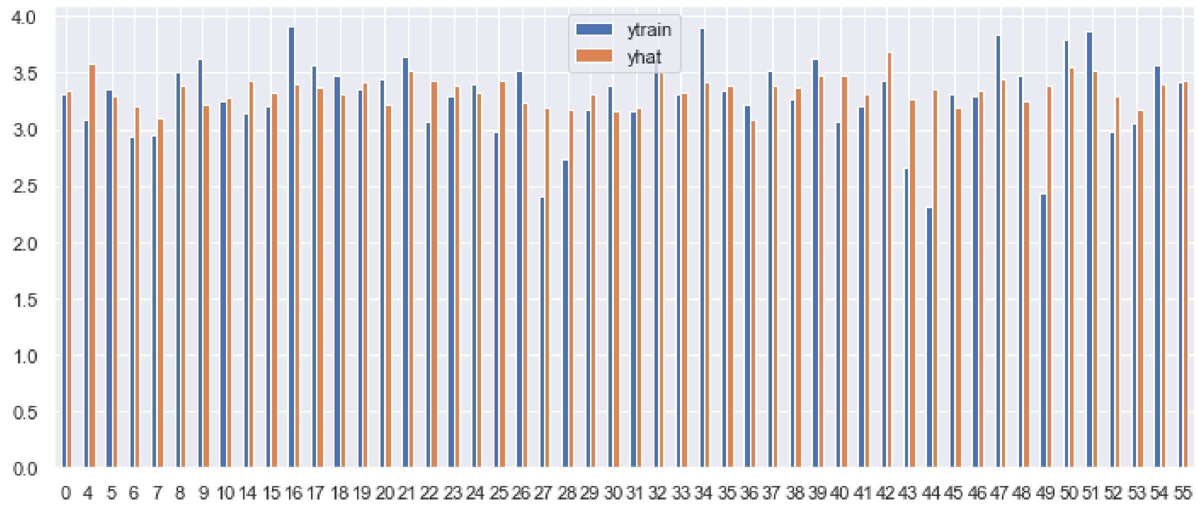
In [62]:
```
# make predictions
lin_yhat=lin_reg.predict(Xtrain)

#print rmse of the linear model
lin_rmse = np.sqrt(mean_squared_error(ytrain, lin_yhat))
lin_rmse
```

Out[62]:
```
0.3739606486126967
```

We will visualize the predictions and the real values for the first 50 instances. Ytrain represents the current values and lin_yhat the predictions.

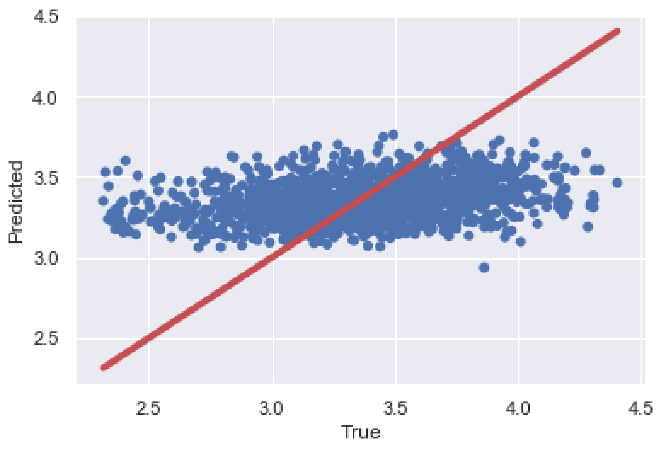
In [63]:
```
# create a temporary dataframe containing two columns
lin_df_tmp = pd.DataFrame({"ytrain": ytrain[:50], "yhat": lin_yhat[:50]})

# plot the dataframe
lin_df_tmp.plot(figsize=(12,5), kind="bar", rot=0)
```

Out[63]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x254c9032e10>
```



We will create a scatter plot to visualise the predicted values versus their real values.

In [64]:
```
fig, ax = plt.subplots()
ax.scatter(ytrain, lin_yhat, edgecolors='None')
ax.plot([ytrain.min(), ytrain.max()], [ytrain.min(), ytrain.max()], 'r', lw=4)
ax.set_xlabel('True')
ax.set_ylabel('Predicted')
plt.show()
```



If the values were correctly predicted, they would follow the red line. It is sadly not the case here. Since our R2 is low, it is not such a big surprise.

4.2.2 Decision Trees

We will tune the hyperparameters for the decision trees using GridSearch. It will enable us to find the optimal hyperparameters for the algorithm among several combinations that we will give.

In [65]:
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV


#We will search for the best hyperparameters for the decision trees, using GridSearch
# and thus cross-validation. We give here several combinations for the hyperparameters to compare.
dt_param_grid= {'min_samples_split': [2, 3, 4, 5], 'max_depth': [2, 4, 6, 8, None]}
#n_estimators: Number of trees in random forest
#max_depth: Maximum number of levels in tree
# min_samples_split: Minimum number of samples required to split a node

Dec_tree_reg = DecisionTreeRegressor()

# start a timer
dt_start=time.time()

#Cross-validation with 10 splits
# we also want it to return the train score later
dt_grid_search = GridSearchCV(Dec_tree_reg, dt_param_grid, cv=10,
                              scoring='neg_mean_squared_error', return_train_score=True)

#We fit the training data to the best model (and thus estimators)
dt_grid_search.fit(Xtrain, ytrain)
```

Out[65]:
```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=DecisionTreeRegressor(criterion='mse', max_depth=None,
                                             max_features=None,
                                             max_leaf_nodes=None,
                                             min_impurity_decrease=0.0,
                                             min_impurity_split=None,
                                             min_samples_leaf=1,
                                             min_samples_split=2,
                                             min_weight_fraction_leaf=0.0,
                                             presort=False, random_state=None,
                                             splitter='best'),
             iid='warn', n_jobs=None,
             param_grid={'max_depth': [2, 4, 6, 8, None],
                         'min_samples_split': [2, 3, 4, 5]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_squared_error', verbose=0)
```

In [66]:
```
# details on the best model for the decision tree algorithm
best_dt=dt_grid_search.best_estimator_
best_dt
```

Out[66]:
```
DecisionTreeRegressor(criterion='mse', max_depth=2, max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=3, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=None, splitter='best')
```

The best decision tree model has a max_depth=2 and min_samples_split=2.

Use of cross-validation to find the average accuracy score for this model.

In [67]:
```
dt_cv_scores= cross_val_score(best_dt, Xtrain, ytrain, scoring="neg_mean_squared_error", cv=10)

dt_cv_rmse_scores = np.sqrt(-dt_cv_scores)
display_scores(dt_cv_rmse_scores)
```

```
Scores: [0.37386173 0.38605892 0.39303714 0.34206869 0.38361367 0.38536461
 0.35223175 0.42367925 0.38917399 0.34939939]
Accuracy: 0.3778489145170766
Standard deviation: 0.02319098553768898
```

On average, the decision tree model has a lower and better RMSE than the baseline but a slightly lower accuracy than the linear regression.

In [68]:
```
# training and validation RMSE

dt_val_scores = dt_grid_search.cv_results_["mean_test_score"]
dt_train_scores = dt_grid_search.cv_results_["mean_train_score"]
dt_params = dt_grid_search.cv_results_["params"]

for dt_val_score, dt_train_score, dt_param in zip(dt_val_scores, dt_train_scores, dt_params):
    print(np.sqrt(-dt_val_score), np.sqrt(-dt_train_score), dt_param)
```

```
0.378556803002787 0.37445167654118855 {'max_depth': 2, 'min_samples_split': 2}
0.3785568030027869 0.37445167654118855 {'max_depth': 2, 'min_samples_split': 3}
0.378556803002787 0.37445167654118855 {'max_depth': 2, 'min_samples_split': 4}
0.378556803002787 0.37445167654118855 {'max_depth': 2, 'min_samples_split': 5}
0.3904523010043059 0.359984400751391 {'max_depth': 4, 'min_samples_split': 2}
0.3898777146172578 0.359984400751391 {'max_depth': 4, 'min_samples_split': 3}
0.3904523010043059 0.360003160021297 {'max_depth': 4, 'min_samples_split': 4}
0.3911358319854297 0.360003160021297 {'max_depth': 4, 'min_samples_split': 5}
0.4198603042568489 0.33271467492323487 {'max_depth': 6, 'min_samples_split': 2}
0.41822094102217616 0.3328075125233207 {'max_depth': 6, 'min_samples_split': 3}
0.4187735676052253 0.33293751262822174 {'max_depth': 6, 'min_samples_split': 4}
0.4186446748667805 0.33336148082060096 {'max_depth': 6, 'min_samples_split': 5}
0.45597971843929264 0.2907531607698169 {'max_depth': 8, 'min_samples_split': 2}
0.45740570300165484 0.29129363130058966 {'max_depth': 8, 'min_samples_split': 3}
0.4523109544770185 0.29218246213759014 {'max_depth': 8, 'min_samples_split': 4}
0.4521986603816638 0.2937471726961729 {'max_depth': 8, 'min_samples_split': 5}
0.5491147999752065 2.0034767885056066e-05 {'max_depth': None, 'min_samples_split': 2}
0.5460063704767049 0.04892042065847464 {'max_depth': None, 'min_samples_split': 3}
0.5419715735907679 0.07845373125725683 {'max_depth': None, 'min_samples_split': 4}
0.5388489195045926 0.10544829847848682 {'max_depth': None, 'min_samples_split': 5}
```

The RMSE on the validation set is slightly higher than the training set's.But they are quite similar. So, it is encouraging and there is little sign of overfitting (performing well on the training dataset but bad on the validation set).

In [69]:
```
dt_rmse_score=np.sqrt(-dt_grid_search.best_score_)
print(f'The best Decision Trees model has a RMSE of: {dt_rmse_score}')
```

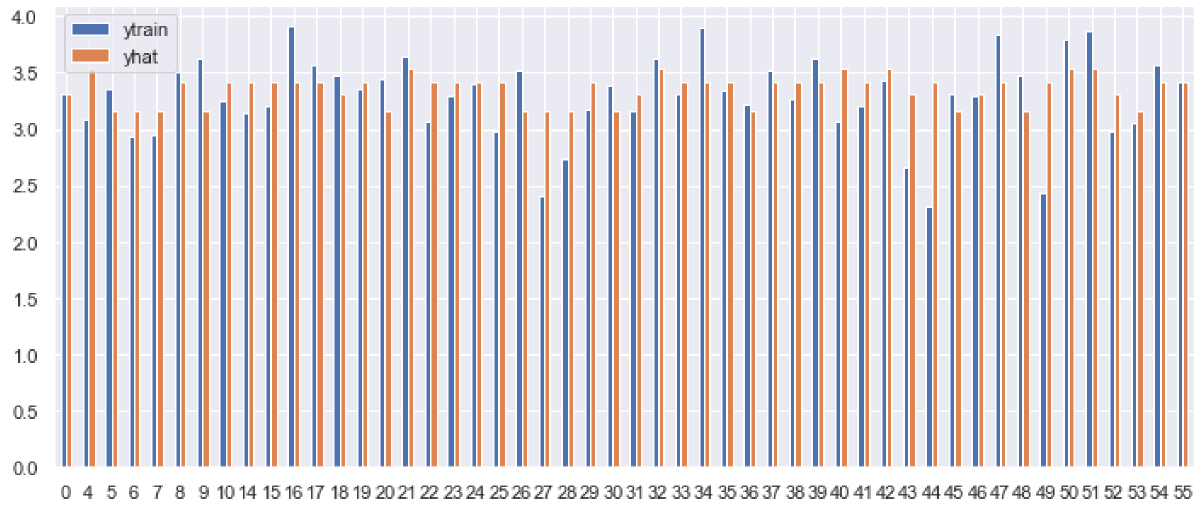
```
The best Decision Trees model has a RMSE of: 0.3785568030027869
```

In [70]:
```
# Let's predict on the training set
dt_yhat= best_dt.predict(Xtrain)

# Calculate how much time it took to tune the hyperparameters and train the model
dt_duration = time.time() - dt_start
print(f'The Decision Trees model took {dt_duration:.3f} seconds')
```

```
The Decision Trees model took 1.854 seconds
```

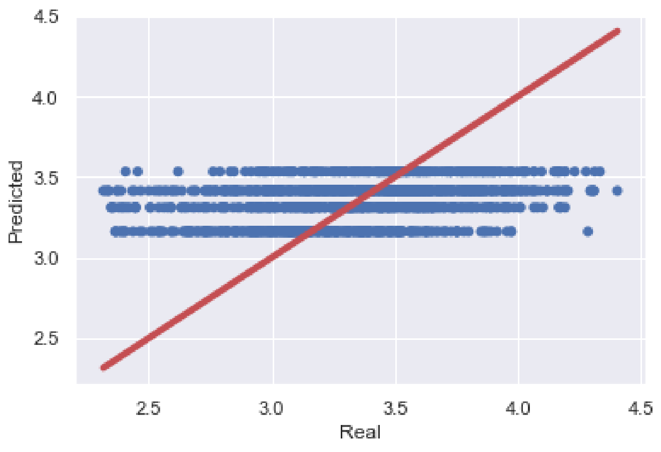
We will visualise the predictions for the first 50 test instances.

```
In [71]: # create a temporary dataframe containing two columns
         dt_df_tmp = pd.DataFrame({"ytrain": ytrain[:50], "yhat": dt_yhat[:50]})

         # plot the dataframe
         dt_df_tmp.plot(figsize=(12,5), kind="bar", rot=0)

Out[71]: <matplotlib.axes._subplots.AxesSubplot at 0x254c8981f28>
```



Scatter plot to visualise the predicted values versus their real values.

```
In [72]: fig, ax = plt.subplots()
         ax.scatter(ytrain, dt_yhat, edgecolors='None')
         ax.plot([ytrain.min(), ytrain.max()], [ytrain.min(), ytrain.max()], 'r', lw=4)
         ax.set_xlabel('Real')
         ax.set_ylabel('Predicted')
         plt.show()
```



Same case as for the linear regression, the decision tree model is not really good at predicting accurate average movie ratings.

We will create a new folder where we will keep all trained models.

```
In [73]: import os
         from joblib import dump

         # create a folder where all trained models will be kept
         if not os.path.exists("models"):
             os.makedirs("models")

         dump(best_dt, 'models/dt_reg.joblib')

Out[73]: ['models/dt_reg.joblib']
```

4.2.3 Random Forest

We want to find the best value for the hyperparameter by using Random Grid Search, this time. It works like GridSearch but we have to set in addition the number of random combinations to try.

In [74]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor

# specify the hyperparameters and their values
# n_estimators = number of decision trees used in the random forest
# max depth :  maximum number of levels in the trees
# min_samples_Split = Minimum number of samples required to split a node

param_grid = {'n_estimators': [3, 10, 20], 'max_depth': [2, 4, 6, 8, None]}

forest_reg = RandomForestRegressor()

# start a new timer
rf_start=time.time()

# we'll use 10-fold cross-validation with 10 random combinations to try
# we also want to get the train score for later
rf_random_grid_search = RandomizedSearchCV(forest_reg, param_grid, cv=10, n_iter=10, scoring='neg_mean_
squared_error', random_state=8, return_train_score=True)

# fit the estimator on the training set
rf_random_grid_search.fit(Xtrain, ytrain)
```

Out[74]:
```
RandomizedSearchCV(cv=10, error_score='raise-deprecating',
                   estimator=RandomForestRegressor(bootstrap=True,
                                                   criterion='mse',
                                                   max_depth=None,
                                                   max_features='auto',
                                                   max_leaf_nodes=None,
                                                   min_impurity_decrease=0.0,
                                                   min_impurity_split=None,
                                                   min_samples_leaf=1,
                                                   min_samples_split=2,
                                                   min_weight_fraction_leaf=0.0,
                                                   n_estimators='warn',
                                                   n_jobs=None, oob_score=False,
                                                   random_state=None, verbose=0,
                                                   warm_start=False),
                   iid='warn', n_iter=10, n_jobs=None,
                   param_distributions={'max_depth': [2, 4, 6, 8, None],
                                        'n_estimators': [3, 10, 20]},
                   pre_dispatch='2*n_jobs', random_state=8, refit=True,
                   return_train_score=True, scoring='neg_mean_squared_error',
                   verbose=0)
```

In [75]:
```
# the best random forest model
best_rf = rf_random_grid_search.best_estimator_
best_rf
```

Out[75]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=2,
                      max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=20,
                      n_jobs=None, oob_score=False, random_state=None,
                      verbose=0, warm_start=False)
```

The best model has 20 estimators, a max depth of 2 and a minimum of samples split of 2.

Use cross-validation to find the average accuracy score for this model.

In [76]:
```
rf_cv_scores= cross_val_score(best_rf, Xtrain, ytrain, scoring="neg_mean_squared_error", cv=10)

rf_cv_rmse_scores = np.sqrt(-rf_cv_scores)
display_scores(rf_cv_rmse_scores)
```

```
Scores: [0.37115534 0.38243532 0.389658   0.34147616 0.38233991 0.38071845
 0.35116717 0.42133541 0.38846569 0.34717283]
Accuracy: 0.3755924295940073
Standard deviation: 0.022721104681939826
```

On average, the random forest model has a RMSE better than the baseline's and the decision trees' and quite similar to the linear regression model, which is not reassuring since this model has a very low R2.

In [77]:
```
# training and validation RMSE

val_scores = rf_random_grid_search.cv_results_["mean_test_score"]
train_scores = rf_random_grid_search.cv_results_["mean_train_score"]
params = rf_random_grid_search.cv_results_["params"]

for val_score, train_score, param in zip(val_scores, train_scores, params):
    print(np.sqrt(-val_score), np.sqrt(-train_score), param)
```

```
0.38835231637228074 0.33183561374793324 {'n_estimators': 3, 'max_depth': 6}
0.38362734527561837 0.2744879155674241 {'n_estimators': 20, 'max_depth': 8}
0.3758902108240475 0.37173248189712055 {'n_estimators': 20, 'max_depth': 2}
0.43863031223622534 0.2206396548968979 {'n_estimators': 3, 'max_depth': None}
0.4026282342827458 0.16879980964057295 {'n_estimators': 10, 'max_depth': None}
0.38752087418699965 0.28041321532254837 {'n_estimators': 10, 'max_depth': 8}
0.3852978278619608 0.31894902331691466 {'n_estimators': 10, 'max_depth': 6}
0.3792091504074903 0.3732009846309224 {'n_estimators': 3, 'max_depth': 2}
0.38736386614563323 0.15563764565479402 {'n_estimators': 20, 'max_depth': None}
0.38310920416739963 0.31741903415756395 {'n_estimators': 20, 'max_depth': 6}
```

The best model has a validation rmse slightly higher than the training set. But they are quite similar. There may be some overfitting.

In [78]:
```
# the best model's RMSE
rf_rmse_score=np.sqrt(-rf_random_grid_search.best_score_)
print(f'The best Random Forest model has a RMSE of: {rf_rmse_score}')
```

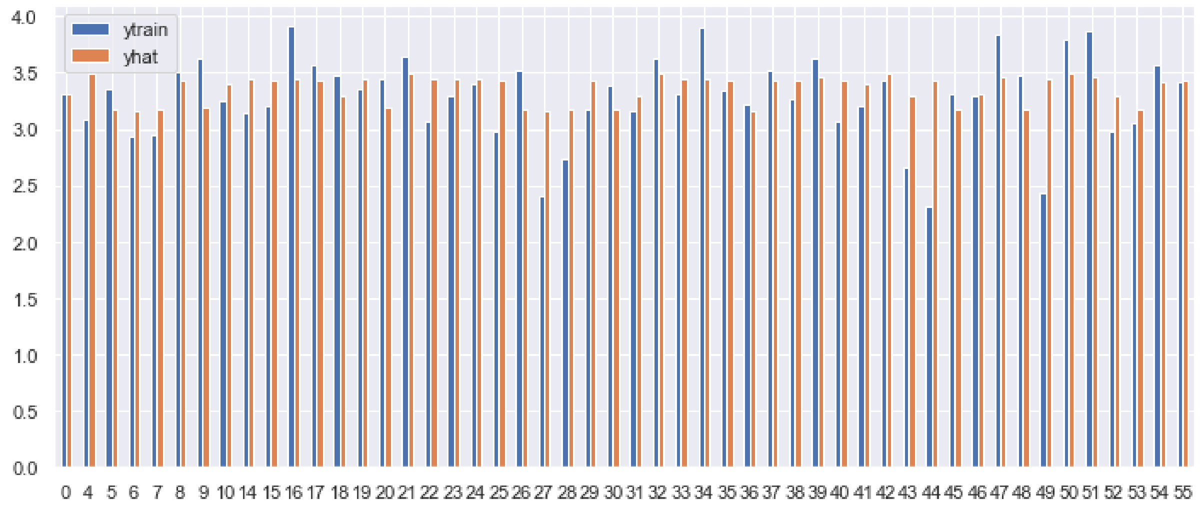
```
The best Random Forest model has a RMSE of: 0.3758902108240475
```

In [79]:
```
# make predictions
rf_yhat = best_rf.predict(Xtrain)

# Calculate how much time it took to tune the hyperparameters and train the model
rf_duration = time.time() - rf_start
print(f'The Random Forest model took {rf_duration:.3f} seconds')
```

```
The Random Forest model took 6.490 seconds
```

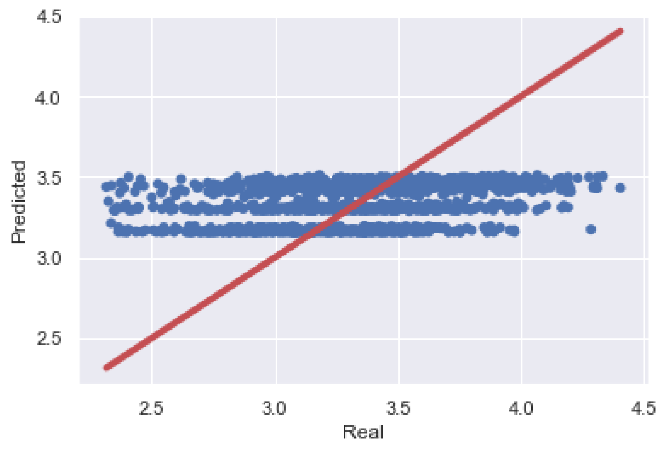
Visualise the predictions for the first 50 test instances.

```
In [80]: # create a temporary dataframe containing two columns
         rf_df_tmp = pd.DataFrame({"ytrain": ytrain[:50], "yhat": rf_yhat[:50]})

         # plot the dataframe
         rf_df_tmp.plot(figsize=(12,5), kind="bar", rot=0)

Out[80]: <matplotlib.axes._subplots.AxesSubplot at 0x254cab50278>
```



Create a scatter plot to visualise the predicted values versus their real values.

```
In [81]: fig, ax = plt.subplots()
         ax.scatter(ytrain, rf_yhat, edgecolors='None')
         ax.plot([ytrain.min(), ytrain.max()], [ytrain.min(), ytrain.max()], 'r', lw=4)
         ax.set_xlabel('Real')
         ax.set_ylabel('Predicted')
         plt.show()
```



We have the same problem as for the other models. Most predictions do not seem to be accurate.

```
In [82]: # save the model
         dump(best_rf, 'models/rf_reg.joblib')

Out[82]: ['models/rf_reg.joblib']
```

4.2.4 Support Vector Regressor

We will tune the hyperparameters for the Support Vector Regressor with GridSearch.

In [83]:
```
from sklearn.svm import SVR

svr_param_grid = [
    {'C': [1.0, 10, 100,10000],
     'gamma': ["scale", "auto", 0.01, 0.1, 1, 3, 5, 10]
    },
]


svr = SVR(kernel="rbf")

# start a new timer
svr_start = time.time()

#10 fold cross_validation and access to train score for later
sv_grid_search = GridSearchCV(svr, svr_param_grid, cv=10, scoring='neg_mean_squared_error',
                              return_train_score=True)

# fit the best model and hyperparameters to the training set
sv_grid_search.fit(Xtrain, ytrain)
```

Out[83]:
```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=SVR(C=1.0, cache_size=200, coef0=0.0, degree=3,
                           epsilon=0.1, gamma='auto_deprecated', kernel='rbf',
                           max_iter=-1, shrinking=True, tol=0.001,
                           verbose=False),
             iid='warn', n_jobs=None,
             param_grid=[{'C': [1.0, 10, 100, 10000],
                          'gamma': ['scale', 'auto', 0.01, 0.1, 1, 3, 5, 10]}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_squared_error', verbose=0)
```

In [84]:
```
#Best svr model
best_sv = sv_grid_search.best_estimator_
best_sv
```

Out[84]:
```
SVR(C=10000, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='scale',
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

The best model has C=10000 and gamma= scale.

Use cross-validation to find the average accuracy score for this model.

In [85]:
```
sv_cv_scores= cross_val_score(best_sv, Xtrain, ytrain, scoring="neg_mean_squared_error", cv=10)

sv_cv_rmse_scores = np.sqrt(-sv_cv_scores)
display_scores(sv_cv_rmse_scores)
```

```
Scores: [0.37001542 0.38651038 0.38259263 0.34617254 0.39003272 0.38469793
 0.36211629 0.4201804  0.39181604 0.34367178]
Accuracy: 0.37778061245203753
Standard deviation: 0.02181791705981524
```

The Support Vector Regression model has on average a RMSE of 0.3779, which is the worst accuracy score compared to the other models but better than the baseline.

In [86]:
```python
#training and validation RMSE
val_scores = sv_grid_search.cv_results_["mean_test_score"]
train_scores = sv_grid_search.cv_results_["mean_train_score"]
params = sv_grid_search.cv_results_["params"]

for val_score, train_score, param in zip(val_scores, train_scores, params):
    print(np.sqrt(-val_score), np.sqrt(-train_score), param)
```

```
0.39411677606837264 0.39293512455847796 {'C': 1.0, 'gamma': 'scale'}
0.3952826095290285 0.09306858614969328 {'C': 1.0, 'gamma': 'auto'}
0.3976390369035476 0.09656203762201102 {'C': 1.0, 'gamma': 0.01}
0.39520903040937705 0.09304682137740106 {'C': 1.0, 'gamma': 0.1}
0.39509998507906036 0.09304350939403672 {'C': 1.0, 'gamma': 1}
0.39509998531547524 0.09304360188823343 {'C': 1.0, 'gamma': 3}
0.39509998531547524 0.09304360188823343 {'C': 1.0, 'gamma': 5}
0.39509998531547524 0.09304360188823343 {'C': 1.0, 'gamma': 10}
0.3922089878054019 0.39043313336513086 {'C': 10, 'gamma': 'scale'}
0.3952826095290285 0.09306858614969328 {'C': 10, 'gamma': 'auto'}
0.3976832612094749 0.09332432580323853 {'C': 10, 'gamma': 0.01}
0.39520903040937705 0.09304682137740106 {'C': 10, 'gamma': 0.1}
0.39509998507906036 0.09304350939403672 {'C': 10, 'gamma': 1}
0.39509998531547524 0.09304360188823343 {'C': 10, 'gamma': 3}
0.39509998531547524 0.09304360188823343 {'C': 10, 'gamma': 5}
0.39509998531547524 0.09304360188823343 {'C': 10, 'gamma': 10}
0.3820088343557463 0.37995177216208403 {'C': 100, 'gamma': 'scale'}
0.3952826095290285 0.09306858614969328 {'C': 100, 'gamma': 'auto'}
0.3976832612094749 0.09332432580323853 {'C': 100, 'gamma': 0.01}
0.39520903040937705 0.09304682137740106 {'C': 100, 'gamma': 0.1}
0.39509998507906036 0.09304350939403672 {'C': 100, 'gamma': 1}
0.39509998531547524 0.09304360188823343 {'C': 100, 'gamma': 3}
0.39509998531547524 0.09304360188823343 {'C': 100, 'gamma': 5}
0.39509998531547524 0.09304360188823343 {'C': 100, 'gamma': 10}
0.37840454388440725 0.37370979249119474 {'C': 10000, 'gamma': 'scale'}
0.3952826095290285 0.09306858614969328 {'C': 10000, 'gamma': 'auto'}
0.3976832612094749 0.09332432580323853 {'C': 10000, 'gamma': 0.01}
0.39520903040937705 0.09304682137740106 {'C': 10000, 'gamma': 0.1}
0.39509998507906036 0.09304350939403672 {'C': 10000, 'gamma': 1}
0.39509998531547524 0.09304360188823343 {'C': 10000, 'gamma': 3}
0.39509998531547524 0.09304360188823343 {'C': 10000, 'gamma': 5}
0.39509998531547524 0.09304360188823343 {'C': 10000, 'gamma': 10}
```

The best model does not show overfitting or underfitting because the performances on the training and validation sets are roughly similar.

In [87]:
```python
# Best model RMSE
sv_rmse_score=np.sqrt(-sv_grid_search.best_score_)
print(f'The best SVR model has a RMSE of: {sv_rmse_score}')
```

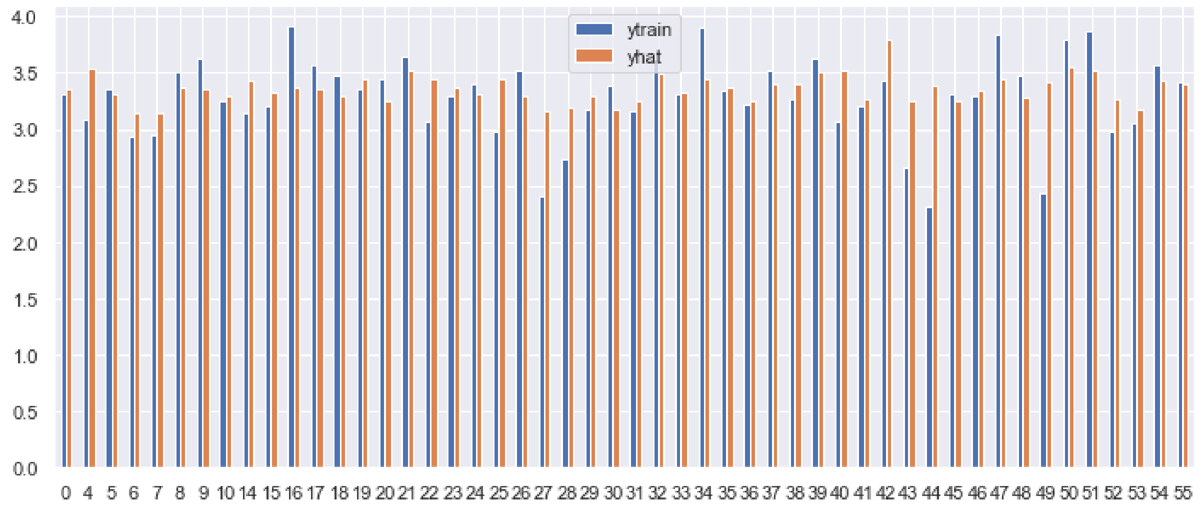
```
The best SVR model has a RMSE of: 0.37840454388440725
```

In [88]:
```python
# make predictions
svr_yhat = best_sv.predict(Xtrain)

# Calculate how much time it took to tune the hyperparameters and train the model
svr_duration = time.time() - svr_start
print(f'The SVR model took {svr_duration:.3f} seconds')
```

```
The SVR model took 83.133 seconds
```

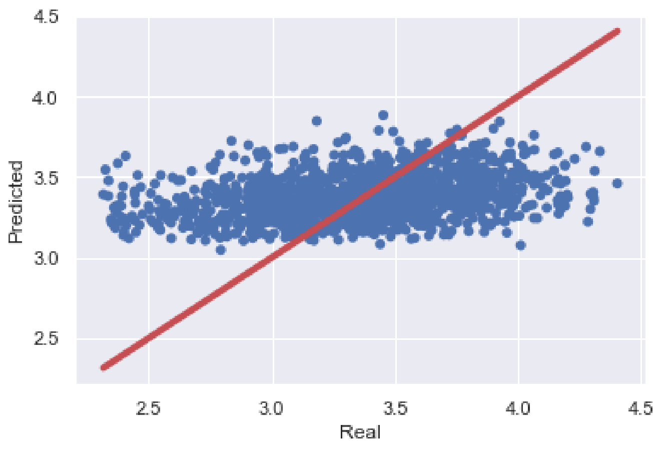
Visualise the predictions for the first 50 test instances.

```
In [89]: # create a temporary dataframe containing two columns
         svr_df_tmp = pd.DataFrame({"ytrain": ytrain[:50], "yhat": svr_yhat[:50]})

         # plot the dataframe
         svr_df_tmp.plot(figsize=(12,5), kind="bar", rot=0)

Out[89]: <matplotlib.axes._subplots.AxesSubplot at 0x254cb222ac8>
```



Create a scatter plot to visualise the predicted values versus their real values.

```
In [90]: fig, ax = plt.subplots()
         ax.scatter(ytrain, svr_yhat, edgecolors='None')
         ax.plot([ytrain.min(), ytrain.max()], [ytrain.min(), ytrain.max()], 'r', lw=4)
         ax.set_xlabel('Real')
         ax.set_ylabel('Predicted')
         plt.show()
```



Same problem as for the other models: the predictions are not enough accurate.

```
In [91]: # Save the model
         dump(best_sv, 'models/sv_reg.joblib')

Out[91]: ['models/sv_reg.joblib']
```

4.2.5 Adaboost Regressor

We tune the hyperparameter for the Adaboost Regressor by using Random Grid Search.

```
In [92]: from sklearn.ensemble import AdaBoostRegressor

         adb_reg=AdaBoostRegressor()

         adb_param_grid = {'n_estimators': [3, 10, 20, 50], 'learning_rate': [0.001, 0.01, 0.1, 0.25, 0.5, 0.75,
         1],
              'loss' : ['linear', 'square', 'exponential']}

         adb_reg=AdaBoostRegressor(random_state=8)

         #start a new timer
         adb_start = time.time()

         # we'll use 10-fold cross-validation and want to have access to the train score
         adb_random_grid_search = RandomizedSearchCV(adb_reg, adb_param_grid, cv=10, n_iter=10,
                                          scoring='neg_mean_squared_error', random_state=8, return_train_score=T
         rue)

         #fit the best model and hyperparameters to the training dataset
         adb_random_grid_search.fit(Xtrain, ytrain)
```

```
Out[92]: RandomizedSearchCV(cv=10, error_score='raise-deprecating',
                            estimator=AdaBoostRegressor(base_estimator=None,
                                                        learning_rate=1.0, loss='linear',
                                                        n_estimators=50,
                                                        random_state=8),
                            iid='warn', n_iter=10, n_jobs=None,
                            param_distributions={'learning_rate': [0.001, 0.01, 0.1,
                                                                   0.25, 0.5, 0.75, 1],
                                                 'loss': ['linear', 'square',
                                                          'exponential'],
                                                 'n_estimators': [3, 10, 20, 50]},
                            pre_dispatch='2*n_jobs', random_state=8, refit=True,
                            return_train_score=True, scoring='neg_mean_squared_error',
                            verbose=0)
```

```
In [93]: # the best model
         best_adb = adb_random_grid_search.best_estimator_
         best_adb
```

```
Out[93]: AdaBoostRegressor(base_estimator=None, learning_rate=0.001, loss='linear',
                           n_estimators=50, random_state=8)
```

The best Adaboost model has 50 estimators, 8 for random state, a learning rate of 0.001 and loss=linear.

Cross-validation to find the average accuracy score for this model.

```
In [94]: adb_cv_scores= cross_val_score(best_adb, Xtrain, ytrain, scoring="neg_mean_squared_error", cv=10)

         adb_cv_rmse_scores = np.sqrt(-adb_cv_scores)
         display_scores(adb_cv_rmse_scores)
```

```
Scores: [0.37349878 0.38577502 0.38960461 0.34535309 0.3831018  0.37971753
 0.35149335 0.41853834 0.39113513 0.34622229]
Accuracy: 0.3764439944423551
Standard deviation: 0.021941468367239 32
```

On average, the AdaBoost model has an accuracy rate of 0.3764, which is our best RMSE among the models.

```
In [95]: # training and validation RMSE

         adb_val_scores = adb_random_grid_search.cv_results_["mean_test_score"]
         adb_train_scores = adb_random_grid_search.cv_results_["mean_train_score"]
         adb_params = adb_random_grid_search.cv_results_["params"]

         for adb_val_score, adb_train_score, adb_param in zip(adb_val_scores, adb_train_scores, adb_params):
             print(np.sqrt(-adb_val_score), np.sqrt(-adb_train_score), param)
```

```
0.3823976427423386 0.3702522505625894 {'C': 10000, 'gamma': 10}
0.3772780116624631 0.36798346002282173 {'C': 10000, 'gamma': 10}
0.3787864541879901 0.36823045448970915 {'C': 10000, 'gamma': 10}
0.3883477034561994 0.37045685031349634 {'C': 10000, 'gamma': 10}
0.3851211364379346 0.3694716815021636 {'C': 10000, 'gamma': 10}
0.37873308008611783 0.36458232173633537 {'C': 10000, 'gamma': 10}
0.38828874366140087 0.3638991593295995 {'C': 10000, 'gamma': 10}
0.3784166831961917 0.36848616554122093 {'C': 10000, 'gamma': 10}
0.3840030811099085 0.3700938413517432 {'C': 10000, 'gamma': 10}
0.3770805016370175 0.3678329431763814 {'C': 10000, 'gamma': 10}
```

The model has a rmse on the validation set slightly higher than on the training set. But they are still similar.

```
In [96]: #RMSE score
         adb_rmse_score = np.sqrt(-adb_random_grid_search.best_score_)
         print(f'The best Adaboost model has a RMSE of: {adb_rmse_score}')
```

```
The best Adaboost model has a RMSE of: 0.3770805016370175
```

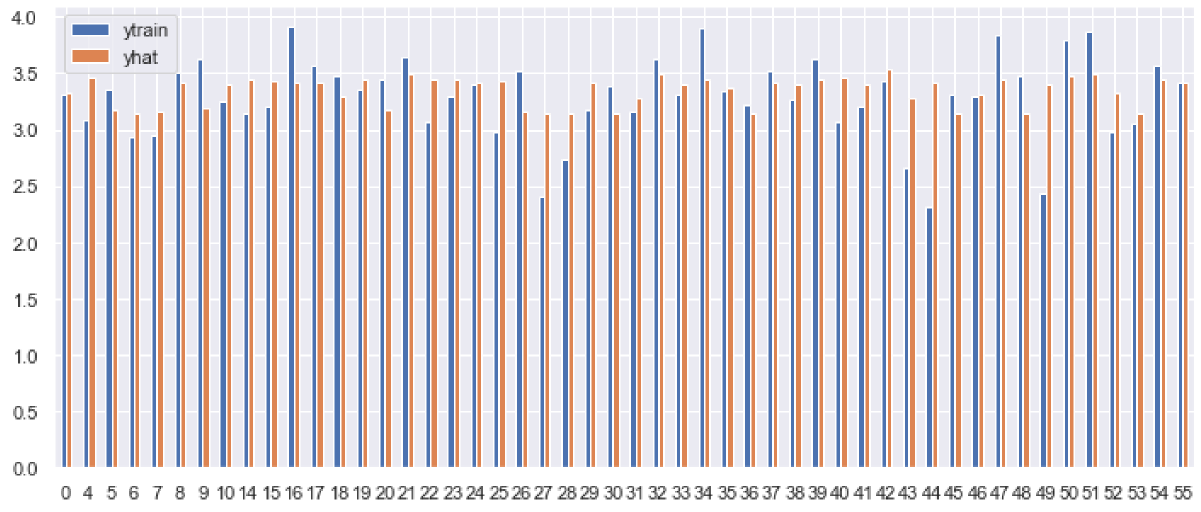
```
In [97]: # make predictions
         adb_yhat = best_adb.predict(Xtrain)

         # Calculate how much time it took to tune the hyperparameters and train the model
         adb_duration = time.time() - adb_start
         print(f'The Adaboost model took {adb_duration:.3f} seconds')
```

```
The Adaboost model took 6.774 seconds
```

Visualise the predictions for the first 50 test instances.

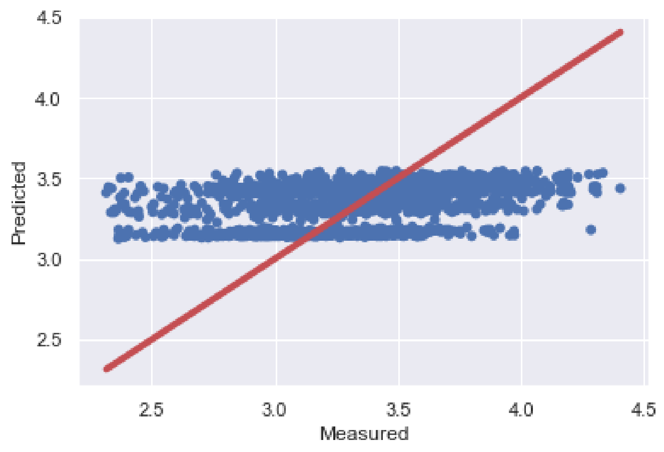
```
In [98]: # create a temporary dataframe containing two columns
         adb_df_tmp = pd.DataFrame({"ytrain": ytrain[:50], "yhat": adb_yhat[:50]})

         # plot the dataframe
         adb_df_tmp.plot(figsize=(12,5), kind="bar", rot=0)
```

```
Out[98]: <matplotlib.axes._subplots.AxesSubplot at 0x254cca25208>
```



Create a scatter plot to visualise the predicted values versus their real values.

In [99]:
```
fig, ax = plt.subplots()
ax.scatter(ytrain, adb_yhat, edgecolors='None')
ax.plot([ytrain.min(), ytrain.max()], [ytrain.min(), ytrain.max()], 'r', lw=4)
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```



As for the other models, the AdaBoost model seems to lack accuracy to predict the average movie ratings.

Save the model

In [100]:
```
dump(best_adb, 'models/adv_reg.joblib')
```

Out[100]:
```
['models/adv_reg.joblib']
```

4.2.6 Compare the models

In terms of execution speed:

In [101]:
```
print(f'Decision Trees: {dt_duration:.3f} seconds')
print(f'Random Forest: {rf_duration:.3f} seconds')
print(f'Support Vector Machine Regressor: {svr_duration:.3f} seconds')
print(f'Adaboost: {adb_duration:.3f} seconds')
```

```
Decision Trees: 1.854 seconds
Random Forest: 6.490 seconds
Support Vector Machine Regressor: 83.133 seconds
Adaboost: 6.774 seconds
```

The Decision Tree model was the fastest to run and the Support Vector Regressor the slowest one. The Random Forest and the Adaboost models have similar execution speed.

Cross-validation scores

```
In [102]: print('Linear Regression')
          display_scores(rmse_lin_scores)
          print('')
          print('Decision Trees')
          display_scores(dt_cv_rmse_scores)
          print('')
          print('Random Forest')
          display_scores(rf_cv_rmse_scores)
          print('')
          print('Support Vector Regression')
          display_scores(sv_cv_rmse_scores)
          print('')
          print('AdaBoost')
          display_scores(adb_cv_rmse_scores)
```

```
Linear Regression
Scores: [0.36532339 0.38827114 0.38816454 0.33848736 0.38884871 0.38132385
 0.3500855  0.42528398 0.39013122 0.3496958 ]
Accuracy: 0.3765615481424965
Standard deviation: 0.024499181317022143

Decision Trees
Scores: [0.37386173 0.38605892 0.39303714 0.34206869 0.38361367 0.38536461
 0.35223175 0.42367925 0.38917399 0.34939939]
Accuracy: 0.3778489145170766
Standard deviation: 0.02319098553768898

Random Forest
Scores: [0.37115534 0.38243532 0.389658   0.34147616 0.38233991 0.38071845
 0.35116717 0.42133541 0.38846569 0.34717283]
Accuracy: 0.3755924295940073
Standard deviation: 0.022721104681939826

Support Vector Regression
Scores: [0.37001542 0.38651038 0.38259263 0.34617254 0.39003272 0.38469793
 0.36211629 0.4201804  0.39181604 0.34367178]
Accuracy: 0.37778061245203753
Standard deviation: 0.02181791705981524

AdaBoost
Scores: [0.37349878 0.38577502 0.38960461 0.34535309 0.3831018  0.37971753
 0.35149335 0.41853834 0.39113513 0.34622229]
Accuracy: 0.3764439944423551
Standard deviation: 0.02194146836723932
```

Following the cross-validation, on average, the Decision Tree and the Support Vector Regression models have the worst accuracy scores. Even though the linear regression model and the Random Forest models have similar RMSE, the latter has a lower variance/standard deviation. The model with the best rmse is the AdaBoost.

Let's compare the RMSE of each best model against the baseline.

```
In [103]: RMSE= {
              'Baseline RMSE': [baseline_rmse],
              'Linear Regression': [rmse_lin_scores.mean()],
              'Decision Trees': [dt_rmse_score],
              'Random Forest': [rf_rmse_score],
              'Support Vector Machine': [sv_rmse_score],
              'Adaboost': [adb_rmse_score]
          }

          scores_df=pd.DataFrame(RMSE).transpose()
          scores_df.columns=['RMSE']
          scores_df["Vs Baseline"] = 100*(baseline_rmse-scores_df["RMSE"])/baseline_rmse
          scores_df
```

```
Out[103]:
```

	RMSE	Vs Baseline
Baseline RMSE	0.395299	0.000000
Linear Regression	0.376562	4.740118
Decision Trees	0.378557	4.235373
Random Forest	0.375890	4.909948
Support Vector Machine	0.378405	4.273890
Adaboost	0.377081	4.608837

The best models seem to be the random forest and the linear regression (!). The decision trees and the svm have the highest rmse, even though their rmse are lower (and thus better) than the baseline. Since the linear regression has a low R2 coefficient, I have decided to choose the **Random Forest** and the **Adaboost** models.

4.3 Analyzing the best two Models

4.3.1 Feature Importance

What are the most important variables according to each model?

```
In [104]: # Load the models if they took a lot of time to train
          #from joblib import dump

          #best_rf=load('models/rf_reg.joblib')
          #best_adb = load('models/adv_reg.joblib')
```

Most important features for the **Random Forest model**:

```
In [105]: # Put the features into a variable for each model
          rf_feature_importances = best_rf.feature_importances_
          adb_feature_importances = best_adb.feature_importances_
```

```
In [106]: # We will see the results in a dataframe we will create
          rf_imp_feat_df = pd.DataFrame(rf_feature_importances, Xtrain.columns)
          rf_imp_feat_df.rename(columns={0:'Feature importance'}, inplace=True)
          # sort the feature importance by descending order
          rf_imp_feat_df.sort_values('Feature importance', ascending=False)
```

Out[106]:

	Feature importance
duration	0.715763
Year_of_release	0.172423
actor_1_facebook_likes	0.040809
num_voted_users	0.032079
Movie_Id	0.016084
ROI	0.010133
gross	0.008144
imdb_score	0.004565
num_critic_for_reviews	0.000000
num_user_for_reviews	0.000000
cast_total_facebook_likes	0.000000
movie_facebook_likes	0.000000
Color	0.000000
PG	0.000000

Most important features for the **Adaboost model**:

```
In [107]: # Let's have a look at the feature importance in a new dataframe
          adb_imp_feat_df = pd.DataFrame(adb_feature_importances, Xtrain.columns)
          adb_imp_feat_df.rename(columns={0:'Feature importance'}, inplace=True)
          # sort the feature importance by descending order
          adb_imp_feat_df.sort_values('Feature importance', ascending=False)
```

Out[107]:

	Feature importance
duration	0.558578
Year_of_release	0.176565
Movie_Id	0.061276
actor_1_facebook_likes	0.036875
ROI	0.034013
num_voted_users	0.031398
cast_total_facebook_likes	0.027726
gross	0.025158
num_user_for_reviews	0.020452
num_critic_for_reviews	0.012721
imdb_score	0.006847
movie_facebook_likes	0.005378
Color	0.003011
PG	0.000000

For both models, the **most important predictor is duration** with more than 0.55 in each model.

Then, we have:

- the year of release,
- the ROI,
- the lead actor's number of Facebook likes,
- the number of voted users
- and the cast's total number of Facebook likes.

In both cases, color, PG and the number of movie_facebook_likes come last. Knowing that in the 2000's and now, most of the movies are in color and many of them are PG, it is not suprising that they do not have an impact on the movie ratings. The Imdb score does not seem to be that significant in the rating of the movie on Netflix. It may be due to the fact, that since the viewers pay for their Netflix's subscription, they just want to watch a movie, no matter its imdb score, that they might check.

We will keep only the features we like in the training and test sets.

```
In [108]: # creation of new dataframes with the features we want
          Xtrain_new=Xtrain.loc[:,["Movie_Id","duration","Year_of_release", 'actor_1_facebook_likes', 'ROI',
                                  'num_voted_users', 'cast_total_facebook_likes']]
          Xtest_new=Xtest.loc[:,["Movie_Id","duration","Year_of_release", 'actor_1_facebook_likes', 'ROI',
                                  'num_voted_users', 'cast_total_facebook_likes']]
          print (Xtrain_new.shape)
          print (Xtest_new.shape)
```

```
(1491, 7)
(678, 7)
```

Since we modified the train and test sets, we need to find for each model the best hyperparamaters and train them on the "new" datasets.

4.3.2 Rebuilding and Training the best Models

4.3.2.1 Random Forest

Hyperparameter tuning with Randomized Search.

In [109]:
```
rf_param_grid2 = {'n_estimators': [3, 10, 20], 'max_depth': [2, 4, 6, 8, None]}

forest_reg2 = RandomForestRegressor(random_state=8)

#start a new timer
rf_start2 = time.time()

# we'll use 10-fold cross-validation
rf_random_grid_search2 = RandomizedSearchCV(forest_reg2, rf_param_grid2, cv=10, n_iter=10,
                                 scoring='neg_mean_squared_error', random_state=8, return_train_score=T
rue)

#Fit the best model (and best estimator) to the new training set
rf_random_grid_search2.fit(Xtrain_new, ytrain)
```

Out[109]:
```
RandomizedSearchCV(cv=10, error_score='raise-deprecating',
                   estimator=RandomForestRegressor(bootstrap=True,
                                                   criterion='mse',
                                                   max_depth=None,
                                                   max_features='auto',
                                                   max_leaf_nodes=None,
                                                   min_impurity_decrease=0.0,
                                                   min_impurity_split=None,
                                                   min_samples_leaf=1,
                                                   min_samples_split=2,
                                                   min_weight_fraction_leaf=0.0,
                                                   n_estimators='warn',
                                                   n_jobs=None, oob_score=False,
                                                   random_state=8, verbose=0,
                                                   warm_start=False),
                   iid='warn', n_iter=10, n_jobs=None,
                   param_distributions={'max_depth': [2, 4, 6, 8, None],
                                        'n_estimators': [3, 10, 20]},
                   pre_dispatch='2*n_jobs', random_state=8, refit=True,
                   return_train_score=True, scoring='neg_mean_squared_error',
                   verbose=0)
```

In [110]:
```
# the best model
rf_final = rf_random_grid_search2.best_estimator_
rf_final
```

Out[110]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=2,
                      max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=20,
                      n_jobs=None, oob_score=False, random_state=8, verbose=0,
                      warm_start=False)
```

The best model has n_estimators=20, max_depth=2.

In [111]:
```
#the best model's RMSE
rf_final_rmse = np.sqrt(-rf_random_grid_search2.best_score_)
print(f'The best new Random Forest model has a RMSE of: {rf_final_rmse}')
```

```
The best new Random Forest model has a RMSE of: 0.37694516613254686
```

Its RMSE is slightly higher than the first random forest model (0.3765). So, this model is less good than the first one, even though we removed some variables.

```
In [112]: # training and validation RMSE

          rf_val_scores = rf_random_grid_search2.cv_results_["mean_test_score"]
          rf_train_scores = rf_random_grid_search2.cv_results_["mean_train_score"]
          rf_params = rf_random_grid_search2.cv_results_["params"]

          for rf_val_score, rf_train_score, rf_param in zip(rf_val_scores, rf_train_scores, rf_params):
              print(np.sqrt(-rf_val_score), np.sqrt(-rf_train_score), rf_param)
```

```
0.39070640266414464 0.33741391273527443 {'n_estimators': 3, 'max_depth': 6}
0.3831304633311535 0.2796175051772477 {'n_estimators': 20, 'max_depth': 8}
0.37694516613254686 0.3715452349386388 {'n_estimators': 20, 'max_depth': 2}
0.44591992070305136 0.2272211861478854 {'n_estimators': 3, 'max_depth': None}
0.4032491999483958 0.17166035108800298 {'n_estimators': 10, 'max_depth': None}
0.3882984629153159 0.2835836907094717 {'n_estimators': 10, 'max_depth': 8}
0.3808893784918807 0.32374521805262735 {'n_estimators': 10, 'max_depth': 6}
0.37924808776361213 0.37368210335758084 {'n_estimators': 3, 'max_depth': 2}
0.39322370925163685 0.15905560718780587 {'n_estimators': 20, 'max_depth': None}
0.3791840917654251 0.32080722339743967 {'n_estimators': 20, 'max_depth': 6}
```

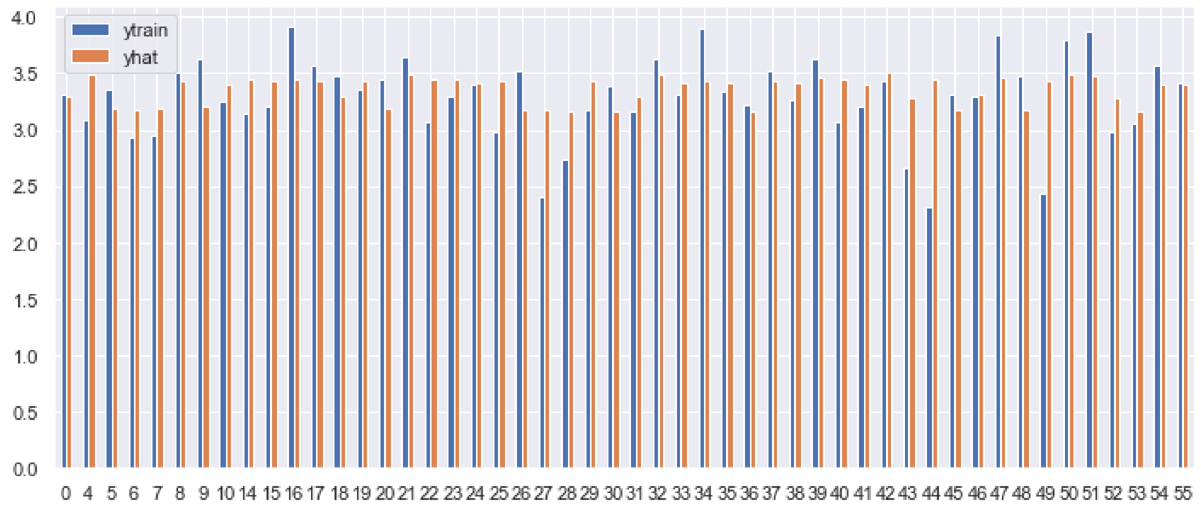
The training and validation RMSE scores for the best model are quite close.

```
In [113]: # let's predict on the new training set
          rf_yhat3= rf_random_grid_search2.predict(Xtrain_new)

          # Calculate how much time it took to tune the hyperparameters and train the model
          best_rf_duration = time.time() - rf_start2
          print(f'The best new Random Forest model took {best_rf_duration:.3f} seconds')
```

```
The best new Random Forest model took 4.202 seconds
```

Visualise the predictions for the first 50 test instances.

```
In [114]: # create a temporary dataframe containing two columns
          rf_df_tmp_3 = pd.DataFrame({"ytrain": ytrain[:50], "yhat": rf_yhat3[:50]})

          # plot the dataframe
          rf_df_tmp_3.plot(figsize=(12,5), kind="bar", rot=0)

Out[114]: <matplotlib.axes._subplots.AxesSubplot at 0x254cca13588>
```



The predictions seem to be fine most of the time.

```
In [115]: # save the new random forest model

          dump(rf_final, 'models/rf_reg_final.joblib')

Out[115]: ['models/rf_reg_final.joblib']
```

4.3.2.2 Adaboost Regressor

```
In [116]: #creation of a new parameter grid
          # we will use the randomized grid search to find the best parameters for the new Adaboost regressor mod
          el
          adb_param_grid2 = {'n_estimators': [3, 10, 20, 50], 'learning_rate': [0.001, 0.01, 0.1, 0.25, 0.5, 0.75
          , 1],
               'loss' : ['linear', 'square', 'exponential']}

          adb_reg2=AdaBoostRegressor(random_state=8)

          #start a new timer
          adb_start2 = time.time()

          # we'll use 10-fold cross-validation
          adb_random_grid_search2 = RandomizedSearchCV(adb_reg2, adb_param_grid2, cv=10, n_iter=10,
                                            scoring='neg_mean_squared_error', random_state=8, return_train_score=T
          rue)
          #We fit the new best model with the best hyperparameters to the new training dataset
          adb_random_grid_search2.fit(Xtrain_new, ytrain)
```

```
Out[116]: RandomizedSearchCV(cv=10, error_score='raise-deprecating',
                             estimator=AdaBoostRegressor(base_estimator=None,
                                                         learning_rate=1.0, loss='linear',
                                                         n_estimators=50,
                                                         random_state=8),
                             iid='warn', n_iter=10, n_jobs=None,
                             param_distributions={'learning_rate': [0.001, 0.01, 0.1,
                                                                    0.25, 0.5, 0.75, 1],
                                                  'loss': ['linear', 'square',
                                                           'exponential'],
                                                  'n_estimators': [3, 10, 20, 50]},
                             pre_dispatch='2*n_jobs', random_state=8, refit=True,
                             return_train_score=True, scoring='neg_mean_squared_error',
                             verbose=0)
```

```
In [117]: # the best model
          adb_final = adb_random_grid_search2.best_estimator_
          adb_final
```

```
Out[117]: AdaBoostRegressor(base_estimator=None, learning_rate=0.001, loss='linear',
                            n_estimators=50, random_state=8)
```

The best model has n_estimators=50, a learning rate of 0.001 and loss=linear.

```
In [118]: #RMSE score
          adb_final_rmse = np.sqrt(-adb_random_grid_search2.best_score_)
          print(f'The best new Adaboost model has a RMSE of: {adb_final_rmse}')

          The best new Adaboost model has a RMSE of: 0.37661955455807716
```

This Adaboost model has a higher RMSE than the first model and it is slightly lower than the second random forest model.

```
In [119]: # training and validation RMSE
          adb_val_scores2 = adb_random_grid_search2.cv_results_["mean_test_score"]
          adb_train_scores2 = adb_random_grid_search2.cv_results_["mean_train_score"]
          adb_params2 = adb_random_grid_search2.cv_results_["params"]

          for adb_val_score2, adb_train_score2, adb_param2 in zip(adb_val_scores2, adb_train_scores2, adb_params2
          ):
              print(np.sqrt(-adb_val_score2), np.sqrt(-adb_train_score2), adb_param2)
```

```
0.38211050772195676 0.37059123652279646 {'n_estimators': 3, 'loss': 'linear', 'learning_rate': 0.01}
0.3784199448840347 0.3685272580902094 {'n_estimators': 10, 'loss': 'exponential', 'learning_rate': 0.
1}
0.37874968424271693 0.36830420688395477 {'n_estimators': 10, 'loss': 'square', 'learning_rate': 0.1}
0.39052829639328623 0.37132390571284074 {'n_estimators': 3, 'loss': 'square', 'learning_rate': 1}
0.38359131670000735 0.36925136149392496 {'n_estimators': 3, 'loss': 'exponential', 'learning_rate': 0.
25}
0.37936256650648004 0.36431296632486837 {'n_estimators': 20, 'loss': 'exponential', 'learning_rate':
0.5}
0.38379838064284594 0.3649111641879808 {'n_estimators': 20, 'loss': 'square', 'learning_rate': 1}
0.3778476438337894 0.3685301993349548 {'n_estimators': 10, 'loss': 'linear', 'learning_rate': 0.001}
0.38386585902307246 0.3694162610308001 {'n_estimators': 3, 'loss': 'square', 'learning_rate': 0.5}
0.37661955455807716 0.36762969454327704 {'n_estimators': 50, 'loss': 'linear', 'learning_rate': 0.001}
```

The best Adaboost model has a RMSE score lower on the validation set than the training set.

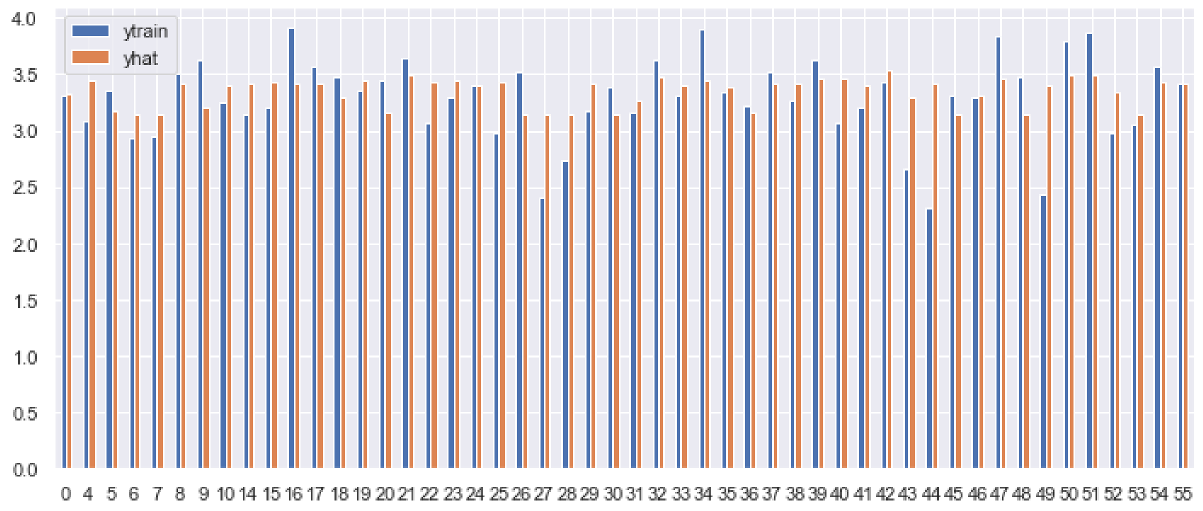
```
In [120]: # let's predict on the new training set
          adb_yhat2= adb_random_grid_search2.predict(Xtrain_new)

          # Calculate how much time it took to tune the hyperparameters and train the model
          best_adb_duration = time.time() - adb_start2
          print(f'Thebest new Adaboost model took {best_adb_duration:.3f} seconds')
```

```
Thebest new Adaboost model took 3.756 seconds
```

Visualise the predictions for the first 50 test instances.

```
In [121]: # create a temporary dataframe containing two columns
          adb_df_tmp_2 = pd.DataFrame({"ytrain": ytrain[:50], "yhat": adb_yhat2[:50]})

          # plot the dataframe
          adb_df_tmp_2.plot(figsize=(12,5), kind="bar", rot=0)
```

```
Out[121]: <matplotlib.axes._subplots.AxesSubplot at 0x254ccba85c0>
```



The predictions seem to be alright.

```
In [122]: # save the new Adaboost model

          dump(adb_final, 'models/adb_reg_final.joblib')
```

```
Out[122]: ['models/adb_reg_final.joblib']
```

Comparison of the models

In terms of speed of execution:

In [123]:
```
print(f'The best new Random Forest model took {best_rf_duration:.3f} seconds')
print(f'The best new Adaboost model took {best_adb_duration:.3f} seconds')
```

```
The best new Random Forest model took 4.202 seconds
The best new Adaboost model took 3.756 seconds
```

The fastest model is the Adaboost model.The removal of some variables increased the execution speed of the models.

Let's have a look at their accuracy rate (RMSE)

In [124]:
```
RMSE_Fi= {
    'Baseline RMSE': [baseline_rmse],
    'Random Forest': [rf_rmse_score],
    'Random Forest Final': [rf_final_rmse],
    'Adaboost': [adb_rmse_score],
    'Adaboost Final': [adb_final_rmse]
}

rmse_fi_df=pd.DataFrame(RMSE_Fi).transpose()
rmse_fi_df.columns=['RMSE_Fi']
rmse_fi_df["Vs Baseline"] = 100*(baseline_rmse-rmse_fi_df["RMSE_Fi"])/baseline_rmse
rmse_fi_df
```

Out[124]:

	RMSE_Fi	Vs Baseline
Baseline RMSE	0.395299	0.000000
Random Forest	0.375890	4.909948
Random Forest Final	0.376945	4.643073
Adaboost	0.377081	4.608837
Adaboost Final	0.376620	4.725444

The feature selection did improve the accuracy of the random forestand the Adaboost models, especially for the Adaboost (+0.12 pts of accuracy versus before).

4.3.2.3 Feature Importance Analysis

For the **Random Forest** model

In [125]:
```
rf_fi = rf_final.feature_importances_
rf_fi_df = pd.DataFrame(rf_fi, Xtrain_new.columns)
rf_fi_df.rename(columns={0:'Feature importance'}, inplace=True)
rf_fi_df.sort_values('Feature importance', ascending=False)
```

Out[125]:

	Feature importance
duration	0.745444
Year_of_release	0.155814
ROI	0.030053
actor_1_facebook_likes	0.023448
num_voted_users	0.021066
Movie_Id	0.013270
cast_total_facebook_likes	0.010907

For the **Adaboost** model

In [126]:
```
adb_fi = adb_final.feature_importances_
adb_fi_df = pd.DataFrame(adb_fi, Xtrain_new.columns)
adb_fi_df.rename(columns={0:'Feature importance'}, inplace=True)
adb_fi_df.sort_values('Feature importance', ascending=False)
```

Out[126]:

	Feature importance
duration	0.565615
Year_of_release	0.181262
Movie_Id	0.079041
num_voted_users	0.045224
cast_total_facebook_likes	0.045095
ROI	0.043411
actor_1_facebook_likes	0.040352

The most important feature is duration and then the year of release.

5.0 Evaluating the best Models on the Test Dataset

Now, we will evaluate the Random Forest and Adaboost models on the test dataset.

In [127]:
```
# code to import the models if they take to much time to train (just in case)
#final_rf = load("models/rf_reg_final.joblib")
#finalt_adb = load("models/adb_reg_final.joblib")
```

5.1 Random Forest

In [128]:
```
# we make the predictions on the test set.
rf_final_yhat = rf_final.predict(Xtest_new)

rf_test_rmse = np.sqrt(mean_squared_error(ytest, rf_final_yhat))
print(f'Random Forest RMSE: {rf_test_rmse}')
```

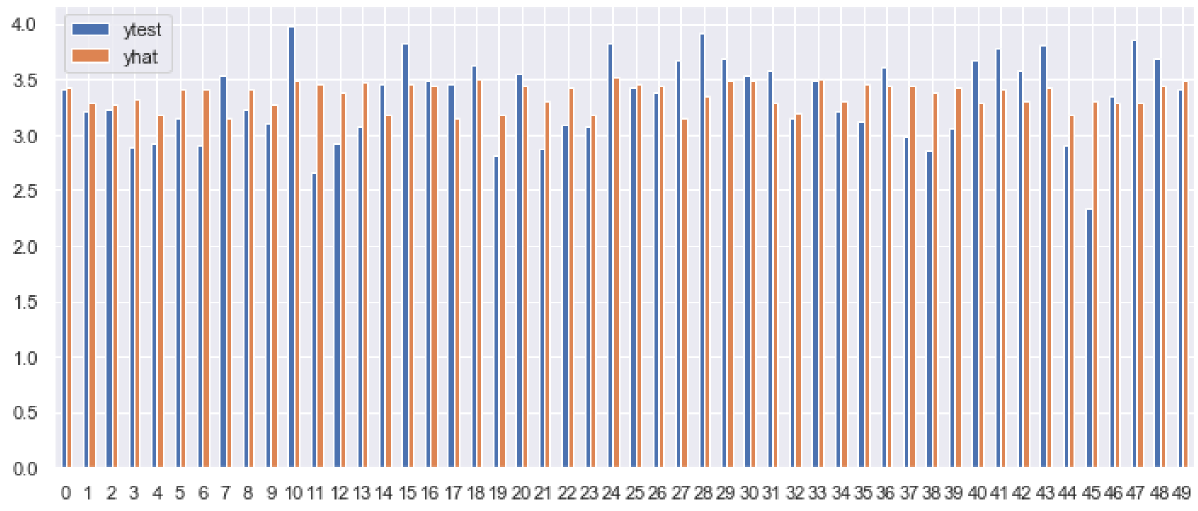
```
Random Forest RMSE: 0.35723425760390753
```

In [129]:
```
print(f'Average movie score with the Random Forest model: {rf_final_yhat.mean()}')
```

```
Average movie score with the Random Forest model: 3.3516830365356327
```

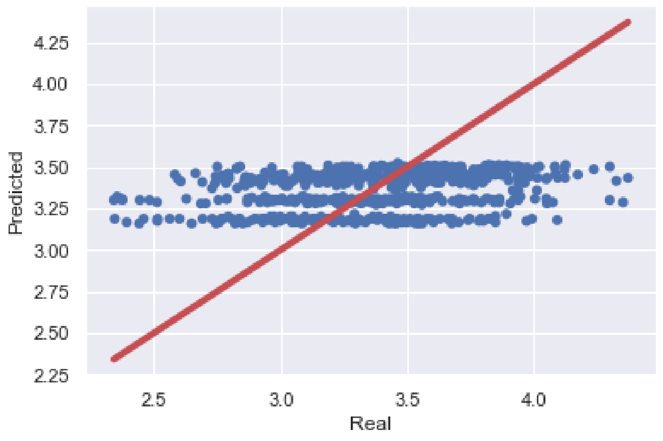
We will visualize the predictions on the test dataset for the first 50 instances.

```
In [130]: # create a temporary dataframe containing two columns
          rf_final_df_tmp = pd.DataFrame({"ytest": ytest[:50], "yhat": rf_final_yhat [:50]})

          # plot the dataframe
          rf_final_df_tmp .plot(figsize=(12,5), kind="bar", rot=0)

Out[130]: <matplotlib.axes._subplots.AxesSubplot at 0x254cad37358>
```



Let's visualise the prediction errors.

```
In [131]: fig, ax = plt.subplots()

          ax.scatter(ytest, rf_final_yhat, edgecolors='None')
          ax.plot([ytest.min(), ytest.max()], [ytest.min(), ytest.max()], 'r', lw=4)
          ax.set_xlabel('Real')
          ax.set_ylabel('Predicted')
          plt.show()
```



Since the beginning, we have the same problem as with the other models, that is to say the lack of prediction accuracy for many observations.

We will have a look at the rmse of the different random forest models and the initial baseline.

```
In [132]: rf_rmse_compa= {
              'Baseline RMSE': [baseline_rmse],
              'Random Forest initial Train': [rf_rmse_score],
              'Random Forest Final Train': [rf_final_rmse],
              'Random Forest Final Test': [rf_test_rmse],
          }

          rf_rmse_df=pd.DataFrame(rf_rmse_compa).transpose()
          rf_rmse_df.columns=['rf_rmse_compa']
          rf_rmse_df["Vs Baseline"] = 100*(baseline_rmse-rf_rmse_df["rf_rmse_compa"])/baseline_rmse
          rf_rmse_df
```

Out[132]:

	rf_rmse_compa	Vs Baseline
Baseline RMSE	0.395299	0.000000
Random Forest initial Train	0.375890	4.909948
Random Forest Final Train	0.376945	4.643073
Random Forest Final Test	0.357234	9.629400

The performance on the test set is slightly better than on the training set (0.3572 vs 0.3769). But overall, they are quite similar. This model reduced the error rate by -9.62% versus the baseline. However, the RMSE is rather high with 0.35.

5.2 Adaboost

```
In [133]: adb_final_yhat = adb_final.predict(Xtest_new)

          adb_test_rmse = np.sqrt(mean_squared_error(ytest, adb_final_yhat))
          print(f'Adaboost RMSE: {adb_test_rmse}')

          Adaboost RMSE: 0.3578527091108525
```

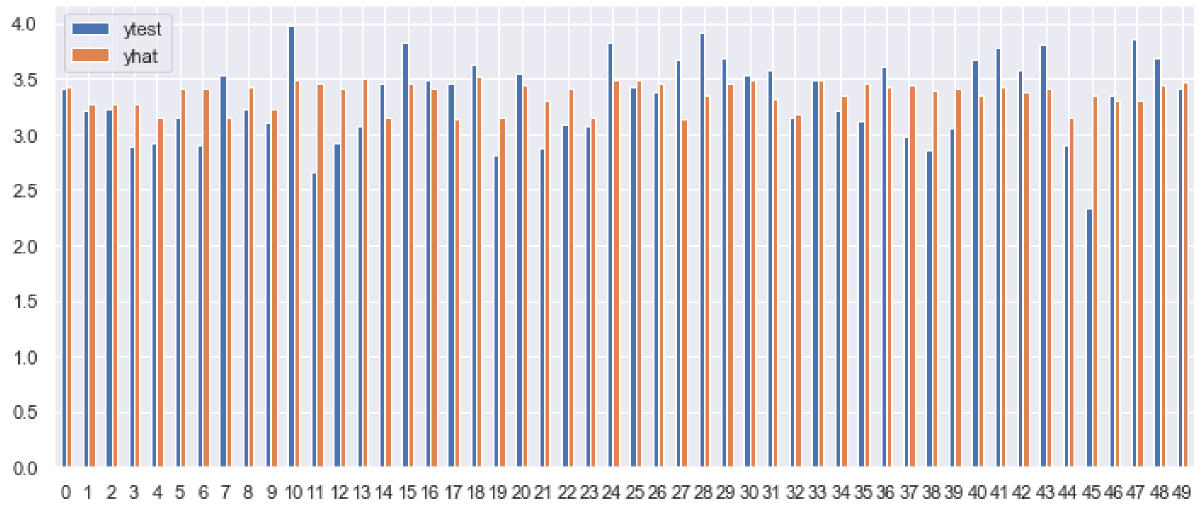
```
In [134]: print(f'Average movie score with the Adaboost model: {adb_final_yhat.mean()}')

          Average movie score with the Adaboost model: 3.3484007136845673
```

The Adaboost model gives on average an average movie rating slightly lower than the Random Forest model.

We will visualize the predictions on the test dataset for the first 50 instances.

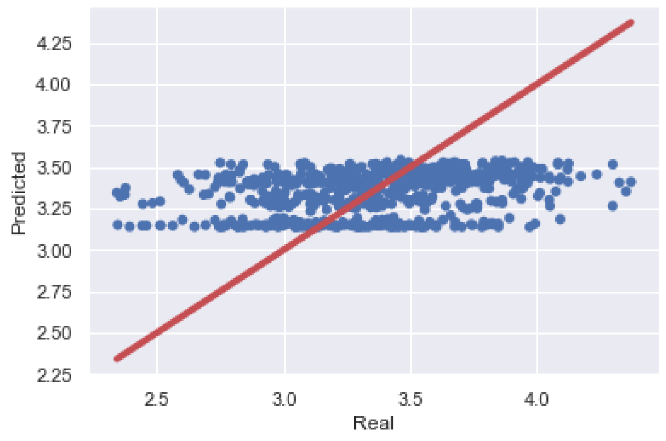
```
In [135]: # create a temporary dataframe containing two columns
          adb_final_df_tmp = pd.DataFrame({"ytest": ytest[:50], "yhat": adb_final_yhat [:50]})

          # plot the dataframe
          adb_final_df_tmp .plot(figsize=(12,5), kind="bar", rot=0)
```

```
Out[135]: <matplotlib.axes._subplots.AxesSubplot at 0x254cd129710>
```



Let's visualise the prediction errors.

In [136]:
```
fig, ax = plt.subplots()

ax.scatter(ytest, adb_final_yhat, edgecolors='None')
ax.plot([ytest.min(), ytest.max()], [ytest.min(), ytest.max()], 'r', lw=4)
ax.set_xlabel('Real')
ax.set_ylabel('Predicted')
plt.show()
```



Lack of prediction accuracy for many observations.

We will have a look at the rmse of the different Adaboost models and the initial baseline.

In [137]:
```
adb_rmse_compa= {
    'Baseline RMSE': [baseline_rmse],
    'Adaboost initial Train': [adb_rmse_score],
    'Adaboost Final Train': [adb_final_rmse],
    'Adaboost Final Test': [adb_test_rmse ],
}

adb_rmse_df=pd.DataFrame(adb_rmse_compa).transpose()
adb_rmse_df.columns=['adb_rmse_compa']
adb_rmse_df["Vs Baseline"] = 100*(baseline_rmse-adb_rmse_df["adb_rmse_compa"])/baseline_rmse
adb_rmse_df
```

Out[137]:

	adb_rmse_compa	Vs Baseline
Baseline RMSE	0.395299	0.000000
Adaboost initial Train	0.377081	4.608837
Adaboost Final Train	0.376620	4.725444
Adaboost Final Test	0.357853	9.472948

The Adaboost RMSE on the test set is as for the random forest model lower than the training set but slighltly higher than the random forest test RMSE.

6.0 Conclusion and Discussion of Results

At the end, the Adaboost and the Random Forest models have roughly similar performances but they have a poor/fair accuracy with a RMSE around 0.35.

We started with a dataset of 17,700 movies and we ended up with a dataset of less than 3,500 movies,. Because we merged the Netflix dataset with the imdb one and remove many outliers and duplicated rows, we had to reduce our dataset by roughly -80%. It is a lot. In my opinion, it had a negative impact on the accuracy of our models.

Thus, at the end, I did not have enough data (because of its initial poor quality) to train my models and enable them to have a good accuracy rate.

Because of the lack of data, my models were fast to train but this is rarely the case in real life.

7.0 Possible Future Improvements and Business Scenarios

Since Netflix will add on a regular basis new movies and new TV shows, we will have to re-merge the training and test sets, re-train-test split the dataset in a random way (and maybe with a higher test size), retrain the model(s) on the new training set, including the new content, and test it again, on a regular basis. This frequency would depend on the speed of the algorithms and of Netflix's computers. Retrain it twice a month or every week may be reasonable. It would be also interesting to use new algorithms to have a better accuracy score.

Then, to improve my models in terms of accuracy, we need more (good) data. Netflix has access to many data that we did not have access to with the Kaggle dataset, for example, how long a viewer watched a movie, which movies s/he watched before... The more (good) data we can have on the viewers, the better we can predict the average rating of a movie, if we should buy the license of this movie and also predict which movies could interest each viewer, based on the movies and shows they previously watched and other variables.

So, there could be:

- a general algorithm for Netflix to decide whether they should buy the license rights of a particular movie,
- another one, which will calculate personalized movie interest scores for each customer for each movie,
- and another one on top of the second one, which will suggest to each viewer movies and TV shows, based on their preferences (thus movie interest scores).

Finally, the algorithm should be improved in a way to be directly linked to the Netflix database or platform, in order to maintain and update it regularly and easily.

In [138]:
```python
# Finish Timer
notebook_duration = round((time.time() - start)/60, 5)
print(f'The completion of the notebook took {notebook_duration} minutes.')
```

```
The completion of the notebook took 2.05408 minutes.
```

In [ ]: