Predicting Movie Ratings and Income from IMDb, Youtube, and Twitter Data

Prince Joseph Erneszer Javier
MSc Data Science
Asian Institute of Management

Executive Summary

Various combinations of categorical data encoding, features scaling, and regression algorithms were run on a movies dataset prepared by Mehreen Ahmed of the National University of Sciences and Technology (NUST), Pakistan. The goal was to predict the gross income and rating based on features sourced from IMDB, Youtube, and Twitter.

23.73% was the highest coefficient of determination r^2 achieved when predicting movie ratings. The regression algorithm used is Ridge Regression with parameter $\alpha = 0.1$. One-hot encoding was applied on the `Genre` feature, and min-max scaling was applied on each of the feature. A total of 11 features were used.

The number of likes on the movie trailer on Youtube is the feature with the largest positive coefficent. The number of dislikes on Youtube trailers has the largest negative coefficient, followed closely by the number of comments on the trailers.

When predicting gross income, the highest r^2 of 61.96% was achieved using kNN Regression using 11 neighbors. One-hot encoding was applied on the genre feature, and standard scaling was applied on each of the feature. A total of five features were used namely the number of screens on which the movie was initially launched in the US and its logarithm, three movie genres.

Regression model results are summarized below:

Variable Predicted	Scaling	Features	Machine Learning Method	Test r^2	Parameter	Optimal Parameter Value
Ratings	Min-Max	11	Ridge	0.2373	C	0.1
Gross Income	Std	5	kNN Reg	0.6196	n neighbors	11.0

The dataset used spanned from 2014 to 2015, and used Twitter and Youtube data. Further studies could explore to expand to capture more years as well as more video and social media sites. Feature engineering may also improve accuracy in the future.

Data Description

The dataset used was compiled by Mehreen Ahmed of the Department of Computer Software Engineering, National University of Sciences and Technology (NUST), Pakistan. The dataset contains 231 movies and 14 columns (12 features and 2 target variables). Features per movie were taken from IMDB, Twitter, and Youtube.

The features are:

1. Genre (Action, Adventure, Drama, etc. mapped to numeric values)
2. Budget from IMDB (US Dollars)
3. Number of Screens on which the movie was initially launched in the US
4. Sequel (1 if first release, 2 if second release, etc.)
5. Aggregate Actor Followers: Sum of followers of top 3 cast from Twitter
6. Number of views on trailers on Youtube
7. Number of likes on trailers on Youtube
8. Number of dislikes on trailers on Youtube
9. Number of comments on trailers on Youtube
10. Sentiment score from Twitter (0 for neutral, + for positive sentiment, and - for negative sentiment)

The target variables are:

1. Ratings (ranges from 1 to 10) collected from IMDB
2. Gross Income in USD from IMDB

Functions and Packages

Below are the functions and packages that were used for exploratory data analysis and predictive modeling. They were placed into this section first as a single location for all coded functions and second, in order not to crowd the rest of the sections below.

Python Packages

`Numpy` gives a wide array of functionality when dealing with arrays (pun intended). `Matplotlib` and `Seaborn` allow us to make charts easily. `Pandas` enable us to transform data into tables that are more understandable and transformable.

We also import `Counter` that allows us to quickly count the number of unique values in a sequence of values like a list, array, or dictionary.

We import seven packages from `sklearn`. `Train_test_split` allows us to split samples in a dataset into test set and training set. Training set is what we feed the regression algorithms, while test set is what we use to test the accuracy of the algorithm. The regression algorithms were implemented using the `sklearn` packages `KNeighborsRegressor`, `LinearRegression`, `Ridge`, and `Lasso`. `TransformerMixin` was used to allow `fit_transform` method in class. `MinMaxScaler` and `StandardScaler` scales data by range (min-max) or by mean and standard deviation (standard). Finally, `LabelEncoder` converts categorical data to numerical dummy variables.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
import seaborn as sns
from collections import Counter

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso

from sklearn.base import TransformerMixin
from sklearn.preprocessing import MinMaxScaler, StandardScaler

from sklearn.preprocessing import LabelEncoder
```

Data Exploration Functions

To aid with data exploration, the function `conv_to_numeric` was developed. This function scans through a dataframe and will try to convert the contents to numerical values. If it's unable to convert the contents automatically, it will notify which columns contained non-numeric values and the values that couldn't be converted. This is useful for example if we want to convert "1500" encoded as string or text to the number 1500 or to identify "2000 pesos" as a value that couldn't be converted. We can then manually change it to 2000.

In [2]:
```
# Defining convertion function
def conv_to_numeric(df, inds_0):
    """
    Convert all values in dataframe to numeric if possible, otherwise, skip.
    Print list of values that cannot be converted and must be converted
    manually. Returns new indices of columns that were not completely
    converted.

    Inputs
    ======
    df: DataFrame of values to convert
    inds_0: List of indices of columns in DataFrame to convert
    """

    # Initial pass: convert to float, ignore data if unconvertable
    for i in inds_0:
        df.iloc[:, i] = df.iloc[:, i].astype(float, errors='ignore')

    # Second pass, show the unconvertable data per feature
    inds_1 = []
    for i in inds_0:
        typ = df.iloc[:, i].dtype
        if typ == object:
            inds_1.append(i)

    print("No. of columns that must be changed to numerical:", len(inds_0))
    print("No. of columns automatically changed to numbers, 1st pass:",
          len(inds_0) - len(inds_1))

    print()
    print("Remaining columns with values that must be changed to numbers:")

    if len(inds_1) == 0:
        print("All values numerical already.")

    else:
        index = []
        cols = []
        vals = []
        for i in inds_1:
            v = df.iloc[:, i][np.logical_not(
                (df.iloc[:, i]).str.isnumeric())].values
            index.append(i)
            cols.append(df.columns[i])
            vals.append(v)

        index2 = pd.Series(index)
        cols2 = pd.Series(cols)
        vals2 = pd.Series(vals)

        df2 = pd.concat([index2, cols2, vals2], axis=1)
        df2.columns = ['Index', 'Names', 'Values']
        display(df2)

    return inds_1
```

The function `column_non_num` accepts a dataframe and returns the column indices and names of columns that contain non-numerical values. This is useful when looking for columns that are expected to contain numerical data, but still contains non-numerical data. For example, if a column for weights include "32 kg", this is read by the computer as non-numerical because of " kg". This function will help us more easily generate a list of those columns.

In [3]:
```
def column_non_num(df):
    """
    Accept dataframe and print columns with index of columns containing
    non-numerical data

    Input
    =====
    df: Dataframe with multiple columns

    Returns
    =======
    List of tuples of indices and columns with non-numerical data
    """
    inds = []
    cols = []
    for i in range(len(df.columns)):
        typ = df.iloc[:, i].dtype
        if typ != 'float64' and typ != 'int64':
            inds.append(i)
            cols.append(df.columns[i])
    print("No. of columns containing non-numerical data:", len(inds))

    return list(zip(inds, cols))
```

To fill in `NaN` values with the median value along a column, we use `DatFrameImputer` class defined below.

In [4]:
```
class DataFrameImputer(TransformerMixin):

    def __init__(self):
        """Impute missing values.

        Columns of dtype object are imputed with the most frequent value
        in column.

        Columns of other types are imputed with median of column.

        """
    def fit(self, X, y=None):

        self.fill = pd.Series([X[c].value_counts().index[0]
            if X[c].dtype == np.dtype('O') else X[c].median() for c in X],
            index=X.columns)

        return self

    def transform(self, X, y=None):
        return X.fillna(self.fill)
```

Regression Models

Below is the function defined for four regression algorithms: k Nearest Neighbors Regression (kNN), Linear Regression, Ridge regression, and Lasso regression. The dataset that will be analyzed is composed of observations or samples with various features and two target variables (movie rating and gross income). The regression models are supervised, which means we know from the start the features and the values of the target variables of the observations we will use to train the models.

The general methodology for "training" the regression algorithms will be as follows: The datapoints/samples/observations in the dataset will be randomized. Then they will be split into two: training set and test set. The training set will be used to train the classification models while the test set, which the models shouldn't have "seen" before will be used to check the models' accuracy. The classification model's objective is to correctly classify a new observation based on its features.

The function below can be used to select one of the four regression models, generate a plot of accuracy vs. model parameters, and output a report showing the maximum average accuracy achieved, the variance of accuracy over multiple iterations, and the optimal parameters.

In [5]:
```
class ML_Regressor:

    def fit(self, feature, target, ml_type="knn_reg",
            param_range=range(1, 30), seed_settings=range(0, 30),
           max_iter=10000):
        """
        Fit data to machine learning regressor. Iterate regression model
        mutiple times. Return the maximum accuracy achieved and the
        corresponding parameter.

        Inputs
        ======
        feature: Dataframe of features
        target: Series of target values
        param_range: Range of values for parameters
        seed_settings: Range of seed settings to run

        Outputs
        =======
        acc_max: Float. Maximum regression accuracy achieved.
        param_max: Float. Regressor parameter that gives maximum accuracy.
        """

        feature = pd.DataFrame(feature)

        self.param_range = param_range
        self.ml_type = ml_type

        train_acc = []
        test_acc = []

        # Initiate counter for number of trials
        self.iterations = 0

        # create an array of cols: parameters and rows: seeds
        for seed in seed_settings:

            # count one trial
            self.iterations += 1

            # split data into test and training sets
            X_train, X_test, y_train, y_test = train_test_split(feature,
                                                                target,
                                                                random_state=seed)
            train = []
            test = []
            coefs = []

            # make a list of accuracies for different parameters
            for param in param_range:
                # build the model
                if ml_type == 'knn_reg':
                    clf = KNeighborsRegressor(n_neighbors=param)

                elif ml_type == 'lin_reg':
                    clf = LinearRegression()

                elif ml_type == 'ridge':
                    clf = Ridge(alpha=param, max_iter=max_iter)

                elif ml_type == 'lasso':
                    clf = Lasso(alpha=param, max_iter=max_iter)

                # fit training set to classifier
                clf.fit(X_train, y_train)

                # record training set accuracy
                train.append(clf.score(X_train, y_train))

                # record generalization accuracy
                test.append(clf.score(X_test, y_test))

                # record coefficients if ml_type != knn_class
                if ml_type != "knn_reg" and param == 0.01:  # get coef @ 0.01
                    coefs.append(clf.coef_)
```

```
            # append the list to _acc arrays
            train_acc.append(train)
            test_acc.append(test)

        # compute mean and error across columns
        self.train_all = np.mean(train_acc, axis=0)
        self.test_all = np.mean(test_acc, axis=0)

        # compute mean coefficients
        if ml_type != "knn_reg":
            self.coefs_all = np.mean(coefs, axis=0).ravel()

        # compute variance of accuracies
        self.var_train = np.var(train_acc, axis=0)
        self.var_test = np.var(test_acc, axis=0)

        # compute the best parameter and maximum accuracy
        self.max_inds = np.argmax(self.test_all)
        self.acc_max = np.amax(self.test_all)
        self.param_max = (self.param_range)[self.max_inds]

        return np.round(self.acc_max, 4), self.param_max

    def plot(self, show_PCC=True, report=True):
        """
        Plot accuracy vs parameter for test and training data. Print
        maximum accuracy and corresponding parameter value. Print number of
        trials.

        Inputs
        ======
        show_PCC: Boolean. will show PCC on plot if True
        report: Boolean. Will show report if True

        Outputs
        =======
        Plot of accuracy vs parameter for test and training data
        Report showing number of maximum accuracy, optimal parameters, PCC,
        and no. of iterations
        """

        if self.ml_type != "knn_reg":
            plt.xscale("log")

        # plot train and errors and standard devs
        plt.plot(self.param_range, self.train_all, c='b',
                 label="training set", marker='.')
        plt.fill_between(self.param_range,
                         self.train_all + self.var_train,
                         self.train_all - self.var_train,
                         color='b', alpha=0.1)

        # plot test and errors and standard devs
        plt.plot(self.param_range, self.test_all,
                 c='r', label="test set", marker='.')
        plt.fill_between(self.param_range,
                         self.test_all + self.var_test,
                         self.test_all - self.var_test,
                         color='r', alpha=0.1)

        plt.xlabel('Parameter Value')
        plt.ylabel('Accuracy')
        plt.title(self.ml_type + ": Accuracy vs Parameter Value")
        plt.legend(loc=0)

        plt.tight_layout()
        plt.show()

        if report == True:
            print('Report:')
            print('=======')
            print("Max average accuracy: {}".format(
                np.round(self.acc_max, 4)))
            print("Var of accuracy at optimal parameter: {0:.4f}".format(
                self.var_test[self.max_inds]))
            print("Optimal parameter: {0:.4f}".format(self.param_max))
            print('Total iterations: {}'.format(self.iterations))
```

Data Preparation

Importing Data

The dataset is saved in an excel file `2014 and 2015 CSM dataset.xlsx`. The dataset was loaded in a pandas dataframe, `df`, which organizes the data in table form. Each column in the pandas dataframe corresponds to a feature of an observation or sample. For example, one feature is the movie budget. Meanwhile, the rows represent each observation or sample, which in this case, is each movie.

The dataset contains 14 columns and 231 rows or samples.

In [6]:
```
df = pd.read_excel("2014 and 2015 CSM dataset.xlsx")
```

In [7]:
```
# cols and rows
print("No. of features/columns in the dataset: {}".format(df.shape[1]))
print("No. of samples/rows in the dataset: {}".format(df.shape[0]))
```

```
No. of features/columns in the dataset: 14
No. of samples/rows in the dataset: 231
```

In [8]:
```
_ = df.copy()

print("Showing the first three rows:")
print("==========")
display(_.head(3))

print()
print("Showing the last three rows:")
print("==========")
display(_.tail(3))
```

```
Showing the first three rows:
==========
```

	Movie	Year	Ratings	Genre	Gross	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislikes	C
0	13 Sins	2014	6.3	8	9130	4000000.0	45.0	1	0	3280543	4632	425	63
1	22 Jump Street	2014	7.1	1	192000000	50000000.0	3306.0	2	2	583289	3465	61	18
2	3 Days to Kill	2014	6.2	1	30700000	28000000.0	2872.0	1	0	304861	328	34	47

```
Showing the last three rows:
==========
```

	Movie	Year	Ratings	Genre	Gross	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislik
228	Unfinished Business	2015	5.4	8	10200000	35000000.0	2777.0	1	7	3450614	6823	325
229	War Room	2015	5.4	1	12300000	3000000.0	NaN	1	10	66872	400	67
230	The Gallows	2015	4.4	15	22600000	100000.0	2720.0	1	-5	659772	2841	431

In [9]:
```
# columns
[print(i) for i in df.columns];
```

```
Movie
Year
Ratings
Genre
Gross
Budget
Screens
Sequel
Sentiment
Views
Likes
Dislikes
Comments
Aggregate Followers
```

Cleaning the Data

A clean dataset is imperative to a good model. We first checked that there are three columns with some null values. And one column `Aggregate Followers` has 35 null values or 15% of the total data. We decided to drop this column.

In [10]:
```
# percent null values per column in ascending order
a = np.sum([df.isnull().any()])
print("Percent of columns with null: {}".format(a))
print("===========")
(df.isnull().sum() / len(df) * 100).sort_values(ascending=False)
```

```
Percent of columns with null: 3
===========
```

Out[10]:
```
Aggregate Followers    15.151515
Screens                 4.329004
Budget                  0.432900
Comments                0.000000
Dislikes                0.000000
Likes                   0.000000
Views                   0.000000
Sentiment               0.000000
Sequel                  0.000000
Gross                   0.000000
Genre                   0.000000
Ratings                 0.000000
Year                    0.000000
Movie                   0.000000
dtype: float64
```

44 rows have null values. Two of these rows have null values of 14%.

In [11]:
```
# percent null values per row descending order, top 10
a = df[df.isnull().any(axis=1)].shape[0]
print("No. of rows with null: {}".format(a))
_ = df[df.isnull().any(axis=1)].isnull().sum(axis=1) / df.shape[1] * 100
print("10 rows with highest number of null values, in percent")
_.sort_values(ascending = False)[:10]
```

```
No. of rows with null: 44
10 rows with highest number of null values, in percent
```

Out[11]:
```
115    14.285714
229    14.285714
109     7.142857
98      7.142857
95      7.142857
94      7.142857
84      7.142857
80      7.142857
76      7.142857
69      7.142857
dtype: float64
```

We drop columns with null values greater than 15%.

In [12]:
```
# drop all columns with na exceeding threshold.

prop = 0.15  # maximum set proportion of null values per column, drop if exceeded
thresh = (df.shape[0] * (1 - prop)) # proportion of num of rows

_df = df.dropna(thresh=thresh, axis=1).reset_index(drop=True)  # drop rows that have < non-null thresh values
display(_df.isnull().sum())

# shape after dropping
display(_df.shape)
```

```
Movie         0
Year          0
Ratings       0
Genre         0
Gross         0
Budget        1
Screens      10
Sequel        0
Sentiment     0
Views         0
Likes         0
Dislikes      0
Comments      0
dtype: int64
```

```
(231, 13)
```

There's only one column with non-numerical data: `Movie`, which is as expected.

In [13]:
```
# non-numerical data
a = column_non_num(_df)
inds = pd.Series([i for i, j in a], name='Index')
cats = pd.Series([j for i, j in a], name='Feature Name')

_ = pd.concat([inds, cats], axis=1)
display(_)
```

```
No. of columns containing non-numerical data: 1
```

	Index	Feature Name
0	0	Movie

In [14]:
```
_df4 = _df.copy()
```

Missing values were filled in with the median value per column. The final cleaned dataframe is saved in df_clean.

In [15]:
```
# Impute missing values in the dataset
print("No. of null values prior to imputing:", _df4.isnull().sum().sum())
_df5 = DataFrameImputer().fit_transform(_df4)
print("No. of null values after imputing:", _df5.isnull().sum().sum())
# shape after imputing
display(_df5.shape)
```

```
No. of null values prior to imputing: 11
No. of null values after imputing: 0
```

```
(231, 13)
```

In [16]:
```
# saving to df_clean
df_clean = _df5.copy()
df_clean.head()
```

Out[16]:

	Movie	Year	Ratings	Genre	Gross	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislikes
0	13 Sins	2014	6.3	8	9130	4000000.0	45.0	1	0	3280543	4632	425
1	22 Jump Street	2014	7.1	1	192000000	50000000.0	3306.0	2	2	583289	3465	61
2	3 Days to Kill	2014	6.2	1	30700000	28000000.0	2872.0	1	0	304861	328	34
3	300: Rise of an Empire	2014	6.3	1	106000000	110000000.0	3470.0	2	0	452917	2429	132
4	A Haunted House 2	2014	4.7	8	17300000	3500000.0	2310.0	2	0	3145573	12163	610

In [17]:
```
# No more null values
df_clean.isnull().sum().sum()
```

Out[17]: 0

Separating Categorical Variables from Numerical Variables

In [18]:
```
df_sample_0 = df_clean.copy().drop(['Movie', 'Year'], axis=1)
```

We added logarithm values for each numerical variable.

In [19]:
```
temp = df_sample_0.drop(['Ratings', 'Gross'], axis=1)
for c in temp.columns:
    if c != 'Sentiment' and c !='Genre':
        temp[c+'_log'] = np.log(df_sample_0[c] + 10**-30)
```

Separate dataframes were created for categorical and numerical variables. `Genre` is the only categorical variable, although it's encoded as numbers.

In [20]:
```
# categorical variables
cats = ["Genre"]
nums = [i for i in temp.columns if i not in cats]
```

In [21]:
```
df_sample = temp.copy()
```

The two target variables are `Ratings` and `Gross`. Two series `_y1` and `_y2` were created to contain these variables.

In [22]:
```
# split into target, categorical, and numerical data
target1 = 'Ratings'
target2 = 'Gross'

_y1 = df_sample_0[target1]
_y2 = df_sample_0[target2]
```

In [23]:
```
_y1.head()
```

Out[23]:
```
0    6.3
1    7.1
2    6.2
3    6.3
4    4.7
Name: Ratings, dtype: float64
```

In [24]:
```
_y2.head()
```

Out[24]:
```
0         9130
1    192000000
2     30700000
3    106000000
4     17300000
Name: Gross, dtype: int64
```

Categorical variables were saved in `df_cat` and numerical variables were saved in `df_num`. `Year` was removed from the dataframe.

In [25]:
```
df_cat = df_sample.loc[:, cats]
df_num = df_sample.loc[:, nums]
```

In [26]:
```
df_cat.head()
```

Out[26]:

	Genre
0	8
1	1
2	1
3	1
4	8

In [27]:
```
df_num.head()
```

Out[27]:

	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislikes	Comments	Budget_log	Screens_log	Sec
0	4000000.0	45.0	1	0	3280543	4632	425	636	15.201805	3.806662	0.0(
1	50000000.0	3306.0	2	2	583289	3465	61	186	17.727534	8.103494	0.69
2	28000000.0	2872.0	1	0	304861	328	34	47	17.147715	7.962764	0.0(
3	110000000.0	3470.0	2	0	452917	2429	132	590	18.515991	8.151910	0.69
4	3500000.0	2310.0	2	0	3145573	12163	610	1082	15.068274	7.745003	0.69

The two target variables were resaved to `y1` and `y2.`

In [28]:
```
y1 = _y1[:]
y2 = _y2[:]
```

Finally, we make another set of dataframes, this time, dataframes of "undummified" categories plus the targets added to the last column. The same was done for numerical categories. These will be used for data exploration below.

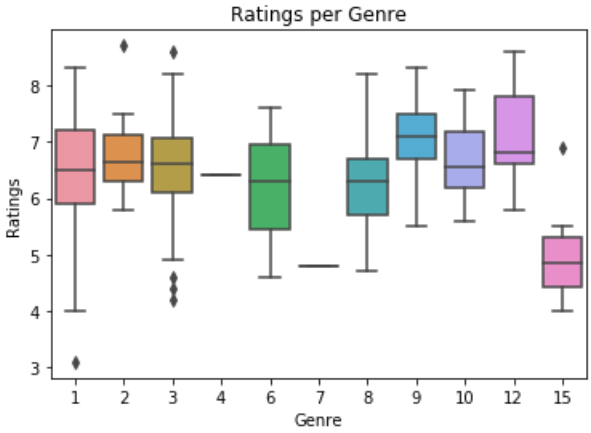
In [29]:
```
df_cat_targ1 = pd.concat([df_cat, y1], axis=1) # get categories with targets 1
df_num_targ1 = pd.concat([df_num, y1], axis=1) # get num categories with targets 1
```

In [30]:
```
df_cat_targ2 = pd.concat([df_cat, y2], axis=1) # get categories with targets 2
df_num_targ2 = pd.concat([df_num, y2], axis=1) # get num categories with targets 2
```

Data Exploration

The chart below shows the distributions of ratings per genre. Movies under genre 15 have the lowest ratings. Movies under genre 1 tend to have the widest range of scores. Movies under genre 9 have the highest median of ratings.

In [31]:
```
plt.title("Ratings per Genre")
sns.boxplot(x="Genre", y="Ratings", data=df_cat_targ1)
plt.show()
```



The movie in the dataset with the highest rating of 8.7 is Interstellar (2014). The movie with the lowest rating of 3.1 is Left Behind (2014).

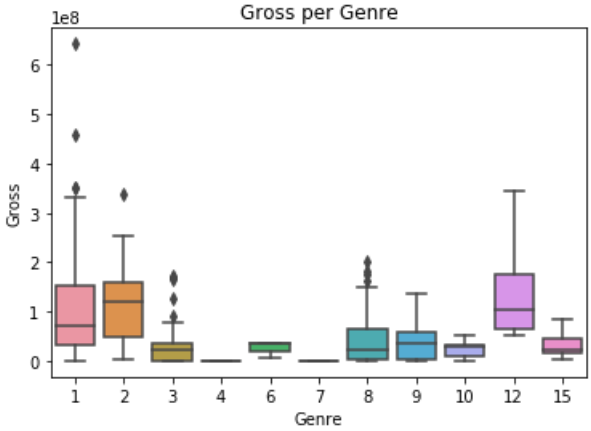
In [32]:
```
display(df_clean[df_clean['Ratings'] == max(df_clean['Ratings'])])
display(df_clean[df_clean['Ratings'] == min(df_clean['Ratings'])])
```

	Movie	Year	Ratings	Genre	Gross	Budget	Screens	Sequel	Sentiment	Views	Likes	Disli
55	Interstellar	2014	8.7	2	188000000	165000000.0	3561.0	1	2	5421705	16635	751

	Movie	Year	Ratings	Genre	Gross	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislikes	C
65	Left Behind	2014	3.1	1	14000000	16000000.0	1825.0	1	0	5611593	11187	2111	7

The chart below shows the distribution of gross income per genre. Genre 1 has the widest range of gross income. Movies under Genre 2 have the highest median gross income.

In [33]:
```
plt.title("Gross per Genre")
sns.boxplot(x="Genre", y="Gross", data=df_cat_targ2)
plt.show()
```



The movie in the dataset with the highest gross income is Jurassic World (2015) while the movie with the lowest gross income is Locker 13.

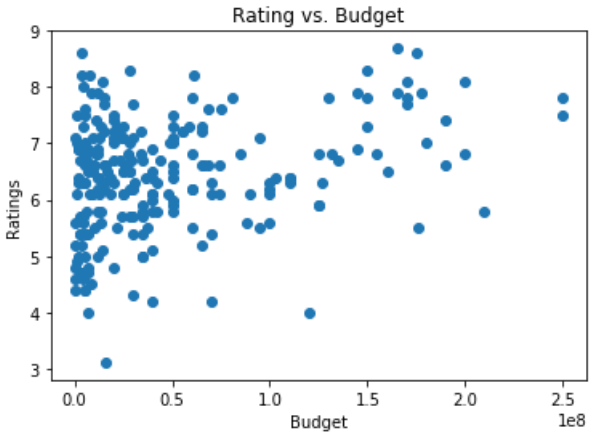
In [34]:
```
display(df_clean[df_clean['Gross'] == max(df_clean['Gross'])])
display(df_clean[df_clean['Gross'] == min(df_clean['Gross'])])
```

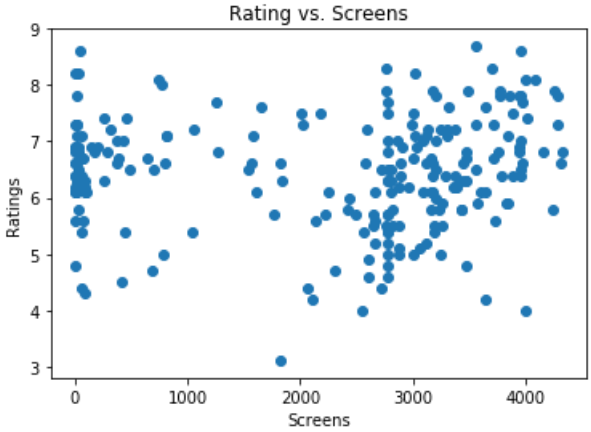
	Movie	Year	Ratings	Genre	Gross	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislik
163	Jurassic World	2015	7.3	1	643000000	150000000.0	4274.0	4	1	9143740	34746	1074

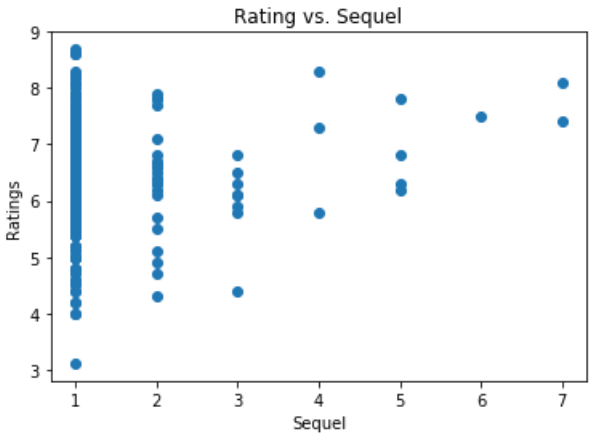
	Movie	Year	Ratings	Genre	Gross	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislikes	Commen
68	Locker 13	2014	4.8	7	2470	300000.0	3.0	1	0	30529	18	4	2

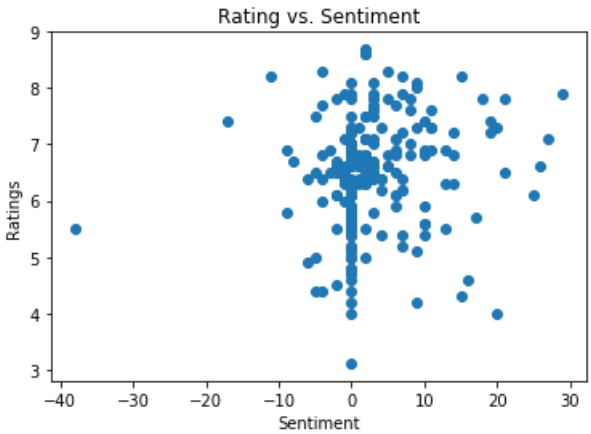
Scatterplots below show relationship between ratings and numerical features in the dataset. Visually, it looks that Ratings are more correlated with budget and sequel.

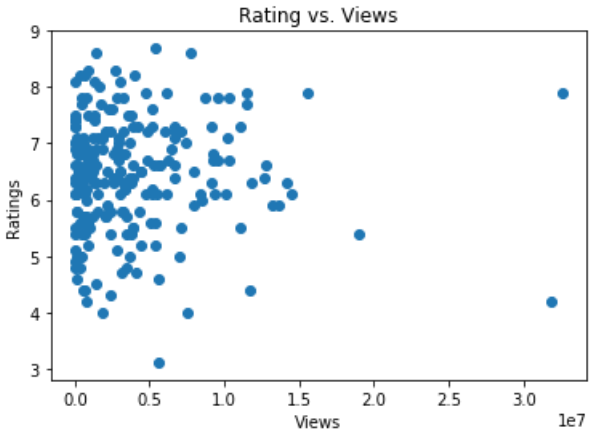
In [35]:
```
for col in range(len(df_num_targ1.columns)-1):
    plt.title("Rating vs. {}".format(df_num_targ1.columns[col]))
    plt.ylabel("Ratings")
    plt.xlabel(df_num_targ1.columns[col])
    plt.scatter((df_num_targ1.iloc[:, col]), y1)
    plt.show()
```

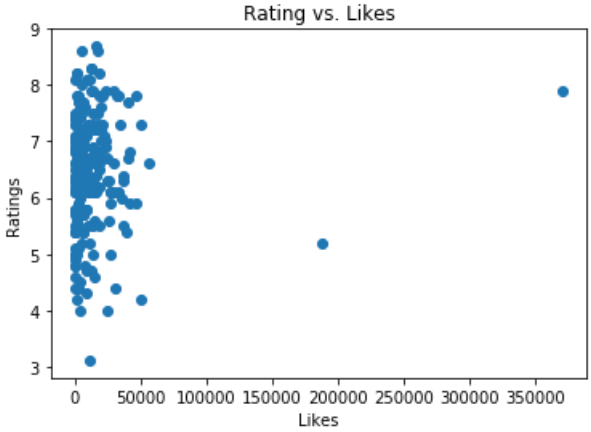

Rating vs. Budget
Ratings
Budget
1e8

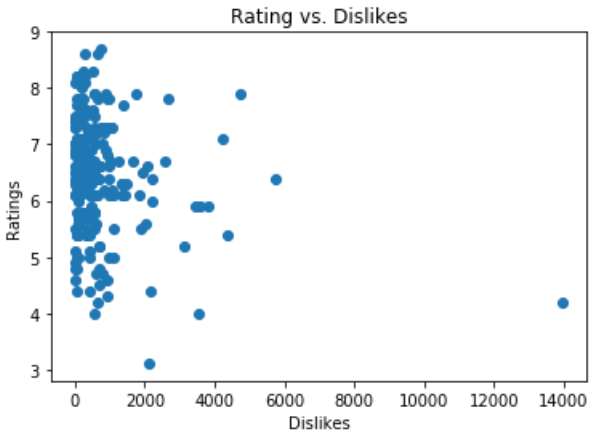

Rating vs. Screens
Ratings
Screens

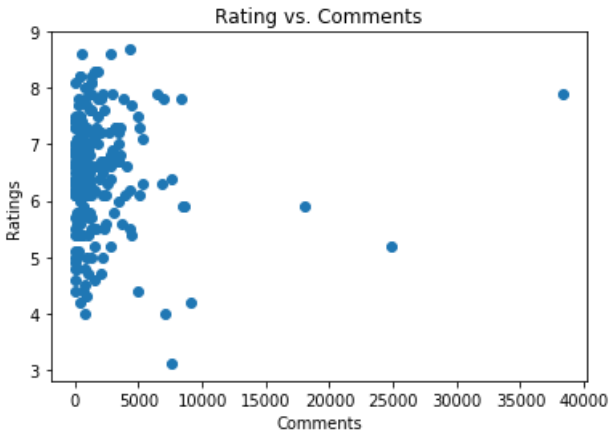

Rating vs. Sequel
Ratings
Sequel

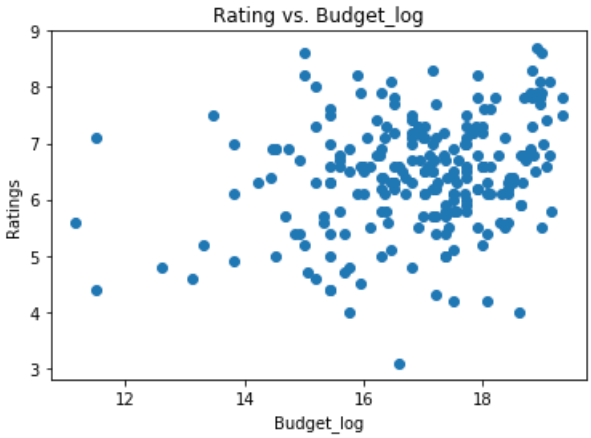

Rating vs. Sentiment
Ratings
Sentiment

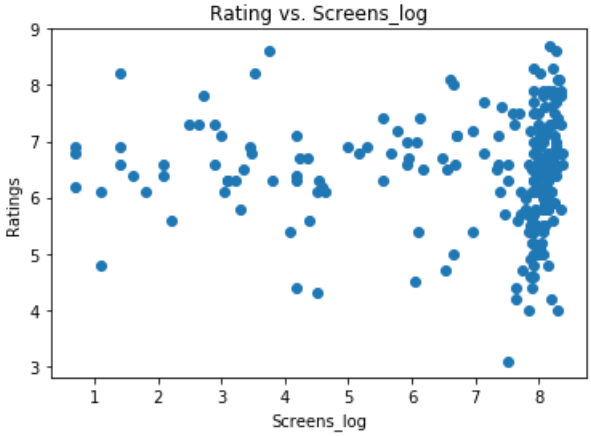

Rating vs. Views
Ratings
Views
1e7

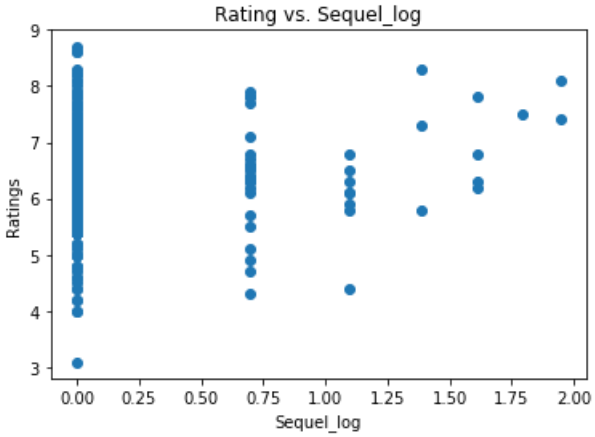

Rating vs. Likes
Ratings
Likes

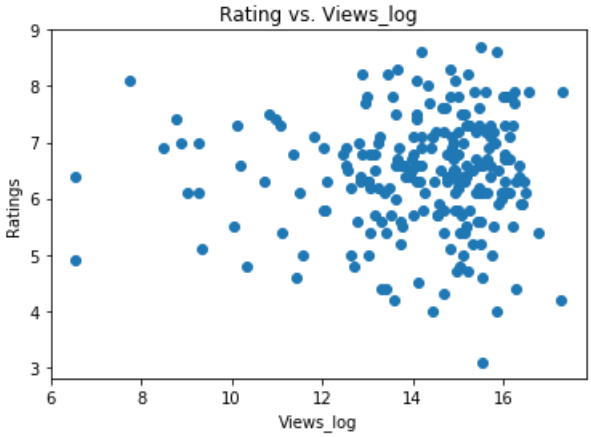

Rating vs. Dislikes
Ratings
Dislikes

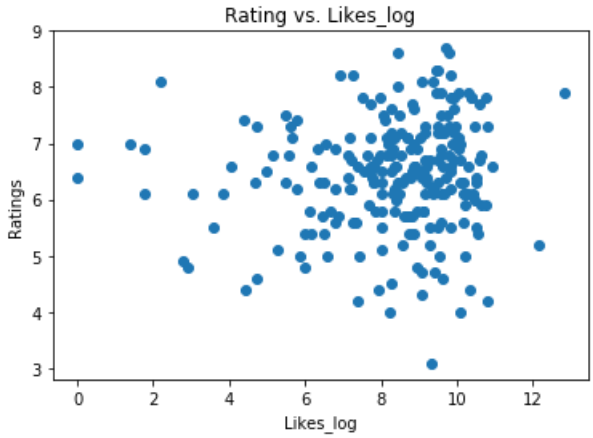

Rating vs. Comments
Ratings
Comments

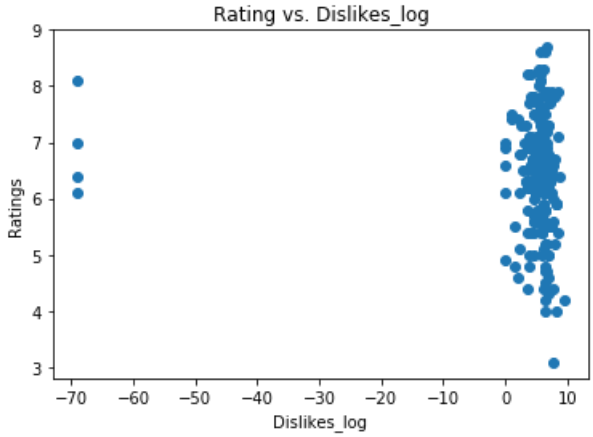

Rating vs. Budget_log
Ratings
Budget_log

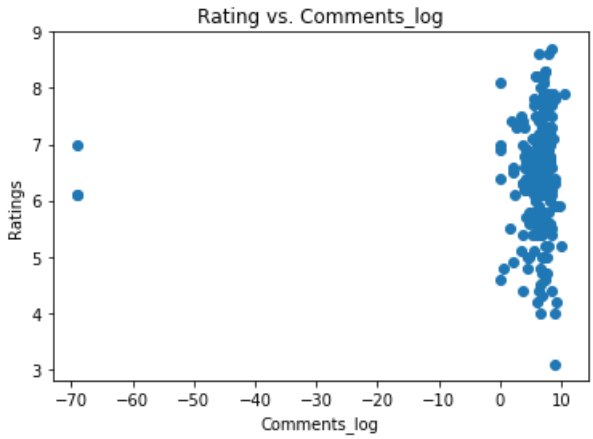

Rating vs. Screens_log
Ratings
Screens_log


Rating vs. Sequel_log
Ratings
Sequel_log


Rating vs. Views_log
Ratings
Views_log

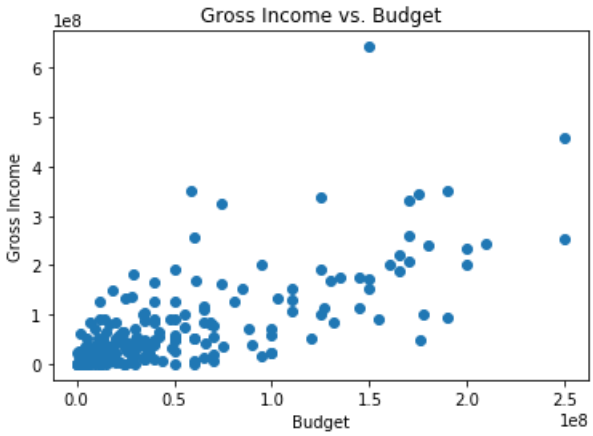


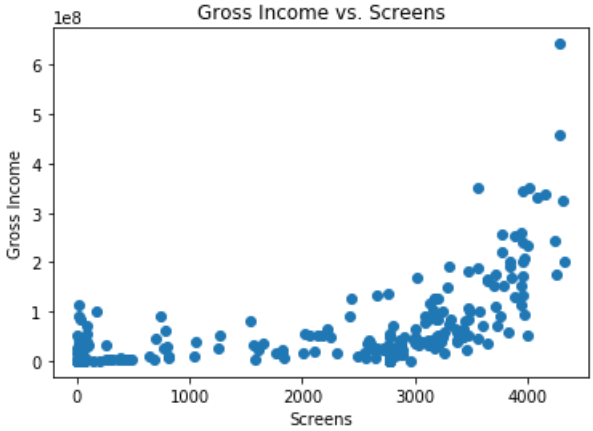


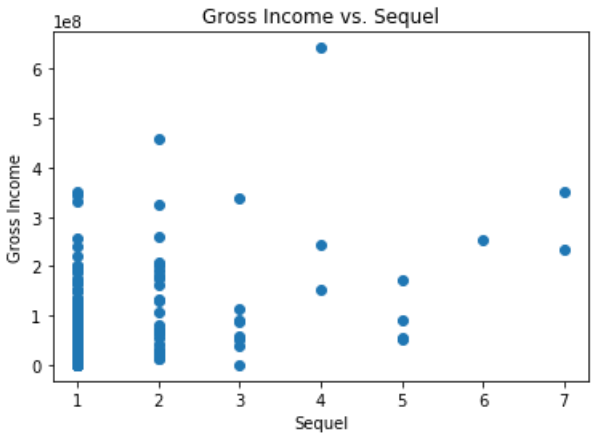


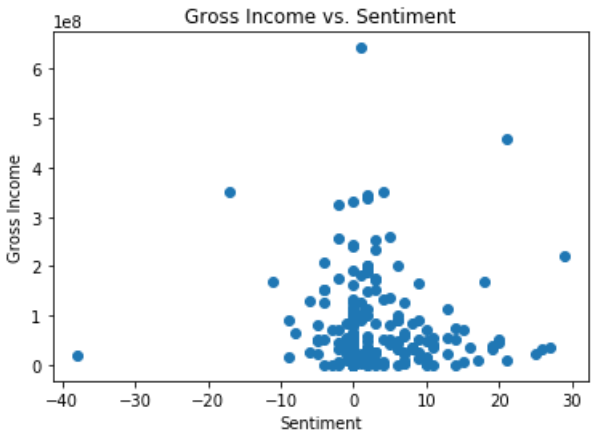
Scatterplots below show relationship between Gross Income and numerical features in the dataset. Visually, it looks that gross income is correlated with budget and screens.

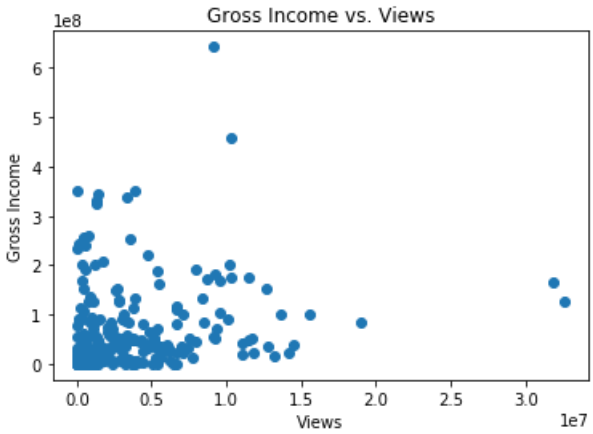
In [37]:
```
for col in range(len(df_num_targ2.columns)-1):
    plt.title("Gross Income vs. {}".format(df_num_targ2.columns[col]))
    plt.ylabel("Gross Income")
    plt.xlabel(df_num_targ2.columns[col])
    plt.scatter(df_num_targ2.iloc[:, col], y2)
    plt.show()
```

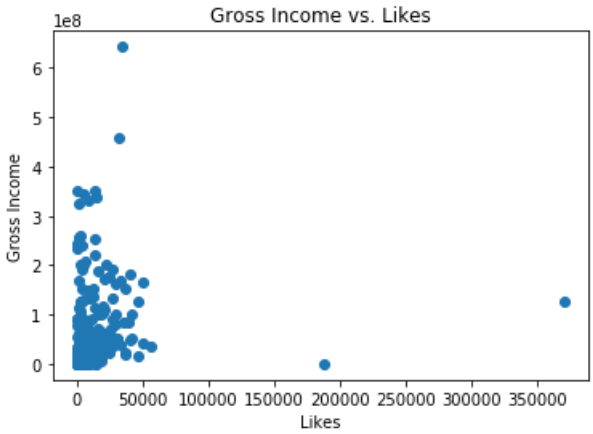

Gross Income vs. Budget
1e8
Gross Income
Budget
1e8

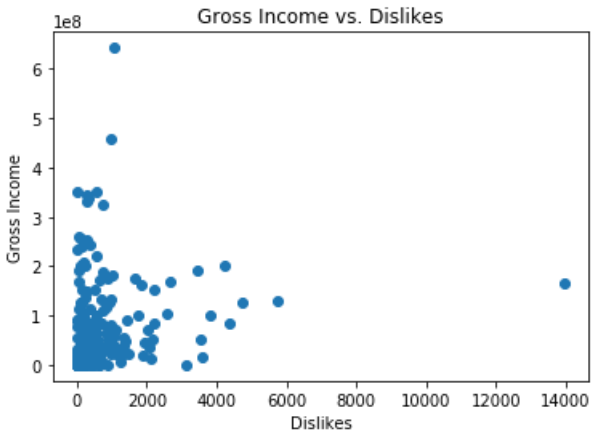

Gross Income vs. Screens
1e8
Gross Income
Screens

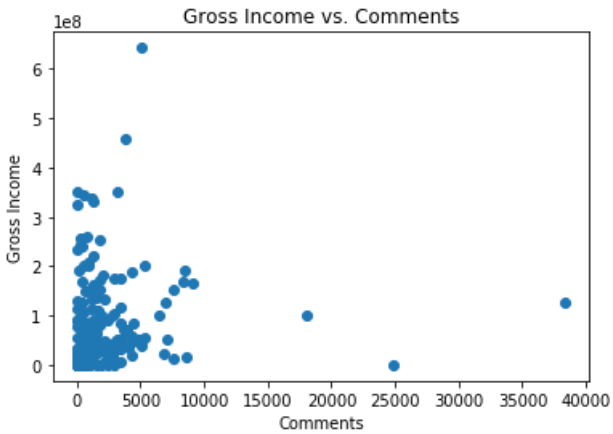

Gross Income vs. Sequel
1e8
Gross Income
Sequel

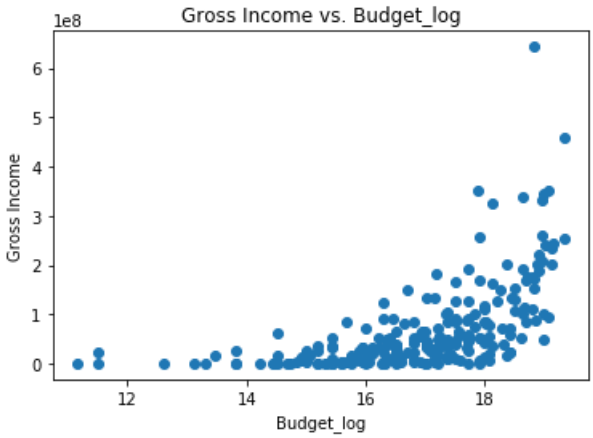

Gross Income vs. Sentiment
1e8
Gross Income
Sentiment

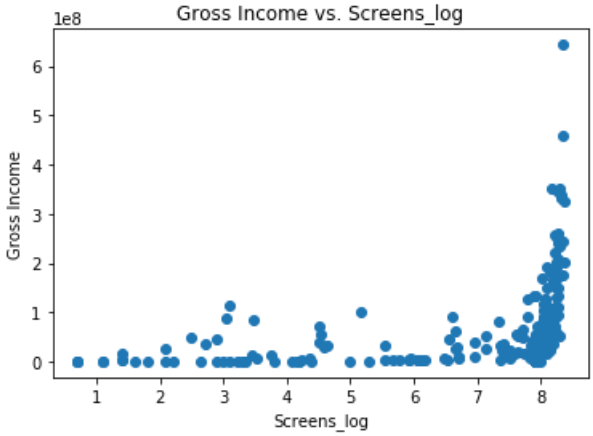

Gross Income vs. Views
1e8
Gross Income
Views
1e7

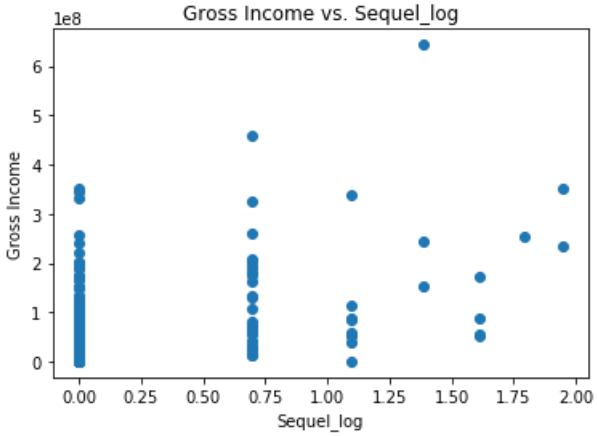

Gross Income vs. Likes
1e8
Gross Income
Likes

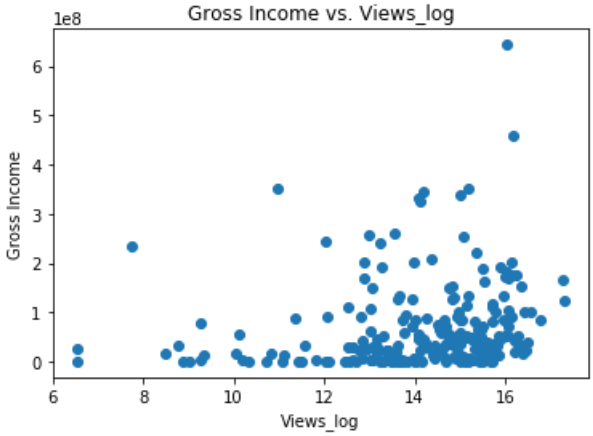

Gross Income vs. Dislikes
1e8
Gross Income
Dislikes

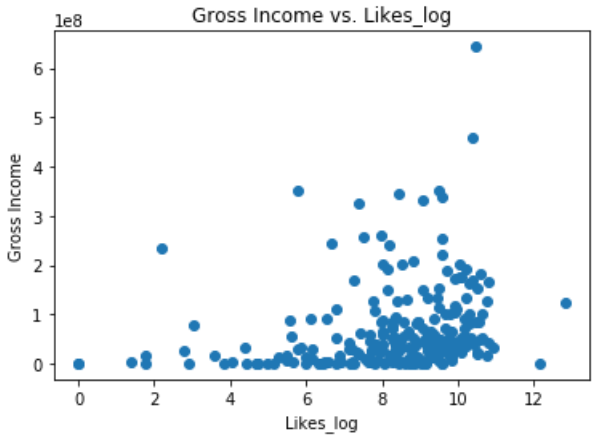

Gross Income vs. Comments
1e8
Gross Income
Comments

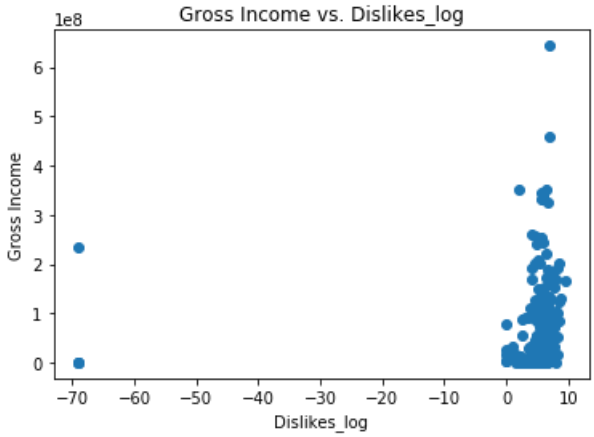

Gross Income vs. Budget_log
1e8
Gross Income
Budget_log

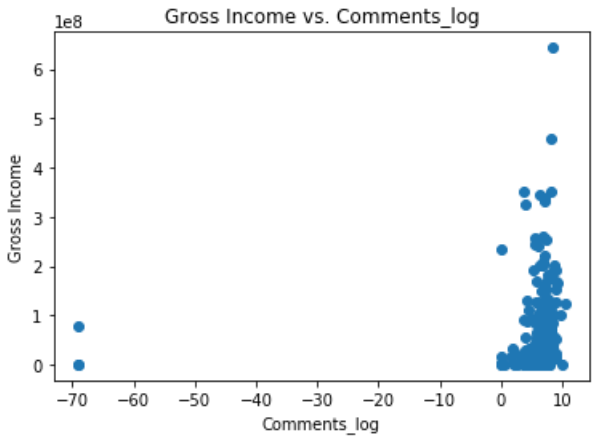

Gross Income vs. Screens_log
1e8
Gross Income
Screens_log


Gross Income vs. Sequel_log
1e8
Gross Income
Sequel_log


Gross Income vs. Views_log
1e8
Gross Income
Views_log







Predicting Movie Ratings

K Nearest Neighbors Regression, Linear Regression, Ridge Regression, and Lasso Regression were the four regression algorithms used to predict Movie `Ratings`. Naturally, `Ratings` and `Gross Income`, the two variables being predicted, were removed from the features. The results were compared with each other and the algorithm that yields the highest "accuracy" measured in r^2 was identified.

Individual Features

The function below checks the accuracy of the regression model for each feature then returns the list of features from the one that gives the highest accuracy to the lowest accuracy.

In [38]:
```
def ml_each_feature(X, y, t="knn_reg", params=range(1, 50),
                    seeds=range(0, 30), show=False):

    """
    Return a list of features and corresponding classification accuracy
    when each feature is taken one by one.
    """

    # First pass, checking each feature
    acc = []
    ml = ML_Regressor()

    for i in range(len(X.columns)):
        x = pd.DataFrame(X.iloc[:, i])

        a, p = ml.fit(x, y, ml_type=t, param_range=params,
                      seed_settings=seeds)
        acc.append(a)

    inds = np.argsort(acc)[::-1]
    sorted_acc = np.array(acc)[inds]
    cols = np.array(X.columns)[inds]

    if show == True:
        g = pd.DataFrame(sorted_acc, cols)
        g.columns = ['Accuracy']
        display(g.head(5))  # display top 5 predictors

    return cols  # return features from best predictor to least predictor
```

Combinations of Features

The function below checks combinations of features that give highest regression accuracy. It works by first calculating the accuracy using the most predictive feature, then adds the next most predictive feature. If accuracy increases, then the features are taken together before checking the next feature. If the accuracy decreases upon adding another feature, then this feature is skipped moving to the next one.

In [39]:
```
def optimize_feats(X, y, cols, t="knn_reg", params=range(1, 50),
                   seeds=range(0, 30), plot=True, max_iter=10000):

    """
    Get features that will maximize classificationa accuracy
    """

    ml = ML_Regressor()

    lst = []  # indices of optimal feats
    accs = [0]  # accuracies
    p_opts = [0]  # optimal parameters

    for i in range(0, len(X.columns)):
        lst2 = lst[:]
        lst2.append(i)
        acc, p_opt = ml.fit(pd.DataFrame(X[cols[lst2]]), y, ml_type=t,
                            param_range=params, seed_settings=seeds, max_iter=max_iter)
        if acc > accs[-1]:
            accs.append(acc)
            p_opts.append(p_opt)
            lst.append(i)

    if plot == True:
        ml.fit(X[cols[lst]], y, ml_type=t,
               param_range=params, seed_settings=seeds, max_iter=max_iter)
        ml.plot(show_PCC=False)

    return accs[-1], p_opts[-1], lst
```

One-Hot Encoding

First, the `Genre` feature was transformed to one-hot encoding. The idea is that per genre, the movie can either be under that genre (value = 1) or not (value = 0).

In [40]:
```
enc="One-Hot"
```

In [41]:
```
# One-hot encoding
df_cat2 = pd.get_dummies(df_cat.astype("str"), drop_first=True)
df_cat2.head()
```

Out[41]:

	Genre_10	Genre_12	Genre_15	Genre_2	Genre_3	Genre_4	Genre_6	Genre_7	Genre_8	Genre_9
0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	1	0

The dataframe with one-hot encoded genre is shown below.

In [42]:
```
X0 = pd.concat([df_num_targ1, df_cat2], axis=1).drop("Ratings", axis=1)
conv_to_numeric(X0, range(0, len(X0.columns)));
X0.head()
```

```
No. of columns that must be changed to numerical: 25
No. of columns automatically changed to numbers, 1st pass: 25

Remaining columns with values that must be changed to numbers:
All values numerical already.
```

Out[42]:

	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislikes	Comments	Budget_log	Screens_log
0	4000000.0	45.0	1.0	0.0	3280543.0	4632.0	425.0	636.0	15.201805	3.806662
1	50000000.0	3306.0	2.0	2.0	583289.0	3465.0	61.0	186.0	17.727534	8.103494
2	28000000.0	2872.0	1.0	0.0	304861.0	328.0	34.0	47.0	17.147715	7.962764
3	110000000.0	3470.0	2.0	0.0	452917.0	2429.0	132.0	590.0	18.515991	8.151910
4	3500000.0	2310.0	2.0	0.0	3145573.0	12163.0	610.0	1082.0	15.068274	7.745003

5 rows × 25 columns

Scaling

Two scaling methods were used: min-max scaling or standard scaling. Min-max scaling scales the values in a column by the range of values in that column (minimum values to maximum values). Standard scaling scales the values in a column by the mean and standard deviation of the values in that column.

In [43]:
```
# min-max scaling
mms = MinMaxScaler()
X_mm = pd.DataFrame(mms.fit_transform(X0), columns=X0.columns)
X_mm.head()
```

Out[43]:

	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislikes	Comments	Budget_log	Screens_log
0	0.015724	0.009949	0.000000	0.567164	0.100528	0.012498	0.030444	0.016578	0.494523	0.405470
1	0.199776	0.764461	0.166667	0.597015	0.017857	0.009348	0.004370	0.004848	0.803265	0.965041
2	0.111751	0.664044	0.000000	0.567164	0.009323	0.000882	0.002436	0.001225	0.732388	0.946714
3	0.439843	0.802406	0.166667	0.567164	0.013861	0.006552	0.009456	0.015379	0.899644	0.971346
4	0.013724	0.534012	0.166667	0.567164	0.096391	0.032821	0.043696	0.028204	0.478200	0.918355

5 rows × 25 columns

In [44]:
```
# standard scaling
stdsc = StandardScaler()

X_std=pd.DataFrame(stdsc.fit_transform(X0), columns=X0.columns)
X_std.head()
```

Out[44]:

	Budget	Screens	Sequel	Sentiment	Views	Likes	Dislikes	Comments	Budget_log	Screens_
0	-0.810739	-1.527274	-0.372283	-0.402417	-0.096040	-0.281630	-0.204677	-0.333876	-1.135265	-1.504973
1	0.040033	0.748123	0.663831	-0.115951	-0.695253	-0.322203	-0.497933	-0.460163	0.549579	0.609956
2	-0.366858	0.445295	-0.372283	-0.402417	-0.757108	-0.431267	-0.519686	-0.499172	0.162798	0.540688
3	1.149735	0.862556	0.663831	-0.402417	-0.724216	-0.358222	-0.440732	-0.346785	1.075538	0.633787
4	-0.819987	0.053154	0.663831	-0.402417	-0.126025	-0.019801	-0.055632	-0.208711	-1.224340	0.433504

5 rows × 25 columns

Ratings values stored in `y1` was copied to `y` for modelign purposes.

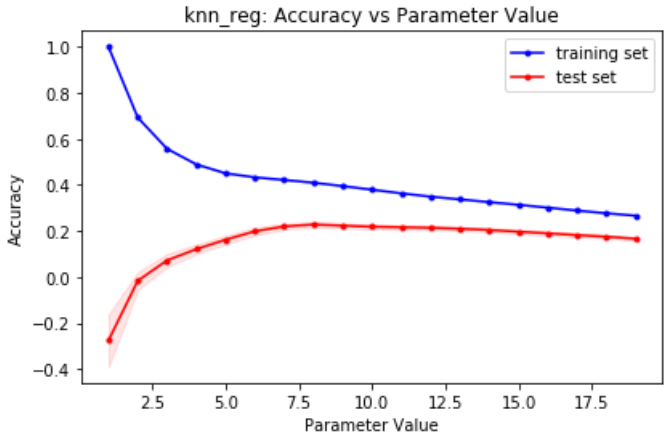
In [45]:
```
y = y1
```

The number of iterations was set to 100 as coded through the seeds parameter from 0 to 99 below.

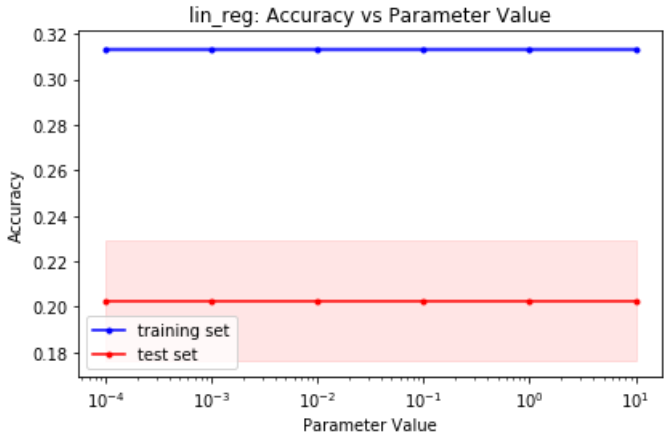
In [46]:
```
seeds = range(0, 100) # 0 to 99 excluding 100
```

X Min-Max Scaled

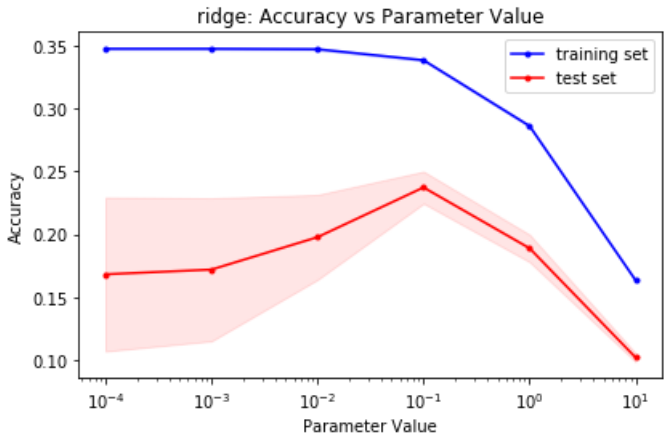
In [47]:
```
X = X_mm

params_knn = range(1, 20)
params_lr_svd = [0.0001, 0.001, 0.01, 0.1, 1, 10]

accuracies = []
opt_param = []
num_feats = []

types = ['knn_reg', 'lin_reg', 'ridge', 'lasso']

for t in types:
    if t == "knn_reg":
        params = params_knn
        cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
        a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                   seeds=seeds, plot=True)

        accuracies.append(a)
        opt_param.append(p)
        num_feats.append(len(ind))

    else:
        params = params_lr_svd
        cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
        a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                   seeds=seeds, plot=True, max_iter=10000)

        accuracies.append(a)
        opt_param.append(p)
        num_feats.append(len(ind))
```



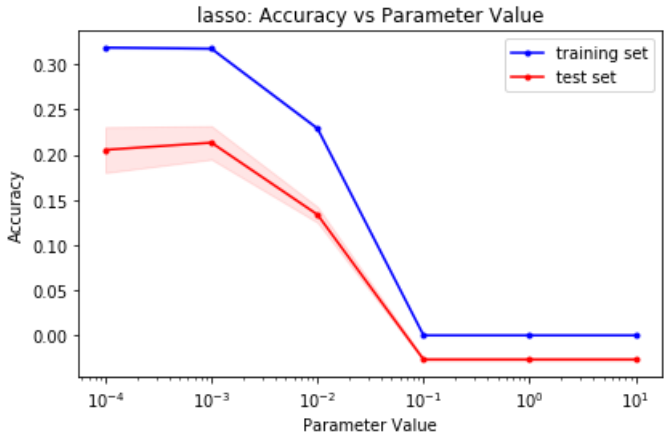

```
Report:
=======
Max average accuracy: 0.2281
Var of accuracy at optimal parameter: 0.0126
Optimal parameter: 8.0000
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.2026
Var of accuracy at optimal parameter: 0.0264
Optimal parameter: 0.0001
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.2373
Var of accuracy at optimal parameter: 0.0128
Optimal parameter: 0.1000
Total iterations: 100
```



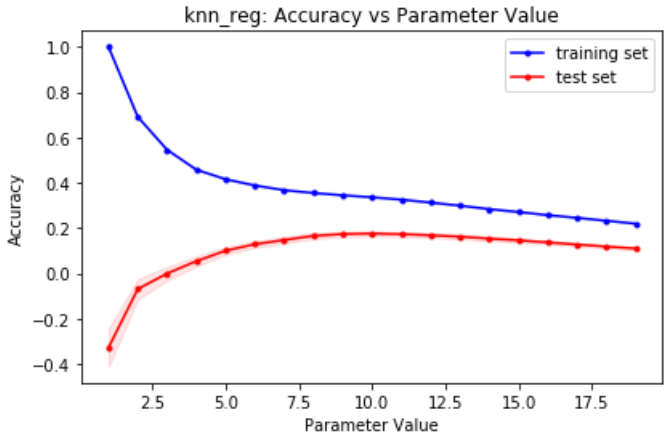

```
Report:
=======
Max average accuracy: 0.2134
Var of accuracy at optimal parameter: 0.0185
Optimal parameter: 0.0010
Total iterations: 100
```

In [48]:
```
models = pd.Series(['kNN Reg', 'Linear Reg', 'Ridge',
                    'Lasso'],
                   name='Machine Learning Method')
accs = pd.Series(accuracies, name='Test Accuracy')
params = pd.Series(['n neighbors', 'C', 'C', 'C'], name='Parameter')
param_val_2 = pd.Series(opt_param, name='Optimal Parameter Value')
encoder = pd.Series([enc]*4, name="Encoder")
scaling = pd.Series(["Min-Max"] * 4, name="Scaling")
features = pd.Series(num_feats, name="Features")
df_2 = pd.concat([encoder, scaling, features, models, accs,
                  params, param_val_2], axis=1)
display(df_2)
```

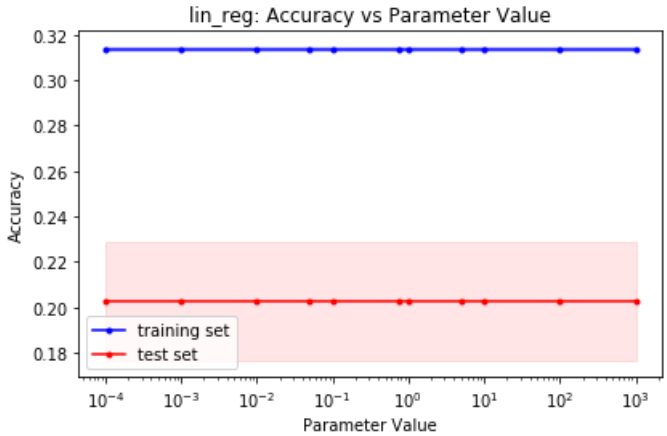
	Encoder	Scaling	Features	Machine Learning Method	Test Accuracy	Parameter	Optimal Parameter Value
0	One-Hot	Min-Max	10	kNN Reg	0.2281	n neighbors	8.0000
1	One-Hot	Min-Max	8	Linear Reg	0.2026	C	0.0001
2	One-Hot	Min-Max	11	Ridge	0.2373	C	0.1000
3	One-Hot	Min-Max	9	Lasso	0.2134	C	0.0010

X Std Scaled

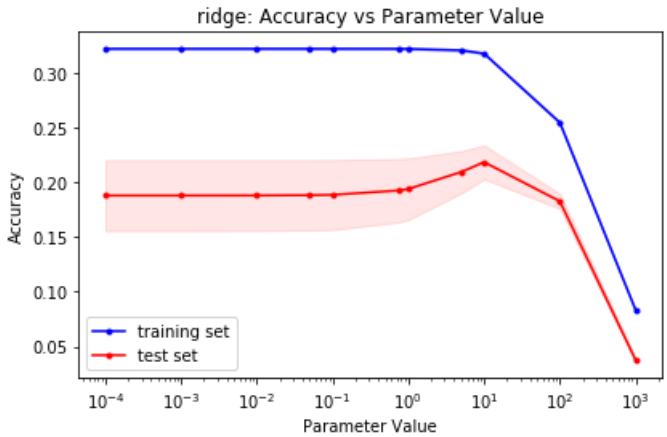
In [49]:
```
X = X_std

params_knn = range(1, 20)
params_lr_svd = [0.0001, 0.001, 0.01, 0.05, 0.1, 0.75, 1, 5, 10, 100, 1000]

accuracies = []
opt_param = []
num_feats = []

types = ['knn_reg', 'lin_reg', 'ridge', 'lasso']

for t in types:
    if t == "knn_reg":
        params = params_knn
        cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
        a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                   seeds=seeds, plot=True)

        accuracies.append(a)
        opt_param.append(p)
        num_feats.append(len(ind))

    else:
        params = params_lr_svd
        cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
        a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                   seeds=seeds, plot=True, max_iter=10000)

        accuracies.append(a)
        opt_param.append(p)
        num_feats.append(len(ind))
```



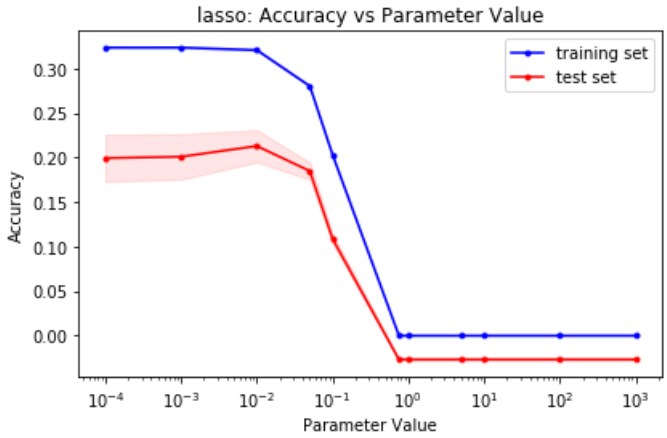

```
Report:
=======
Max average accuracy: 0.1761
Var of accuracy at optimal parameter: 0.0118
Optimal parameter: 10.0000
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.2027
Var of accuracy at optimal parameter: 0.0263
Optimal parameter: 0.0001
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.2185
Var of accuracy at optimal parameter: 0.0155
Optimal parameter: 10.0000
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.2129
Var of accuracy at optimal parameter: 0.0180
Optimal parameter: 0.0100
Total iterations: 100
```

In [50]:
```
models = pd.Series(['kNN Reg', 'Linear Reg', 'Ridge',
                    'Lasso'],
                   name='Machine Learning Method')
accs = pd.Series(accuracies, name='Test Accuracy')
params = pd.Series(['n neighbors', 'C', 'C', 'C'], name='Parameter')
param_val_2 = pd.Series(opt_param, name='Optimal Parameter Value')
encoder = pd.Series([enc]*4, name="Encoder")
scaling = pd.Series(["Std"] * 4, name="Scaling")
features = pd.Series(num_feats, name="Features")
df_3 = pd.concat([encoder, scaling, features, models, accs,
                  params, param_val_2], axis=1)
display(df_3)
```

	Encoder	Scaling	Features	Machine Learning Method	Test Accuracy	Parameter	Optimal Parameter Value
0	One-Hot	Std	10	kNN Reg	0.1761	n neighbors	10.0000
1	One-Hot	Std	9	Linear Reg	0.2027	C	0.0001
2	One-Hot	Std	10	Ridge	0.2185	C	10.0000
3	One-Hot	Std	9	Lasso	0.2129	C	0.0100

Predicting Movie Ratings Summary

The table below summarizes the results of the runs above. Highest regression accuracy (r^2) was achieved using one-hot encoding, min-max scaling, ridge regression, parameter value of 0.100 and 11 features.

In [51]:
```
df_all_1 = pd.concat([df_2, df_3]).reset_index(drop=True)
df_all_1 = df_all_1.sort_values(by="Test Accuracy", ascending=False).reset_index(drop=True)
df_all_1
```

Out[51]:

	Encoder	Scaling	Features	Machine Learning Method	Test Accuracy	Parameter	Optimal Parameter Value
0	One-Hot	Min-Max	11	Ridge	0.2373	C	0.1000
1	One-Hot	Min-Max	10	kNN Reg	0.2281	n neighbors	8.0000
2	One-Hot	Std	10	Ridge	0.2185	C	10.0000
3	One-Hot	Min-Max	9	Lasso	0.2134	C	0.0010
4	One-Hot	Std	9	Lasso	0.2129	C	0.0100
5	One-Hot	Std	9	Linear Reg	0.2027	C	0.0001
6	One-Hot	Min-Max	8	Linear Reg	0.2026	C	0.0001
7	One-Hot	Std	10	kNN Reg	0.1761	n neighbors	10.0000

Predicting Using Best Method

The selected model was rerun below.

In [52]:
```
enc="One-Hot"
```

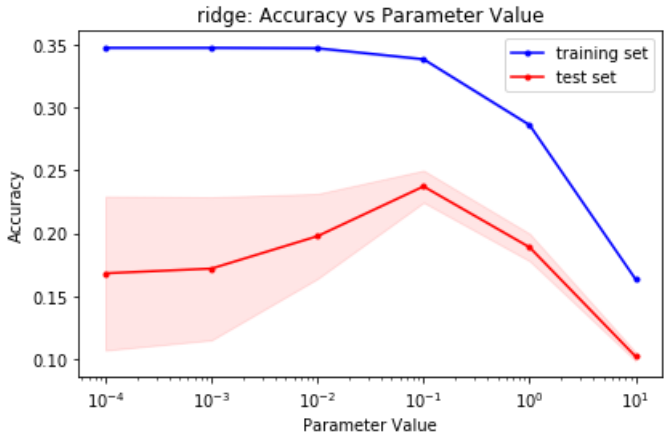
In [53]:
```
# One-hot encoding
df_cat2 = pd.get_dummies(df_cat.astype("str"), drop_first=True)
```

In [54]:
```
X0 = pd.concat([df_num_targ1, df_cat2], axis=1).drop("Ratings", axis=1)
conv_to_numeric(X0, range(0, len(X0.columns)));
```

```
No. of columns that must be changed to numerical: 25
No. of columns automatically changed to numbers, 1st pass: 25

Remaining columns with values that must be changed to numbers:
All values numerical already.
```

In [55]:
```
# min-max scaling
mms = MinMaxScaler()
X_mm = pd.DataFrame(mms.fit_transform(X0), columns=X0.columns)
```

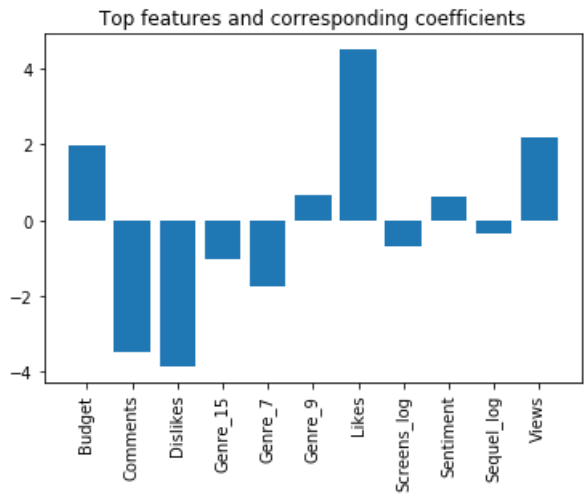
In [56]:
```
X = X_mm

params = [0.0001, 0.001, 0.01, 0.1, 1, 10]

accuracies = []
opt_param = []
num_feats = []

t = 'ridge'
cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                    seeds=seeds, plot=True, max_iter=10000)
```




```
Report:
=======
Max average accuracy: 0.2373
Var of accuracy at optimal parameter: 0.0128
Optimal parameter: 0.1000
Total iterations: 100
```

The chart below shows the 11 features used. The number of likes is the top predictor, correlating positively with ratings. Comments and dislikes are the next two predictors, correlating negatively with ratings.

In [57]:
```
ml = ML_Regressor()
ml.fit(X[cols[ind]], y, ml_type=t, param_range=params, seed_settings=seeds)
plt.bar(cols[ind], ml.coefs_all)
plt.title("Top features and corresponding coefficients")
plt.xticks(rotation=90);
```



Predicting Gross Income

K Nearest Neighbors Regression, Linear Regression, Ridge Regression, and Lasso Regression were the four regression algorithms used to predict Movie `Gross Income`. Naturally, `Ratings` and `Gross Income`, the two variables being predicted, were removed from the features. The results were compared with each other and the algorithm that yields the highest "accuracy" measured in r^2 was identified.

Scaling

Two scaling methods were used: min-max scaling or standard scaling. Min-max scaling scales the values in a column by the range of values in that column (minimum values to maximum values). Standard scaling scales the values in a column by the mean and standard deviation of the values in that column.

In [58]:
```
# min-max scaling
mms = MinMaxScaler()
X_mm = pd.DataFrame(mms.fit_transform(X0), columns=X0.columns)
```

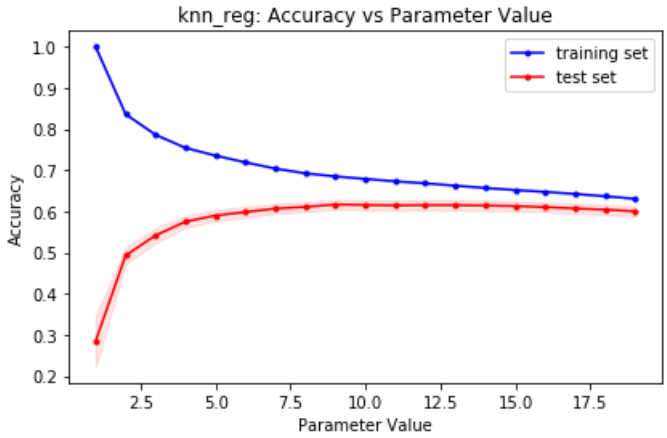
In [59]:
```
# standard scaling
stdsc = StandardScaler()

X_std=pd.DataFrame(stdsc.fit_transform(X0), columns=X0.columns)
```

Ratings values stored in `y2` was copied to `y` for modeling purposes.

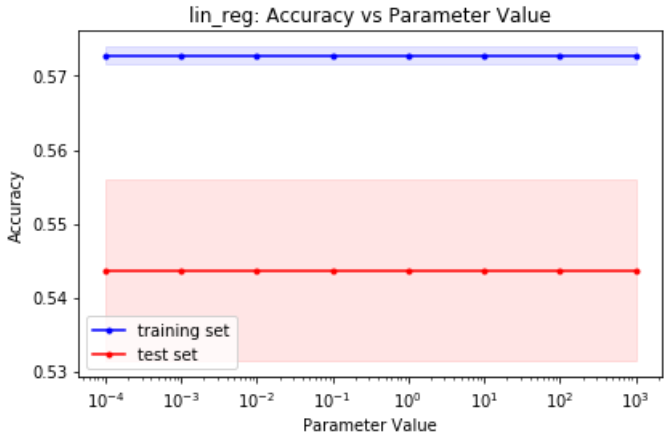
In [60]:
```
y = (y2)
```

X Min-Max Scaled

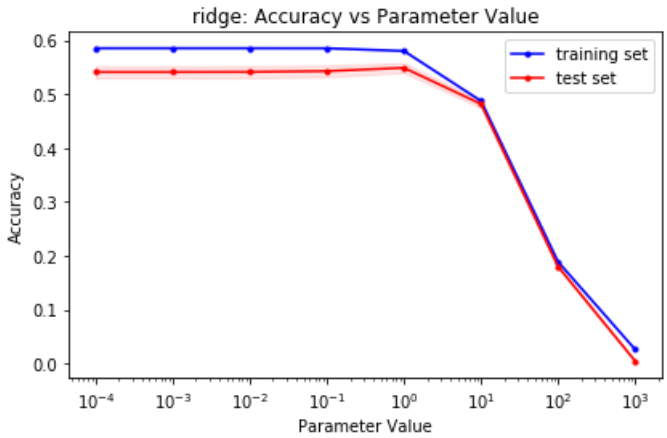
In [61]:
```
X = X_mm

params_knn = range(1, 20)
params_lr_svd = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

accuracies = []
opt_param = []
num_feats = []

types = ['knn_reg', 'lin_reg', 'ridge', 'lasso']

for t in types:
    if t == "knn_reg":
        params = params_knn
        cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
        a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                   seeds=seeds, plot=True)

        accuracies.append(a)
        opt_param.append(p)
        num_feats.append(len(ind))

    else:
        params = params_lr_svd
        cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
        a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                   seeds=seeds, plot=True, max_iter=10000)

        accuracies.append(a)
        opt_param.append(p)
        num_feats.append(len(ind))
```



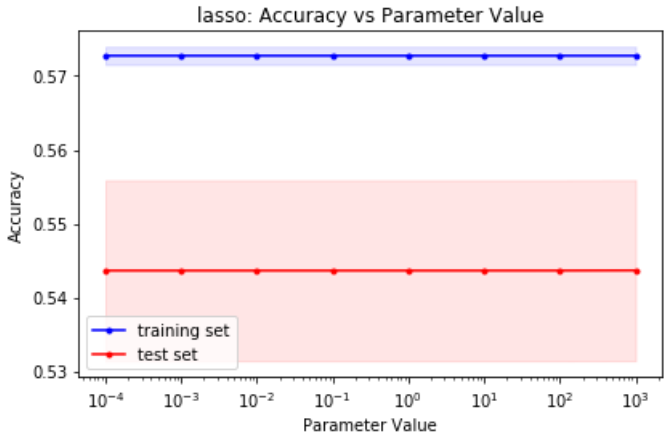

```
Report:
=======
Max average accuracy: 0.6171
Var of accuracy at optimal parameter: 0.0117
Optimal parameter: 9.0000
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.5436
Var of accuracy at optimal parameter: 0.0122
Optimal parameter: 0.0001
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.5488
Var of accuracy at optimal parameter: 0.0092
Optimal parameter: 1.0000
Total iterations: 100
```



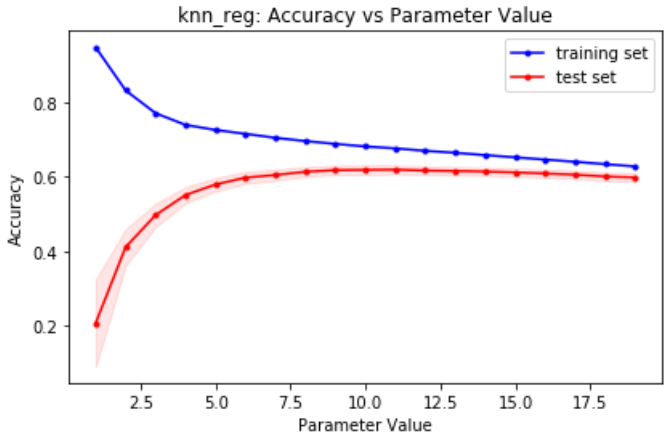

```
Report:
=======
Max average accuracy: 0.5437
Var of accuracy at optimal parameter: 0.0122
Optimal parameter: 1000.0000
Total iterations: 100
```

In [62]:
```
models = pd.Series(['kNN Reg', 'Linear Reg', 'Ridge',
                    'Lasso'],
                   name='Machine Learning Method')
accs = pd.Series(accuracies, name='Test Accuracy')
params = pd.Series(['n neighbors', 'C', 'C', 'C'], name='Parameter')
param_val_2 = pd.Series(opt_param, name='Optimal Parameter Value')
encoder = pd.Series([enc]*4, name="Encoder")
scaling = pd.Series(["Min-Max"] * 4, name="Scaling")
features = pd.Series(num_feats, name="Features")
df_22 = pd.concat([encoder, scaling, features, models, accs,
                   params, param_val_2], axis=1)
display(df_22)
```

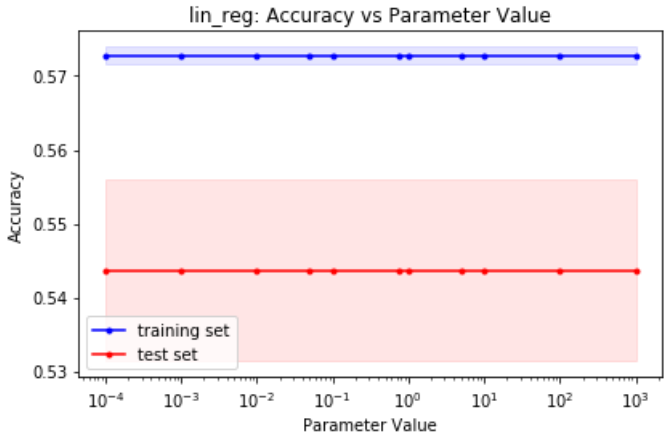
	Encoder	Scaling	Features	Machine Learning Method	Test Accuracy	Parameter	Optimal Parameter Value
0	One-Hot	Min-Max	8	kNN Reg	0.6171	n neighbors	9.0000
1	One-Hot	Min-Max	5	Linear Reg	0.5436	C	0.0001
2	One-Hot	Min-Max	7	Ridge	0.5488	C	1.0000
3	One-Hot	Min-Max	5	Lasso	0.5437	C	1000.0000

X Std Scaled

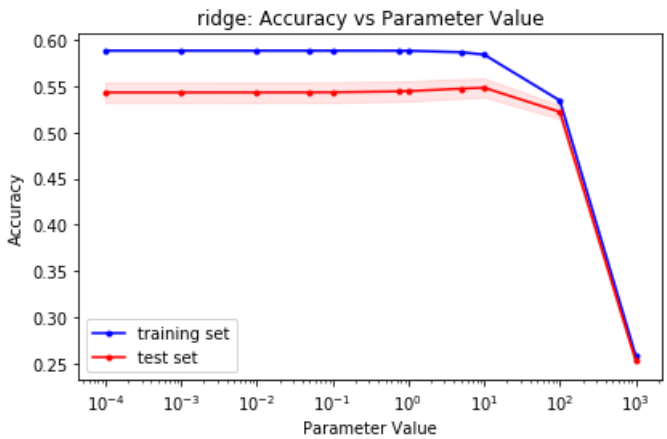
In [63]:
```
X = X_std

params_knn = range(1, 20)
params_lr_svd = [0.0001, 0.001, 0.01, 0.05, 0.1, 0.75, 1, 5, 10, 100, 1000]

accuracies = []
opt_param = []
num_feats = []

types = ['knn_reg', 'lin_reg', 'ridge', 'lasso']

for t in types:
    if t == "knn_reg":
        params = params_knn
        cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
        a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                   seeds=seeds, plot=True)

        accuracies.append(a)
        opt_param.append(p)
        num_feats.append(len(ind))

    else:
        params = params_lr_svd
        cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
        a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                   seeds=seeds, plot=True, max_iter=10000)

        accuracies.append(a)
        opt_param.append(p)
        num_feats.append(len(ind))
```



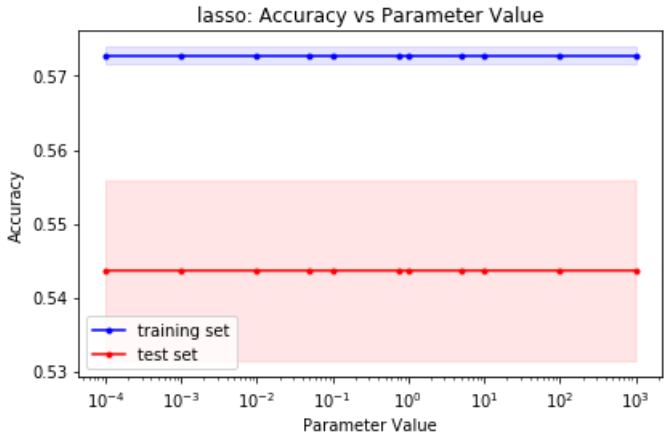

```
Report:
=======
Max average accuracy: 0.6196
Var of accuracy at optimal parameter: 0.0118
Optimal parameter: 11.0000
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.5436
Var of accuracy at optimal parameter: 0.0122
Optimal parameter: 0.0001
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.5482
Var of accuracy at optimal parameter: 0.0101
Optimal parameter: 10.0000
Total iterations: 100
```




```
Report:
=======
Max average accuracy: 0.5436
Var of accuracy at optimal parameter: 0.0122
Optimal parameter: 1000.0000
Total iterations: 100
```

In [64]:
```
models = pd.Series(['kNN Reg', 'Linear Reg', 'Ridge',
                    'Lasso'],
                   name='Machine Learning Method')
accs = pd.Series(accuracies, name='Test Accuracy')
params = pd.Series(['n neighbors', 'C', 'C', 'C'], name='Parameter')
param_val_2 = pd.Series(opt_param, name='Optimal Parameter Value')
encoder = pd.Series([enc]*4, name="Encoder")
scaling = pd.Series(["Std"] * 4, name="Scaling")
features = pd.Series(num_feats, name="Features")
df_23 = pd.concat([encoder, scaling, features, models, accs,
                   params, param_val_2], axis=1)
display(df_23)
```

	Encoder	Scaling	Features	Machine Learning Method	Test Accuracy	Parameter	Optimal Parameter Value
0	One-Hot	Std	5	kNN Reg	0.6196	n neighbors	11.0000
1	One-Hot	Std	5	Linear Reg	0.5436	C	0.0001
2	One-Hot	Std	6	Ridge	0.5482	C	10.0000
3	One-Hot	Std	5	Lasso	0.5436	C	1000.0000

Predicting Gross Income Summary

The table below summarizes the results of the runs above. Highest regression accuracy (r^2) was achieved using one-hot encoding, min-max scaling, ridge regression, parameter value of 0.100 and 12 features.

In [65]:
```
df_all_2 = pd.concat([df_22, df_23]).reset_index(drop=True)
df_all_2 = df_all_2.sort_values(by="Test Accuracy", ascending=False).reset_index(drop=True)
df_all_2
```

Out[65]:

	Encoder	Scaling	Features	Machine Learning Method	Test Accuracy	Parameter	Optimal Parameter Value
0	One-Hot	Std	5	kNN Reg	0.6196	n neighbors	11.0000
1	One-Hot	Min-Max	8	kNN Reg	0.6171	n neighbors	9.0000
2	One-Hot	Min-Max	7	Ridge	0.5488	C	1.0000
3	One-Hot	Std	6	Ridge	0.5482	C	10.0000
4	One-Hot	Min-Max	5	Lasso	0.5437	C	1000.0000
5	One-Hot	Min-Max	5	Linear Reg	0.5436	C	0.0001
6	One-Hot	Std	5	Linear Reg	0.5436	C	0.0001
7	One-Hot	Std	5	Lasso	0.5436	C	1000.0000

Predicting Using Best Method

The selected model was rerun below.

In [66]:
```
enc="One-Hot"
```

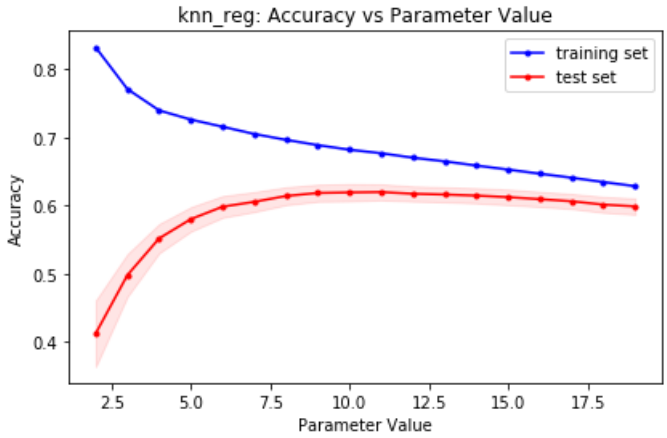
In [67]:
```
# One-hot encoding
df_cat2 = pd.get_dummies(df_cat.astype("str"), drop_first=True)
```

In [68]:
```
X0 = pd.concat([df_num_targ1, df_cat2], axis=1).drop("Ratings", axis=1)
conv_to_numeric(X0, range(0, len(X0.columns)));
```

```
No. of columns that must be changed to numerical: 25
No. of columns automatically changed to numbers, 1st pass: 25

Remaining columns with values that must be changed to numbers:
All values numerical already.
```

In [69]:
```
# min-max scaling
X_std = pd.DataFrame(stdsc.fit_transform(X0), columns=X0.columns)
```

In [70]:
```
X = X_std

params = range(2, 20)

accuracies = []
opt_param = []
num_feats = []

t = 'knn_reg'
cols = ml_each_feature(X, y, t=t, params=params, seeds=seeds)
a, p, ind = optimize_feats(X, y, cols, t=t, params=params,
                                    seeds=seeds, plot=True)
```




```
Report:
=======
Max average accuracy: 0.6196
Var of accuracy at optimal parameter: 0.0118
Optimal parameter: 11.0000
Total iterations: 100
```

The four features used are `Screens_log`, `Screens`, `Genre_12`, `Genre_7`, and`Genre_6`.

In [71]:
```
cols[ind]
```

Out[71]:
```
array(['Screens_log', 'Screens', 'Genre_12', 'Genre_7', 'Genre_6'],
      dtype=object)
```

Summary

In [72]:
```
df_best = pd.concat([df_all_1.iloc[[0]], df_all_2.iloc[[0]]])
df_best.head()
```

Out[72]:

	Encoder	Scaling	Features	Machine Learning Method	Test Accuracy	Parameter	Optimal Parameter Value
0	One-Hot	Min-Max	11	Ridge	0.2373	C	0.1
0	One-Hot	Std	5	kNN Reg	0.6196	n neighbors	11.0

Various combinations of categorical data encoding, features scaling, and regression algorithms were run on a movies dataset prepared by Mehreen Ahmed of the National University of Sciences and Technology (NUST), Pakistan. The goal was to predict the gross income and rating based on features sourced from IMDB, Youtube, and Twitter.

23.73% was the highest coefficient of determination r^2 achieved when predicting movie ratings. The regression algorithm used is Ridge Regression with parameter $\alpha = 0.1$. One-hot encoding was applied on the genre feature, and min-max scaling was applied on each of the feature. A total of 11 features were used namely:

1. Likes
2. Dislikes
3. Comments
4. Views
5. Budget
6. Genre_7
7. Genre_15
8. Genre_9
9. Screens_log
10. Sentiment
11. Sequel_log

The number of likes on the movie trailer on Youtube is the feature with the largest positive coefficent. The number of comments on Youtube trailers has the largest negative coefficient, followed closely by the number of dislikes.

When predicting gross income, the highest r^2 of 61.96% was achieved using kNN Regression using 11 neighbors. One-hot encoding was applied on the genre feature, and standard scaling was applied on each of the feature. A total of five features were used namely:

1. Screens_log
2. Screens
3. Genre_12
4. Genre_7
5. Genre_6

Limitations and Avenues for Further Research

The dataset used only spanned from 2014 to 2015, and only used Twitter and Youtube data. Further studies could explore to expand to capture more years as well as more video and social media sites. Feature engineering may also improve accuracy in the future.

References and Acknowledgements

I would like to acknowledge Prof. Monterola, Prof. Legara, and the rest of AIM Access lab Professors and lab team.

Data was sourced from
https://archive.ics.uci.edu/ml/datasets/CSM+%28Conventional+and+Social+Media+Movies%29+Dataset+2014+and+2015
(https://archive.ics.uci.edu/ml/datasets/CSM+%28Conventional+and+Social+Media+Movies%29+Dataset+2014+and+2015)

Ahmed M, Jahangir M, Afzal H, Majeed A, Siddiqi I. Using Crowd-source based features from social media and Conventional features to predict the movies popularity. InSmart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference on 2015 Dec 19 (pp. 273-278). IEEE.