IMDb Datasets Analysis to Predict the Average Rating

Ziting Tang   08/25/2019

Data Cleaning

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import pyspark
from pyspark.sql import SparkSession
from sklearn.metrics import r2_score,mean_squared_error,mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import PolynomialFeatures
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
```

There are seven datasets on [https://www.imdb.com/interfaces/ (https://www.imdb.com/interfaces/)](https://www.imdb.com/interfaces/). Here I used three of them to analysize the average rating. The variable avreageRating is only included dataset TitleRatings.tsv. The other two data sets are joined with dataset TitleRatings.tsv according to the variable tconst, which is the alphanumeric unique identifier of the title. Since the size of all data sets are very large. The PySpark package is used to read the datasets.

In [2]:
```
spark = SparkSession.builder.getOrCreate()
dfTitleRatings = spark.read.options(header='true', inferschema='true', delimit
er='\t').csv("C:/Users/Ziting Tang/Desktop/Audible/TitleRatings.tsv")
dfTitleBasics= spark.read.options(header='true', inferschema='true', delimiter
='\t').csv("C:/Users/Ziting Tang/Desktop/Audible/TitleBasics.tsv")
dfTitleCrew = spark.read.options(header='true', inferschema='true', delimiter=
'\t').csv("C:/Users/Ziting Tang/Desktop/Audible/TitleCrew.tsv")
```

From below, we can see all the variables from the combined dataset df: tconst, averageRating, numVotes, titleType, primaryTitle, originalTitle, isAdult, startYear, endYear, runtimeMinutes, genres, directors and writers. A '\N' is used to denote that a particular field is missing or null for that title/name. In Python, '\N' is represented by '\\N'.

In [3]:
```
df = dfTitleRatings.join(dfTitleBasics, on=['tconst'], how='left_outer').join(
dfTitleCrew, on=['tconst'], how='left_outer')
df.show(5)
```

```
+---------+-------------+--------+---------+--------------------+------------
--------+-------+---------+-------+--------------+---------------+---------+-
------------------+
|   tconst|averageRating|numVotes|titleType|        primaryTitle|       origi
nalTitle|isAdult|startYear|endYear|runtimeMinutes|         genres|directors|
writers|
+---------+-------------+--------+---------+--------------------+------------
--------+-------+---------+-------+--------------+---------------+---------+-
------------------+
|tt0000658|          6.4|     148|    short|The Puppet's Nigh...|Le cauchemar
de F...|      0|     1908|     \N|             2|Animation,Short|nm0169871|
\N|
|tt0001732|          7.1|       8|    short|The Lighthouse Ke...|The Lighthou
se Ke...|      0|     1911|     \N|            \N|    Drama,Short|nm0408436|
\N|
|tt0002253|          4.2|       5|    short|           Home Folks|           Ho
me Folks|      0|     1912|     \N|            17|    Drama,Short|nm0000428|
nm0940488|
|tt0002473|          6.8|      57|    short|     The Sands of Dee|     The Sand
s of Dee|      0|     1912|     \N|            17|  Romance,Short|nm0000428|
nm0455504|
|tt0002588|          6.8|      10|    short|Zigomar contre Ni...|Zigomar cont
re Ni...|      0|     1912|     \N|            18|          Short|nm0419327|n
m0419327,nm0768577|
+---------+-------------+--------+---------+--------------------+------------
--------+-------+---------+-------+--------------+---------------+---------+-
------------------+
only showing top 5 rows
```

In [4]:
```
df.head()
```

Out[4]:
```
Row(tconst='tt0000658', averageRating=6.4, numVotes=148, titleType='short', p
rimaryTitle="The Puppet's Nightmare", originalTitle='Le cauchemar de Fantoch
e', isAdult=0, startYear='1908', endYear='\\N', runtimeMinutes='2', genres='A
nimation,Short', directors='nm0169871', writers='\\N')
```

In [5]:

```
data=df.toPandas()   # convert df in SparkSession to Pandas data frame
print(data.head())
data.info()
```

```
      tconst  averageRating  numVotes titleType               primaryTitle  \
0  tt0000658            6.4       148     short      The Puppet's Nightmare
1  tt0001732            7.1         8     short       The Lighthouse Keeper
2  tt0002253            4.2         5     short                  Home Folks
3  tt0002473            6.8        57     short             The Sands of Dee
4  tt0002588            6.8        10     short  Zigomar contre Nick Carter

                originalTitle  isAdult startYear endYear runtimeMinutes  \
0    Le cauchemar de Fantoche        0      1908      \N              2
1       The Lighthouse Keeper        0      1911      \N             \N
2                  Home Folks        0      1912      \N             17
3            The Sands of Dee        0      1912      \N             17
4  Zigomar contre Nick Carter        0      1912      \N             18

            genres  directors              writers
0  Animation,Short  nm0169871                   \N
1      Drama,Short  nm0408436                   \N
2      Drama,Short  nm0000428            nm0940488
3    Romance,Short  nm0000428            nm0455504
4            Short  nm0419327  nm0419327,nm0768577
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 964364 entries, 0 to 964363
Data columns (total 13 columns):
tconst            964364 non-null object
averageRating     964364 non-null float64
numVotes          964364 non-null int32
titleType         964364 non-null object
primaryTitle      964364 non-null object
originalTitle     964364 non-null object
isAdult           964364 non-null int32
startYear         964364 non-null object
endYear           964364 non-null object
runtimeMinutes    964364 non-null object
genres            964364 non-null object
directors         964364 non-null object
writers           964364 non-null object
dtypes: float64(1), int32(2), object(10)
memory usage: 88.3+ MB
```

In [6]:
```
data.describe()
```

Out[6]:

	averageRating	numVotes	isAdult
count	964364.000000	9.643640e+05	964364.000000
mean	6.890414	9.633980e+02	0.018691
std	1.401085	1.564202e+04	0.135432
min	1.000000	5.000000e+00	0.000000
25%	6.100000	9.000000e+00	0.000000
50%	7.100000	2.000000e+01	0.000000
75%	7.900000	7.700000e+01	0.000000
max	10.000000	2.125804e+06	1.000000

We can see that for three numeric variables averageRating, numVotes and isAdult, there are no missing values. They're all positive. There are no weird values.

In [7]:
```
for column in data.columns:
    uniques = sorted(data[column].unique())
    print('{0:20s} {1:5d}\t'.format(column, len(uniques)), uniques[:5])
```

```
tconst                 964364      ['tt0000001', 'tt0000002', 'tt0000003', 'tt0
000004', 'tt0000005']
averageRating             91       [1.0, 1.1, 1.2, 1.3, 1.4]
numVotes               17210       [5, 6, 7, 8, 9]
titleType                 10       ['movie', 'short', 'tvEpisode', 'tvMiniSerie
s', 'tvMovie']
primaryTitle           742455      ['!Next?', '!Women Art Revolution', '"#selfi
e" by The Chainsmokers', '"1 jikan buchi nuki de Sasuke ga ô abare dattebayo
supesharu": Arashi o yobu otoko!! Sasuke no gejimayu-ryû taijutsu!', '"1 jika
n buchi nuki de Sasuke ga ô abare dattebayo supesharu": Date ni okureta wake
janai! Kyûkyoku ôgi - Chidori tanjô!!']
originalTitle          752860      ['!Next?', '"#selfie" by The Chainsmokers',
'"1 jikan buchi nuki de Sasuke ga ô abare dattebayo supesharu": Arashi o yobu
otoko!! Sasuke no gejimayu-ryû taijutsu!', '"1 jikan buchi nuki de Sasuke ga
ô abare dattebayo supesharu": Date ni okureta wake janai! Kyûkyoku ôgi - Chid
ori tanjô!!', '"1" More']
isAdult                    2       [0, 1]
startYear                139       ['1874', '1878', '1881', '1883', '1885']
endYear                   76       ['1927', '1933', '1939', '1949', '1950']
runtimeMinutes           658       ['0', '1', '10', '100', '1000']
genres                  1945       ['Action', 'Action,Adult', 'Action,Adult,Adv
enture', 'Action,Adult,Animation', 'Action,Adult,Comedy']
directors              236346      ['\\N', 'nm0000005', 'nm0000008', 'nm000000
9,nm0169065', 'nm0000010']
writers                410926      ['\\N', 'nm0000005', 'nm0000005,nm0110119',
'nm0000005,nm0279027', 'nm0000005,nm0340471']
```

From above, we can see the number of unique values for each variable and the first five unique values. There are no missing values for four variables tconst, averageRating, numVotes, isAdult. Since the unique values for variables tconst, primaryTitle and originalTitle are too large, these three variables won't be used in the model.

In [8]:
```
print(data[data["titleType"]=='\\N'].shape[0]) #0
print(data[data["primaryTitle"]=='\\N'].shape[0]) #0
print(data[data["originalTitle"]=='\\N'].shape[0]) #3
print(data[data["startYear"]=='\\N'].shape[0]) # 109
print(data[data["endYear"]=='\\N'].shape[0]) # 943583
print(data[data["runtimeMinutes"]=='\\N'].shape[0]) # 262733
print(data[data["genres"]=='\\N'].shape[0]) # 22978
```

```
0
0
3
109
943583
262733
22978
```

There is no missing value for variable titleType. The number of missing values for variable endYear is 943583. Almost all are missing values for endYear, so drop endYear.

In [9]:
```
print(data["directors"].value_counts(dropna=False).head(10))
print(data["writers"].value_counts(dropna=False).head(10))
```

```
\N                        136689
nm1337210                   2848
nm3766090                   1085
nm0123273                    910
nm2078274                    645
nm3005544                    635
nm0842611                    594
nm1121649                    563
nm0053484,nm0360253          561
nm0669120                    555
Name: directors, dtype: int64
\N                                  211281
nm3005544                              865
nm3766090                              862
nm1108327                              861
nm0868066                              752
nm1444457                              670
nm0663789                              599
nm2095966,nm2242949,nm0846969          595
nm1121649                              586
nm0242472                              554
Name: writers, dtype: int64
```

There's not enough information for missing directors and writers. It is not appropriate to impute missing values with the mode. For valid directors and writers, the number for each category are less than 3000. The largest numbers are 2848 and 865 for directors and writers, respectively. But for missing values, the numbers are 136689 and 211281 for directors and writers, respectively. There are too many missing values. It is not appropriate to treat them as separate category and use them as a different level. For simplicity, rows/observations of the data with missing directors and writers will be dropped.

Currently, the varaibles averageRating, numVotes, titleType, isAdult, startYear, runtimeMinutes, genres, directors and writers are kept in the data set.

In [10]:
```
movie=data[[ 'averageRating','numVotes', 'titleType' , 'isAdult','startYear' ,
'runtimeMinutes' ,'genres' ,'directors' ,'writers']]
movie=movie[movie["directors"]!='\\N']
movie=movie[movie["writers"]!='\\N']

movie.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 708102 entries, 2 to 964363
Data columns (total 9 columns):
averageRating     708102 non-null float64
numVotes          708102 non-null int32
titleType         708102 non-null object
isAdult           708102 non-null int32
startYear         708102 non-null object
runtimeMinutes    708102 non-null object
genres            708102 non-null object
directors         708102 non-null object
writers           708102 non-null object
dtypes: float64(1), int32(2), object(6)
memory usage: 48.6+ MB
```

In [11]:
```
movie.loc[movie['runtimeMinutes']=='\\N','runtimeMinutes']="NaN"
movie['runtimeMinutes']=movie['runtimeMinutes'].astype(str).astype(float) #con
vert to an integer.
#movie['runtimeMinutes']=pd.to_numeric(movie['runtimeMinutes']) # same method
```

Change the type of variable runtimeMinutes from object to float only after the missing value symbol '\N' is replaced by np.nan, i.e., NaN.

In [12]:

```
movie.info() # check the type
print(movie.describe())
print(movie[movie['runtimeMinutes']>2000])
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 708102 entries, 2 to 964363
Data columns (total 9 columns):
averageRating     708102 non-null float64
numVotes          708102 non-null int32
titleType         708102 non-null object
isAdult           708102 non-null int32
startYear         708102 non-null object
runtimeMinutes    557240 non-null float64
genres            708102 non-null object
directors         708102 non-null object
writers           708102 non-null object
dtypes: float64(2), int32(2), object(5)
memory usage: 48.6+ MB
       averageRating      numVotes        isAdult  runtimeMinutes
count  708102.000000  7.081020e+05  708102.000000   557240.000000
mean        6.876295  1.292878e+03       0.009586       58.656832
std         1.376301  1.824053e+04       0.097439       46.626703
min         1.000000  5.000000e+00       0.000000        0.000000
25%         6.100000  1.000000e+01       0.000000       24.000000
50%         7.100000  2.700000e+01       0.000000       50.000000
75%         7.800000  1.170000e+02       0.000000       90.000000
max        10.000000  2.125804e+06       1.000000     5700.000000
        averageRating  numVotes titleType  isAdult startYear  runtimeMinutes  \
59586             6.2       247  tvSeries        0      2006          3900.0
64474             7.2       126  tvSeries        0      2006          2150.0
68375             7.3        12  tvSeries        0      1972          2925.0
210126            4.5        28     video        0      2006          5700.0
394121            5.4         9  tvSeries        0      2000          2288.0
408754            5.7        44  tvSeries        0      1991          3000.0
454690            6.3       300  tvSeries        0      1997          3600.0
507072            5.1       340     movie        0      1987          5220.0
646200            8.7        67  tvSeries        0      2017          3825.0

                       genres  \
59586                   Drama
64474                   Drama
68375   Action,Adventure,Drama
210126                  Drama
394121              Talk-Show
408754                 Comedy
454690                  Drama
507072      Documentary,Music
646200          Drama,History

                                                directors  \
59586   nm0227486,nm0192463,nm0324426,nm0942185,nm0248119
64474                                           nm1646055
68375                                           nm0613668
210126                                          nm4246748
394121                                          nm4468748
408754                      nm6558891,nm6558892,nm6551382
454690                                          nm0220720
507072                                          nm1267224
646200                                          nm6785971
```

```
                                                  writers
59586    nm1548597,nm1919867,nm1107925,nm2352020,nm2358...
64474                                 nm6971503,nm2265321
68375                                           nm1713458
210126                                          nm4246748
394121                                          nm0755704
408754                                nm6551383,nm2590336
454690                                nm0222956,nm1818285
507072                                          nm1007137
646200                                nm2346525,nm3188723
```

Check if the observations with too large runtimeMinutes are outliers: The title of most movies with run time larger than 2000 min is tvSeries and video. This makes sense. These observations will not be considered as outliers. Impute the missing values of variable runtimeMinutes with median.

In [13]:

```
median = movie['runtimeMinutes'].median()
movie['runtimeMinutes'].fillna(median, inplace = True)
```

In [14]:

```
print(movie.columns.values)
print(movie.columns)
labels=movie["averageRating"]# get the response varible
numeric_features=movie.loc[:, movie.columns != "averageRating"]._get_numeric_d
ata().columns.values.tolist()# get numeric features
print(numeric_features)
movie.describe()
```

```
['averageRating' 'numVotes' 'titleType' 'isAdult' 'startYear'
 'runtimeMinutes' 'genres' 'directors' 'writers']
Index(['averageRating', 'numVotes', 'titleType', 'isAdult', 'startYear',
       'runtimeMinutes', 'genres', 'directors', 'writers'],
      dtype='object')
['numVotes', 'isAdult', 'runtimeMinutes']
```

Out[14]:

	averageRating	numVotes	isAdult	runtimeMinutes
count	708102.000000	7.081020e+05	708102.000000	708102.000000
mean	6.876295	1.292878e+03	0.009586	56.812483
std	1.376301	1.824053e+04	0.097439	41.514208
min	1.000000	5.000000e+00	0.000000	0.000000
25%	6.100000	1.000000e+01	0.000000	30.000000
50%	7.100000	2.700000e+01	0.000000	50.000000
75%	7.800000	1.170000e+02	0.000000	84.000000
max	10.000000	2.125804e+06	1.000000	5700.000000

In [15]:
```python
categorical_features=['titleType', 'startYear','genres', 'directors', 'writer
s']
numeric_features.remove("isAdult")
categorical_features.append("isAdult") #get categorical features
print(numeric_features)
print(categorical_features)
```

```
['numVotes', 'runtimeMinutes']
['titleType', 'startYear', 'genres', 'directors', 'writers', 'isAdult']
```

In [16]:
```python
movie['titleType'].value_counts()
movie['directors'].value_counts()
movie["isAdult"].value_counts()
```

Out[16]:
```
0    701314
1      6788
Name: isAdult, dtype: int64
```

In [17]:
```python
movie[categorical_features]=movie[categorical_features].astype(str)
print(numeric_features)
print(categorical_features)
movie["isAdult"]=movie["isAdult"].astype(int)
```

```
['numVotes', 'runtimeMinutes']
['titleType', 'startYear', 'genres', 'directors', 'writers', 'isAdult']
```

In [18]:
```python
movie.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 708102 entries, 2 to 964363
Data columns (total 9 columns):
averageRating     708102 non-null float64
numVotes          708102 non-null int32
titleType         708102 non-null object
isAdult           708102 non-null int32
startYear         708102 non-null object
runtimeMinutes    708102 non-null float64
genres            708102 non-null object
directors         708102 non-null object
writers           708102 non-null object
dtypes: float64(2), int32(2), object(5)
memory usage: 48.6+ MB
```

In [19]:
```
print(movie["startYear"].value_counts())
print(movie["genres"].value_counts())
```

```
2016    31634
2017    30558
2015    30418
2014    28923
2013    28621
        ...
1904        8
1899        8
1902        7
1895        6
1894        3
Name: startYear, Length: 127, dtype: int64
Comedy                                76241
Drama                                 65396
Documentary                           23820
Comedy,Drama                          17477
Crime,Drama,Mystery                   17468
                                      ...
Biography,Documentary,Talk-Show           1
Biography,History,Musical                 1
Adventure,Game-Show,Mystery               1
Music,Romance,Sci-Fi                      1
Adventure,Crime,Sport                     1
Name: genres, Length: 1803, dtype: int64
```

The number of missing values for variables startYear and genresis small. The most frequent method is used to impute the missing values of category variables. From above result, we can see that the most frequent categories for variables startYear and genresis are '2016' and 'Comedy'.

In [20]:
```
movie.loc[movie["startYear"]=='\\N',"startYear"]=2016
movie.loc[movie["genres"]=='\\N',"genres"]="Comedy"
movie["startYear"]=movie["startYear"].astype(int)
```

In [21]:
```
# This is the same most frequent impution method as above, but using SimpleImp
uter function. This will take longer time to operate.
#imp=SimpleImputer(missing_values='\\N',strategy="most_frequent") #Number of m
issing values is small. The most frequent method is used to impute the missing
values of category variables.
#movie[['startYear', 'genres']]=imp.fit_transform(movie[['startYear', 'genre
s']])
```

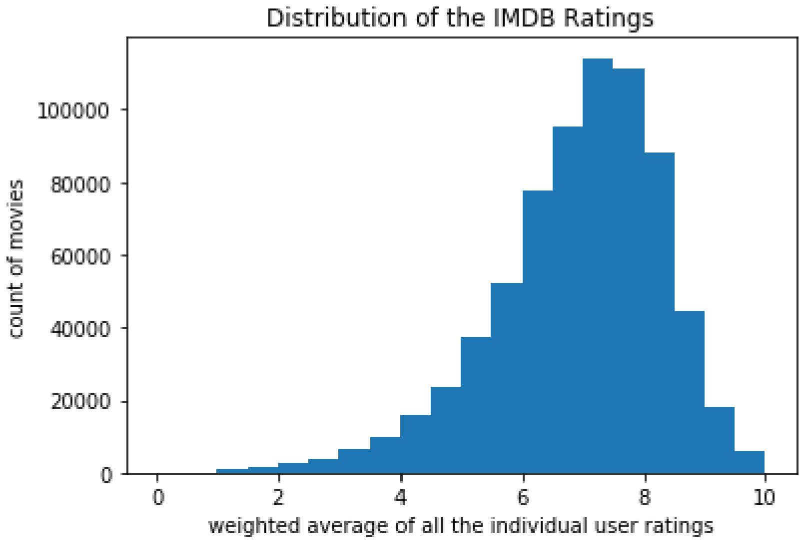
In [22]:
```
movie.info()
print(movie.describe())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 708102 entries, 2 to 964363
Data columns (total 9 columns):
averageRating     708102 non-null float64
numVotes          708102 non-null int32
titleType         708102 non-null object
isAdult           708102 non-null int32
startYear         708102 non-null int32
runtimeMinutes    708102 non-null float64
genres            708102 non-null object
directors         708102 non-null object
writers           708102 non-null object
dtypes: float64(2), int32(3), object(4)
memory usage: 45.9+ MB
       averageRating      numVotes        isAdult      startYear  \
count  708102.000000  7.081020e+05  708102.000000  708102.000000
mean        6.876295  1.292878e+03       0.009586    1997.396022
std         1.376301  1.824053e+04       0.097439      21.104196
min         1.000000  5.000000e+00       0.000000    1894.000000
25%         6.100000  1.000000e+01       0.000000    1988.000000
50%         7.100000  2.700000e+01       0.000000    2005.000000
75%         7.800000  1.170000e+02       0.000000    2013.000000
max        10.000000  2.125804e+06       1.000000    2019.000000

       runtimeMinutes
count   708102.000000
mean        56.812483
std         41.514208
min          0.000000
25%         30.000000
50%         50.000000
75%         84.000000
max       5700.000000
```

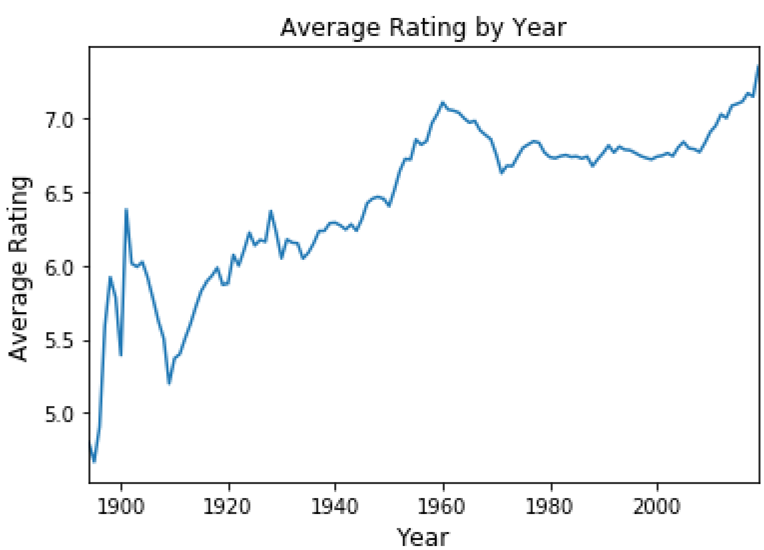
Data Visualization

In [23]:
```
plt.hist(labels,bins=20,  range=[0, 10])
plt.title("Distribution of the IMDB Ratings")
plt.xlabel('weighted average of all the individual user ratings')
plt.ylabel('count of movies')
plt.show()
```



From above histogram, we can see that most of the average ratings fall betweeen 6 and 9. Most of people are satisfied with the movies they saw. The distribution of the average ratings is a little left skewed. It is kind of bell shaped, but not symmetric. The averageRating variable doesn't follow the normal distribution exactly.

In [24]:
```
movie.groupby('startYear')['averageRating'].mean().plot()
plt.title('Average Rating by Year')
plt.ylabel('Average Rating', fontsize=12)
plt.xlabel('Year', fontsize=12)
plt.show()
```

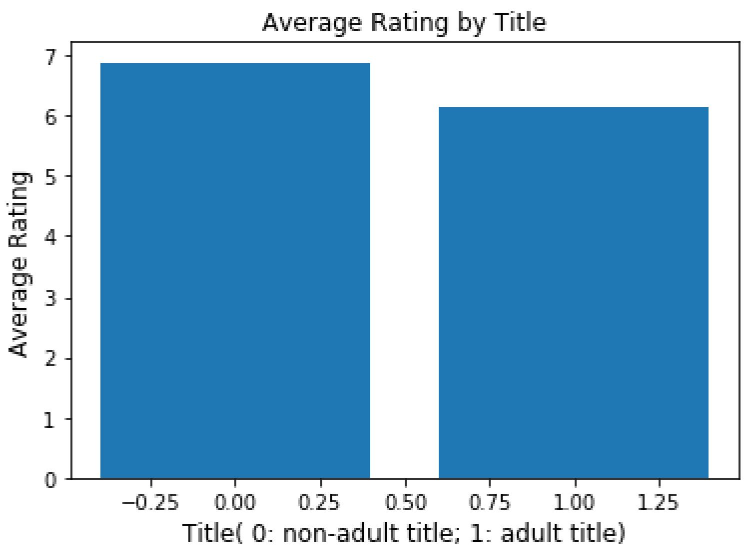


From above, we can see that the group mean of the average rating is kind of increasing with the year. The average rating is significantly time related. This will be further analysed if there is enough time.

In [25]:
```
movie.groupby('isAdult')['averageRating'].mean()
```

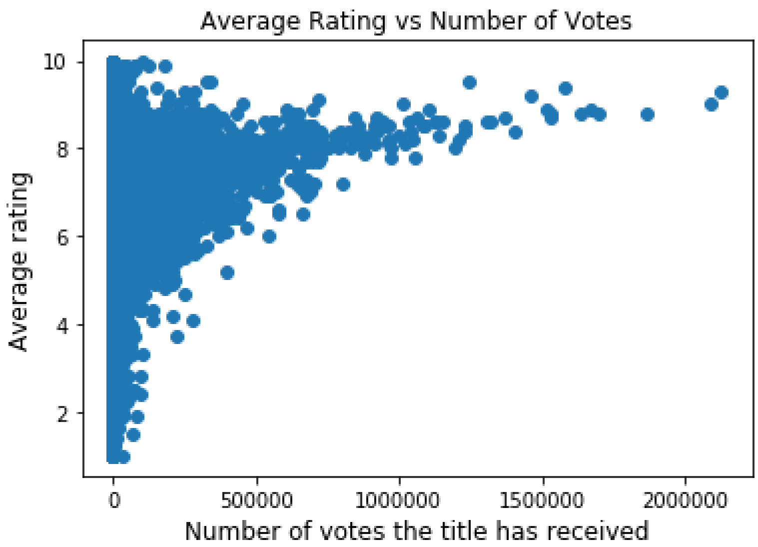
Out[25]:
```
isAdult
0    6.883403
1    6.142001
Name: averageRating, dtype: float64
```

In [26]:
```
# Visualization of adult
plt.bar([0,1],movie.groupby('isAdult')['averageRating'].mean())
plt.title('Average Rating by Title')
plt.ylabel('Average Rating', fontsize=12)
plt.xlabel('Title( 0: non-adult title; 1: adult title)', fontsize=12)
plt.show()
```



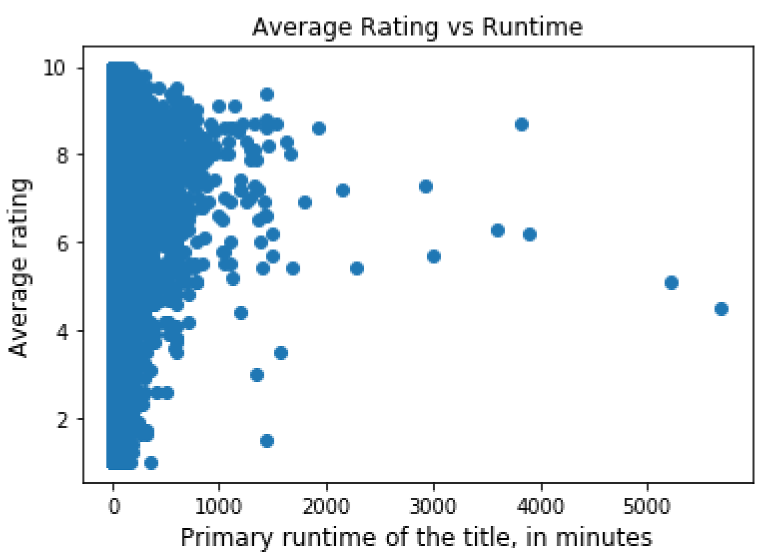
From above, we can see that the non-adult title tends to get higher average rating.

In [27]:
```
plt.scatter('numVotes','averageRating',data=movie)
plt.title('Average Rating vs Number of Votes')
plt.ylabel('Average rating', fontsize=12)
plt.xlabel('Number of votes the title has received', fontsize=12)
plt.show()
```



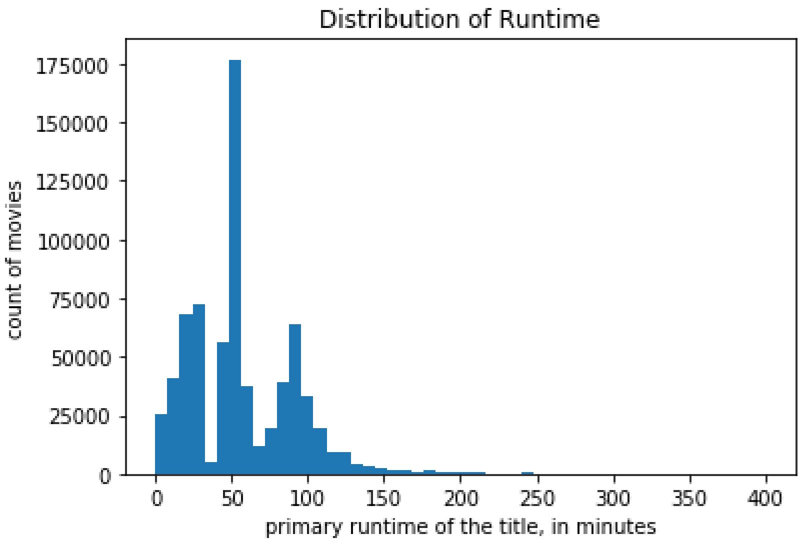
There is an increasing trend between variables numVotes and averageRating. But it is not linear, kind of quadratic.

In [28]:
```
plt.scatter('runtimeMinutes','averageRating',data=movie)
plt.title('Average Rating vs Runtime')
plt.ylabel('Average rating', fontsize=12)
plt.xlabel('Primary runtime of the title, in minutes', fontsize=12)
plt.show()
```

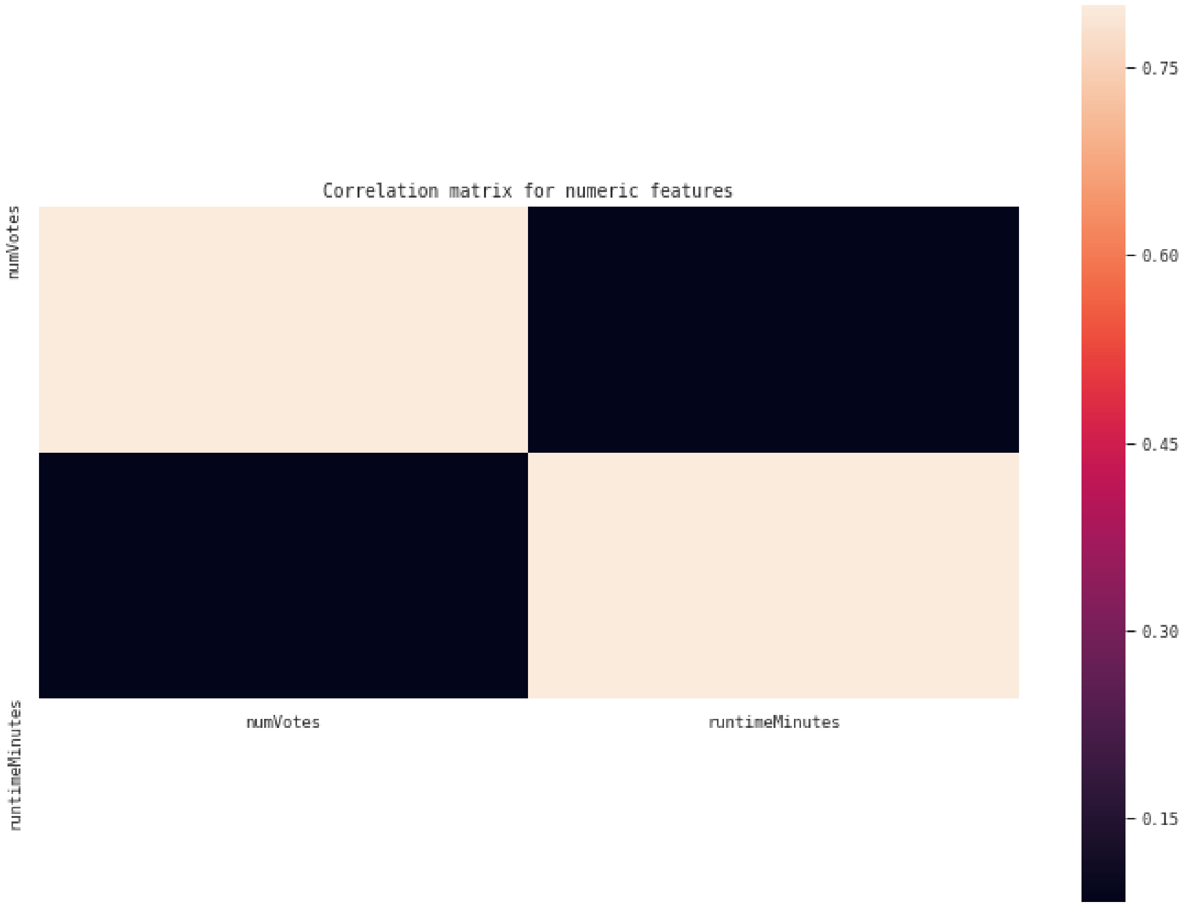


There is no linear trend between runtimeMinutes and averageRating.

In [29]:
```
plt.hist(movie['runtimeMinutes'],bins=50,  range=[0, 400])
plt.title("Distribution of Runtime")
plt.xlabel('primary runtime of the title, in minutes')
plt.ylabel('count of movies')
plt.show()
```



In [30]:
```
def corrmap(features, title):
    sns.set(context="paper", font="monospace")
    corrmat = movie[features].corr()
    f, ax = plt.subplots(figsize=(12, 9))
    plt.title(title)
    # Draw the heatmap using seaborn
    sns.heatmap(corrmat, vmax=.8, square=True)
corrmap(numeric_features,"Correlation matrix for numeric features")
```



There is almost no correlation between variables numVotes and runtimeMinutes.

In [31]:
```
import operator
from scipy.stats import pearsonr
correl={}
for f in numeric_features:
    correl[f]=pearsonr(movie[f], labels)
sorted_cor = sorted(correl.items(), key=operator.itemgetter(1), reverse=True)
print (sorted_cor)
```

```
[('numVotes', (0.016247686388758377, 1.469748170029789e-42)), ('runtimeMinute
s', (-0.19857033119223186, 0.0))]
```

The Pearson correlation coefficient beween numVotes and averageRating is 0.016247686388757378. The Pearson correlation coefficient beween runtimeMinutes and averageRating is -0.1985703311921796. They're all very small. There is not obvious linear relationship between averageRating and these two variables.

Data Preparation

The variable isAdult is already dummy variable. The rest categorical variables titleType, startYear,genres, directors, writers need to be recoded. Here the LabelEncoder is used. I prefer OneHotEncoder method, but there exists memory problem. There're two methods for the one hot encoder, one is get_dummies, the other is OneHotEncoder.

In [32]:
```
class MultiColumnLabelEncoder:
    def __init__(self,columns = None):
        self.columns = columns # array of column names to encode

    def fit(self,X,y=None):
        return self # not relevant here

    def transform(self,X):
        '''
        Transforms columns of X specified in self.columns using
        LabelEncoder(). If no columns specified, transforms all
        columns in X.
        '''
        output = X.copy()
        if self.columns is not None:
            for col in self.columns:
                output[col] = LabelEncoder().fit_transform(output[col])
        else:
            for colname,col in output.iteritems():
                output[colname] = LabelEncoder().fit_transform(col)
        return output

    def fit_transform(self,X,y=None):
        return self.fit(X,y).transform(X)
```

In [33]:
```
movie=MultiColumnLabelEncoder(columns = ['titleType', 'genres', 'directors',
'writers']).fit_transform(movie)
```

In [34]:
```
#mov = movie.copy()
#for feat in categorical_features:
#    mov=pd.concat([mov, pd.get_dummies(mov[feat], prefix=feat, dummy_na=Tru
e)],axis=1)
```

In [35]:
```
# TODO: create a OneHotEncoder object, and fit it to all of X

# 1. INSTANTIATE
#enc = OneHotEncoder(sparse=True,categories='auto')
#enc=LabelEncoder()
# 2. FIT
#enc.fit_transform(movie[categorical_features])
# 3. Transform
#onehotlabels = enc.transform(movie[categorical_features]).toarray()
#onehotlabels.shape
```

For numeric features, do the standardized transformation. This is necessary for advanced modeling, like distance based algorithm. If there is additional time, this part can be explored.

In [36]:
```
scl=StandardScaler()
movie[numeric_features]=scl.fit_transform(movie[numeric_features])
movie[numeric_features].head()
```

Out[36]:

	numVotes	runtimeMinutes
2	-0.070605	-0.959009
3	-0.067755	-0.959009
4	-0.070331	-0.934921
5	-0.023458	-0.983097
6	-0.070386	5.857939

In [37]:
```
movie.describe()
movie.head()
```

Out[37]:

	averageRating	numVotes	titleType	isAdult	startYear	runtimeMinutes	genres	directors	wri
2	4.2	-0.070605	1	0	1912	-0.959009	1392	278	208
3	6.8	-0.067755	1	0	1912	-0.959009	1759	278	128
4	6.8	-0.070331	1	0	1912	-0.934921	1782	35716	120
5	5.1	-0.023458	1	0	1914	-0.983097	1017	52523	
6	5.8	-0.070386	0	0	1914	5.857939	0	24870	66

Build Models

Although the linear regression assumptions are not satisfied, the linear regression is still built to do comparision with other models.

In [38]:
```
### Linear Model

X=movie.loc[:, movie.columns != "averageRating"]
y=labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
m_state=1) #split data set

lr = LinearRegression(normalize=True)
lr.fit(X_train,y_train)
y_pred = lr.predict(X_test)
print(lr.score(X_train,y_train))
print(r2_score(y_test,y_pred))
# calculate RMSE using scikit-learn
print(np.sqrt(mean_squared_error(y_test,y_pred)))
```

```
0.060365792914010674
0.061296882111901874
1.3405029634062784
```

We can see that the R square is 0.060365792914010674. It is very small. It confirms the previous conclusion: the linear relationship doesn't exist. But the The Root Mean Squared Error is 1.3405029634062784. It is small. Although the linear regression doesn't fit well, it can predict pretty good average rating.

In [39]:
```
# print the intercept and coefficients
print(lr.intercept_)
print(lr.coef_)
```

```
-4.542829243829906
[ 4.17491430e-02  6.67870102e-02 -8.90849834e-01  5.75778219e-03
 -2.37160353e-01 -1.32003442e-04 -1.75939157e-07 -2.86377211e-07]
```

In [40]:
```
## Polynomial Regression

XPol= PolynomialFeatures(degree=2, include_bias=False).fit_transform(X_train)
lrPol = LinearRegression().fit(XPol, y_train)
```

In [41]:
```
XPolT= PolynomialFeatures(degree=2, include_bias=False).fit_transform(X_test)
y_pred = lrPol.predict(XPolT)
print(lrPol.score(XPol,y_train))
print(r2_score(y_test,y_pred))
print(np.sqrt(mean_squared_error(y_test,y_pred)))
```

```
0.1443656802348655
0.14507144355839208
1.2792886890811008
```

The Root Mean Squared Error is 1.2792886890811008. It is smaller than 1.3405029634062784. This model is a little improved compared to the previous linear regression.

Since Decision Tree and Random Forest can capture non-linear relationship well, I wil fit these two models to see if better prediction can be achieved.

In [42]:
```
## Decision Tree Regression

regressor = DecisionTreeRegressor(random_state = 0)  # create a regressor obje
ct
regressor.fit(X_train, y_train) # fit the regressor
```

Out[42]:
```
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=0, splitter='best')
```

In [43]:
```
y_pred = regressor.predict(X_test)
df=pd.DataFrame({'Actual':y_test, 'Predicted':y_pred})
df
```

Out[43]:

	Actual	Predicted
850479	7.7	7.9
412095	8.2	8.4
838616	7.6	7.5
706338	8.5	7.8
390848	7.8	6.9
...	...	...
722934	7.4	7.7
949893	8.3	9.5
92668	5.9	6.0
289387	6.8	5.7
403029	6.3	5.5

141621 rows × 2 columns

We can see that the predicted value and the actural value are not so far away with each other.

In [44]:
```
print('Mean Absolute Error:', mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(mean_squared_error(y_test, y_pred)))
```

```
Mean Absolute Error: 0.9778110507384038
Mean Squared Error: 1.9908711471256961
Root Mean Squared Error: 1.4109823340941219
```

The Root Mean Squared Error is 1.4109823340941219. It is close to the the corresponing values in the previous models.

In [45]:
```
## Random Forest Regression

rf = RandomForestRegressor(n_estimators = 50, random_state = 42) # Instantiate
model with 50 decision trees
rf.fit(X_train, y_train) # Train the model on training data
```

Out[45]:
```
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=50,
                      n_jobs=None, oob_score=False, random_state=42, verbose=
0,
                      warm_start=False)
```

In [47]:
```
predictions = rf.predict(X_test) # Use the forest's predict method on the test
data
errors = abs(predictions - y_test) # Calculate the absolute errors
# Print out the mean absolute error (mae)
print('Mean Absolute Error:', round(np.mean(errors), 2))
print('Root Mean Squared Error:', np.sqrt(mean_squared_error(y_test, y_pred)))
```

```
Mean Absolute Error: 0.72
Root Mean Squared Error: 1.4109823340941219
```

The Root Mean Squared Error is 1.4109823340941219. It gets the same accuracy as the decision tree regression.

In [48]:
```
# Calculate mean absolute percentage error (MAPE)
mape = 100 * (errors / y_test)
# Calculate and display accuracy
accuracy = 100 - np.mean(mape)
print('Accuracy:', round(accuracy, 2), '%.')
```

```
Accuracy: 87.07 %.
```

The accuracy is 87.07 %. The model predicts pretty well.

Future Work

If there are additional time, I would like to explore the rest data sets among those seven data sets on IMDb website. When cleaning the data, I saw that the variable genres has more specific descriptions. They are strings separated by comma. Next time, I can split genres into several variable, then create a few more categorical variables. For recoding part, I wonder if model will be improved when one hot encoder is used instead of the label encoder. For model buiding part, I would like to try machine learning pipeline. I think this can produce more complex models and give better prediction.