1 Introduction [10 points]

Team name:

Turtle School Son Goku.

Group member(s):

Liting Xiao.

Division of labor:

The whole of the project, including data exploration, data processing, basic visualization, matrix factorization, matrix factorization visualization, and the writing of this report.

2 Basic Visualizations [20 points]

0. Data preprocessing.

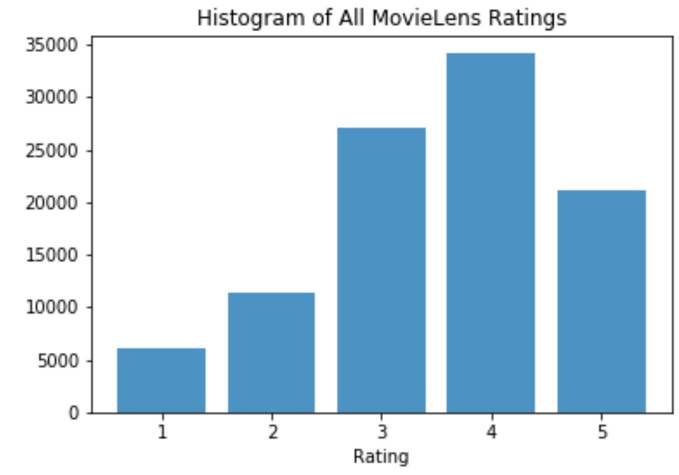
I used *pandas* to read in the data files ‘movies.txt‘ and ‘data.txt‘, and I added in corresponding column names to the dataframes for easier access to both datasets.

There are movies with duplicate titles but different IDs in ‘movies.txt‘. And user ratings are assigned to different IDs that are actually the same movies. So, I grouped together these duplicate movie entries in ‘movies.txt‘ to record the duplicate IDs. Then in ‘data.txt‘, for each duplicate ID group, I replaced the duplicates with the first (lowest numbered) ID registered to the same movies.

1. All ratings in the MovieLens Dataset.

A histogram of ratings of all movies in the MovieLens dataset is plotted below. The mean is 3.53, and the standard deviation is 1.126.

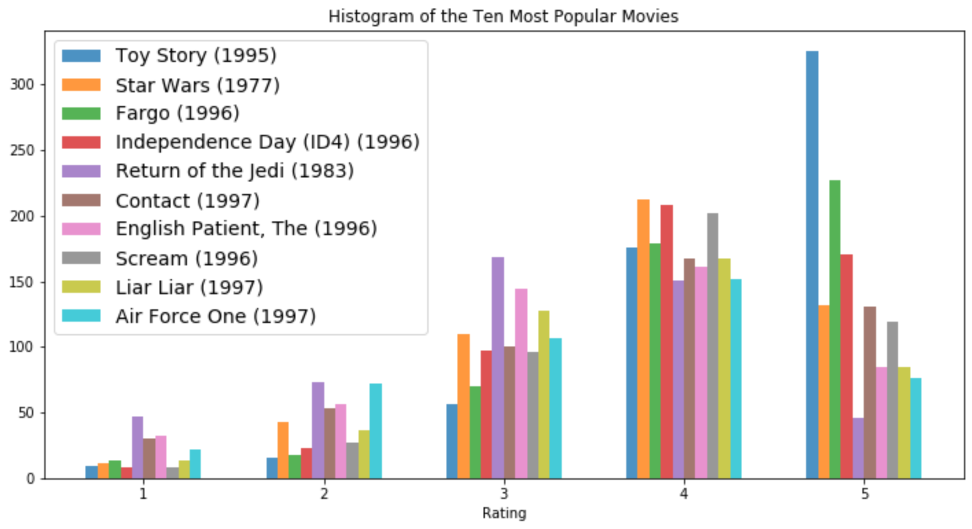
There are more movies with ratings of 3, 4, 5 than of 1, 2. The distribution is skewed towards the right. With the general shape of the all rating distribution in mind, let’s move on to more specific aspects of our datasets.



2. All ratings of the ten most popular movies (movies which have received the most ratings).

A histogram of ratings of the 10 most popular movies in the MovieLens dataset is plotted below. The mean is 3.77, and the standard deviation is 1.079.

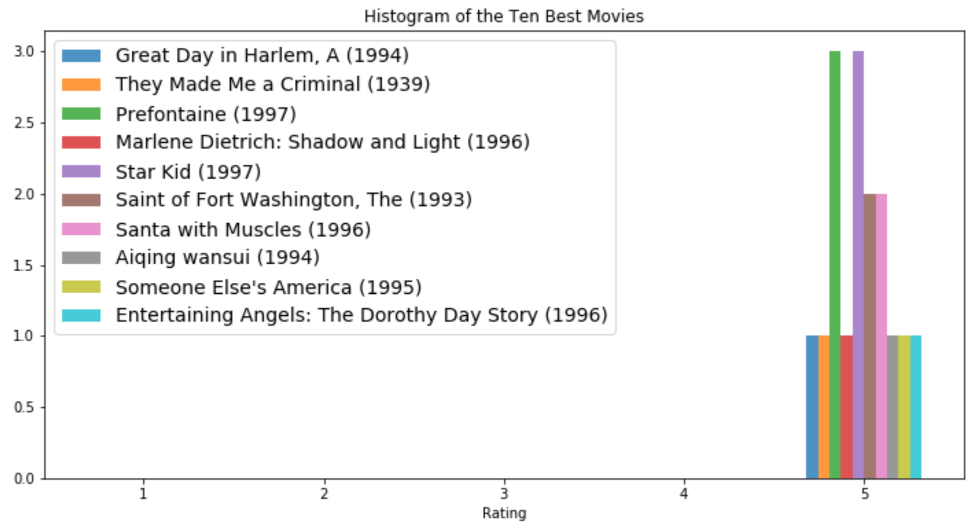
Compared to the histogram of all ratings, the distribution shifts even more towards the higher rating part. This is as expected since these movies are most popular most likely because more people watched them, liked them and recommended them.



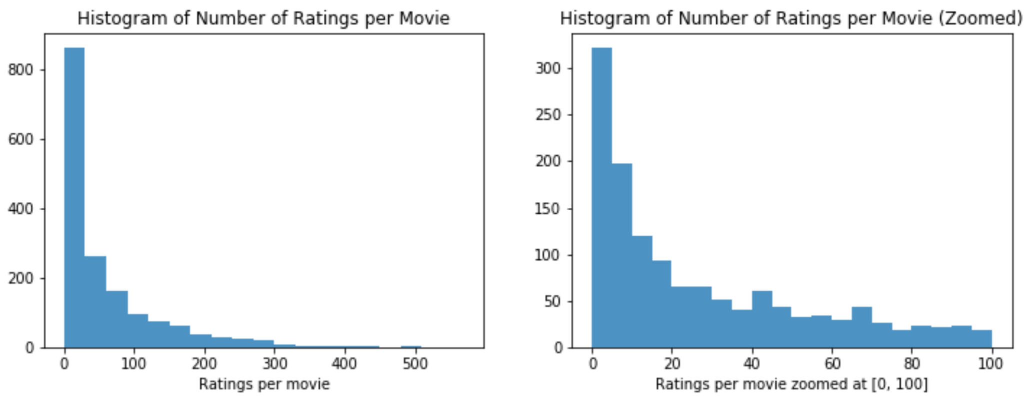
3. All ratings of the ten best movies (movies with the highest average ratings).

Naively, a histogram of ratings of the 10 movies with the highest average ratings in the MovieLens dataset is plotted below. The mean is 5.0, and the standard deviation is 0.

These 10 movies only have ratings of 5, but notice that's because only 1-3 people have rated them. This plot is not very informative in a sense that these movies are not very popular and could not be termed as "the best" movies. With the small number of ratings gathered for these movies, the distribution is not meaningful as well with such a small sample size.



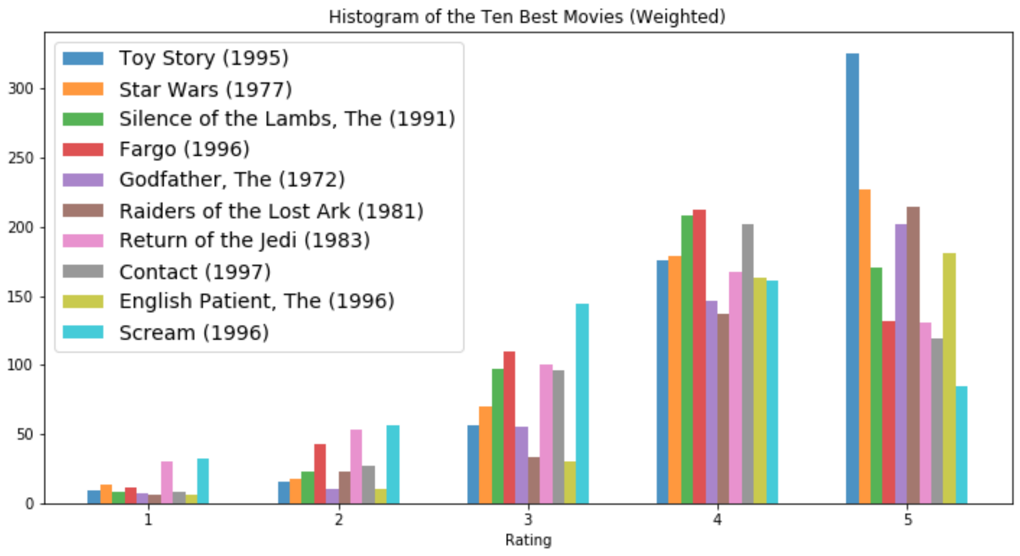
We can take a look at the distribution of number of ratings per movie. And it's noticeable that we have a skewed dataset towards lower number of ratings. Let's also see this by zooming in to [0, 100] ratings per movie range.



Therefore, a more useful plot would be a histogram of all ratings of the ten best movies (movies with the highest average ratings, **weighted** by the number of ratings). Such a histogram is shown below.

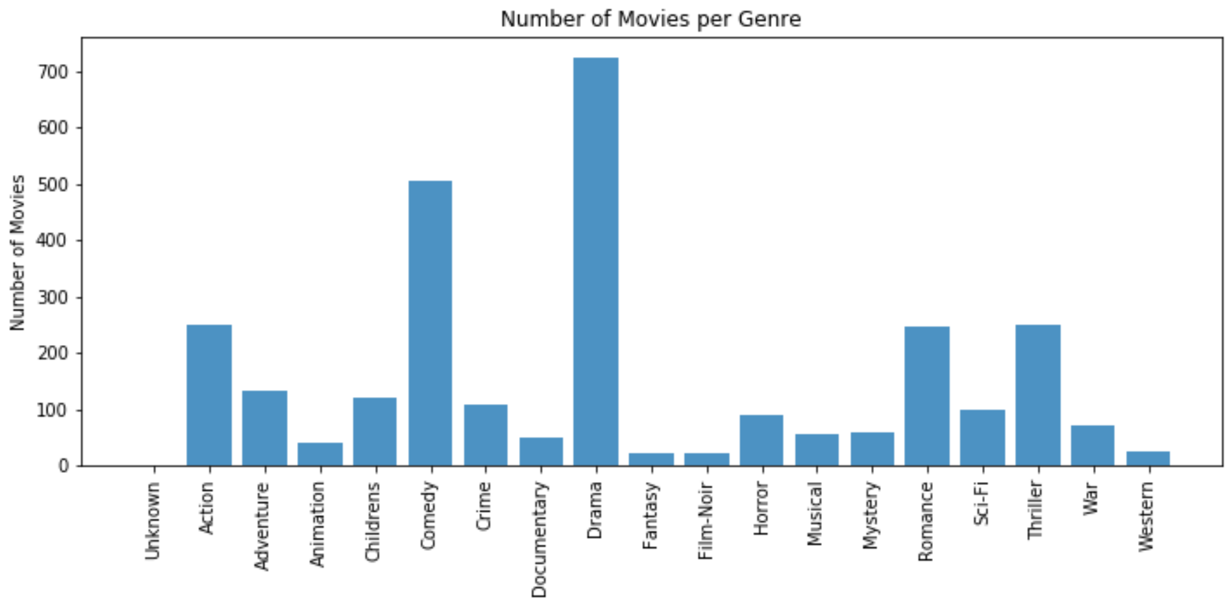
For the actual 10 best movies, the mean is 4.01, and the standard deviation is 1.015.

Compared to the 10 most popular movies, we see that these 2 categories have 7 overlaps: *Toy Story (1995), Star Wars (1977), Fargo (1996), Return of the Jedi (1983), Contact (1997), English Patient, The (1996), Scream (1996)*. Thus, as expected, the distribution looks alike, with the 10 best movie rating distribution shifting even further towards the right.



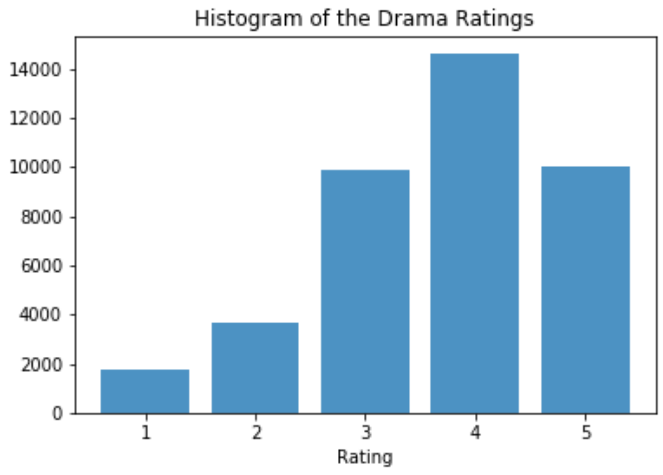
4. All ratings of movies from three genres of your choice (create three separate visualizations).

Let's first visualize the number of movies per genre in our dataset and then pick a range of 3 from them.

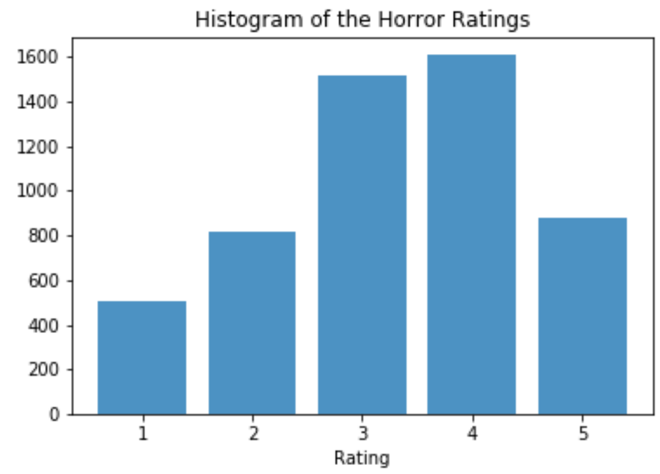


I picked 3 genres with the most, and medium, and few number of movies labeled: *Drama, Horror, Fantasy*. We can see from the following 3 plots that the more a genre is popular, the more likely that the viewers will rate them higher, at least in our MovieLens dataset. Popularity of movies seems to be associated with their viewer ratings.

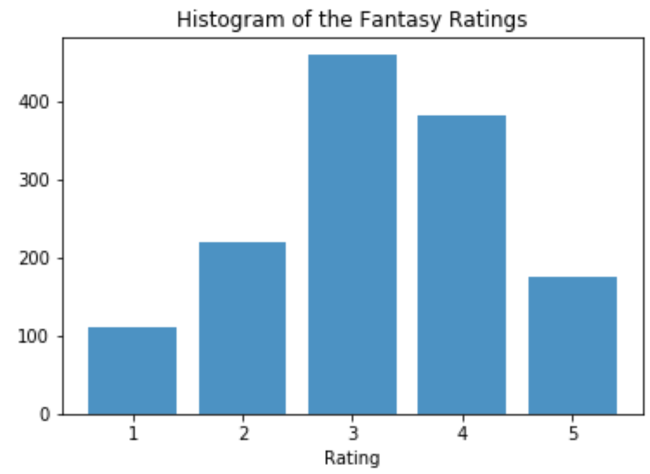
The mean of ratings in the most popular *Drama* genre is 3.69, and the standard deviation 1.079.



The mean of ratings in the medium popular *Horror* genre is 3.29, and the standard deviation 1.187.



The mean of ratings in the less popular *Fantasy* genre is 3.22, and the standard deviation 1.119.



3 Matrix Factorization Methods [40 points]

1. Re-use code from HW5, not including biases.

The first method is the standard SVD from HW5. The standard loss function with regularization is:

$$J = \frac{\lambda}{2}(||U||^2 + ||V||^2) + \frac{1}{2}\sum_{(i,j)\in S}(y_{ij} - u_i^T v_j)^2.$$

For implementation, I calculate the gradients ∂_{u_i} and ∂_{v_j}.

$$\partial_{u_i} = \lambda u_i - \sum_j v_j(y_{ij} - u_i^T v_j),$$
$$\partial_{v_j} = \lambda v_j - \sum_i u_i(y_{ij} - u_i^T v_j).$$

Then we can use the gradients to calculate U and V by stochastic gradient descent.

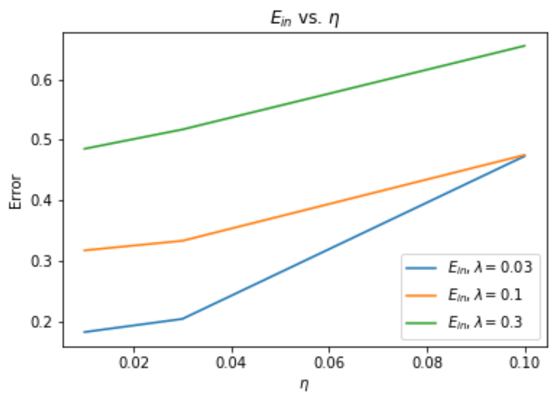
$$u_i = u_i - \eta\partial_{u_i},$$
$$v_j = v_j - \eta\partial_{v_j}.$$

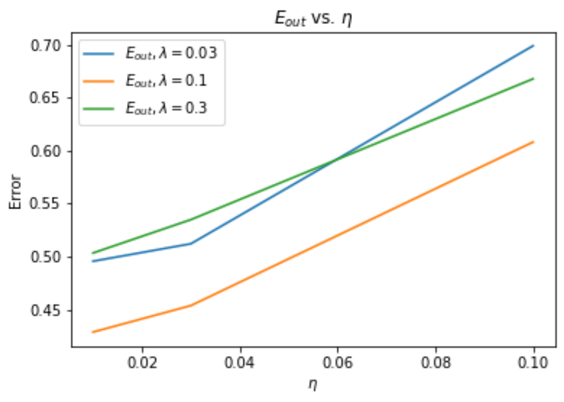
For training, I initialize U and V to be uniform random variables in [-0.5, 0.5]. Our data are 1-indexed, and this is dealt with in the training process. Here, we use $K = 20$ latent factors, maximum epochs of 300. We also use early stopping technique such that when the fraction of loss descent is less than $\epsilon = 0.0001$, we will stop the algorithm.

To determine values of other hyper-parameters (in our case, regularization λ and learning rate η), we loop through these hyper-parameters, train the model, and see the in-sample and out-of-sample performance, E_{in} and E_{out}.

From HW5, we know that $\lambda = 0.1$ and $\eta = 0.03$ work pretty well already. So, let's try values in the vicinities. I tried $\lambda = [0.03, 0.1, 0.3]$ and $\eta = [0.01, 0.03, 0.1]$ in the interest of time. The following 2 plots show the performance of the method in these hyper-parameter ranges. We can see that $\lambda = 0.1$ and $\eta = 0.01$ work the best with the lowest E_{out} and no overfitting or underfitting occurs.

Lastly, we train using the whole dataset with $\lambda = 0.1$ and $\eta = 0.01$.

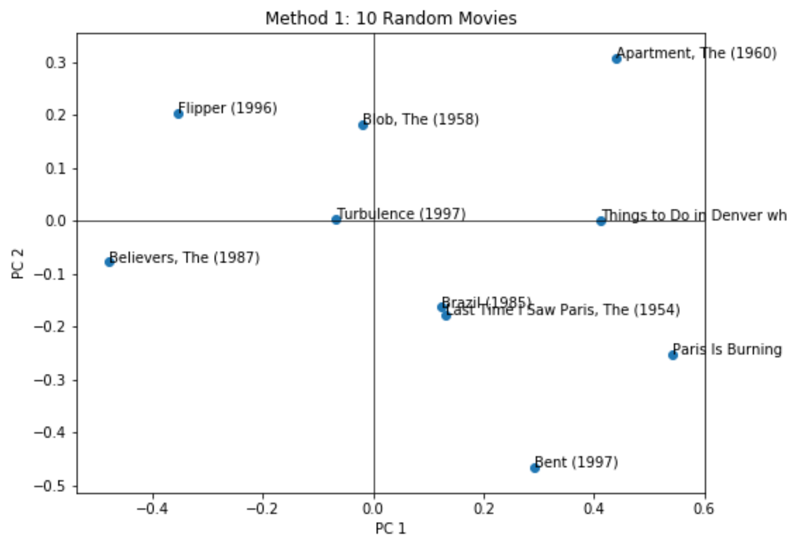




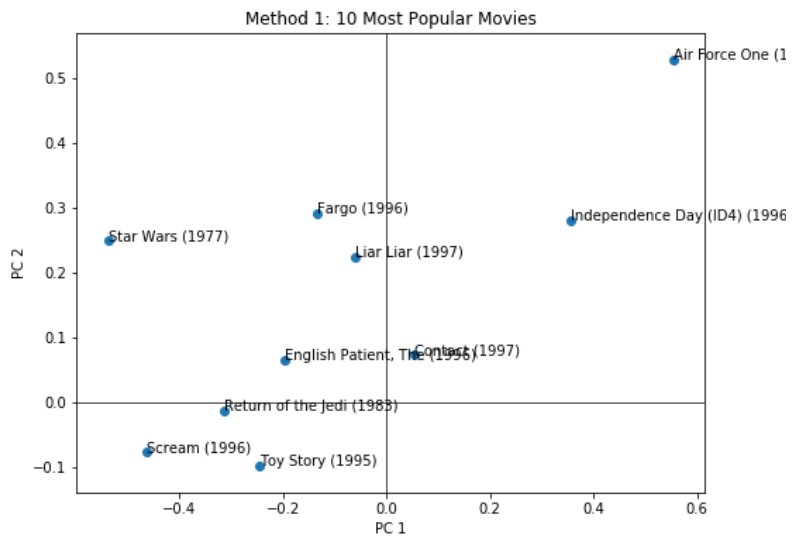
Now, in order to visualize the resulting latent factors, I apply SVD to $V = A\Sigma B$ and use the first 2 columns of A to project U and V into a 2D space. The projection is given by $\tilde{U} = A_{1:2}^T U$ and $\tilde{V} = A_{1:2}^T V$.

Now, we visualize the following:

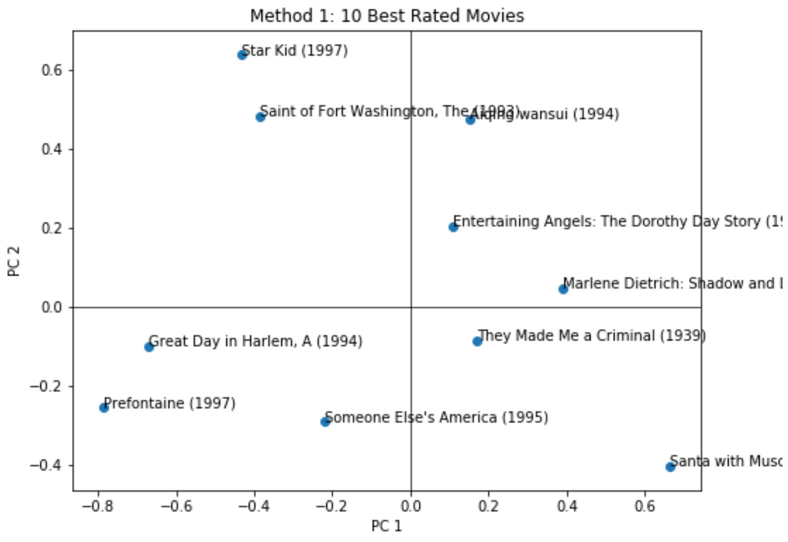
(a) 10 random movies from the MovieLens dataset.



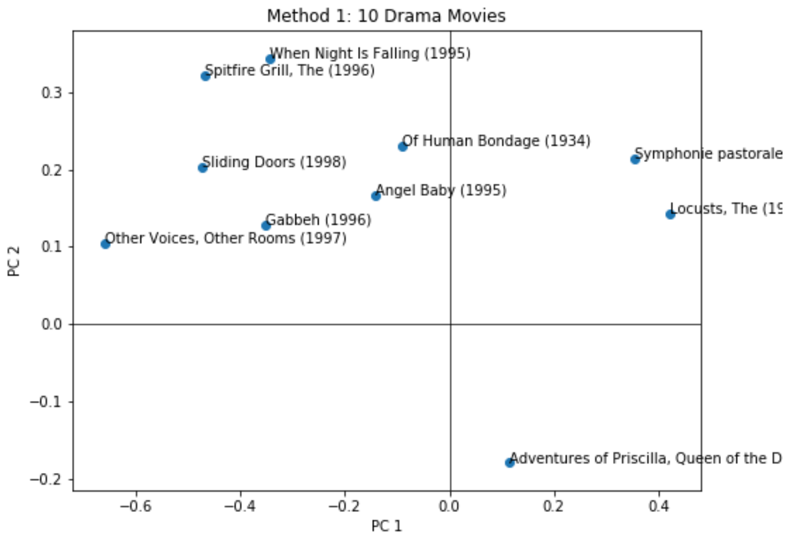
(b) The 10 most popular movies (movies which have received the most ratings).

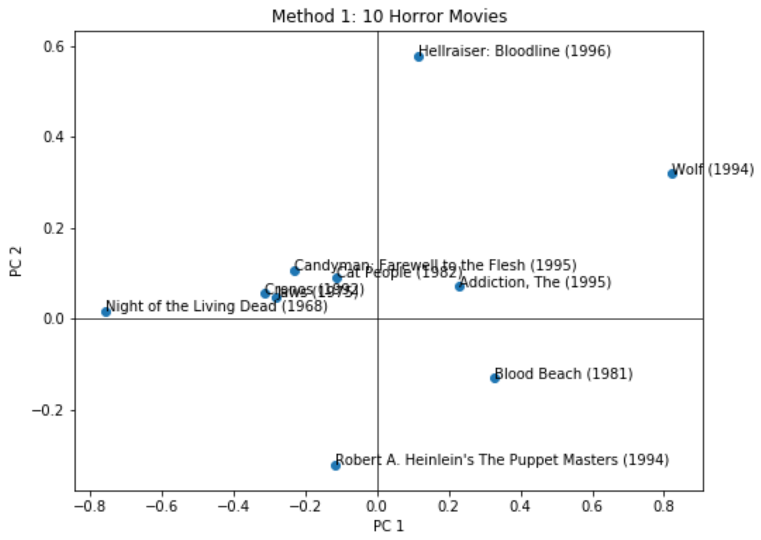


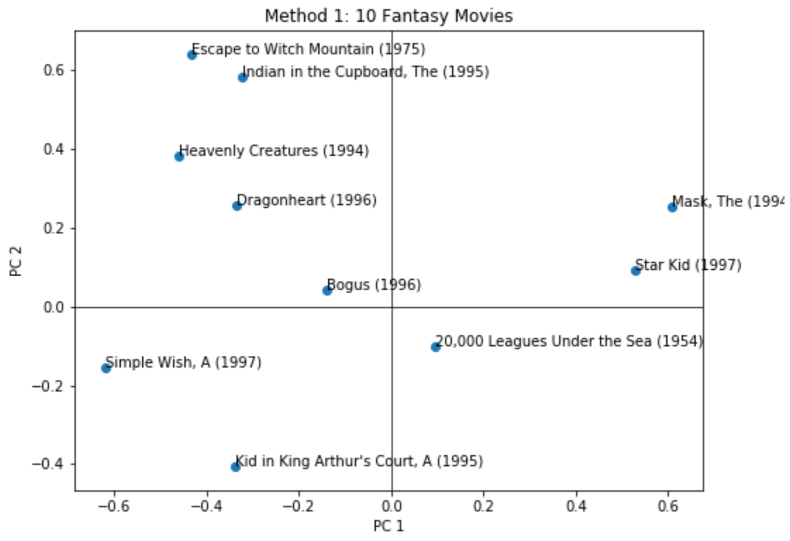
(c) The 10 best movies (movies with the highest average ratings).



(d) 10 Movies from [*′Drama′*, *′Horror′*, *′Fantasy′*].







2. Re-use code from HW5, but incorporate bias terms for each user and movie.

The second method is similar to method 1, but now we also model the global biases of movies and users. The loss function with regularization is then:

$$J = \frac{\lambda}{2}(||U||^2 + ||V||^2 + ||a||^2 + ||b||^2) + \frac{1}{2}\sum_{(i,j)\in S}((y_{ij} - \mu) - (u_i^T v_j + a_i + b_j))^2.$$

The gradients ∂_{u_i}, ∂_{v_j}, ∂_{a_i}, ∂_{b_j} are:

$$\begin{aligned}
\partial_{u_i} &= \lambda u_i - \sum_j v_j((y_{ij} - \mu) - (u_i^T v_j + a_i + b_j)),\\
\partial_{v_j} &= \lambda v_j - \sum_i u_i((y_{ij} - \mu) - (u_i^T v_j + a_i + b_j)),\\
\partial_{a_i} &= \lambda a_i - \sum_i((y_{ij} - \mu) - (u_i^T v_j + a_i + b_j)),\\
\partial_{b_j} &= \lambda b_j - \sum_i((y_{ij} - \mu) - (u_i^T v_j + a_i + b_j)).
\end{aligned}$$

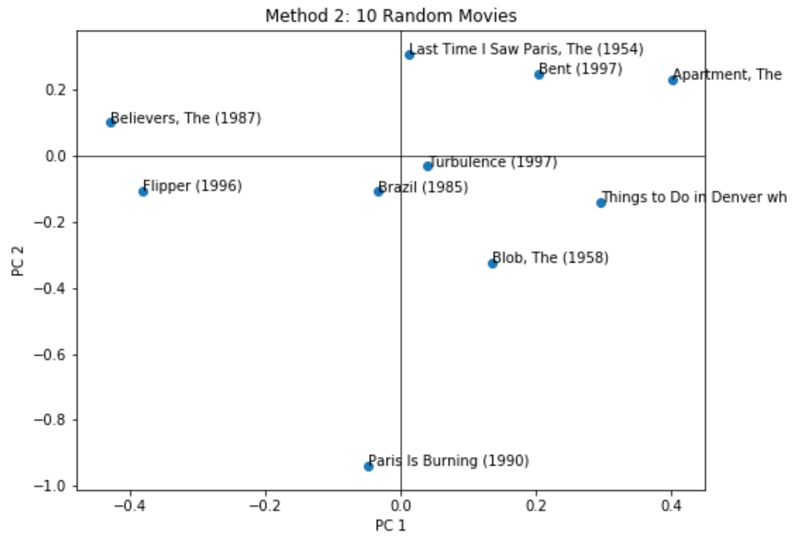
Then we can use these gradients to calculate U, V, a and b by stochastic gradient descent.

$$\begin{aligned}
u_i &= u_i - \eta\partial_{u_i},\\
v_j &= v_j - \eta\partial_{v_j},\\
a_i &= a_i - \eta\partial_{a_i},\\
b_j &= b_j - \eta\partial_{b_j}.
\end{aligned}$$

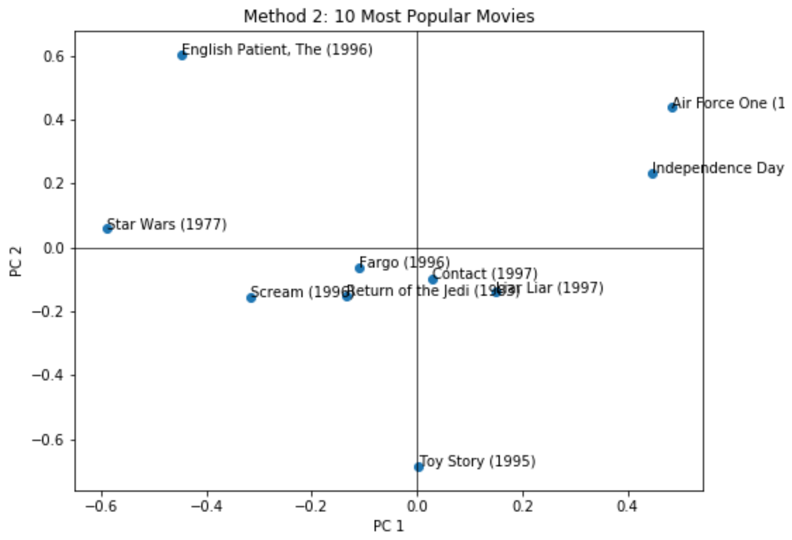
Again, for training, I initialize U, V, a and b to be uniform random variables in [-0.5, 0.5]. Other variables and details are the same. I modified functions in method 1 to take in additional arguments if 'bias' is set to 'True'. As for regularization and learning rate, since this method is quite similar with method 1, we could assume that the same set of optimal hyper-parameters apply here. So we use $\lambda = 0.1$ and $\eta = 0.01$.

Again, we apply SVD to V and project V into a 2D space. Now, we visualize the following:

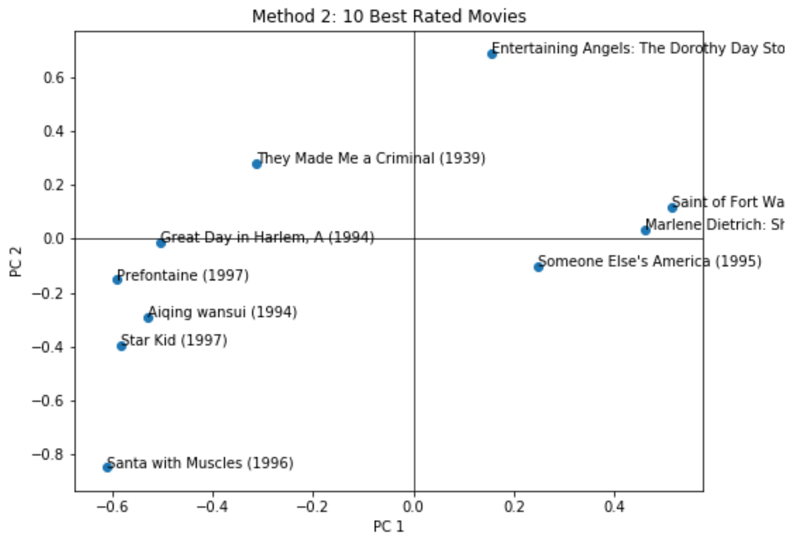
(a) 10 random movies from the MovieLens dataset.



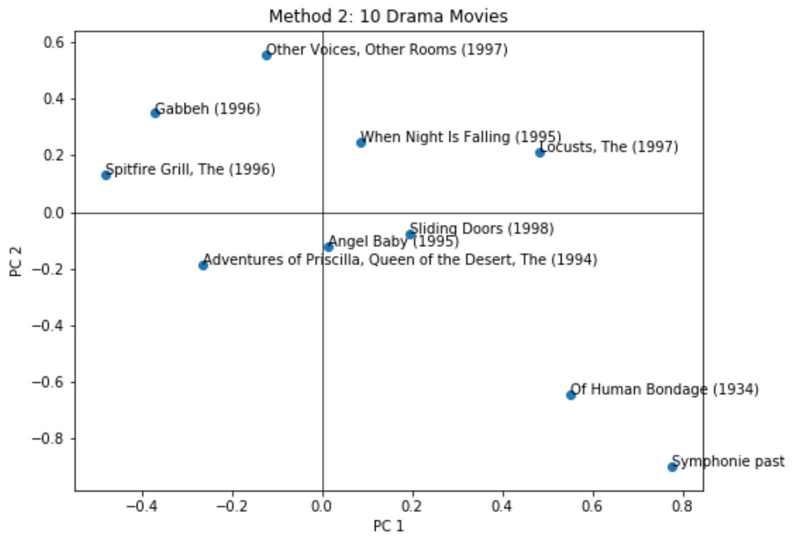
(b) The 10 most popular movies (movies which have received the most ratings).

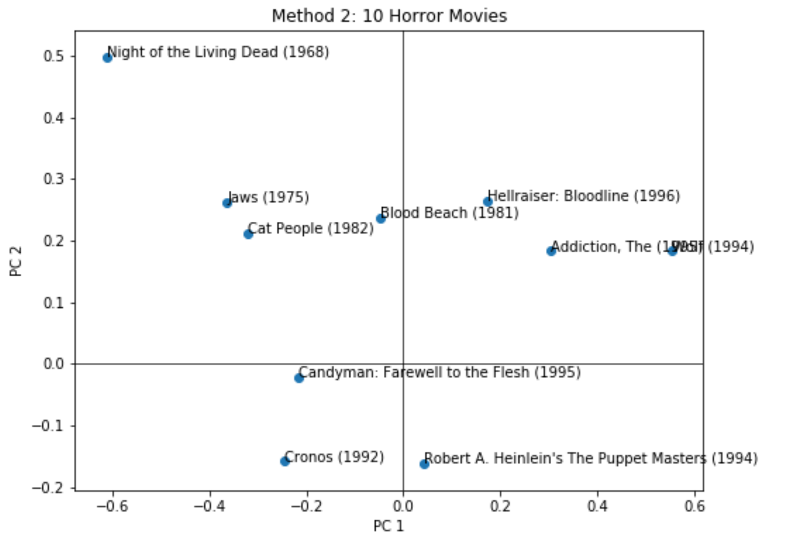


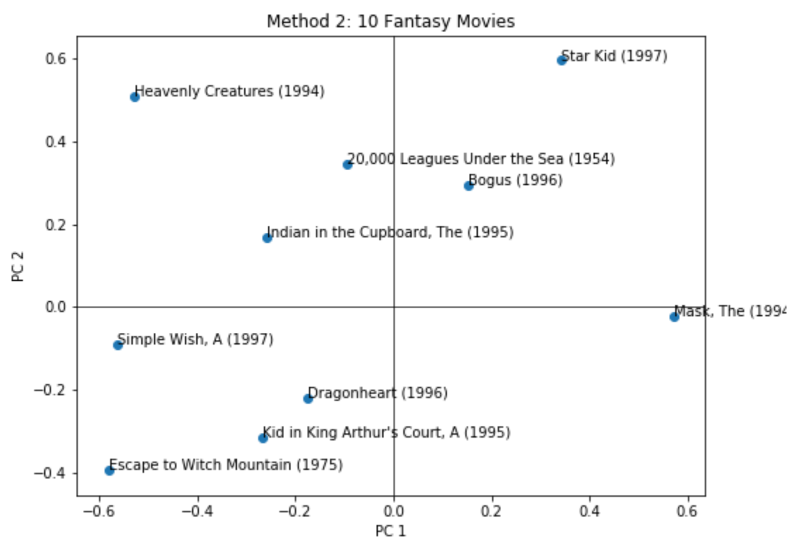
(c) The 10 best movies (movies with the highest average ratings).



(d) 10 Movies from [*'Drama'*, *'Horror'*, *'Fantasy'*].


Method 2: 10 Drama Movies
Other Voices, Other Rooms (1997)
Gabbeh (1996)
When Night Is Falling (1995)
Locusts, The (1997)
Spitfire Grill, The (1996)
Sliding Doors (1998)
Angel Baby (1995)
Adventures of Priscilla, Queen of the Desert, The (1994)
Of Human Bondage (1934)
Symphonie past
PC 1
PC 2


Method 2: 10 Horror Movies
Night of the Living Dead (1968)
Jaws (1975)
Hellraiser: Bloodline (1996)
Blood Beach (1981)
Cat People (1982)
Addiction, The (1995)
Wolf (1994)
Candyman: Farewell to the Flesh (1995)
Cronos (1992)
Robert A. Heinlein's The Puppet Masters (1994)
PC 1
PC 2



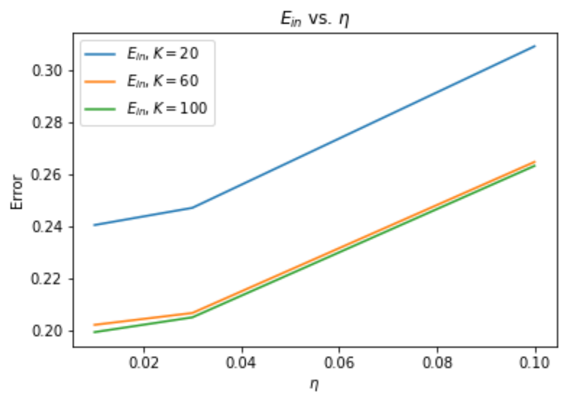
3. Off-the-shelf implementation using Surprise SVD.

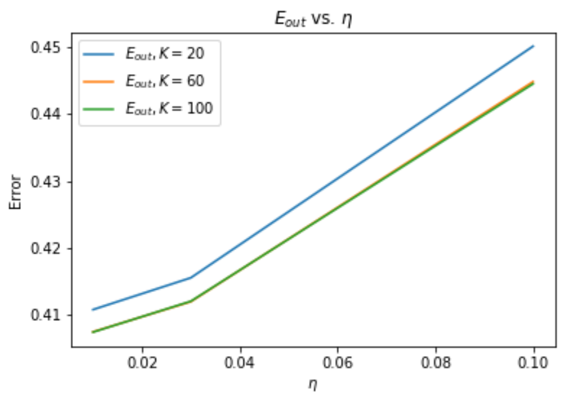
Lastly for the third method, I use an off-the-shelf implementation of matrix factorization, the *Surprise* module.

Before implementation of this method, we notice that some movies don't have ratings in the training set 'train.txt'. This will mess up with the factorization and indexing and the predictions won't be accurate because of these missing entries. So first, 'Movie Id' needs to be re-indexed to [1, len(movies in the train set)].

Then I load the train set, the test set, and the whole data set from *pandas* dataframe objects to *Surprise* Dataset objects.

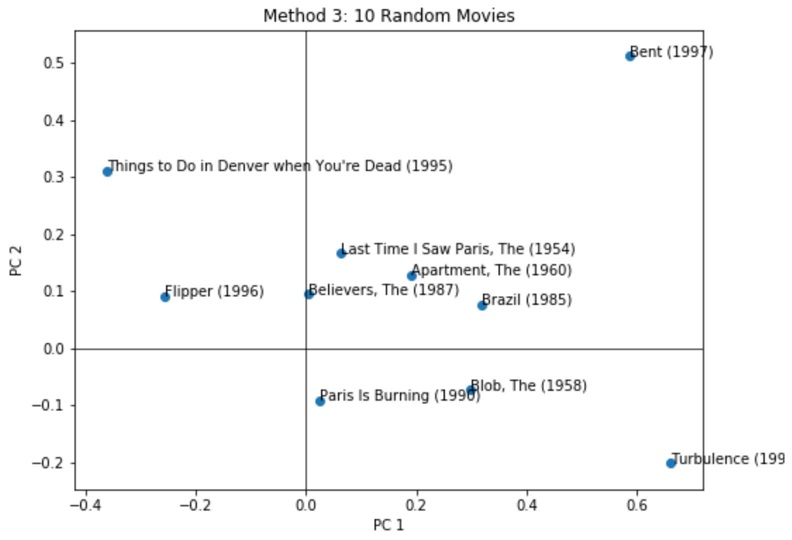
To choose hyper-parameters, we look at the number of latent factors and learning rate here. I decided not to look at regularization because it doesn't matter too much for us anyway and in the interest of time. So, I looped through $K = [20, 60, 100]$ and $\eta = [0.01, 0.03, 0.1]$ to see this method's performance. And the following 2 plots show the in-sample and out-of-sample performance. We can see that $K = 100$ and $\eta = 0.01$ work the best with the lowest E_{out} and no overfitting or underfitting occurs.



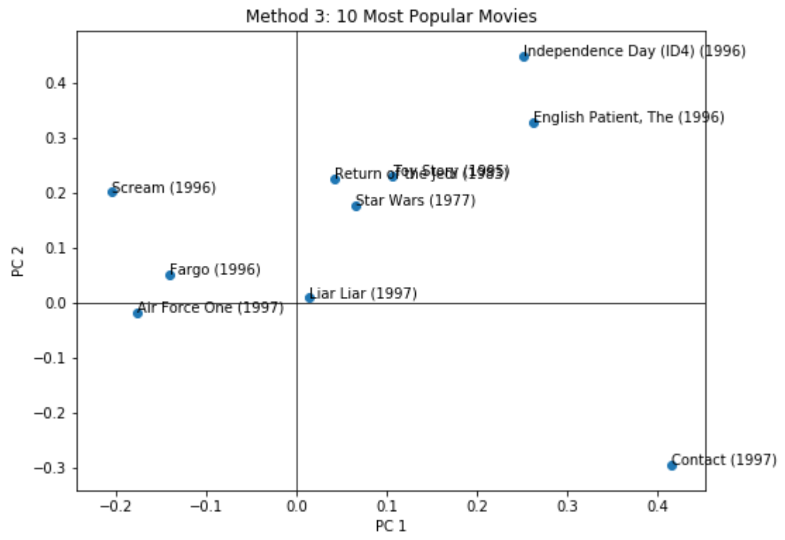


Again, we apply SVD to V and project V into a 2D space. Now, we visualize the following:

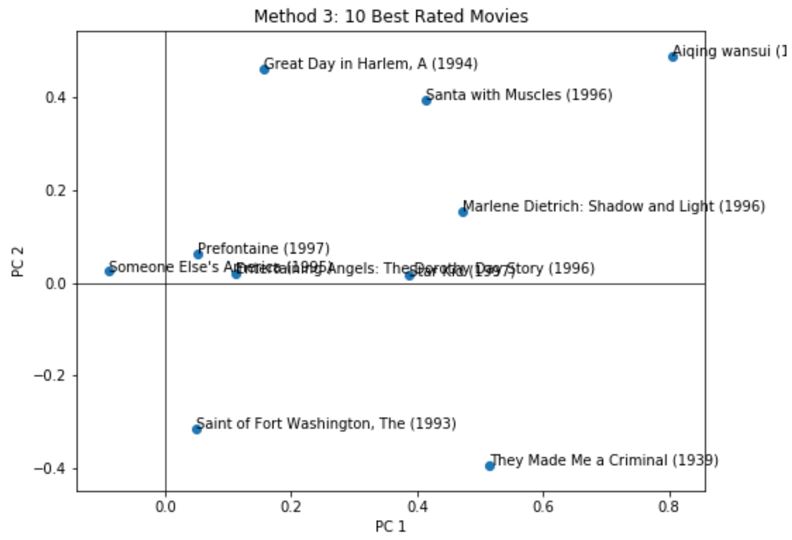
(a) 10 random movies from the MovieLens dataset.



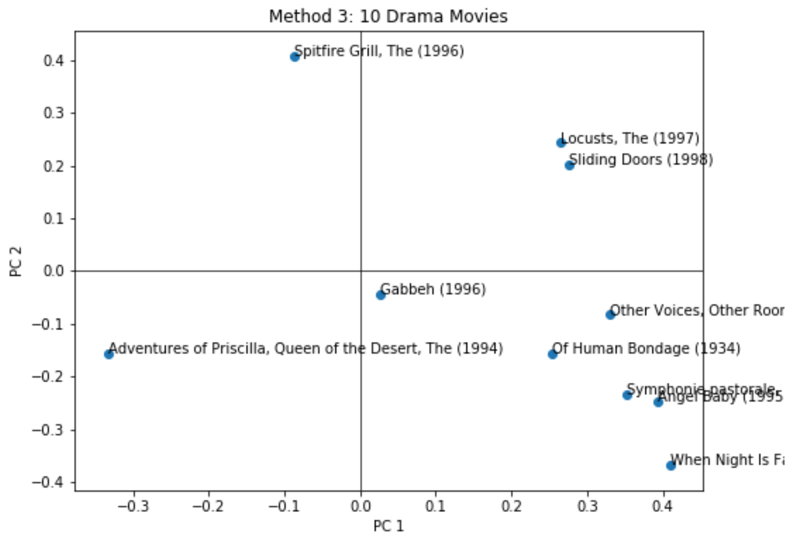
(b) The 10 most popular movies (movies which have received the most ratings).

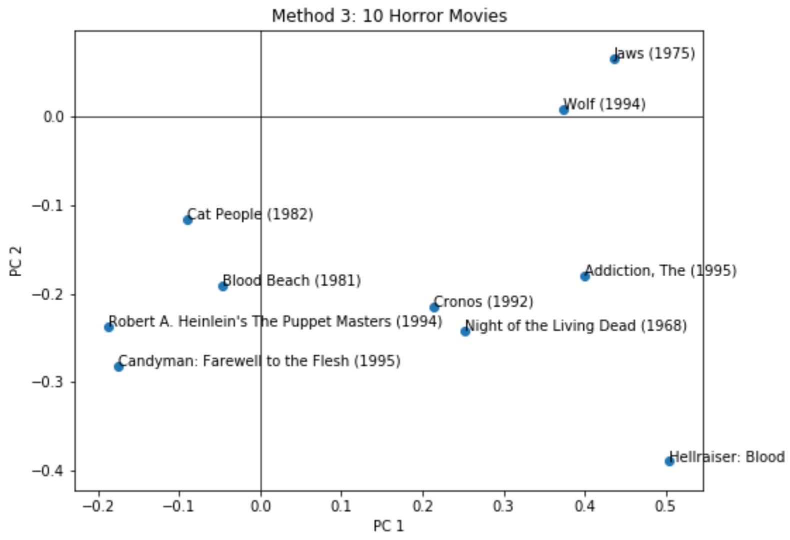


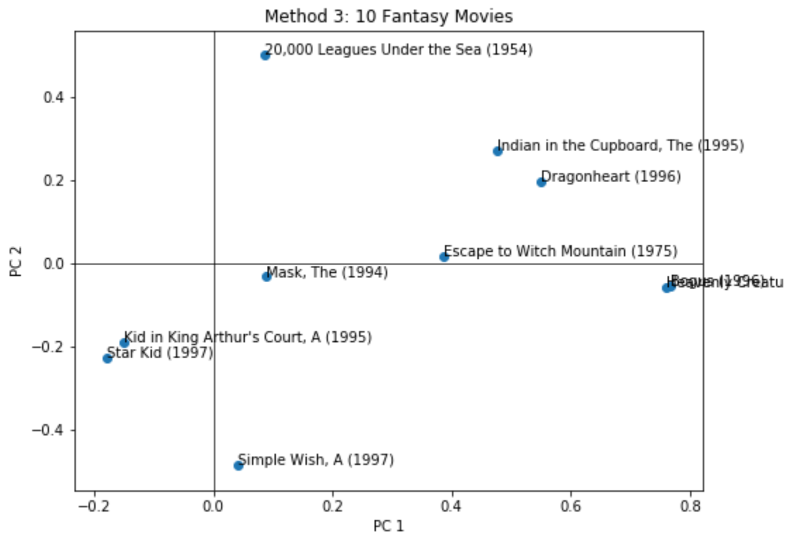
(c) The 10 best movies (movies with the highest average ratings).



(d) 10 Movies from [*'Drama'*, *'Horror'*, *'Fantasy'*].


Method 3: 10 Drama Movies
Spitfire Grill, The (1996)
Locusts, The (1997)
Sliding Doors (1998)
Gabbeh (1996)
Other Voices, Other Roor
Adventures of Priscilla, Queen of the Desert, The (1994)
Of Human Bondage (1934)
Symphonie pastorale,
Angel Baby (1995
When Night Is Fa
PC 1
PC 2


Method 3: 10 Horror Movies
Jaws (1975)
Wolf (1994)
Cat People (1982)
Addiction, The (1995)
Blood Beach (1981)
Cronos (1992)
Robert A. Heinlein's The Puppet Masters (1994)
Night of the Living Dead (1968)
Candyman: Farewell to the Flesh (1995)
Hellraiser: Blood
PC 1
PC 2



4. Comparison of the three methods.

Method 1 and 2 are similar in the sense that I implemented these two myself using SGD while method 3 is using off-the-shelf implementation.

Method 2 and 3 are similar in the sense that these two use the same MSE loss function including global biases, while method 1 doesn't include biases.

The performance in terms of E_{in} and E_{out} are shown below for these three methods. Method 1 and 2 have almost the same performance E_{out}, but E_{in} is lower for method 2 than for method 1, because method 2 has two more vectors of global biases to fit in the model. So we can expect method 2 to have better E_{in} than method 1.

Method 1: E_in = 0.3172, E_out = 0.4287.

Method 2: E_in = 0.2550, E_out = 0.4274.

Method 3: E_in = 0.1993, E_out = 0.4075.

Method 3 works the best both in E_{in} and E_{out}, because here we are fitting using more latent factors $K = 100$. So the better performance is expected as well.

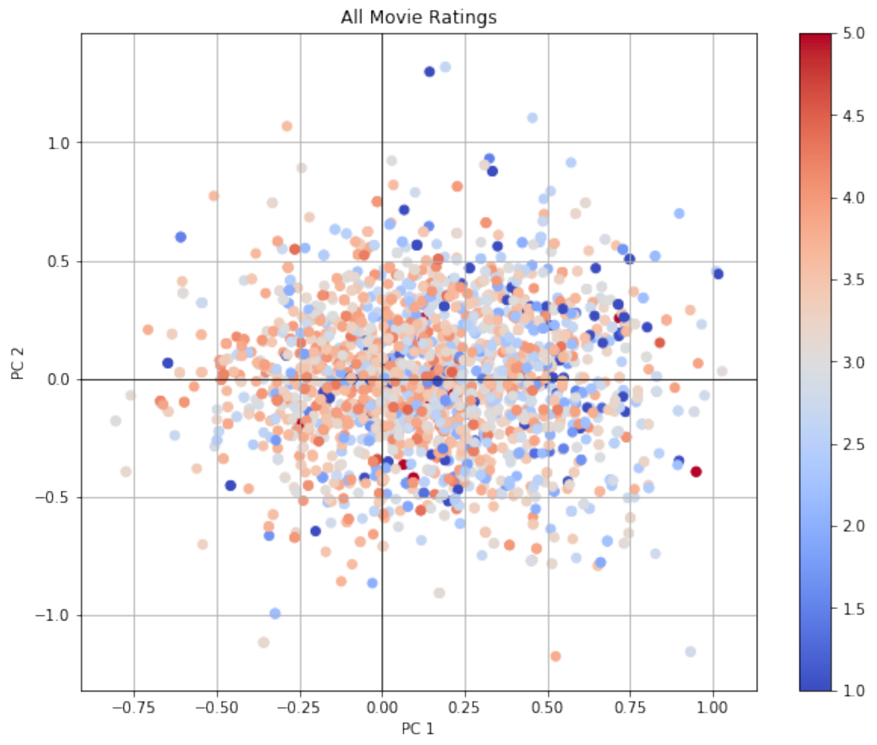
I found method 3 also worked much faster than the other two, even with a much larger number of latent factors. And method 3 has the best performance. So, for real-life implementation, I would go with the off-the-shelf modules.

4 Matrix Factorization Visualizations [30 points]

(I would leave most of the plots up in Section 3 to save space and mostly just summarize my findings here.)

- General comments:

In general, I think the 2D visualizations are still very randomly distributed in space. I would not say that they are very good visualizations. It's hard to discover many trends out of these 2D visualizations (to my eyes). Perhaps going up to 3D would help. Here, I show a plot of all the movies in this 2D space using method 2, with their ratings (mean of all ratings of a movie) as the color bar. I think method 2 shows my favorite characteristics: highly rated movies seems to cluster near the origin.



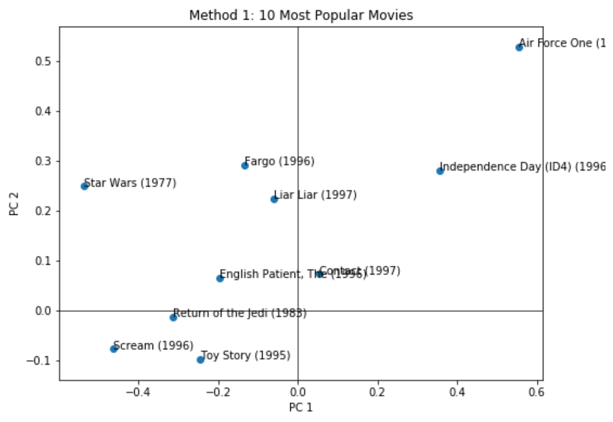
- For the 10 random movies:

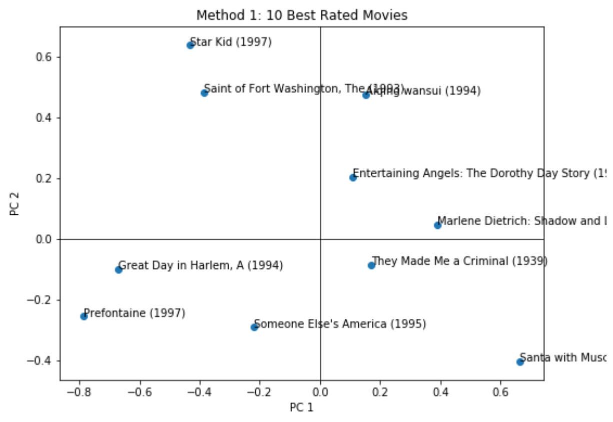
Just as expected, they were randomly distributed in space for all 3 methods. Nothing special here.

- For the 10 most popular movies compared to the 10 best movies:

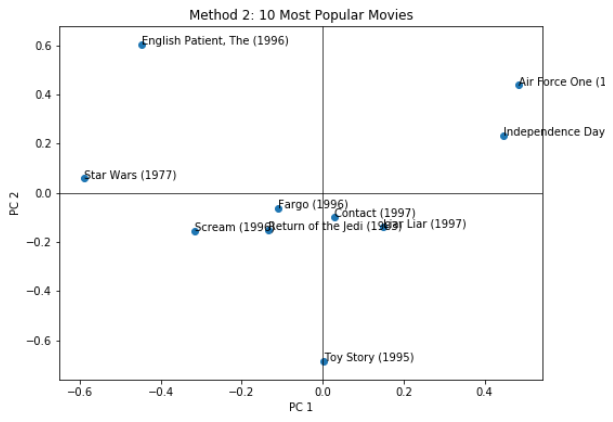
These 2 categories seem to show opposite traits here.

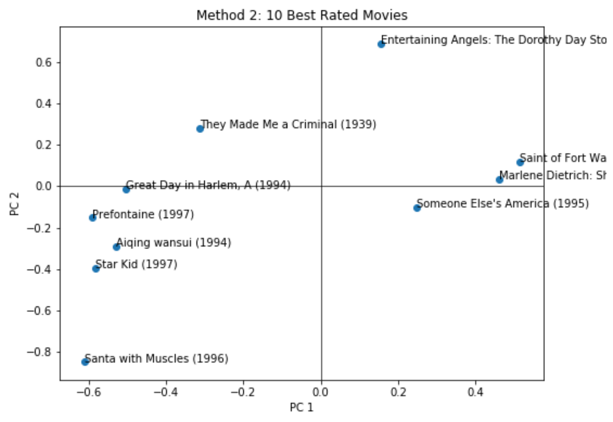
For method 1, the 10 most popular seem to cluster near $\text{PC2} = 0$, while the 10 best seem to disperse from $\text{PC2} = 0$. (Note that the scales of the plots are different.)



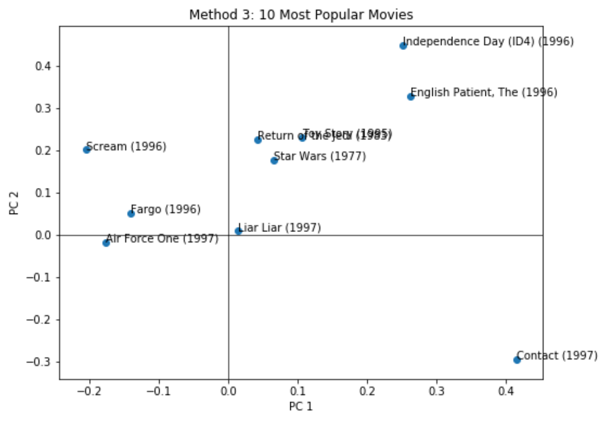


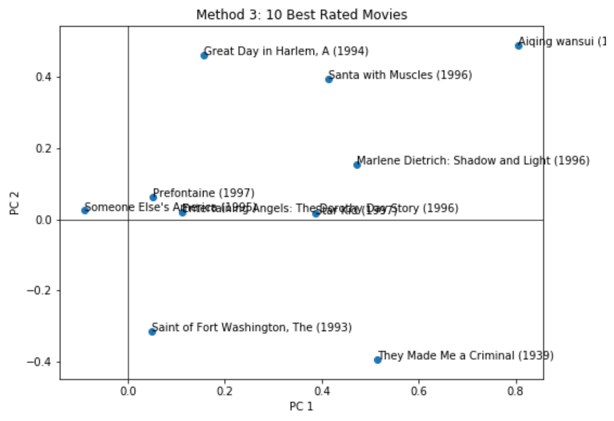
For method 2, the 10 most popular seem to cluster near the origin, while the 10 best seem to disperse from it.





For method 3, the 10 most popular seem to cluster near $\mathrm{PC1} = 0$, while the 10 best seem to disperse from $\mathrm{PC1} = 0$. (Note that the scales of the plots are different.)





This seems to be saying that the most popular movies are very different from the best rated movies, which is counterintuitive. And just as I explained and digged deeper in Section 2, the best movies here are not really "best rated" since they have only 1-3 ratings and they could not represent the whole distributions well.

- For Drama, Horror, Fantasy genres:

For method 1: Drama movies seem to cluster in the $\mathrm{PC2} > 0$ quadrants; Horror movies cluster near $\mathrm{PC2} = 0$; Fantasy movies are more away from the origin, perhaps indicating their inpopularity.

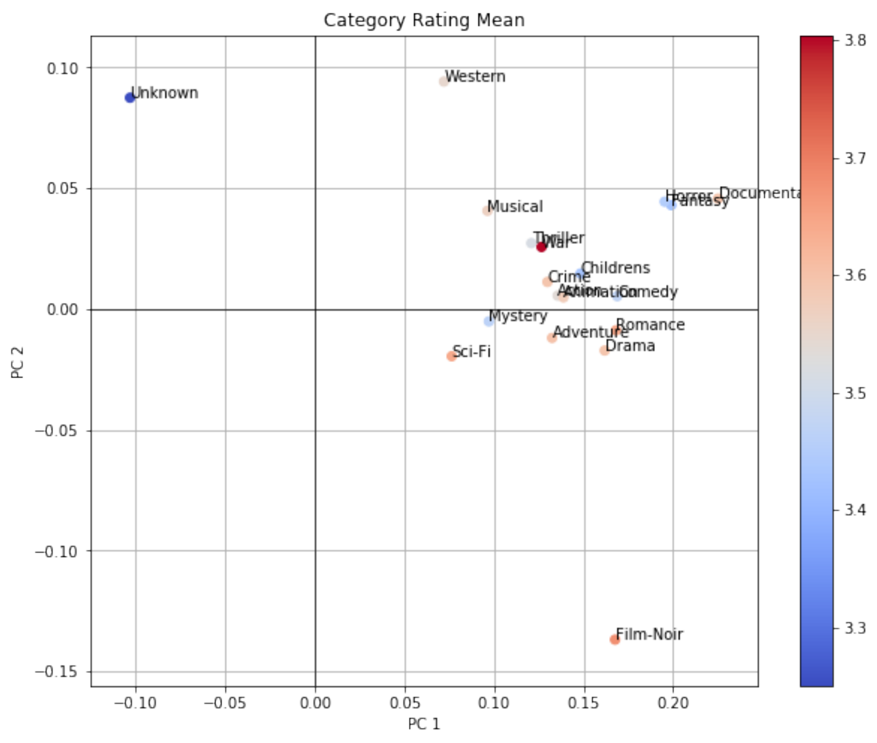
For method 2: Drama movies are near origin; Horror and Fantasy are more away from the origin.

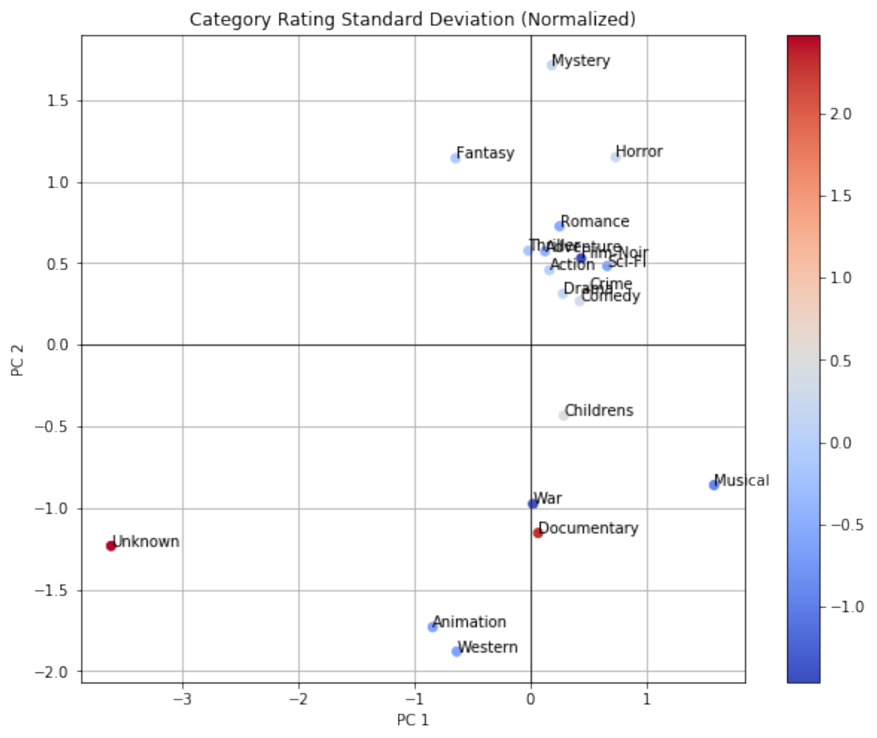
For method 3: Horror movies are mostly below $\mathrm{PC2} = 0$; Drama and Fantasy are more located $\mathrm{PC1} > 0$.

- Comparing the 3 methods:

In general, they produce very different visualizations for the same movies, but there seems to be some patterns lying underneath for each method, albeit differently. Still, I think the 2D representations are hard to spot many trends/patterns, at least to my eyes.

- Comparing genres: My piazza post @349 discusses the comparison of genres using mean and standard deviation of ratings of all movies in a genre. In short, by clustering movies into genres, we seem to be able to separate more popular genres from the less popular ones, especially using standard deviation of all ratings within a genre. We can train a classifier to determine whether a movie is in a popular genre, or a regressor on how likely it is for a movie to be popular. Please read my piazza post for more on this.


Category Rating Mean
PC 1
PC 2
Unknown
Western
Musical
Horror
Fantasy
Documentary
Thriller
War
Childrens
Crime
Action
Animation
Comedy
Mystery
Romance
Adventure
Drama
Sci-Fi
Film-Noir


Category Rating Standard Deviation (Normalized)
PC 1
PC 2
Mystery
Fantasy
Horror
Romance
Thriller
Adventure
Film-Noir
Action
Sci-Fi
Drama
Crime
Comedy
Childrens
Musical
War
Documentary
Unknown
Animation
Western

project2_viz

February 29, 2020

1 Project 2 - Visualization

Author: Liting Xiao

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams.update({'font.size': 12})
```

1.1 Data Pre-processing

[2]:
```
# read in the movie genre file
movies = pd.read_csv('data/movies.txt', delimiter="\t", header=None)

# add column names to be more descriptive and informative
movies.columns = ['Movie Id', 'Movie Title', 'Unknown', 'Action', 'Adventure',␣
 ↪'Animation',
                  'Childrens', 'Comedy', 'Crime', 'Documentary', 'Drama',␣
 ↪'Fantasy', 'Film-Noir',
                  'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi',␣
 ↪'Thriller', 'War', 'Western']

# read in train/test/data and add column names
data = pd.read_csv('data/data.txt', delimiter="\t", header=None)
Y_train = pd.read_csv('data/train.txt', delimiter="\t", header=None)
Y_test = pd.read_csv('data/test.txt', delimiter="\t", header=None)

data.columns = ['User Id', 'Movie Id', 'Rating']
Y_train.columns = ['User Id', 'Movie Id', 'Rating']
Y_test.columns = ['User Id', 'Movie Id', 'Rating']
```

Deal with duplicate movie IDs by replacing the second `Movie Id` with the first in the `data` dataframe:

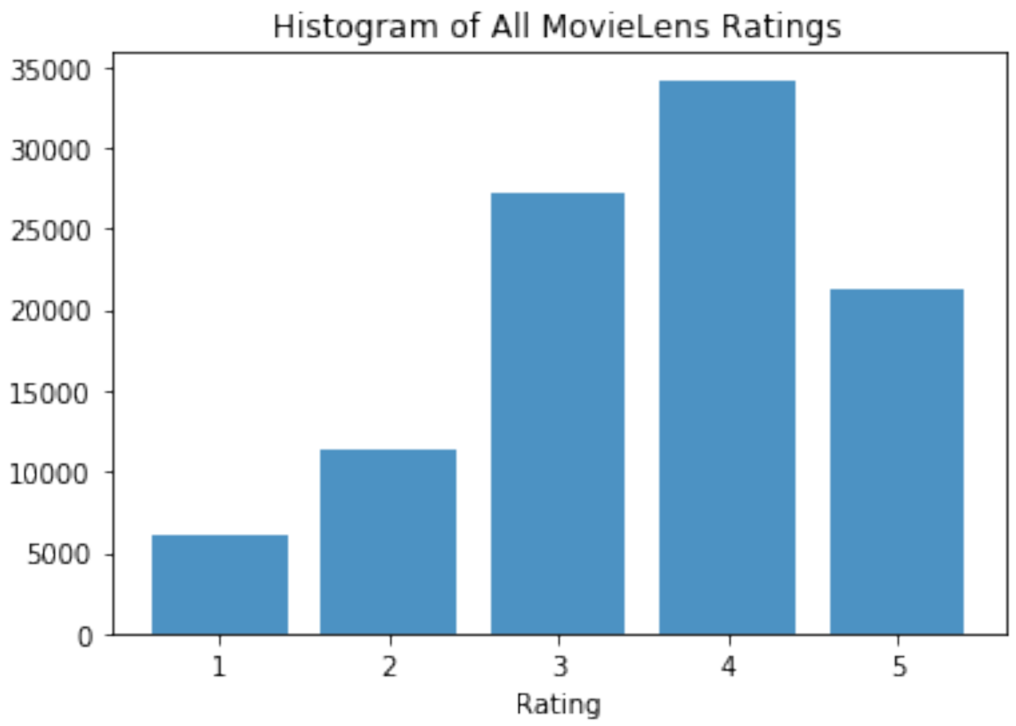
[3]:
```python
# group the duplicated movie ids by movie titles
dup = movies[movies.duplicated('Movie Title', keep=False)]
dup_id = dup.groupby('Movie Title')['Movie Id'].apply(np.array).
 ↪reset_index(name='Dup Id')['Dup Id']

# replace the duplicated 'Movie Ids' with the first
for did in dup_id:
    for i in range(1, len(did)):
        data['Movie Id'].replace({did[i]: did[0]}, inplace=True)
        Y_train['Movie Id'].replace({did[i]: did[0]}, inplace=True)
        Y_test['Movie Id'].replace({did[i]: did[0]}, inplace=True)
```

1.2 Basic Visualizations

1. All ratings in the MovieLens Dataset.

[4]:
```python
# define nicer bin edges to be [rating-0.4, rating+0.4] for better viz
edges = []
for i in range(1, 6):
    edges.append(i - 0.4)
    edges.append(i + 0.4)

print('The mean and std of ratings of all movies are {:.2f}, {:.3f}.'
      .format(data['Rating'].mean(), data['Rating'].std()))
plt.hist(data['Rating'], bins=edges, alpha=0.8)
plt.title('Histogram of All MovieLens Ratings')
plt.xlabel('Rating')
plt.savefig('figures/rating_all_hist.png')
```

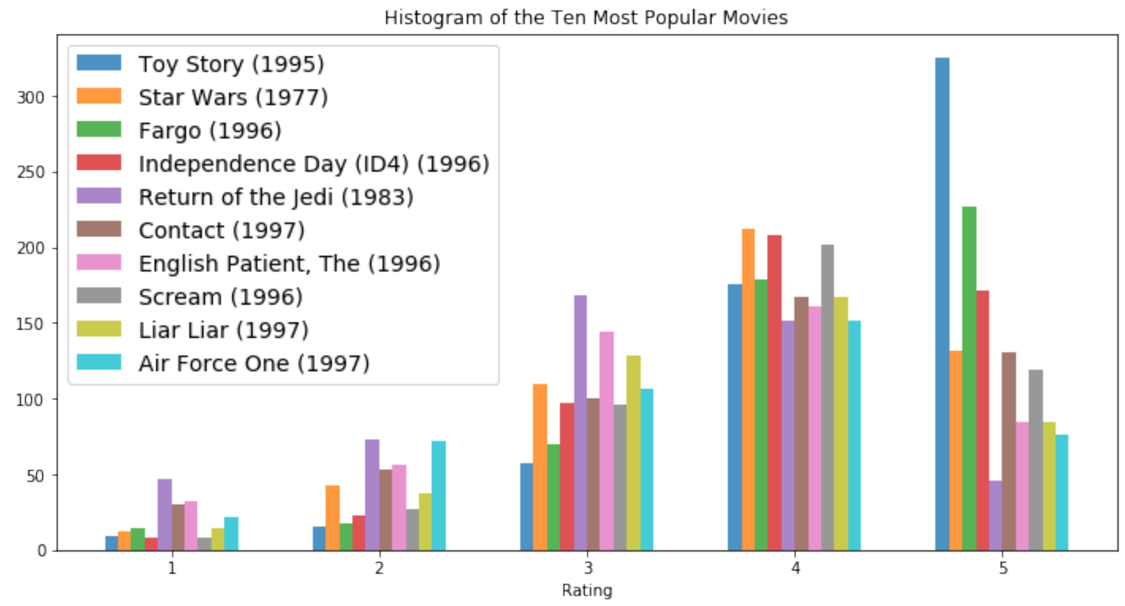
```
The mean and std of ratings of all movies are 3.53, 1.126.
```



2. All ratings of the 10 most popular movies (movies which have received the most ratings).

[5]:
```
# group ratings by movie ids
ratings_by_movie = data.groupby('Movie Id')['Rating']

# find the titles and rating arrays of the 10 most popular movies
most_pop_id = ratings_by_movie.count().nlargest(10).reset_index(name='Most␣
↪ratings')['Movie Id']
most_pop_title = movies['Movie Title'].loc[movies['Movie Id'].isin(most_pop_id)]
most_pop_rating = ratings_by_movie.apply(np.array)[most_pop_id].
↪reset_index(name='Rating')['Rating']

# plot hist
print('The mean and std of ratings of the 10 most popular movies are {:.2f}, {:.
↪3f}.'
      .format(np.hstack(most_pop_rating.values).mean(), np.
↪hstack(most_pop_rating.values).std()))
plt.figure(figsize=(12, 6))
plt.hist(most_pop_rating, bins=edges, label=most_pop_title, alpha=0.8)
plt.legend(fontsize=14)
plt.title('Histogram of the Ten Most Popular Movies')
plt.xlabel('Rating')
plt.savefig('figures/rating_pop_hist.png')
```

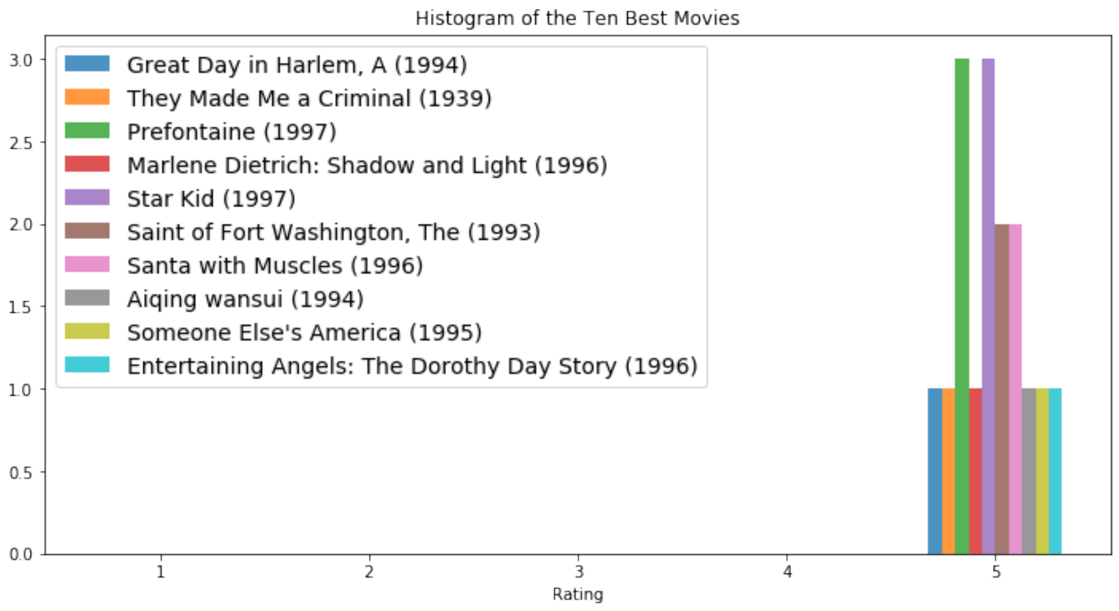
```
The mean and std of ratings of the 10 most popular movies are 3.77, 1.079.
```



3. All ratings of the 10 best movies (movies with the highest average ratings).

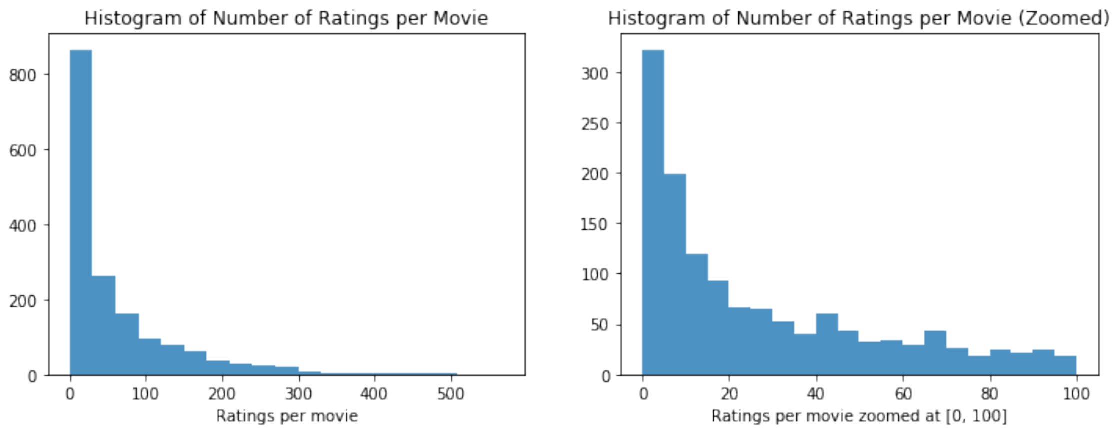
[6]:
```
# find the titles and rating arrays of the 10 best movies
best_mov_id = ratings_by_movie.mean().nlargest(10).reset_index(name='Highest␣
↪mean ratings')['Movie Id']
best_mov_title = movies['Movie Title'].loc[movies['Movie Id'].isin(best_mov_id)]
best_mov_rating = ratings_by_movie.apply(np.array)[best_mov_id].
↪reset_index(name='Rating')['Rating']

# plot hist
print('The mean and std of ratings of the 10 best movies are {:.2f}, {:.3f}.'
      .format(np.hstack(best_mov_rating.values).mean(), np.
↪hstack(best_mov_rating.values).std()))
plt.figure(figsize=(12, 6))
plt.hist(best_mov_rating, bins=edges, label=best_mov_title, alpha=0.8)
plt.legend(fontsize=14)
plt.title('Histogram of the Ten Best Movies')
plt.xlabel('Rating')
plt.savefig('figures/rating_best_hist.png')
```

```
The mean and std of ratings of the 10 best movies are 5.00, 0.000.
```



Seems like we have a spread of `number of ratings per movie`. Let's histogram that. And looks like we have a skewed dataset towards lower number of ratings. Let's zoom in to [0, 100] ratings per movie range.

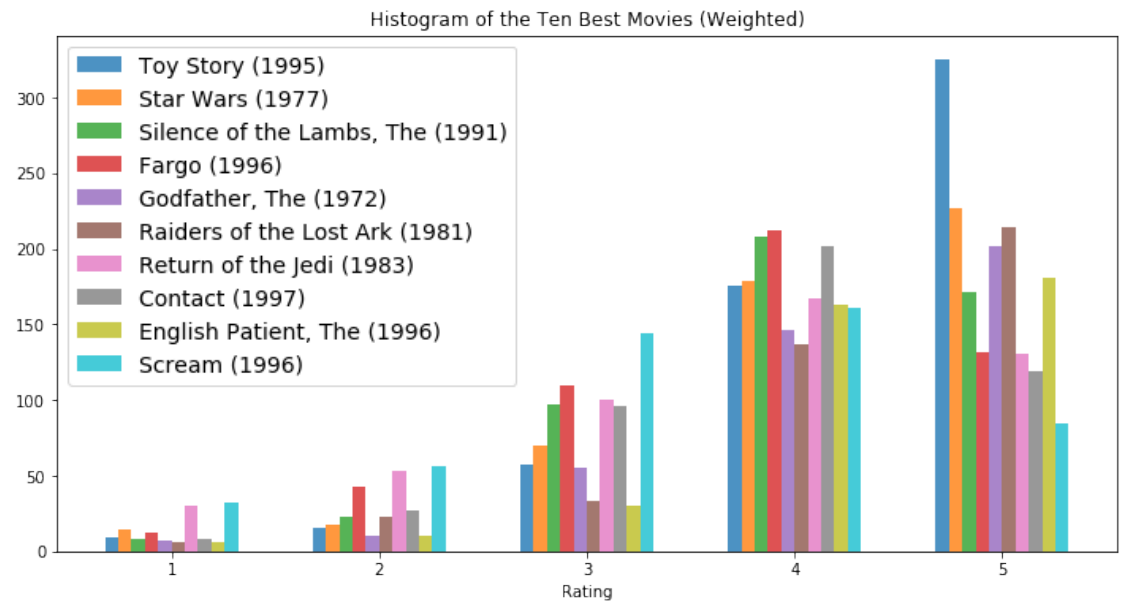
[7]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,4))
ax1.hist(ratings_by_movie.count(), bins=np.arange(0, 600, 30), alpha=0.8)
ax1.set_xlabel('Ratings per movie')
ax1.set_title('Histogram of Number of Ratings per Movie')
ax2.hist(ratings_by_movie.count(), bins=np.arange(0, 105, 5), alpha=0.8)
ax2.set_xlabel('Ratings per movie zoomed at [0, 100]')
ax2.set_title('Histogram of Number of Ratings per Movie (Zoomed)')
plt.savefig('figures/ratings_per_movie.png')
```



Now let's use weighted mean and find the actual 10 best movies with highest weighted average ratings.

```
[8]: weighted_avg = ratings_by_movie.mean() * ratings_by_movie.count() /␣
      ↪ratings_by_movie.count().sum()
     best_mov_id2 = weighted_avg.nlargest(10).reset_index(name='Highest weighted␣
      ↪mean ratings')['Movie Id']
     best_mov_title2 = movies['Movie Title'].loc[movies['Movie Id'].
      ↪isin(best_mov_id2)]
     best_mov_rating2 = ratings_by_movie.apply(np.array)[best_mov_id2].
      ↪reset_index(name='Rating')['Rating']

     # plot hist
     print('The weighted mean and std of ratings of the 10 best movies are {:.2f}, {:.
      ↪3f}.'
           .format(np.hstack(best_mov_rating2.values).mean(), np.
      ↪hstack(best_mov_rating2.values).std()))
     plt.figure(figsize=(12, 6))
     plt.hist(best_mov_rating2, bins=edges, label=best_mov_title2, alpha=0.8)
     plt.legend(fontsize=14)
     plt.title('Histogram of the Ten Best Movies (Weighted)')
     plt.xlabel('Rating')
     plt.savefig('figures/rating_best_hist2.png')
```

```
The weighted mean and std of ratings of the 10 best movies are 4.01, 1.015.
```



We have some overlaps between the 10 most popular and the 10 best weighted average. Let's find them.

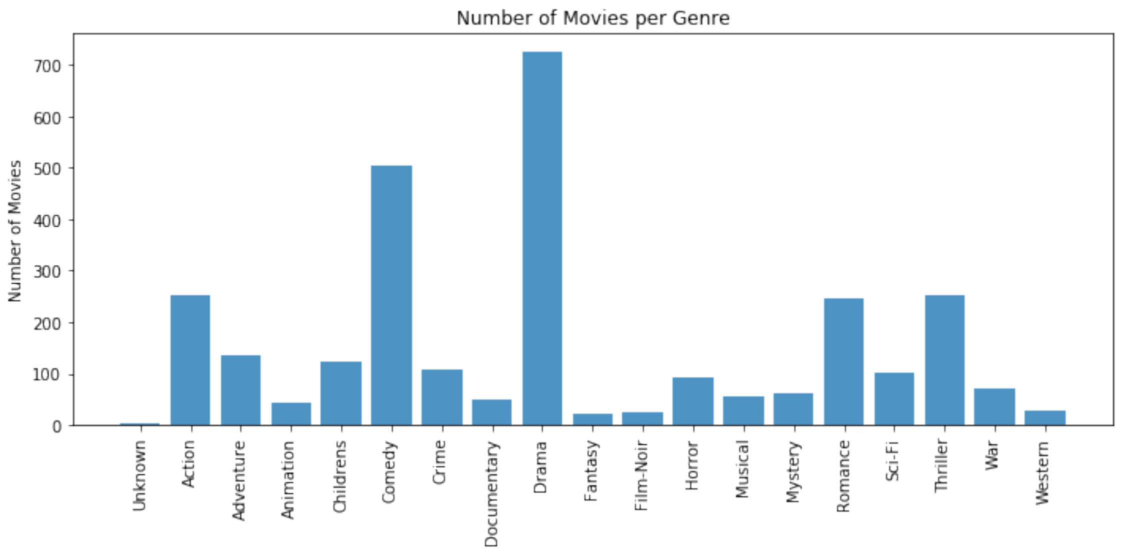
```
[9]: overlap = []
     for pop in most_pop_title:
         for best in best_mov_title2:
             if pop == best:
                 overlap.append(pop)
                 break
     print('The {} movies in both 10 most popular and 10 best are: \n{}'.
      ↪format(len(overlap),overlap))
```

```
The 7 movies in both 10 most popular and 10 best are:
['Toy Story (1995)', 'Star Wars (1977)', 'Fargo (1996)', 'Return of the Jedi
(1983)', 'Contact (1997)', 'English Patient, The (1996)', 'Scream (1996)']
```

4. All ratings of movies from three genres of your choice (create three separate visualizations).

Let's visualize the number of movies per genre.

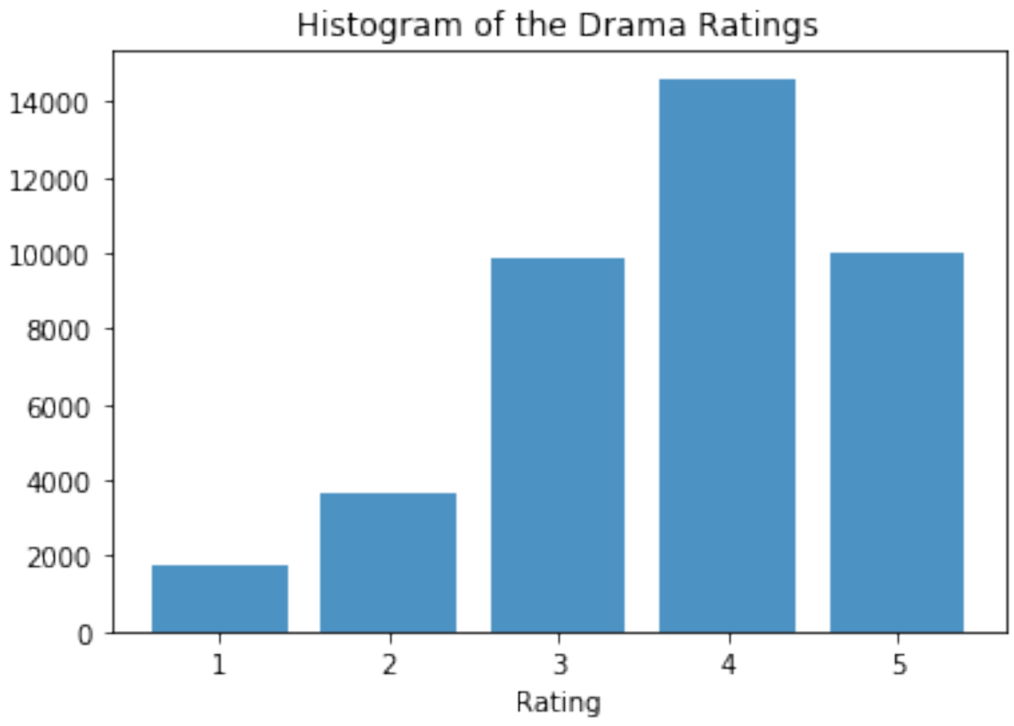
```
[10]: genre_sum = movies.drop(['Movie Id', 'Movie Title'], axis=1).sum()
      plt.figure(figsize=(10, 5))
      plt.bar(movies.columns[2:], genre_sum, align='center', alpha=0.8)
      plt.xticks(rotation=90)
      plt.title('Number of Movies per Genre')
      plt.ylabel('Number of Movies')
      plt.tight_layout()
      plt.savefig('figures/genre_num_movies.png')
```



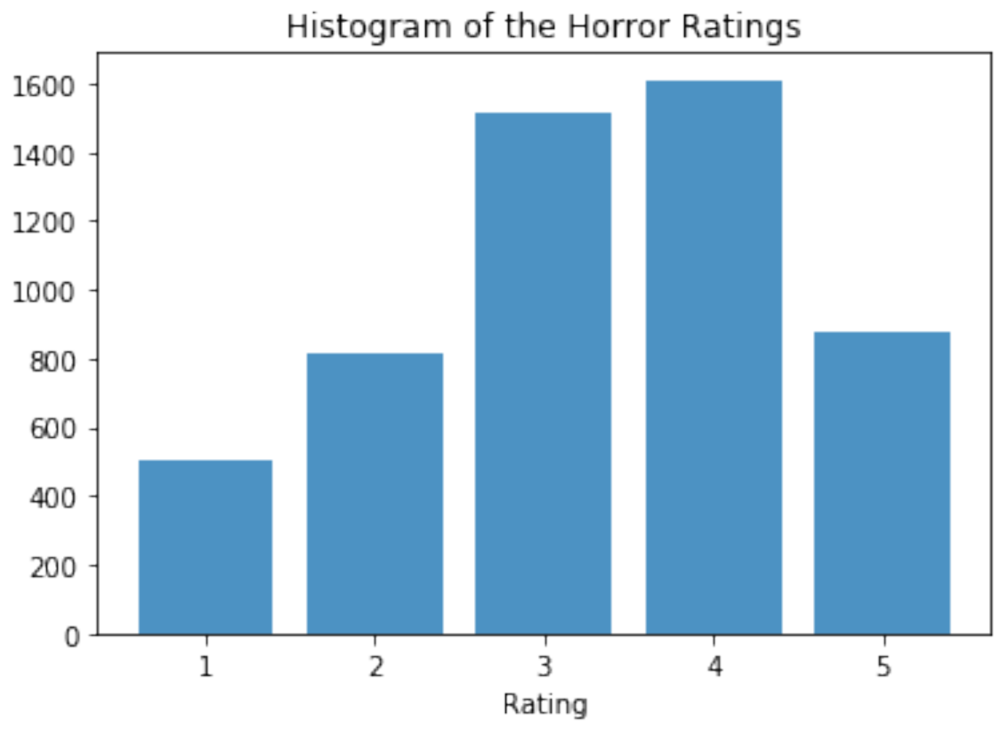
Now let's pick 3 genres with the most, medium, few labeled movies: [Drama, Horror, Fantasy].

[11]:
```python
for genre in ['Drama', 'Horror', 'Fantasy']:
    picked_id = movies['Movie Id'][movies[genre] == 1]
    picked_rating = data['Rating'].loc[data['Movie Id'].isin(picked_id)]

    print('The mean and std of ratings in the {} genre are {:.2f}, {:.3f}.'
          .format(genre, picked_rating.mean(), picked_rating.std()))
    plt.hist(picked_rating, bins=edges, alpha=0.8)
    plt.title('Histogram of the {} Ratings'.format(genre))
    plt.xlabel('Rating')
    plt.savefig('figures/rating_{}_hist.png'.format(genre))
    plt.show()
```

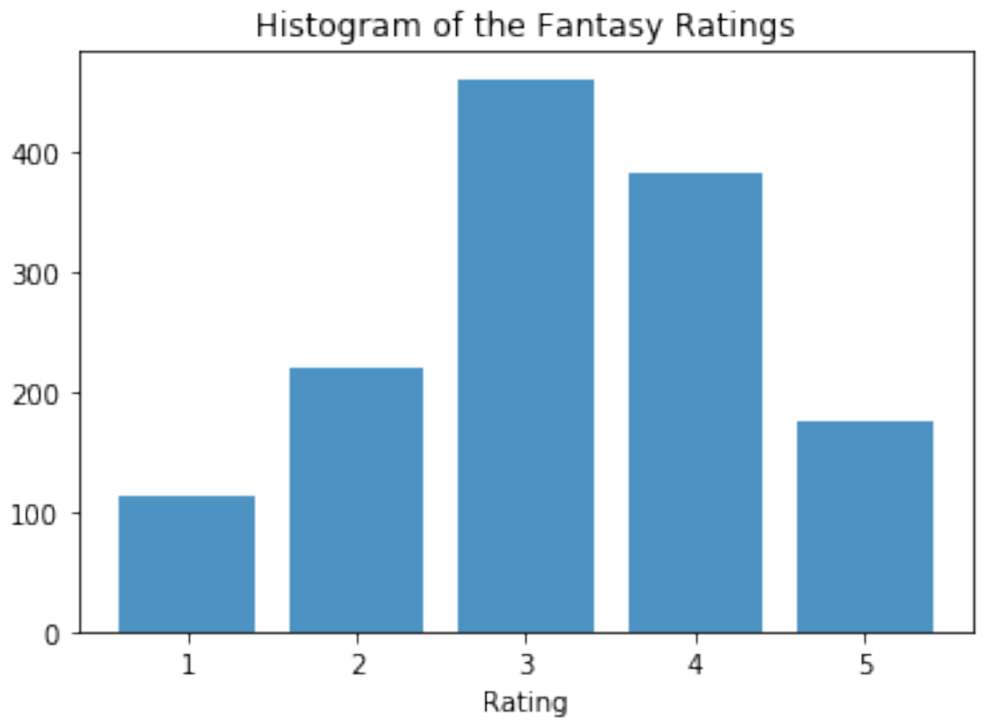
```
The mean and std of ratings in the Drama genre are 3.69, 1.079.
```



```
The mean and std of ratings in the Horror genre are 3.29, 1.187.
```



```
The mean and std of ratings in the Fantasy genre are 3.22, 1.119.
```



1.3 Matrix Factoriazation

1. Re-use code from HW5. Not including biases.

Code from HW5 was modified to include bias terms for use in Method 2.

[51]:
```
def grad_U(Ui, Yij, Vj, reg, eta, mu=0, ai=0, bj=0):
    """
    Takes as input Ui (the ith row of U), a training point Yij, the column
    vector Vj (jth column of V^T), reg (the regularization parameter lambda),
    and eta (the learning rate).

    Returns the gradient of the regularized loss function with
    respect to Ui multiplied by eta.
    """
    dUi = reg * Ui - Vj * ((Yij - mu) - (np.dot(Ui, Vj) + ai + bj))
    return eta * dUi

def grad_V(Vj, Yij, Ui, reg, eta, mu=0, ai=0, bj=0):
    """
   Takes as input the column vector Vj (jth column of V^T), a training point Yij,
    Ui (the ith row of U), reg (the regularization parameter lambda),
    and eta (the learning rate).

    Returns the gradient of the regularized loss function with
    respect to Vj multiplied by eta.
    """
    dVj = reg * Vj - Ui * ((Yij - mu) - (np.dot(Ui, Vj) + ai + bj))
    return eta * dVj

def grad_A(Ui, Vj, Yij, eta, mu=0, ai=0, bj=0):
    dAi = reg * ai - ((Yij - mu) - (np.dot(Ui, Vj) + ai + bj))
    return eta * dAi

def grad_B(Ui, Vj, Yij, eta, mu=0, ai=0, bj=0):
    dBj = reg * bj - ((Yij - mu) - (np.dot(Ui, Vj) + ai + bj))
    return eta * dBj

def get_err(U, V, Y, reg=0.0, mu=0, A=None, B=None):
    """
    Takes as input a matrix Y of triples (i, j, Y_ij) where i is the index of a␣
 ↪user,
    j is the index of a movie, and Y_ij is user i's rating of movie j and
    user/movie matrices U and V.
    Returns the mean regularized squared-error of predictions made by
    estimating Y_{ij} as the dot product of the ith row of U and the jth column␣
 ↪of V^T.
    """
    reg_term = reg * (np.linalg.norm(U)**2 + np.linalg.norm(V)**2)
    if A is not None:
        reg_term += reg * np.linalg.norm(A)**2
    if B is not None:
        reg_term += reg * np.linalg.norm(B)**2

    mse_term = 0
    for sample in Y:
        i, j, rating = sample
        # data are 1-indexed
        ab_term = 0
        if A is not None:
            ab_term += A[i-1]
        if B is not None:
            ab_term += B[j-1]
       mse_term += ((rating - mu) - (np.dot(U[i-1,:], V.T[:,j-1]) + ab_term))**2
    return (reg_term + mse_term) / 2. / len(Y)

def train_model(M, N, K, eta, reg, Y, eps=0.0001, max_epochs=300, bias=False):
    """
    Given a training data matrix Y containing rows (i, j, Y_ij)
    where Y_ij is user i's rating on movie j, learns an
    M x K matrix U and N x K matrix V such that rating Y_ij is approximated
    by (UV^T)_ij.

    Uses a learning rate of <eta> and regularization of <reg>. Stops after
    <max_epochs> epochs, or once the magnitude of the decrease in regularized
    MSE between epochs is smaller than a fraction <eps> of the decrease in
    MSE after the first epoch.

    Returns a tuple (U, V, err) consisting of U, V, and the unregularized MSE
    of the model.
    """
    # matrices init
    U = np.random.uniform(-0.5, 0.5, size=(M, K))
    V = np.random.uniform(-0.5, 0.5, size=(N, K))
    if bias:
        A = np.random.uniform(1, 5, size=(M,))
        B = np.random.uniform(1, 5, size=(N,))
        mu = np.mean(Y[:,2])
        last_err = get_err(U, V, Y, reg=reg, mu=mu, A=A, B=B)
    else:
        last_err = get_err(U, V, Y, reg=reg)
    # training loop
    for epoch in range(max_epochs):
        # shuffle the dataset
        shuffle_idx = np.random.permutation(len(Y))
        Y_shuf = Y[shuffle_idx]

        # training update
        for sample in Y_shuf:
            i, j, rating = sample
            # the data are 1-indexed
            i -= 1
            j -= 1

            # calc gradients
            if bias:
                grad_a = grad_A(U[i,:], V.T[:,j], rating, eta, mu=mu, ai=A[i],␣
↪bj=B[j])
                grad_b = grad_B(U[i,:], V.T[:,j], rating, eta, mu=mu, ai=A[i],␣
↪bj=B[j])
                grad_ui = grad_U(U[i,:], rating, V.T[:,j], reg, eta, mu=mu,␣
↪ai=A[i], bj=B[j])
                grad_vj = grad_V(V.T[:,j], rating, U[i,:], reg, eta, mu=mu,␣
↪ai=A[i], bj=B[j])
            else:
                grad_ui = grad_U(U[i,:], rating, V.T[:,j], reg, eta)
                grad_vj = grad_V(V.T[:,j], rating, U[i,:], reg, eta)

            # SGD
            if bias:
                A[i] -= grad_a
                B[j] -= grad_b
            U[i,:] -= grad_ui
            V.T[:,j] -= grad_vj

        # obtain current MSE
        if bias:
            err = get_err(U, V, Y_shuf, reg=reg, mu=mu, A=A, B=B)
        else:
            err = get_err(U, V, Y_shuf, reg=reg)

        # check for convergence
        if (last_err - err) / last_err < eps: break

        # update last_err
        last_err = err
    if bias:
        return (U, V, A, B, err)
    return (U, V, err)
```

[13]:
```
# define some hyperparameters
M = max(max(Y_train.values[:,0]), max(Y_test.values[:,0])).astype(int) # users
N = max(max(Y_train.values[:,1]), max(Y_test.values[:,1])).astype(int) # movies
print("Factorizing with ", M, " users, ", N, " movies.")
K = 20
```

```
Factorizing with  943  users,  1682  movies.
```

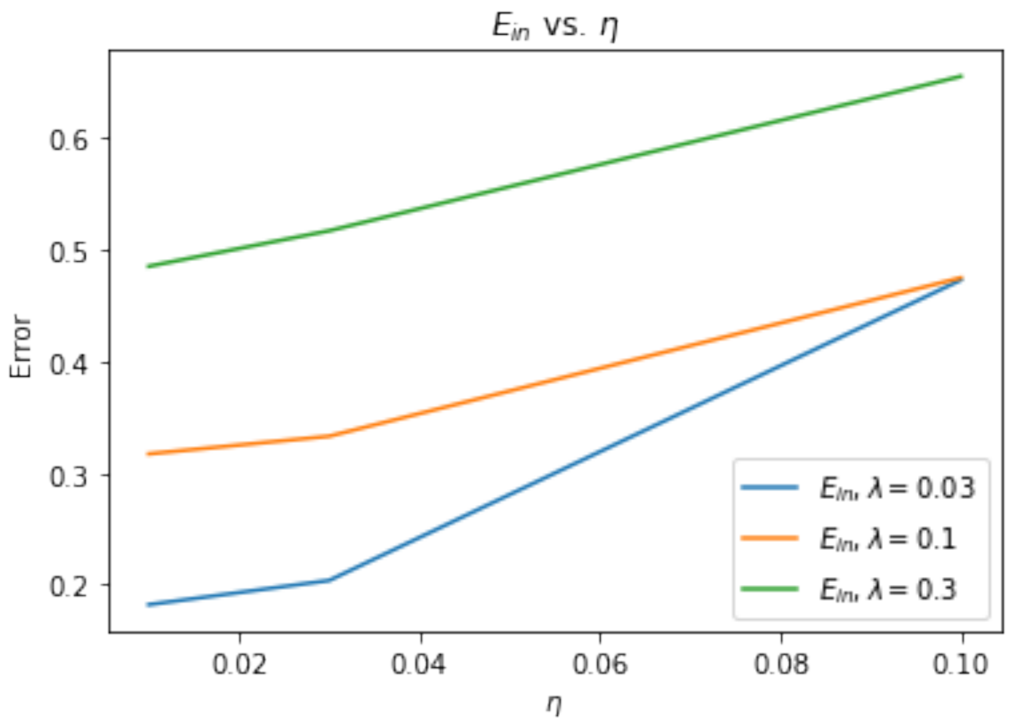
Now let's find good regularization parameter and learning rate.

[18]:
```
def ein_eout_thru_hyperparams(regs, etas, Y_train, Y_test, plot_name,␣
↪bias=False):
    E_ins = []
    E_outs = []

    # Compute Ein and Eout
    for reg in regs:
        E_ins_for_lambda = []
        E_outs_for_lambda = []

        for eta in etas:
            print("Training model with M = %s, N = %s, K = %s, eta = %s, reg =␣
↪%s"%(M, N, K, eta, reg))
            if bias:
                U, V, A, B, e_in = train_model(M, N, K, eta, reg, Y_train.
↪values, bias=True)
                eout = get_err(U, V, Y_test.values, mu=np.mean(Y_test.values),␣
↪A=A, B=B)
            else:
                U, V, e_in = train_model(M, N, K, eta, reg, Y_train.values)
                eout = get_err(U, V, Y_test.values)

            E_ins_for_lambda.append(e_in)
            E_outs_for_lambda.append(eout)

        E_ins.append(E_ins_for_lambda)
        E_outs.append(E_outs_for_lambda)

    E_ins = np.array(E_ins)
    E_outs = np.array(E_outs)

    # Plot values of E_in across eta for each value of lambda
    for i in range(len(regs)):
        plt.plot(etas, E_ins[i], label='$E_{in}, \lambda=$'+str(regs[i]))
    plt.title('$E_{in}$ vs. $\eta$')
    plt.xlabel('$\eta$')
    plt.ylabel('Error')
    plt.legend()
    plt.savefig('figures/{}_ein.png'.format(plot_name))
    plt.show()
    plt.clf()

    # Plot values of E_out across eta for each value of lambda
    for i in range(len(regs)):
        plt.plot(etas, E_outs[i], label='$E_{out}, \lambda=$'+str(regs[i]))
    plt.title('$E_{out}$ vs. $\eta$')
    plt.xlabel('$\eta$')
    plt.ylabel('Error')
    plt.legend()
    plt.savefig('figures/{}_eout.png'.format(plot_name))
    plt.show()

    return (E_ins, E_outs)
```

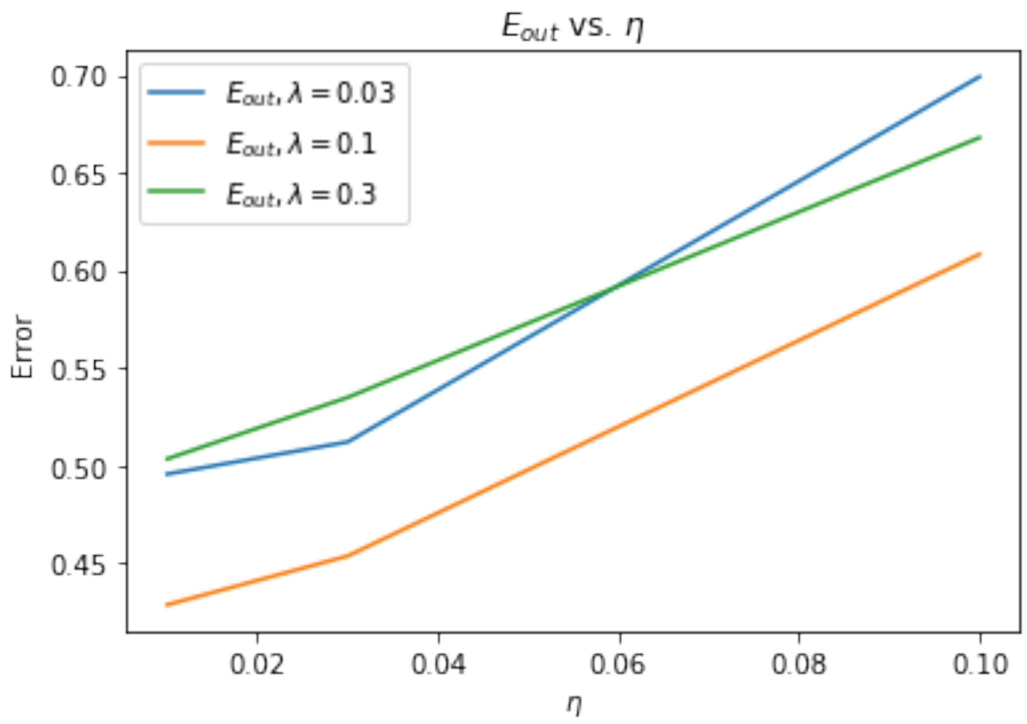
Recall that in HW5, the best $\lambda = 0.1$, and $\eta = 0.03$ was used. And that works well already. So, let's try values around that.

[15]:
```
regs = [0.03, 0.1, 0.3]
etas = [0.01, 0.03, 0.1]

E_ins, E_outs = ein_eout_thru_hyperparams(regs, etas, Y_train, Y_test, 'sgd')
```

```
Training model with M = 943, N = 1682, K = 20, eta = 0.01, reg = 0.03
Training model with M = 943, N = 1682, K = 20, eta = 0.03, reg = 0.03
Training model with M = 943, N = 1682, K = 20, eta = 0.1, reg = 0.03
Training model with M = 943, N = 1682, K = 20, eta = 0.01, reg = 0.1
Training model with M = 943, N = 1682, K = 20, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, K = 20, eta = 0.1, reg = 0.1
Training model with M = 943, N = 1682, K = 20, eta = 0.01, reg = 0.3
Training model with M = 943, N = 1682, K = 20, eta = 0.03, reg = 0.3
Training model with M = 943, N = 1682, K = 20, eta = 0.1, reg = 0.3
```



E_{in} vs. η
Error
0.2
0.3
0.4
0.5
0.6
0.02
0.04
0.06
0.08
0.10
η
$E_{in}, \lambda = 0.03$
$E_{in}, \lambda = 0.1$
$E_{in}, \lambda = 0.3$



E_{out} vs. η
Error
0.45
0.50
0.55
0.60
0.65
0.70
0.02
0.04
0.06
0.08
0.10
η
$E_{out}, \lambda = 0.03$
$E_{out}, \lambda = 0.1$
$E_{out}, \lambda = 0.3$

From the above plots, $\eta = 0.01$, reg $= 0.1$ gives the minimum E_out. No overfitting.

So, let's use them to train on the whole dataset and get the final U and V for method 1.

```
[16]: E_ins_method = []
E_outs_method = []
matrices_and_err = []

index = np.unravel_index(np.argmin(E_outs), E_outs.shape)
reg = regs[index[0]]
eta = etas[index[1]]
print("Training model with M = %s, N = %s, K = %s, eta = %s, reg = %s"%(M, N, K,␣
 ↪eta, reg))

# Append E_in and E_out for the first method with reg=0.1 and eta=0.01
E_ins_method.append(E_ins[index])
E_outs_method.append(E_outs[index])

# Append the matrices and final MSE for method 1
matrices_and_err.append(train_model(M, N, K, eta, reg, data.values))
```

```
Training model with M = 943, N = 1682, K = 20, eta = 0.01, reg = 0.1
```

2. Incorporate bias terms a for each user and b for each movie.

Since this method is similar to method1, let's use $\eta = 0.01$, $\lambda = 0.1$.

```
[75]: # Find train / test error
K = 20
eta = 0.01
reg = 0.1
U, V, A, B, e_in = train_model(M, N, K, eta, reg, Y_train.values, bias=True)

# Append E_in and E_out for the first method with reg=0.1 and eta=0.01
E_ins_method.append(e_in)
E_outs_method.append(eout)
```

```
0.25497589241895147 0.427409019156015
```

```
[76]: # Final Model
print("Training model with M = %s, N = %s, K = %s, eta = %s, reg = %s"%(M, N, K,␣
 ↪eta, reg))
matrices_and_err.append(train_model(M, N, K, eta, reg, data.values, bias=True))
```

```
Training model with M = 943, N = 1682, K = 20, eta = 0.01, reg = 0.1
```

3. Off-the-shelf implementation using Surprise SVD.

Re-index to deal with movies with no reviews in `Y_train`. Also re-index `Y_test` accordingly.

```
[23]: # find the missing movie IDs in Y_train
missing = movies['Movie Id'][~movies['Movie Id'].isin(Y_train['Movie Id'])].
 ↪values

# re-index
val = 0
for m in sorted(missing):
    Y_train['Movie Id'][Y_train['Movie Id'] > (m-val)] -= 1
    Y_test['Movie Id'][Y_test['Movie Id'] > (m-val)] -= 1
    val += 1
```

Now, implement the matrix factorization.

```
[24]: from surprise import SVD, Dataset, Reader, accuracy
```

```
[63]: # Load datasets from pandas.dataframe to Surprise
reader = Reader(sep='\t')

# need to use unchanged original data
data = pd.read_csv('data/data.txt', delimiter="\t", header=None)
data.columns = ['User Id', 'Movie Id', 'Rating']

train3 = Dataset.load_from_df(Y_train, reader=reader)
test3 = Dataset.load_from_df(Y_test, reader=reader)
data3 = Dataset.load_from_df(data, reader=reader)

train3 = train3.build_full_trainset()
test3 = test3.build_full_trainset().build_testset()
data3 = data3.build_full_trainset()
```

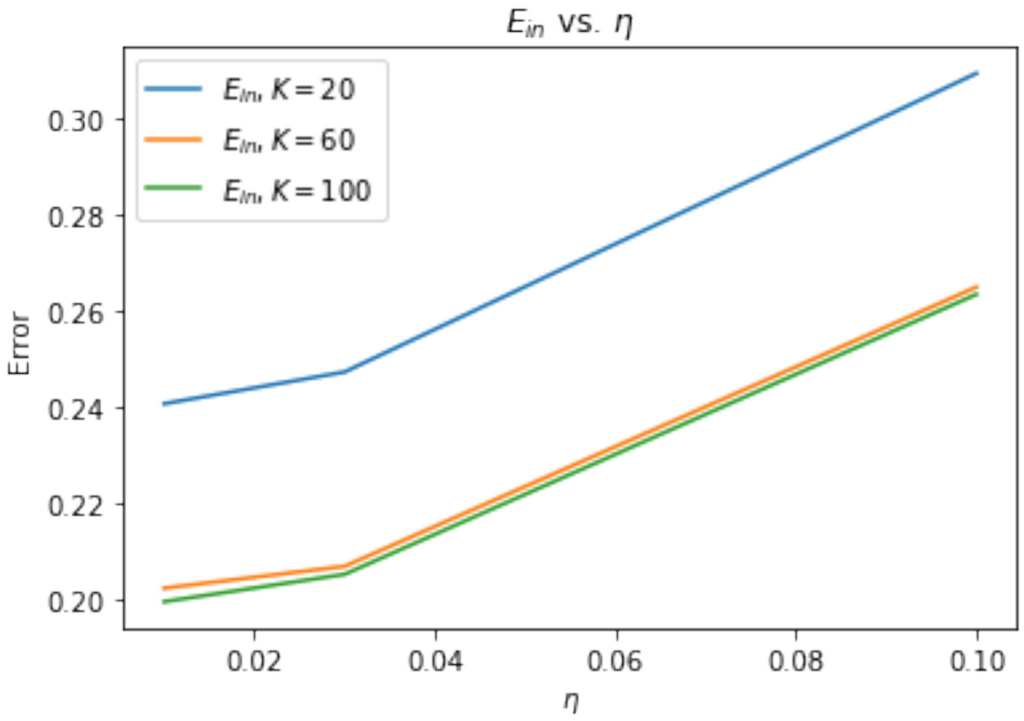
Train and test to see performance by looking at MSE.

To choose ~optimal hyperparams, like before, loop through different params. Not gonna loop through reg in the interest of time, since it doesn't matter much. Looping through K's and η's.

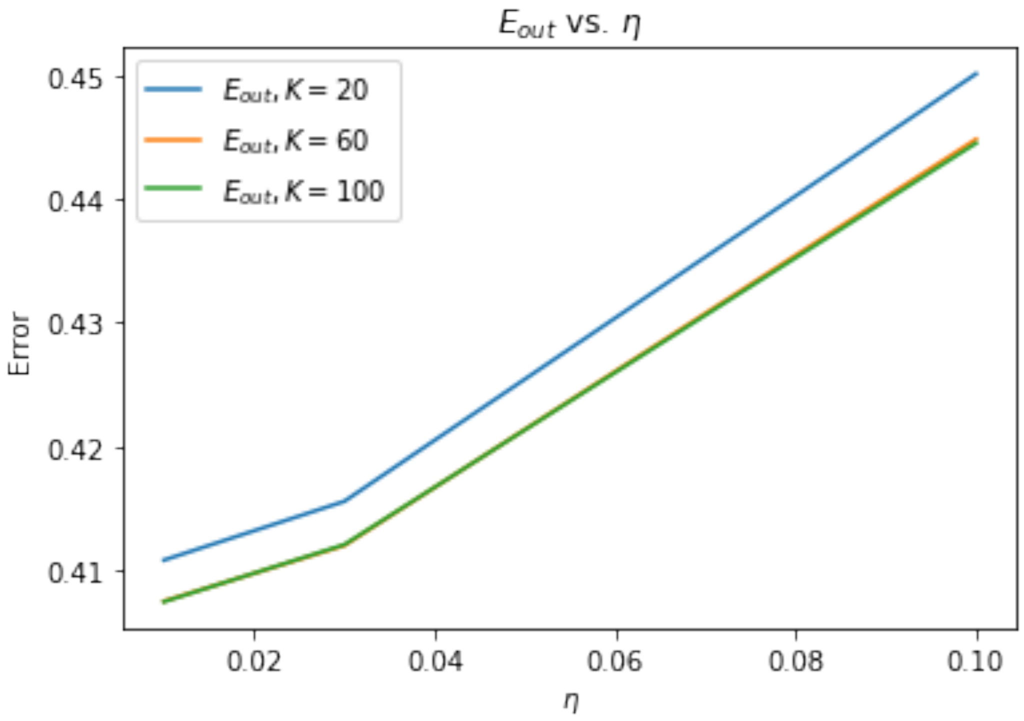
```
[26]: etas = [0.01, 0.03, 0.1]
Ks = [20, 60, 100]
E_ins = []
E_outs = []

# Compute Ein and Eout
for K in Ks:
    E_ins_for_k = []
    E_outs_for_k = []

    for eta in etas:
        print("Training model with M = %s, N = %s, K = %s, eta = %s, reg =␣
 ↪%s"%(M, N, K, eta, reg))
```

```
        algo = SVD(n_epochs=300, n_factors=K, lr_all=eta, reg_all=reg)
        preds = algo.fit(train3).test(test3)

        e_in = accuracy.mse(algo.test(train3.build_testset()), verbose=False) /␣
 ↪2.
        eout = accuracy.mse(preds, verbose=False) / 2.
        E_ins_for_k.append(e_in)
        E_outs_for_k.append(eout)

    E_ins.append(E_ins_for_k)
    E_outs.append(E_outs_for_k)

E_ins = np.array(E_ins)
E_outs = np.array(E_outs)

# Plot values of E_in across eta for each value of lambda
for i in range(len(Ks)):
    plt.plot(etas, E_ins[i], label='$E_{in}, K=$'+str(Ks[i]))
plt.title('$E_{in}$ vs. $\eta$')
plt.xlabel('$\eta$')
plt.ylabel('Error')
plt.legend()
plt.savefig('figures/surprise_ein.png')
plt.show()
plt.clf()

# Plot values of E_out across eta for each value of lambda
for i in range(len(Ks)):
    plt.plot(etas, E_outs[i], label='$E_{out}, K=$'+str(Ks[i]))
plt.title('$E_{out}$ vs. $\eta$')
plt.xlabel('$\eta$')
plt.ylabel('Error')
plt.legend()
plt.savefig('figures/surprise_eout.png')
```

```
Training model with M = 943, N = 1682, K = 20, eta = 0.01, reg = 0.1
Training model with M = 943, N = 1682, K = 20, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, K = 20, eta = 0.1, reg = 0.1
Training model with M = 943, N = 1682, K = 60, eta = 0.01, reg = 0.1
Training model with M = 943, N = 1682, K = 60, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, K = 60, eta = 0.1, reg = 0.1
Training model with M = 943, N = 1682, K = 100, eta = 0.01, reg = 0.1
Training model with M = 943, N = 1682, K = 100, eta = 0.03, reg = 0.1
Training model with M = 943, N = 1682, K = 100, eta = 0.1, reg = 0.1
```



E_{in} vs. η
$E_{in}, K=20$
$E_{in}, K=60$
$E_{in}, K=100$
Error
0.20
0.22
0.24
0.26
0.28
0.30
0.02
0.04
0.06
0.08
0.10
η



E_{out} vs. η
$E_{out}, K=20$
$E_{out}, K=60$
$E_{out}, K=100$
Error
0.41
0.42
0.43
0.44
0.45
0.02
0.04
0.06
0.08
0.10
η

```
[27]: # Append E_in and E_out for the first method with K=100 and eta=0.01
      index = np.unravel_index(np.argmin(E_outs), E_outs.shape)
      K3 = Ks[index[0]]
      eta = etas[index[1]]

      E_ins_method.append(E_ins[index])
      E_outs_method.append(E_outs[index])
```

```
[66]: # finally, train on the whole set to get final matrices
      print("Training model with M = %s, N = %s, K = %s, eta = %s, reg = %s"%(M, N,␣
       ↪K3, eta, reg))
      final_algo = SVD(n_epochs=300, n_factors=K3, lr_all=eta, reg_all=reg)
      final_algo.fit(data3)
      matrices_and_err.append([final_algo.pu, final_algo.qi, final_algo.bu,␣
       ↪final_algo.bi,
                               accuracy.mse(final_algo.test(data3.build_testset()),␣
       ↪verbose=False)])
```

```
Training model with M = 943, N = 1682, K = 100, eta = 0.01, reg = 0.1
```

1.3.1 Compare the performance of the 3 methods

```
[79]: method = 1
      for ein, eout in zip(E_ins_method, E_outs_method):
          print('Method {}: E_in = {:.4f}, E_out = {:.4f}.'.format(method, ein, eout))
          method += 1
```

```
Method 1: E_in = 0.3172, E_out = 0.4287.
Method 2: E_in = 0.2550, E_out = 0.4274.
Method 3: E_in = 0.1993, E_out = 0.4075.
```

Thus, method 3 performs the best.

1.4 Matrix Factoriazation Visualizations

Apply SVD to V.

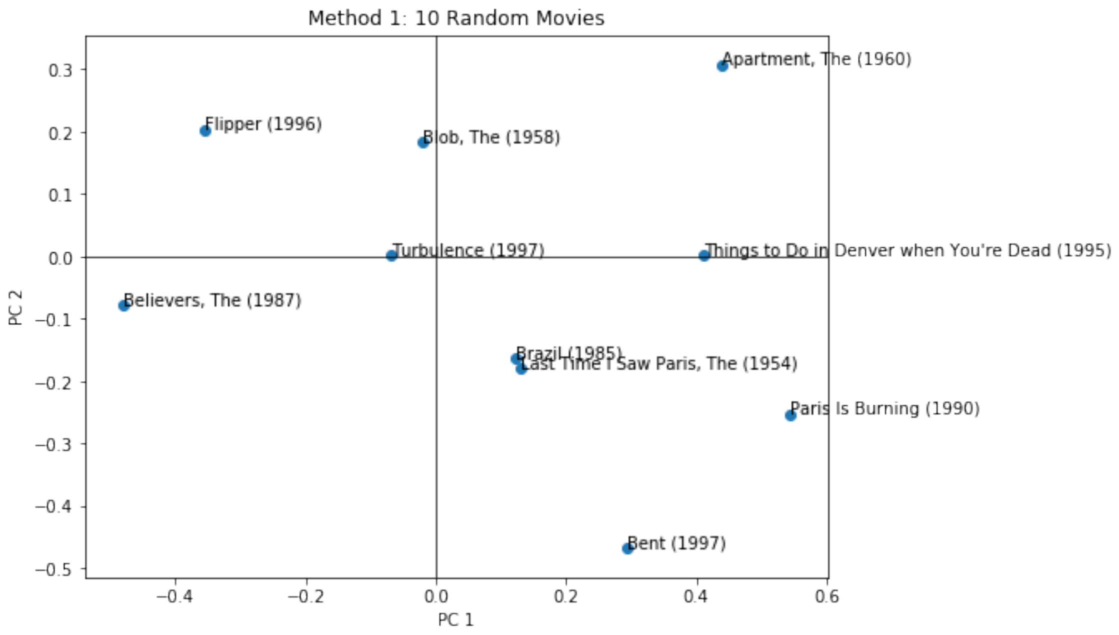
```
[159]: A_svd = np.array([np.linalg.svd(matrices_and_err[i][1].T) for i in range(3)])[:
        ↪,0]
       viz = [np.matmul(A_svd[i][:,1:3].T, matrices_and_err[i][1].T) for i in range(3)]
```

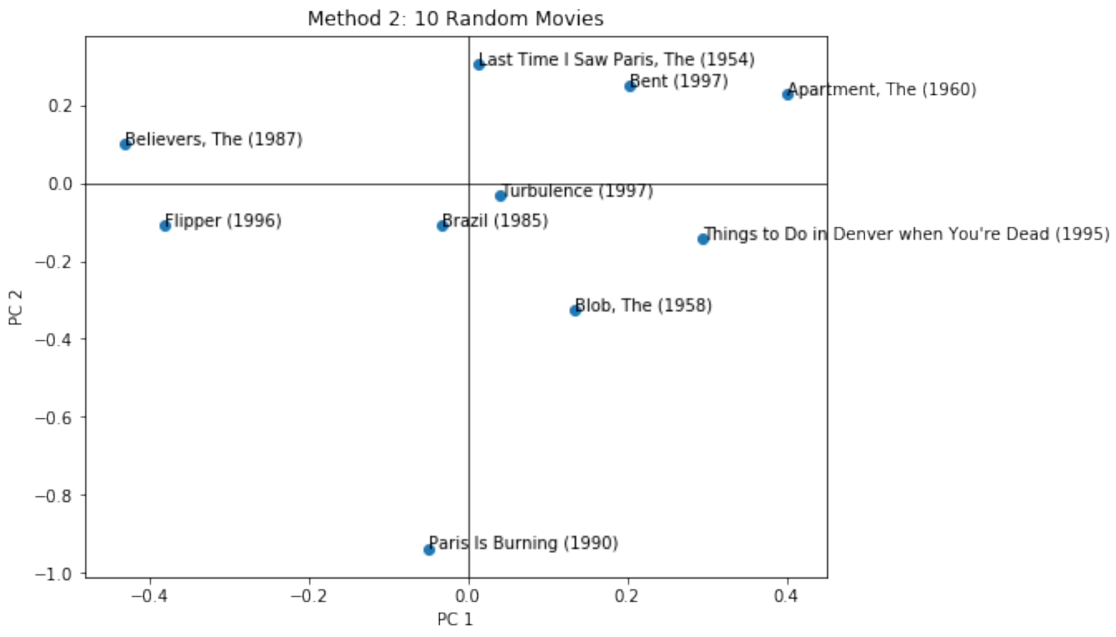
```
[166]: def movie_viz_2d(vizmat, picked_id, picked_title, plot_title, plot_name):
           plt.figure(figsize=(8, 6))
           plt.axhline(0, color='k', linewidth=0.7)
           plt.axvline(0, color='k', linewidth=0.7)
           plt.xlabel('PC 1')
```

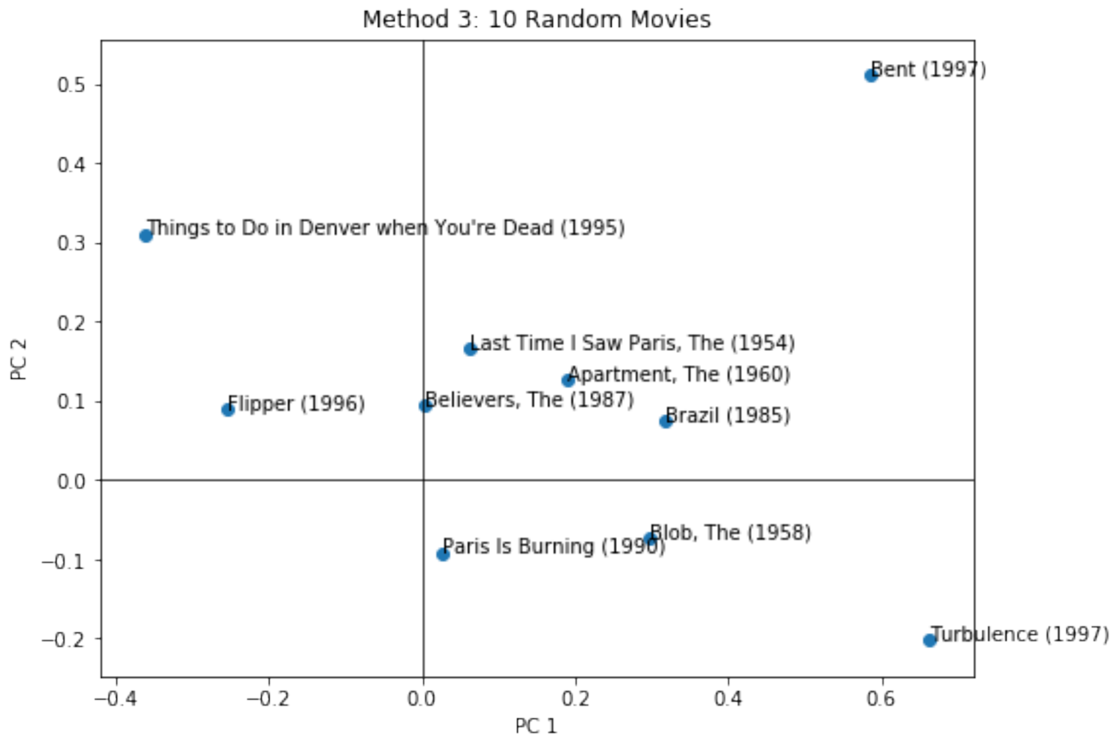
```
    plt.ylabel('PC 2')
    plt.title(plot_title)
    plt.scatter(vizmat.T[picked_id][:,0], vizmat.T[picked_id][:,1])

    # add movie titles
    for i, txt in enumerate(picked_title):
        plt.annotate(txt, (vizmat.T[picked_id][:,0][i], vizmat.T[picked_id][:
↪,1][i]))

    plt.savefig('figures/{}.png'.format(plot_name))
```

(a) 10 random movies from the MovieLens dataset.

[172]:
```
random_id = movies.sample(n = 10)['Movie Id'].values
random_title = movies['Movie Title'].loc[movies['Movie Id'].isin(random_id)].
↪values

for i, vizmat in enumerate(viz):
    movie_viz_2d(vizmat, random_id, random_title,
                 'Method {}: 10 Random Movies'.format(i+1), 'random'+str(i+1))
```

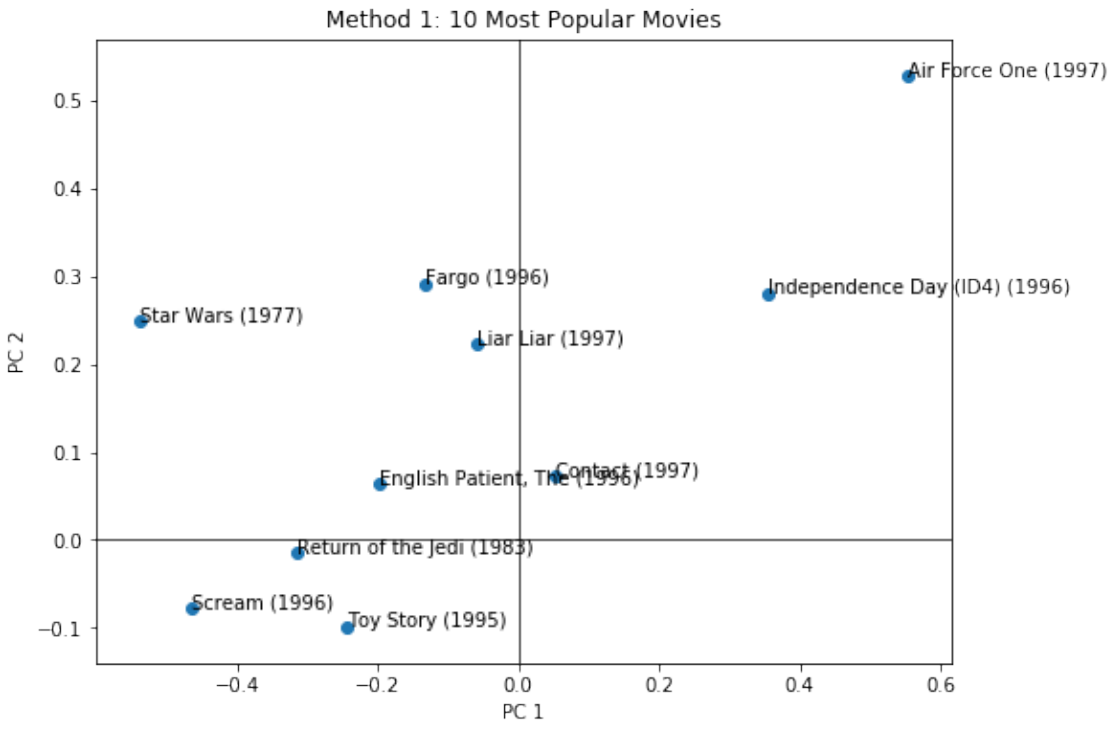


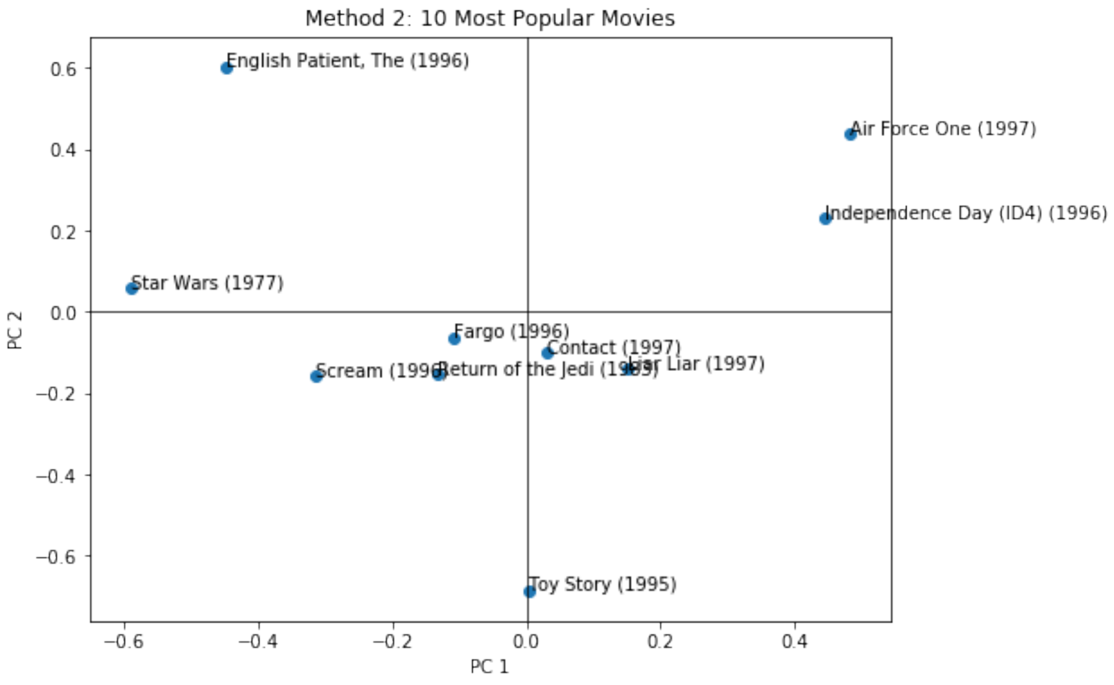


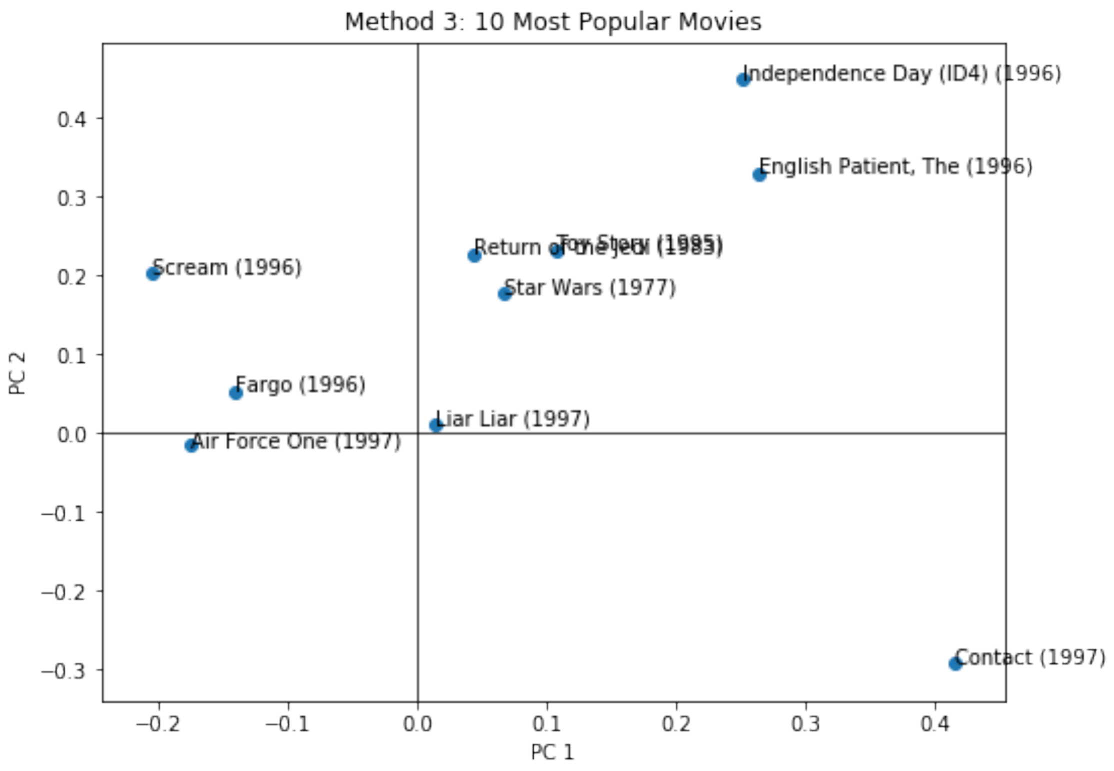


(b) 10 most popular movies.

```
[173]: for i, vizmat in enumerate(viz):
           movie_viz_2d(vizmat, most_pop_id.values, most_pop_title.values,
                        'Method {}: 10 Most Popular Movies'.format(i+1), 'pop'+str(i+1))
```

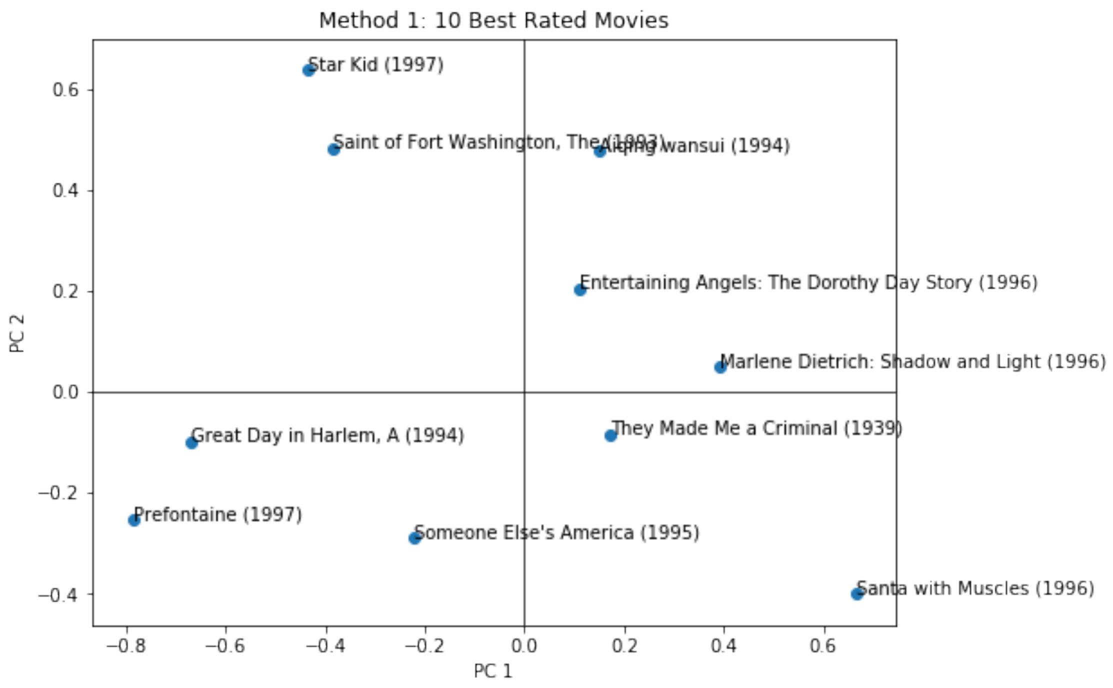


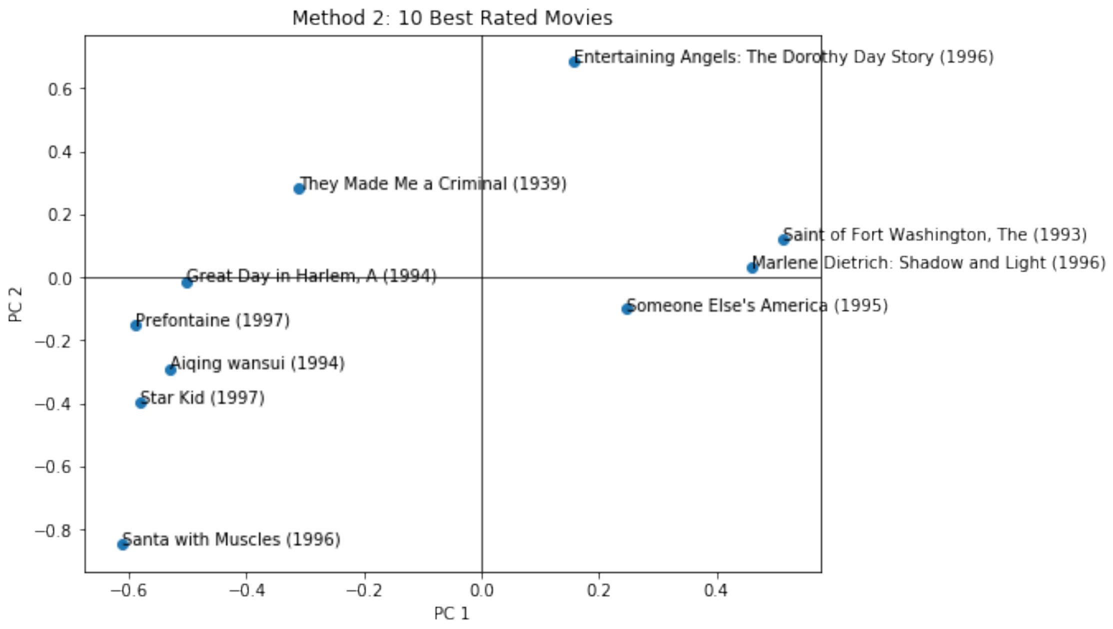


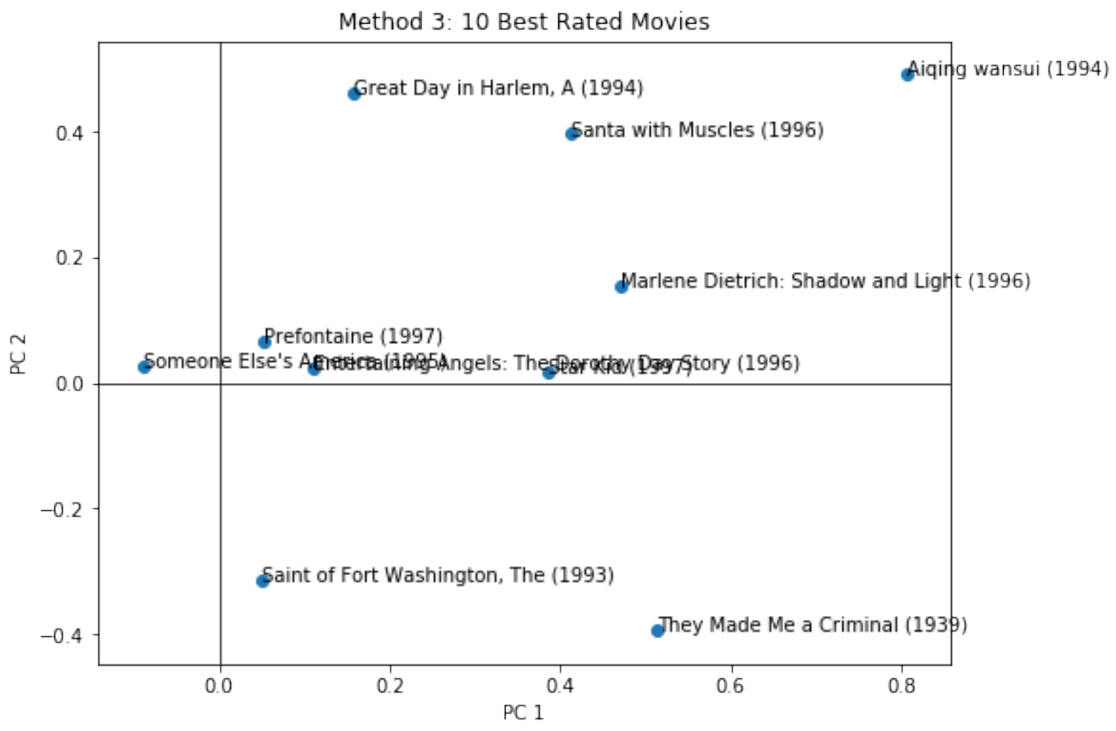


(c) 10 best movies.

[174]:
```
for i, vizmat in enumerate(viz):
    movie_viz_2d(vizmat, best_mov_id.values, best_mov_title.values,
                 'Method {}: 10 Best Rated Movies'.format(i+1), 'best'+str(i+1))
```

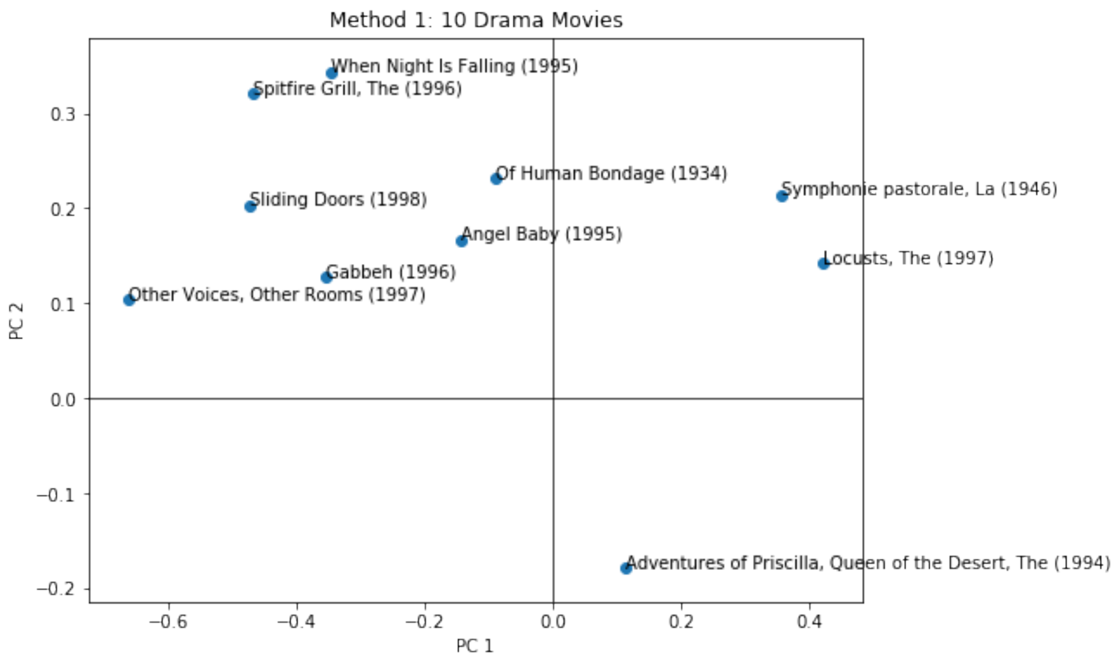

Method 1: 10 Best Rated Movies
Star Kid (1997)
Saint of Fort Washington, The (1993)
Aiqing wansui (1994)
Entertaining Angels: The Dorothy Day Story (1996)
Marlene Dietrich: Shadow and Light (1996)
Great Day in Harlem, A (1994)
They Made Me a Criminal (1939)
Prefontaine (1997)
Someone Else's America (1995)
Santa with Muscles (1996)
PC 1
PC 2

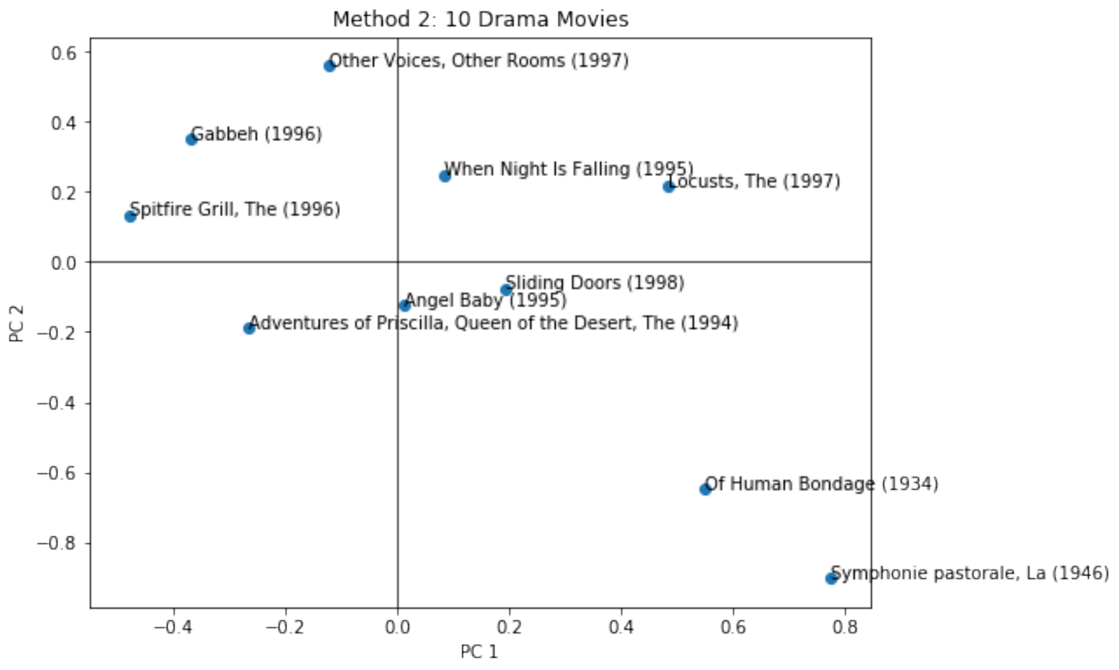

Method 2: 10 Best Rated Movies
Entertaining Angels: The Dorothy Day Story (1996)
They Made Me a Criminal (1939)
Saint of Fort Washington, The (1993)
Marlene Dietrich: Shadow and Light (1996)
Great Day in Harlem, A (1994)
Someone Else's America (1995)
Prefontaine (1997)
Aiqing wansui (1994)
Star Kid (1997)
Santa with Muscles (1996)
PC 1
PC 2

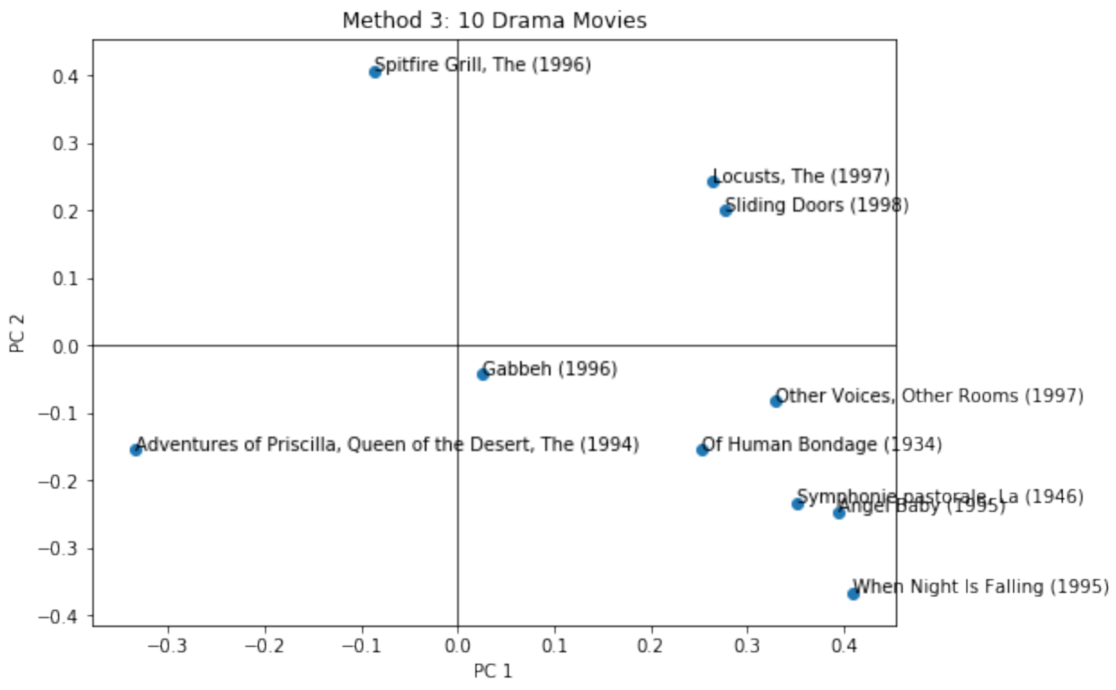


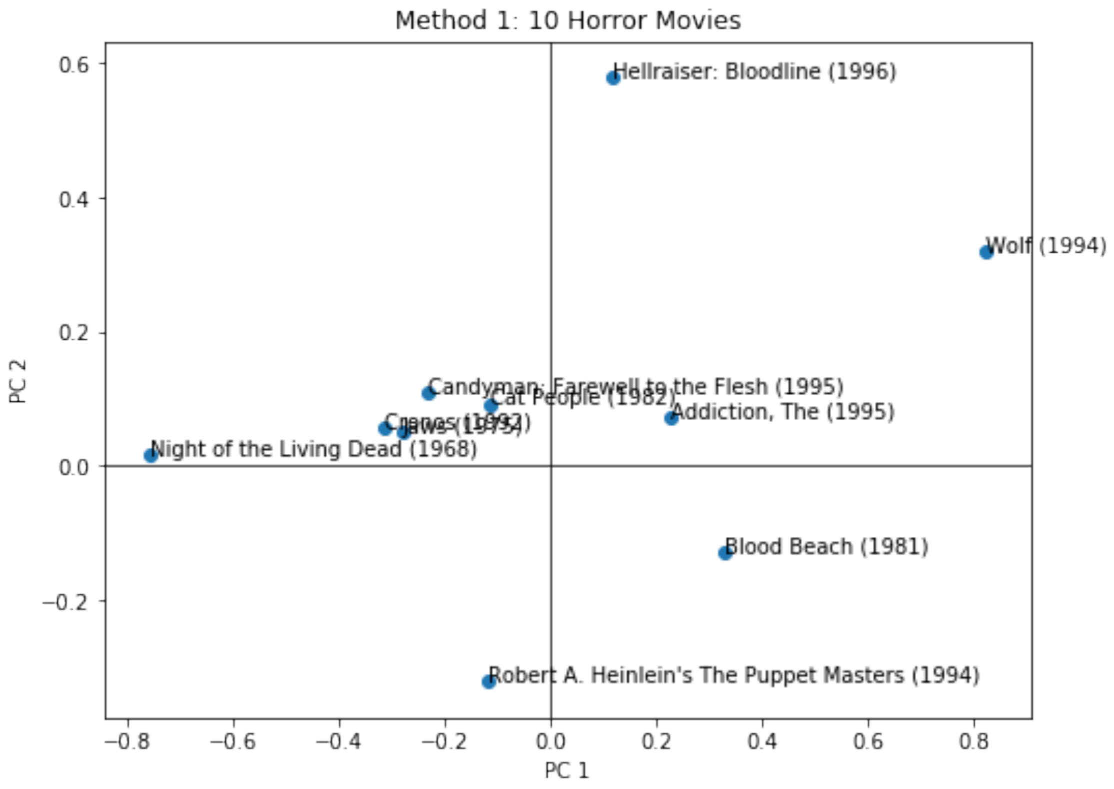
(d) 10 each from ['Drama', 'Horror', 'Fantasy'].

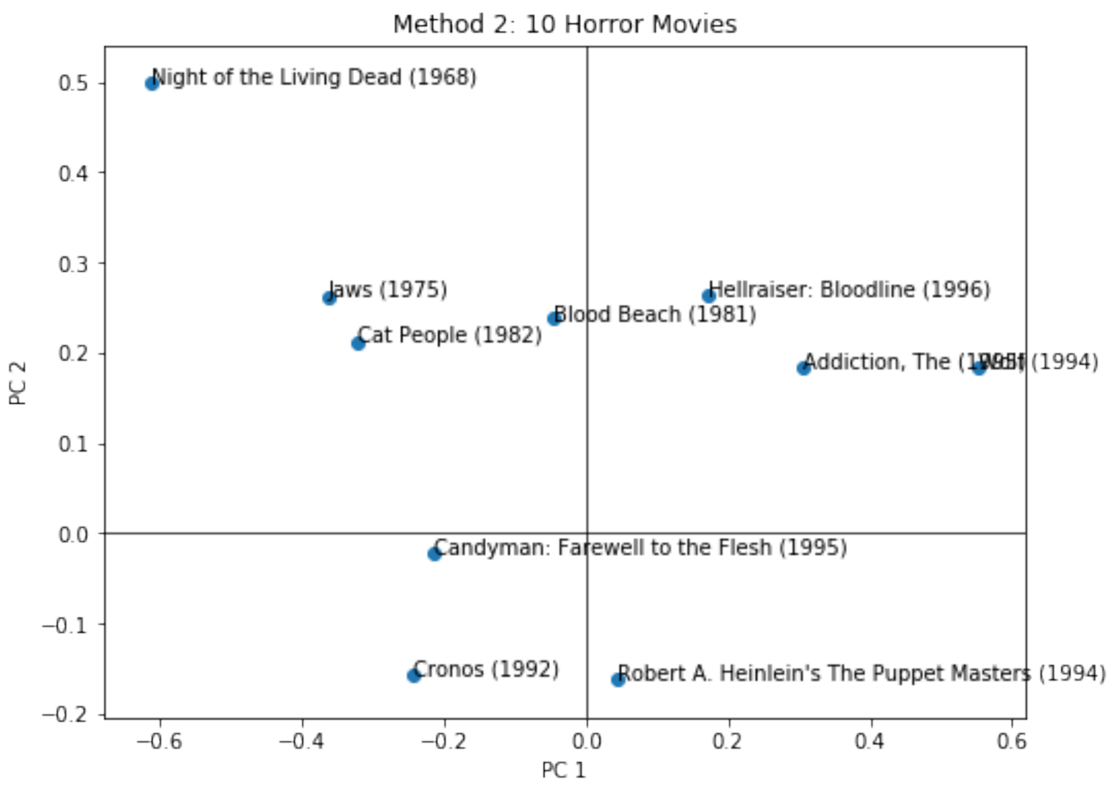
[175]:
```
for genre in ['Drama', 'Horror', 'Fantasy']:
    genre_id = movies[movies[genre] == 1].sample(n = 10)['Movie Id'].values
    genre_title = movies['Movie Title'].loc[movies['Movie Id'].isin(genre_id)].
 ↪values
    for i, vizmat in enumerate(viz):
        movie_viz_2d(vizmat, genre_id, genre_title,
                     'Method {}: 10 {} Movies'.format(i+1, genre), genre+str(i+1))
```

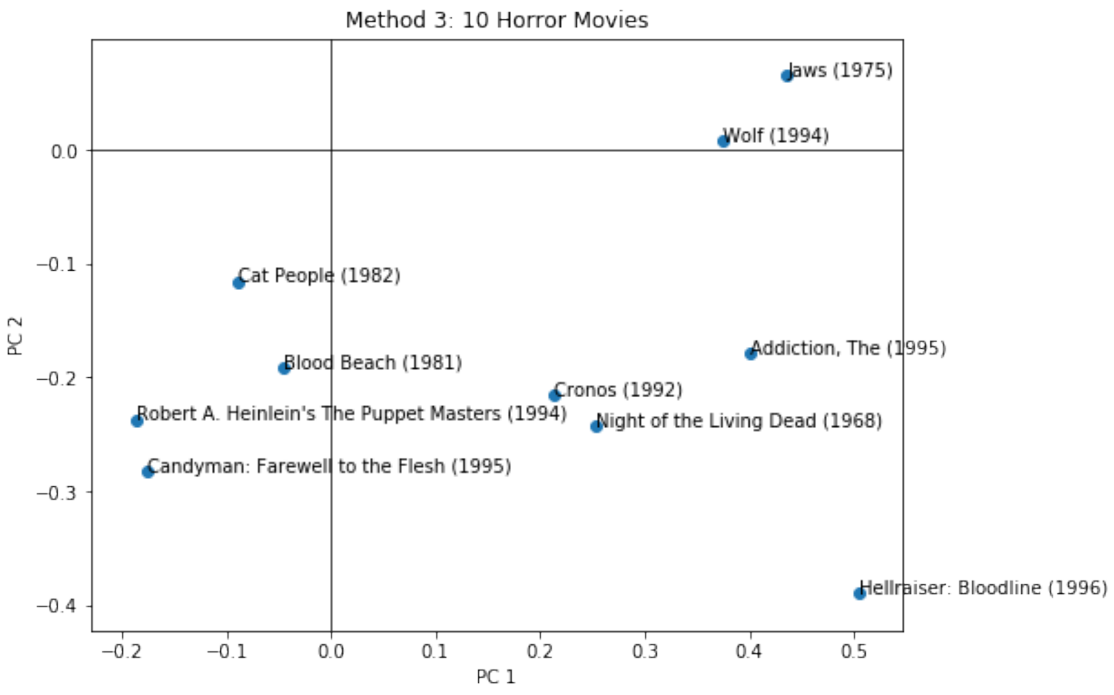

Method 1: 10 Drama Movies
When Night Is Falling (1995)
Spitfire Grill, The (1996)
Of Human Bondage (1934)
Symphonie pastorale, La (1946)
Sliding Doors (1998)
Angel Baby (1995)
Locusts, The (1997)
Gabbeh (1996)
Other Voices, Other Rooms (1997)
Adventures of Priscilla, Queen of the Desert, The (1994)
PC 1
PC 2
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.6
−0.4
−0.2
0.0
0.2
0.4

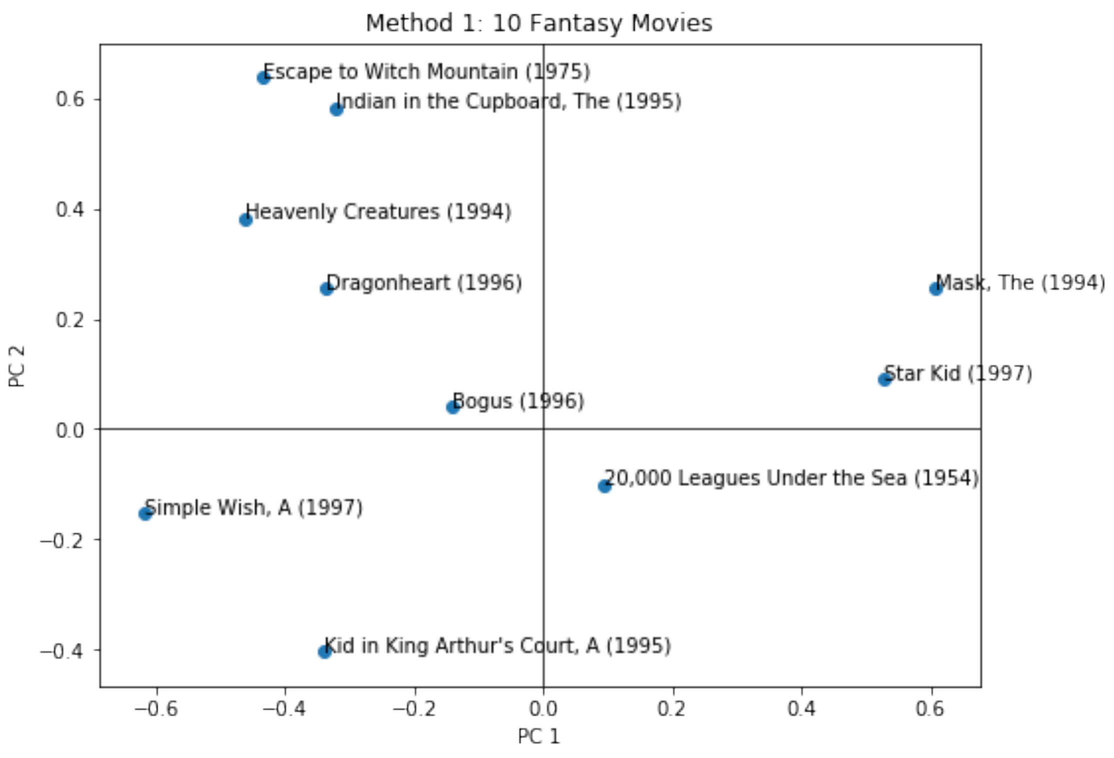

Method 2: 10 Drama Movies
Other Voices, Other Rooms (1997)
Gabbeh (1996)
When Night Is Falling (1995)
Locusts, The (1997)
Spitfire Grill, The (1996)
Sliding Doors (1998)
Angel Baby (1995)
Adventures of Priscilla, Queen of the Desert, The (1994)
Of Human Bondage (1934)
Symphonie pastorale, La (1946)
PC 1
PC 2
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8

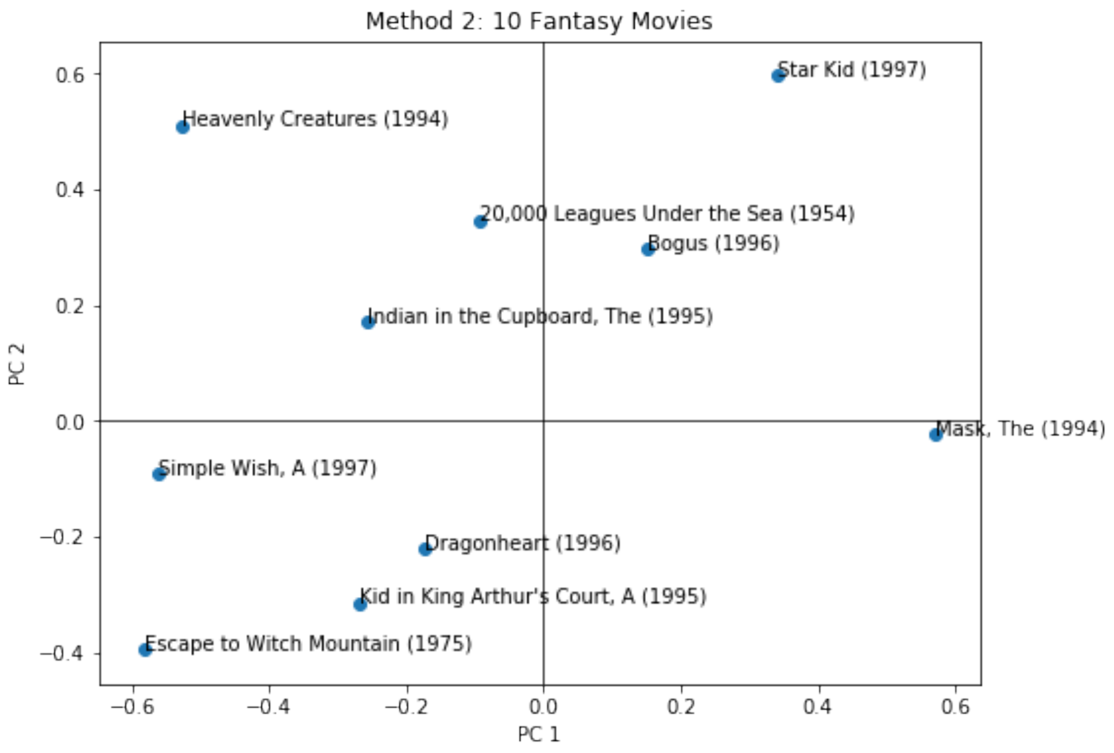

Method 3: 10 Drama Movies
Spitfire Grill, The (1996)
Locusts, The (1997)
Sliding Doors (1998)
Gabbeh (1996)
Other Voices, Other Rooms (1997)
Adventures of Priscilla, Queen of the Desert, The (1994)
Of Human Bondage (1934)
Symphonie pastorale, La (1946)
Angel Baby (1995)
When Night Is Falling (1995)
PC 1
PC 2

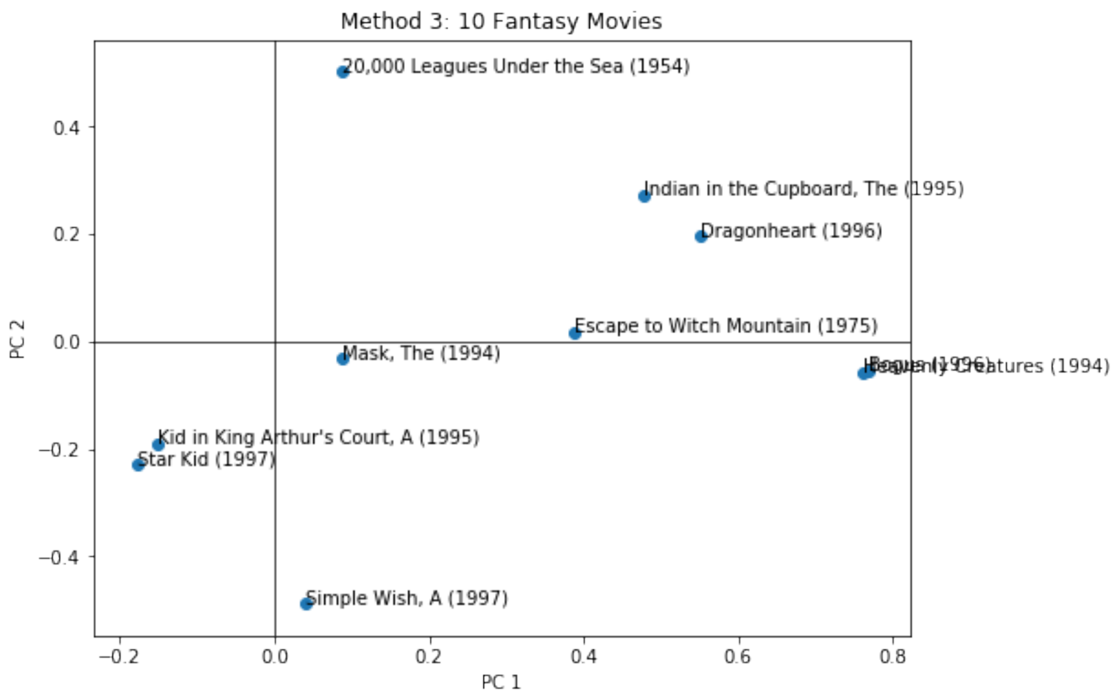

Method 1: 10 Horror Movies
Hellraiser: Bloodline (1996)
Wolf (1994)
Candyman: Farewell to the Flesh (1995)
Cat People (1982)
Addiction, The (1995)
Cronos (1992)
Jaws (1975)
Night of the Living Dead (1968)
Blood Beach (1981)
Robert A. Heinlein's The Puppet Masters (1994)
PC 1
PC 2


Method 2: 10 Horror Movies
Night of the Living Dead (1968)
Jaws (1975)
Hellraiser: Bloodline (1996)
Blood Beach (1981)
Cat People (1982)
Addiction, The (1995)
Wolf (1994)
Candyman: Farewell to the Flesh (1995)
Cronos (1992)
Robert A. Heinlein's The Puppet Masters (1994)
PC 1
PC 2


Method 3: 10 Horror Movies
Jaws (1975)
Wolf (1994)
Cat People (1982)
Addiction, The (1995)
Blood Beach (1981)
Cronos (1992)
Robert A. Heinlein's The Puppet Masters (1994)
Night of the Living Dead (1968)
Candyman: Farewell to the Flesh (1995)
Hellraiser: Bloodline (1996)
PC 1
PC 2


Method 1: 10 Fantasy Movies
Escape to Witch Mountain (1975)
Indian in the Cupboard, The (1995)
Heavenly Creatures (1994)
Dragonheart (1996)
Mask, The (1994)
Star Kid (1997)
Bogus (1996)
20,000 Leagues Under the Sea (1954)
Simple Wish, A (1997)
Kid in King Arthur's Court, A (1995)
PC 1
PC 2


Method 2: 10 Fantasy Movies
Star Kid (1997)
Heavenly Creatures (1994)
20,000 Leagues Under the Sea (1954)
Bogus (1996)
Indian in the Cupboard, The (1995)
Mask, The (1994)
Simple Wish, A (1997)
Dragonheart (1996)
Kid in King Arthur's Court, A (1995)
Escape to Witch Mountain (1975)
PC 1
PC 2


Method 3: 10 Fantasy Movies
20,000 Leagues Under the Sea (1954)
Indian in the Cupboard, The (1995)
Dragonheart (1996)
Escape to Witch Mountain (1975)
Mask, The (1994)
Bogus (1996)
Heavenly Creatures (1994)
Kid in King Arthur's Court, A (1995)
Star Kid (1997)
Simple Wish, A (1997)
PC 1
PC 2