In [1]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn import decomposition
from sklearn import preprocessing
```

In [4]:
```python
df = pd.read_csv("ratings.csv")
```

Out[4]:

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931
5	1	70	3.0	964982400
6	1	101	5.0	964980868
7	1	110	4.0	964982176
8	1	151	5.0	964984041
9	1	157	5.0	964984100
10	1	163	5.0	964983650
11	1	216	5.0	964981208
12	1	223	3.0	964980985
13	1	231	5.0	964981179
14	1	235	4.0	964980908
15	1	260	5.0	964981680
16	1	296	3.0	964982967
17	1	316	3.0	964982310
18	1	333	5.0	964981179
19	1	349	4.0	964982563
20	1	356	4.0	964980962
21	1	362	5.0	964982588
22	1	367	4.0	964981710
23	1	423	3.0	964982363
24	1	441	4.0	964980868
25	1	457	5.0	964981909
26	1	480	4.0	964982346
27	1	500	3.0	964981208
28	1	527	5.0	964984002
29	1	543	4.0	964981179
...	...	...	...	...
100806	610	150401	3.0	1479543210
100807	610	152077	4.0	1493845817
100808	610	152081	4.0	1493846503
100809	610	152372	3.5	1493848841
100810	610	155064	3.5	1493848456
100811	610	156371	5.0	1479542831
100812	610	156726	4.5	1493848444
100813	610	157296	4.0	1493846563
100814	610	158238	5.0	1479545219
100815	610	158721	3.5	1479542491
100816	610	158872	3.5	1493848024
100817	610	158956	3.0	1493848947
100818	610	159093	3.0	1493847704
100819	610	160080	3.0	1493848031
100820	610	160341	2.5	1479545749
100821	610	160527	4.5	1479544998
100822	610	160571	3.0	1493848537
100823	610	160836	3.0	1493844794
100824	610	161582	4.0	1493847759
100825	610	161634	4.0	1493848362
100826	610	162350	3.5	1493849971
100827	610	163937	3.5	1493848789
100828	610	163981	3.5	1493850155
100829	610	164179	5.0	1493845631
100830	610	166528	4.0	1493879365
100831	610	166534	4.0	1493848402
100832	610	168248	5.0	1493850091
100833	610	168250	5.0	1494273047
100834	610	168252	5.0	1493846352
100835	610	170875	3.0	1493846415

100836 rows × 4 columns

1. Transforming Data

In [20]:
```python
userids = pd.unique(df['userId'])
movieids =pd.unique(df['movieId'])

matrix_T = pd.DataFrame(columns=movieids)
for user_id in userids:
    row = []
    temp_df = df[df['userId'] == user_id]
    for movie_id in movieids:
        if movie_id not in list(temp_df['movieId']):
            row.append(0)
        else:
            rate = temp_df[temp_df['movieId'] == movie_id].iloc[0,2]
            row.append(rate)
    matrix_T = matrix_T.append(pd.Series(row, index = movieids), ignore_index=True)

matrix_T.to_csv("matrix_T.csv", index = False)
print("Done.")
```

```
Done.
```

The user-movie ratings matrix is stored in "matrix_T.csv".

In [2]:
```python
matrix_T = pd.read_csv("matrix_T.csv")
matrix_T
```

Out[2]:

	1	3	6	47	50	70	101	110	151	157	...	147662	148166	149011	152372	158721	160341	160527	160836	163937	163981
0	4.0	4.0	4.0	5.0	5.0	3.0	5.0	4.0	5.0	5.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

4	4.0	0.0	0.0	0.0	4.0	0.0	0.0	4.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	5.0	4.0	4.0	1.0	0.0	0.0	5.0	4.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6	4.5	0.0	0.0	0.0	4.5	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	4.0	5.0	0.0	0.0	3.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	0.0	0.0	5.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13	0.0	0.0	0.0	5.0	0.0	0.0	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14	2.5	0.0	0.0	3.5	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
15	0.0	0.0	0.0	3.5	4.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	4.5	0.0	0.0	4.0	4.5	0.0	0.0	4.5	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
17	3.5	0.0	4.0	4.5	5.0	3.5	0.0	4.5	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
18	4.0	3.0	0.0	3.0	0.0	2.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20	3.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
21	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
22	0.0	0.0	4.0	0.0	4.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
23	0.0	0.0	4.5	0.0	4.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
25	0.0	0.0	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
26	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
27	0.0	0.0	3.5	3.0	3.5	0.0	0.0	3.5	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
28	0.0	0.0	0.0	0.0	3.5	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
29	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
580	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
581	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
582	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
583	5.0	0.0	0.0	5.0	0.0	0.0	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
584	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
585	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
586	5.0	0.0	0.0	0.0	5.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
587	0.0	3.0	5.0	3.0	5.0	0.0	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
588	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
589	4.0	3.0	3.5	3.0	4.5	0.0	0.0	4.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
590	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
591	0.0	0.0	3.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
592	0.0	0.0	0.0	0.0	4.5	0.0	0.0	3.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
593	0.0	4.0	0.0	0.0	0.0	3.5	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
594	0.0	0.0	0.0	0.0	5.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
595	4.0	0.0	0.0	0.0	3.5	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
596	4.0	0.0	3.0	4.0	5.0	2.0	5.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
597	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
598	3.0	1.5	4.5	4.0	3.5	3.5	2.5	3.5	0.0	1.5	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
599	2.5	0.0	0.0	4.0	0.0	0.0	4.5	2.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
600	4.0	0.0	0.0	4.0	5.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
601	0.0	0.0	3.0	5.0	5.0	0.0	0.0	5.0	4.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
602	4.0	0.0	4.0	0.0	0.0	4.0	4.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
603	3.0	0.0	3.0	0.0	0.0	0.0	0.0	3.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
604	4.0	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
605	2.5	0.0	0.0	3.0	4.5	4.0	0.0	3.5	0.0	4.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
606	4.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
607	2.5	2.0	0.0	4.5	4.5	3.0	0.0	4.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
608	3.0	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
609	5.0	0.0	5.0	5.0	4.0	4.0	0.0	4.5	0.0	0.0	...	3.0	3.5	3.5	3.5	3.5	2.5	4.5	3.0	3.5	3.5

610 rows × 9724 columns

In [15]:
```
matrix_T.shape
```
Out[15]:
```
(610, 9724)
```

Since every user has only rated a very small part of the movie and the missing values are replaced by 0 in the user-movie ratings matrix, the most common entry in this contructed matrix is 0.
We call a matrix like this a recommander system.

2. Principle Component Analysis

(a)

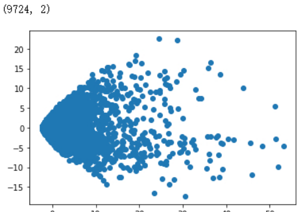
In [4]:
```
# Transform the data
matrix = matrix_T.T
```

In [33]:
```
# Mean center the data
m_scaled = preprocessing.scale(matrix, with_std=False)
```

(b)

In [34]:
```
# Apply PCA with number of components k = 2
pca = decomposition.PCA(n_components=2)
pca.fit(m_scaled)
m_trans = pca.transform(m_scaled)
print(m_trans.shape)

# Plot the new representations of the movies with a scatter plot.
plt.scatter(m_trans[:,0],m_trans[:,1])
plt.show()
```

```
(9724, 2)
```



(c)

In [35]:
```
pca.explained_variance_ratio_
```
Out[35]:
```
array([0.17620942, 0.04189505])
```

From the plot，we can see clearly that data spare more widely on the x-axis, hile the explained variance ratio indicated that the first dimension has greater variance.

(d)

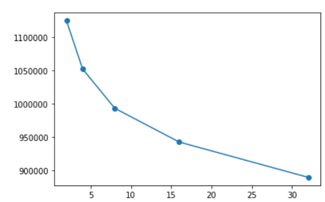
In [31]:
```
variance_sum = 0
k=1
while variance_sum < 0.8:
    pca = decomposition.PCA(n_components=k)
    pca.fit(m_scaled)
    variance_sum = sum(pca.explained_variance_ratio_)
    k += 1
print("k = " +str(k))
print("Total variance: " + str(variance_sum))
```
```
k = 155
Total variance: 0.8003133112551502
```

With k=2, only 21.7% of data is explained. To explain 80% of the variance of the data, 155 principle components are needed. When it comes to visualization, we notice that it's impossible to plot the data with 155 dimensions, while k=2 can be easily polted. Therefore, we can see that when k gets larger, it can explain more data, but it might not be able to visualize.

3. K-Mean

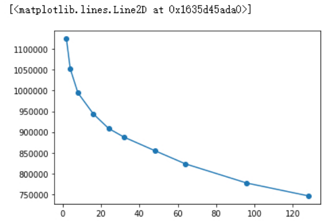
(a)

In [10]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.cluster import KMeans
import matplotlib

ks=[2,4,8,16,32]
inerias = []
for k in ks:
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(matrix)
    y_kmeans = kmeans.predict(matrix)
    ineria = kmeans.inertia_
    inerias.append([k, ineria])
    print(ineria)

inerias = np.array(inerias)
plt.scatter(inerias[:, 0], inerias[:, 1])
plt.plot(inerias[:, 0], inerias[:, 1])
```
```
1124737.0197068886
1052374.2609276343
993351.0115743998
942772.9060929905
889871.0869614722
```
Out[10]:
```
[<matplotlib.lines.Line2D at 0x1635d452b00>]
```



Add more values of k

In [11]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.cluster import KMeans
import matplotlib

ks=[2,4,8,16,24,32,48,64,96,128]
inerias = []
for k in ks:
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(matrix)
    y_kmeans = kmeans.predict(matrix)
    ineria = kmeans.inertia_
    inerias.append([k, ineria])
    print(ineria)

inerias = np.array(inerias)
plt.scatter(inerias[:, 0], inerias[:, 1])
plt.plot(inerias[:, 0], inerias[:, 1])
```
```
1124737.0197068886
1052373.6109772539
994368.0794482677
943864.4636031041
908651.1631666054
887832.3477151245
855284.4860510383
823387.3922342558
777122.7756901473
746326.0422153994
```
Out[11]:
```
[<matplotlib.lines.Line2D at 0x1635d45ada0>]
```



(b)

Choose k = 24.
From the plot, we can see that 24 is the ideal number of k. It is at the "elbow" of the plot. Therefore, it provides a good tradeoff between accuracy (low value of inertia) and complexity (small value of k).

(c)

Movies could be clustered together by the type of movies, i.e. action, sci-fi, horror...
This type of interpretation might not always be possible. Since by choosing different k, the number of clusters is different, which can merge several types of movies into a single type of movie with a small k. For example, if k is really small, comedy, romance, and animation could be grouped together as "fun"movie. While, horror, war, and crime may be clustered as "sad" movie. However, if k a large number, one type of movie might be refined into several types of movies. For

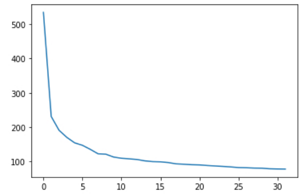
example, action movies can be interpreted into different types by its publishers or actors.

4. Singular Value Decomposition

(a)

In [19]:
```
from sklearn.decomposition import TruncatedSVD
from sklearn.random_projection import sparse_random_matrix

svd = TruncatedSVD(n_components=32)
svd.fit(matrix)
singular = svd.singular_values_
plt.plot(singular)
```

Out[19]: `[<matplotlib.lines.Line2D at 0x1635c746048>]`

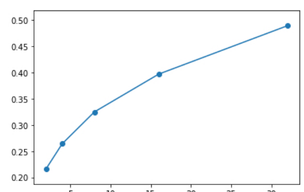


(b)

In [14]:
```
variances = []
for k in [2,4,8,16,32]:
    svd = TruncatedSVD(n_components=k)
    svd.fit(matrix)
    variance = sum(svd.explained_variance_ratio_)
    variances.append([k, variance])
    print(str(k) + ":")
    print(svd.singular_values_)
    print(sum(svd.explained_variance_ratio_))
    print("\n")

variances = np.array(variances)
plt.scatter(variances[:, 0], variances[:, 1])
plt.plot(variances[:, 0], variances[:, 1])
```

```
2:
[534.41989777 231.23661061]
0.21642917529752204


4:
[534.41989777 231.236611   191.15084454 170.42188613]
0.2641428516880421


8:
[534.41989777 231.23661123 191.15084979 170.42239983 154.5522061
 147.33464029 135.64513529 122.61186255]
0.32499747803948353


16:
[534.41989777 231.2366114  191.1508759  170.42249307 154.55274805
 147.33552678 135.65525637 122.66035184 121.43769458 113.08261962
 109.56533707 107.83483322 105.85954835 101.77109818  99.57623689
  99.06365971]
0.39724086776976014


32:
[534.41989777 231.23661141 191.15087615 170.42250816 154.55294482
 147.33574096 135.6555122  122.66253061 121.44176937 113.11034866
 109.5964902  107.92580094 105.96758061 102.04653537  99.86115755
  99.27106335  97.10015173  93.29345448  92.24387347  90.89076604
  90.15260453  88.71208896  87.06893178  85.85554522  84.90663537
  82.52429351  82.05864364  81.32257983  80.60800026  79.10004555
  78.48117571  76.88147308]
0.4896308009255697
```

Out[14]: `[<matplotlib.lines.Line2D at 0x1635c5d8da0>]`

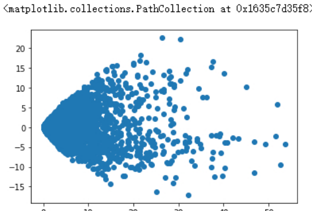


When increasing k in k-mean, the inertia values decreases which means that entities are getting closer to its cluster's center point. As for SVD, increasing k can increase explained variance ratio. In that case, more data can be explained after dimensionality reduction. I chose k = 24 in the previous question. From the plot above, we can see that around 45% of data can be explained by this number of components.

(c)

In [21]:
```
svd = TruncatedSVD(n_components=2)
svd.fit(matrix)
m_reduced = svd.fit_transform(matrix)
m_reduced
plt.scatter(m_reduced[:, 0], m_reduced[:, 1])
```

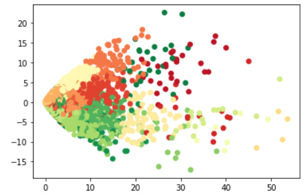
Out[21]: `<matplotlib.collections.PathCollection at 0x1635c7d35f8>`



(d)

In [22]:
```
kmeans = KMeans(n_clusters=24)
kmeans.fit(matrix)
y_kmeans = kmeans.predict(matrix)
plt.scatter(m_reduced[:, 0], m_reduced[:, 1], c=y_kmeans, cmap=plt.cm.RdYlGn)
```

Out[22]: `<matplotlib.collections.PathCollection at 0x1635cac3470>`



From the plot above, we can see that data tranformed by SVD has almost the same shape as data transformed by PCA when the number of components is 2. Besides, movies are colored by the cluster memberships discovered by k-mean. It obvious that movies in the same cluster are grouped together. This support that clustering works well.