KLukowiakAssignment1

June 10, 2018

1 Recomender Systems

1.1 Assignment 1

This assignment is for CUNY's DATA 643 Recomender Systems.

Briefly describe the recommender system that you're going to build out from a business perspective, e.g. "This system recommends data science books to readers."

This project's goal is to build a joke recomender system to make us laugh in these trying times.

If one wants to imporove ones mood, the more high quality jokes and fewer low quality jokes encountered for a particular user the better. We will use baisic mean imputation as well as joke and user bias, to predict joke values on non-rated jokes. This will be validated on a test set.

Find a dataset, or build out your own toy dataset. As a minimum requirement for complexity, please include numeric ratings for at least five users, across at least five items, with some missing data.

I downloaded the dataset from this site. The dataset is quite wide at 151 rows.

This is known as a user-item matrix because each row is a user and each column is an item, in this case a joke.

```
In [1]: import pandas as pd
        import numpy as np
        df = pd.read_excel('jester-data-1.xls', header=None)

In [2]: df.iloc[0:5, 0:10]
```

Out[2]:

	0	1	2	3	4	5	6	7	8	9
0	74	-7.82	8.79	-9.66	-8.16	-7.52	-8.50	-9.85	4.17	-8.98
1	100	4.08	-0.29	6.36	4.37	-2.38	-9.66	-0.73	-5.34	8.88
2	49	99.00	99.00	99.00	99.00	9.03	9.27	9.03	9.27	99.00
3	48	99.00	8.35	99.00	99.00	1.80	8.16	-2.82	6.21	99.00
4	91	8.50	4.61	-4.17	-5.39	1.36	1.60	7.04	4.61	-0.44

```
In [3]: df.shape
```

```
Out[3]: (24983, 101)
```

The dataset contains 100 jokes as well the count of the ratings. The user ID is given by the pandas index

```
In [3]: from bokeh.io import output_notebook, show
        from bokeh.plotting import figure

        output_notebook()
```

```
In [4]: jokes = df.iloc[:, 1:].replace(99, np.NAN)
```

```
In [5]: def remover(p):
            # Set axis to invisiable
            p.xaxis.axis_line_width = 0.00001
            p.yaxis.axis_line_width = 0.00001
            # Fonts
            p.title.text_font = "times"
            p.title.text_font_style = "normal"
            p.xaxis.axis_label_text_font = 'times'
            p.xaxis.axis_label_text_font_style = 'normal'
            p.yaxis.axis_label_text_font = 'times'
            p.yaxis.axis_label_text_font_style = 'normal'
            # This removes the outline of the graph.
            p.outline_line_color = None
            p.toolbar.logo = None
            p.toolbar_location = None
            p.xgrid.grid_line_color = None
            p.ygrid.grid_line_color = None
            return p
```

```
In [7]: from bokeh.layouts import gridplot
        from bokeh.plotting import figure, show, output_file

        p1 = figure(title="Frequency of Number of Jokes Rated")

        hist, edges = np.histogram(df[0], density=True, bins=30)

        #x = np.linspace(-2, 2, 1000)
        p1.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:],
            fill_color="skyblue", line_color="grey")
        p1.xaxis.axis_label = 'Total Jokes Rated'
        p1.yaxis.axis_label = 'Proportion of Responses'
        p1 = remover(p1)
        show(p1)
```

We have some complete cases here, however, even if complete cases were not correlated with joke preferences, we still would be thowing out a crazy amount of data for training.

Since we must train on data that is not a complete case, we need to validate it. Because we must predict NA values for people who have been used as training sets, we need a way to validate.

To do this we can select random ratings to be our test set.

I removed the first column because it was the number of jokes answered.

This is less interesting to us because we don't need to understand how many jokes were answered.

```
In [6]: p1 = figure(title="Rating Distrobution")
        allHist = pd.melt(jokes)
        hist, edges = np.histogram(allHist['value'].dropna(), density=True, bins=30)
        p1.quad(top=hist, bottom=0, left=edges[:-1], right=edges[1:],
            fill_color="skyblue", line_color="grey")
        p1 = remover(p1)
        show(p1)
```

The data looks like there is a preference for jokes just above average. Also, more people seem to be fine giving the max negative value indicating they hated the joke while less give the max positive value. Conversly, there are more people giving positive values.

1.2 Simple Predictions

We are tasked with predicting joke ratings for jokes that users have not yet rated. There is no way to verify what they will rate. Instead we will have to validate with a training and test set.

Once we have some strategy, we can apply it to the `NA` values.

Because the NA values are distributed amoung all but a few responders.

Break your ratings into separate training and test datasets.

```
In [9]: np.random.seed(101)
        trainTestMask = np.random.choice([True, False], size=jokes.shape, p=[0.7, 0.3])
        trainTestMask = pd.DataFrame(trainTestMask)
        trainTestMask.head()
```

Out[9]:

	0	1	2	3	4	5	6	7	8	9	...	\
0	True	True	True	True	True	False	True	False	False	True	...	
1	True	True	True	True	False	True	True	True	False	True	...	
2	True	True	True	True	False	True	True	True	True	True	...	
3	True	True	True	True	True	True	False	True	True	True	...	
4	True	True	True	True	True	True	True	True	True	True	...	

	90	91	92	93	94	95	96	97	98	99
0	True	True	True	True	True	False	True	True	False	True
1	True	True	True	True	True	False	True	False	True	True
2	True	True	False	False	False	True	True	False	True	True
3	True	True	True	True	True	False	False	True	True	True

```
4  False  True   True   True  False   True  False   True   True  True

[5 rows x 100 columns]
```

This gives us a way to randomly select training and test sets, but we still need to test for NA values.

```
In [10]: train = pd.DataFrame(np.where(trainTestMask, jokes, np.NAN))
         test = pd.DataFrame(np.where(np.invert(trainTestMask), jokes, np.NAN))
```

We have our training and test data. We first need to calculate the mean for the total training set and calculate the RMSE on the test set.

```
In [11]: rawMean = train.stack().mean()
         rawMean

Out[11]: 0.8783613614046262
```

The raw mean is close to zero. We will replace all na values with this and calculate the RMSE.

Using your training data, calculate the raw average (mean) rating for every user-item combination.

- Calculate the RMSE for raw average for both your training data and your test data.

```
In [12]: testRaw = train.fillna(value=rawMean)

In [13]: np.sqrt(((pd.melt(test)['value'] - rawMean)**2).mean()) #  This ignores NA values tha

Out[13]: 5.2324418349094142

In [14]: np.sqrt(((pd.melt(train)['value'] - rawMean)**2).mean()) #  This ignores NA values th

Out[14]: 5.2373261451264863
```

So we have a RMSE of 5.23 for the test set and surprisingly, a higher one for the training set. This is a bit weird because we would expect some over fitting. However, given that we are just taking the meanit is unlikely there is a statistical difference.

1.3 More accuracy

To get more accuracy, we will need to add the column means and row means to the raw mean.

The formula should look like:

$$pred = RawMean + SpecificColumnMean + SpecificRowMean$$

Another way to think of it is:

$$pred_{i,j} = RawMean + RowMean_j + ColMean_i$$

Using your training data, calculate the bias for each user and each item.

```
In [15]: colMean = train.mean()
         rowMean = train.mean(axis=1)

In [16]: preds = np.full(jokes.shape, rawMean)
         preds += colMean
         preds =preds + rowMean[:,np.newaxis]

In [17]: preds = pd.DataFrame(preds)
         preds.head()
```

Out[17]:

	0	1	2	3	4	5	6	\
0	-1.885926	-2.646217	-2.469234	-4.251214	-2.334262	-1.162556	-3.202825	
1	3.927500	3.167209	3.344193	1.562212	3.479165	4.650870	2.610602	
2	8.840250	8.079959	8.256942	6.474962	8.391914	9.563620	7.523351	
3	5.250582	4.490291	4.667274	2.885294	4.802246	5.973952	3.933683	
4	5.170784	4.410493	4.587476	2.805496	4.722449	5.894154	3.853885	

	7	8	9	...	90	91	92	\
0	-3.425682	-3.344824	-1.531391	...	-0.757232	-1.621067	-0.324863	
1	2.387745	2.468603	4.282036	...	5.056195	4.192359	5.488563	
2	7.300494	7.381352	9.194785	...	9.968944	9.105109	10.401313	
3	3.710827	3.791684	5.605117	...	6.379276	5.515441	6.811645	
4	3.631029	3.711887	5.525319	...	6.299478	5.435643	6.731847	

	93	94	95	96	97	98	99
0	-1.583458	-1.836889	-1.352553	-1.123196	-2.061757	-2.840812	-1.467379
1	4.229969	3.976538	4.460873	4.690230	3.751670	2.972614	4.346047
2	9.142718	8.889287	9.373623	9.602980	8.664419	7.885364	9.258797
3	5.553050	5.299619	5.783955	6.013312	5.074751	4.295696	5.669129
4	5.473252	5.219822	5.704157	5.933514	4.994953	4.215898	5.589331

[5 rows x 100 columns]

```
In [18]: np.sqrt(((preds - test)**2).stack().mean())

Out[18]: 4.68875588784186

In [19]: np.sqrt(((preds - train)**2).stack().mean())

Out[19]: 4.6173021500924856
```

Adding the column and row biases lead to significant imporovement of the RMSE. We now see the training data slightly overfitting when compared to the test data.

1.4 Predictions,

We have already made predictions for the entire data frame, now we just need to add them to the main dataset.

```
In [20]: isNA = jokes.isnull()
```

```
In [21]: fullDF = pd.DataFrame(np.where(isNA, preds, jokes))

In [22]: fullDF.head(20)
```

Out[22]:

	0	1	2	3	4	5	6	7	\
0	-7.820000	8.790000	-9.660000	-8.160000	-7.52	-8.500000	-9.85	4.17	
1	4.080000	-0.290000	6.360000	4.370000	-2.38	-9.660000	-0.73	-5.34	
2	8.840250	8.079959	8.256942	6.474962	9.03	9.270000	9.03	9.27	
3	5.250582	8.350000	4.667274	2.885294	1.80	8.160000	-2.82	6.21	
4	8.500000	4.610000	-4.170000	-5.390000	1.36	1.600000	7.04	4.61	
5	-6.170000	-3.540000	0.440000	-8.500000	-7.09	-4.320000	-8.69	-0.87	
6	6.109847	5.349555	5.526539	3.744558	8.59	-9.850000	7.72	8.79	
7	6.840000	3.160000	9.170000	-6.210000	-8.16	-1.700000	9.27	1.41	
8	-3.790000	-3.540000	-9.420000	-6.890000	-8.74	-0.290000	-5.29	-8.93	
9	3.010000	5.150000	5.150000	3.010000	6.41	5.150000	8.93	2.52	
10	-2.910000	4.080000	4.622986	2.841006	-5.73	5.929664	2.48	-5.29	
11	1.310000	1.800000	2.570000	-2.380000	0.73	0.730000	-0.97	5.00	
12	7.024204	6.263912	6.440896	4.658916	5.87	7.747574	5.58	0.53	
13	9.220000	9.270000	9.220000	8.300000	7.43	0.440000	3.50	8.16	
14	8.790000	-5.780000	6.020000	3.690000	7.77	-5.830000	8.69	8.59	
15	-3.500000	1.550000	2.330000	-4.130000	4.22	-2.280000	-2.96	-0.49	
16	3.702775	-9.270000	3.119467	1.337487	-7.38	4.426145	8.74	-6.31	
17	3.160000	7.620000	3.790000	8.250000	4.22	7.620000	2.43	0.97	
18	4.220000	3.640000	2.668232	0.886252	2.52	3.974910	4.13	-5.19	
19	1.112847	7.620000	0.529539	-1.252442	-8.64	2.430000	8.93	-6.60	

	8	9	...	90	91	92	93	\
0	-8.980000	-4.760000	...	2.820000	-1.621067	-0.324863	-1.583458	
1	8.880000	9.220000	...	2.820000	-4.950000	-0.290000	7.860000	
2	7.381352	9.194785	...	9.968944	9.105109	10.401313	9.080000	
3	3.791684	1.840000	...	6.379276	5.515441	6.811645	0.530000	
4	-0.440000	5.730000	...	5.190000	5.580000	4.270000	5.190000	
5	-6.650000	-1.800000	...	-3.540000	-6.890000	-0.680000	-2.960000	
6	4.650949	6.464382	...	7.238541	6.374705	7.670909	6.412315	
7	-5.190000	-4.420000	...	7.230000	-1.120000	-0.100000	-5.680000	
8	-7.860000	-1.600000	...	4.370000	-0.290000	4.170000	-0.290000	
9	3.010000	8.160000	...	7.604834	4.470000	8.037203	6.778608	
10	3.747397	1.460000	...	6.334988	6.170000	6.767357	5.508762	
11	-7.230000	-1.360000	...	1.460000	1.700000	0.290000	-3.300000	
12	5.565306	7.140000	...	8.152898	7.289062	7.520000	7.326672	
13	5.970000	8.980000	...	8.110000	-1.020000	5.580000	6.840000	
14	-5.920000	7.520000	...	2.720000	-5.490000	-8.590000	8.690000	
15	2.910000	1.990000	...	3.110000	1.700000	0.240000	-5.920000	
16	2.243878	2.330000	...	4.831470	3.967634	5.263838	4.005243	
17	0.530000	0.830000	...	0.830000	5.680000	3.690000	0.190000	
18	1.792643	7.910000	...	4.380235	0.050000	4.812603	3.554008	
19	-0.346051	-9.470000	...	2.241541	1.377705	2.673909	1.415315	

```
          94        95        96        97        98        99
0  -1.836889 -1.352553 -5.630000 -2.061757 -2.840812 -1.467379
1  -0.190000 -2.140000  3.060000  0.340000 -4.320000  1.070000
2   8.889287  9.373623  9.602980  8.664419  7.885364  9.258797
3   5.299619  5.783955  6.013312  5.074751  4.295696  5.669129
4   5.730000  1.550000  3.110000  6.550000  1.800000  1.600000
5  -2.180000 -3.350000  0.050000 -9.080000 -5.050000 -3.450000
6   6.158884  2.330000  6.872576  5.934016  5.154960  6.528393
7  -3.160000 -3.350000  2.140000 -0.050000  1.310000  0.000000
8  -0.290000 -0.290000 -0.290000 -0.290000 -3.400000 -4.950000
9   6.525177  7.009512  7.238870  6.300309  5.521254  6.894686
10  5.255331  5.739667  5.969024  5.030463  4.251408  5.624841
11  3.450000  5.440000  4.080000  2.480000  4.510000  4.660000
12  7.073241  7.557576  7.786934  6.848373  6.069318  7.442750
13  5.530000 -5.920000  8.200000  8.980000 -8.160000  6.500000
14 -8.740000 -3.010000  8.300000 -4.810000 -2.380000 -5.970000
15  7.280000 -1.360000  3.740000  2.820000 -2.860000  3.450000
16  3.751813  4.236148  4.465505  3.526944  2.747889  4.121322
17  0.290000  3.590000  0.490000  8.060000  0.490000  7.620000
18  3.300578  3.784913  1.650000  3.075709  2.296654  3.670087
19  1.161884  1.646219  1.875576  0.937016  0.157960  1.531393

[20 rows x 100 columns]
```

We can see that more bias we can account for, them lower RMSE we have.
