Machine Learning Engineer Nanodegree

Capstone Project

Sam Ariabod
September 16, 2018

I. Definition

Project Overview

The amount of information on the internet is growing exponentially. 90% of the data on the internet has been created since 2016[1].

Recommender systems in the internet have become so normal that we tend to forget how critical they are[2]. From shopping, to music, to education – the amount of options available is staggering. Netflix posted a Kaggle competition with a reward of $1,000,000.00 USD to build a recommender model that performed better than the one they were using or surpassed the previous year winner[3].

Connecting users to content that is relevant in a timely fashion is important to providing a good user experience. This can be the difference between a user that never comes back to a repeat customer.

The origin of this project stems from a client I work with that offers online training. The courses number in the hundreds across a dozen different categories. In order to help a user find a course that is relevant we will leverage the power of machine learning to analyze historical and current course/user data to build a recommendation engine.

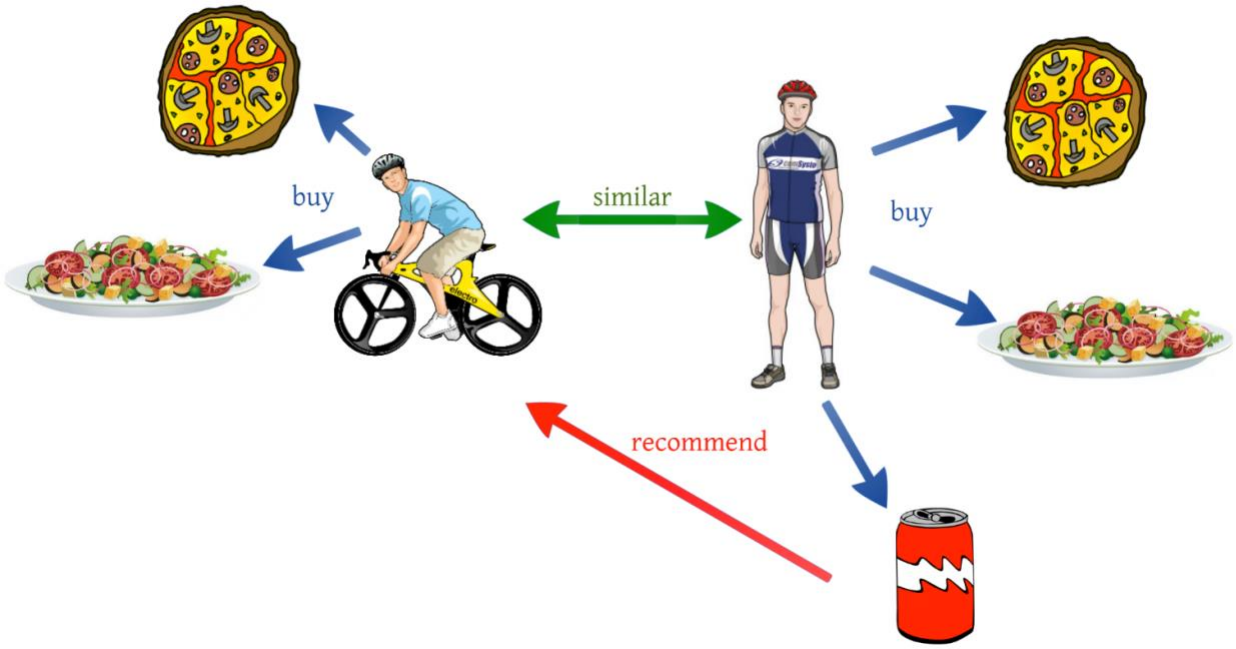
Once the model is built it will be served up via a RESTful API then integrated into the live site.

[1] https://public.dhe.ibm.com/common/ssi/ecm/wr/en/wrl12345usen/watson-customer-engagement-watson-marketing-wr-other-papers-and-reports-wrl12345usen-20170719.pdf
[2] http://www.commons.fm/?p=25017
[3] https://en.wikipedia.org/wiki/Netflix_Prize

Problem Statement

The eLearning Course provider has too many courses spread over a dozen categories for users to reasonably go through. The site offers course reviews and offers a text search for users to try and narrow down the offerings. Neither of these account for any relationship between users or the format of the courses (some are text heavy, some are video based, some are interactive).

The solution for this problem assumes that people tend to like things similar to other things they like, and things that are liked by other people with similar taste.



[4]

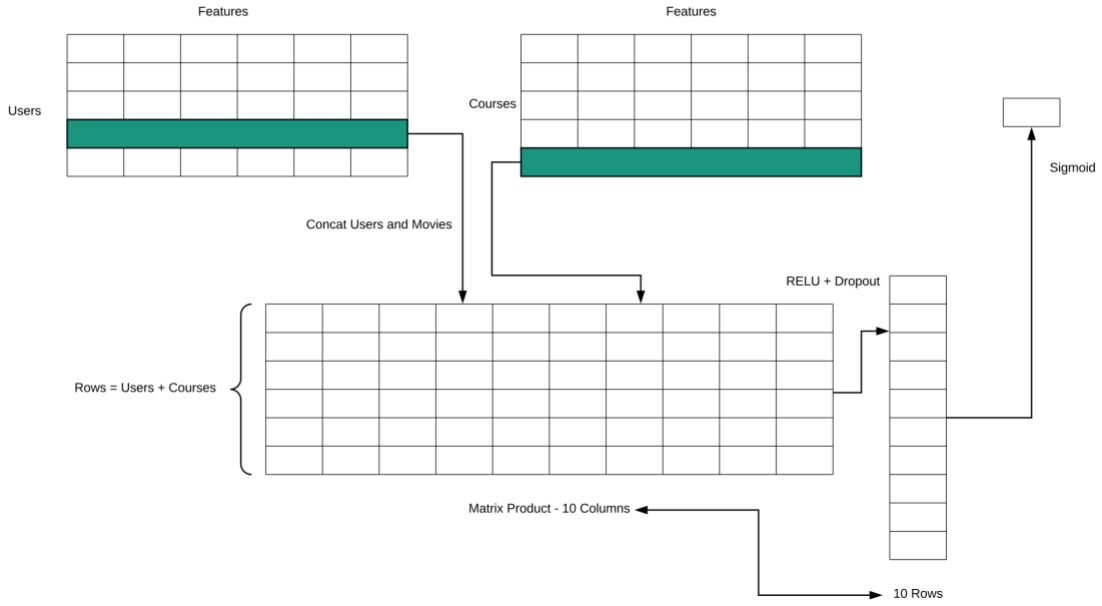
To solve this, we will be taking all user profiles that have completed and rated courses and all course ratings and use collaborative filtering to find hidden relationships.

The technique we will be using for this collaborative filtering model is Matrix (or Tensor) Factorization:

"Matrix factorization can be used to discover latent features underlying the interactions between two different kinds of entities. (Of course, you can consider more than two kinds of entities and you will be dealing with tensor factorization, which would be more complicated.) And one obvious application is to predict ratings in collaborative filtering."[5]

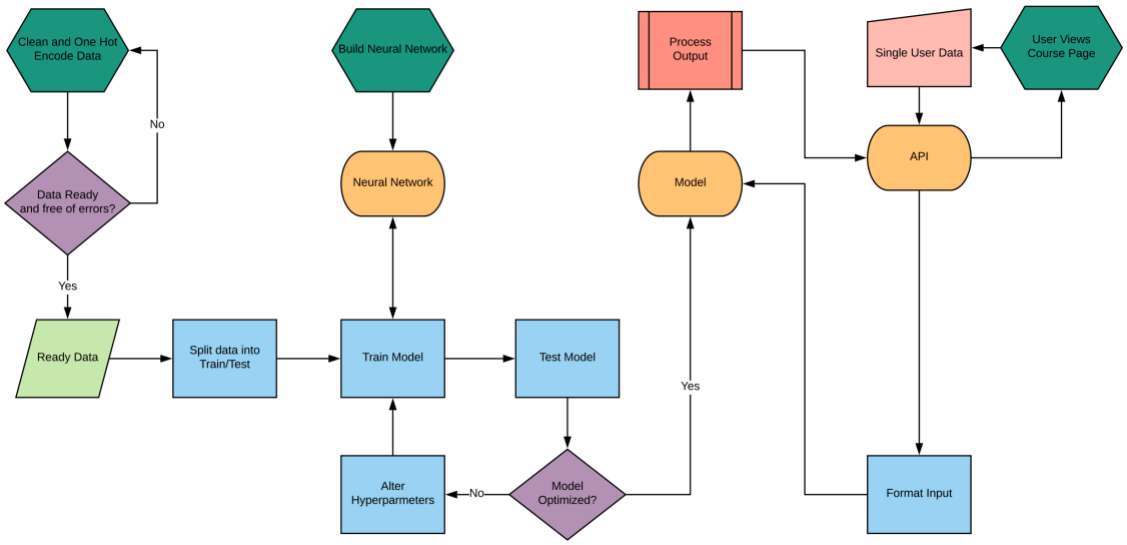
[4] https://medium.com/@tomar.ankur287/user-user-collaborative-filtering-recommender-system-51f568489727
[5] http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/

Matrix Factorization with a Neural Net Basic Flow:



Once the model is built, we will be able to pass in the user information and the system will be able to calculate ratings for missing entries (courses the user has not taken). We can then serve this up through an API.

Birds eye view of the process from data analysis to going live with API:



Metrics

To evaluate the model accuracy, we will be using a very standard Root Mean Squared Error:

$$RMSE = \sqrt{\frac{\sum_{n=1}^{n=N} (P_n - O_n)^2}{N-1}}$$

"Root Mean Square Error (RMSE) is the standard deviation of the residuals (prediction errors). Residuals are a measure of how far from the regression line data points are; RMSE is a measure of how spread out these residuals are. In other words, it tells you how concentrated the data is around the line of best fit. Root mean square error is commonly used in climatology, forecasting, and regression analysis to verify experimental results."[6]

As we train the model, we will try to minimize the error rate of prediction ratings vs actual ratings.

II. Analysis

Data Exploration

The data includes 6 features (all categorical) and 1 target variable.

Features:

User – This is the user that rated the course (cardinality: 52118)

Course – Course that the rating was given on (cardinality: 219)

Category – The category the course belongs to (cardinality: 14)

Job – This is the job category of the user (cardinality: 45)

Institution – This is the institution type that the user belongs to (cardinality: 15)

State – This is the state the user resides in (cardinality: 60)

Target:

[6] http://www.statisticshowto.com/rmse/

Rating – This the overall rating for a course, it will have a potential range of 0 – 5 where 0 is on the low end and 5 being a perfect score. This will be what we are trying to predict given a user and a course.

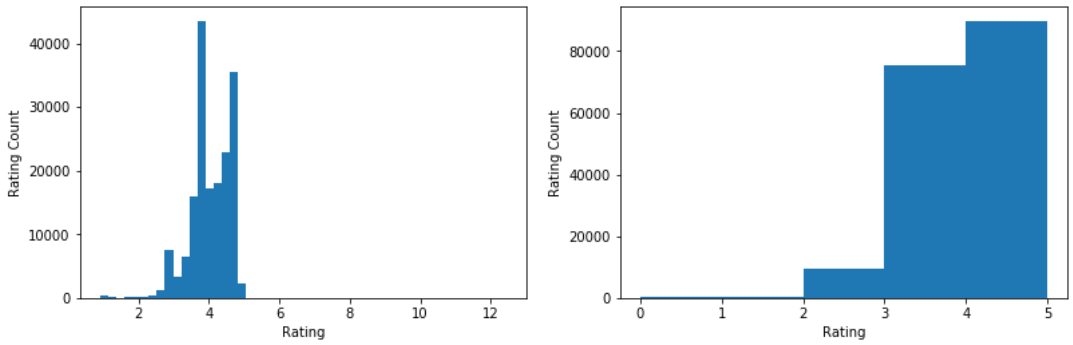
Exploratory Visualization

Rating Stats:

	rating
count	175347.000000
mean	4.030533
std	0.571618
min	0.910000
25%	3.690000
50%	4.000000
75%	4.590000
max	12.450000

Looking at the max, it looks like we have some bad data that we need to clean up. A rating of 12.45 is just not possible.

Rating Distribution:



Excluding outliers, a majority of the ratings are between 3-5 with a high spike right around 4.

Mean: 4.03053282919012 - Median: 4.0

Overall, the records are clean. No missing values across the different features.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 175347 entries, 0 to 175346
Data columns (total 7 columns):
user           175347 non-null object
course         175347 non-null object
category       175347 non-null object
rating         175347 non-null float64
job            175347 non-null object
institution    175347 non-null object
state          175347 non-null object
dtypes: float64(1), object(6)
memory usage: 9.4+ MB
```

Algorithms and Techniques

The main technique we will be using is matrix factorization with embeddings. Embeddings allow us to implement additional features (properties) of user and courses that will help create more accurate predictions. We will be running the model over multiple iterations to help reduce the bias. We will also be taking a portion of the training data out to create a validation set. This along with dropouts will help reduce variance.

Benchmark

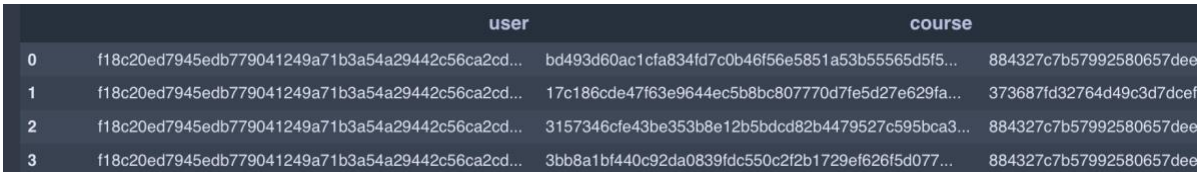
As a benchmark, the data is divided into a testing set and a training set. The model never sees the test set so it cannot train against it creating an isolated set of data to test against. We are also going to take a portion of the training set to create a validation set. The validation set will help reduce variance by monitoring the training error rate vs the validation error rate. Overall, the error will be calculated using RMSE. We are shooting for a RMSE rate of .5 or less.

III. Methodology

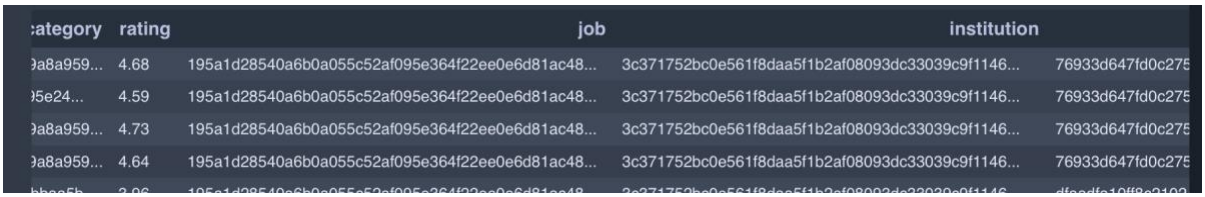
Data Preprocessing

1. We will load in all the data into a Pandas Dataframe for easy manipulation.
2. Based on our analysis we determined there are some ratings that are invalid. We removed all the rows that have a rating above 5 as that is an error.
3. We then defined are categorical features and our continuous features and labeled them in the dataframe.
4. We then converted all our categorical variables into contiguous integers and created a look up table.

Sample Data Pre Conversion:



	user	course	
0	f18c20ed7945edb779041249a71b3a54a29442c56ca2cd...	bd493d60ac1cfa834fd7c0b46f56e5851a53b55565d5f5...	884327c7b57992580657dee
1	f18c20ed7945edb779041249a71b3a54a29442c56ca2cd...	17c186cde47f63e9644ec5b8bc807770d7fe5d27e629fa...	373687fd32764d49c3d7dcef
2	f18c20ed7945edb779041249a71b3a54a29442c56ca2cd...	3157346cfe43be353b8e12b5bdcd82b4479527c595bca3...	884327c7b57992580657dee
3	f18c20ed7945edb779041249a71b3a54a29442c56ca2cd...	3bb8a1bf440c92da0839fdc550c2f2b1729ef626f5d077...	884327c7b57992580657dee



:ategory	rating	job	institution	
)a8a959...	4.68	195a1d28540a6b0a055c52af095e364f22ee0e6d81ac48...	3c371752bc0e561f8daa5f1b2af08093dc33039c9f1146...	76933d647fd0c275
)5e24...	4.59	195a1d28540a6b0a055c52af095e364f22ee0e6d81ac48...	3c371752bc0e561f8daa5f1b2af08093dc33039c9f1146...	76933d647fd0c275
)a8a959...	4.73	195a1d28540a6b0a055c52af095e364f22ee0e6d81ac48...	3c371752bc0e561f8daa5f1b2af08093dc33039c9f1146...	76933d647fd0c275
)a8a959...	4.64	195a1d28540a6b0a055c52af095e364f22ee0e6d81ac48...	3c371752bc0e561f8daa5f1b2af08093dc33039c9f1146...	76933d647fd0c275

Sample Data Post Conversion:

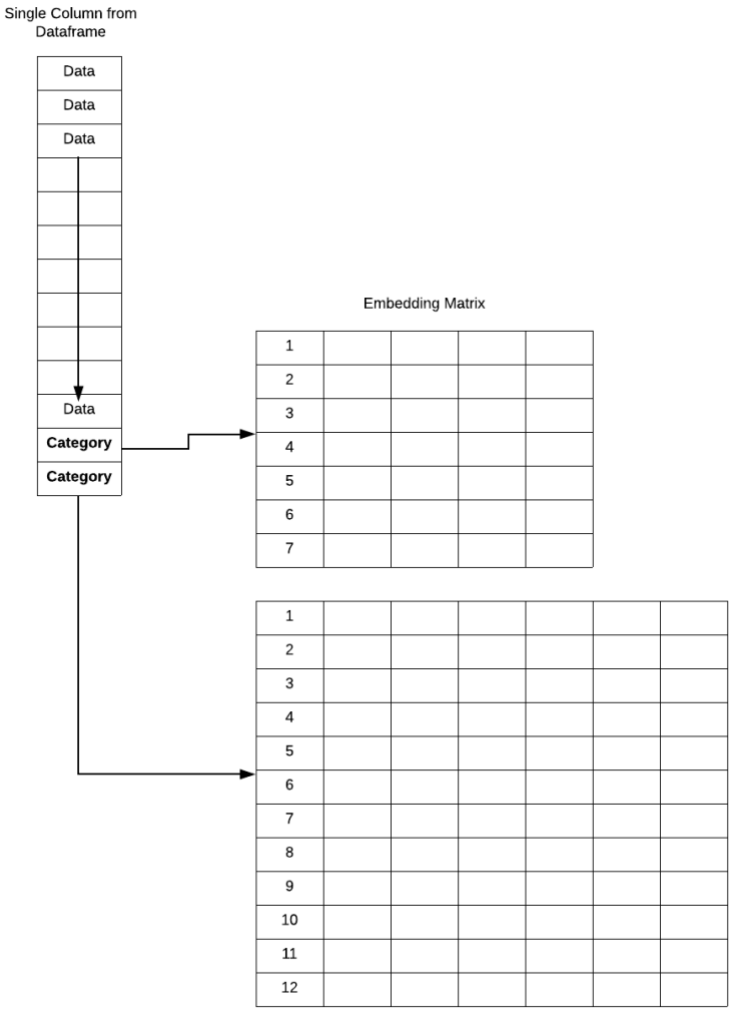
	user	course	category	job	institution	state
114672	41230	191	7	6	11	50
19331	13460	174	7	12	0	30
159684	39731	128	7	24	0	23
47069	39972	126	7	6	0	30
92441	22612	189	0	28	0	45
90730	3043	203	7	6	14	24
79111	36713	28	7	6	14	57
115156	44617	189	0	28	11	45
83933	7965	43	8	6	0	52
18342	2325	128	7	6	9	30

Implementation

Once the data is cleaned we are ready to start building and training a model. An initial model is made passing in the validation ids and the batch size.

We then have to create the embedding matrix size to pass into the learner. An embedding matrix is a “mini” matrix that is representative of a categorical variable. Instead of trying to use a categorical value directly in the layer, you use an embedding matrix to create weight instead. Each categorical feature will have its own matrix of weights. We will use a lookup table to find the value of the categorical feature. The number of rows should be equal to the unique value count of the feature. In our case we did value count + 1 to leave a spot for an ‘unknown’ value. The number of columns can vary, we picked a pattern suggested by the folks over at Fast.ai. Columns would be equal to half the rows or 50, whichever is less.

Visually, what an embedding matrix looks like:



Once the embedding matrix values are calculated we can build the initial learner. This is the first opportunity to significantly alter how the model behaves. At this state we pass in some critical values:

1. Embedding matrix drop out rate
2. Layer sizes
3. Layer drop rates
4. Optimizer (We used Adam)
5. Weight Decay
6. Output Size (in our case its 1, we just need a single rating returned)
7. Output Range (in our case it is 0 – 5)

Model Overview:

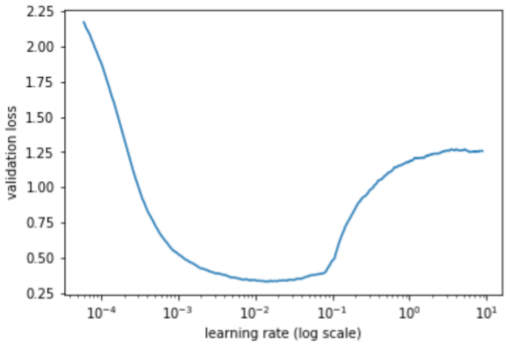
The model structure will remain the same, to optimize, we will just play with the dropouts, layer sizes, etc.

After the model is built, it is time to select a learning rate. The graph below shows how the learning changes and lets us pick and optimal rate that is not too slow, but also does not make huge jumps. The model is using the Adam optimizer. The optimizer is used to update weights during training. We search for a good learning rate and get the below results.

```
In [210]: #lets find a learning rate
          m.lr_find()

          HBox(children=(IntProgress(value=0, description='Epoch', max=1), HTML(value='')))
          epoch      trn_loss   val_loss
              0      1.266841   1.272447

In [211]: m.sched.plot(100)
```




```
In [212]: lr = 1e-3
```

We select 1e-3 as it is near the bottom of the curve. As you get close to and pass 1e-2 it starts to flatten down then gets significantly worse.

Now we have a learning rate, it is time to start fitting the model. Here we will use the learning rate from above and also select the number of Epochs (how many cycles to run) and if want a restart and a weight decay.

Refinement

Initially, I started with big layers (500,1000) and really low dropout rates (.01 and .001 respectively). This caused overfitting (the model performed well on the training set but performed poorly on the validation set). I gradually reduced the layer sizes, increased the dropout, added a weight decay and added a scheduler to reset the learning rate after a certain number of epochs (a single pass through the entire data set). I kept iterating through this cycle until we hit a sweet spot. Our training error rate is a little below the validation set and we were still able to hit our target RMSE.

```
In [19]:
# GPU
# lets fit the model, 10 epochs with restarts
m.fit(lr, 10, cycle_len=1, wds=1e-3)
```

```
HBox(children=(IntProgress(value=0, description='Epoch', max=10), HTML(value='')))

epoch      trn_loss   val_loss
    0      0.268506   0.255554
    1      0.234577   0.226035
    2      0.198637   0.223214
    3      0.194547   0.223632
    4      0.195265   0.22416
    5      0.183143   0.224411
    6      0.186299   0.224348
    7      0.182715   0.222204
    8      0.182431   0.221946
    9      0.176393   0.223074

[array([0.22307])]
```

IV. Results

Model Evaluation and Validation

The model is performing right around the target we were trying to reach. We used a validation set to ensure the model was not overfitting. The model was tested on new data that came in after we pulled the initial data set. We used these to test the model and the results came within specs even though the input can be sparse at times. We wanted a RMSE of .5 or less and the final model hit a RMSE of .47.

For additional validation, we went back to the live database and pulled in records that were inserted after the initial data was pulled. With this unknown data, the model resulted in a RMSE of .51. This performed slightly worse than the initial trained model, but still acceptable for being unknown data.

```
In [45]:
post_test.head(10)
```

	user	course	category	rating	job	institution	state	prating
0	38762	71	8	4.05	29	0	41	3.707968
1	31857	37	8	3.23	6	0	41	3.636474
2	24228	37	8	3.77	6	0	41	3.811103
3	12717	44	7	4.59	6	9	57	4.475251
4	1599	63	9	3.73	6	14	39	3.703550
5	20543	186	9	2.82	6	9	41	3.875622
6	20543	177	7	3.73	6	9	41	3.924339
7	20543	38	7	3.73	6	9	41	3.895814
8	36724	203	7	4.77	6	2	41	4.200868
9	16321	174	7	3.73	6	14	41	3.872567

```
In [46]:
rms_pt = sqrt(mean_squared_error(post_test['rating'], post_test['prating']))
```

```
In [47]:
rms_pt
```

```
0.5132613564108592
```

Justification

This model is able to hit our target RMSE. It did not perform significantly better, but it also did not perform significantly worse. The model is fast to train, so it will make it easy to add new data moving forward. The model is also very fast at making predictions, so it can be linked up to a RESTful API and return results in near real-time.

V. Conclusion

Reflection

The project was straight forward. Once the data is cleaned and the embedding features are identified, you can jump right into building the model. At that point, you just need to play with the hyperparameters to tune the model. This process can be used for another dataset with minimal alterations as long as the features used for the embeddings are categorical. One key take away of this project is the use of a validation set. This really helped to identify overfitting. The most difficult part of this project was getting the model to load in the API for reuse.

Since the model is static, it is critical to keep feeding it training data as it comes in. This creates in interesting problem. We would need to create a job to pull data at specified intervals and train the existing model. Also, when new data comes in that is completely unknown, the matrix needs to be big enough to adjust for this. I actually ran into this and had to rebuild the model when I went to test it on new data.

Improvement

One way this model can be improved is loading the saved model to be used in platform agnostic systems. Saving and loading the model is very clunky and required some “creative” solutions. We had to save out all the model data using serialization in order to recreate the model on the API side. This process is not particularly difficult, but it does add extra overhead. Also, on the API side, since we are recreating the model infrastructure then importing the data we had to load all the necessary python modules. Ideally, we should be able to load and use the model without these. PyTorch is being combined with Caffe2 in the next release (1.0). This will offer a production ready solution to launch and use the model.