DEEP BEERS: Playing with Deep Recommendation Engines Using Keras

Pierre Gutierrez
Jan 17, 2018 · 11 min read

A nice image of BapBap beer. Shameless advertising for a Parisian brewery I know. Now, let's start.

This is Part 1 of a series of articles, "Deep Beers, Playing with deep

recommendation engine using Keras”

Part 2 | Part 3 | More from Data in the Trenches

Explicit recommendation engines

Full disclaimer, I am a bit of a data science beer geek. A few years ago, I scraped with my friend @alexvanacker a beer rating website. I wanted at the time to test different recommendation algorithms.

More recently, I was advised to follow this excellent class by Charles Ollion (Heuritech) and Olivier Grisel, to learn more about some specific aspects of deep learning. When I came across the second lab on factorisation machine and deep recommendations, I remembered my old beer dataset and decided to give it a shot.

In the following blog post, I’ll discuss the different experiments I was able to run using keras. My code is more than heavily inspired by the class, so don’t get alarmed if you detect obvious copy-paste.

Because there were too many subjects I wanted to cover, this project will be posted in several parts:

- In Part 1 (here we are), I will explore the data and create our first two explicit recommendation engines using Keras. The first one will rely on a simple matrix factorization while the second one will be based on a deeper architecture.

- In Part 2, I will discuss if we can make sense of the embeddings (representations) computed by the models. For example, can we find clusters of beer origin or style ?
- In Part 3, I will explore the possible model architectures and try to make use of metadata.
- If I get the time someday, I will write on implicit recommender systems architectures as well as how to efficiently serve the recommendations.

I did the project using Python, pandas, scikit-learn and Keras. Throughout the blog post, I'll share the Keras code when it seems appropriate. To make it easier to follow, you can also check out the complete Ipython notebook.

The Data

Ratings

We scraped a beer rating website and ended up with several datasets. The most important one contains the reviews (scores given by the users) and is composed of:

- "beer_id": a unique id for a beer
- "user_id": a unique id for a user
- "score": the review score, between 1 and 5
- "date": the timestamp of when the user posted the review

- “review”: a review of the beer. This is optional, and left empty 75 % of the time. I won’t use this information in this blog post though it would be interesting to integrate it somehow.

In total, I got 4.563.152 ratings. Here is a head of this dataset.

	beer_id	user_id	score	date	review
15	0	15	4.10	2014-08-24 00:00:00	I like this brewer and their Oyster Stout is a great session stout. Perfect last beer of the ev...
24	0	24	4.00	2014-08-17 00:00:00	Pours a rich and dark. If it's not completely black it may as well be because there are no hues ...
67	0	67	3.89	2014-07-11 00:00:00	Appearance - Extremely dark, practically black . Okay head that lasted for a while and would slo...
69	0	69	4.21	2014-07-10 00:00:00	Vintage 2013 Appearance: It has a nice jet black color to it. It has a nice thick creamy tan hea...
72	0	72	3.63	2014-07-09 00:00:00	Appearance- poured from a glass into a guiness style glass. this shit is DARK. not much head, la...
76	0	76	4.13	2014-07-04 00:00:00	A - poured a one finger thick coffee-colored head into snifter that left a thin ring throughout....
96	0	96	3.51	2014-06-22 00:00:00	A: Dark brown to black. Forms a small head that dissappears to nothing. Swirling the beer in my ...
155	0	155	3.76	2014-04-30 00:00:00	2013 Vintage poured into DFH snifter. A - inky black with mocha head S - lots of dark roasted m...
167	0	167	4.43	2014-04-05 00:00:00	A - 4 - Nice dark brown/black beer with a big mocha head on it, leaves nice lacing. S - 4.25 - ...
179	0	179	4.15	2014-03-25 00:00:00	Vintage from 2010. 12oz poured into an oversized tulip. A - Black but not opaque. Small cap of t...

A view of the rating table

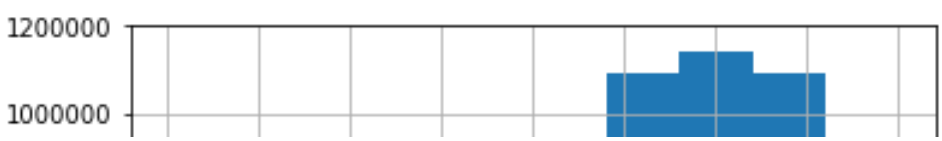
You can notice that most scores seem to be between 3.75 and 4.25. We can get an idea of the ratings distribution using pandas:

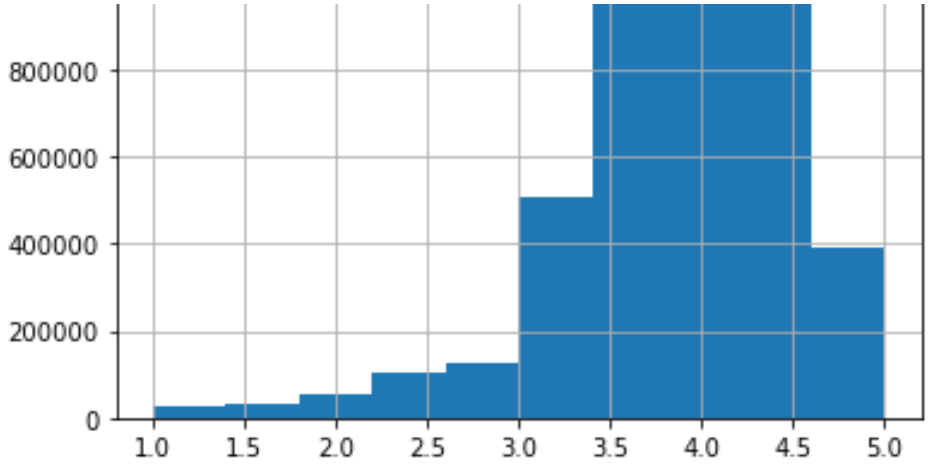
```
beer_reviews["score"].describe()
```

```
count    4.563152e+06
mean     3.852421e+00
std      6.533483e-01
min      1.000000e+00
25%      3.500000e+00
50%      4.000000e+00
75%      4.250000e+00
max      5.000000e+00
Name: score, dtype: float64
```

Ratings statistics

And by plotting the distribution:





Beer ratings distribution

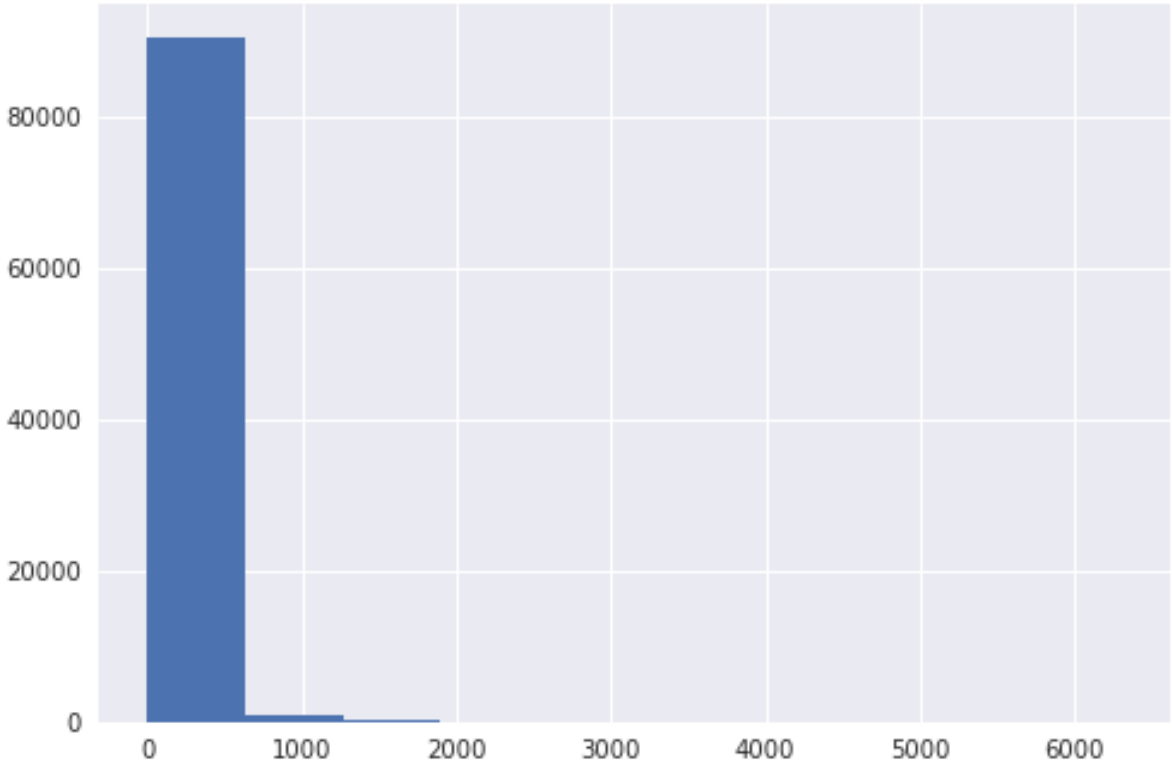
The median is 4. This is very important because it means the ratings are skewed toward high values. This is a common bias in internet ratings where people tend to rate items or movies that they liked, and rarely spend time to comment something they dislike or are indifferent to (unless they are haters, of course). This distribution shape will have a big impact on the results of our recommendation engines — ’ll discuss this further later on.

In the review dataset, we have 91,645 distinct users and 78,518 beers. This means that on average, each user has rated around 50 beers, which seems quite high. So I looked for answers to the questions: How many ratings do we have per beer? Per user ? What are the corresponding distributions?

```
users_nb = beer_reviews['user_id'].value_counts().reset_index()
users_nb.columns= ['user_id','nb_lines']
users_nb['nb_lines'].describe()
```

```
count    91645.000000
mean        49.791609
std        173.584107
min          1.000000
25%          1.000000
50%          4.000000
75%         23.000000
max       6311.000000
Name: nb_lines, dtype: float64
```

Distribution of number of ratings per users



Distribution of number of ratings per users

As expected, the distribution is very skewed. With 50 percent of people having done no more than four reviews… and one person going crazy with more than 6,000 ratings (or is it a bot?)! This has the following implications: we will have few beers to characterise most users, but for at least 25 percent of the users, we will have at least 23 ratings. This is probably enough information to start generating good recommendations.

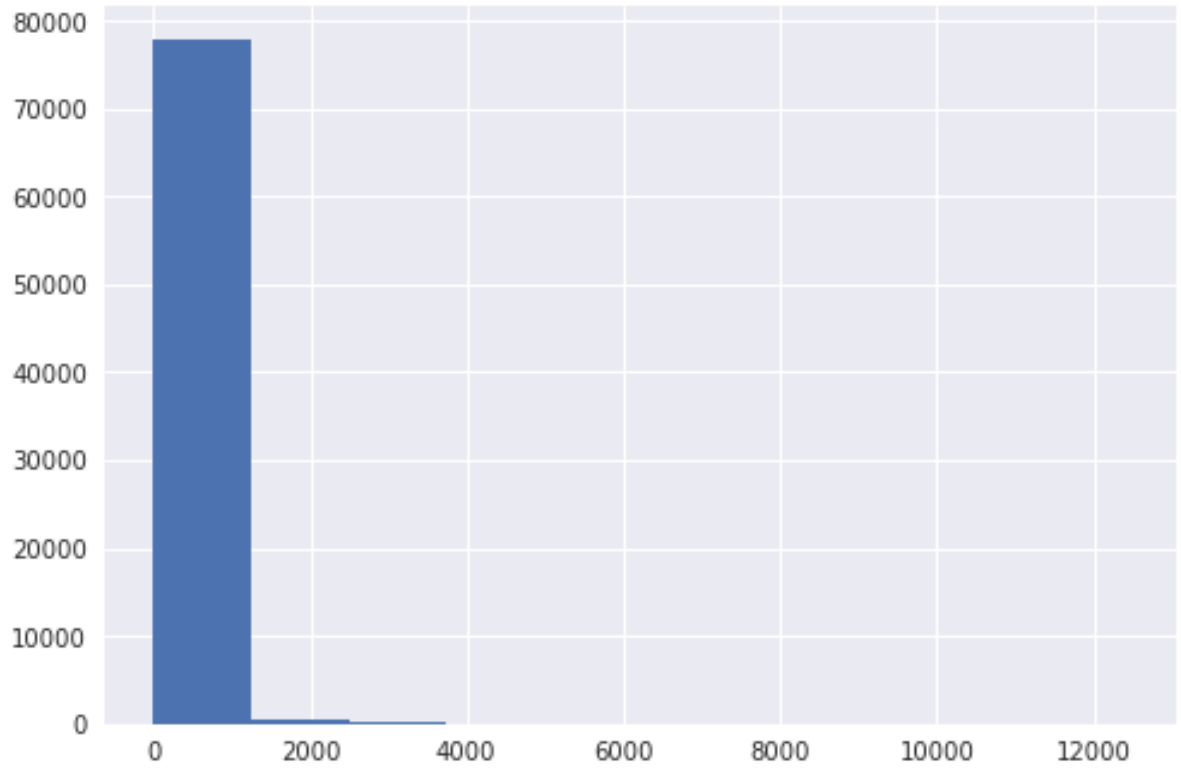
Now let's have a look at the beers:

```
beers_nb = beer_reviews['old_id'].value_counts().reset_index()
beers_nb.columns= ['old_id','nb_lines']
beers_nb['nb_lines'].describe()
```

```
count    78518.000000
mean        58.115999
std        313.841229
min          1.000000
25%          2.000000
```

```
25%            2.000000
50%            5.000000
75%           18.000000
max        12450.000000
Name: nb_lines, dtype: float64
```

Distribution of the number of ratings per beer



Distribution of the number of ratings per beers

Again, the distribution is very skewed with 50 % of the beers having 5 ratings or less. Even worse, 75% of beers have less than 18 ratings. Though this distribution is expected for users, I assumed the distribution of beer counts to be less skewed, with a good number of well-known beers.

Let's now have a look at the metadata.

Metadata

I scraped information about each beer as well: the brewery with which they are associated as well as user data. With this I can extend the

analysis by adding some "beer geek knowledge." Let's have a look at the beers that received the most ratings for example.

```
all_info[['name','brewery_country','style','nb_lines']].sort('nb_lines',ascending=False).head(10)
```

	name	brewery_country	style	nb_lines
67170	90 Minute IPA	United States	American Double / Imperial IPA	12450.0
59285	Founders Breakfast Stout	United States	American Double / Imperial Stout	11694.0
52528	Two Hearted Ale	United States	American IPA	10698.0
67171	Pliny The Elder	United States	American Double / Imperial IPA	10467.0
67172	Hopslam Ale	United States	American Double / Imperial IPA	9825.0
42703	Old Rasputin Russian Imperial Stout	United States	Russian Imperial Stout	9620.0
67173	Stone Ruination IPA	United States	American Double / Imperial IPA	9400.0
52530	Sculpin IPA	United States	American IPA	9200.0
52529	60 Minute IPA	United States	American IPA	9017.0
7265	Sierra Nevada Pale Ale	United States	American Pale Ale (APA)	8650.0

Most common beers in the ratings dataset

Though I know most of these beers, some of them do not ring a bell at all. This is because of two biases:

- Most of the users probably come from the United States, which explains why all of the most rated beers above are from there. I am on the other side, from France. In France, we can find some of these beers (like the Sierra, Stone or Founder). However, only in dedicated shops. Thus, the beers are more likely purchased by a specific population, which leads us to the second bias.
- Most of the people rating beers on this website have a "beer geek" profile. Since they will rate mostly beers they liked, we can expect the most rated beer to be US common quality craft beers. To me, that's 90 minute IPA or Sierra Nevada.

We can verify the first assumption by looking at the user metadata.

```
users_infos = pd.read_csv('/data/pgutierrez/beer/users.csv.gz', sep=',')
users_infos = users_infos.fillna('no_data')
```

```
country_count = users_infos.location.value_counts().reset_index()
country_count.columns = ["location","nb_users"]
country_count['perc_users'] = country_count['nb_users'].astype(float)/country_count['nb_users'].sum()
country_count['cum_perc_users'] = country_count['perc_users'].cumsum()
country_count.head(15)
```

	location	nb_users	perc_users	cum_perc_users
0	no_data	16269	0.186059	0.186059
1	California	7016	0.080238	0.266297
2	Pennsylvania	5488	0.062763	0.329060
3	New York	4247	0.048570	0.377630
4	Illinois	4220	0.048262	0.425892
5	Massachusetts	3854	0.044076	0.469968
6	Texas	3342	0.038220	0.508188
7	Ohio	2894	0.033097	0.541285
8	Florida	2652	0.030329	0.571615
9	Michigan	2394	0.027379	0.598994
10	New Jersey	2262	0.025869	0.624863
11	Virginia	2125	0.024302	0.649165
12	North Carolina	2031	0.023227	0.672392
13	Minnesota	1792	0.020494	0.692887
14	Indiana	1575	0.018012	0.710899

Distribution of the users countries

So we have around 20% of unknown locations and more than 50% of American users. In fact if we look at the non US entries in the list we get:

- The state of Ontario(Canada) as the first entry with 718 users
- United Kingdom and Australia as the two first countries with respectively 348 and 306 users
- France arrives at the 69th position with 62 users. Sad.

These metadata enable us to understand biases that appear in the model, but they can also be used in the recommendation system (see Part 3). Finally, let's have a look at the bottom and top rated beers.

```
all_info[all_info['nb_lines']>=500][["name",'style','avg_rating']].sort('avg_rating').head(5)
```

	name	style	avg_rating
491	Natural Light	Light Lager	1.673766

74533	Keystone Ice	American Adjunct Lager	1.699634
74522	Natural Ice	American Adjunct Lager	1.736900
504	Milwaukee's Best Light	Light Lager	1.785730
502	Miller 64	Light Lager	1.788587

Bottom rated beers (with at least 500 ratings)

I have to say, I never tried any of these beers. You can't really find light beers in France but this seems to confirm both biases. All beers come from the US, and light beers will be rated low here because they do not target a beer geek audience. Let's have a look at the top ones.

```
all_info[all_info['nb_lines']>=500][["name",'style','avg_rating']].sort('avg_rating',ascending=False).head(5)
```

	name	style	avg_rating
59364	Hunahpu's Imperial Stout - Double Barrel Aged	American Double / Imperial Stout	4.780888
67174	Heady Topper	American Double / Imperial IPA	4.731133
42741	Bourbon Barrel Aged Vanilla Bean Dark Lord	Russian Imperial Stout	4.726511
59375	Barrel-Aged Abraxas	American Double / Imperial Stout	4.716880
67345	King Sue	American Double / Imperial IPA	4.704760

Top rated beers (with at least 500 ratings)

I have no idea what these beers are. This may be because the best beers (according to our ratings) are probably craft beers and hence less well-spread. However, we see that the best rated beers are all imperial (Stout or IPA), which is — again — expected.

Our first Explicit Recommendation Engine

Explicit or Implicit recsys

You can learn more about the different types of neural recommender systems as well as explicit vs implicit recommendation engines in the excellent slides by Ollion and Grisel.

Basically, explicit feedback is when your users voluntarily give you information on what they like and dislike. In our case, we have explicit beer ratings ranging from one to five. Other examples would be the reviews you can see on Amazon (see their dataset) or on MovieLens (a well known dataset for recommendation engines).

However, in many cases, we don't have this information or it is not rich enough. Thus, we need to extract implicit information from the interaction between the system and the user. For example, when you type a Google query, you do not explicitly notify Google of the search result pertinence. But you do click on one or more of the proposed links and spend a certain amount of time on the corresponding pages. That's implicit feedback that Google can take into account to increase its search engine performance. Following the same idea, most people won't rate their purchase on Amazon, giving no explicit feedback. However, Amazon can record all products searched, viewed, or purchased to infer customer tastes.

Back to our beer recommendation engine. The implicit information could be anything, like the website pages visited, time spent on these pages, etc. However, in the data, I only have ratings. The implicit information can also simply be the list of beers people reviewed (or drank), i.e., for all possible tuple (user,item) 1 for beer drank, else 0. Though this information seems less rich than a rating, it helps the system understand what people do not drink and might dislike.

In what follows, I will start with the explicit recommendation engine. It basically boils down to a regression problem where I try to predict the ratings for each user. This means that I will recommend a beer to a user that (s)he is likely to rate highly upon drinking.

To evaluate the model, I randomly split the data into training and test sets. Note that I could do things more properly by splitting the user ratings based on increasing timestamps — this would make it possible to predict the next beers a user will drink. But I'll leave this aside for now.

Matrix Factorization

Now that I've described the data and the method, let's create a simple model using Keras. For the algorithm in Keras to work, I had to remap all beers and user ids to an integer between 0 and either the total number of users or the total number of beers.

```
users = beer_reviews.user_id.unique()
user_map = {i:val for i,val in enumerate(users)}
inverse_user_map = {val:i for i,val in enumerate(users)}
beers = beer_reviews.beer_id.unique()
beer_map = {i:val for i,val in enumerate(beers)}
inverse_beer_map = {val:i for i,val in enumerate(beers)}

beer_reviews["user_id"] = beer_reviews["user_id"].map(inverse_user_map
beer_reviews["old_id"] = beer_reviews["beer_id"] # copying for join wi
beer_reviews["beer_id"] = beer_reviews["beer_id"].map(inverse_beer_map
```

idea is to project beers and users in a common latent space.

In keras, we can define our model this way:

```
user_id_input = Input(shape=[1], name='user')
item_id_input = Input(shape=[1], name='item')

embedding_size = 30
user_embedding = Embedding(output_dim=embedding_size, input_dim=users.
                           input_length=1, name='user_embedding')(user
item_embedding = Embedding(output_dim=embedding_size, input_dim=beers.
                           input_length=1, name='item_embedding')(item
```

```
                                    input_length=1, name='item_embedding')(item
9
10  user_vecs = Reshape([embedding_size])(user_embedding)
11  item_vecs = Reshape([embedding_size])(item_embedding)
12
13  y = Dot(1, normalize=False)([user_vecs, item_vecs])
14
15  model = Model(inputs=[user_id_input, item_id_input], outputs=y)
16
17  model.compile(loss='mse',
```

prediction. When I train the model, the embeddings parameters are learned, which gives me a latent representation.

In the code:

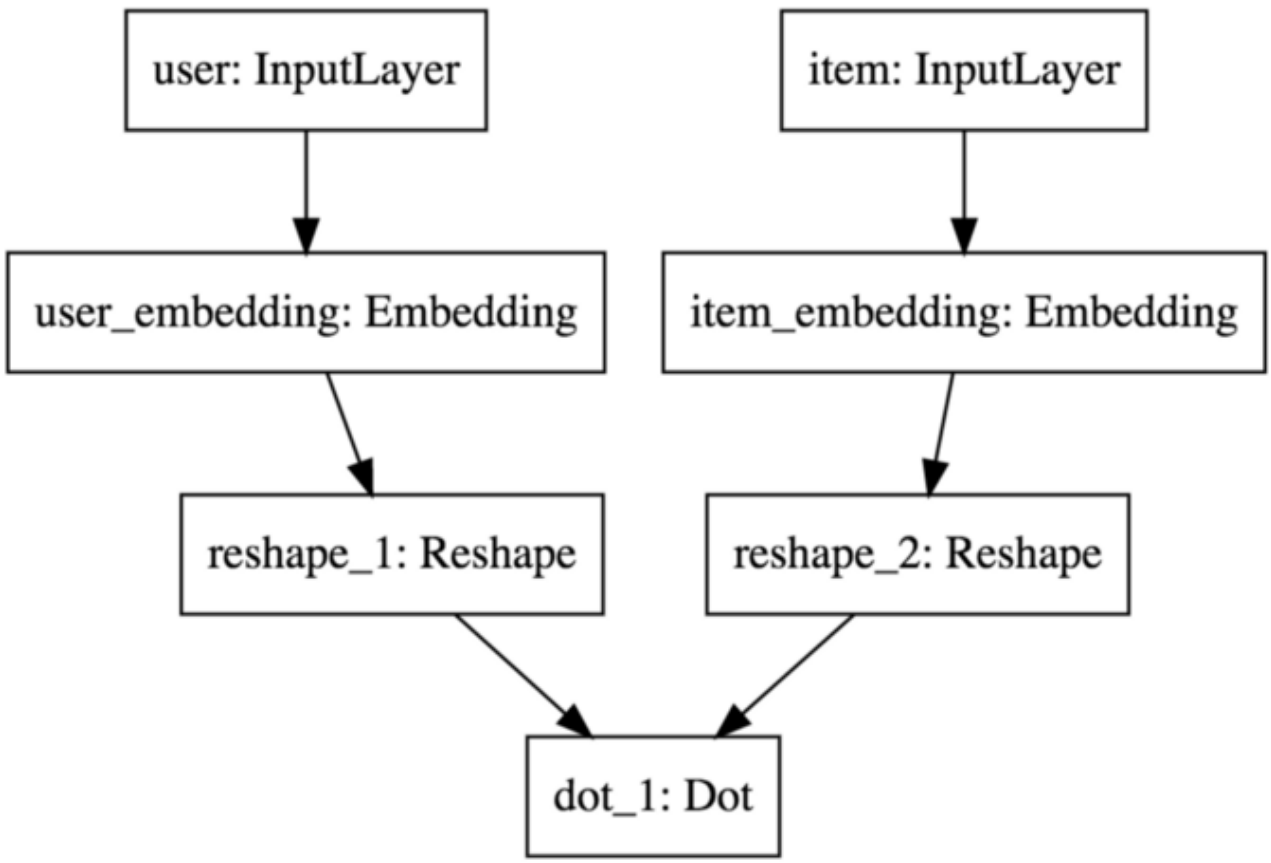
- Line 1 and 2 we declare inputs to the model.
- Line 4 we define the size of embeddings as a parameter.
- Line 5 to 8 we apply an embedding layer to both beer and user inputs.
- Line 10 and 11 reshape from shape (batch_size, input_length,embedding_size) to (batch_size, embedding_size). A Flatten layer can be used as well.
- Line 13 declare our output as being the dot product between the two embeddings.
- We declare a model (Line 15) that takes beers and users as input and output y, our prediction.

- This model need to be compiled with the right loss and optimizer (line 17).

Notice that keras provide a way to display each model as svg or save them as images.

```
1  from IPython.display import SVG
2  from keras.utils.vis_utils import model_to_dot
3  from keras.utils import plot_model
4  import pydot
5  SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

Which gives us, the very readable



schema of our matrix factorisation model

To train the model, we simply need to call the model's fit method (because our dataset fits in memory, else we could have used

fit_generator).

The code from lines 1 to 8 and 17 to 19 allows me to save histories and best models (with Keras callbacks). Notice also that I use an internal random cross validation scheme (the validation_split=0.1 parameter). This is because I'm going to grid search several parameters and architectures. Since I'm exploring and going to follow the most promising leads, I will be prone to manually overfit. The test set will be kept to verify the quality of recommendations at the end of part 3.

Here's what I get when I plot the training and evaluation MSE loss:

Training and evaluation loss for the matrix factorisation model

The training loss stabilises around 0.15. After 10 epochs, the model starts overfitting — the best MSE validation loss is around 0.465. A quick grid search on the embedding sizes gives me:

Training and evaluation loss for the compared models

This shows that choosing large values of embedding sizes leads to overfitting. Hence, for most of the experimentations I'll keep this embedding size of 10 (giving me around 0.42 validation mse)

Now, let's go deeper.

Going Deeper

Using the architecture above, I'm trying to predict a rating by performing a dot product. The dot product enables the model to perform the matrix factorisation approach. But we can relax this dot structure by instead using whatever network we want to predict the rating. For example, I can use a concatenate layer followed by a dense layer. This means that instead of relying on a simple dot product, the network can find the way it wants to combine the information itself.

With a two layer deep neural network, this gives us using keras:

This is exactly the same code as before, except I changed the Dot layer by a Concatenate layer followed by several Dense ones. Here is the model representation that can be compared to the previous one:

schema of our dense model

When training the model, I get the following chart:

Training and evaluation loss for the deep model

It's worth noting that though my training error goes down with a rather common shape, the validation error plateaus after just one or two iterations. But did I get better? Here's a graph of the comparison with the previous best model:

Training and evaluation loss for the compared models

Obviously, the performance got way better! From 0.42 to 0.20 validation loss. We can also notice the following points:

- We converge really fast to the best model. After one or two epochs, the model starts overfitting or at least the validation loss does not seem to go down anymore.
- When comparing to the previous model, we almost manage to match the training error with our validation error! This may mean that we are close to reaching the best possible validation error.
- I actually added some dropout in this architecture. It gives me a few extra points.

I can also grid search around this architecture. What happens if I add another layer on top of the first one? What happens if I decrease the embedding size? (Modifications commented in the code above.)

Train and eval loss for the compared models.

Adding a layer does not help much. The training error is quite similar, even if I may get a little improvement on the validation loss. In the opposite direction, simplifying the model by reducing embedding size worsens the training and validation errors.

That's it! I showed how to create our first matrix factorisation and deep models. Now before going further in the architecture grid search, you'll get a grasp of what the model does by looking at our generated embeddings. This will be covered Part 2 of this blog post series.

Thanks to Alivia Smith, Léo Dreyfus-Schmidt, Simon Marmorat, and Alex Vanacker.

Machine Learning Recommendation System Recommender Systems

Deep Learning Keras