Recommender system based on latent classes

Gaëlle Guillou, Marie Laval

Introduction

Recommender systems have been a major hype in data science recently with the growth of ad-targeting (Criteo, millemercis), media websites and services (Netflix, Google News, Spotify) for whom recommendation is a major revenue-driver. Recommender systems, who have been around since the 1990's, use in general content-based or collaborative approaches. Using movies as examples, content-based filtering is like saying: if you enjoyed Star Trek, you should like Star Wars, because they are both Science-Fiction movies. Collaborative approaches have proved themselves to be quite efficient in general: you empirically consider that if you have watched the Silence of the Lambs and loved it you should like the Shawshank Redemption even though it's not the same kind of movie, because most users who like one movie liked the other.

In this paper, we present the Probabilistic Semantic Latent Indexing approach to recommender systems, a collaborative system. Introduced by Hoffman in 1999, the PLSI approach introduces latent classes to recommend items to users. Users, like items, can belong to several latent classes. Those can be better apprehended if we consider them as groups gathering fans of similar movies, for example. To test our algorithm, we will be using the Movielens dataset from the Grouplens website. It is an aggregation of about twenty million ratings applied to 27,000 movies by 138,000 users.

Description of the PLSI algorithm

Background: assumptions, introducing latent classes (Hoffman and al., 1999)

For the sake of simplicity and realism - considering privacy regulations -, we assume that we have no external knowledge about the objects or the users.

Now, let's dive into the statistical model presented by Hoffman in 1999. We introduce three random variables: users u, objects s and a latent class variable z.The main statistical assumption we make is that u and s are independent conditioned on z. Therefore, we have:

$$P(u, s) = \sum_{z \in \mathcal{Z}} P(z)P(u \mid z)P(s \mid z)$$

This yields a most relevant result for our recommender system:

$$P(s \mid u) = \sum_{z \in \mathcal{Z}} P(z \mid u)P(s \mid z)$$

As Hoffman underlines it himself, there is no assumption that persons form groups or that objects form clusters, which allows for great flexibility regarding our recommendations.

Expectation-Maximization algorithm (Dempster and al., 1977)

For maximizing likelihood estimation in latent variable models, it is usual to use the Expectation Maximization algorithm.

By maximizing the log likelihood, we are minimizing the empirical logarithmic loss, defined here:

$$L(\theta) = -\frac{1}{T}\sum_{t=1}^{T} log\big(p(s \mid u;\theta)\big)$$

The EM algorithm consists of two steps:

- Expectation (E) step: computing posterior probabilities for latent variables, based on current estimates,
- Maximization (M) step: updating parameters for posterior probabilities.

As indicated, we will be using it in our own implementation of PLSI.

Our implementation of the PLSI algorithm, inspired by The Google News implementation (Das and al., 2007)

Our implementation is largely based on the Google News MapReduce implementation, as presented in Das and al., 2007.

The PLSI algorithm that we will implement here is based on Das description of the Google News recommendation system. The algorithm is based on the following :

- The relationship between users and movies is learned by modeling the joint distribution of users and items as a mixture distribution,
- To capture this relationship, we introduce the hidden variable z (latent variable), that can be seen as a representation of user communities (same preferences) and movie communities (sames genres).

All in all, we try to compute the following probability for each (user, movie) couple : $p(s \mid u) = \sum p(s \mid z)p(z \mid u)$, which is the probability for a given user to see a given movie. This is obtained by summing for each community the probability for a movie s to be seen given a community z times the probability to be in the community z given a user u.

Initialisation

E-STEP - Compute $q(z \mid (u, s))$: the probability that the (user, movie) couple belongs to the class z

This step is first initialized at random :

- To each couple (u, s), assign each possible community through a cartesian product Example: with number of classes = 2, the lines (Marie, Star Wars) and (Gaëlle, Matrix) will give (Marie, Star Wars, 1), (Marie, Star Wars, 2), (Gaëlle, Matrix, 1), (Gaëlle, Matrix, 2)
- To each line, assign a random probability. This random probability corresponds to $q^*(z \mid (u, s))$. For example if I have (Marie, Star Wars, 1, 0.3), then the probability that the couple (Marie, Star Wars) is in class 1 is 0.3.
- Initialise the log-likelihood at 0:

LogLik = 0

Iteration

M-STEP - Compute $p(s \mid z)$ and $p(z \mid u)$ based on $q(z \mid (u, s))$

- Compute $p(s \mid z)$: sum the probas associated to every couple (s, z) and divide it by the sum of probas associated to this z
- Compute $p(z \mid u)$: sum the probas associated to every couple (u, z) and divide by the sum of probas associated to this u

E-STEP - Compute the new $q(z \mid (u,s)) = \frac{p(s|z)p(z|u)}{\sum p(s|z)p(z|u)}$

- For each (u,s,z), compute $p(s \mid z) * p(z \mid u)$
- For each (u,s), compute $\sum_z p(s \mid z) * p(z \mid u)$ which is $p(s \mid u)$
- For each (u,s,z), compute $\frac{p(s|z)p(z|u)}{\sum p(s|z)p(z|u)}$ which is the new $q(z \mid (u,s))$

Update LogLik

$$\sum_{(s,u)} \Big(log\big(\sum_z p(s \mid z) * p(z \mid u) \big) \Big) = \sum_{(s,u)} log\big(p(s \mid u)\big)$$

Iterate again until LogLik converges : this means that it has reached its maximum and we have found the best estimation of $p(z \mid u)$ and $p(s \mid z)$.

We can now predict the probability that Gaëlle will watch Star Wars :

$$p(StarWars \mid Gaëlle) = p(1 \mid Gaëlle) * p(StarWars \mid 1) + p(2 \mid Gaëlle) * p(StarWars \mid 2)$$

Discussion: choosing the number of latent classes and the number of iterations

Since we haven't taken into account scalability and accuracy metrics, this discussion is bound to the understandability of the algorithm and likelihood considerations.

The number of iterations represents the number of times we need to run our loop for the EM algorithm to reach the convergence of the log likelihood. In our first attempts, as seen below, it became evident that around 15 iterations made the cut. We also could have defined a simple convergence criteria (`python while prev_loglik - current loglik > cutoff_value`) that would have worked on our particular data. However, for the purpose of explorations, we set the number of iterations at 20.

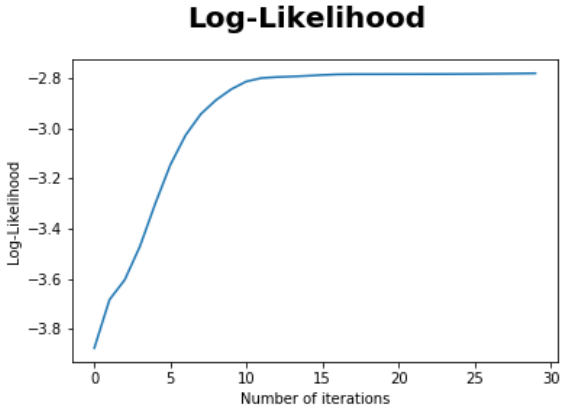

Figure 1: Convergence of loglikelihood for 3 clusters and 30 iterations

The same conclusion can be drawn from the more complete Figure 2. Around 15 iterations, we reach a point of convergence.

As for choosing the number of clusters Z, we of course have a strong tradeoff between the efficiency of our model, because choosing more clusters allows for more granularity in our recommendations, and computing complexity. On the above graph, we can see that the marginal increase in log likelihood reduces with the amount of clusters we add. We also have to take into account that imagining 20 classes for instance stands complicated and strongly reduces our probabilities (because it reduces the probability to belong to a particular class). 10 clusters seem like an ideal middle ground.

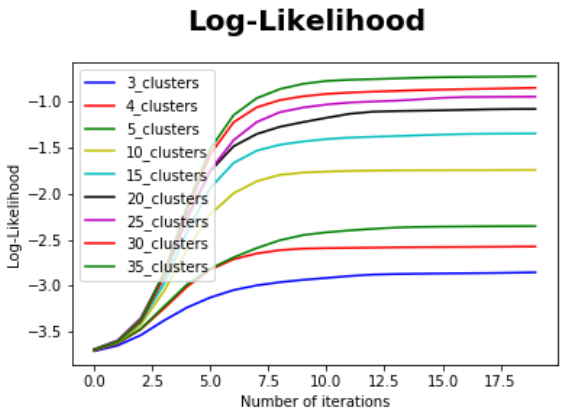

Figure 2: Comparison of the loglikelihood convergence as a function of the number of clusters

Comments to main fragments of code

Initialize random probabilities so that they sum to 1

```
27 #Create a vector of probabilities that sum to 1 every three probas
28 proba0 = np.random.rand(int(ordered_rdd.count()/nb_z), nb_z)
29 random_p = list((proba0 / np.reshape(proba0.sum(1), (int(ordered_rdd.count()/nb_z), 1))).flatten())
30
31 #Assign a probability to each triplet (user, movie, class)
32 q = ordered_rdd.map(lambda x : (x, random_p.pop(0)))
33 num_partitions = q.getNumPartitions()
34
```

Figure 3: Initialization of probabilities

During the initialization of the algorithm (Figure 3), we assign a random probability to each triplet (user, movie, class). However, we need to make sure that for each couple (user, movie), the sum of probabilities over the classes sum to 1. Otherwise we end up with probabilities larger than 1, which is impossible.

In order to respect this constraint, we created an array of probabilities summing to 1 every 3 probas (if the number of classes was 3). Then, we used the pop() function to assign the successive probabilities to each triplet in the RDD.

Control the number of partitions after a join operation

```
#Divide the probability of the (movie, class) couple by the probability of the class
Nsz = Nsz.map(lambda x : (x[0][1], (x[0][0], x[1])))
Psz = Nsz.join(Nz).coalesce(num_partitions)
Psz = Psz.map(lambda x : ((x[1][0][0], x[0]), x[1][0][1] / x[1][1])) #This gives us p(s | u)
```

Figure 4: Control of the number of partitions

Every time we performed a join operation, we realized that running the code would take a very long time - at the end, each iteration step of the algorithm would take at least 30 minutes. To solve this issue, we had to fix the number of partitions at the begining (Figure 4). Then, by using the coalesce function along with the initial number of partitions, we were able to control the number of partitions generated in our RDD after a join operation. Using this function was crucial in order to speed up the execution of the algorithm.

Choose the treshold to decide whether or not a user will see a movie

```
def prediction(rdd, threshold):
    return(rdd.map(lambda x : (x[0],x[1], x[2] >=threshold)))
```

Figure 5: Prediction threshold

```
result = prediction(Psu, 0.02)
result.collect()[0:10]
```

Figure 6: Collecting our predictions

Once the algorithm is trained, we can build a recommendation system using the estimations we obtained of theta = (p(z|u) and p(s|z)).

We first built an rdd with every possible combination of (user, movie), so that we can compute the probability that a user has been or will be watching the movie. In order to do so, we used the hidden variables Z (the classes) that help us in the computation of the probability. Thus, we needed to build an rdd with every possible combination of ((user, movie), class).

After that, we computed the probability that a user has been or will be watching a movie using the hidden variables : : p(movie | user) = p(movie | class) * p(class | movie). We obtained the probability distribution as shown in Figure 7.

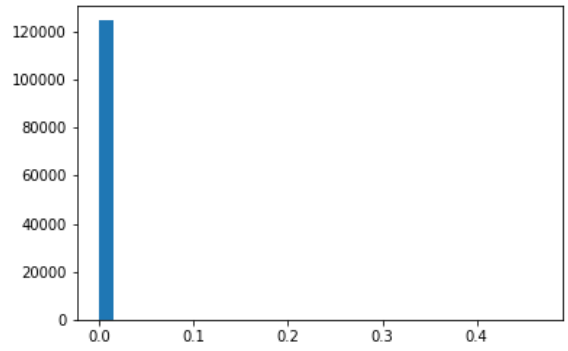

Figure 7: Probability distribution that a user will see a movie (Z = 10)

Given the above plot, we decided to set a treshold at 1.18e-117 (90% quantile). If the probability of the movie given the user is higher than 0.02, we can advise this movie to the user.

Concretely: running the EM algorithm on the cluster 101

- Connect to your cluster : `sudo /mnt/config/cluster/connect.sh`
- Run in your terminal `wget https://www.dropbox.com/s/ks1e8yu4m9d84r2/ratings.csv`, `wget https://www.dropbox.com/s/erghb7qaf681py8/recommender_system.py` to retrieve the files
- Run successively `hdfs dfs -mkdir /user/hadoop/recommend` and `hdfs dfs -put ratings.csv /user/hadoop/recommend`
- Run `spark-submit recommender_system.py`

Conclusion

We have implemented the PLSI algorithm on Spark (local and cluster). This approach is a flexible and computationally efficient approach to recommender systems: it has limited prerequisites for the data, and limited assumptions on the user, object pairs.

Appendix: code

Here are the documents attached to this report:

- `PLSI Algorithm.ipynb` which is the local Spark implementation, step by step, of our recommendation algorithm.
- `recommender_system.py` which is the script version meant to be run on the cluster.