Database Management project - Simple recommender system based on latent classes - Probabilistic Semantic Latent Indexing (PLSI)

Sacha Izadi - Clément Ponsonnet - Amir Benmahjoub

April 1, 2018

Abstract

Collaborative filtering is a technique that aims at learning user preferences and making recommendations based on user and community data. In this project, we implement Das & al. (2007) MapReduce formulation of PLSI, a collaborative filtering algorithm. This algorithm was one of the building bricks of the Google News recommender system. We first clarify the statistical foundations of the algorithm by expliciting the maths behind it, before implementing it in Spark on a small fraction of the database MovieLens. We then scale the algorithm for running it on an AWS cluster.

1 Introduction

Collaborative filtering emerged because the growth of the Internet has made it much more difficult to effectively extract useful information from all the available online ressources. The overwhelming amount of data necessitates mechanisms for efficient information filtering.

The challenge is to find interesting content for a web-user, as Das & al. states: "Search engines help solve the problem where you are looking for something specific that can be formulated as a keyword query. However, in many cases, a user may not even know what to look for. Often this is the case with things like news, movies etc., and users instead end up browsing sites looking around for things that might interest them". For its part, collaborative filtering helps present recommendations to a user based on his interests (e.g. extracted from his past online activity), and the interests of users that are similar to him.

Probabilistic latent semantic indexing (PLSI) is a collaborative filtering technique introduced by Hoffman based on the assumption that some hidden variables explain relationship between content and user. These hidden variables can be interpreted as clusters of like-minded people and clusters of similar contents. The general hypothesis is that "like-minded" people tend to like similar content. Since then, it was implemented by Google to build the recommender system for Google News.

The paper by Das & al. introduces a MapReduce formulation of PLSI. Based on this paper, we implemented a film recommender system, in Spark, on the MovieLens dataset.

We have organised this report as follow :

1. Introduction
2. Presentation of the dataset and expected output of the algorithm
3. Statistical foundations of Das & al. MapReduce formulation of PLSI
4. Description of the algorithm in pseudo-code
5. Implementation in Spark (Laptop version - `Jupyter` notebook)
6. Fine-tuning and analysis of performance and results.
7. Implementation in Spark using RosettaHub (Cluster version - `.py` files)
8. Comparison with Spark LDA algorithm (Laptop version - `.py` files

2 Presentation of the dataset and expected output of the algorithm

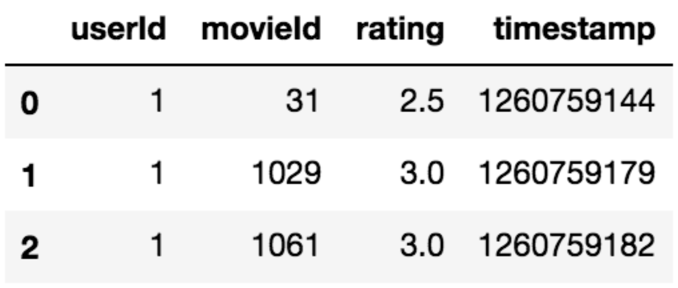
2.1 Input

As explained previously, we built our recommender system on MovieLens. The `ratings` dataset consists of the following variables :

- `userId`
- `movieId`
- `rating`
- `timestamp`

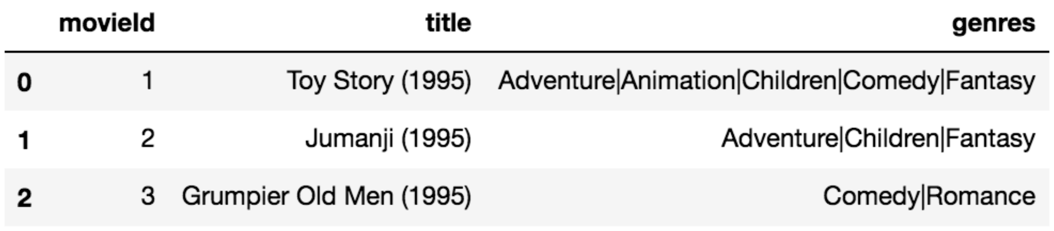
We decided to implement the "simplest" version of PLSI: we only considered the couples (`userId`, `movieId`) of films s seen by user u , without considering if the user actually liked the movie or not.

The `movies` dataset helps us retrieve the title and the genre of the movie given its `movieId`.



	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182

Figure 1: Ratings dataset



	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance

Figure 2: Movies dataset

2.2 Output

We want to tell if a film s is likely to be seen by a user u , so we compute the probability $p(s \mid u)$ that the film s was seen or is going to be seen knowing the user u. If this probably reaches a certain threshold t , we predict it as a candidate to propose to the user. We assess the quality of our model by splitting our dataset into a train and a test set, and evaluate the precision and the recall of our recommendations on both of them. This helps us fine-tuning the parameter t of the recommender system and the number of hidden variables mentionned in the introduction.

Metrics to mesure performance

To mesure the performance of the recommendations given by the model, Das & al. use the recall and the precision metrics, defined as :

- $precision = \frac{TP}{FP+TP}$

- $recall = \frac{TP}{FN+TP}$
- $F_1 score = 2 \times \frac{recall \times precision}{recall + precision}$

Graphically :

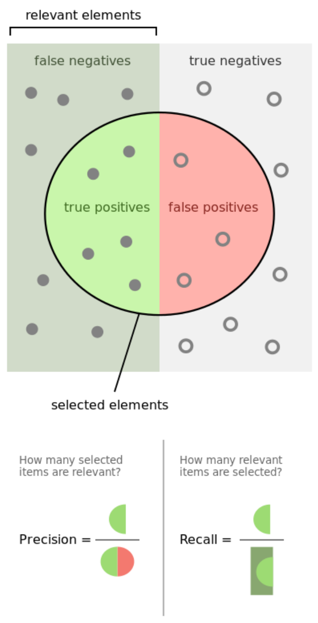


Figure 3: Precision and recall

The F_1 score is the harmonic average of the precision and recall.

3 Statistical foundations of Das & al. MapReduce formulation of PLSI

3.1 Model

This paragraph is taken from Das & al.

PLSI was introduced by Hoffman in 1999. He developed probabilistic latent semantic models for performing collaborative filtering. It models users ($u \in \mathcal{U}$) and items ($s \in \mathcal{S}$) as random variables, taking values from the space of all possible users and items respectively. The relationship between users and items is learned by modeling the joint distribution of users and items as a mixture distribution. A hidden variable Z (taking values from $z \in \mathcal{Z}$, and $|Z| = L$) is introduced to capture this relationship, which can be thought of as representing user communities (like-minded users) and item communities (genres). Formally, the model can be written in the form of a mixture model given by the equation:

$$p(s|u;\theta) = \sum_{z=1}^{L} p(s|z)p(z|u)$$

3.2 Notations

- A user will be noted as $u \in \mathcal{U}$, and a film as $s \in \mathcal{S}$.
- The authors introduce $Z \in \mathcal{Z} = [1, \ldots, L]$ representing the hidden variable capturing the relationship between users and variables ($\simeq$ clusters of films-users).
- They also introduce $\theta = (p(z|u), p(s|z))_{(z \in \mathcal{Z}, u \in \mathcal{U}, s \in \mathcal{S})}$ The parameter we aim at estimating.

- Once the parameter is estimated we obtain the probability that the film s has been watched / is going to be watched by the user u by computing $p(s|u;\theta) = \sum_{z=1}^{L} p(s|z)p(z|u)$.
- The log-likelihood of our statistical model is $L(\theta) = \log\left(p(\underline{s}|\underline{u};\theta)\right) = \sum_{t=1}^{N} \log p(s_t|u_t;\theta)$, where $(\underline{u},\underline{s}) = (u_t, s_t)_{t=1\dots,N}$ represent our dataset.
- In the demonstration we will introduce the function $q^*(z;u,s;\theta)$, already introduced in the paper. The notation has to be understood as a function of z, θ and of **the couple** $(\mathbf{u},\mathbf{s})$ representing the event "user u has actually watched film s". A better yet heavier notation would have been $q^*(z;(u,s);\theta)$.

3.3 Assumption

The main assumption of the PLSI model is that under $\mathbb{P}_{Z=z}$, U and S are independent.

3.4 Framework : the EM Algorithm

We want to estimate $\theta^* = \arg\max_\theta L(\theta)$, however a direct optimization is not possible. Instead we use the EM algorithm, a classical method for being able to estimate parameters of a distribution involving hidden variables. We based our demonstration on F. Bach's course on EM algorithm available here.

For any discrete probability distribution $q(z)$ we have the concavity inequality :

$$\log(p(\underline{s}|\underline{u};\theta)) \geq \sum_{\underline{z}} q(\underline{z}) \log\left(p(\underline{s},\underline{z}|\underline{u};\theta)\right) - \sum_{\underline{z}} q(\underline{z}) \log(q(\underline{z})) = \mathcal{L}(q,\theta)$$

With equality iff $q(\underline{z}) = q^*(\underline{z};\underline{u},\underline{s};\theta) := p(\underline{z}|\underline{u},\underline{s};\theta)$

Instead of maximizing $L(\theta)$, EM aims at maximizing $\mathcal{L}(\theta)$

The EM algorithm consists in 2 alternating phases:

- **E-step**: $q_{t+1} = \arg\max_q \mathcal{L}(q,\theta_t) = q^*_{t+1} = p(z|\underline{u},\underline{s};\theta)$
- **M-step**: $\theta_{t+1} = \arg\max_\theta \mathcal{L}(q_{t+1},\theta)$

3.5 The algorithm and its demonstration

E-step

Applying Bayes formula, we have:

$$\begin{aligned} q^* = p(z|u,s;\theta_t) = \frac{p_{\theta_t}(z;u,s)}{p_{\theta_t}(u,s)} &= \frac{p_{\theta_t}(u,s|z)p_{\theta_t}(z)}{\sum_{z'} p_{\theta_t}(u,s|z')p_{\theta_t}(z')} \\ &=_{(u|z)\perp(s|z)} \frac{p_{\theta_t}(u|z)p_{\theta_t}(s|z)p_{\theta_t}(z)}{\sum_{z'} p_{\theta_t}(u|z')p_{\theta_t}(s|z')p_{\theta_t}(z')} \\ &= \frac{p_{\theta_t}(z|u)p_{\theta_t}(s|z)p_{\theta_t}(u)}{\sum_{z'} p_{\theta_t}(z'|u)p_{\theta_t}(s|z')p_{\theta_t}(u)} \end{aligned}$$

We eventually obtain, the formula of the E-step:

$$q^*_{t+1} = q^*(z;u,s;\theta_t) := \frac{p_t(z|u)p_t(s|z)}{\sum_{z'} p_t(z'|u)p_t(s|z')}$$

M-step

We now consider the maximization problem:

$$\max_{\theta} \mathcal{L}(q^*_{t+1}, \theta) = \max_{\theta} \sum_{\underline{z}} q^*_{t+1}(\underline{z}) \log p_\theta(\underline{s}, \underline{z}|\underline{u})$$

With $\mathcal{L}(q^*_{t+1}, \theta) = \sum_i \sum_{z_i} q^*(z_i; u_i, s_i; \theta_t) \log\left(p_\theta(s_i|z_i) p_\theta(z_i|u_i)\right)$

As we want $p(s|z)$ and $p(z|u)$ to be 2 probability distributions, we impose the constraints $\sum_s p(s|z) = 1$ and $\sum_z p(z|u) = 1$. To solve this optimization under constraints problem we introduce the lagrangian $J(q_{t+1}, \theta, \lambda_z, \lambda_u)$:

$$J(q_{t+1}, \theta, \lambda_z, \lambda_u) = \mathcal{L}(q_{t+1}, \theta) - \lambda_z \left(\sum_s p_\theta(s|z) - 1\right) - \lambda_u \left(\sum_z p_\theta(z|u) - 1\right)$$

The computation of the gradient of J w.r.t. θ stands as:

$$\begin{aligned} \frac{\partial J}{\partial p_\theta(s|z)} &= -\lambda_z + \sum_i q^*(z; u_i, s, \theta_t) \frac{\mathbb{I}_{s_i=s, z_i=z}}{p_\theta(s|z)} \\ &= -\lambda_z + \sum_i \sum_u q^*(z; u, s, \theta_t) \frac{\mathbb{I}_{s_i=s, u_i=u, z_i=z}}{p_\theta(s|z)} \\ &= -\lambda_z + \sum_u \left(\frac{q^*(z; u, s, \theta_t)}{p_\theta(s|z)} \sum_i \mathbb{I}_{s_i=s, u_i=u, z_i=z} \right) \end{aligned}$$

and $\sum_i \mathbb{I}_{s_i=s, u_i=u, z_i=z} = 1$ under the asumptions of the model.

$$\frac{\partial J}{\partial p_\theta(s|z)} = 0 \Leftrightarrow 0 = -\lambda_z + \sum_u \frac{q^*(z; u, s; \theta_t)}{p_{\theta^*}(s|z)} \Leftrightarrow p_{\theta^*}(s|z) = \frac{\sum_u q^*(z; u, s; \theta_t)}{\lambda_z}$$

Using the constraint $\sum_s p(s|z) = 1$, we obtain $\lambda_z = \sum_s \sum_u q^*_{t+1}$
We eventually obtain, the formula of the M-step:

$$p_{t+1}(s|z) = \frac{\sum_u q^*(z; u, s; \theta_t)}{\sum_s \sum_u q^*(z; u, s; \theta_t)}$$

And similarly:

$$p_{t+1}(z|u) = \frac{\sum_s q^*(z; u, s; \theta_t)}{\sum_z \sum_s q^*(z; u, s; \theta_t)}$$

3.6 Das & al. MapReduce formulation

$q^*(z; u, s; \theta)$ can be rewritten very easily as :

$$q^*(z; u, s; \hat{\theta}) = \frac{\frac{N(z,s)}{N(z)} \hat{p}(z|u)}{\sum_{z \in \mathcal{Z}} \frac{N(z,s)}{N(z)} \hat{p}(z|u)}$$

with

- $N(z, s) = \sum_u q^*(z; u, s; \hat{\theta})$
- $N(z) = \sum_s \sum_u q^*(z; u, s; \hat{\theta})$
- $\hat{p}(z|u) = \frac{\sum_s q^*(z;u,s;\hat{\theta})}{\sum_z \sum_s q^*(z;u,s;\hat{\theta})}$

3.7 How to initialize the algorithm ?

So far, we have explained how to compute an iteration of the algorithm ($t \to t+1$), but not how to initialize it. During this project we came upon a surprising phenomenon: we naively initialized the algorithm with a uniform law ... and got stuck into a local maximum of the likelihood. This shows the importance of rightly initializing the algorithm.

Indeed, it is known that the EM algorithm can converge to local maxima of the likelihood function (thus, not necessarily to its global maximum). In his course, F. Bach states that "It does not converge to a global maximum but rather to a local maximum because we are dealing here with a non-convex problem. (...) Because EM gives a local maximum, it is clever to choose a θ_0 relatively close to the final solution."

As an example, imagine that $\hat{p}_0(s|z) = \frac{1}{card(\mathcal{S})}$ and $\hat{p}_0(z|u) = \frac{1}{card(\mathcal{Z})}$ are two uniform laws (the property can easily be generalized to any law that does not depend on z), then[1]:

At $t = 1$:

- $q_1(z;u,s) = \frac{\frac{1}{card(\mathcal{S})}\frac{1}{card(\mathcal{Z})}}{\sum_z \frac{1}{card(\mathcal{S})}\frac{1}{card(\mathcal{Z})}} = \frac{\frac{1}{card(\mathcal{S})}\frac{1}{card(\mathcal{Z})}}{card(\mathcal{Z})\frac{1}{card(\mathcal{S})}\frac{1}{card(\mathcal{Z})}} = \frac{1}{card(\mathcal{Z})}$

- $\hat{p}_1(s|z) = \frac{\sum_u \frac{1}{card(\mathcal{Z})}}{\sum_u \sum_s \frac{1}{card(\mathcal{Z})}} = \frac{\frac{card(\mathcal{U}|s)}{card(\mathcal{Z})}}{\sum_s \frac{card(\mathcal{U}|s)}{card(\mathcal{Z})}} = \frac{card(\mathcal{U}|s)}{card(\mathcal{U},\mathcal{S})}$

- $\hat{p}_1(z|u) = \frac{\sum_s \frac{1}{card(\mathcal{Z})}}{\sum_z \sum_s \frac{1}{card(\mathcal{Z})}} = \frac{1}{card(\mathcal{Z})}$

And at $t = 2$:

- $q_2(z;u,s) = \frac{\frac{card(\mathcal{U}|s)}{card(\mathcal{U},\mathcal{S})}\frac{1}{card(\mathcal{Z})}}{\sum_z \frac{card(\mathcal{U}|s)}{card(\mathcal{U},\mathcal{S})}\frac{1}{card(\mathcal{Z})}} = \frac{1}{card(\mathcal{Z})}$

- ... and we remain stuck on the same position as previously ...

However, we do not have any "simple" idea of the final solution and thus fail to choose $\hat{p}_0$ correctly.

- One approach would be to first make a pre-tagging of movies into categories z (e.g. $z_1 \simeq$ romantic movies, $z_2 \simeq$ horror movies ...) and initialize $\hat{p}_1(s|z)$ accordingly. To do so, we could use NMF.

- Or adopt the approach in Das & al.: "For the first iteration, we set $\hat{p}$ to appropriately normalized random values that form a probability distribution." - NB: the $\hat{p}_0$'s don't need to be normalized, since the E_1-step normalizes the q^*'s and the M_1-step normalizes the $\hat{p}$'s; the only requirement is to chose $\hat{p}_0$ not "singular" (i.e. lots of 0's which might lead to a $\frac{0}{0}$ in the E_1-step)

[1] $card(\mathcal{U},\mathcal{S}) = card(\texttt{train})$ and $card(\mathcal{U}|s) = card(\{\texttt{x} \in \texttt{train} \mid \texttt{x[1]} == \texttt{s}\})$

4 Description of the algorithm in pseudo-code

Algorithm 1: PLSI algorithm

```
Data: MovieLens = (user,film)
Result: $(p(s|u))_{\forall u,s}$
Initialization:;
Split data in train and test ;
$\forall (s,u,z) \in \texttt{train} \times \mathcal{Z} :\ q(z;u,s;\theta_0) \leftarrow \texttt{random}([0;1])$;
$\texttt{i} \leftarrow 0$ and $\texttt{logLik}_{\texttt{i}-1} \leftarrow -\infty$, $\texttt{logLik}_{\texttt{i}+1} \leftarrow +\infty$;
while $\texttt{i} < \texttt{nb}_{\texttt{iterations}}$ and $\texttt{logLik}_{\texttt{i}+1} - \texttt{logLik}_{\texttt{i}} > \epsilon$ do
    # M-step (Reduce operation);
    $\texttt{Puz} \leftarrow ((\texttt{user},\texttt{z}), \frac{\sum_s q^*}{\sum_z \sum_s q^*} = \hat{p}(z|u))$;
    $\texttt{Nsz\_normalized} \leftarrow ((\texttt{film},\texttt{z}), \frac{\sum_u q^*}{\sum_s \sum_u q^*} = \frac{N(z,s)}{N(z)} = \hat{p}(s|z))$;
    # E-step (Map operation);
    $\texttt{Q} \leftarrow (\texttt{user},\texttt{film},\texttt{z}, \frac{\frac{N(z,s)}{N(z)}\hat{p}(z|u)}{\sum_{z\in\mathcal{Z}} \frac{N(z,s)}{N(z)}\hat{p}(z|u)} = q^*(z;u,s;\hat{\theta}_{i+1}))$;
    # Termination of the algorithm;
    $\texttt{i} \leftarrow \texttt{i}+1$;
    $\texttt{Psu} \leftarrow ((\texttt{user},\texttt{film}), p(s|u) = \sum_z p(s|z)p(z|u))$;
    $\texttt{logLik}_{\texttt{i}-1} \leftarrow \texttt{logLik}_{\texttt{i}}$;
    $\texttt{logLik}_{\texttt{i}} \leftarrow \frac{1}{T}\sum_{u,s\in\texttt{train}} \log \hat{p}(s|u)$;
return: Psu
```

5 Implementation in Spark

5.1 From the pseudo-code to Spark

We implemented the PLSI algorithm in Spark using `PySpark` API. Full code is available in the jupyter notebook `PLSI_small_dataset.ipynb` provided with this report. As we followed exactly the steps we just described in the pseudo-code algorithm, we will only focus on some of the tricky points we encountered during the implementation. Namely,

- **How to normalize the probabilities we compute in the algorithm.**

  Indeed we can easily obtain the rdd $((\texttt{film},\texttt{z}), \sum_u q^* = N(s,z))$ by calling

```
# q = (u,s,z,q*)
# Nsz = (s,z,N(s,z))
Nsz = q.map(lambda Q : (Q[0].split(',')[1]+','+Q[0].split(',')[2],Q[1])).\
               reduceByKey(lambda x,y : x+y)
```

  Similarly for $(\texttt{z}, \sum_s \sum_u q^* = N(z))$

```
# q = (u,s,z,q*)
Nz = Nsz.map(lambda N : (N[0].split(',')[1], N[1])).\
                  reduceByKey(lambda x,y : x+y)
```

  But when we want to compute $\texttt{Nsz\_normalized} \leftarrow ((\texttt{film},\texttt{z}), \frac{N(z,s)}{N(z)})$ we need to join the previous rdds (using key `z`). The idea is :

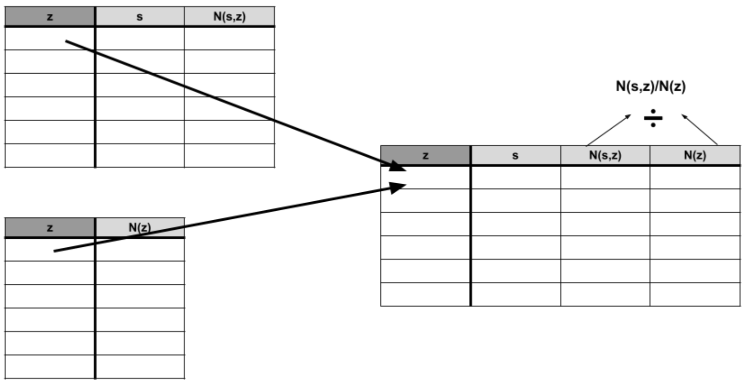


Figure 4: Normalizing probabilities

The following naive implementation did not scale at all since the join operation multiplied the number of partitions which, in the long run, only contained very few elements and made computations very inefficient....

```
# Naive implementation - returns (s,z, N(s,z)/N(z))
Nsz = Nsz.map(lambda x : (x[0].split(',')[1], (x[0].split(',')[0],x[1])))
tmpN = Nsz.join(Nz)
Nsz_normalized = tmpN.map(lambda x : (x[1][0][0]+','+x[0], x[1][0][1]/x[1][1])
    )
```

To deal with this issue, we used the `coalesce(int)` transformation which returns an rdd that is partitioned into a given number of partitions (and thus avoids the proliferation of empty rdds due to the join).

```
# Proper implementation - returns (s,z, N(s,z)/N(z))
Nsz = Nsz.map(lambda x: (x[0].split(',')[1], (x[0].split(',')[0],x[1])))
tmpN = Nsz.join(Nz).coalesce(num_part) #num_part defined in the initialization
Nsz_normalized = tmpN.map(lambda x: (x[1][0][0]+','+x[0], x[1][0][1]/x[1][1]))
```

- **How not to compute the same thing many times.**

  The Spark paradigm is based on the lazy evaluation of transformations, so when we encounter such a situation:

  ```
  rdd1 = rdd0.map(...).collect()
  rdd2 = rdd0.map(...).collect()
  ```

  Spark will execute the instructions that lead to `rdd0`

  - once to get `rdd1`
  - another time to get `rdd2` ...

  so we compute many times the same thing! In order to cache intermediary results we use the action `persist()` that calls the execution of the intermediary transformations before storing the results in the RAM for future uses. Since $\mathtt{Q} = (\mathtt{user}, \mathtt{film}, \mathtt{z}, q^*(z; u, s; \hat{\theta}_{i+1}))$ is the seed to all computations performed in an iteration of the EM algorithm, we decided to persist it for each step of the algorithm.

  ```
  while (curr_logLik-prev_logLik)>epsilon and nb_iter<max_iter) :
      nb_iter+=1
      Compute N(s,z), p(z|u) [...]
      Compute q [...]
      q.persist()
      Compute logLik [...]
  ```

- **How to detect the convergence of the algorithm.**

  Since the EM algorithm maximizes the log-likelihood of the model, we have chosen to use it as an indicator of the convergence of the algorithm. We initially thought that a condition such as $\mathtt{logLik_{i+1}} - \mathtt{logLik_i} > \epsilon$ (associated with a maximum number of iterations not to exceed) would be enough. But we rapidly came upon this learning curve :

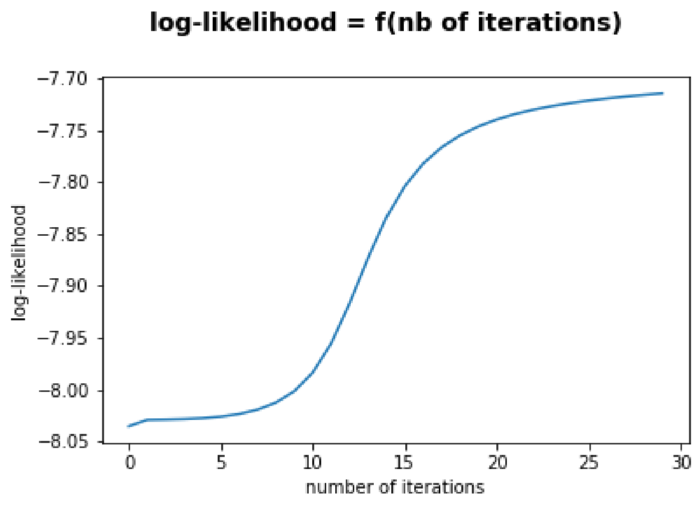


Figure 5: Convergence of the EM algorithm

We see that the speed of convergence of the algorithm is not constant, and a condition like $\texttt{logLik}_{\texttt{i+1}} - \texttt{logLik}_{\texttt{i}} > \epsilon$ would not be verified at the very begining of the algorithm, so we need to add a minimum number of iteration to complete before pretending the algorithm has converged.

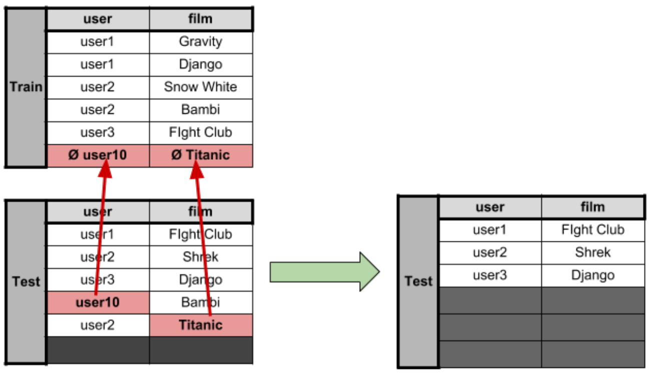
```
while (currLogLik-prevLogLik)>epsilon and nb_iter<max_it and nb_iter>min_it) :
    nb_iter+=1
    Compute N(s,z), p(z|u) [...]
    Compute q [...]
    q.persist()
    Compute logLik [...]
```

5.2 How to compute the recall and precision metrics

Does the recommender system we built work well? Will people click on the recommendations we give them? Here are the fundamental questions we should ask about our recommender system.

- **Splitting data into train and test**

  As we do not have real users to test our algorithm and we can only rely on the data from the MovieLens dataset, we adopt Das & al. approach: classicaly splitting data in a training and a testing set and use the precision and recall metrics we previously mentionned. Using this approach raises a simple issue: what if we have new users or new films in the testing set? We would not be able to predict anything about them which may lead to underestimating the performance of the recommender system. To deal with this issue, we filter users and film in the testing set that are also in the training set.



Train

user	film
user1	Gravity
user1	Django
user2	Snow White
user2	Bambi
user3	Flght Club
Ø user10	Ø Titanic

Test

user	film
user1	Flght Club
user2	Shrek
user3	Django
user10	Bambi
user2	Titanic

Test

user	film
user1	Flght Club
user2	Shrek
user3	Django

Figure 6: Filtering the training and testing sets

```python
def splitTrainTest(document, alpha=0.1, seed=42):
    train_test = document.randomSplit([1-alpha, alpha], seed=seed)
    train, test = train_test[0], train_test[1]

    # We keep film 's' which are in the test set AND in the train set
    s_train = train.map(lambda x : x.split(',')[1])
    s_test = test.map(lambda x : x.split(',')[1])
    s_common = sc.broadcast(s_test.intersection(s_train).collect())


    # Same for users 'u', we keep those who are in the test set AND in the
    train set
    u_train = train.map(lambda x : x.split(',')[0])
    u_test = test.map(lambda x : x.split(',')[0])
    u_common = sc.broadcast(u_test.intersection(u_train).collect())


    # filter the test set : If an element from the test set is not in the
    train set, we delete it
    test = test.map(lambda x : (x.split(',')[0], x.split(',')[1]))\
                .filter(lambda x : x[1] in s_common.value)\
                .filter(lambda x : x[0] in u_common.value)\
                .map(lambda x : (x[0]+','+x[1]))

    return([train,test])
```

- **Computing the precision and recall**

  To compute the precision and recall of our predictions on the testing set, we need to evaluate the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). Evaluating TP and FN is quite easy: for each couple `(user,film)` (i.e. for each event *user* `u` *saw film* `s`) in the testing set we check what kind of prediction we made:

  – if we predicted $p(s|u) \geq t$, it accounts as a TP,

  – if we predicted $p(s|u) < t$, it accounts as a FN.

  When it comes to evaluating TN and FP it becomes a bit more tricky.

  – **Evaluating FP:** Imagine that `user1` saw Titanic (i.e. (`user1`,Titanic)∈`train`) and we hopefully predict that $p(Titanic|user1) \geq t$. We know that (`user1`,Titanic)∉`test`, so if we check that our positive prediction is in the test set it will be considered as a FP ... whereas it is not!

  – **Evaluating TN:** Similarly, imagine now that `user1` saw Titanic (i.e. (`user1`,Titanic)∈`train`) and we hopelessly predict that $p(Titanic|user1) < t$. We know that (`user1`,Titanic)∉`test`, so if we check that this negative prediction is in the test set it will be considered as a TN ... whereas it is not!

  The conclusion is that we need to **only** consider the prediction made on events that did not happen in the train set, in order to properly evaluate precision and recall.

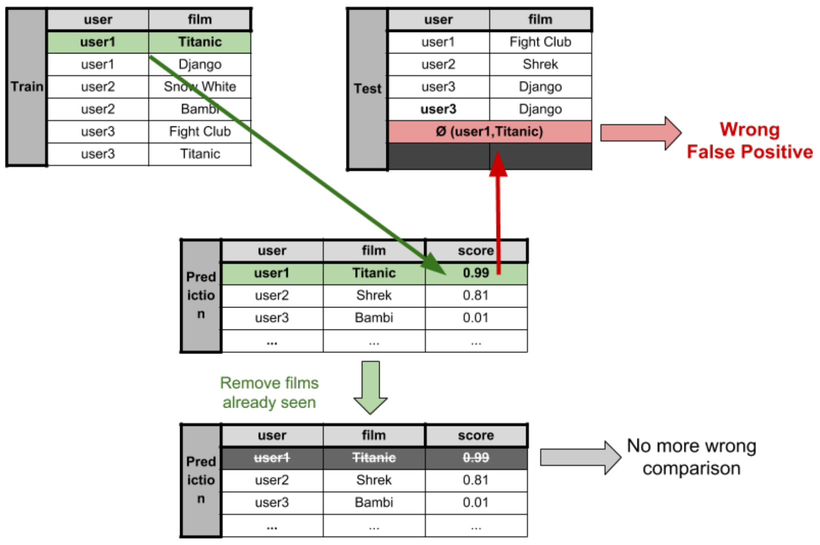


Figure 7: How not to produce wrong FP

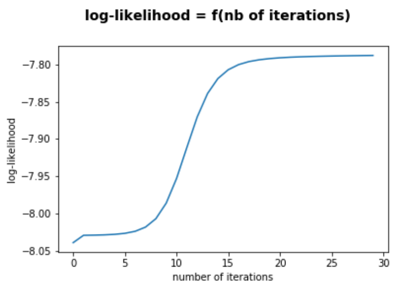
```python
def precisionRecallTest(pred, train, test) :
    # pred is of the form [('u,s', boolean)]
    # train is of the form : ['u,s']
    # test is of the form : ['u,s']


    # Remove from pred the films already seen by users (i.e. in the train set). To do so we use substractByKey() so we need to append something to the train rdd
    to_remove = train.map(lambda x : (x,1))
    predict = pred.subtractByKey(to_remove)


    # positive predictions (we predicted that user u will see film s)
    pos = predict.filter(lambda x : x[1])\
                .map(lambda x : x[0])
    nb_pos = pos.count()
    # TP = positive predictions that are also in the test set
    nb_TP = test.intersection(pos).count()
    # FP = P - TP
    nb_FP = nb_pos - nb_TP


    #negatives predictions (we predicted that user u will not see film s)
    neg = predict.filter(lambda x : not(x[1]))\
                .map(lambda x : x[0])
    nb_neg = neg.count()
    # FN = negative predictions that are actually in the test set
    nb_FN = test.intersection(neg).count()
    # TN = N - FN
    nb_TN = nb_neg - nb_FN


    # compute precision and recall
    precision = nb_TP / (nb_FP + nb_TP)
    recall = nb_TP / (nb_FN + nb_TP)

    return((precision,recall))
```

6 Fine-tuning and analysis of performance and results

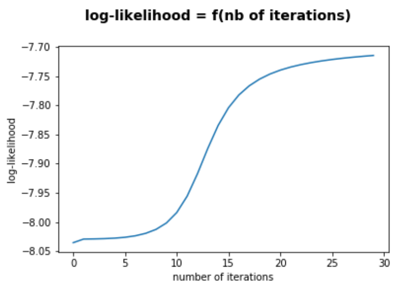
In the following section, we consider a limited part of the MovieLens dataset of 100,000 user reviews. This enabled us to compute the results very quickly on our laptop and understand the main behaviours of our algorithm before scaling up to RosettaHub on a bigger dataset. In order to compute the final results, we needed to find an optimal combination between the number of iterations of the EM algorithm, the number L of clusters and the threshold t .

6.1 Number of iterations

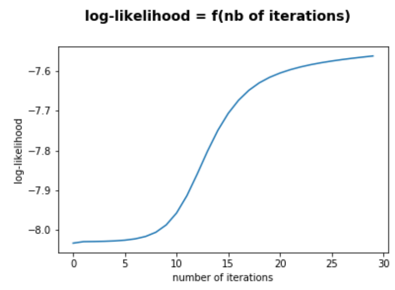
We want to find here the optimal number of iterations to run the EM algorithm, i.e. the number of iteration needed to reach the convergence of our algorithm. As indicated in the section above, we use the log-likelihood of the model as an indicator of the convergence of the algorithm. With all other parameters remaining fixed, a first method to set the optimal number of iterations is to detect the step when the log-likelihood reaches convergence i.e when $\texttt{logLik}_{\texttt{i+1}} - \texttt{logLik}_{\texttt{i}} < \epsilon$ *(we do not need absolute value since the loglikelihood increases at each step).* It is also necessary to associate this condition with a maximum number of iterations not to exceed in case the convergence is too slow. We plotted the log-likelihood curve for different numbers of clusters (i.e. different numbers of hidden variables) :



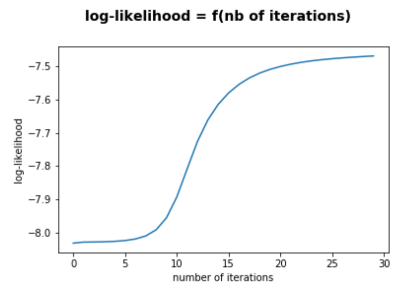
(a) 2 clusters



(b) 3 clusters



(c) 5 clusters



(d) 6 clusters

We noticed that the speed of convergence of the algorithm varies with the numbers of clusters (L), and that a condition like $\texttt{logLik}_{\texttt{i+1}} - \texttt{logLik}_{\texttt{i}} > \epsilon$ would not be verified at the very begining of the algorithm, so we added a minimum number of iteration to complete before pretending the algorithm has converged.

Also, we tested the impact of the number of clusters (from 2 to 20) on the speed of convergence. Why did we choose 20 clusters? Because these clusters can be interpreted as clusters of like-minded people and clusters of similar content (types of movies for example : Action, Science fiction...) and there is 15 types of movies in the Netflix classification. So we assumed that the number of cluster will not exceed 20 here.

We plot the log-likelihood convergence curve for a number of clusters between 2 and 20.

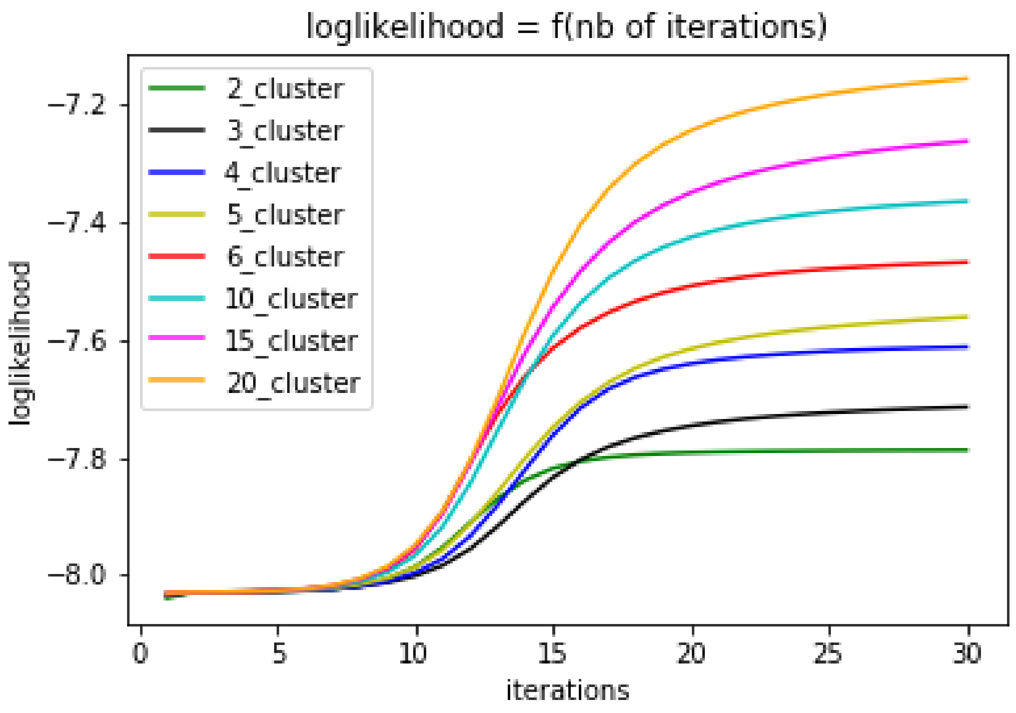


Figure 9: Convergence of the EM algorithm for different clusters numbers

From the plot above we conclude that :

- As the number of clusters increases, the loglikelihood at convergence ($\lim_{t\to\infty} L(\theta_t)$) increases. In fact, with a lot of clusters we manage to better capture user-films interactions. The more parameters we inject in the model, the higher the loglikelihood we compute on the training set. However, this leads to a risk of overfitting. This final point will be discussed in the part below.
- The curve has the same shape for all the cluster numbers, and the beginning of the jump happens at the same time, around 7 iterations.
- As the number of iterations increases, the convergence speed slows down (but very slightly). So we need to be carefull and plot the curve to check the convergence for a high number of clusters. However, it seems that for a cluster number between 2 and 20, 30 iterations are enough (35 iterations for 15 to 20 clusters if we want to be more precise).

Given these different plots and the numerical results for the different values of L, we noticed that for our database, the following parameters can ensure convergence:

parameter	value
$NbIterations_{min}$	15
ϵ	0.005
$NbIterations_{max}$	30

6.2 Optimal number of clusters L and optimal threeshold t

For a given database and assuming that we have chosen a sufficient number of iterations of the algorithm, the goal is to find the optimal number of clusters L and optimal threshold t.

To find this parameters, we repeat the following process for z = 3, 5, 10, 15 and 20 clusters :

1. Compute the result $p(s|u)$ for z clusters.
2. For $t \in \{10^{-x} \mid x \in \{1, 2, 3, 4, 5, ...\}\} compute the prediction$ p(s|u)>t$, then the precision, the recall and the F1 score.

3. Consider more specific values of t if needed.
4. Choose the value of t that makes the best trade off between precision and recall.
5. Compute the precision, the recall and the F1 score for this t.

Finally, we select the number of clusters z that gives the best precision-recall trade-off (higher F1 score).

Our results :

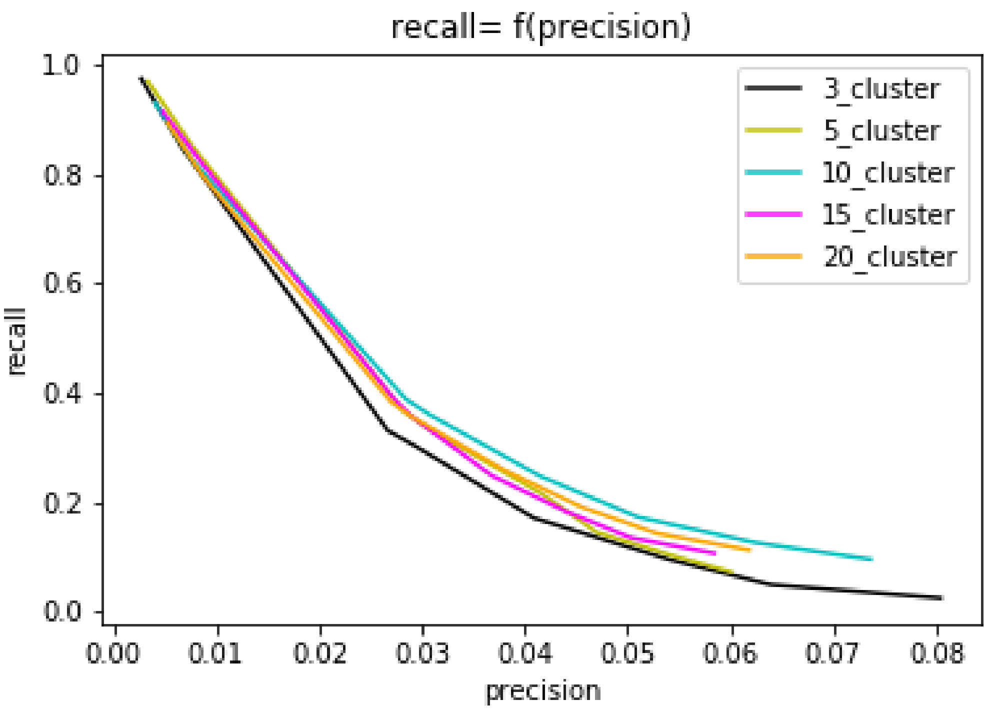


Figure 10: Recall vs Precision for different z values

We found that the best combination is:

parameter	value
L - Nb of clusters	10
t - threshold	0.0025
Max number of iterations	30

6.3 Final results and algorithm performance

With the parameters above we obtain the folowing results:

metric	score
precision	0.063
recall	0.126
F1 score	0.084

.

Unfortunately we were not able to reproduce as good results as those presented by Google in its paper... We think the main reason is that we considered too small a fraction of the MovieLens dataset: we lack data to properly identify communities ...

7 AWS Parallelization using RosettaHub

So far we have run all Spark in local mode on our personal computers. Because of this, all code was run on the 'small' Movielens dataset of 100 000 ratings. The full Movielens dataset contains

over 20 000 000 ratings, which is much too large for the EM algorithm to run on a single machine. Now, let's run our code on a remote cluster, and see how well our algorithm scales as we increase the size of the dataset. As a measure of this capacity to scale, we will look at the average time one iteration of the EM algorithm takes for datasets of size 1000, 100 000, 1 000 000, 5 000 000 and 20 000 000 ratings.
The process we followed to set up RosettaHub is explained in **Annex A**. The Spark is given in the `rosetta_just_em_loop.py` script provided with this report.

We run the spark script by using the "spark-submit" command in the rosettahub console, without specifying optional parameters. This means that Rosetta will use the default configuration which is: 5 executors with 4 cores each, 5GB of memory per executor. So using "spark-submit script.py" is the same as specifying:

```
spark-submit --deploy-mode client \
  --master yarn \
  --num-executors 5 \
  --executor-cores 4 \
  --executor-memory 5g \
  script.py
```

Results for computation times of the EM algorithm for different dataset sizes are given below:

Size of data	Initialization time	Avg Iteration time
1000	11.2 s	9.7 s
100 000	11.1 s	23.0 s
1 000 000	10.1 s	161.1 s
5 000 000	9.8	2555.3 s
20 000 000	N/A	N/A

Something interesting is happening: from 1000 to 100 000 ratings, we multiply the size of the dataset by 100, but the average EM iteration time is multiplied by less than 3. Then we multiply the size of the dataset by 10, and the iteration time is multiplied by 7. When we move from 1 million ratings to 5 million ratings, the size of the dataset is multiplied by 5, but the iteration time is multiplied this time by 15!
One explanation could be that for small datasets, the power of our cluster is not fully utilized, and so iteration time does not increase much with dataset size. However, at a certain point, our dataset is so big that the cluster is operating at maximum capacity, and increasing the size of the dataset greatly increases computing time for each EM iteration.

This shows how problematic it would be to fine-tune our model on a very large dataset. To find the optimal number of latent variables, and the optimal threshold t (explained in **6.2**), we will need many iterations of the EM algorithm, which itself iterates many times before converging. This would be infeasible on a large dataset with our current cluster configuration.

For 20 000 000 ratings we cannot even use this basic configuration: the executors do not have enough combined memory to deal with the memory intensive operations the EM algorithm requires. Therefore we run into the following error: an executor is killed for exceding memory limits, after this the program stalls and does not start again.

```
ExecutorLostFailure (executor 3 exited caused by one of the running tasks) Reason:
    Container killed by YARN for exceeding memory limits. 5.1 GB of 5.0 GB physical
     memory used. Consider boosting spark.yarn.executor.memoryOverhead.
18/03/31 17:08:22 INFO BlockManagerMasterEndpoint: Trying to remove executor 3 from
     BlockManagerMaster.
18/03/31 17:08:22 INFO BlockManagerMaster: Removal of executor 3 requested
18/03/31 17:08:22 INFO ExecutorAllocationManager: Existing executor 3 has been
    removed (new total is 0)
18/03/31 17:08:22 INFO YarnSchedulerBackend$YarnDriverEndpoint: Asked to remove non
    -existent executor 3
```

To move to ever bigger datasets, we need to create a larger cluster formation to process the data. Unfortunately, this is not possible within RosettaHub, and needs to be done directly through AWS. The focus of this project was implementing the PLSI algorithm in Spark, and therefore we do not include a section on moving from RosettaHub to AWS, though this could be the subject of future work.

8 Comparison with Spark LDA Implementation

We want to compare our implementation of the EM algorithm with the "ready-made" implementation of LDA in Spark. LDA (Latent Dirichlet Allocation) is a generative model often used in natural language programming, specifically to extract topics from a list of documents. It connects two variables: `documents` and the `words` present in the documents, through a third latent variable: `topics`. It does so by modelling the probability of a word appearing in a particular topic, and the probility of a document belonging to a particular topic.

This is very similar to the PLSI model: we are also connecting two variables: `users u` and `movies s`, through a third variable `z`, our latent classes. Though the distributional assumptions of the PLSI algorithm and of the LDA model are different, we can use both to recommend movies to users and compare the quality of these predictions.

8.1 Spark LDA function

- The `LDA` function, from the `ml` library in Spark (there is also an implementation of LDA int the `mllib` library, which we do not use) takes a dataframe as input. The dataframe contains 2 columns: the first column contains integers which corresponding to each user. The second column contains Vectors which represent the movies seen by the user. In fact the input resembles the following adjacency matrix:

User id	Movies seen
1	0, 0, 1, 0, ... , 1, 1, 0
2	0, 1, 0, 1, ... , 0, 1, 0
3	0, 0, 0, 1, ... , 0, 0, 0

  Where a 1 indicates that the user has seen movie i, while a 0 means he has not seen movie i. As we remember, the input we had used for PLSI was of another form (see the first 2 columns of *Figure 1*), so we need to transform this input. This can be done with the following code:

```python
from pyspark.ml.clustering import LDA
from pyspark.ml.linalg import Vectors, SparseVector

document = sc.textFile("hdfs:///user/hadoop/plsi/input/ratings.csv")
#Removing the header
header = document.first()
document = document.filter(lambda x: x != header)

# Each line is converted to a tuple: user, movie
parts = document.map(lambda l: l.split(","))
ratings = parts.map(lambda p: (p[0], p[1].strip()))
#aggregate on users: for each user, get string of all movies seen
ag_rating = ratings.reduceByKey(lambda x, y: "{0},{1}".format(x, y))
#Transform string of movies to list
ag_rating = ag_rating.map(lambda p: (p[0], p[1].strip().split(',')))
#Convert: key from string to int, value from list of strings to vector of
    doubles
for_LDA = ag_rating.map(lambda line: [int(line[0]), Vectors.dense([float(x)
    for x in line[1]])])

#Use sparseVectors to construct adjacency matrix
#Our Vectors will be of size max_movie_id + 1
max_movie_id = ratings.map(lambda x: int(x[1])).max()
adjmat = for_LDA.map(lambda line: [int(line[0]), Vectors.sparse(max_movie_id
    +1, line[1], [1] * len(line[1]))])

#Now put this in a dataframe with a label column, and a features column
adjmatDf = spark.createDataFrame(adjmat)
adjmatDf = adjmatDf.selectExpr("_1 as label", "_2 as features")
```

- We can now use our `adjmatDf` as input for the `LDA` function. We can also specify the following parameters in the function call:
  - k: Number of topics

– maxIter: Set the maximum number of iterations.

– docConcentration: Dirichlet parameter for prior over documents' (users') distributions over topics.

– topicConcentration: Dirichlet parameter for prior over topics' distributions over words (movies).

– We can also use the setOptimizer("em") method to specify that we want to use the EM implementation of LDA.

```
#Let's fit it with 3 topics
nb_k = 3
lda = LDA(k=nb_k).setOptimizer("em")
model = lda.fit(adjmatDf)
```

- The `describeTopics(maxTermsPerTopic)` attribute of our ldaModel object allows us to see the distribution of movies, given topics (topics in rows). The maxTermsPerTopic parameter controls how many movies are shown. In the following code, we only output the top 2 movies per topic. The termWeights column corresponds to the probability for the movies in the termIndices column:

```
>>> model.describeTopics(2).show(truncate=False)
+-----+-----------+------------------------------------------+
|topic|termIndices|termWeights                               |
+-----+-----------+------------------------------------------+
|0    |[2762, 318]|[0.01211652472647228, 0.010371437480348338] |
|1    |[1196, 594]|[0.007903917617181157, 0.007687240022711729]|
|2    |[592, 50]  |[0.014176817793207869, 0.012452081227526293]|
+-----+-----------+------------------------------------------+
```

This can be interpreted as the $P(s|z)$ from the PLSI algorithm. All we have to do is set maxTermsPerTopics to the total number of movies, in order to have the probabilities for all movies. We save these values in an rdd of the form $P_{z,s} = ([z, s], P(s|z))$.

```
topics = model.describeTopics(max_movie_id).rdd
Pzs = topics.flatMap(lambda x: [ ([x[0], x[1][i]], x[2][i]) for i in range(len
    (x[1])) ])
```

We still need to extract the $P(z|u)$ from the LDA algorithm in order to compute $P(s|u) = \sum_{z=1}^{Z} P(s|z) * P(z|u)$

- We can get the topic distribution per document with the following code:

```
>>> transformed = model.transform(adjmatDf)
>>> transformed.show()
+-----+--------------------+--------------------+
|label|            features|   topicDistribution|
+-----+--------------------+--------------------+
|    1|(106488,[31,1029,...|[0.25629798748565...|
|    4|(106488,[10,34,11...|[0.07975691068975...|
|    8|(106488,[32,45,47...|[0.74512188818623...|
|    9|(106488,[1,17,26,...|[0.57147978936416...|
|   10|(106488,[50,152,3...|[0.26910405580349...|
|   12|(106488,[253,529,...|[0.65466876353935...|
|   14|(106488,[594,1196...|[0.32160460911560...|
|    2|(106488,[10,17,39...|[0.17161248737304...|
|    3|(106488,[60,110,2...|[0.61871734154778...|
|    5|(106488,[3,39,104...|[0.71281049116724...|
|    6|(106488,[111,158,...|[0.56070588568633...|
|    7|(106488,[1,10,21,...|[0.15589421772295...|
|   11|(106488,[50,70,12...|[0.35173574946563...|
|   13|(106488,[1,47,110...|[0.61121943103524...|
|   15|(106488,[1,2,5,6,...|[0.22607439445159...|
+-----+--------------------+--------------------+
```

The topicDistribution column contains the distribution of topics per document. These are our $P(z|u)$

```
>>> transformed.select('topicDistribution').show(truncate=False)
+------------------------------------------------------------+
|topicDistribution                                           |
+------------------------------------------------------------+
|[0.256297996483167,0.42030121408237026,0.32340078943446277]  |
|[0.07975678094227423,0.842175570138486,0.07806764891923987]  |
|[0.7451220357000946,0.1384786408326583,0.11639932346724707]  |
|[0.5714798027858738,0.20607613873134992,0.2224440584827761]  |
|[0.26910406943190746,0.24974228270501578,0.48115364786307674]|
|[0.6546686408922139,0.18117911562261402,0.1641522434851721]  |
|[0.32160222637203983,0.42994331368036437,0.24845445994759582]|
|[0.17161242854200304,0.15834212633288935,0.6700454451251077] |
|[0.6187173047551272,0.1846660197397016,0.19661667550517128]  |
|[0.7128106179535887,0.14804403506752997,0.13914534697888115] |
|[0.5607062403697537,0.2528695958414716,0.18642416378877466]  |
|[0.1558942057104207,0.27530949891036693,0.5687962953792122]  |
|[0.3517480692470812,0.41344013253977974,0.2348117982131391]  |
|[0.6112194991702711,0.1981457663149516,0.19063473451477722]  |
|[0.22607443250600262,0.20328124839182252,0.570644319102175]  |
+------------------------------------------------------------+
```

We save these values in an rdd of the form $P_{u,z} = ([u, z], P(z|u))$

```
transformedRdd = transformed.rdd
Puz = transformedRdd.flatMap(lambda x: [ ([x[0], i], x[2][i]) for i in range(
    nb_k) ])
```

- Now we can deduce $P(s|u)$ using $P(s|u) = \sum_{z=1}^{Z} P(s|z) * P(z|u)$

```
1  #Prepare (z, [u, P(z|u)])
2  Puz_bis = Puz.map(lambda x : (x[0][1], [x[0][0],x[1]]))
3  #Prepare (z, [s, P(s|z)])
4  Pzs_bis =  Pzs.map(lambda x : (x[0][0], [x[0][1],x[1]]))
5
6  #Join them: you get (z, ([u, P(z|u)], [s, P(s|z)]))
7  #Need to do a broadcast join. Possible because Puz_bis is small and Pzs_bis is
       large
8  smallDict=dict( (x[0], x[1]) for x in Puz_bis.collect() )
9  bc=sc.broadcast(smallDict)
10 mapJoined = Pzs_bis.map(lambda x : (x[0], (bc.value[x[0]], x[1])))
11
12 #Map to: ((u, s), P(z|u)*P(s|z))
13 Psu = mapJoined.map(lambda  x: ((x[1][0][0] , x[1][1][0]), (x[1][0][1]*x
       [1][1][1])))
14 #Sum over same (u,s) couples to get P(s|u)
15 Psu = Psu.reduceByKey(lambda x,y : x+y)
```

Notice that we used a broadcast join on lines 8-10 to join `Puz_bis` and `Pzs_bis`. The second rdd was too large to use a regular rdd join on our laptops in local model.

8.2 Comparison with PLSI

With the calculated $P(s|u)$, we are in a similar situation as in PLSI. We can now define a threshold and recommend a movie if P(s|u) is higher than this threshold. We then use these recommendations to calculate precision and recall.

To compare with PLSI, we take the 100 000 rating dataset, split it in the same train and test set as we had done for PLSI, and fit the LDA model on the train set. We can then compute precision and recall using the same functions as the ones we had defined for PLSI. We do not give the detail of the code but it can be found in the `LDA_script.py file`. We use the parameters that we found to be optimal for the PLSI algorithm, that is:

parameter	value
L - Nb of clusters	10
t - threshold	0.0025

The results are the following:

metric	LDA train	LDA test	PLSI test
precision	0.311	0.0	0.063
recall	0.452	0.0	0.126

The LDA does quite well on the train set, but we need to look at how well it performs on the test set. PLSI already had low precision and recall on the train set. LDA does even more poorly, scoring 0 on both ! LDA has only managed to predict couples that were observed in the training set, but none of those observed in the test set. We must note that the parameters that were optimal for PLSI, are not necessarily optimal for LDA. We did not do a full study to find the optimal number of clusters and optimal threshold for LDA. We tested a few possible values for the threshold and all results were quite bad. For example, taking $t = 0.0005$ as the threshold, we get $precision = 0.0032$ and $recall = 0.5$ on the test set for the LDA algorithm. Overall we would say results are similarly mediocre for PLSI and LDA.

To conclude, we would like to point out that the reason why both models seem to generalize so badly to new data is probably due to the small size of the dataset. The 100 000 MovieLens dataset we used is a very small subset of the 20 000 000 ratings dataset. With little data, it is not surprising that our model mainly predicts couples that are already observed, but not the couples of the test set. However, this prediction capability would increase as we increase the size of the dataset. To get the type of results Das & al. present in their paper, we would need much larger clusters to handle very big datasets - something Google excels at.

A
Setting up RosettaHub

Here we describe the steps followed to set up RosettaHub for the analysis conducted on the scaling of the algorithm (part 7).

- Launch a cluster on RosettaHub and connect using ssh as seen in the labs.
- Activate the cluster using the command:

```
sudo /mnt/config/cluster/connect.sh
```

- Download the dataset from Dropbox (or another online location). Many of the datasets used can be found in the following folder:
  https://www.dropbox.com/sh/2po5ikn6w0etrrc/AABtkkBaCQ0bgIjefNZvo_4ja
  Use the following command:

```
wget https://www.dropbox.com/s/mx0009vk3tjbzqa/ratings_1m.csv
```

- Create an input and an output directory in hdfs with the following commands

```
hdfs dfs -mkdir /user/hadoop/plsi
hdfs dfs -mkdir /user/hadoop/plsi/input
hdfs dfs -mkdir /user/hadoop/plsi/output
```

- Put the dataset in the input directory in hdfs

```
hdfs dfs -put ratings_1m.csv /user/hadoop/plsi/input
```

- Download the script to use from dropbox or another location

```
wget https://www.dropbox.com/s/tf5c1qe7gxooz2r/rosetta_just_em_loop.py
```

- Launch the script

```
spark-submit rosetta_just_em.loop.py
```