Recommender Systems

A guide to Collaborative filtering using Alternating Least Square aka. ALS

UBP / ISIMA

Students : Yahya Amaroch & Mohamed Ez-zaouia

Supervised by : Mr. Professor Engelbert MEPHU NGUIFO

Table of Contents

Introduction

Nowadays there is too much information on the net, we have many more options and choices that we used to have and - for sure - the number is increasing day after day. Sometimes it happens to be unable to make a choice with a multitude of options and choices.

Recommender systems are useful alternatives to make a choice, by providing meaningful recommendations to a collections of users for items or products (books, movies, etc.) that might interest them.

So, recommender systems are a set of methods for filtering through both information space (all the available items that user could select, choose or rate, etc. from) and observations space (what user really *observed*) in order to provide a meaningful recommendations in the information space that user does not have any observation yet.

Amazon, Netflix are the widely known using recommender systems in the industry.

Approaches to Recommendation

Many approaches are used in the field of the recommender systems the most used are :

- **Collaborative filtering** : Recommend items based only on users past behaviors.
  - User-based : Recommend to a user what similar users liked.
  - Item-based : Recommend to a user similar items to those he has previously liked.
- **Content-based** : Recommend based on items features.

Collaborative filtering differs from Content-based methods in the sense that users and items do not play a role in the recommendation, no need to know about items or even users contents. For example, deciding that two users may share similar tastes because they are the same age is not a collaborative filtering. Deciding that two users may both like the same a movie because they play many other same songs is an example.

Let ***U*** and ***D*** be - respectively - the set of users and the set of items, if we represent the interaction between users and items with a matrix ***UxD***, collaborative filtering aim then to fill entries of a user-item association matrix (item which the user has not yet observed).

In order to illustrate the principle of collaborative filtering, we are going to use the example of Netflix or Movielens when a user is used to rate movies.

In the next parts we focus only on Collaborative filtering approach.

Collaborative Filtering Basic Idea

Let's take a space of 5 users and 7 items (movies), an example of matrix representation of this space may looks like the table below.

Let's call the user-item matrix **M,** the '**0**' in the table mean that the user has not yet observed or rated the movie, recommendation system aim to predict those missing values (ratings) such that the values would be consistent with the existing ratings in the matrix, i.e, how users would rate those items, if determined so, the system can make meaningful recommendations to the users.

Matrix representation of the user-item association

	D1	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**
U1	5	4	3	**0**	4	2	5
U2	4	3	**0**	**0**	**0**	2	4
U3	**0**	**0**	3	**0**	2	**0**	3
U4	1	3	**0**	4	5	**0**	1
U5	**0**	1	**0**	**0**	3	**0**	2

Matrix factorization as solution

The first fact to solve this problem is, users' ratings are - in general - not arbitrary, there should be some ***hidden*** reasons or factors that determine how a user rates an item. For example, some users would give a high rating to a certain movie if they like the director of the movie or the movie is a science movie (for example the user may like the category of the movie), etc. So, discovering this *hidden* factors or features allows to predict the ratings for users and items. This class of algorithms is called **latent-factor** models, they try to explain observed interactions between large number of users and products through a relatively small number of *unobserved, hidden or underlying reasons.*

The second fact is that the Matrix which represent the association user-item is sparse; most of the entries are '**0**' this means that a few of all possible user-item associations actually appear in the data; one approach to overcome this is to factor **M** as a product of two matrices, **A** and **B** with a few columns **k** which is the number of **latent-factors**, such that :

$$X = AB^T \qquad (1)$$

$$M = AB^T \qquad (2)$$

$$M \approx X = AB^T \qquad (3)$$

- Each row in **A** is represented as a vector of latent-factors, this mean each row represents the strength of the associations between a user and the latent-factors (features), the same thing for **B**, each row represents the strength of the associations between an item and the the latent-factors.
- The matrix **M** is sparse, but the product ***(1)*** is dense.
- There is now exact solution to the equation ***(2)***, but we can give an approximation to the solution, this means ***(1)*** should still be as close to **M**.
- The approximate solution to a such equation ***(2)*** is called ***Least Squares***, except that here we have two *unknown* variables **A** and **B**, to do so, we have another algorithm called the ***Alternating Least Squares*** aka. ***ALS***, the idea behind this algorithm is to estimate **A** using **B** and estimate **B** using **A**, and after enough number of iterations, we are aiming to reach a convergence point where the two matrices **A** and **B** are no longer changing, or the change is very small.

Objective function to solve

The objective function to solve is formulated in the equation ***(4)*** which is composed from two parts, the first one (blue color) will minimize the square error of the observed ratings and the second part (green color) is a regularization with a constant parameter μ in order to overcome the overfitting of the model. Such that, $M \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{n\times k}$

$$\min_{A,B}\sum_{u,d}(m_{u,d} - A_u B^T{}_d)^2 + \mu\left(\sum_u \|A_u\|^2 + \sum_d \|B_d\|^2\right) \qquad (4)$$

In order to penalize the items that do not have a ratings, this allows the model to depend only on the rated movies from users, to do so, we can use the weight matrix ***C*** *defined in* ***(5)***

$$C_{u,d} = 1 \text{ if } m_{u,d} > 0 \text{ otherwise } 0 \qquad (5)$$

In this cas the equation ***(4)*** will be :

$$J(A,B) = \min_{A,B} \sum_{u,d} C_{u,d}\,(m_{u,d} - A_u B^T{}_d)^2 + \mu\left(\sum_u \|A_u\|^2 + \sum_d \|B_d\|^2\right) \qquad (6)$$

Remark:

There is a version of the ALS algorithm called Weighted ALS (aka. W-ALS) aimed to optimize a collaborative filtering system from a dataset built from implicit feedback (e.g. views, clicks, purchases, likes, shares etc.). W-ALS introduces different *confidence* levels for which items are preferred by the user, in this case the objective function will be exactly like (**f**) but different ways could applied to compute the confidence measure, for example : $\mathbf{c_{u,d} = 1 + \alpha m_{u,d}}$, $\mathbf{c_{u,d} = 1 + \alpha \log(1 + m_{u,d}/\zeta)}$ are the most applied.

ALS algorithm

Begin

1. Initialize matrix **A** randomly or by assigning the average rating for that movie as the first row, and small random numbers for the remaining entries.
2. Fix **A** , solve **B** by minimizing the **J**(**A,B**) function.
3. Fix **B**, solve **A** by minimizing the **J**(**A,B**) function similarly.
4. Repeat 2. and 3. until convergence.

end.

To minimize **J**(**A,B**) it should be $\partial \mathbf{J}(\mathbf{A},\mathbf{B})/\partial \mathbf{A} = \mathbf{0}$ and $\partial \mathbf{J}(\mathbf{A},\mathbf{B})/\partial \mathbf{B} = \mathbf{0}$ (∂ denote the differentiation of **J**(**A,B**) with the respect of the two variables **A**, **B** respectively), the solutions for the vectors **A** and **B** are :

let $C(A,B) = \sum_{u,d} c_{u,d} (m_{u,d} - A_u B^T{}_d)^2 + \mu(\sum_u \|A_u\|^2 + \sum_d \|B_d\|^2)$

So,

$$\partial C(A,B)/\partial A = -2\sum_d c_{u,d} (m_{u,d} - A_u B_d{}^T) B^T{}_d + 2\mu A_u$$

$$= -2\sum_d c_{u,d} (m_{u,d} - B^T{}_d A_u) B^T{}_d + 2\mu A_u$$

$$= -2B^T C^u m(u) + 2B^T C^u B A_u + 2\mu A_u$$

- Each row of the matrix $B \in \mathbb{R}^{n \times k}$ is $B^T{}_d$
- $C^u \in \mathbb{R}^{n \times n}$ is the diagonal matrix which has the coefficient $c_{u,d}$ in each row/column d
- $m(u) \in \mathbb{R}^n$ contains element $m_{u,d}$ in row d

For the optimal $\partial C(A,B)/\partial A = 0$,

we have, $$B^T C^u m(u) = (B^T C^u B + \mu I) A_u$$

do, $$A_u = (B^T C^u B + \mu I)^{-1} B^T C^u m(u)$$

- I is the identity matrix

By doing the same thing with $\partial C(A,B)/\partial B = 0$, we get

$$B_d = (A^T C^d A + \mu I)^{-1} A^T C^d m(d)$$

- Each row of the matrix $\boldsymbol{A} \in \mathbb{R}^{m \times k}$ is $\boldsymbol{A}^T{}_{\boldsymbol{u}}$
- $\boldsymbol{C}^d \in \mathbb{R}^{m \times m}$ is the diagonal matrix which has the coefficient $\boldsymbol{c}_{\boldsymbol{u},\boldsymbol{d}}$ in each row/column $\boldsymbol{u}$
- $\boldsymbol{m(d)} \in \mathbb{R}^m$ contains element $\boldsymbol{m}_{\boldsymbol{u},\boldsymbol{d}}$ in row $\boldsymbol{u}$

Example

In this part we are going to build a collaborative filtering recommender system using the **Apache Spark**'s **Python** implementation of the ALS.

MovieLens is a research website run by the University of Minnesota, the website use Collaborative Filtering to make movies recommendations to users. The group of researchers behind this project provide two datasets a small dataset with **100.235** ratings, **8.928** movies and **718** users, and a full dataset with **21.622.187** ratings, **30.106** movies and **234.934** users.

In this example we are going to use the small dataset to choose over cross validation the best latent-factor from factors, after that we are going to generate a model using this best factor and test it a test set, also we have used this best factor on the full dataset but for brevity raisons we will give a link on our Github to check that out.

1. Download and unzip the files from urls provided by MovieLens, please refer to the link below for full details.

```
full_dataset_url = 'http://files.grouplens.org/datasets/movielens/
ml-latest.zip'
small_dataset_url = 'http://files.grouplens.org/datasets/movielens/
ml-latest-small.zip'
```

2. In the dataset we have a file ratings.csv each row contains : userId,movieId,rating,timestamp; and a movies.csv file each row contains : movieId,title,genres

```
In [10]:
# ratings.csv : userId,movieId,rating,timestamp
small_ratings_file = os.path.join(datasets_path, 'ml-latest-small',
'ratings.csv')
small_ratings_raw_data = sc.textFile(small_ratings_file)
small_ratings_raw_data_header = small_ratings_raw_data.take(1)[0]

# remove the header and the timestamp
small_ratings_data = small_ratings_raw_data.filter(lambda line: line
!= small_ratings_raw_data_header)\
    .map(lambda line: line.split(",")).map(lambda tokens:
(tokens[0],tokens[1],tokens[2])).cache()
```

```
# example of 3 first elements and RDD object small_ratings_data
small_ratings_data.take(3)

Out[10]:
[('1', '1', '5.0'), ('1', '2', '3.0'), ('1', '10', '3.0')]



In [11]:
# movies.csv : movieId,title,genres
small_movies_file = os.path.join(datasets_path, 'ml-latest-small',
'movies.csv')

small_movies_raw_data = sc.textFile(small_movies_file)
small_movies_raw_data_header = small_movies_raw_data.take(1)[0]

# remove the header and the genres
small_movies_data = small_movies_raw_data.filter(lambda line: line !
= small_movies_raw_data_header)\
    .map(lambda line: line.split(",")).map(lambda tokens:
(tokens[0],tokens[1])).cache()

# example of 3 first elements in the RDD object small_movies_data
small_movies_data.take(3)

Out[11]:
[('1', 'Toy Story (1995)'),
 ('2', 'Jumanji (1995)'),
 ('3', 'Grumpier Old Men (1995)')]
```

3. Now we will split the dataset into training set, validation set and a test set

```
In [14]:
# split small_ratings_data into 60% training set, 20% validation set
and 20% test set
training_RDD, validation_RDD, test_RDD =
small_ratings_data.randomSplit([6, 2, 2], seed=0)
validation_for_predict_RDD = validation_RDD.map(lambda x: (x[0],
x[1]))
test_for_predict_RDD = test_RDD.map(lambda x: (x[0], x[1]))
```

4. Now we will compute the best latent-factor called here rank, so, we will generate a model using ALS implementation and find the best rank over cross validation using validation set.

```
In [30]:
from pyspark.mllib.recommendation import ALS
import math

'''
```

```
  Here we will try to find the best latent factor over cross
validation
  called k in the paper by computing the mean square (the Root Mean
Square Error aka. RMSE)
  error for k in 4, 8, 12
'''

seed = 5
iterations = 10
# specifies the regularization parameter in ALS.
regularization_parameter = 0.1
# is the number of latent factors in the model.
ranks = [4, 8, 12]
errors = [0, 0, 0]
err = 0

min_error = float('inf')
best_rank = -1
best_iteration = -1

for rank in ranks:

    # ALS.train will train and generate our model
    model = ALS.train(training_RDD, rank, seed=seed,
iterations=iterations, lambda_=regularization_parameter)

    # predict ratings for the validation model using the validation
dataset
    predictions =
model.predictAll(validation_for_predict_RDD).map(lambda r: ((r[0],
r[1]), r[2]))
    rates_and_preds = validation_RDD.map(lambda r: ((int(r[0]),
int(r[1])), float(r[2]))).join(predictions)

    # compute the error (validation_dataset -
predicted_validation_dataset)
    error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1]
[1])**2).mean())

    errors[err] = error
    err += 1

    print('For rank {} the RMSE is {}'.format(rank, error))
    if error < min_error:
        min_error = error
        best_rank = rank

# print the best latent factor (rank) which generate a minimum RMSE
```

```
print('The best model was trained with rank equals to 
{}'.format(best_rank))

For rank 4 the RMSE is 0.9290115466719586
For rank 8 the RMSE is 0.9411651453706049
For rank 12 the RMSE is 0.9428971812389256
The best model was trained with rank 4
```

5. Example of the predicted ratings with the model ((userId, movieId) , (userRating, modelPredictedRating))

```
In [20]:
rates_and_preds.take(3)

Out[20]:
[((244, 35836), (3.5, 3.7881972952871883)),
 ((616, 2318), (5.0, 3.7703969474117467)),
 ((99, 41569), (3.5, 2.6824938834166203))]
```

6. Now we will generate the model using the computed best rank and then test it using the test set by computing the RMSE

```
In [31]:
# now we will train our model using the best_rank (best latent 
factor)
# and then test using the test dataset
model = ALS.train(training_RDD, best_rank, seed=seed, 
iterations=iterations, lambda_=regularization_parameter)

# now we will test our model using test dataset
predictions = model.predictAll(test_for_predict_RDD).map(lambda r: 
((r[0], r[1]), r[2]))

# compute the error test_dataset_rating - test_predicted_rating
rates_and_preds = test_RDD.map(lambda r: ((int(r[0]), int(r[1])), 
float(r[2]))).join(predictions)
error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - 
r[1][1])**2).mean())

# print the RMSE
print('For testing data the RMSE is {}'.format(error))

For testing data the RMSE is 0.9316026750566204
```

7. For the work on the full dataset please refer to this link: https://github.com/ezzaouia/sparkvm/blob/master/notebook/notebook_als_movielens/ml_spark.ipynb

Conclusion

In this paper we have seen the basic idea behind the ALS algorithm used in the field of recommender system specially the collaborative filtering. We have first tied to find the best latent factor by using a cross validation over 3 factors, then we have built a model using the best one the test phase has given an RMSE equals to 0.9316026750566204 which can be acceptable for this model.

For the full dataset with plus than 21 millions ratings we have got and RMSE equals to 0.8335530555506335 which is much better.

It looks **465.774 seconds** (**7,76 minutes**) to train the model over the full dataset in a machine with **4Gb** with **50% of a CPU 2,6 GHz Intel Core i5**, which is quite good as it's done on a guest machine on top of virtual box for a such dataset in the industry context, for example with a datacenter, the result could be done in few seconds if it's not in realtime.

References

- Advanced Analytics with Spark O'REILLY edition
- http://www.atmos.washington.edu/~dennis/MatrixCalculus.pdf
- http://stanford.edu/~rezab/dao/notes/lec14.pdf
- http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/
- http://ampcamp.berkeley.edu/5/exercises/movie-recommendation-with-mllib.html
- http://bugra.github.io/work/notes/2014-04-19/alternating-least-squares-method-for-collaborative-filtering/