Matrix Factorization for recommender systems using ALS and SGD

AHMADI Itaf & HAMIDOU Wissem

December 15, 2017

Abstract

Recommending new products to users is a common challenge faced by a lot of Internt companies (such as Amazone, Facebook, Netflix etc...). In this context, matrix factorization models are known to be one the excellent approaches for producing recommendations. Therefore, in this project, we are going to implement and interpret two of the famous algorithms of matrix factorization for recommender systems : Alternating Least Square(ALS) and Parallelized Stochastic Gradient Descent.

1 Introduction

Using spark in python, we are going to implement Alternating Least Square using the *algorithm 1* and Parallelized Stochastic Gradient Descent using *algorithm 3*.
In this project, we used MovieLens *20M (190 MB)* and *100K (5MB)* (for testing) Dataset from the Movielens website : `https://grouplens.org/datasets/movielens/`.

2 Requirements

- Anaconda 3 or +
- Spark 2.2

3 Model presentation

We define U as the set of users and I as the set of items.
For each couple (u;i), weather the user u did not see the movie i and we don't have data, or we have a rating $R_{u,i}$ of the movie i given by the user u.
We suppose that there is a latent space C, such that we can write the matrix of rating as a scalar product of tow matrix P and Q where $Q_{u:}$ is representation of the client in the space C and $P_{:i}$ is the representation of the items in this hidden (latent) space.
Thus we can write : $R_{u,i} = \sum_{c \in C} Q_{u,c} P_{c,i}$
The power of this model is that it is enable us to predict a score for the user u if he didn't see the movie i and to recommend him movies that he didn't see but that he is apt to like.
To solve this problem we need to use the following optimization problem based on the least square with regularization :

$$(Q^*, P^*) = argmin_{P,Q} \ \frac{1}{2} \sum_{(u,i) \in K} (R_{u,i} - \sum_{c \in C} Q_{u,c} P_{c,i})^2 + \frac{\rho}{2} (\sum_{u \in U, c \in C} Q_{u,c}^2 + \sum_{i \in I, c \in C} P_{c,i}^2)$$

$$= argmin_{P,Q} \ \frac{1}{2} \| 1_K o (R - QP) \|_F^2 + \frac{\rho}{2} \|P\|^2 + \frac{\rho}{2} \|Q\|_F^2$$

The gradient of the objective function is given by :

$$\bigtriangledown f(P,Q) = \begin{pmatrix} -Q^T \left[1_K o (R - QP) \right] + \rho P \\ - \left[1_K o (R - QP) \right] P^T + \rho Q \end{pmatrix}$$

Parameters :
- K is the set of the couples (u,i) for which the rating $R_{u,i}$ is known.
- $(1_K)_{u,i} = 0$ if (u, i) $\notin K$ and $(1_K)_{u,i} = 1$ if $(u, i) \in K$ this is represented in our code by the mask matrix. In fact this mask matrix will enable us in the recommandation phase to avoid recommanding a movie that the user had already seen.
- $(AoB)_{u,i} = A_{u,i} B_{u,i}$, the 'o' operator represent the product of the tow matrix term by term.
- $\|.\|_F$ is the Frobenius norm.
- ρ is a regularization parameter.

Notice that this objective is non-convex function (because of the $\sum_{c \in C} Q_{u,c} P_{c,i}$ term). As an approximate approach, Gradient descent can be used here, however it costs a lot of iterations and it's very slow since this problem is NP-Hard to optimize.
In this project, two method will be implemented and interpreted to perform matrix factorization : Alternating Least Square(ALS) and Parallelized Stochastic Gradient Descent.

4 Alternating Least Square(ALS)

Algorithm 1 ALS Algorithm

1: **for** $k \geq 1$ **do**
2: $P_k \longleftarrow argmin_P \ \frac{1}{2}\|1_K o(R - Q_{k-1}P)\|_F^2 + \frac{\rho}{2}\|P\|^2 + \frac{\rho}{2}\|Q_{k-1}\|_F^2$
3: $Q_k \longleftarrow argmin_Q \ \frac{1}{2}\|1_K o(R - QP_{k-1})\|_F^2 + \frac{\rho}{2}\|P_{k-1}\|^2 + \frac{\rho}{2}\|Q\|_F^2$
repeat until convergence

Procedure description

Alternating Least Square is an approach in which we fix one variable (let's say Q) and do minimization over the second variable (matrix P), then in this case, the objective function becomes convex function of Q and vice versa. Therefore, for each iteration, we will first fix Q and optimize P, then fix the obtained P and optimize Q , and repeat the procedure until convergence.

Programming Part

In order to compute the argmin of the objective function, we coded a gradient linear search algorithm based on the Armijo–Goldstein condition[1] that returns the minimum of the objective function.

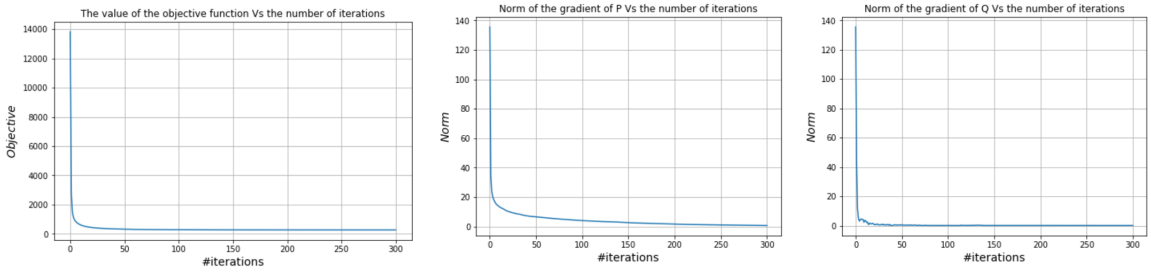
```python
def search_gamma(func, P, c1, c2):
    n = 0
    gamma = c2 * (c1 ** n)
    P1 = P - gamma * func(P)[1]
    while func(P1)[0] >= func(P)[0] + np.vdot(func(P)[1], (P1-P)) + 1. / ( 2 * gamma) * (np.linalg.norm(P1-P) ** 2) :
        n += 1
        gamma = c2 * (c1 ** n)
        P1 = P - gamma * func(P)[1]
    return c2 * (c1 ** (n-1))


def grad_line_search(func, P0, epsilon):
    P = P0
    max_iter = 0
    gamma = 0.5
    while (np.linalg.norm(func(P)[1]) > epsilon and max_iter < 300):
        c2 = gamma
        gamma = search_gamma(func ,P, 0.5, 0.5)
        P = P - gamma * func(P)[1]
        max_iter += 1
    return P, max_iter
```

Then, we coded the ALS function that uses the functions **objective objective_P** and **objective_Q** in the python file *functions.py* :

```python
def ALS(P0, Q0, mask, rho, eps, max_iter):
    Q = Q0
    P = P0
    k = 0

    val, grad_P = objective_P(P0, Q0, R, mask, rho)
    _, grad_Q = objective_Q(P0, Q0, R, mask, rho)

    list_obj = []
    list_norm_P = []
    list_norm_Q = []

    list_obj.append(val)
    list_norm_P.append(np.linalg.norm(grad_P))
    list_norm_Q.append(np.linalg.norm(grad_Q))

    while (np.linalg.norm(grad_P) > eps or np.linalg.norm(grad_Q) > eps) and k < max_iter:

        # minimizing P and Q using a line search
        P, _ = grad_line_search(lambda P : objective_P(P, Q, R, mask, rho), P, eps)
        Q, _ = grad_line_search(lambda Q : objective_Q(P, Q, R, mask, rho), Q, eps)

        val, grad_P = objective_P(P, Q, R, mask, rho)
        _, grad_Q = objective_Q(P, Q, R, mask, rho)
        k += 1

        list_obj.append(val)
        list_norm_P.append(np.linalg.norm(grad_P))
        list_norm_Q.append(np.linalg.norm(grad_Q))

        if k == 1 or k % 10 == 0:
            print("iteration = %d"%k)
            print("Norm of grad_P = %0.5f "%np.linalg.norm(grad_P))
            print("Norm of grad_Q = %0.5f "%np.linalg.norm(grad_Q))

    return P, Q, max_iter, list_obj, list_norm_P, list_norm_Q
```

Results

We first tested our code on small data (Matrix with size 100 x 100) and we plotted the variation of the objective function vs the number of iterations as well as the norm of both the gradient of the matrix P and Q and we notice that this values increases with increasing number of iterations which proves that we are converging to the solution.
Here are the obtained results :



In this section we need to compare our sparse matrix R to the dense predicted matrix QP. A way to measure the difference is to compute the relative error using the mask matrix.
Here is the obtained result:

```
relative_error = np.linalg.norm(mask*(R - np.dot(Q, P))) / np.linalg.norm(R) * 100

print("Relative error : %f"%relative_error)
print("Total objective f(P,Q) = %0.5f "%(objective(P, Q, R, mask, rho)[0]))
```

```
Relative error : 10.801425
Total objective f(P,Q) = 269.09040
```

Recommendation

In order to find the movie that we'll recommand to the user number u, we have to consider the decomposition $R = QP$.
In fact, by multiplying the u^{th} row of Q by the matrix P, we obtain a vector r of size I x 1 that contains all the estimated ratings of movies for this user u.
Now, we only need to consider the highest score and take its index (which correspond to the recommanded movie), however we shoudn't forget that we have to avoid recommanding a movie that the user had already seen, that's why we need to multiply our vector r by the opposite of the mask.
Here is the recommendation that we obtained for unser number **45** :

What recommanding to the user n° 45 ?

```
u = 45
r = np.dot(Q[u,:], P)

r = r * (1 - mask[u,:])

movie_index = np.argmax(r)
print("We recommand to the user n° %s the movie n°  %d"%(u,movie_index))
print ("The rating of this movie is : %0.3f "%r[movie_index])
```

```
We recommand to the user n° 45 the movie n°  31
The rating of this movie is : 4.729
```

5 Parallelized Stochastic Gradient Descent

Algorithm 2 SGD Algorithm

1: **for** $t = 1 \dots T$ **do**
2: Pick randomly j $\in \{1..m\}$
3: $x_{t+1} = x_t - \eta \, \partial f(x_t)$

Algorithm 3 Parallel SGD Algorithm[4]

1: **for** $i = 1 \dots K$ **do**
2: $v_i = SGD()$ on each client machine
Aggregate from all the computers $v = \frac{1}{K} \sum_{k=1}^{K} v_k$
3: **return** v

The Parallelized Stochastic Gradient Descent relies on the distributed computation of gradients locally on each computer which holds parts of the data and subsequent aggregation of gradients to perform a global update step. This algorithm scales linearly instead of calculating a whole stochastic gradient descent which can become computationally infeasible to process all data by a single machine since it's an inherently sequential algorithm and we may even get the results in the matter of days.

Implemtation

In this part, we will implement two functions based on the previous algorithms. First, we will implement the **SGD** function. It takes as an input a matrix R which is in our case a sparse matrix, the mask matrix and Q, P which are the factorisation of R, we use the minimisation of the l2 norm as loss function to minimze the distance between $(R - QP)$ as we explained in the model presentation part.
Since we do the SGD on each cluster (i.e matrix block) we will parallelize the complilation of the SGD later on in the second function using *spark.paralleize* as if we preform an SGD on each block and then we aggregate the results.
The second function is the *Parallelized_SGD* function. We show below the implementation of the SGD function :

```
def SGD(R, Q, P, mask, Ni, Nj, blockRange):
    """
    This function is an implementation of the SGD algorithm described above.
    Input : R, Q, P, mask, Ni, Nj, blockRange
    Output : Q, P, n, blockRange
    """

    global rho
    eta = .01 # first step size
    R_new = R.nonzero()
    n = R_new[0].size

    for i in range(n):

        j = random.randint(0, n-1) # Pick randomly an element j
        row, col = R_new[0][j], R_new[1][j] # retrieve the row and column of the random j

        # take a small blocks from R, mask, Q and P
        Ri = R[row,col]
        maski = mask[row,col]
        Qi = Q[row,:]
        Pi = P[:,col]

        # compute the gradient of Qi and Pi
        _, grad_Q = objective_Q(Pi, Qi, Ri, maski, rho/Ni[row])
        _, grad_P = objective_P(Pi, Qi, Ri, maski, rho/Nj[col])
        eta = eta * (1 + i) ** (- 0.5)

        # update the blocks of P and Q
        Q[row,:] = Qi - eta * grad_Q
        P[:,col] = Pi - eta * grad_P
        #print(np.linalg.norm(Q[row,:]))

    return (Q, P, n, blockRange)
```

At this stage we take the whole matrix, we transform it to blocks. This process is done by dividing the number of rows by the number of spark workers (or block_number in the function) so each sub-block will measure(N_i, N_j) then we will divide this sublock also into two matrixes $Q(N_j, c)$ and $P(c, N_j)$ with c a factorisation factor that may be defined by the user (It's the latent variable defined in the model presentation part). Afterwards, each sublock of R is taken as input to the SDG function along with the corresponding Q and P.

Results

We computed the relative error using this algorithm and here is the obtained result :

```
relative_error = np.linalg.norm(mask*(R - np.dot(Q, P))) / np.linalg.norm(R) * 100
print("Relative error : %f"%relative_error)
```

```
Relative error : 26.429458
```

In term of recommendation, we found :

What recommanding to the user n° 45 ?

```
u = 45
r = np.dot(Q[u,:], P)

r = r * (1 - mask[u,:])

movie_index = np.argmax(r)
print("We recommand to the user n° %s the movie n°  %d"%(u,movie_index))
print ("The rating of this movie is : %0.3f "%r[movie_index])
```

```
We recommand to the user n° 45 the movie n°  55
The rating of this movie is : 3.447
```

6 Conclusions

Since we don't have enough cores in our PCs to work with the whole data set(190 MB), we preformed our analysis on a small chunk of 100k. Working with mini data helped us accelerate our algorithms and thus we obtained the results in a matter of half an hour .The two algoritrhms used above have different sturcutres but they both minimize the same function. The ALS alternetevly try to approach the real matrix but the parallel SGD tries to distribute the matrix over many spark workers (block number) and then aggregate the result. When we compare the relative error, we find that ALS gives better results than parallelized SGD and this may be due to the small number of iterations we choose as well as the number of workers we defined. In practice, Netflix for example doesn't use this type of solutions for it's recommender system since NMF, SVD, parallel SGD or even ALS predict well for static matrices.In fact the viewing data obviously makes a big difference because we do not know whether the recommendations work for people who are going to "watch now" or "watch in the future.
For real accurate prediction, we need to work with data from instant streaming services, as result new algorithms other than those we tried in this project

shall be applied like Restricted Boltzmann Machines (RBM).

References

[1] *Armijo condition* https://en.wikipedia.org/wiki/Backtracking_line_search

[2] *Distributed Algorithms and Optimization, Spring 2015* https://stanford.edu/ rezab/classes/cme323/S15/notes/lec14.pdf

[3] *Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems.* https://datajobs.com/data-science-repo/RecommenderSystems-[Netflix].pdf

[4] *Martin Zinkevich, Parallelized Stochastic Gradient Descent* http://martin.zinkevich.org/publications/nips2010.pdf

[5] *Collaborative Filtering in Spark RDD-based API* https://spark.apache.org/docs/2.2.0/mllib-collaborative-filtering.html

[6] *Why Netflix Never Implemented The Algorithm That Won The Netflix $1 Million Challenge :* https://www.techdirt.com/articles/20120409/03412518422/why-netflix-never-implemented-algorithm-that-won-netflix-1-million-challenge.shtml