Matrix Factorization with Alternating Least Squares

Project Overview:

Ever since Netflix released it Netflix Prize open competition in 2009, popularity and interest in collaborative filtering/recommendation systems has grown among the software engineering and data science community. Our project attempts to analyze the Netflix dataset in a distributed manner in an effort to make predictions on how much a user will enjoy a particular movie based on their movie preferences, or known information about the user. From a high level, this requires taking in a very large dataset that represents a sparse matrix of users by movies where each cell value in the ratings matrix represents the rating given by the user. To produce the predictions, this requires us to first decompose the ratings matrix into two smaller matrices (P and Q) using matrix factorization. Choosing the ideal factor matrices involves using the cost function Alternating Least Squares (ALS) which iteratively alternates between a fixed matrix P/Q to calculate the values for the other corresponding matrix P/Q, in an effort to find the ideal configuration of two matrices that minimises the error of the cost function or until the cost converges to an acceptable threshold. Once these two matrices are found, we can simply perform matrix multiplication to generate a new ratings dense matrix with all ratings filled in, including the newly predicted ratings.

Input Data

https://www.kaggle.com/netflix-inc/netflix-prize-data

The data is comprised of the following:

- Movie IDs range from 1 to 17,770 sequentially.
- Customer IDs range from 1 to 2,649,429, *with gaps*. There are 480,189 users.
- Ratings are on a five star (integral) scale from 1 to 5.

With the help of some preprocessing done in Python on the original datasets, we have created the following input data where each line read in represents a cell (or rating) in the sparse rating matrix.

Example of input data:

user,movie,rating
1,29,3
1,156,2
1,172,4
1,174,5
2,29,4
2,172,3
3,156,3
3,172,4
3,57,5

4,29,4
4,156,5
4,172,3
4,174,1
4,57,2

Algorithm and Program Analysis

The Alternating Least Squares (ALS) approach is a gradient descent algorithm that decomposes a given large user/item matrix R into lower k-dimensional user factor matrix P and an item factor matrix Q. In the most simple approach you can then estimate the user rating (or in general preference) by computing the inner (dot) product for the corresponding user and item vectors in the factor matrices. ALS represents a different approach to optimizing the loss function.

The key insight while computing the factors is that you can turn the non-convex optimization problem into an "easy" quadratic problem. By holding P or Q constant the problem can be turned into the Ordinary Least Squares problem which has a unique and guaranteed global minimum. In ALS we fix each one of the factor matrices alternatively and use the closed form solution to compute a new factor matrix that minimizes the cost function. When one is fixed, the other one is computed, and vice versa. This is iteratively continued until you reach a set of P and Q matrices that meet the convergence criteria (i.e the iterations terminate once we reach an acceptable delta of error).

ALS Algorithm

We perform Alternating Least Squares algorithm as follows:

1. Partition the Ratings matrix by userID to create R_U, and similarly partition Ratings by ItemID to create R_I (so there are two copies of Ratings with different partitionings). In R_U, all ratings by the same user are on the same machine, and in R_I all ratings for same item are on the same machine.

2. With k factors and ratings matrix $R_{u \times i}$, create matrices $P_{k \times u}$ and $Q_{k \times i}$ (Figure 1). Then randomly assign values to Q as the starting point.

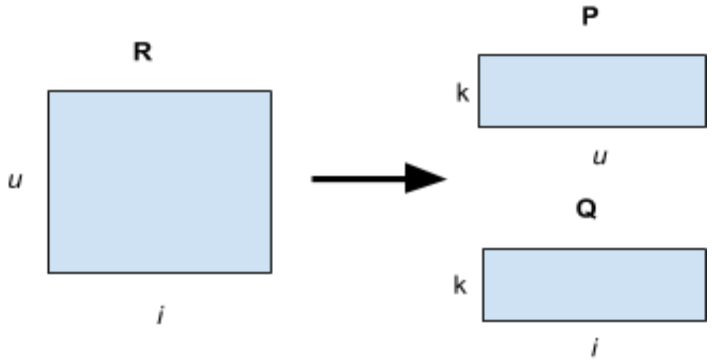


Figure 1

3. Broadcast the current P and Q matrices to each partition.

4. Using R row r_u and Q columns q_j (corresponding to non-empty columns in r_u), use equation [2] to compute the update of p_u locally on each machine, where p_u is a column belonging to the P matrix (see Figure 2).

5. Using R row r_i and P columns p_i (corresponding to non-empty columns in r_i), use equation [3] to compute the update of q_i locally on each machine, where q_i is a column belonging to the Q matrix.

6. Compute the least squares error by taking the summation of all differences between each $r`_{ui}$ (equal to $p_u^T \cdot q_i$), and the corresponding non-empty value in $R_{u \times i}$ using the cost function [1].

7. Repeat steps 3-7 and compute difference to minimize the least squares error of the observed ratings until convergence.

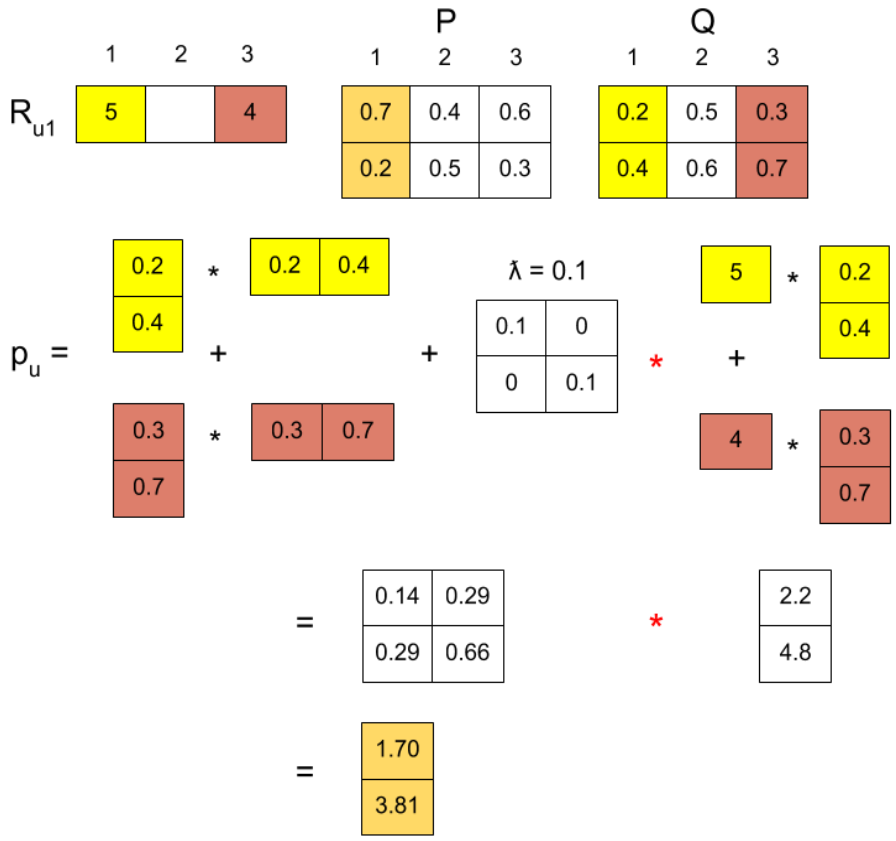


Figure 2

Algorithm - Key Formulas

Cost Function (includes L2 regularization)

$$min(P, Q) = \sum_{r_{u,i}\, obs} (r_{u,i} - p_u^T q_i)^2 + \lambda(\sum_u ||p_u||^2 + \sum_i ||q_i||^2) \quad [1]$$

Update formula for factor matrix p (users)

$$p_u = (\sum_{r_{u,i}\, \varepsilon\, r_{u,*}} q_i^T q_i + \lambda I_k)^{-1} * \sum_{r_{u,i}\, \varepsilon\, r_{u,*}} r_{u,i} q_i \quad [2]$$

Update formula for factor matrix q (items)

$$q_i = (\sum_{r_{u,i}\, \varepsilon\, r_{*,i}} p_u^T p_u + \lambda I_k)^{-1} * \sum_{r_{u,i}\, \varepsilon\, r_{*,i}} r_{u,i} p_u \quad [3]$$

Pseudocode

```
 var Q = DenseMatrix.fill(nFactors, sortedItems.length)(minRating + rand.nextDouble()
 * (maxRating - minRating) + 1)

 var q_bdcast = sc.broadcast(Q)

 val tolerance = 0.0
 val lambda = 0.1
 val convergenceIterations = 100

 while(costDiff >= tolerance && iterations < convergenceIterations ) {

   // Step to calculate New P
   // Calculates gradient for new P in RDD form
   val newP = R_u.groupByKey()
     .mapValues(row => computeGradient(row,q_bdcast,lambda))
     .collect()

   // converts newP to a new dense matrix P
   var P = DenseMatrix(newP.map(_.toArray):_*).t

   // Rebroadcast P
   p_bdcast = sc.broadcast(P)

   // calculates gradient for new Q in RDD form
   val newQ = R_i.groupByKey()
     .mapValues(row => computeGradient(row,p_bdcast,lambda))
     .collect()

   // converts newQ to a new dense matrix Q
   var Q = DenseMatrix(newQ.map(_.toArray):_*).t

   // Rebroadcast Q
   q_bdcast = sc.broadcast(Q)

   residual = 0

   // #### Step to compute cost ####
   R_u.foreach{ case (userId, (movieId, r_ij)) =>
     val q_i = Q(::, movieId.toInt)
     val p_u = P(::, userId.toInt)
     residual += math.pow(r_ij - (p_u.t * q_i), 2)
   }

   // Computes norms for each latent matrix
   val pu_norm = sum(sum(P *:* P, Axis._0))
   val qi_norm = sum(sum(Q *:* Q, Axis._0))

   // Computes total cost function for iteration
   totalCost = residual + (lambda * (pu_norm + qi_norm))

   // Computes deltas between previous cost and current cost
   costDiff = math.abs(totalCost - prevCost)

   logger.info("Iter: " + iterations + " Cost: " + totalCost + " Delta: " + costDiff)

   prevCost = totalCost

  iterations += 1
}
```

Data Analysis

Total Input	Total (Possible) Output Volume
|R| (100+ million ratings)	|u*k| + |i*k| (480,189 * k) + (17,770 * k) = 497,959*k values where k is a hyperparameter corresponding to the number of latent factors

Experiments

The matrix factorization program was run on AWS m5.xlarge machines at 100 iterations each time. Small clusters represent 5 machines, while large clusters represent 10 machines. The max filter was filtered on userID, and varied to measure scalability and the cluster size was varied to measure speedup. Different regularization parameter, lambda values, were run to seek an optimal value for convergence.

Run	Lambda	Max Filter	Input (records)	Output (factors)	Cluster Size	Run Time
1	0.075	100k	3,737,128	$P_{18094*50} + Q_{1770*50}$	Small	1 hour, 1minute
2	0.075	100k	3,737,128	$P_{18094*50} + Q_{1770*50}$	Large	33 minutes
3	0.075	33k	1,243,184	$P_{5869*50} + Q_{1770*50}$	Large	13 minutes
4	0.075	66k	2,473,519	$P_{11909*50} + Q_{1770*50}$	Large	23 minutes
5	0.650	100k	3,737,128	$P_{18094*50} + Q_{1770*50}$	Large	32 minutes
6	0.050	100k	3,737,128	$P_{18094*50} + Q_{1770*50}$	Large	32 minutes
7	0.100	100k	3,737,128	$P_{18094*50} + Q_{1770*50}$	Large	32 minutes

Logs

Run 1	*Run 2*	*Run 3*	*Run 4*	*Run 5*	*Run 6*	*Run 7*

Outputs

Output 1	*Output 2*	*Output 3*	*Output 4*	*Output 5*	*Output 6*	*Output 7*

Speedup

Cluster Size	Max Filter	Lambda	Run Time
Small (5 workers)	100k	0.075	1 hour, 1minute
Large (10 workers)	100k	0.075	33 minutes

For speedup, two runs were recorded on a small cluster of 5 workers and a large cluster of 10 workers. Each of these runs were given the same hyperparameters and input with a max filter of 100k. As shown in the table, increasing from a small cluster to a large cluster resulted in a decrease in running time by approximately 50%. Thus, the scaleup of our program is very reasonable, providing a linear decrease in running time given an increase in workers.

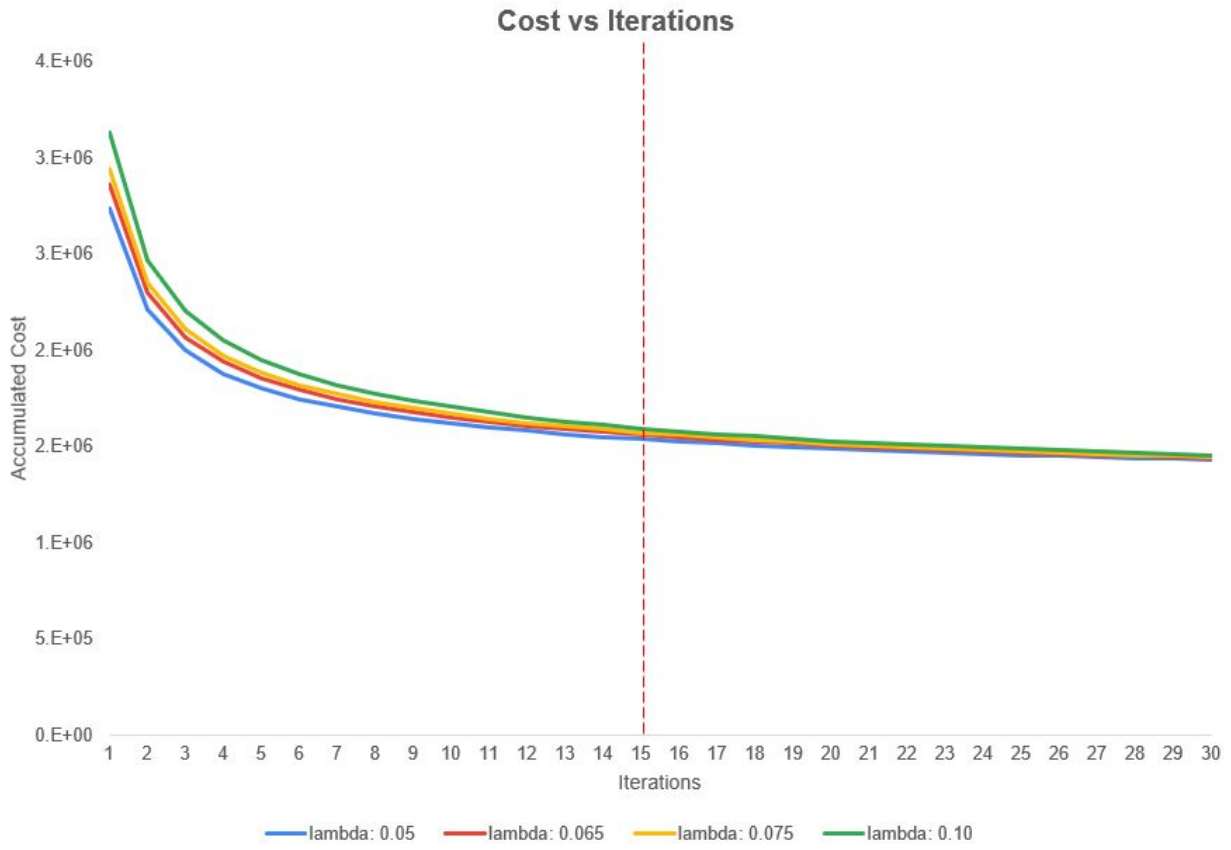
Scalability

Max Filter	Input (records)	Lambda	Run Time
33k	1,243,184	0.075	13 minutes
66k	2,473,519	0.075	23 minutes
100k	3,737,128	0.075	33 minutes

For scalability, three runs were recorded with a max filter of 33k, 66k, and 100k on a large cluster with the same configuration. With each increase in max filter there was an observed linear increase in input record size. Additionally a linear increase in input records resulted in an observed linear increase in running time. Thus the scalability of our program shows a linear trend when increasing the max filter and ultimately the size of the input.

Optimization

It is important to have the most optimal hyperparameters for ALS to converge the quickest. Our runs experimented with various regularization parameters (lambda) to find the optimal one. As per the graph and table below, it can be concluded that a hyperparameter of 0.050 had the quickest convergence and smallest change in delta after 100 iterations. All runs converged around 15 iterations as shown by the red dotted line.



Lambda	Max Filter	Cluster Size	Run Time	Delta (after 100 iterations)	Change in Delta (after 100 iterations)
0.050	100k	Large	32 minutes	625.38	1.73%
0.650	100k	Large	32 minutes	631.39	1.83%
0.075	100k	Large	33 minutes	627.90	1.86%
0.100	100k	Large	32 minutes	613.85	1.91%

Result Sample

The result output is the current Cost (calculated by the Cost Function) and Cost Delta (change in cost) per iteration:

Iteration(96) Cost: 1330938.3228906982 Delta: 663.4889064121526
Iteration(97) Cost: 1330287.8295311453 Delta: 650.4933595529292
Iteration(98) Cost: 1329649.9319488101 Delta: 637.8975823351648
Iteration(99) Cost: 1329024.2448263355 Delta: 625.6871224746574
Iteration(100) Cost: 1328410.397023809 Delta: 613.8478025265504

Conclusions

As highlighted, ALS can be used to estimate every user-item pair efficiently to approximate a new matrix of predicted ratings. Using this approach we were able to accurately generate a predicted ratings matrix for each user-item pair which can be used to predict movies that a user may like.

The appeal of the ALS solution is in being able to solve matrix factorization for One Class Collaborative Filtering (OC-CF) efficiently since it is hard if not impossible to solve using gradient-based method. However, this approach is prohibitively expensive for most real-world datasets. A second (and more holistic) approach is to use the P_u and Q_i as features in another learning algorithm, incorporating these features with others that are relevant to the prediction task.

A significant extension that could be later to the project is to add Block ALS, it is a method of partitioning that sends only the appropriate user (P) or item (Q) columns to each partition, not the entire P and Q matrix. Another possible extension would be to provide functionality that would return a rating recommendation for any user/item combination or return a list of recommended movies a given user may like that they have not seen.