10605 BigML Assignment 7:

Distributed SGD for Matrix Factorization on Spark

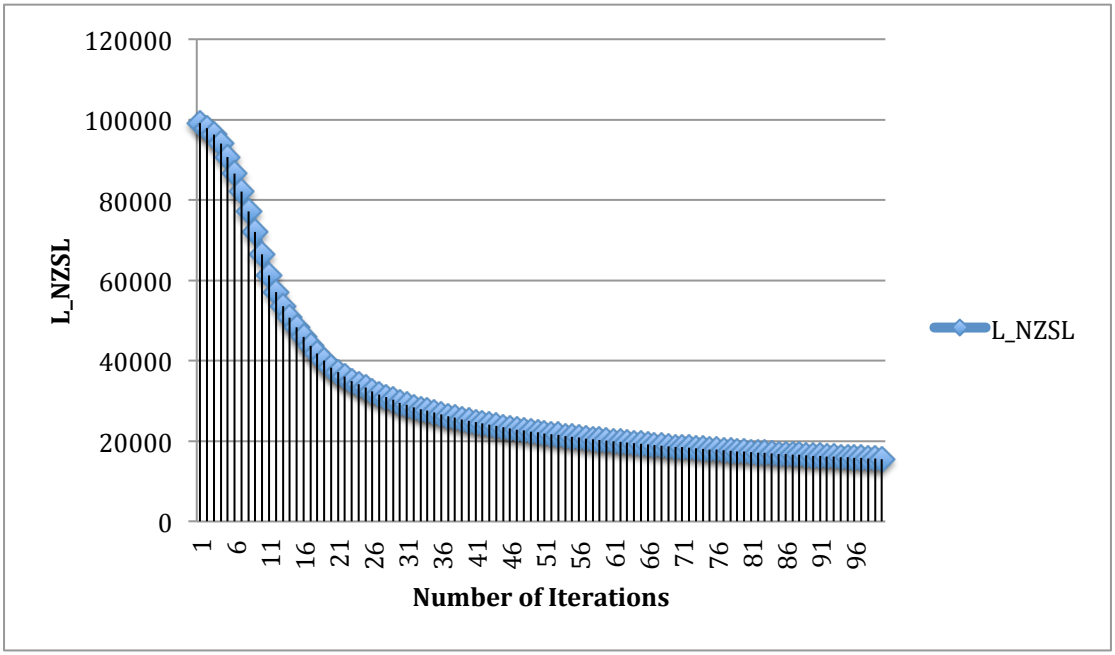
Andrew Id: juipinw

Name: Jui-Pin Wang

Due: 4/16/2015

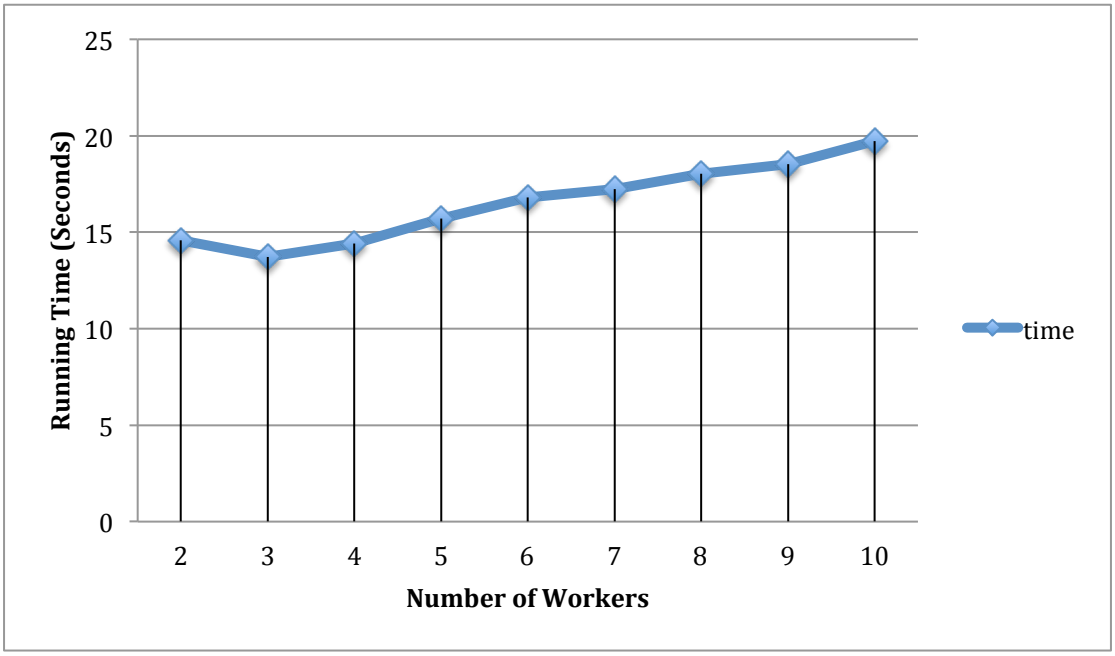
(1) Did you receive any help whatsoever from anyone in solving this assignment?
Yes. I have discussed with San Chuan Hung about the implementation.
If you answered 'yes', give full details: _____ (e.g. "Jane explained to me what is asked in Question 3.4") .
(2) Did you give any help whatsoever to anyone in solving this assignment?
Yes. I have discussed with San Chuan Hung about the implementation.

Exp1.



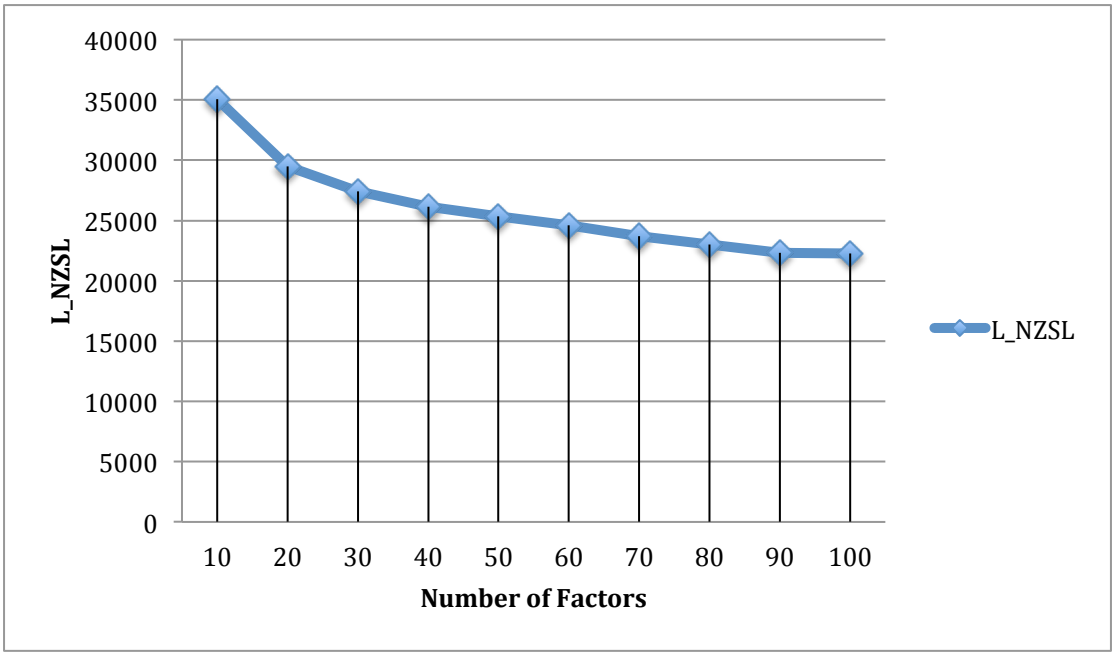
From this figure, we could see loss decreases as the number of iterations increases. More iteration means more SGD we could perform. Therefore, it provides more chance to minimize the loss function. As a result, more iteration could lead to less loss.

Exp2.



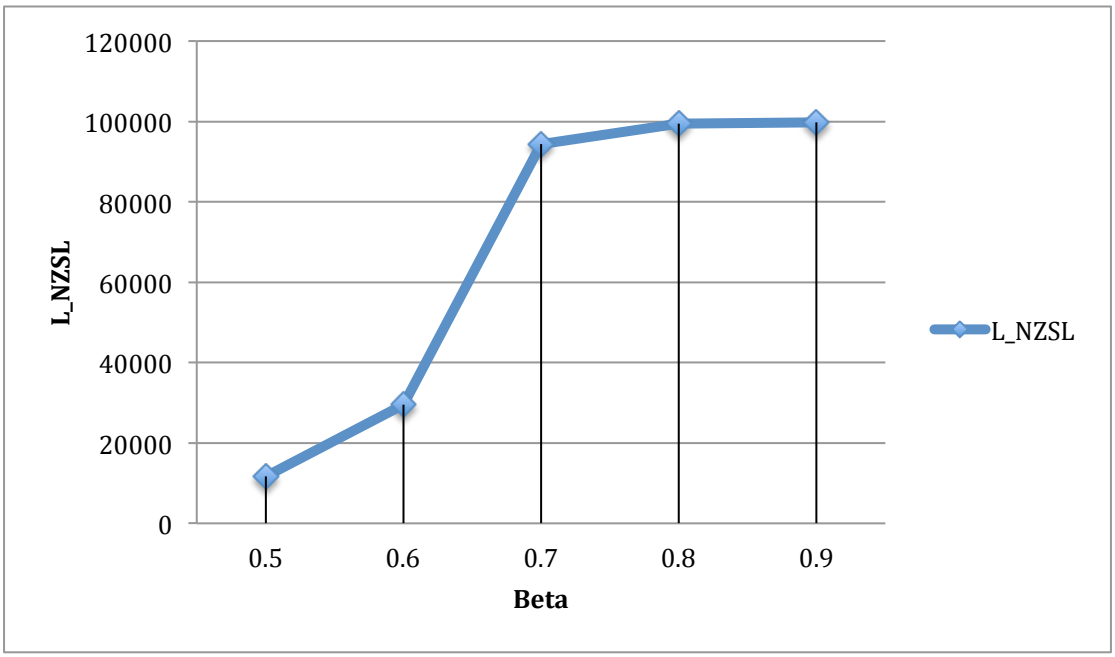
From this figure, we could see running time increases as the number of workers increases. We believe this is because the "nf_subsample.csv" is so small that the overhead of more workers like communication cost is larger than the benefit from parallel computing. Therefore, the running time is more when the workers are more.

Exp3.



From this figure, we could see loss decreases as the number of factors increases. The reason is that the more latent factors, the more complex model we use. And because the model is complicated with more latent factors, it is easier to fit the training data. As a result, we could get lower loss with more factors.

Exp4.



From this figure, we could see loss increases as Beta increases. It makes sense since we found that when Beta is large, the learning rate decay so fast that the step size of each gradient is too small. Small step sizes of gradient make our model stop much earlier before arriving good local minimum of the loss function. As a result, larger Betas have larger loss since their step sizes are too small to be optimized well.

Deliverables:

Q1. Is there any advantage to using DSGD for Matrix Factorization instead of Singular Value Decomposition (SVD) which also finds a matrix decomposition that can be used in recommendation systems?

For DSGD in MF, we could divide the original matrix into disjoin blocks as a strata and calculate the gradient for each block. As long as these blocks satisfy the interchangeability, we could calculate and update the gradient for each block independently. Therefore, it is easy to compute in parallel for DSGD in MF. On the other hand, when we do decomposition in SVD, we don't have this good property thus it is hard to work in parallel for SVD.

Q2. Explain clearly and concisely your method (used in the code you have written) for creating strata at the beginning of every iteration of the DSGD-MF algorithm.

Suppose we have B blocks, I initialize an integer array named strata where strata[i] = i for i = 0 to B-1. After each iteration, I will add one to all number in strata[]. If strata[k] = B, then let strata[k] = 0. By this operation, we could iterate every block after B iterations.

Q3. If you were to implement two versions of DSGD-MF using MapReduce and Spark, do you think you will find a relative speedup factor between MapReduce and Spark implementations, keeping other parameters like the total number of iterations and number of workers fixed? Which implementation do you think will be faster? Why? If your answer depends on any general optimization tricks related to MapReduce or Spark that you know, please state them as well.

The implementation of Spark should be faster than MapReduce. The main reason is that Spark could stay data in memory, which is a very good mechanism for the iterative computation for calculating gradient in DSGD-MF.
On the other hand, MapReduce has to write and read data to the disk after every iteration. As a result, MapReduce would be slower since the speed of access disk is much slower than memory.

Q4. Match the Spark RDD transformations to their descriptions. No explanation required.

(1) coalesce	(e) Reduce the number of partitions in the RDD possibly without shuffling data
(2) repartition	(a) shuffle data randomly into a given number of partitions
(3) groupWith	(d) Merge multiple RDDs based on common keys
(4) cache	(b) Pin the RDD in memory for repeated use
(5) foldByKey	(c) Reduce all values per key to yield a single aggregate value for each key

Q5.

Link of github:

https://github.com/r44/10605_hw7