Amazon Product Ratings Prediction at Scale

DS 410 Final Project Report

Yuya Jeremy Ong & Yiyue Zou

Introduction

Online retail services on the web has significantly dominated the digital landscape within the past decade, making it easier for consumers to have easy access to a wider variety of products without having to physically go to a retail location to make their purchases. Even more recently, services for providing fresh produce and groceries have even become more prevalent, offering a wider variety of products for consumers to select from. Consequently, this has allowed for these web services to aggregate and record massive amounts of purchase records as well as product reviews in large scale data warehouse infrastructure. Companies such as Amazon, Jet, and Walmart have already aggregated massive amounts of data from their consumers, which can be analyzed to further observe various trends and patterns to provide a better customer experience, especially through the implementation of a better product recommendation system.

In our project, we propose to utilize the dataset aggregated from Amazon.com, curated by McAuley et. al [1, 2], to analyze data across products, reviews, meta-data information and related products towards the implementation of a ratings prediction model which bridges as a stepping stone for a complete product recommendation systems. The majority of these recommendation based models in the industry today utilize a variant of the collaborative filtering algorithm, which usually is based on a content agnostic analysis of purchase behaviors to drive recommendation models for consumers. However, there have been other variations of these models which not only uses user interactions but other implicit features such as customer sentiment, product descriptions and other indirect features which can be combined together in an ensemble fashion, therefore yielding a hybrid collaborative filtering model to better drive customer recommendations [3, 4, 5]. However, as this is beyond the scope of our project, we will not further consider complex feature modeling processes.

In this project, we utilize a large scale data analytics platform, Apache Spark along with the corresponding Machine Learning library, MLlib, to construct and build our large scale data analytics pipeline. We plan to utilize implementation techniques which allow us to leverage the power and scalability of both the hardware and software level parallelism has to offer to handle the scale of our models. Furthermore, we focus on analyzing and evaluating the various performance metrics and program flexibility associated with these big data analytics tools, including cluster configurations, runtime performance and scalability through systematic experimentation of parameters.

Objectives & Project Criteria

0.1 Project Objectives

To achieve our objective of constructing constructing a scalable rating analysis pipeline for our models, we have focused on the following sub-components of our project which can help aim to achieve this:

- Implement a Collaborative Filtering based modeling pipeline in Apache Spark using MLlib, which predicts the user's rating of a product based on analyzing the user to product interactions through their rating behaviors.

- Systematically execute various runtime and codebase configurations across different optimization parameters and evaluate their performance using the SparkUI tools provided by Apache Spark.
- Conduct a systemaitc investigation of performance and bottlenecks to search for potential areas of optimizations and improvements to the original implementation.
- Perform a trade-off analysis of the various methods based on quantitative and qualitative metrics obtained from our experiments.

0.2 Model Objectives

In an industrial setting, we would typically set objectives as per requirements by the company based on their expectations in a production setting. Similarly, we have set the following objectives and expectations for our model:

- Build a scalable model architecture, such that retraining of the model can be performed on a daily basis to ensure product recommendations reflect the new purchasing patterns from the previous day.
- Our prediction models should perform fairly well, given the dimensional and sparsity constraints of the dataset.
- Devise a dynamic system architecture by which the performance of our models can be scaled proportionally by additional hardware resource defined optimizations.

Notably, the last objective listed above will be our primary focus throughout the project, where we have explore the scope of the performance and optimization capabilities in-depth.

Amazon Dataset

The dataset consists of approximately 142.8 million records (approximately 20 GB) of product reviews aggregated between May, 1996 to July, 2014 and split into three different key components including the reviews, product metadata and link information. In our work, we will be utilizing the preprocessed version of the dataset, where the author of the dataset has removed duplicates of the data. Permission to utilize the full-scale dataset has been authorized by the author through email contact. The newly scaled down dataset contains 84.3 million records (approximately 18 GB). The review component consists of the customer's ratings (scaled from 1 to 5), text reviews and the helpfulness vote associated with the product. The product metadata provides further information associated with the product's description, category, price, brand and image features. Finally, the dataset also consists of link information pertaining to the information on what consumers have also viewed or purchased Amazon. The provided dataset was given in the form of a compressed gzipped json file.

Due to issues of feasibility and scope of our project, we have utilized only a subset of the dataset which corresponds to the given departments within Amazon. The primary components of the dataset are divided into the product reviews and the product metadata themselves. The schema and the dataset features are detailed in Appendix A (provided in the last pages of this report). In our project we utilized the largest subset of the entire dataset from the Books department. The total number of products we have in the dataset is approximately 2,370,585 records and the total number of reviews were 8,898,041 records.

Prior to performing any modeling processes, we performed a high level exploratory analysis of our dataset to observe the general distributions to obtain a better intuition of the dataset to identify some of the potential patterns to lookout for when building our models. We perform an exploratory analysis of our dataset to devise any patterns and attributes from our samples. One of the critical areas we have looked into was the distribution of the reviews against the products and customers. The following figures below shows the various statistical summaries for each of the analyzed relationships we have observed:

	User-Review	**Product-Review**
Count	603,668	3,679,982
Mean	14.7399	24.1806
Std Dev	51.7764	66.3029
Min	5	5
Max	23,222	7,440

Figure 1: Review Distribution Statistics

Count	4,756,837
Mean	0.7349
Std Dev	0.3427
Min	0.0
Max	1.0

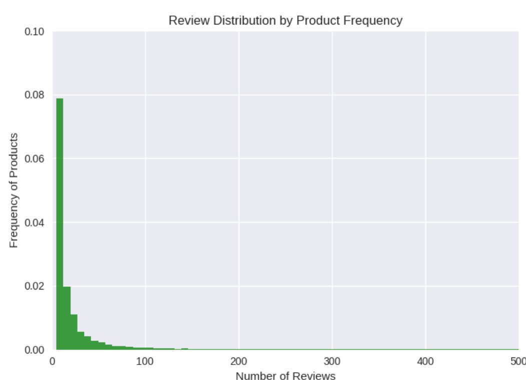
Figure 2: Helpfulness Statistics

1 Star	323,833
2 Star	415,110
3 Star	955,189
4 Star	2,223,094
5 Star	4,980,815

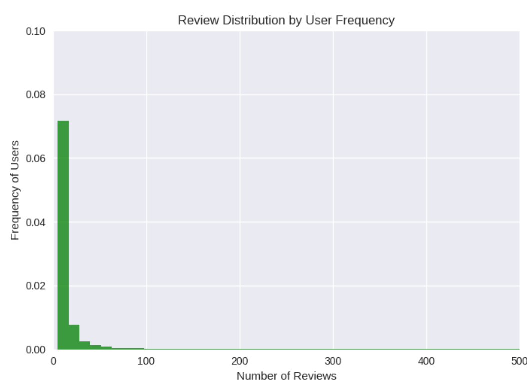
Figure 3: Ratings Distributions

Based on the statistical summaries above, we can make some key observations about the dataset and the implications towards our modeling process. First one of the key areas to note is the apparent skew in the User to Review and Product to Review distributions. This skew is can be observed graphically as a distribution plot as seen in figure 4, where the plot has a strong positive skew. This consequently indicates to us that the data is significantly sparse as the majority of the reviews with higher value counts tends to have very few users or products. Sparsity in our features can yield poor performance both for our model and the computational performance.

Figure 4: Review Distribution by Product and User Frequency



(a) Product Frequency



(b) User Frequency

Therefore, we have devised several different feature engineering methods which will allow us to make use of the data we have at hand or use other features which can help to better build our models. One suggested method of reducing the number of sparsity in the datset was to discard a certain portion of the dataset by it's frequency. This will allow us to ensure that the model we produce will generate a significantly more confident predict and thus provide with better results. However, in terms of interoperability of model, we will be very limited to a much smaller sample size and potentially hurt the results.

In deciding a filtration criteria, we have experimented with various values and have found that we can reduce the dataset by removing any user or review that has less than $ceil(mean) - 1$. We have specifically chosen this metric as, use the use of the mean will ensure that the confidence of our predictions would be at a much higher threshold. Thus, based on our summary statistics provided in Figure 1 and using the proposed function, we filter any users that have less than 14 product reviews and any products which have less than 24 product reviews in total.

As a result of performing such filteration processes, we were able to significantly reduce the dimensionality of our feature size from 2,221,487,373,976 to 10,426,542,170 records, approximately an 82.58% decrease in our dimension space, which will largely impact our processing time significantly. We present the effects of the filtration process in both counts and percent change based metrics under Figure 5. As observed, the number of records significantly decrease by cutting off a significant portion of the data, while shifting all the other parameters (except for the maximum) to a higher value.

	User-Review	**Product-Review**
Count	134,354 (−47%)	77,605 (−98%)
Mean	40.9745 (+178%)	77.3837 (+220%)
Std Dev	105.5534 (+104%)	131.0095 (+98%)
Min	15 (+200%)	15 (+200%)
Max	23,222 (0%)	7,440 (0%)

Figure 5: Filtered Review Distribution Statistics

Problem Formulation

In the following section, we will formally define the problem for our large scale modeling objective and algorithm, or the Collaborative Filtering algorithm by Zhou et. al [6], which we will utilize to predict user product ratings based on the user to product reviews.

Given a set of m users, $U = \{u_1, u_2, u_3, ..., u_m\}$ and a list of n products, $P = \{p_1, p_2, p_3, ..., p_n\}$ with a matrix $A = \{r_{i,j}\}$. Each user, $u \in U$, provides a review between 1 to 5, denoted by $A_{i,j}$ for the ith user and jth product. We also define the function $r(u_i, p_j)$, which is a simple conditional function indicating whether or not a user has provided a rating - if so, then the resulting value would be 1 and conversely 0. Our objective, based on these variables is to predict the missing values of the ratings between the users and the products.

The implementation of Collaborative Filtering algorithm within Apache Spark's MLlib implementation utilizes a Matrix Factorization technique based on the Alternating Least Squares method. We define the matrix A as an estimated inner product between the matrices X (for the user parameters) and Y (for the product parameters) by XY^T with the objective of our model attempts to learn through the Alternating Least Squares method to learn the hidden latent variables associated between each of the matrices X and Y.

Therefore we must minimize the following objective function, denoted as J, to achieve a sufficient model that best approximates A. The first part of the function, $\|R - XY^T\|_2$, describes the loss between the actual ratings matrix and the inner product of the learned parameters, while the second half of the function, $\lambda(\|X\|_2 + \|P\|_2)$, describes the associated regularization function with the regularization term λ. The addition of the regularization function prevents the model from overfitting the parameters entirely.

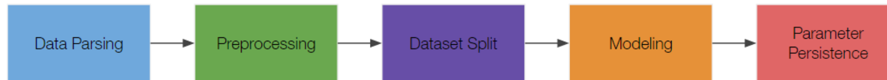
$$\min_{X,Y} J = \|R - XY^T\|_2 + \lambda(\|X\|_2 + \|P\|_2)$$

For our data analytics pipeline, the MLlib library requires the user to only provide the number of latent factors for which we have fixed to 10. In our modeling process, we also defined to have the model run for 20 epochs with a learning rate $\alpha = 0.01$. For the purposes of keeping our focused on a performance oriented analysis, we will be keeping all of our model based parameters fixed to ensure that we can obtain consistent measurements across each of our performance evaluations.

Processing & Modeling Pipeline

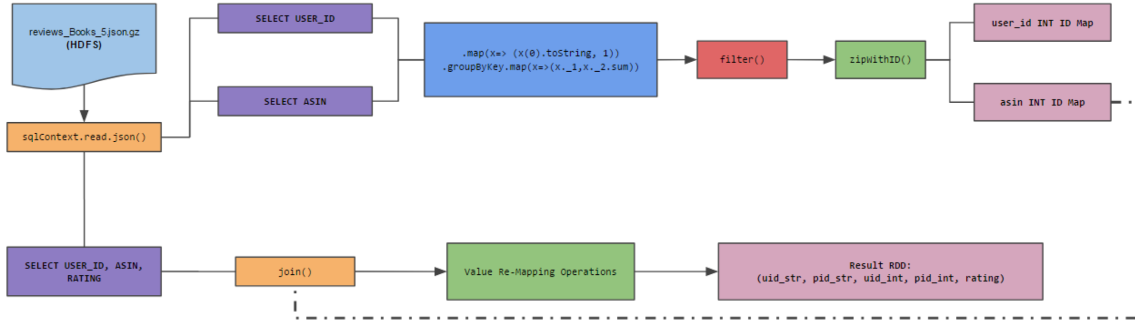
In this section, we outline the modeling process and the data flow pipline by Apache Spark throguh the analysis of the RDD Lineage Chain. The performance analysis of our modeling process can be broken down essentially into two different pieces - the Data Preprocessing Phase and the Model Construction phase. We will closely look at the operations involved in the process of the data transformations performed on the data, then further analyze the pipeline to seek out potential bottlenecks and avenues for further optimizations.

Figure 6: Data Preprocessing RDD Pipeline



The first half of our data processing pipeline entails the dataset initialization and preprocessing phases. In this phase, we load the data into memory from an HDFS based file storage structure using a JSON parser provided by SparkSQL. Prior to loading the data, we then perform our preprocessing routine where we perform the conditional filtering process based on the proposed filtration criteria presented in the previous section. The subsequent RDDs are then zipped with an integer based ID as this was a required parameter type as part of the modeling framework of MLlib. The corresponding integer representations of the user ID and the product ids were then remapped to the associated ratings through a join function and then structured in the final RDD tuple structure as denoted in the figure below.

Figure 7: High Level Data Modeling Pipeline



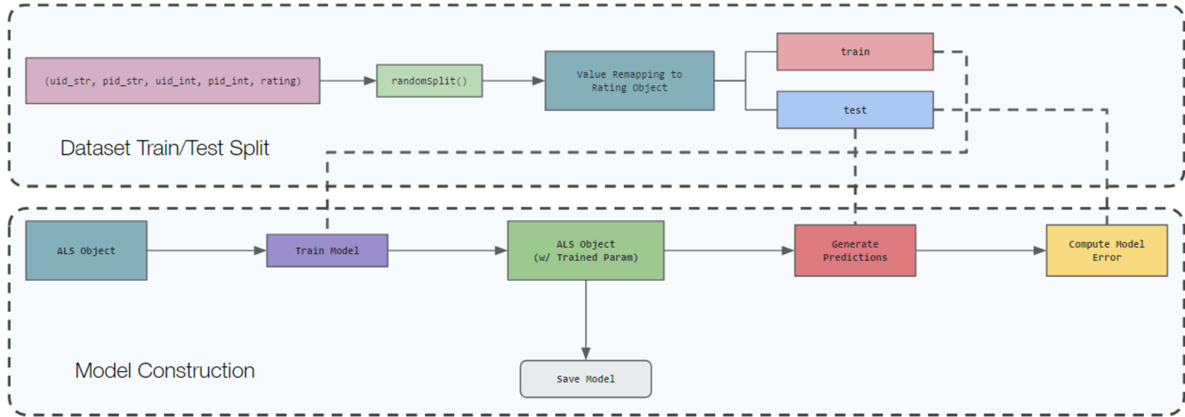
After these data process routine, we perform model construction for the data. First, we use the randomSplit function on the on the RDD which guarantees a (pseudo-stochastic) sub-sampling of the given dataset into two categories: training and test. The training set contains the data to feed into the model, and the test set contains data to monitor the training progress by computing the model error of predicted value with true value.

For the training data feeding into the model, we apply Alternating Least-Squares(ALS) algorithms. The Alternating Least-Squares algorithm from a traditional single machine implementation is shown as follows, where for each of the latent variables each of the parameters are learned by fixating one of the parameters and adjusting it by the values associated to the error. As noted by the psuedo-code implementation below, we can see that each of the inner for-loop parameters are considered to be independent and do not have any dependence to each other as the errors are computed independently from each other.

```
initialize U, P
for 1 to num_epoch (or until convergence):
    //Update User Parameters
    for j = 1 to n:
        // Update k weights for users (keeping product weights fixed)
    end for
    // Update Product Parameters
    for j = 1 to m:
        // Update k weights for products (keeping user weights fixed)
    end for
end for
```

Therefore, this presents the possible opportunity to be able to perform an asynchronous computation of the weight values simultaneously for all of the latent parameters. All further optimizations and computation of the parameters are done by Apache Spark's MLlib, which utilizes the proposed methods from Hu et. al [7].

Figure 8: Modeling RDD Pipeline



Model Performance Evaluation

In this section we evaluate the quantitative aspects of our model. For our evaluation criteria, we based it on the Root-Squared Mean Error metric, common to evaluating regression based models. To establish a baseline comparison of this metric, we will use this metric to compare the performance difference between to evaluate the effectiveness of reducing the dataset feature space as presented in our preprocessing phase. Each reported metrics were based on a 5-fold cross validation process where we have taken the average of the errors across each folds (this was only done for the model evaluation metrics, and not for the actual performance evaluation). The following figure below shows the two different runtime errors. As observed below, the filtration process of the model has significantly improved the results of our model.

	Without Filtering	**With Filtering**
RMSE	3.3724	1.8256

Figure 9: RMSE Comparison of Models

Performance Optimization

The following section will cover our systematic process and methodology for optimizing the original implementation of our program. In each of our optimizations, we focus on a specific element of the optimization criteria by making systematic observations and adjustments to the parameters that are provided to us. We look into the following three different areas where we hope to see performance gains from different methods of improving the overall performance.

To evaluate the performance of our models, all of our analysis will be based on empirical results and raw historical runtime data obtained from the SparkUI profiling tool. The basis of our evaluation will be evaluated and measured by execution runtime and average memory consumption per node. Within each model we will be systematically evaluating different configurations for execution parameters utilized under the spark-submit command through fixating one parameter and spanning our runtime analysis across each parameter spaces. Our two primary parameters we will be utilizing will be between number of clusters and the number of core executors per node. Prior to obtaining the best possible configuration per model, we can

then perform a heterogeneous model comparison of each of our methods against each other to evaluate the overall best runtime to evaluate the general behavior of our program. As an additional evaluation criteria, we will also provide include some qualitative results from our models to observe how certain quantitative trade-offs can affect the overall ”experience” of the results obtained from our model. Through this method, we are allowed to standardize our methodology in understanding the various trade-off behaviors between each associated parameters and resulting output.

Baseline Implementation

To make comparisons on performance metrics, we evaluate all of our metrics based on a standard or control parameter. In this cause our control parameter will be based on an execution of the spark-submit job without passing any parameters in. As a result, Spark will decide on a default parameter value for the job execution and will execute with some pre-determined value. For our baseline, we run the program using the spark-submit command without passing any parameters as follows:

```
spark-submit target/scala-2.11/review_predict_2.11-0.1-SNAPSHOT.jar
```

Upon submitting the job, Apache Spark allocates **two nodes** and takes approximately **20 mins** to complete the execution of the entire program. Thus, any further optimizations, for it to be considered valid, should at least perform better these metrics presented in this section.

Spark-Submit Parameters

For our first set of optimization, we have chosen to tune the spark-submit parameters and observe the various effects it has on the overall runtime performance of our model construction process. No additional code-based optimization techniques have been employed in this process. We systematically tested various configuration using an automated bash script, which permutes through various configurations for the following optimization parameters: Executor Nodes, Executor Cores and Memory (Same Values Used for Master and Executors). Figure 10 presents a graphical view of the performance comparison metrics based on comparing memory and cores to the execution nodes. Figure 11 shows the complete table of results of our experiments on the various spark-submit parameters.

Figure 10: Comparison of Execution Nodes vs Memory and Cores

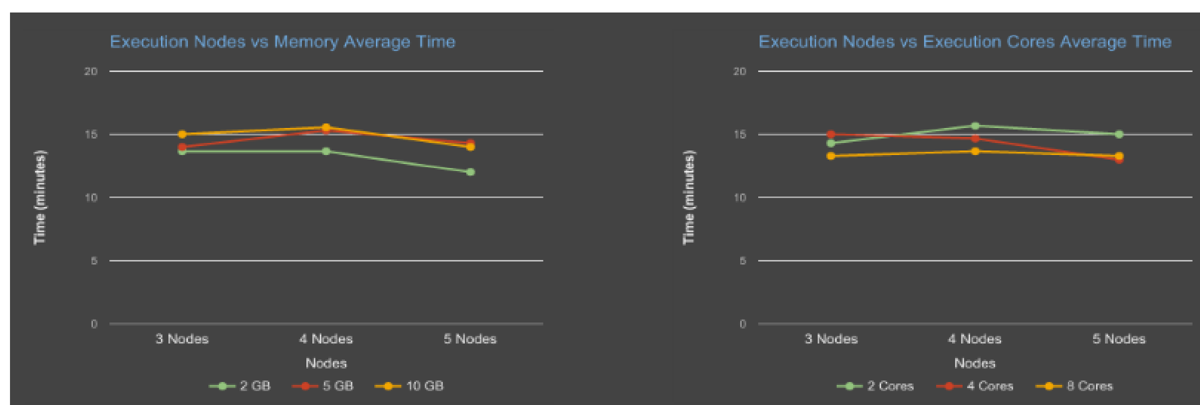


Figure 11: Spark Submit Performance Comparison

a) 3 Node Performance

Cores/Memory	2 GB	5 GB	10 GB	**Avg.**
2 Cores	13 m	15 m	15 m	14.3 m
4 Cores	15 m	14 m	16 m	15 m
8 Cores	13 m	13 m	14 m	13.3 m
Avg.	13.67 m	14 m	15 m	14.2 m

b) 4 Node Performance

Cores/Memory	2 GB	5 GB	10 GB	**Avg.**
2 Cores	15 m	15 m	16 m	15.67 m
4 Cores	13 m	17 m	14 m	14.67 m
8 Cores	13 m	14 m	14 m	13.67 m
Avg.	13.67 m	15.3 m	14.67 m	14.54 m

c) 5 Node Performance

Cores/Memory	2 GB	5 GB	10 GB	**Avg.**
2 Cores	13 m	16 m	16 m	15 m
4 Cores	**11 m**	13 m	15 m	13 m
8 Cores	12 m	14 m	14 m	13.3 m
Avg.	12 m	14.3 m	15 m	13.76 m

Persistence

Persistence is another fundamental feature provided by Apache Spark to temporarily store the RDD onto memory or disk, depending on the parameters passed within the persistence settings. As Apache Spark performs operations based on the execution of a operation (i.e. collect, take, etc.), repeating such operations can be considered computationally and temporally expensive. Therefore, to avoid such redundant operations, the use of persistence can help to let Spark explicitly know that certain RDD types are going to be reused in future operations, and therefore optimizes the code runtime significantly. In our process for investigating whether persistence is a valid optimization, we evaluated our implementation and spotted key areas in which the introduction of persist can perhaps improve the performance. In each of the phases presented previously, we evaluated two key areas in which we hypothesized that adding persist can potentially improve the overall runtime.

First, within the Data Loading and Preprocessing phase, we sought to implement the persistence prior to loading the data into memory as the RDD operations from this RDD splits into two separate operations which require two redundant calls to reprocess the RDD again from SparkSQL then through the conversion to an RDD form. Prior to completing the preprocessing phase, we have decided to introduce an unpersist operation to ensure that we make more space for future persist operations that may potentially require more memory space. Another key location that we have decided to inser persistence was after the randomSplit process where we split the dataset into the training and testing set. We especially placed the persistence operation onto the RDD used for training as the model will often make calls to this data upon every epoch during training. However, looking into MLlibs implementation for ALS, it has already implemented a persistence operation onto the data when the training data tuples are passed in and led us to hypothesize that applying persistence to the training externally from the ALS

model was unnecessary and can be a redundant optimization.

To validate our experiments, we evaluate the persistence across the different persistence options available for us to use. We record each of the runtime for each experimental run for the different persist configuration and present our results in Figure 13. Furthermore, to keep the experiments consistent, we use the designated runtime parameters derived from the previous experiment (shown in Figure 12) as a control parameter to ensure that we can perform a valid comparison throughout each of the different persist options.

Parameters	**Value**
Driver Memory	10 GB
Executor Memory	10 GB
No. of Executors	5
Executor Cores	8

Figure 12: Spark-Submit Job Parameters

Persistence Type	**Runtime**
NONE	14 mins
DEFAULT	13 mins
MEMORY ONLY	14 mins
MEMORY AND DISK	15 mins
MEMORY AND SER	14 mins
MEMORY AND DISK SER	13 mins
DISK ONLY	14 mins

Figure 13: Experimental Runtime Results

Block Parallelization

Block parallelization is a parameter which is only special to MLlib's Collaborative Filter model object. The blocks within the context of this model object is an internal operation which is used to exploit the use of sub-matricies to break the problem down into smaller pieces, and therefore allocate smaller shards of the data across many threads and nodes. As a result, increasing the number of shards should hypothetically improve the result since there would be reduced wall-time per each process as all threads would be working on some component of the sub-matrix. Our experimentation will be based on evaluating the optimal number of "blocks" we will be utilizing. By default, Apache Spark provides an automated configuration of the number of blocks for the model to use by setting a -1 parameter as the arguments into the ALS modeling object. However, the library also provides the user to try out various values of this block parameter, and thus we have tried some key number of blocks to see how well this would perform. We also utilize a similar fixed spark-submit parameter, as presented in the figure below along with the result of our experiments:

Parameters	**Value**
Driver Memory	10 GB
Executor Memory	10 GB
No. of Executors	5
Executor Cores	8

Figure 14: Spark-Submit Job Parameters

Block Parameter	**Runtime**
-1	14 mins
10	14 mins
25	18 mins
50	14 mins
100	13 mins

Figure 15: Experimental Runtime Results

Discussion

In this experiment, we discuss the various observations and key findings from our experiments based on the empirical values and measurement metrics we have used to evaluate the performance of the various configurations.

From our first experiments concerning the model error performance, we have seen that the suspected issue of sparsity is seen to be reduced by our explicit data filtration process. Hence, we can see the effects of reducing our error significantly. However, in a more realistic setting, the amount of sparsity would be even higher and would need a much better method/algorithm to alleviate this. Perhaps we believe that using some other key user to product features other than ratings or a combined feature representation can help to potentially reduce such sparsity in our dataset.

In our next set of experiments where we have tested various configurations, we can primarily conclude that all optimizations we have made in our part have contributed greatly towards performance gains of up to approximately 50%. This is especially significant when we want to perform this analysis on an even larger set of users and products as this shows the promising capacity at which Apache Spark can handle large datasets.

From looking at the various runtime options in the spark-submit, we can see that the best runtime achieved in this entire experiment came from the configuration of using 5 nodes, 4 executor cores and 2 GB executor memory per nodes, thus yielding the fastest runtime of 11 mins. Furthermore, by analyzing the graphs under Figure 10, we can see that increasing executor nodes and executor cores have a significant effect in improving the runtime of a distributed job. Hence, if this were in an enterprise setting, the implication of adding more commodity hardware and memory can improve the overall execution of the system and vastly improve performance as we scale proportionally to the amount of data we are processing.

However, looking at the other optimization parameters that we have utilized including persistence and block parallelization, we can see that we have not been able to make significant optimization strides as compared to the effects of changing spark-submit parameters. Although all of the metrics have indeed improved the overall runtime performance of the pipeline, this is certainly not a definitive statement we can establish as we have fixed the parameters based on the best run provided run time - which can be called as a potential confounding/lurking variable in our experiments. To better understand the true nature of the effects of these other implementations, we can use the real baseline implementations utilized to see the real effects of the runtime of the modeling process.

Conclusion

In our project we have successfully implemented a large scale model based system capable of generating predictions for user ratings on products. From our empirical observations, we have shown significant improvements through various methods of optimization including from both a hardware and software based perspective and have successfully achieved our listed objectives. In short, we show that augmentation to parameters, external from the program's algorithmic process, can improve the model's runtime execution performance.

References

[1] J. McAuley, C. Targett, Q. Shi, and A. Hengel. 2015b. *Image-based recommendations on styles and substitutes.* In Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR15). ACM, New York, NY, 4352. DOI: http://dx.doi.org/10.1145/2766462.2767755

[2] J. McAuley, R. Pandey, and J. Leskovec. 2015. *Inferring Networks of Substitutable and Complementary Products*

[3] D. Gurini, F. Gasparetti. 2013. *A Sentiment-Based Approach to Twitter User Recommendation*

[4] G. Linden, B. Smith, and J, York. 2013. *Amazon.com Recommendations - Item to Item Collaborative Filtering*

[5] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. 2001. *Item-based Collaborative Filtering Recommendation Algorithms*

[6] Y. Zhou, D. Wilkinson, R. Schreiber, R, Pan. 2008. *Large-Scale Parallel Collaborative Filtering for the Netflix Prize,* Proc. 4th Intl Conf. Algorithmic Aspects in Information and Management, LNCS 5034, Springer, 2008, pp. 337-348. the Netflix Prize

[7] Y.F. Hu, Y. Koren, and C. Volinsky, *Collaborative Filtering for Implicit Feedback Datasets,* Proc. IEEE Intl Conf. Data Mining (ICDM 08), IEEE CS Press, 2008, pp. 263-272.

Appendix A: JSON Dataset File Schema

```
# Books Review Dataset
root
 |-- asin: string (nullable = true)
 |-- helpful: array (nullable = true)
 |    |-- element: long (containsNull = true)
 |-- overall: double (nullable = true)
 |-- reviewText: string (nullable = true)
 |-- reviewTime: string (nullable = true)
 |-- reviewerID: string (nullable = true)
 |-- reviewerName: string (nullable = true)
 |-- summary: string (nullable = true)
 |-- unixReviewTime: long (nullable = true)

# Books Metadata Dataset
root
|-- _corrupt_record: string (nullable = true)
|-- asin: string (nullable = true)
|-- brand: string (nullable = true)
|-- categories: array (nullable = true)
|    |-- element: array (containsNull = true)
|    |    |-- element: string (containsNull = true)
|-- description: string (nullable = true)
|-- imUrl: string (nullable = true)
|-- price: double (nullable = true)
|-- related: struct (nullable = true)
|-- title: string (nullable = true)
```