Name: Pankaj Bhootra
Matriculation No.: A0144919W
NUSNET ID: E0009011

CS4225 Assignment 2 Report

Task 1

Comparisons on programming with Hadoop and Spark:

While Hadoop and Spark do many of the same tasks, there are also few noticeable differences. The first thing I noticed was that Spark did not have its own file management system, so it still relies on Hadoop's Distributed File System (HDFS) or other similar solutions. Hence, I think that the pairing of Spark's optimal data processing engine with Hadoop's DFS is a smart, efficient and powerful solution for big data applications.

However, Spark is very different from Hadoop's MapReduce as it has a much faster real-time data processing capability as opposed to MapReduce that is more disk-bound and batch-oriented, giving Spark solutions a faster execution time of 10-100 times compared to the Hadoop counterparts.

I especially noticed this while solving Task 1 of CommonWords example, as we wrote both Hadoop and Spark version for this problem. I found some noticeable conveniences in working with Spark instead, as mentioned below:

1. Much less verbosity: The code written using Spark is a lot less verbose compared to Hadoop, as in case of Hadoop we have to define Mapper and Reducer classes for each sub-part of the problem that is a MapReduce job in itself. In comparison, Spark provides a simpler RDD data type interface, on which functions like map and reduce can be called just like normal methods. Hence, a MapReduce operation on an input dataset can be performed in just one line of code in Spark using RDDs and Functional Programming, whereas for Hadoop it would take many lines of multiple Mapper/Reducer classes.
2. Expressive API/ease of use: Spark comes with many predefined MapReduce protos such as groupBy, aggregateByKey, sortByKey, flatMap, mapValues, reduce, reduceByKey, etc. In case of my CommonWords solution for Hadoop, I had to write multiple MapReduce classes for even the simple tasks such as sorting the final output by word count. However, in case of Spark, the sortByKey library provided allows me to achieve the same objective in a much simpler manner (also in just one line!). Hence, this allows the programmer to focus more on solving the actual problem rather than spending time figuring out how to solve the very simple parts of the problem.

The CommonWords task took around 200 lines of code in Hadoop as opposed to 40 lines approximately in Spark, making the latter much more convenient and programmer-friendly.

Comparisons on runtime execution with Hadoop and Spark:

Spark not only provides greater convenience and flexibility in terms of programming, but as mentioned before, it provides a much greater boost in performance thanks to its data processing engine. While MapReduce jobs are slow and batch-oriented, Spark jobs run in real-time and can easily transfer results from one operation (such as map, reduce or sort) to the other without any intermediate I/O operations. Eliminating the intermediate I/O operations which involve reading from and writing to disk saves a lot of time in case of Spark as the result of one operation is directly fed to the next one.

In case of the CommonWords implementation in both Hadoop and Spark, the Spark job took only around 5-6 seconds whereas the Hadoop solution, which had multiple MapReduce jobs in it, took around 1 minute, giving a 10x boost in time efficiency.

This makes sense as Spark processes all the different steps for CommonWords in memory. This leads me to believe that Spark's in-memory processing can deliver near real-time analytics for many large datasets as compared to MapReduce's batch-processing mechanism. MapReduce was never really built to provide blinding speed, because its original goal was to just continuously crawl and index information from websites (during its development at Google) and there were no "real-time" requirements for this data.

Other differences:

Besides its ease of use and performance, another advantage of Spark over Hadoop is that it comes with user-friendly APIs across multiple programming languages such as Scala (its native language), Java, Python and even SQL. In fact, Spark SQL is so conveniently abstracted that on its surface, it appears identical to SQL 92, so there is no additional learning curve required to learn how to use it.

In terms of costs, Spark requires a lot of RAM to compute processes in-memory, whereas Hadoop MapReduce requires faster disks since its processing is disk-based and batch-oriented. Also, Spark's technology reduces the number of systems required to solve the same task vs. Hadoop's MapReduce, so it runs with significantly fewer systems (though, with additional RAM).

I also want to share a quote that highlights all the above benefits of Spark: "Spark has been shown to work well up to petabytes. It has been used to sort 100 TB of data 3x times faster than Hadoop MapReduce on one-tenth of the machines. This feat won Spark the 2014 Daytona GraySort Benchmark."

Hence, Spark is not only efficient in terms of programming, but also more time efficient in runtime execution, learning and costs. It provides a significant improvement in data processing and can be used in tandem with Hadoop's Distributed File System to solve many big data applications.

Task 2

Analysis of Results:

[Please refer to the “output.txt” file submitted for Task 2 for the clustering results.]

Looking at the results of K-Means Clustering on the QA_data.csv dataset provided to us, the following observations were made:

1. There are certain domains that receive a higher median/average score than the others, indicating that these domains are more popular and trending on StackOverflow. For example, the “Machine Learning” domain has 3 clusters in the result (same for all the other domains too), and across these clusters, the minimum median score it received was 232 (average score 275.8) while the maximum was 4441 (average score 5007.0). This shows that there are higher scored answers for questions in the Machine Learning domain as opposed to several other domains, showing that people tend to ask and answer more questions related to Machine Learning rather than other domains. Apart from Machine Learning, even domains like Big Data and Deep Learning were found to be highly scored domains.
2. Similar to point 1, there are domains which tend to receive much smaller scores than the others, for example Software Engineering and Architecture domains. These insights are indicative of how old domains like Software Engineering receive lesser attention and have fewer questions compared to more recent domains like Deep Learning.
3. There are also some domains which are pretty evened out (i.e. they get both lower scores and higher scores). Eg. Compute-Science and Internet-Service-Providers. Although these are old domains, the results show that there is still some trending activity in terms of question-answering for these domains, which makes sense as these are very generic domains and have sub-domains like Software Engineering and Machine Learning, some of which may be popular while others may not.
4. Another insight was how certain domains receive a lot more questions than the others. Example: Big Data has 400+ questions across its clusters whereas Software Engineering has only 216. This is again indicative of the amount of activity for each domain, which allows us to infer which domains are more popular on StackOverflow.
5. For each cluster, we notice that the percentage of the dominant domain in the cluster comes out to 100%. This shows that the K Means algorithm is very effective in clustering the questions correctly by domain.
6. Another thing to notice was that the clusters with extremely high scores had very few questions as compared to clusters with mid-range scores and low scores. This shows that high scored questions get clustered in their own unique cluster for that domain. In general, questions of the same domain with similar scores tend to get clustered together.
7. There is some disparity between median and average score of each cluster, as average is much greater than the median scores in some cases. This shows that certain questions tend to get much higher scores than other questions in the same cluster.

Analysis of the parameters in K-Means:

The values of the 4 key parameters were adjusted to different values and their effect on clustering and performance was examined. The following observations were made for each parameter:

1. *DomainSpread* : This determines how far apart questions from different domains are in the vector space. The initial value of 50000 provided perfect clustering as the all questions were correctly clustered by domain in very few iterations of the K Means. Since this value was initially large enough, further increasing it did not make a difference to correctness of clustering or iterations for convergence. However, decreasing it too much caused incorrect clustering of some questions (% of dominant domain in some clusters goes down). Also, the K-means took longer to converge.

2. *kmeansKernels* : This determines the number of clusters, and it is required that it should be a multiple of the number of domains. The initial value of 45 formed 3 clusters for each domain (partly due to how my code samples initial centroids equally from each domain), and the questions with similar scores were grouped in the same cluster. For instance, questions with very high scores in a domain get clustered separately from the ones with mid-range scores. I also got correct clustering for this value set to 15 and 75. However, if this value is less than the number of domains, then it is certain that the K-Means will fail to cluster questions by domain correctly as the number of clusters will be too less.

3. *kmeansEta* : This determines the convergence criteria. If the total distance between the new centroids and the corresponding initial centroids in a K-means iteration is less than this value, then we reached convergence. The initial value of 20.0D was good enough as the code reached convergence in very few iterations and clustered all questions correctly. However, increasing this value too much will allow the K-means to converge too early, giving incorrect clustering. On the other hand, decreasing this value too much will take K-means longer to converge (though correctness is still ensured).

4. *kmeansMaxIterations* : This is the maximum number of iterations of K-Means that is allowed before the code terminates on the last set of calculated centroids. The initial value of 120 was sufficient as the code reached convergence before reaching this many iterations. Usually, for many ranges of the above 3 parameters, the K-Means reaches convergence within 30 iterations. Hence, to ensure correctness, this value should be higher than 30.

Hence, in general, the parameter *DomainSpread* makes the utmost difference out of the 4 parameters above as it determines how much the questions from different domains are separated in the vector space. However, the values set for the other parameters are also important in ensuring the correctness of the K-means clustering results.

Further discussion on the System Performance:

While the predefined set of parameters can significantly alter the algorithm performance in terms of time efficiency, there are also system-level factors that come into play. For instance, the cluster configuration, or the number of data nodes on which the job runs concurrently matters as increasing the number of nodes provides the Spark job with more RAM for parallel computation, hence giving faster results. Other factors such as CPU speed also affect the time, but Spark jobs are designed to run on commodity hardware only, with the idea being to reduce the requirement of high speed disk storage by performing parallel computations in-memory instead.

Ideally, running this K-Means job does not require too many data nodes since Spark does not perform intermediate I/O operations on disk. However, increasing the amount of RAM on each node will speed up the processing.

There are also other optimizations that can be introduced. Using DataFrames as opposed to RDDs will provide query optimization and low garbage collection overhead. Bucketing is another way to provide optimizations on queries, aggregations and joins as it introduces a data partitioning mechanism.

Using optimal data format as input will also highly optimize the performance. Spark supports many formats such as CSV, JSON and XML. However, it is most optimized for PARQUET with snappy compression, which is the default behavior in Spark 2.0. PARQUET stores data in columnar format and is highly optimized in Spark.

Caching can also help to optimize time efficiency and Spark provides native methods such as `.cache()` for this purpose.

In general, any approach that uses memory efficiently will be key to optimizing the execution of Spark jobs.
