Machine Learning with Dataflows

Ang Li

Abstract

Machine learning is one of the fashionable technologies over the past few decades. As training a reliable model requires extensive data set, the data processing efficiency becomes one crucial factor that impacts the performance of ML applications. On the other hand, researchers introduce RDDs and MapReduce, which are scalable and fault-tolerance dataflow models, to reduce the response latency of the search engine caused by the rapid growth in internet data volume. People then realize the feasibility of reducing the training time cost of ML algorithms by utilizing the dataflow models. In this paper, we implement and evaluate k-means clustering with RDDs and MapReduce. The results indicate that the implementation with new dataflow models has the functionality expected and higher efficiency than the traditional approach when handling an extensive data set.

1 Introduction

Machine learning is one of the fashionable technologies that have enormous commercial potential over the past few decades. As the majority of machine learning algorithms, including linear regression, logistic regression, and k-means clustering, require extensive training data set to improve their predictions or decisions, computational resources become the crucial factor that restricts the performance of one application. Meanwhile, search engine developers are also looking for an economic mechanism to boost their computational efficiency due to the rapid growth in internet data volume. Researchers from the University of California, Berkeley, and Google, LLC propose Resilient Distributed Datasets (RDDs) and MapReduce, which are dataflow models based on cluster computing, to overcome the high latency in the big data processing.

In this paper, we utilize the Apache Spark framework, which provides the interface for dataflow models mentioned above, to investigate how data parallelism affects the performance of k-means clustering in terms of accuracy and efficiency. We implement the algorithm with two approaches: Spark and matrix solution. By comparing the output of different implementations, we verify the precision of both methods and conclude that the implementation with Spark has a higher efficiency than another when handling an extensive data set. RDDs, combined with MapReduce, offers a practical and robust mechanism that lessens a large amount of training time cost for ML algorithms.

The remainder of this paper is organized as follows. Section two introduces the background of RDDs, MapReduce, and k-means clustering. Section three describes how we implement k-means clustering with two different approaches. Section four contains evaluations of our implementations. Section five concludes and discusses future works.

2 Background

2.1 RDDs

Resilient Distributed Datasets (RDDs) are fundamental abstractions of Spark, which represent immutable, partitioned elements collections. RDDs are fault-tolerant because of their immutability and coarse-grained transformation. They distribute elements to memories on clusters to improve the computational efficiency by taking advantage of data parallelism.

Zaharia et al. [2] proposed RDDs to ameliorate the inefficiency performance software frameworks had when handling two types of applications: iterative algorithms and interactive data mining tools. However, they showed that RDDs were expressive to capture several existing cluster computing models, which include MapReduce. Some user applications built with Spark are in-memory analytics, traffic modeling, and Twitter spam classification.

2.2 MapReduce

MapReduce is a programming model and a transformation for distributed computing. It contains two processes, namely Map and Reduce. The Map function generates intermediate key/value pairs based on the input key/value pair, while for each key, the Reduce

function merges all its associated intermediate values. MapReduce contributes to data parallelism as workers among different CPU cores on clusters could process portions of RDDs synchronously and later combine their outputs.

Dean and Ghemawat [3] proposed MapReduce as a concise and powerful interface that hid details of cluster computing (parallelism, fault-tolerance, data distribution, and load balancing) from users, which allowed the latter to express their simple computations interested. Some practical applications are the distributed grep/sort, count of URL access frequency, the reverse web-link graph, and term-vector per host.

2.3 K-means clustering

The k-means clustering is an unsupervised learning algorithm applied for classify clusters of elements in a dataset. In the beginning, the algorithm randomly initializes centroids of clusters. It then iteratively assigns elements to their closest centroid and computes the new centroid of each cluster until derived centroids converge as guaranteed. Notice in an edge case, one centroid could have no element assigned. We take the straightforward solution in this paper, that is, repeat the algorithm over again.

Stuart Lloyd of Bell Labs [4] proposed the standard algorithm of k-means clustering in 1956 as a technique for pulse-code modulation. However, over the past few decades, k-means clustering has widespread adoption in ML field, including vector quantization, cluster analysis, and feature learning.

3 Implementations

We implement k-means clustering on Python 3.8 with PySpark API. Python has a concise language structure, an object-oriented approach, and various supported libraries, which make it a practical method for conducting experiments and presenting results.

3.1 Implementation with Spark

The implementation with Spark interprets coordinates of sample points as RDDs. It iteratively performs the MapReduce process to derive new centroids until the algorithm converges. The map function assigns each sample point to its closest centroid and returns (centroid index, (1, coordinate)) pair, where we utilize the first element in the value pair to count the size of one cluster. For each centroid, the reduce function sums their associated values. The program then derives new centroids by dividing the sum of coordinates by the number of assigned sample points. Despite the execution of the local mode, Spark partitions RDDs and allocate workers among cores. We cover the detail in the evaluation section.

```
def label(point):
    global centroids
    ...
    return (idx, (1, point))
```

```
def k_means(self, points, threshold):
    labeled = data.map(self.label)
    reduced = labeled.reduceByKey(
    lambda a, b: tuple(map(sum, zip(a,b))))
```

3.2 Implementation without Spark

The implementation without Spark assigns each sample point to its closest centroid and organizes resulted clusters into matrixes, where each row contains one sample point coordinate. For every matrix, the implementation divides its column wised summation by the number of rows to derive a new centroid. We repeat the above procedure until the algorithm converges.

```
def k_means(points, threshold):
        labeled[idx] = np.vstack((labeled[idx],
        points[i]))
        centroid = (np.sum(labeled[i], axis=0)/
        labeled[i].shape[0])
```

4 Evaluation

We conduct all experiments mentioned in this paper with the following presuppositions to simplify the procedure: 2D sample points are placed based on the normal distribution of real centroids with a standard deviation of 0.06, and the latter are generated randomly within the range of -0.5 to 0.5. Both implementations assume convergence of the algorithm when the square error between centroids obtained in two consecutive iterations is below 1E-16.

4.1 Verification of Functionalities

The following plots comprehensibly illustrate the relevance among sample points, real and derived centroids in different implementations. For each plot, we derive three centroids from 3,072 sample points. Figure.1 demonstrates the initial phase of k-means clustering, where it initializes random centroids. Figure.2/Figure.3 and Figure.4/Figure.5 are results of implementations with/without Spark, respectively.

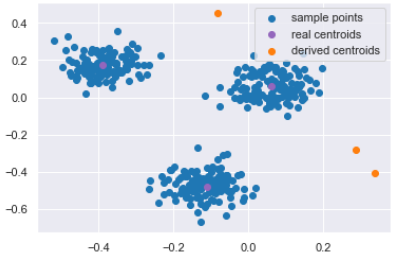

Figure 1: initial phase

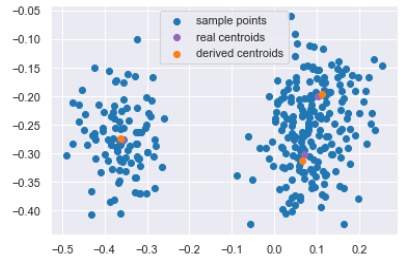

Figure 2: Spark output, 1

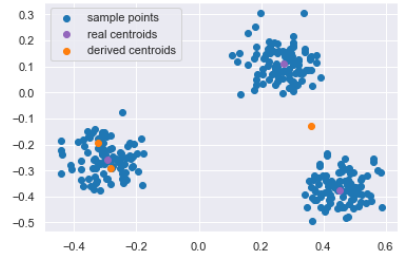

Figure 3: Spark output, 2

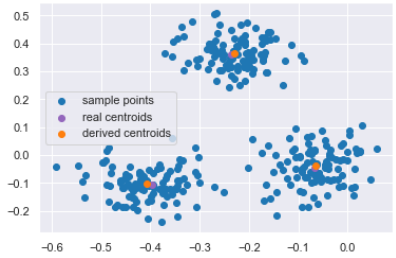

Figure 4: matrix output, 1

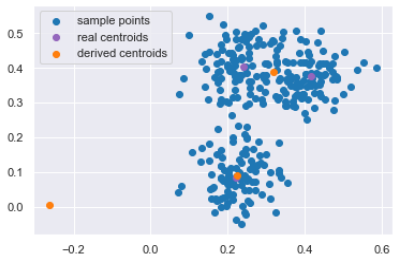

Figure 5: matrix output, 2

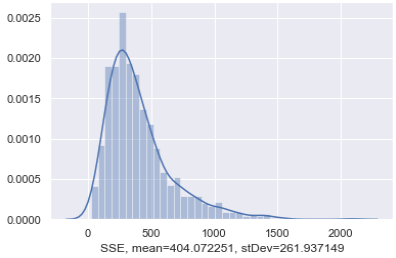

Figure 6: SSE, initial phase

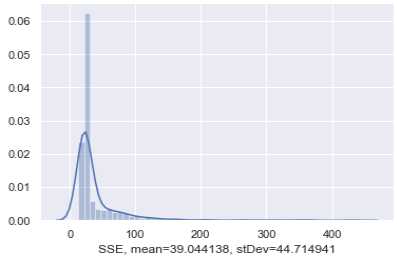

Figure 7: SSE, Spark

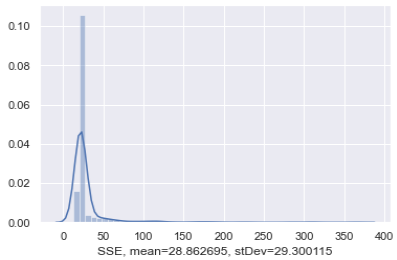

Figure 8: SSE, matrix

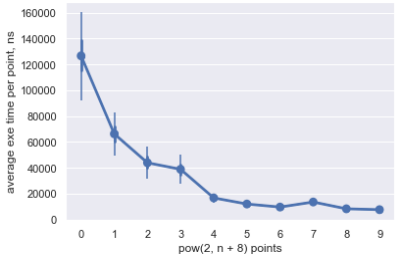

Figure 9: Spark, 32 repetitions

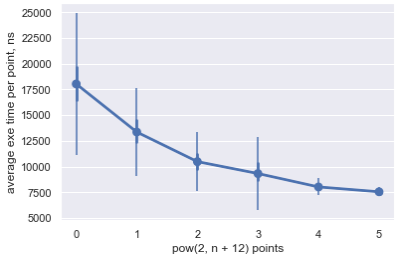

Figure 10: Spark, 64 repetitions

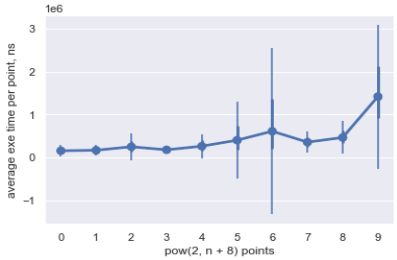

Figure 11: matrix, 32 repetitions

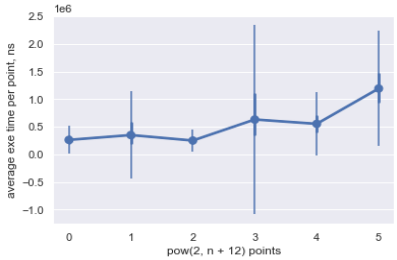

Figure 12: matrix, 64 repetitions

Theoretically, the closer derived centroids locate to real centroids, the better performance one implementation would have. However, there are cases where derived centroids are distant with real centroids due to the randomized initialization of centroids. For example, in the Figure.3, the implementation assigns clusters on the right side to one single centroid, while two centroids split the left side cluster. Since the interval between clusters is large, the assumption of assigning points to their closet centroid is still holding. Moreover, derived centroids on the right side of Figure.2 are further apart from each other compared to real centroids, as the result of the collision between two clusters. The implementation assigns part of one cluster to the unexpected centroid. Despite the unpredictability of derived centroids, it is explicit that both implementations make practical assignments after we examine the coordinate plot.

We also apply the sum of the square error (SSE) between derived centroids and their assigned points to estimate functionalities of implementations. We retrieve the following distribution plots by inspecting the initial phase of k-means clustering and implementations with/without Spark, respectively. For each plot, we derive three centroids from 3,072 sample points and repeat each execution for 1,024 times

The implementation with Spark reduces the mean value and standard deviation of SSE by 90.34% and 82.93%, while the implementation without Spark reduces the numbers by 92.86% and 88.81%. Although both implementations interpret the data as float_64 type, utilizing Spark resulted in a slightly worse performance

(2.52% SSE difference), which could be due to the rounding scheme diversity among dataflow models. For the implementation with Spark, workers round and update the return value each time they receive a new (key, value) pair in the reducing process. In contrast, the implementation without Spark organizes related data in the same matrix, where only one column wised summation would lead to the precision loss. Overall, the considerable SSE reduction in both implementations verifies their functionality.

4.2 Benchmark

We benchmark the execution time to estimate performances of implementations with various sizes of the data set. We first investigate the implementation with Spark by deriving two centroids from 2^8 (256) to 2^{17} (131,072) sample points, doubled in each iteration, and repeating each execution for 32 times, as shown in Figure.9.

The average execution time per sample point has a downward trend as the size of the data set increases. In the beginning, the time cost for lazy evaluated RDDs object initialization, data partition and communication, and worker allocation are significant overheads that slow the process. When the size of the data set grows enormously, the overhead is insignificant compared to the computation time cost, while parallelism contributes to the high efficiency of data processing. Meanwhile, the performance bottleneck is shifting from the dataflow model to hardware configuration, which means the slope becomes flat and should be positive ultimately. We do not cover the final stage due to our hardware limitations.

To verify the parallelism of data processing in the implementation with Spark, we monitor the CPU usage of our machine. According to the Table.1, Spark wakes multiple cores (one, three, five, and seven) to perform the map/reduce process synchronously. According to the Spark documentation [1], the number of partitions in RDDs returned by MapReduce transformation is set to the number of cores on the local machine by default, which means there is a maximum of eight workers operating at the same time in our case. Notice that the implementation with Spark has an overall higher CPU usage compared to the one without Spark, as the former requires extra computational resources to manage the data communication and worker allocation.

Table.1: CPU usage, percentage

core idx	0	1	2	3	4	5	6	7
Spark	70	15	55	10	50	10	45	10
Matrix	30	0	40	0	30	0	45	0
Idle	10	0	10	0	5	0	5	0

We also notice that an exception raised at the seventh iteration, where the resulted mean value (13,612.80) and standard deviation (3,302.29) are higher than the previous term (9,707.23 with 1,235.40). However, according to Figure.10, further inspection eliminates the deviation, where we increase the repeating time for each execution to 64. One explanation is that the result retrieved from 32 times of repeating fails to cover the enormous variance and could not represent the real distribution in the seventh iteration.

We also conduct the same experiment on the implementation without Spark. The average execution time per sample point has a blurry upward trend as the data set size increases. Theoretically, the rising of n sample points contributes to $2 \cdot n$ more comparisons, which would not affect the average execution time per sample point in the idea case. Therefore, matrix computation becomes a significant overhead that slows the efficiency and makes the slope positive. However, we could not draw any conclusion because of the high variance in the result.

By comparing the average execution time per sample point, we conclude that the implementation with Spark has better performance (over 90% running time reduction) in terms of efficiency than the one without Spark when handling an extensive data set. The inference does not apply to the data set with a small size due to the Spark object initialization overhead. We do not include this overhead in our benchmark experiment as the program only initializes the object once at the beginning, and the running time cost could be over five seconds.

5 Conclusions and Future Works

In conclusion, RDDs and MapReduce are two practical and powerful dataflow models that boost the efficiency of data processing by applying cluster computing. We utilize the above models to investigate how data parallelism affects the performance of k-means clustering in terms of accuracy and efficiency. The results indicate that the implementation with new dataflow models has the functionality required and higher efficiency than the traditional approach when handling an extensive data set.

For future works, we are looking for an opportunity to investigate the performance of Spark on the cluster mode, which provides more computational resources compared to the local machine we currently deployed. We do not inspect our implementations with the data set larger than megabytes due to hardware limitations. In contrast, cluster computing is commonly processing petabytes of data.

We are also interested in implementing the neural network with Spark to investigate how cluster computing boosts the efficiency of more sophisticated machine learning algorithms. However, it is tricky to integrate the gradient descent into MapReduce as researchers design the transformation for simple computations, and we may end up applying other dataflow models (perimeter server architecture, for example) or utilizing the existing ML library.

References

[1] Apache Spark. https://spark.apache.org/docs/latest/index.html

[2] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., . . . Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, 12*. doi:10.5555/2228298.2228301

[3] Dean, J., & Ghemawat, S. (2008). MapReduce. *Communications of the ACM, 51*(1), 107-113. doi:10.1145/1327452.1327492

[4] Lloyd, S. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2), 129–137. doi:10.1109/tit.1982.1056489