Improving Performance of K-Means on Spark

Dareddy Manoj Reddy, Tushar Sudhakar Jee

*{mdareddy, tjee}@ucla.edu*

**Abstract**

In our project, we aim to improve the performance of K-Means in Spark. K-Means is a popular unsupervised machine learning algorithm that can provide insights about the data. K-Means is implemented in Spark using the MLlib library. We implemented two versions of updated K-Means in Spark that leverages optimizations in center update step and another algorithm called Yinyang K-Means. We ran experiments on large datasets using 3 nodes on Amazon EC2. We were able to achieve an overall speedup of 1.98x for the center update version and 1.6x using the Yinyang K-Means.

**Introduction**

*Spark*

Apache Spark [3], is a cluster computing framework that provides an interface for programming clusters with implicit fault tolerance and data parallelism. Compared to Hadoop, it supports in-memory cluster computing which is ideal for iterative machine learning algorithms. Data in Spark is represented using partitions as Resilient Distributed Datasets(RDD). RDDs are immutable and lazily evaluated multiset of data items distributed across clusters. RDDs are evaluated lazily using transformations and fault tolerance is achieved by keeping track of lineage information.

*K-Means*

K-means is a popular clustering algorithm. Its vanilla form is referred to as Llyod’s Algorithm. Given, n points each of dimension d, the goal is to generate k groups of points based on Euclidean distance. It involves two steps: assignment and update. In the assignment step, each point is assigned to the closest initial cluster center. The update step then involves calculating the cluster center by computing the mean of all points assigned to that specific cluster.

MLLib

MLlib [5] is Spark’s machine learning library that contains various classification and clustering algorithms. The K-Means in this library is implemented using a scalable version of K-Means++. A sample program is written as follows:



Figure 1: MLLib K-Means

The implementation is based on the following workflow:

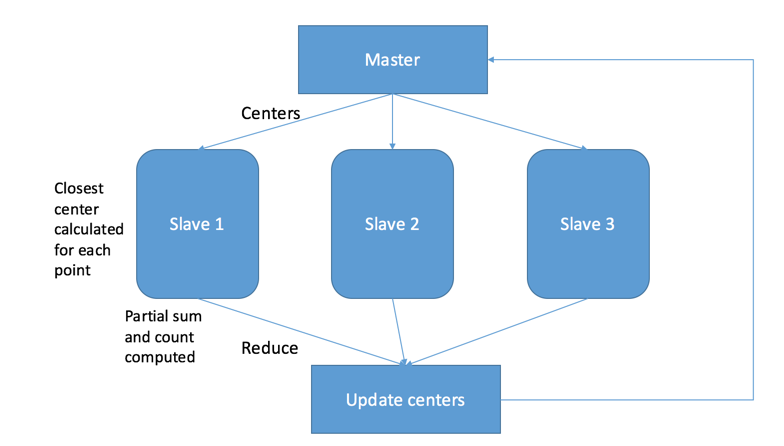


Figure 2: MLLib K-Means implementation workflow

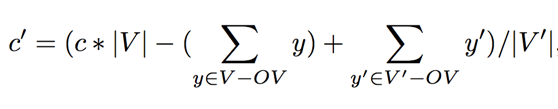
Initially, the data is stored as an RDD and the centers are initialized using the user specified method. In the next step, the centers are broadcasted to the workers and mapPartitions is called on the RDD. In the map partitions method, the closest center is computed for each point and a partial sum, count is computed by each worker. Each worker then returns an iterator that contains the partial sum and count for each center (k values) based on the points in their partitions.

The reduce step then computes the sum across all iterators for the k cluster centers. The center update simply involves scaling the total sum by the total number of points assigned to that cluster. The updated centers are then used for the next iterations and repeated until a user specified number of iterations.

**Optimizations**

*Update Step*

The update step of K-Means can be optimized as follows [2]:



In the above equation, c’ and c represents the new and old cluster center respectively. The symbols V’ and V represent the set of points that were assigned to the cluster in the current, previous iteration. OV represents the intersection between the two sets V’ and V.

The new center update is essentially subtracting the points that have left the cluster and adding the new points that have been added in the current iteration from the overall sum in the previous iteration, followed by division of the new cluster size. This can improve performance since most points do not change cluster assignments after a few iterations and hence, can result in the fewer computations on each worker machine.

This approach requires an overhead of storing the cluster assignment for each data point. We initially transformed the dataset RDD into a new RDD that consists of a (point, cluster assignment) pair where the initial cluster assignment for all points it set to -1. Points are represented as an array of doubles using the class VectorWithNorm.

In each iteration, a map function is applied on the RDD to update the cluster assignment.

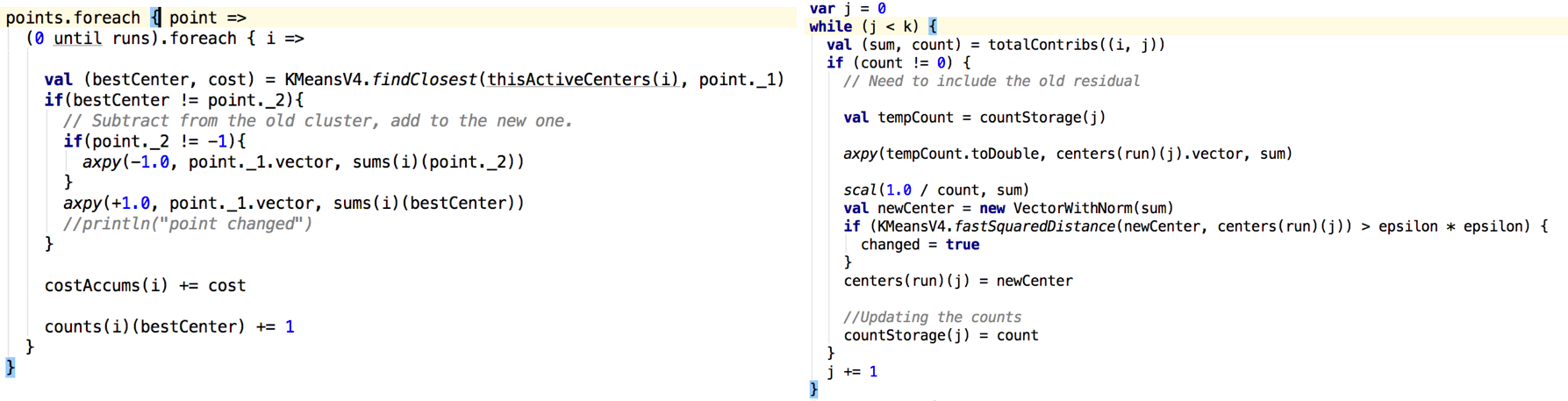


Figure 3: Center Update Implementation

The above image on the left illustrates the conditional statement that checks to see if the current “bestCenter” for a given point is the same as the one in the previous iteration. Only if the cluster assignment has changed, the point is added to the partial sum for the new center and subtracted from the old center. The reduce function remains the same as the one in the original MLlib’s K-Means class.

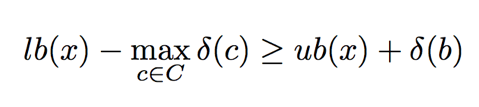
The image on the right displays the center update function which involves adding the previous sum of points for a cluster so as to adhere with the equation mentioned above.

*Yinyang*

Another way to improve the K-Means algorithm is to optimize the cluster assignment step. There have been many approaches in the literature that attempt to tackle this problem and we have chosen to adopt a recent paper at ICML named Yinyang K-Means [2]. The essential idea is to use upper and lower bounds to reduce the number of distance computations. It relies on triangle inequality which states that for any points a, b, c and using the Euclidean distance:



The paper uses a concept called global filtering that can determine if a point changed its cluster assignment based on a simple conditional statement. A point x does not change its cluster if [2]:



where b(x) - closest center assigned to point x, ub(x) - upper bound, lb(x) - lower bound of point x and delta(b) - change in distance of center b. The Yinyang paper modifies the above global filtering and introduces two types of filtering, namely: group and local. We only implement the group filtering aspect in Spark which updates the various bounds as follows [2]:

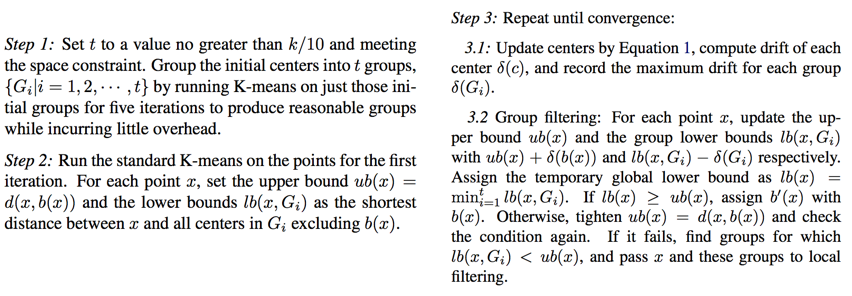


Figure 4: Yinyang K-Means Algorithm

The above algorithm was implemented in Spark using the similar approach to the center update optimization. The main idea here involves transforming each point to a RDD of tuples, where each tuple is represented as below: (point, cluster assignment, upper bound, lower bound). The lower bound value is an array since there exists a lower bound for every point, group combination.

The center update step from the previous section was used as well. Instead of using local filtering, the standard distance computation to each center was used for such points.

**Experiments**

The datasets were generated using make\_blobs function in scikit-learn in python. Varying the value of n\_samples(n) and n\_features(d) allowed us to get the dataset size varying from 250Mb to 4Gb. The data is generated as follows:

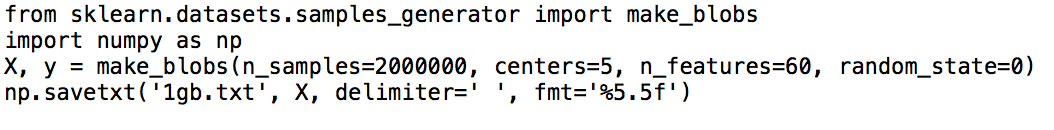


Figure 5: Scikit make\_blobs function

We set up an Amazon EC2 instance using a Large Amazon Machine Image (6.5 ECUs, 2 vCPUs, 2.4 GHz, Intel Xeon E5-2676v3, 8 GiB memory, EBS only) assigning the number of instances to be three. We then created a key-pair, set the Amazon security credentials and launched the instances. After generating the datasets, a master and 2 slaves were initialized and in order to run our version of K-Means a spark job was submitted to the master.

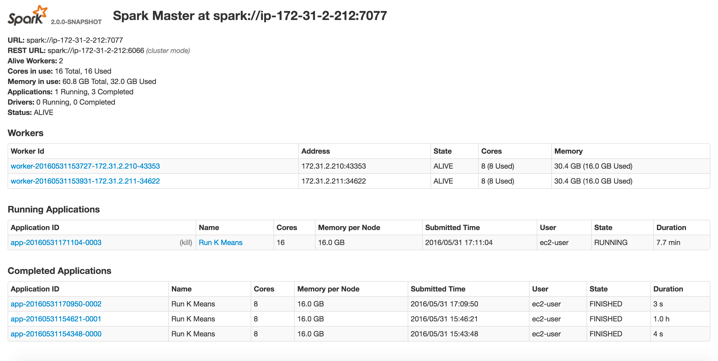
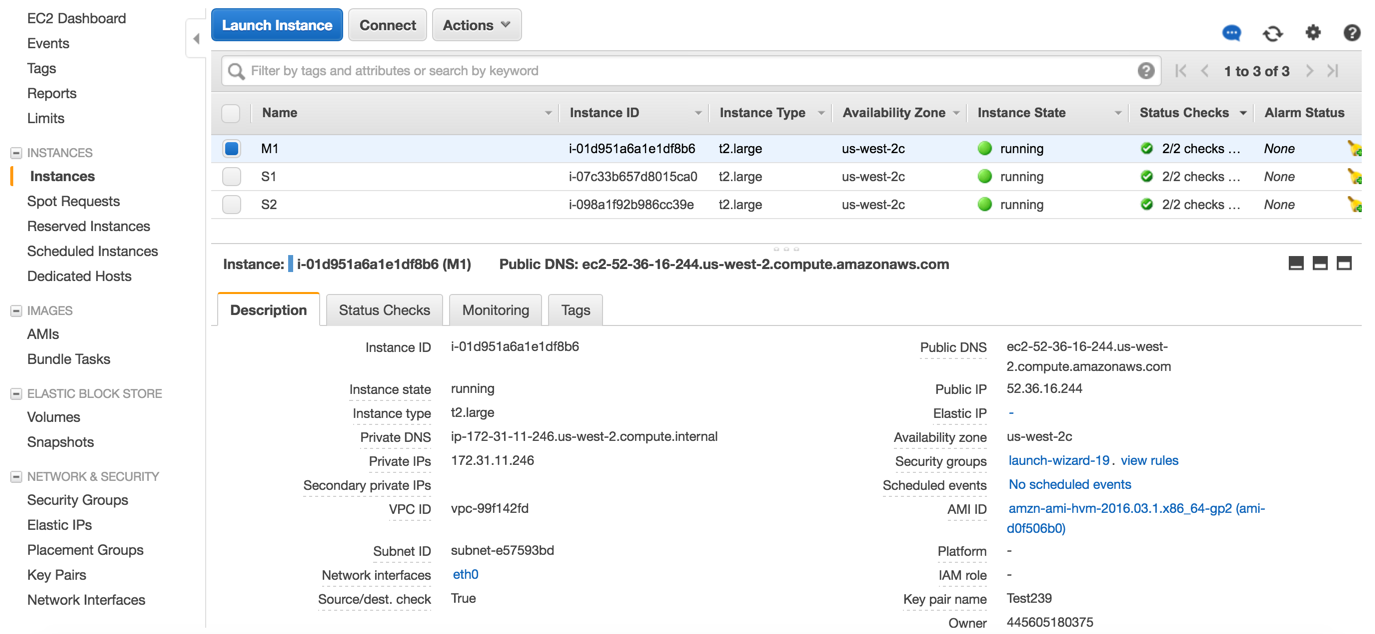


Figure 6: Running Spark on Amazon EC2

**Results**

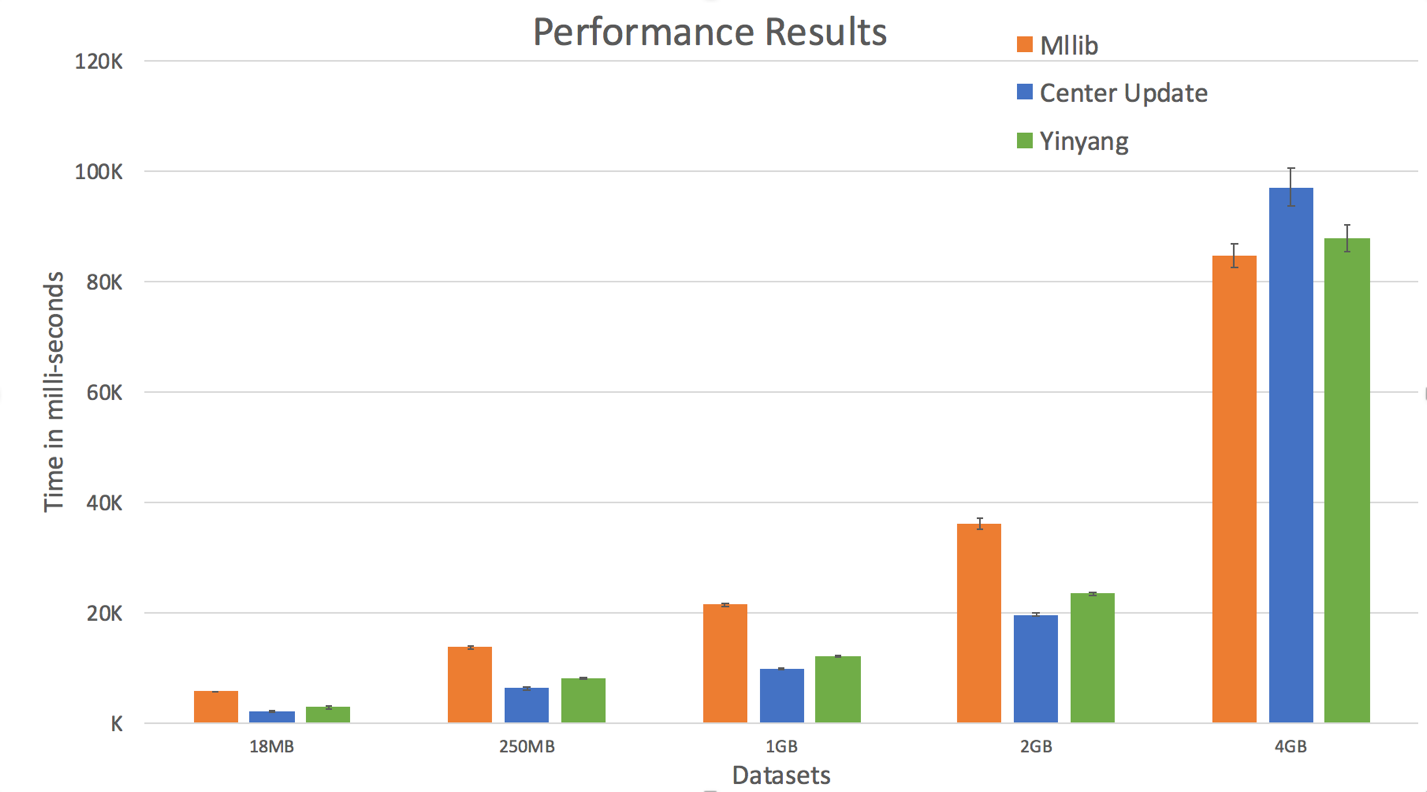


Figure 7: Spark K-Means results for various approaches

**Conclusion**

As shown in the Fig. above the center update optimization and Yinyang K-Means performs better (lower time) than the native MLLib version of K-Means in Spark. It is interesting to note that for the 4GB dataset MLLib is slightly faster than Yinyang and Yinyang is faster than the center update version. We have been able to implement 2 different versions of KMeans in Spark after multiple attempts of the algorithmic design and tested them large datasets. We were successful in deploying our implemented algorithm on a cluster in Amazon EC2. Overall, we have been able to achieve an average speedup of 1.98x for the center update version and 1.6x using the Yinyang K-Means.

**References**

[1] - Zaharia, Matei; Chowdhury, Mosharaf; Franklin, Michael J.; Shenker, Scott; Stoica, Ion. Spark: Cluster Computing with Working Sets (PDF). USENIX Workshop on Hot Topics in Cloud Computing (HotCloud).

[2] - Ding, Y., Zhao, Y., Shen, X., Musuvathi, M., & Mytkowicz, T. (2015). Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In Proceedings of the 32nd International Conference on Machine Learning (ICML-15) (pp. 579-587).

[3] - Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., ... & Stoica, I. (2012, April). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 2-2). USENIX Association.

[4] - <http://spark.apache.org/docs/latest/ec2-scripts.html>

[5] - <http://spark.apache.org/docs/latest/mllib-guide.html>