Distributed MiniBatch-Kmeans for document clustering using Harp

Siddharth Jain Rudrani Angira
jain8@iu.edu rangira@umail.iu.edu
Indiana University Indiana University

ABSTRACT
We discuss the implementation and analysis of MiniBatch K-means algorithm for text categorization for RCV1-RCV2 Reuters data corpus.

Keywords
K-Means, MiniBatch K-means, Harp, Distributed, Multithreaded, Parallelization, Document clustering

1. INTRODUCTION
Clustering, a form of unsupervised learning is very popular in machine learning community. The Lloyd's classic algorithm for clustering is still widely used, but is not suitable for time constrained operations involving large data, because it is computationally expensive. An alternative to it is mini-batch k-means algorithm, which is significantly less computationally expensive and hence delivers quality results quickly. After studying the serial single machine basic mini-batch k-means algorithm, it is obvious that it can be further speed up.

2. PROBLEM DEFINITION
We take the serial single machine mini-batch k-means algorithm and transform it into a faster multi-threaded distributed algorithm using Hadoop with Harp
Framework For the purposes of benchmarking, we use RCV1-RCV2 text categorization collection as dataset. The dataset is a collection of documents (stories). We use this dataset as, news articles and web directories represent some of the most popular and commonly accessed content on the web. Information designers normally define categories that model these knowledge domains (i.e. news topics or web categories) and domain experts assign documents to these categories, whereas with the use of MiniBatch-KMeans we can automate document topic classification and thus reduce the load on domain experts.

3. IMPLEMENTATION
3.1 Dataset
There are **402207** documents in the RCV1-RCV2 data. All the documents are associated with at least one category

3.2 Preprocessing of documents
The RCV1-RCV2 data is in text format therefore we use the vectorized datasets (available in section "B.13.i. RCV1-v2 Vector Files" of RCV1-v2 data set website) for analysis. In the vectorized datasets the documents have already been converted into vector space model and the IDF (inverse document frequency) weights have been pre calculated in the data files by following method:
Suppose the vector for the document is:

$$V_d=[w_{1,d}, w_{2,d}, ..., w_{n,d}]$$

where

$$w_{t,d} = tf_{t,d} * \log\left(\frac{|D|}{|d' \in D || t \in d'|}\right)$$

$tf_{t,d}$ *is term* frequency *of term d in document d*

$\log\left(\frac{|D|}{|d' \in D || t \in d'|}\right)$ = the inverse document frequency.

$|D| = total\ number\ of\ documents$

$|\{d' \in D |\ t \in d'\}|$ = number of documents containing term t

Each vector in a file of vectors is represented by a single line of the form:
<did> [<tid> : <weight>]+
where we have: <did>: Reuters-assigned document id.
<**tid**> : A positive integer term id. Term ids are between 1 and 47,236.
<**weight**> : The numeric feature value, i.e. within document weight, assigned to this term for this document, as described in LYRL2004.

Example of the vector file format:
999995 1:0.03 3:0.047 8:0.38749738478937479 14:0.1 2748:0.03
999996 7:0.13 19:0.138 255:0.58588 314:0.28101 18800:0.005

3.3 Document categories
Topic codes for the document/story are available in TREC qrels format (available under "B.8. On-Line Appendix 8").
Though the document have been hierarchically categorized, we extract their base hierarchical category groups, i.e. C, E, G and M and use them. For example category C31's root base group will be C.
Each line has the format : <category name> <documentID> 1
We use this data for checking the categorization accuracy.

3.4 MiniBatch K-means overview

Algorithm 1 Mini-batch k-Means.

```
Given: k, mini-batch size b, iterations t, data set X
Initialize each c ∈ C with an x picked randomly from X
v ← 0
for i = 1 to t do
    M ← b examples picked randomly from X
    for x ∈ M do
        d[x] ← f(C, x)    // Cache the center nearest to x
    end for
    for x ∈ M do
        c ← d[x]          // Get cached center for this x
        v[c] ← v[c] + 1   // Update per-center counts
        η ← 1/v[c]        // Get per-center learning rate
        c ← (1 − η)c + ηx // Take gradient step
    end for
end for
```

Figure 1: Mini-batch k-Mean algorithm

a.The algorithm takes small batches randomly at each iteration.

b. For each batch a cluster is assigned to each point in the batch.
c. This assignment is based on the previous location of the cluster centroids.
d. New centroids are updated next on the basis of new points in the batch. This is a gradient descent step.

3.4 Intuition for Distributed Multithreaded Algorithm

- With reference to figure 1:
- The whole algorithm works for a batch size b, this whole algorithm can be distributed on n nodes with each having batch size b/n. This will allow us to give low turnaround time, as we will be able to process the partitions of batch in parallel and thus harness the power of CPU of all the nodes.
- Step 5-8 can be executed in parallel on the nodes, as there is no side effect on the shared model on those steps.
- Also step 6 (caching the center) can be safely multi-threaded, as cached center of one point has no dependency on other.
- At first glance steps 10-14 look as if they cannot be executed in parallel on the nodes, as every centroid's update affects its future update. But if we look closely, we could create a mutually exclusive partition of centroid assignment to nodes and can still work in parallel. Once the nodes have worked on their partition of centroid assignment, the partition can be rotated and we can proceed further. This will require rotation of model equivalent to one less than the no. of nodes, but still allows us to compute in parallel. With above in mind we come to a conclusion, the serial algorithm can be distributed across nodes and executed in multi-threaded fashion, while preserving the functionality of the serial algorithm.

3.5 Parallel MiniBatch K-Means Algorithm

1. Each Mapper gets its own chunk of vectorized data
2. The mappers transform their respective chunk of data into list of Document objects. The transformation of lines is parallelized leveraging multithreading.
3. Initial centroids are picked randomly from the list of documents available in the master node and broadcasted to all the other mappers. The picking of centroids is parallelized leveraging multithreading.
4. Initial Sum of Squared Error (SSE) is calculated (further discussed in Experiment section) across the nodes for the whole batch.
5. For given no. of iterations:

a. Each mapper chooses a $\frac{\text{batch size}}{\text{no. of nodes}}$ number of documents randomly (from its pool of documents) for the iteration. The choosing of document on the node is also parallelized leveraging multithreading.
b. For the respective batches' documents in each mapper, the nearest centroid to each document is found using cosine similarity and cached. We parallelize the whole process of cache building and calculating cosine similarity (with parallel additions and multiplications) using multithreading on each mapper.
c. Because of the nature of the algorithm (Algorithm 1, step 9 14, "Sculley, D., 2010 Web-scale k- means clustering"), all the centroids cannot be updated on all the mappers simultaneously. Therefore we assign mappers the responsibility for only updating certain mutually exclusive centroids.
d. Each mapper updates the centroid with its set of data. Centroid updating process is also parallelized to the extent possible using multithreading on each mapper.
e. Then we rotate the cache data in each mapper using Harp's rotate functionality and repeat step d.
f. Repeat step e till all the mappers have not got full cache data view.
g. Update the global centroid data using each mapper's updated centroid.

6. Final SSE is calculated across the nodes for the whole batch.
7. Categorization accuracy (further discussed in experiment section) across the nodes is determined for the whole batch.

3.6 Harp functionalities used

- Broadcast

For initial broadcast of centroids.

- Reduce

To aggregate category accuracy counts at the master node.

- AllReduce

In calculation of SSE across the nodes

- Rotate

For assigning centroid (in step 5.e of section 3.4)

- AllGather

Synchronizing the view of the centroids at the end of each iteration of k-means

- Table with custom data structure

For communicating the document data

.

4. EXPERIMENT

4.1 Cluster Validation

One of the problem with the given data set is that documents are assigned to multiple categories. And since we are using a hard clustering algorithm rather than a soft clustering algorithm, we only get an absolute one-one mapping with a centroid. Thus we do not actually have a baseline data. Thus there arises a need for some technique to determine the goodness of cluster.

4.1.1 Sum of Squared Error (SSE)

We need definite criteria with which we can ascertain the centroids formed at the end of mini-batch are better than the original centroids or not for forming a cluster. SSE is a cluster cohesion measure. SSE is used for determining the goodness of cluster, without any external information.

SSE is mathematically defined as:

$$SSE = \sum_{i=1}^{k} \sum_{p \in C_i} (p - m_i)^2$$

Where,

K = no. of clusters

m_i = centroid of cluster i

We measure the SSE with the original centroids and SSE with the final centroids to judge the goodness of the cluster.

We did "change in SSE" analysis for batch size of 0.5% to 40% of documents and for each of them iterations from 50 to 400. Following are the empirical results:

Centroids	Batch size	Iteration	Change in SSE		Centroids	Batch size	Iteration	Change in SSE
4	80441	400	1051.35229		4	40220	400	615.3529284
4	80441	200	618.5192495		4	40220	200	425.0088961
4	80441	100	1723.167386		4	40220	100	845.0155599
4	80441	50	741.2023612		4	40220	50	2512.077104
4	20110	400	685.3154708		4	10055	400	821.0981474
4	20110	200	1445.460168		4	10055	200	1020.078724
4	20110	100	406.3411203		4	10055	100	1020.818944
4	20110	50	636.8802754		4	10055	50	789.9049987
4	4022	400	528.5827605		4	1005	400	341.604966
4	4022	200	1352.925722		4	1005	200	868.0323249
4	4022	100	3267.305262		4	1005	100	1586.285253
4	4022	50	801.1055292		4	1005	50	2800.192915

From the above it is evident our clustering implementation indeed complies in terms of creating more cohesive clusters.

According to our results, change in SSE does not have a specific pattern with respect to iteration. This type of comparison would have been more valuable had we fixed the initial centroid for each centroid group.

4.1.2 Categorization Accuracy

We measure the accuracy of the categorization in following way:

1. Find the documents most similar to the final centroids found. The reason we find the representative documents is because we only have the category data for the documents in repo and do not have any way to determine the category of a randomly generated point in the space.
2. For every document in every cluster, we find the count of documents classified with at least one common base category as centroid representative document of the centroid it belongs to. It is to be noted this is purely an approximation and might not be the perfect way to measure the accuracy, as every document can actually belong to multiple category.
3. Cluster accuracy across the nodes is aggregated and final accuracy is determined.

Categorization accuracy is calculated at the end, when the final centroids have been determined.

We analyzed this for batch size of 0.5% to 40% of documents and for each of them iterations from 50 to 400. Following are the empirical results:

Centroids	Batch size	Iteration	Accuracy		Centroids	Batch size	Iteration	Accuracy
4	80441	400	0.517457		4	40220	400	0.458845
4	80441	200	0.512854		4	40220	200	0.532071
4	80441	100	0.508026		4	40220	100	0.565246
4	80441	50	0.683547		4	40220	50	0.610914
4	20110	400	0.638381		4	10055	400	0.417034
4	20110	200	0.56461		4	10055	200	0.442254
4	20110	100	0.540863		4	10055	100	0.490272
4	20110	50	0.541458		4	10055	50	0.642836
4	4022	400	0.612832		4	1005	400	0.570523
4	4022	200	0.510941		4	1005	200	0.477526
4	4022	100	0.472186		4	1005	100	0.623663
4	4022	50	0.526388		4	1005	50	0.59949

In general the categorization accuracy should improve with more no. of iterations, but it is not happening so in our experiments. There are two possible reasons, because of which this anomaly is observed:

1. For each iteration group, the initial centroids were different and were randomly chosen. The choice of initial centroid can impact k-means.
2. The basis of this metric is shaky, reason being it is not necessary the established centroids are representing the categories, they might as well be representing some other feature like industry or region, and we don't know what the basis of the centroid formation was. Also we use closest representative document of cluster for comparison and there are multiple categories associated with a document.

4.1.3 Multi-Threaded vs Single-Threaded

Out of curiosity, we also wanted to compare our multi-threaded implementation with single-threaded version.

The single threaded version follows the same algorithm as for the multi-threaded, except that wherever there was multi-threading it has been replaced with single threaded code.

All except one result indicate single-threaded code is faster than multi-threaded test case. Following are the empirical results:

Batch Size	Iteration	Single Threaded Mini-Batch Run Time Lead (ms)	Single Threaded Total Running Time Lead (ms)
80441	200	1321134	1415315
80441	100	132415	195280
40220	200	384847	504225
40220	100	76399	134216
20110	200	-253661	118804
20110	100	82668	177631
10055	400	450446	564063
10055	200	92614	166075
10055	100	91579	220689
4022	400	82748	156839
4022	200	39851	116776
4022	100	26728	120327
1005	200	17152	95959
1005	100	4340	54873

The reason we think the multi-threaded code is slower because we have used parallel streams over parallel streams, which might actually be spawning nested threads (whereas what we assumed was, the threads will be used from a pool) or maybe there is competition for contention of ForkJoin threads, which in turn must be overwhelming the nodes. We observed during the multi-threaded code execution, CPU utilization was more or less above 90% all the time for all the cores. Further analysis is required to understand what exactly caused this.

5. CONCLUSION

Mini-batch k-means can indeed be implemented in distributed fashion and can give good clusters (with respect to SSE).

Harp framework is helpful and does simplify things once you get it right. There is a need for more detailed documentation to get a user started with it.

It was surprising to see single threaded code getting executed faster than multi-threaded code, but it was a learning experience. Multi-threading needs to be done with care, so as to not overwhelm the system.

6. ACKNOWLEDGMENTS

Our thanks to Prof Judy Qiu and the Associate instructors who helped us throughout the semester to develop better understanding of Harp Framework.

7. REFERENCES

1. Yadav, M. K., & Baria, M. J. Mini-Batch K-Means Clustering Using Map-Reduce in Hadoop. International Journal of Computer Science and Information Technology Research, 2(2), 336-342.

2. Wan, J., Yu, W., & Xu, X. (2009, December). Design and implement of distributed document clustering based on MapReduce. In Proceedings of the Second Symposium International Computer Science and Computational Technology (ISCSCT), Huangshan, PR China (pp. 278-280).

3. Lewis, D. D. RCV1-v2/LYRL2004: The LYRL2004 Distribution of the RCV1-v2 Text Categorization Test Collection (5-Mar-2015 Version). http://www.jmlr.org/papers/volume5/lewis04a/lyrl2004_rcv1v2_README.htm

4. Sculley, D., 2010 Web-scale k-means clustering, in: Proceedings of the 19th