Distributed Machine Learning Mini Batch K-Means

Rohit Nair
Indiana University Bloomington
571 West Amaryllis Drive,
Bloomington, Indiana
+1 (812) 361 - 8754
Ronair@iu.edu

Abhijit Karanjkar
Indiana University Bloomington
571 West Amaryllis Drive,
Bloomington, Indiana
+1 (812) 384-7145
Aykaranj@iu.edu

ABSTRACT

An unsupervised clustering algorithm, with tunable parameters - K-Means Mini Batch. We tweak the standard algorithm and perform it in batches in a parallel environment using Indiana University's HARP API. Our approach is specifically for document clustering, which can be used in search engines and in document organization.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

K-means, clustering, Hadoop, Harp, Distributed computing, Parallel processing, Document clustering, Mini-Batch, Stochastic

1. INTRODUCTION

K-Means is an NP-Hard problem, so we can't find the optimal solution to every problem. Our aim in general is to find the sweet spot between accuracy and efficiency using the resources that we have at our disposal.

Full batch approach gives awesome results in general, but suffers latency and for large data sets since it has a time complexity of O(#iterations * #clusters * #examples * #dimensions). On the other extreme, we have the stochastic approach where we keep adding one example to our computation per iteration, and keep stepping towards the solution in increments. The initial iterations are lightning fast because there are very few examples to process, but speed reduces as we progress. The initial error rate is very high but experiments prove that we converge well using the stochastic approach. Then there's the middle ground - Mini-Batch. We choose typically 10/100/1000 example increments in each iteration. It has the speed advantage of stochastic and has much better convergence, close to full batch.

"For small values of k, the mini-batch methods were able to produce near-best cluster centers for nearly a million documents on a single 2.4 GHz machine. This makes real-time clustering practical for user-facing applications." [2]

2. DATASET: RCV1-V2[2]

We have 402207 documents in the dataset. We categorize each of them into several categories specified by an input parameter from the user. Each line in the data files represents words in one document, with the following line structure:

[.I <did>
.W
<textline>+
<blankline>]+

K-means is for numbers and not for words, so we convert words in the document into weights using the below calculations:

1. Word frequency in all documents:
   We calculate which word appears in how many documents

2. Generate Data Points:
   We calculate data points as N files (for N nodes); each consisting of (D/N) rows and W columns; where
   - N is the number of nodes
   - D is the total number of documents
   - W is the number of words across all documents

Detailed Steps:

1. **Calculate word frequency in all documents:**
   a. We calculate the number of files in which each word occurs

   b. The parent node specifies which document ids each node will be working on

   c. Mapper step: We do this in parallel: each node calculates a local

Hashmap<word, count>. Example below:

Word	Number of documents in which it appears (in the 'x' number of documents that the worker node processes)
Rohit	2
Normalize	1

d. Reduce step: The parent node aggregates this and writes to a file (let's call it Global map of word frequencies). Example below:

Word	Number of documents in which it appears
Rohit	3
Abhijit	2
The	9
Normalize	1

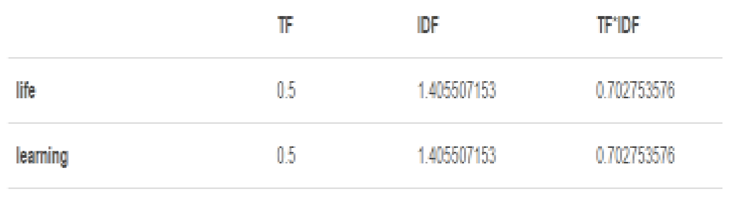
e. We also write a word to id mapping wordToIndexMap<word, index> which will be used by nodes to identify the index of each word that they encounter. Example below:

Word	Index
Rohit	0
Abhijit	1
The	2
Normalize	3

2. Generate Data Points:

This step is responsible for generating final data points that will be processed using Mini-Batch K-means.

We have followed the Mini-batch K-means algorithm as explained here [2]. The main objective is to convert the input data in the form of paragraphs and words to the numerical data on which we can easily perform the calculations like finding Euclidean distance etc. To map words to numerical values we use the concept of Term frequency and inverse document frequency as explained here [4]. For example, the document with words life and learning will produce following result.



	TF	IDF	TF*IDF
life	0.5	1.405507153	0.702753576
learning	0.5	1.405507153	0.702753576

Where TF: Term frequency

IDF: Inverse document frequency

The multiplication TF * IDF will give the final numeric value that represents the corresponding word.

Assumptions:

n: Number of nodes

D: Number of documents = Number of data Points

f: Number of files which will be provided to the nodes to process the data in parallel

Algorithm skeleton:

- Read f files per node in parallel. (This is the mapper stage)
- Load global hash table containing words and corresponding frequencies obtained during initial processing step.
- Load global hash table containing words and corresponding indexes obtained during initial processing step.
- Broadcast both the global hashmaps
- For each word in each document calculate TF and IDF and obtain TF*IDF
- Write the table in files where each tuple contains current document id, TF*IDF for all the words in all documents. The words that are not there in in the current document will have TF*IDF = 0.
- N nodes will write this tabular data into f files.

Consider an example

Doc id: 1

W:

Distributed systems

Doc id: 2

W:

Computer Networks

The final table looks like:

	0(Distributed)	1(Systems)	2(Computer)	3(Networks)
Doc 1	TF*IDF	TF&IDF	0	0
Doc 2	0	0	TF*IDF	TF*IDF

The dataset is also given in a vectored format with TF*IDF calculated per word per document and stored in the below format:

<did> [<tid>:<weight>]+

3. METHOD:

Input from user:

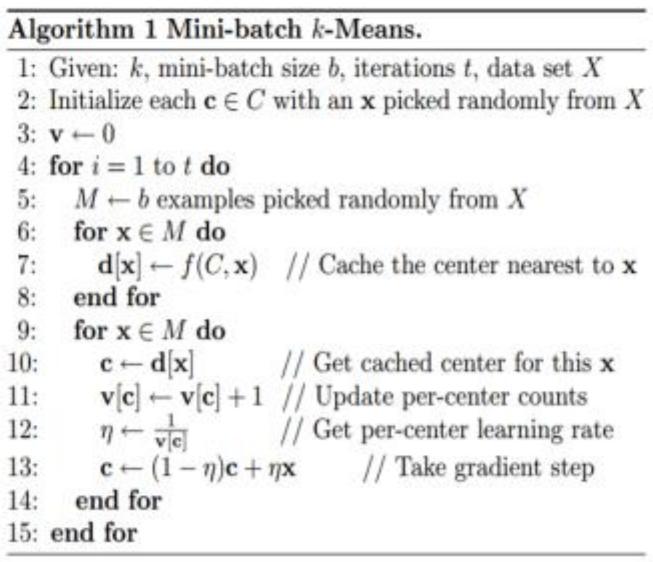
1. N = number of nodes for parallelization
2. O = output files' directory
3. F = input files' directory
4. B = batch size in percent
5. K = number of centroids
6. T = maximum number of iterations

We randomly assign data points as centroids initially and then update them in up to T iterations using B number of data points per iteration per node and calculate which documents are similar

In this step, we perform Mini-Batch K-means algorithm on the vectored documents. N nodes will read f files in pre-defined number of batches. The algorithm that we are going to follow is mentioned here [2].

Algorithm:

- Randomly select the centroids from the data points generated in previous step
- Write these centroids to a file
- Then perform the algorithm below as mentioned in [2] for t iterations or till convergence.



```
Algorithm 1 Mini-batch $k$-Means.
1: Given: $k$, mini-batch size $b$, iterations $t$, data set $X$
2: Initialize each $\mathbf{c} \in C$ with an $\mathbf{x}$ picked randomly from $X$
3: $\mathbf{v} \leftarrow 0$
4: for $i = 1$ to $t$ do
5:    $M \leftarrow b$ examples picked randomly from $X$
6:    for $\mathbf{x} \in M$ do
7:       $\mathbf{d}[\mathbf{x}] \leftarrow f(C, \mathbf{x})$   // Cache the center nearest to $\mathbf{x}$
8:    end for
9:    for $\mathbf{x} \in M$ do
10:      $\mathbf{c} \leftarrow \mathbf{d}[\mathbf{x}]$   // Get cached center for this $\mathbf{x}$
11:      $\mathbf{v}[\mathbf{c}] \leftarrow \mathbf{v}[\mathbf{c}] + 1$   // Update per-center counts
12:      $\eta \leftarrow \frac{1}{\mathbf{v}[\mathbf{c}]}$   // Get per-center learning rate
13:      $\mathbf{c} \leftarrow (1 - \eta)\mathbf{c} + \eta\mathbf{x}$   // Take gradient step
14:   end for
15: end for
```

Using the power of distributed computing:

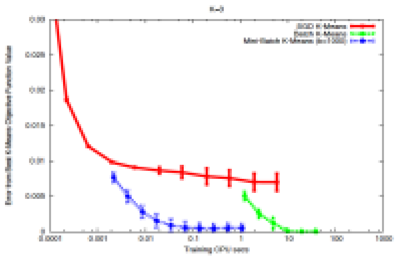
- Given a batch size b (which we tune during input), we distribute b/n documents per iteration to the n nodes. This will allow us to give low turnaround time, as we will be able to process the partitions of batch in parallel and thus harness the power of CPU of all the nodes (but gives a good performance in a single processor mocking multiple nodes as well).

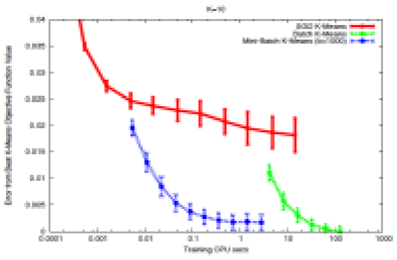
1. We randomly select the specified number of centroids from the given documents
2. These centroids are broadcasted to all the nodes by writing them to a centroid file in the distributed file system.
3. We distribute the vectored data batch by batch per iteration among the n nodes
4. Each node maps the data it is given into document objects.
5. We calculate the sum of squared errors from all the nodes for the batch.
   - To measure how close a certain point is to the different centroids, we sum over all k, for each centroid and calculate $(p - m_i)^2$
   - Where k is the number of clusters and i represents each centroid
6. These tell us which centroid is closest to each data point using cosine similarity formulation for whatever centroids are in the local data's domain, leaving the aggregation to the main node.
7. Based on this calculation, we recalculate centroids, return them to the main node, which aggregates the results and updates the centroid file, which is then read by each node in the next iteration.
8. In the end, we correlate the final centroids to the closest document from our list of documents.

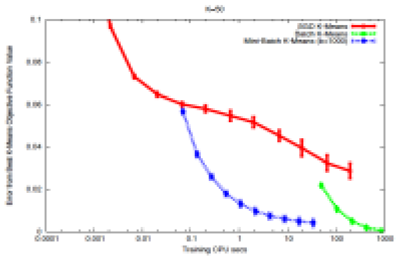
One problem we initially faced in these documents is that the there was no consistency or order in which terms are used. To handle this, we use Hashmaps to help us in the error calculations, because we've to compare like terms across documents.

Harp has various functions that we use in our implementation of K-means mini-batch algorithm:

1. Broadcast
   - Centroid broadcast
2. AllReduce
   - Aggregate the sum of squared errors at the parent node
3. AllGather
   - Used for the collecting all the centroid data from all the nodes







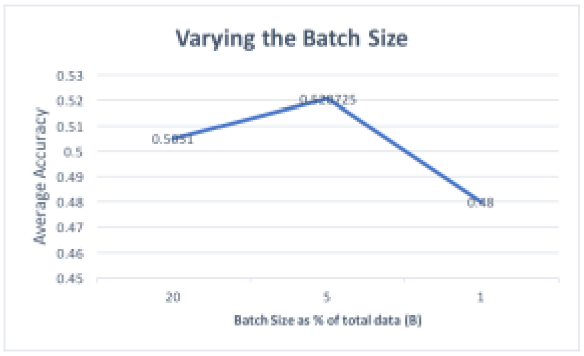
4. EXPERIMENTS:

We experimented on a small dataset with a sequential approach and got correct results for the following documents and the search query "life learning" [4]:
File 1:

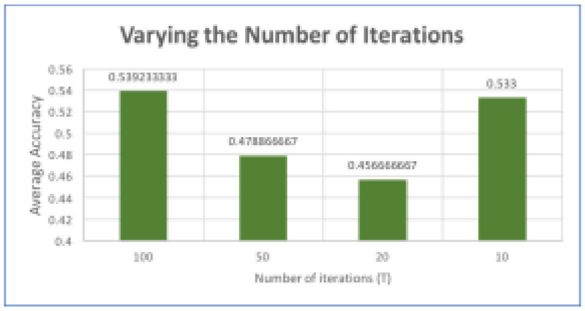
<did:1>: The game of life is a game of everlasting learning
<did:2>: The unexamined life is not worth living
<did:3>: Never stop learning

Then we distributed our processing using Hadoop-Harp for the RCV1-V2 dataset and tried for different batch sized and iterations. We compare our results to the categories' collection in the dataset which is in the following format:
<category name> <did> 1

We tried changing the batch size with different number of iterations and averaging out the results, following is the plot:



We tried changing the number of iterations with different batch sizes and averaging out the results, following is the plot:



Experiments show that we get a big advantage over stochastic as 'k' increases and a marked speed boost with respect to full batch K-Means without a significant loss in convergence. The performance improvements are predicted to be more evident as one or more of k, #examples and #dimensions increases.

5. CONCLUSION:

Mini-batch version of k-means indeed has a very good performance as compared to full batch and gives satisfactory clustering

6. REFERENCES:

[1] *Lewis, D. D.; Yang, Y.; Rose, T.; and Li, F. RCV1: A New Benchmark Collection for Text Categorization Research. Journal of Machine Learning Research, 5:361-397, 2004.http://www.jmlr.org/papers/volume5/lewis04a/lewis04a.pdf.*

[2] [Sculley 2010] Sculley, D., 2010 Web-scale k-means clustering, in: Proceedings of the 19th International Conference on World Wide Web. ACM, pp. 1177–1178.

[3] HARP: a practical projected clustering algorithm http://ieeexplore.ieee.org/document/1339265/

[4] https://janav.wordpress.com/2013/10/27/tf-idf-and-cosine-similarity/