Large-scale Clustering for Big Data Analytics: A MapReduce Implementation of Hierarchical Affinity Propagation

Dillon Mark Rose[a], Jean Michel Rouly[b], Rana Haber[c], Nenad Mijatovic[d], Adrian M. Peter[e]

[a]Computer Science Department, Florida Institute of Technology, Melbourne, Florida; drose2010@my.fit.edu
[b]Computer Science Department, George Mason University, Fairfax, Virginia; jrouly@gmu.edu
[c]Mathematical Sciences Department, Florida Institute of Technology, Melbourne, Florida; rhaber2010@my.fit.edu
[d]Electrical Engineering Department, Florida Institute of Technology, Melbourne, Florida; nmijatov2005@my.fit.edu
[e]Engineering Systems Department, Florida Institute of Technology, Melbourne, Florida; apeter@fit.edu

ABSTRACT

The accelerated evolution and explosion of the Internet and social media is generating voluminous quantities of data (on zettabyte scales). Paramount amongst the desires to manipulate and extract actionable intelligence from vast big data volumes is the need for scalable, performance-conscious analytics algorithms. To directly address this need, we propose a novel MapReduce implementation of the exemplar-based clustering algorithm known as Affinity Propagation. Our parallelization strategy extends to the multilevel Hierarchical Affinity Propagation algorithm and enables tiered aggregation of unstructured data with minimal free parameters, in principle requiring only a similarity measure between data points. We detail the linear run-time complexity of our approach, overcoming the limiting quadratic complexity of the original algorithm. Experimental validation of our clustering methodology on a variety of synthetic and real data sets (Reuters articles, medical data, etc.) demonstrates our competitiveness against other state-of-the-art MapReduce clustering techniques.

Keywords: MapReduce, Cluster, Affinity Propagation, Hierarchical Affinity Propagation, Hadoop

1. INTRODUCTION

In 2010, big data was growing at 2.5 quintillion[14] bytes per day, with the present and near future world-wide data generation flirting with even larger sextillion scales. This overwhelming volume, velocity, and variety of data can be attributed to the ubiquitously spread sensors, perpetual streams of user-generated content on the web, and increased usage of social media platforms—Twitter alone produces 12 terabytes of tweets every day. The sustained financial health of the world's leading corporations are intimately tied to their ability to sift, correlate, and ascertain actionable intelligence from big data in a timely manner. These immense computational requirements have created a heavy demand for advanced analytics methodologies that leverage the latest in distributed, fault-tolerant parallel computing architectures. Among a variety of choices, MapReduce has emerged as one of the leading parallelization strategies, with its adoption rapidly increasing due to the availability of robust open source distributions such as Apache Hadoop.[8] In the present work, we develop a novel MapReduce implementation of a fairly recent[11] clustering approach referred to as Hierarchical Affinity Propagation (HAP).

Clustering techniques are at the heart of many analytics solutions. They provide an unsupervised solution to aggregate similar data patterns, which is key to discovering meaningful insights and latent trends. This becomes even more necessary, but exponentially more difficult, for the big data scenario. Many clustering solutions rely on user input specifying the number of cluster centers (e.g. K-Means clustering[13] or Gaussian Mixture Models[19]), and biasedly group the data into these desired number of categories. Frey et al.[9] introduced an exemplar-based clustering approach called Affinity Propagation (AP). As an exemplar-based clustering approach, the technique does not seek to find a mean for each cluster center, instead certain representative data points are selected as the exemplars of the clustered subgroups. The technique is built on a message passing framework where data points "talk" to each other to determine the most likely exemplar and automatically determine the clusters, *i.e.* there is no need to specify the number of clusters a priori. The sole input is the pairwise similarities between all data points under consideration for clustering—making it ideally suited for a variety of data types (categorical,

AMALTHEA REU Technical Report No. 2013–4; available at www.amalthea-reu.org. Please send your correspondence to Georgios C. Anagnostopoulos.

numerical, textual, etc.) since a method to define similarity between data elements is all that is needed. A recent extension of the AP clustering algorithm is Hierarchical Affinity Propagation (HAP),[11] which groups and stacks data in a tiered manner. Similar to AP, the HAP algorithm does not require the number of clusters as input; it only requires the number of hierarchy levels. It adopts a similar message passing scheme to AP, but now the communication between data points occurs both within a single layer and up and down the hierarchy. To date, AP and HAP have been mainly relegated to smaller, manageable quantities of data. This stems from the prohibitive quadratic scaling of the run time complexity as the data points increase. Our investigations will demonstrate an effective parallelization strategy for HAP using the MapReduce framework, for the first time enabling applications of these powerful techniques on Big Data problems.

First introduced by Google,[4] the MapReduce framework is a programming paradigm designed to facilitate the distribution of computations on a cluster of computers. The ability to distribute processes in a manner that takes the data to the computing is key when mitigating the computational cost of working with extremely large data sets. The parallel programming model depends on a mapper phase that uses key-value identifiers for the distribution of data and subsequent independent executions to generate intermediate results. These are then gathered by a reducing phase to produce the final output key-value pairing. This simple, yet widely applicable parallelization philosophy has empowered many to take machine learning algorithms previously demonstrated only on "toy data" and scale them to enterprise-level processing.[2] In this same vein, we adopted the most popular open source implementation of the MapReduce programming model, Apache Hadoop, to develop the *first ever* parallelized extension of HAP. This allows efficient fault-tolerant clustering of big data, and more importantly, improves the run time complexity to potentially linear time (given enough mappers).

2. RELEVANT WORK

To handle the large explosion of available data, there is now a vast amount of research in computational frameworks to efficiently manage and analyze these massive information quantities. Here we focus on the MapReduce framework[6,14,22,25] for faster and more efficient data mining, covering the most relevant to our approach.

Dai et al.[3] proposed a novel approach to cluster large data sets. They implement the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm, which was originally introduced by Ester et al.[5] in a MapReduce framework. The new approach, DBSCAN-MR, requires a partitioning of the dataset where each partition is sent to different Virtual Machines (VMs) to be clustered. To deal with the data points on the partition boundaries, they replicate those data points and introduce them to both VMs handling these partitions. They introduce a new algorithm to have efficient partitions, Partition with Reduce Boundary Points, and discuss its benefits in improving both execution time and VM load. Though the MapReduce implementation of DBSCAN effectively parallelizes the clustering, the need to perform an initial partitioning of the data can unnecessarily bias the data aggregation. Our MapReduce implementation of HAP allows communication between all data points, promoting discovery of optimal clusters. Additionally, HAP does not employ local data density computations, while DBSCAN does.

Another state of the art MapReduce clustering algorithm is hierarchical K-means, which was implemented by Nair et al.[20] and is referred to as Hierarchical Virtual K-means (HVKM). HVKM uses cloud computing to handle large data sets, while supporting top to bottom hierarchies or a bottom to top approach. Since it derives its roots from K-means, HVKM requires one to specify the number of clusters. Our HAP implementation does not require presetting the number of required clusters; it instead organically and objectively discovers the data partitions.

In Wu et al.,[23] the authors propose to parallelize AP on the MapReduce framework to cluster large scale E-learning resources. The parallelization happens on the individual message level of the AP algorithm. We perform a similar parallelization but significantly go beyond and allow for hierarchical clustering, which enables a deeper understanding of the data's semantic relationships. The authors provide little implementation discussion and limited experimental results on E-learning resources. We detail MapReduce implementation strategies for HAP, thus implicitly also for AP. In addition, our development is designed to work on a variety of data sources; thus, our experiments will showcase results on multiple data modalities, including text, images, and numerical data, as shown in Section 5.

The rest of this paper is organized as follows. In the next section, Section 3, we discuss important background information on the software frameworks which compose the project foundation. In Section 4.1 and Section 4.2 we detail the non-parallel AP and HAP algorithms. Section 4.3 discusses the MapReduce paradigm and Section 4.4 discusses the implementation details for these algorithms. The experimental validations provided in Section 5 demonstrate our favorable performance against several other clustering algorithms which are readily available in the open source project Apache Mahout.[7] Finally, we conclude with a summary of our efforts and future recommendations.

3. SOFTWARE FOUNDATION

Much of this software is heavily dependent upon, and would not be possible without, existing Free and Open Source Software (FOSS) frameworks. Largely using projects developed by the Apache Software Foundation, this software has been built on the shoulders of giants. The Apache Hadoop Project provides the MapReduce implementation and general infrastructure foundation necessary to execute efficiently and securely on a cluster or cloud. Apache Mahout serves as an extension of Hadoop in providing efficient, optimized mathematical functionality targeted toward Machine Learning (ML). MySQL and Apache Hive provide robust and easy to use data storage. Amazon Web Services (AWS) introduces the necessary hardware infrastructure for the entire software project to run smoothly in a distributed environment. Each project or service contributes in an important manner to this project, and it is vital to understand the function and maintenance of each before moving forward.

3.1 Apache Hadoop

The Apache Hadoop* project is a large FOSS framework written in Java with the goal of offering distributed processing power scalable to thousands of computers. Hadoop implements the MapReduce programming paradigm† and comes with fully featured libraries of code to provide a solid basis for development. Hadoop also employs its own implementation of a distributed file system, referred to as the Hadoop Distributed Filesystem (HDFS).

The HDFS is a file management system that balances the load of data storage across clusters of computers (nodes) while providing fault tolerance and optimizations for speed. The HDFS is designed to operate on clusters of commodity computers with the goal of reducing the cost necessary to create a large cluster. A consequence of using commodity computers, however, is increased risk of machine failure, emphasizing the importance of fault tolerance.

Under the HDFS, each node within the cluster is specialized to perform different tasks. At the top of the hierarchy is the single master NameNode which manages the entire cluster. Beneath this node are the slave DataNodes which serve as physical storage machines. The data in the cluster is stored multiple times on different DataNodes, providing redundant, fault tolerant data storage. When files are loaded into the HDFS they are broken into blocks and the NameNode balances the data among its DataNodes. Each DataNode stores and serves back the blocks of data. Although the files may not be stored as whole files, the NameNode maintains file metadata and references to the blocks which comprise contiguous files. With this data, the NameNode is then able to simulate a single disk with contiguous files. When MapReduce tasks are run, the NameNode routes data from DataNodes to the TaskTrackers, which manage task execution. Obviously, the NameNode serves as a single point of failure, requiring the presence of a SecondaryNameNode to monitor the NameNode's functionality at all times. All HDFS operations are taken care of by the Hadoop framework and occur at a low level. All programmer interaction occurs at a high level, interfacing directly with Hadoop's MapReduce programming model.

Much of Hadoop's usability and power comes from additional Apache projects built upon it. Apache Hive and Apache Mahout are two such libraries.

*http://hadoop.apache.org; detailed installation materials available in Appendix A

†The specifics of the MapReduce programming paradigm are discussed in Section 4.3.

3.2 Apache Mahout

Apache Mahout[*] is a FOSS library of mathematical functionality and ML algorithms implemented in MapReduce using the Apache Hadoop framework. At its core, Mahout provides utilities for creating and manipulating mathematical constructs through MapReduce jobs. Combing these utilities, Mahout implements ML algorithms optimized for performance in a cloud computing environment. A full list of functionality and algorithms are available on the Mahout website. When working with Mahout, users can either access the functionality programmatically with Java, or use the built-in Command Line Interface (CLI).

Referencing the Mahout libraries from within a software application provides the programmer with a low-level control of the provided functionality. Programmers can either build their custom applications on top of the Mahout libraries or extend the Mahout classes and add additional functionality. This project manipulates the Mahout libraries in both fashions. See `root.input.reuters21578.VectorizationJob.java` and `root.input.InputJob.java`, respectively[†]. Specifically, the tools used include optimized vector objects and optimized binary file formats, a text vectorization routine, and a CLI argument parser.

Interfacing with Mahout through its CLI sacrifices the ability to create custom applications for the power to quickly use the built-in algorithms. An example of a shell script for running a Hierarchical K-Means job is available in Appendix C.1.2. It is good practice to use the CLI when possible because the commands are properly optimized for performance in a cloud computing environment.

3.3 Apache Hive

Apache Hive[‡] is another FOSS project developed by the Apache Software Foundation. Hive provides a flexible, scalable data management system similar to a typical relational database. The main operational difference between Hive and a typical relational database is that Hive is built on top of the Hadoop framework and integrates seamlessly with the HDFS. Furthermore, Hive executes user queries by generating on-the-fly MapReduce tasks within Hadoop, thus quickly and efficiently accessing large amounts of data potentially distributed across a cloud. These queries are structured under the Hive Query Language, a structured database query syntax very similar to standard SQL.[1]

The purpose of the Hive implementation in this project is post-processing data archival and retrieval, specifically for feeding results into visualization software. Results are generated by the HAP algorithm and then immediately stored on the HDFS. Hive is used to retrieve this information and manipulate it into a usable format. The end goal of this process being user-interactive visualizations, efficiency and low latency access times are key. Hive is efficient when operating over a cluster of machines and managing results retrieved directly from HAP, however ideally all data should be centrally stored on a single machine in a simple relational database to maximize access speed. To this end, MySQL is used to construct the consolidated database which feeds the visualization software.

3.4 MySQL

MySQL[§] is a ubiquitous FOSS relational database tool. It provides the ability to manage and query large amounts of data with extreme responsivity. Queries are executed in standard SQL syntax. Unlike Hive, MySQL is not a distributed system, meaning it is generally run on a single machine. This has the direct effect of decreasing query execution time relative to Hive. This also means that MySQL is unable to manage the same scale of data as Hive. Because of its usage in this project, these are both acceptable consequences. Decreased access times are ideal while decreased storage capacity is acceptable since the database is only storing a barebones consolidated version of the HAP output as opposed to a full copy of data from Hive.

In this project, MySQL provides input support for the visualization software. Once all output information is extracted from the input dataset, including cluster assignments from HAP, the data is passed through Hive, consolidated, and stored in a MySQL database. This database is available online for query by a business

[*] http://mahout.apache.org
[†] Source code listing is available in Appendix D.2.
[‡] http://hive.apache.org
[§] http://www.mysql.com

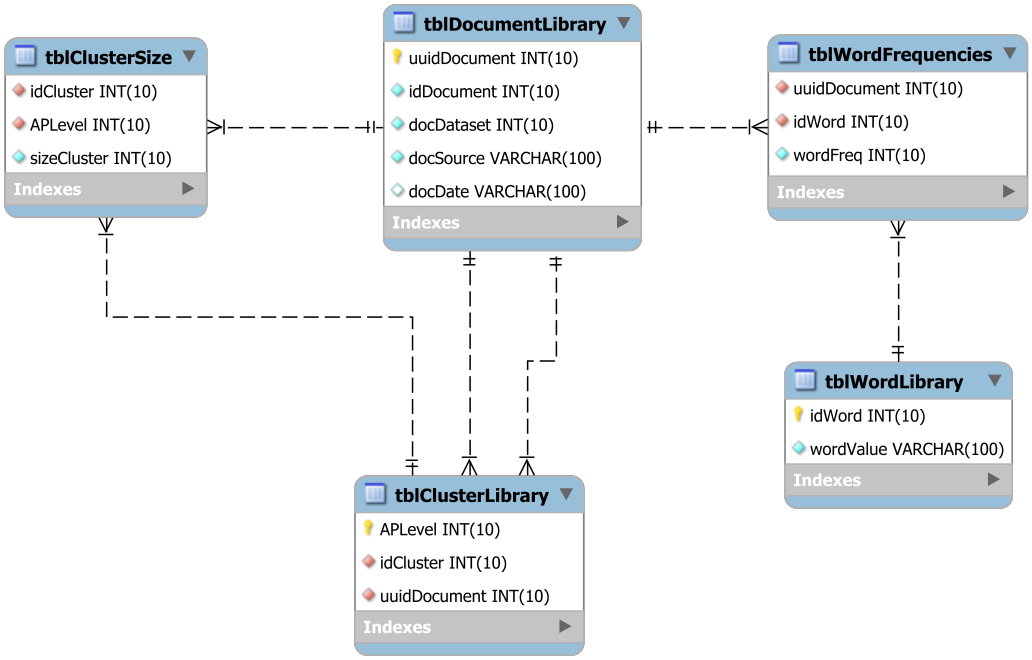


Figure 1: Visual Diagram of MySQL Database

intelligence layer which then feeds directly into the visualization software. Cluster structure is represented implicitly in the database schema developed for this project, along with general data set metadata made available to the visualizations. Figure 1 shows a visual diagram of the schema.

3.5 Amazon Web Services

Amazon.com is one of the largest web-based companies in the world, ranked by Alexa.com as the 6th most accessed website globally.[15] In addition to its popular online shopping service, Amazon offers access to its massively powerful data centers through the AWS portal*. AWS provides a variety of services, including processing time running customizable VMs on the Amazon Elastic Compute Cloud (EC2), persistent data storage on Amazon Simple Storage Service (S3), webhosting, and even application deployment solutions. The most relevamt service, however, is Amazon Elastic MapReduce (EMR).

3.5.1 Amazon Elastic MapReduce

EMR allows users to dynamically allocate EC2 instances into resizable, preconfigured Hadoop clusters. Backed by the full power of EC2, EMR handles all the work of creating a Hadoop cluster on the cloud behind the scenes. This allows users to define custom Hadoop job JARs, execute Pig dataflow scripts, or even run Hive scripts without worrying about any network setup or infrastructure maintenance. EMR instances come preinstalled with a recent version of Hadoop running on a modern Ubuntu VM. Additional third party software, such as Mahout, can be easily installed on an EMR cluster using initialization bootstrap scripts. Sample AWS scripts, such as a bootstrap script for installing Mahout, can be found in Appendix C.1. The following section gives a brief overview of the initial requirements, setup steps, and possible troubleshooting issues that an EMR user may run into.

3.5.2 Using Amazon Elastic MapReduce

Amazon's EMR is not always straightforward to use. The simplest method of use is through the EMR graphical interface, available in the AWS Console. A video training series for setting up your AWS account and beginning to use EMR is available on the AWS training website†.

*http://aws.amazon.com

†http://aws.amazon.com/elasticmapreduce/training/#videos

Once your AWS account is configured properly and allows secure access to EMR clusters using the Amazon Access Key system, the next setup step is to configure and install the EMR CLI. These tools allow, among other things, for EMR clusters to persist after a failed execution step. This is very valuable as Amazon's pricing scheme rounds even a single minute of EC2 use up to the next hour. Utilizing the same cluster for multiple steps reduces wasted time and cost. A thorough introduction to the CLI is available in the Command Line Interface Reference for Amazon EMR in Appendix D.1 and online at the Amazon Developer Guide.[16]

After an EMR user environment has been successfully configured on the developer's machine, it is possible to begin executing custom Hadoop jobs on the Amazon cloud. This is a straightforward, well documented process, but some selected notes are available in Appendix B.1. A discussion of the HAP implementation which is to run on AWS follows.

4. CLUSTERING

4.1 Affinity Propagation

The main ideas behind AP are motivated by the simple fact that given pairwise similarities between all input data, one would like to partition the set so as to maximize the similarity between every data point in a cluster and the cluster's exemplar. Recall that an exemplar is an actual data point that has been selected as the cluster center. Additionally, the situation wherein an exemplar does not select itself is unfavorable and is thus penalized. As we will briefly discuss, these ideas can be effectively represented as an algorithm in a message passing framework. In the landscape of clustering methodologies, which includes such staples as K-means,[13] K-medoids,[17] and Gaussian Mixture Models,[19] predominantly all methods require the user to input the desired number of cluster centers. AP avoids this artificial segmentation by allowing the data points to communicate accumulated evidence encoded in their similarity values, which organically gives rise to a partitioning of the data. In many applications, like document clustering, an exemplar-based clustering technique gives each cluster a more representative and meaningful prototype for the center, versus a fabricated mean.

AP starts off assuming that all data points are possible exemplars. Frey et al.[9] proposes viewing each data point as a node in a network connected to other nodes by arcs such that the weight of the arcs $s(i, j)$ describes how similar the data point with index i is to the data point with index j. AP takes as input this similarity matrix where the entries are the negative real valued weights of the arcs. Having the similarity matrix as the main input versus the data patterns themselves provides an additional layer of abstraction—one that allows seamless application of the same clustering algorithm regardless of the data modality (e.g. text, images, general features, etc.). The similarity can be designed to be a true metric on the feature space of interest or a more general non-metric.[9] Frey et al.[9] uses the negative of the squared Euclidean distance for the similarities in their experiments, *i.e.* the similarity between data points x_i and x_j is given as $s(x_i, x_j) = -\|x_i - x_j\|^2$. One can easily adopt and validate performance across the multitude of similarity measurements typically used in information retrieval.[12] The diagonal values of the similarity matrix, $s(j, j)$, are referred to as the "preferences" which specify how much a data point j wants to be an exemplar. Since the similarity matrix entries are all negative values, $-\infty < s(i, j) \leq 0$, $s(j, j) = 0$ implies data point j has high preference of being an exemplar and $s(j, j) \approx -\infty$ implies it has very low preference. In some cases, as in Frey et al.,[9] Givoni et al.,[11] and Guan et al.,[12] the preference values are set using some prior knowledge, for example uniformly setting them to the average of the maximum and minimum values of $s(i, j)$, or by setting them to random negative constants. Through empirical verification, we experienced better performance with randomizing the performances and adopt this approach for most of our experiments in Section 5.

Once the similarity matrix is constructed and provided as the primary input to the AP algorithm, the network of nodes (data points) recursively transmits two kinds of messages between each other until a good set of exemplars is chosen. The first message is known as the "responsibility" message and the second as the "availability" message. The responsibility messages, $\rho(i, j)$, are sent from data point i to data point j portraying how suitable node j is to be an exemplar for node i. Similarly, availability messages, $\alpha(i, j)$, are sent from data point j to i, indicating how available j is to be an exemplar for data point i. The responsibility and availability

update equations are given in Eq. (1) and Eq. (2), respectively.

$$\rho(i,j) \leftarrow s(i,j) - \max_{k s.t. k \neq j} \{\alpha(i,k) + s(i,k)\} \tag{1}$$

$$\alpha(i,j) \leftarrow \min\left\{0, \rho(j,j) + \sum_{k s.t. k \notin \{i,j\}} \max\{0, \rho(k,j)\}\right\} \tag{2}$$

Eq. (3) is the self-availability equation which reflects the accumulated positive evidence that j can be an exemplar. The self-responsibility messages are updated the same way as the responsibility messages.

$$\alpha(j,j) \leftarrow \sum_{k s.t. k \neq j} \max\{0, \rho(k,j)\} \tag{3}$$

After all messages have been sent and received, the cluster assignments are chosen based on the maximum sum of the availability and responsibility messages. These cluster assignments can be used to extract the list of exemplars.

$$e(i) \leftarrow \arg\max_j \{\alpha(i,j) + \rho(i,j)\} \tag{4}$$

These net message exchanges are seeking to maximize the cost of correctly labeling a point as an exemplar and gathering its representative members (a cluster). This desired effect can be efficiently captured in the following objective function over the valid configurations of the labels $\mathbf{e} = (e_1, \cdots, e_N)$,

$$S(\mathbf{e}) = -\sum_{i=1}^{N} s(i, e_i) + \sum_{j=1}^{N} \delta_j(\mathbf{e}), \tag{5}$$

where $\delta_j(\mathbf{e}) = -\infty$ if $e_j \neq j$ but $\exists i : e_i = j$ and $\delta_j(\mathbf{e}) = 0$ otherwise—a penalty added to enforce the constraint that a node cannot be an exemplar if it does not choose itself to be an exemplar as well. In the original work, Frey et al.[9] suggests the use of a dampening factor, λ, within [0,1] to avoid numerical oscillation when updating the responsibilities and availabilities. The update equations for responsibilities and availabilities now become

$$\rho(i,j) = \lambda \rho^{old}(i,j) + (1-\lambda)\rho^{new}(i,j) \tag{6}$$

$$\alpha(i,j) = \lambda \alpha^{old}(i,j) + (1-\lambda)\alpha^{new}(i,j) \tag{7}$$

Our extensive experimentation has empirically shown that dampening values, such as $\lambda = 0.5$, work reliably well. Though AP works very well, in general, across various data modalities, it has a tendency to produce a larger number of clusters. To effectively aggregate these clusters into larger and larger groups, we next discuss AP's hierarchical extension.

4.2 Hierarchical Affinity Propagation

Hierarchical Affinity Propagation (HAP) extends AP to allow tiered clustering of the data. The algorithm introduced by Givoni et al.[11] allows communication between levels which makes it more optimal then the greedy approach introduced by Xiao et al.[24] Similar to AP, the data points are considered nodes in a network connected by arcs with negative similarity weights. During every iteration, responsibility and availability messages are sent from each node to every other node. These messages are slightly different than Eq. (1) and Eq. (2) because they also take into account preferences between levels. For simplicity, let $\rho(i,j) = \rho_{ij}$, $\alpha(i,j) = \alpha_{ij}$ and $s(i,j) = s_{ij}$. The level-wise update equations are

$$\rho_{ij}^{l>1} \leftarrow s_{ij}^{l} + \min[\tau_i^l, -\max_{k s.t. k \neq j} \{\alpha_{ik}^l + s_{ik}^l\}] \tag{8}$$

$$\alpha_{ij}^{l<L} \leftarrow \min\left\{0, c_j^l + \phi_j^l + \rho_{jj}^l + \sum_{k s.t. k \notin \{i,j\}} \max\{0, \rho_{kj}^l\}\right\} \tag{9}$$

$$\alpha_{jj}^{l<L} \leftarrow c_j^l + \phi_j^l + \sum_{k s.t. k \neq j} \max\{0, \rho_{kj}^l\} \tag{10}$$

where L is the number of levels defined by the user and $l \in \{1, \cdots, L\}$. Similarly to AP, to avoid numerical oscillation, the responsibility and availability messages are dampened by λ at every level l.

$$\rho_{ij}^{l} = \lambda \rho_{ij}^{l}(old) + (1-\lambda)\rho_{ij}^{l}(new) \tag{11}$$

$$\alpha_{ij}^{l} = \lambda \alpha_{ij}^{l}(old) + (1-\lambda)\alpha_{ij}^{l}(new) \tag{12}$$

Notice now that the similarity matrix s_{ij}^{l} is allowed to vary level-wise. A variety of strategies can be employed to update the level-specific similarities. We have achieved good results by simply taking into consideration the cluster relationship of the previous level:

$$s_{ij}^{l+1} = s_{ij}^{l} + k \max_{js.t.j \neq i}[\alpha_{ij}^{l} + \rho_{ij}^{l}] \tag{13}$$

where k is a constant value within [0,1]. This updates the relation in level $l+1$ between data points by negatively increasing the similarity between points that belong to different clusters in level l and enforces the similarity between points that fall under the same cluster in level l.

HAP also introduces two inter-level messages that send messages between the different levels. These messages are denoted by τ, which receives messages from the lower level and ϕ, which receives messages from the upper level.

$$\tau_{j}^{l+1} = c_{j}^{l} + \rho_{jj}^{l} + \sum_{ks.t.k \neq j} \max(0, \rho_{kj}^{l}) \tag{14}$$

$$\phi_{i}^{l-1} = \max_{k}(\alpha_{ik}^{l} + s_{ik}^{l}) \tag{15}$$

In the HAP algorithm, at every level, the cluster preference c_{i}^{l} is updated using Eq. (16).

$$c_{i}^{l} \leftarrow \max_{i}\{\alpha_{ij}^{l} + \rho_{ij}^{l}\} \tag{16}$$

Similar to AP, HAP is an iterative method that continues until responsibilities and availabilities settle down across the levels. This can be monitored to develop the stopping criteria, but in practice the algorithm is typically run for a fixed number of iterations. At the end of the iterations, the cluster assignment e_{j}^{l} is updated as in Eq. (4) for each level l, separately. In Algorithm 15, we detail the pseudo-code implementation of HAP. Given this description of HAP, we now proceed to discuss MapReduce and our particular parallelization strategy.

4.3 MapReduce

The MapReduce programming model[4] is an abstract programming paradigm independent of any language that allows the processing workload of the implemented algorithm to be balanced over separate nodes within a computer cluster. The expectation is that the decomposed and distributed computations run more quickly and reliably across a cluster than a comparable sequential task. Under MapReduce, the data is passed between functions in ⟨key, value⟩ pairs. The type of the keys and values is determined by the nature of the input data set; however, all keys and all values must have matching type to other keys and values, respectively. The data in a MapReduce job are manipulated by the following functions in order: Map, Combine, Partition, and Reduce. To elaborate:

Map A mapper is a function that takes as input one ⟨key, value⟩ pair and outputs a finite set of intermediate ⟨key, value⟩ pairs where the input and output data types need not necessarily correspond in any fashion. After all parallelized Map operations have completed, execution continues to the Combine phase.

Algorithm 1 Hierarchical Affinity Propagation

Input: Similarity, Levels, Iterations, λ
Initialize: $A = 0$, $R = 0$, $\tau = \infty$, $\phi = 0$, $c = 0$, $e = 0$
for $iter = 1 \rightarrow$ Iterations **do**
  for $l = 1 \rightarrow$ Levels **do**
    Update ρ_{ij}^l Eq. (8) & Dampen ρ^l Eq. (11)
    Update α_{ij}^l Eq. (9) & Eq. (10) & Dampen α^l Eq. (12)
    Update ϕ_j^l Eq. (15)
    Update τ_j^l Eq. (14)
    Update c_j^l Eq. (16)
    Optional Update s_{ij}^l Eq. (13)
  end for
end for
for $l = 1 \rightarrow$ Levels **do**
  Update e_j^l Eq. (4)
end for

Combine The Combine phase is an optional step of a MapReduce job that is often omitted, as the mapper has the same computational abilities as the combiner. The combiner acts similarly to the reducer, except it only receives information from a single mapper. There is one combiner per mapper. The combiner affords the programmer an opportunity to process the ⟨key, value⟩ pairs coming out of the corresponding mapper before the Partition phase. Since combiners are limited to information coming out of a single mapper, they are thus limited in terms of performing meaningful computations. In some cases where the data needed to perform the Reduce phase is available on a per-mapper basis, the combiner would act as the reducer. This is advantageous because the Combine phase occurs before the computationally expensive Partition phase.

Partition The Partition takes the output of the mapper or combiner and aggregates ⟨key, value⟩ pairs with the same key to send to the reducers. This phase is computationally expensive due to the frequency of necessary I/O operations when moving data between DataNodes, and thus introduces a large amount of overhead to the MapReduce task.

Reduce A reducer is a function that takes as input one intermediate key and the corresponding aggregate list of values related with that key. This input can be thought of as a set of ⟨key, value⟩ pairs with the same key, *e.g.* ⟨k1,v1⟩, ⟨k1,v2⟩, ⟨k1,v3⟩. The reducer outputs a final finite set of ⟨key, value⟩ pairs.

Our development was conducted using the Hadoop implementation of MapReduce, which in principle incorporates these MapReduce components. The MapReduce paradigm is ideally suited for parallelizing AP and HAP. Both algorithms are modeled as pairwise message passing techniques, which immediately suggests that computational efficiency can be significantly improved by independently and simultaneously calculating the message updates. A detailed description of our approach follows.

4.4 Parallelization Strategy

Our overarching MapReduce approach for HAP was motivated by viewing the major update equations for HAP (see Algorithm 15) as tensorial mathematical constructs.[18] One can simply view these tensorial constructs as two or three dimensional matrices. As an illustration, consider the availability update in Eq. (9). Interpreting this as a 3D tensor translates to the (i, j) pair indexing the row (data point i) and column (data point j), and l indexing the depth of the hierarchy. The remaining equations can be viewed similarly. The HAP algorithm can be parallelized because all the updates to the various tensors require only a subset of the information provided. Therefore, the updates can be split up into different jobs and each job will receive the subset of data needed to evaluate the update.

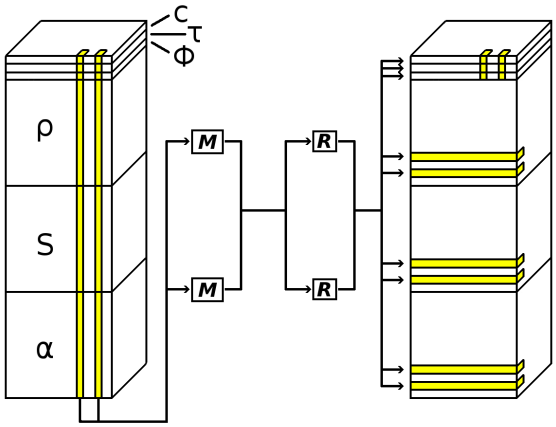


Figure 2: Parallelization Scheme

To achieve a balance between computational partitioning and efficient formatting for data representation on the HDFS, all the data is constructed as three dimensional tensors. In support of the fault tolerance aspect of MapReduce, it is important to retain a copy at all times of the S, α, ρ, c, τ, and ϕ tensors. (Recall S, α, ρ, and c refer to the Similarity, Availability, Responsibility, and Cluster Preferences, respectively.) For the S, α, and ρ tensors, the dimensions represent the nodes, the exemplars, and the levels. Since there are N nodes, N exemplars, and L levels, these tensors contains LN^2 values. For the c, τ, and ϕ tensors, the first two dimensions represent the index and level and the depth dimension has length one. Since there are N indices and L levels, these tensors contains LN values. In the sequel, for the S, α, and ρ tensors, the node dimension will be iterated by i, the exemplar dimension will be iterated by j, and the level dimension iterated by l. As for the c, τ, and ϕ tensors, the index dimension will be iterated by both i and j.

With these underlying structures, data must be deconstructed and represented as ⟨key,value⟩ pairs for use in the MapReduce framework. There are two formats for storing the information: node-based and exemplar-based formatting.

In the node-based format, the keys are string tuples, (i, l, ξ), where i represents the node, l represents the level, and ξ represents the tensor $(\alpha, \rho, ...)$. The values, represented by ν, are the vectors for the i^{th} node of the matrix on the l^{th} level of the tensor.

In the exemplar-based format, the keys are string tuples, (j, l, ξ), where j represents the exemplar, l represents the level, and ξ represents the tensor. The values, represented by ν, are the vectors for the j^{th} exemplar of the matrix on the l^{th} level of the tensor. With the data thus represented, MapReduce jobs must be constructed to manipulate the information using the given HAP equations.

In our parallelization scheme, HAP is broken down into three separate MapReduce jobs. The first job handles updating τ, c, and ρ. The second job handles updating ϕ and α. These first two jobs loop for a set number of iterations. At the end of the iterations, the final job extracts the cluster assignments on each level. Due to dependencies set out in the equations, the ρ update must occur first. Therefore, τ and c are not updated during the first iteration. In all other iterations they occur before the Responsibility update. At the start of each iteration, the data will be in exemplar-based format. After the first job, the data will have switched to node-based format. The second job converts the data back to exemplar-based format to begin a new iteration or to be used as input to the final job. See Figure 2 for a visual representation of the parallelization scheme. The figure represents what happens to the data during either of the first two jobs. The tensors have been stacked to show how the indices line up. The yellow strips on the left represent information being passed to one mapper, one strip per mapper. The focus of each mapper is on providing the reducers with the necessary information. As the data comes out of the job, the switch between exemplar-based and node-based formats can be easily seen. The output is now ready for use by the next job, which will follow a similar flow. The following sections will provide in-depth explanations of each MapReduce job.

4.4.1 Updating τ, c and ρ Job

This job takes as input the exemplar-based representation of the data and outputs the node-based representation of the data with updated values. In the first iteration, τ and c are not updated due to previously mentioned dependencies. In this MapReduce job, the mapper deconstructs the exemplar-based vectors into node-based values for the reducer to reconstruct node-based vectors.

Map Each mapper receives a key describing a unique (j, l, ξ) combination and a value with the corresponding vector. There will be $6LN$ map tasks since there are N unique options for j, L unique options for l, and 6 unique options for ξ. The indices of the vector represent the nodes; thus, the mapper iterates over the vector with i. It is important for values not updated in this job to still be mapped through the reducers. This prevents data loss.

- τ : In order to solve for the $(i, l+1)$ vector of τ, the information needed is the vector at (j, l) of c and ρ. Therefore, the mapper will grab c and ρ by the ξ and send their values to the reducers using the intermediate key $(i, l+1)$. The intermediate value is the string tuple, (j, l, ξ, ν). In this case, the mappers send information up a level to the reducers which receive it from the level below.
- c : In order to solve for the (i, l) vector of c, the information needed is the vector at (j, l) of α and ρ. Therefore, the mapper will grab α and ρ by the ξ and send their values to the reducers using the intermediate key (i, l). The intermediate value is the string tuple, (j, l, ξ, ν). On top of this information, the values where i equals j in α and ρ on the l^{th} level is needed at every (i, l). Therefore, when i equals j, a loop is run iterating k over each node and the values are sent to the reducers using the intermediate key (k, l). The intermediate value is the string tuple (k, l, ξ', ν). ξ' is used to represent an alteration the usual ξ string that is necessary for recognizing the difference between values on the diagonal and the other values.
- ρ : In order to solve for the (i, l) vector of ρ, the information needed is the vector at (j, l) of S, τ, and α. Therefore, the mapper will grab S, τ, and α by the ξ and send their values to the reducers using the intermediate key (i, l). The intermediate value is the string tuple, (j, l, ξ, ν).
- S, ϕ, and α : These values must be sent to the reducers so that information is not lost. Therefore, the mapper will grab S, ϕ, and α by the ξ and send their values to the reducers using the intermediate key (i, l).The intermediate key is string tuple (i, l). The intermediate value is the string tuple, (j, l, ξ, ν).

Reduce Each reducer receives a key describing a unique (i, l) combination and a list of values which will be used to reconstruct the 6 node-based vectors, the 2 node-based vectors from the level below and the 2 special diagonal vectors. There will be LN reduce tasks since there are N unique options for i and L unique options for l. The indices of the constructed vector represent the exemplars so the reducer iterates over the vector with j. It is important for values that are not being updated by this job that the old value is still output so that no data is lost.

- τ : The node-based vectors of the c and ρ for $l-1$ are constructed. The (i, l) vector of τ is calculated following Eq. (14). Then the information is output using the node-based format for use by the next MapReduce job. The output key is the string tuple (i, l, ξ) and the output value is the node-based vector.
- c : The node-based vectors of α and ρ and the special diagonal vectors of α and ρ are constructed. The (i, l) vector of c is calculated following Eq. (16). Then the information is output using the node-based format for use by the next MapReduce job. The output key is the string tuple (i, l, ξ) and the output value is the node-based vector.
- ρ : The node-based vectors for the S, τ and α are constructed. The (i, l) vector for ρ is calculated following Eq. (8). Then the information is output using the node-based format for use by the next MapReduce job. The output key is the string tuple (i, l, ξ) and the output value is the node-based vector.

- S, ϕ, and α : The node-based vectors for the S, ϕ, and α are constructed. Then the information is output using the node-based format for use by the next MapReduce job. The output key is the string tuple (i, l, ξ) and the output value is the node-based vector.

4.4.2 Updating α and ϕ Job

This job takes as input the node-based representation of the data and outputs the exemplar-based representation of the data with updated values. In this MapReduce job, the mapper deconstructs the node-based vectors into exemplar-based values for the reducer to reconstruct exemplar-based vectors.

Map Each mapper receives a key describing a unique (i, l, ξ) combination and a value with the corresponding vector. There will be $6LN$ map tasks since there are N unique options for i, L unique options for l, and 6 unique options for ξ. The indices of the vector represent the nodes so the mapper iterates over the vector with j. It is important for values that are not being updated by this job that the old value is still mapped to the reducers so that no data is lost.

- ϕ : In order to solve for the $(j, l-1)$ vector of ϕ, the information needed is the vector at (i, l) of α and S. Therefore, the mapper will grab α and S by the ξ and send their values to the reducers using the intermediate key $(j, l-1)$. The intermediate value is the string tuple, (i, l, ξ, ν). In this case, the mappers send information down a level to the reducers which receive from the level above.
- α : In order to solve for the (j, l) vector of α, the information needed is the vector at (i, l) of c, ϕ, and ρ. Therefore, the mapper will grab c, ϕ, and ρ by the ξ and send their values to the reducers using the intermediate key (j, l). The intermediate value is the string tuple, (i, l, ξ, ν).
- S, τ, c, and ρ : These values must be sent to the reducers so that information is not lost. The intermediate key is the string tuple (j, l). The intermediate value is the string tuple, (i, l, ξ, ν).

Reduce Each reducer receives a key describing a unique (j, l) combination and a list of values which will be used to reconstruct the 6 exemplar-based vectors and the 2 node-based vectors from the level above. There will be LN reduce tasks since there are N unique options for j and L unique options for l. The indices of the constructed vector represent the nodes so the reducer iterates over the vector with i. It is important for values that are not being updated by this job that the old value is still output so that no data is lost.

- ϕ : The exemplar-based vectors for the α and S for $l+1$ are constructed. The (j, l) vector for ϕ is calculated following Eq. (15). Then the information is output using the exemplar-based format for use by the next MapReduce job. The output key is the string tuple (j, l, ξ) and the output value is the exemplar-based vector.
- α : The exemplar-based vectors for the c, ϕ, and ρ are constructed. The (j, l) vector for α is calculated following Eq. (9). Then the information is output using the exemplar-based format for use by the next MapReduce job. The output key is the string tuple (j, l, ξ) and the output value is the exemplar-based vector.
- S, τ, c, and ρ : The exemplar-based vectors for the S, τ, c, and ρ are constructed. Then the information is output using the node-based format for use by the next MapReduce job. The output key is the string tuple (j, l, ξ) and the output value is the exemplar-based vector.

4.4.3 Extracting Cluster Assignments Job

This job takes as input the exemplar-based representation of the data and outputs the cluster assignments. In this MapReduce job, the mapper deconstructs the exemplar-based vectors into node-based values for the reducer to reconstruct node-based vectors.

Method	Time
HAP-Sequential	$O(kLN^2)$
HAP-MapReduce	$O(kLN^2/M) = O(kN)$

Table 1: Complexity Table

Map Each mapper receives a key describing a unique (j, l, ξ) combination and a value with the corresponding vector. There will be $6LN$ map tasks since there are N unique options for j, L unique options for l, and 6 unique options for ξ. The indices of the vector represent the nodes so the mapper iterates over the vector with i. Since this is the last step, only the required information has to pass to the reducer and the other information can be neglected.

- Cluster Assignment : In order to solve for the (i, l) vector of Cluster Assignments, the information needed is the vector at (j, l) of α and ρ. Therefore, the mapper will grab α and ρ by the ξ and send their values to the reducers using the intermediate key (i, l). The intermediate value is the string tuple, (j, l, ξ, ν). On top of this information, the values where i equals j in α and ρ on the l^{th} level is needed at every (i, l). Therefore, when i equals j, a loop is run iterating k over each node and the values are sent to the reducers using the intermediate key (k, l). The intermediate value is the string tuple (k, l, ξ', ν). ξ' is an alteration of the usual ξ string used to distinguish between values on the diagonal and other values.

Reduce Each reducer receives a key describing a unique (i, l) combination and a list of values which will be used to reconstruct the 2 node-based vectors and the 2 special diagonal vectors. There will be LN reduce tasks since there are N unique options for i and L unique options for l. The indices of the constructed vector represent the exemplars so the reducer iterates over the vector with j.

- Cluster Assignment : The node-based vectors for the α and ρ and special diagonal vectors for the α and ρ are constructed. The (i, l) vector for Cluster Assignments is calculated following Eq. (16). The information is then output to disk as the final output of the algorithm. The output key is the string tuple (i, l) and the output value is the string (e) where e is the exemplar chosen to be the i^{th} node on the l^{th} level.

4.5 Runtime Complexity

A standard sequential HAP implementation must necessarily have a runtime complexity of $O(kLN^2)$ where k represents the number of algorithmic iterations, run either as a hard limit or until convergence is reached, L represents the number of output levels requested, and N represents the size of S, $(L \times N \times N)$. The runtime complexity is a direct result of iterating over all three dimensions of the tensors for each iteration.

By implementing the algorithms in the MapReduce framework, we are able to achieve superior runtime complexity. Under MapReduce, the HAP runtime complexity reduces to a linear relationship with the data, assuming the total number of VMs on the cluster, M, scales to LN.

$$\lim_{M \to LN} O(\frac{kLN^2}{M}) = O(kN) \tag{17}$$

In HAP, M can only scale up to a maximum of LN because M is limited to the number of tasks that can be evaluated at the same time. In this case, it is limited to the minimum of the $6LN$ mapper tasks and the LN reducer tasks, where the constant factor six represents the number of tensor identifications introduced into the algorithm, namely $\alpha, \rho, S, \tau, \phi$, and c. These results are summarized in Table 1 and Eq. (17).

5. EXPERIMENTAL RESULTS

To demonstrate the effectiveness and adaptability of the proposed approach, we executed validation experiments on several data sets with a variety of modalities: textual, imagery, and synthesized numerical. Where applicable, we compared our performance to a popular MapReduce hierarchical clustering algorithm based on code currently available in Mahout. At its core, their hierarchical clustering is based on a level-wise K-means clustering approach which we refer to as Hierarchical K-Means (HK-Means). With K-Means as the foundation, HK-Means requires the number of cluster centers as input. Since our method does not explicitly impose this requirement, we adopted the initialization method of running canopy clustering, also available in Mahout, to discover the "natural" number of centers. We then use these cluster centers to seed HK-Means. In order to truly gain an objective understanding of HAP performance versus HK-Means, we use several extrinsic cluster quality metrics to assess their respective aggregation capabilities.

5.1 Metrics

The extrinsic quality measure is generated based on the predefined cluster labels associated with each data point, *i.e.* the assumed true cluster label of the data. The first extrinsic measure is defined as purity of a cluster[21] which is computed the following way:

$$P = \frac{1}{N} \sum_{r} \max_{i} \{n_r^i\}, \tag{18}$$

where N is the number of data points, r is the number of clusters, i is the number of classes, and n_r^i defines the number of data points from the i^{th} class assigned to cluster k. We compute Recall,

$$R = \frac{1}{N} \sum_{i} \max_{r} \{n_r^i\}, \tag{19}$$

P and R have values between 0 and 1, where 1 is a 100 percent clustering accuracy with respect to the assumed true clustering labels. We compute F-Score, given R and P,

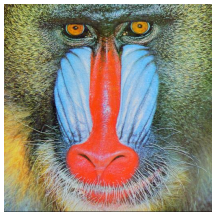
$$F = \frac{2PR}{P+R} \tag{20}$$

We also compute the entropy[26] with the following equation:

$$E(S_r) = -\frac{1}{\log q} \sum_{i=1}^{q} \frac{n_r^i}{n_r} \log \frac{n_r^i}{n_r} \tag{21}$$

$$E = \sum_{r=1}^{k} \frac{n_r}{N} E(S_r) \tag{22}$$

Entropy is a positive value where the closer the 0 the better the clustering algorithm. Perfect clustering is when $E(S_r) = 0$.


a) Original "Mandrill"


b) 15 Exemplars


c) 7 Exemplars


d) 6 Exemplars

Figure 3: "Mandrill" 103x103

5.2 Results

Hierarchical Affinity Propagation has shown great results in image segmentation as shown in Figure 3 & Figure 4. Due to the large data set, running HAP on a standard computer would not output a result in a timely manner. So we run HAP on the MapReduce framework which allowed hierarchical clustering. The "Mandrill" image, Figure 3, is of size 103×103 which makes the data 10609 pixels in size. The similarity input was computed using the negative Euclidean distance between all pixels on the RGB levels. The diagonal, or preference entries, were selected as random numbers within $[-10^6, 0]$. As for the other parameters, we set the iterations to 30, the number of levels to $L = 4$ and the dampening factor to $\lambda = 0.5$.

The results are shown in Figure 3. The top left image is the original image. The top right image is the lowest level where the pixels were grouped into 15 clusters. The bottom left image is the second level where the pixels were grouped into 7 clusters. Finally, the bottom right image is the highest level where the pixels were grouped into 6 clusters. From these images we can still see the mandrill's shape and most of its colors, but at the highest level it appears fuzzier. This is because the members of the same clusters were given the color of the exemplar. Recall how we used $L = 4$ but only show 3 images, this is because our highest 2 levels actually gave the exact same images and it has to do with the number of iterations requested. In some tests, this image resulted in even fewer clusters.

Similarly, the "Buttons" image, Figure 4, is of size 120×100, resulting in a data set of 12000 pixels. We set the similarity matrix as the negative Euclidean distance between all the pixels' RGB vectors and the diagonal preferences were set to random numbers within $[-10^6, 0]$. As for the parameters, we set the iterations to 30, the number of levels to $L = 3$ and the dampening factor to $\lambda = 0.5$.

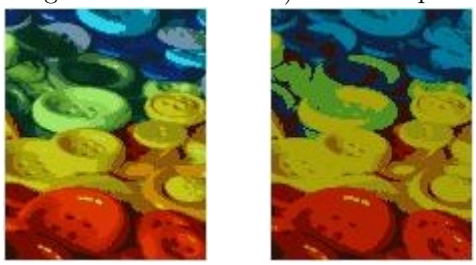
The results of running HAP on the "Buttons" image are shown in Figure 4. The top left image is again the original image. The top right image is the lowest level where the pixels were grouped into 154 clusters. The bottom left image is the second level where the pixels were grouped into 25 clusters. Finally, the bottom right image is the highest level where the pixels were grouped into 11 clusters. Similar to the "Mandrill" data set, the highest level of the hierarchy appears fuzzier than the original image due to similar colors clustering underneath a single exemplar.

For text mining, we utilized several distinct publicly available pre-classified data sets. Results are as yet inconclusive.

5.3 Relative Performance

In order to test the scalability of the HAP algorithm with respect to speed, we use the data set "Aggregation"[10] which is a shape set composed of 788 two-dimensional points. The purpose of these tests was to observe trends

a) Original "Buttons" b) 154 Exemplars



c) 25 Exemplars d) 11 Exemplars

Figure 4: "Buttons" 120x100

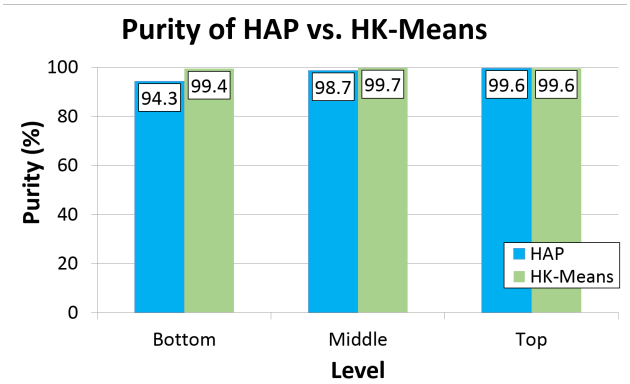


Figure 5: Purity levels of HAP vs. HK-Means. HAP posts results highly competitive with HK-Means.

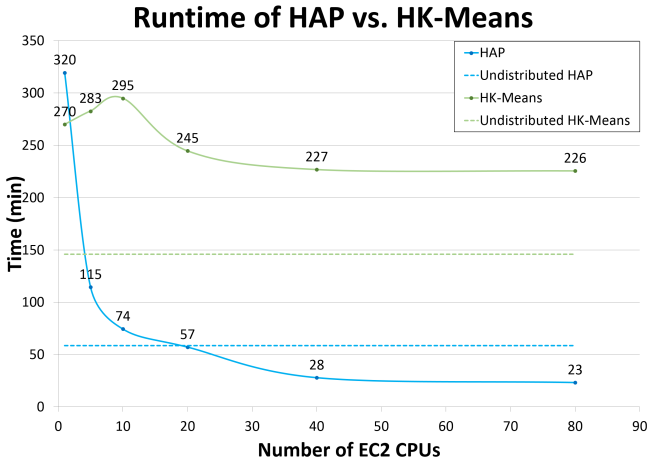


Figure 6: Time vs. Number of EC2 CPUs. Our MapReduce HAP better utilizes available compute resources to significantly improve runtime.

in algorithm runtime as cluster computing power increased, as well as to determine the benefits of running in a distributed environment as compared to an undistributed environment (a single-machine Hadoop cluster). Hadoop clusters were provided using Amazon's EMR service to dynamically create clusters of standard EC2 instances. Cluster computing power was scaled both by increasing the number of VMs within a cluster and by provisioning more powerful VMs. The two VM instance types used are: (1) the m1.small, which has 1.7 GB of memory and is considered to have 1 EC2 Compute Unit (ECU) with 160 GB of instance storage and a 32-bit architecture, and (2) the m1.xlarge, which has 15 GB of memory, 8 ECU, 1690 GB of instance storage, and a 64-bit architecture. The single-machine Hadoop cluster utilized to simulate an undistributed environment has 8 GB of memory, 8 ECU, 40 GB of machine storage, and a 64-bit architecture.

For comparison to another state-of-the-art MapReduce clustering methodology, our HAP algorithm was benchmarked against HK-Means. Due to its inherently parallel design, HAP immediately begins to benefit from being placed in a distributed environment. Represented by a solid blue line in Figure 6, HAP runtime decreases by 64%, from 320 minutes to 115 minutes, when cluster computational power is increased by just 4 additional ECU. HAP eventually reaches the threshold of a linear relationship with the size of the input data at a runtime of around 20 minutes, which is a 94% decrease from the single ECU cluster. Furthermore, at its best, HAP performs 66% faster in a distributed environment than the undistributed environment which is represented by the blue dotted line in Figure 6. In contrast, the HK-Means algorithm used in this experimentation, indicated by the solid red line in Figure 6, is not parallelized to the extent of HAP. Each single iteration of K-Means is structured under Mahout to distribute over a Hadoop cluster, but the hierarchical "Top Down" structure requires iterative executions of K-Means for each level. This lack of an overall parallelization scheme results in reduced performance at scale than HAP. HK-Means runtime initially increases by 8.5% when ECU is increased from 1 to 10 due to Hadoop cluster overhead, including network latency and I/O time. However, at 10 ECU, HK-Means overcomes this overhead and begins to benefit from the MapReduce parallelization scheme. This results in an eventual 16% runtime decrease between 1 and 80 ECU, at which point HK-Means eventually reaches a linear relationship with the data at a runtime around 225 minutes. Unlike HAP, HK-Means never surpasses its undistributed runtime threshold of 146 minutes indicated by the red dotted line in Figure 6.

With significantly faster runtimes, HAP still posts purity levels competitive with HK-Means, shown in Figure 5. This combination of speed and high performance is ideal for processing Big Data in a large-scale cloud computing environment.

6. CONCLUSIONS

The need for efficient and high performing data analysis frameworks remain paramount to the big data community. The Affinity Propagation (AP) clustering algorithm is rapidly becoming a favorite amongst data scientists

due to its high quality grouping capabilities, while requiring minimal user specified parameters. Recently, a multilayer structured version of the AP algorithm, Hierarchical Affinity Propagation (HAP), was introduced to automatically extract tiered aggregations inherent in many data sets. HAP is modeled as a message-based network that allows communication between nodes (data points) and between levels in the hierarchy, and mitigates many of the biases that arise in techniques that require one to input the number of clusters. In the present work, we have developed the first ever extension of HAP to address the big data problem—demonstrating an efficient parallel implementation using MapReduce that directly improves the runtime complexity from quadratic to linear. The novel tensor-based partitioning scheme allows for parallel message updates and utilizes a consistent data representation that is leveraged by mapper and reducer tasks. Our approach seamlessly allows us to cluster a variety of data modalities, which we experimentally showcased on data sets ranging from text to imagery. Our analysis and computational performance is competitive with the state-of-the-art in MapReduce clustering techniques.

6.1 Future Work

In the future, we plan to explore further optimizations in our HAP updates to accelerate convergence and significantly increase the operational scale by leveraging elastic computing resources. In general, however, the final goal for this project is to use the cluster assignment gained from HAP in combination with semantic metadata mined from the input data to create a semantically rich, interactive end user visualization. Achieving this goal has two parts: mining sematic metadata and visualizing the clusters.

6.1.1 Semantic Analysis

When reading in an input dataset, initially nothing is known about it. Clusters are obtained through the execution of HAP, but no metadata or semantic information is acquired. In order to provide descriptions of clusters and data points to the user in a visualization tool, semantic information or metadata must be scraped from the dataset and stored alongside clustering results. Multiple possible methods are available for this metadata mining. For example, sentiment analysis can be performed on textual data using frameworks like GATE[*] or Apache UIMA[†]. Texture analysis can also be run on images, etc. Because HAP is an unsupervised algorithm, the goal is to gather as much information as possible from preprocessing in order to feed the visualization software as much as possible.

6.1.2 Visualization Details

One major future goal which has already undergone prototyping is data visualization in an interactive user environment. The visualization software prototype is a web-based thin-client interface built with the Data-Driven Documents (D3.js) framework[‡]. D3.js is a Free and Open Source Software (FOSS) JavaScript library for creating powerful and interactive data visualizations. The motivating idea behind constructing a thin-client interface for visualization is enterprise-level performance. A thin-client web interface provides both dataset scalability and platform independence. The D3.js library will operate almost indistinguishably on any platform or on any input.

Figure 7 contains an example user interface which highlights how our solution presents meaningful information about an initially unknown dataset in an easy-to-use, easy-to-understand, portable, and scalable web interface.

ACKNOWLEDGMENTS

The authors acknowledge support from National Science Foundation (NSF) grant No. 1263011. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

The authors would also like to thank Nenad Mijatovic and Michal Frystacky for their contributions and efforts throughout the 2013 REU process.

[*] http://gate.ac.uk
[†] http://uima.apache.org
[‡] http://d3js.org

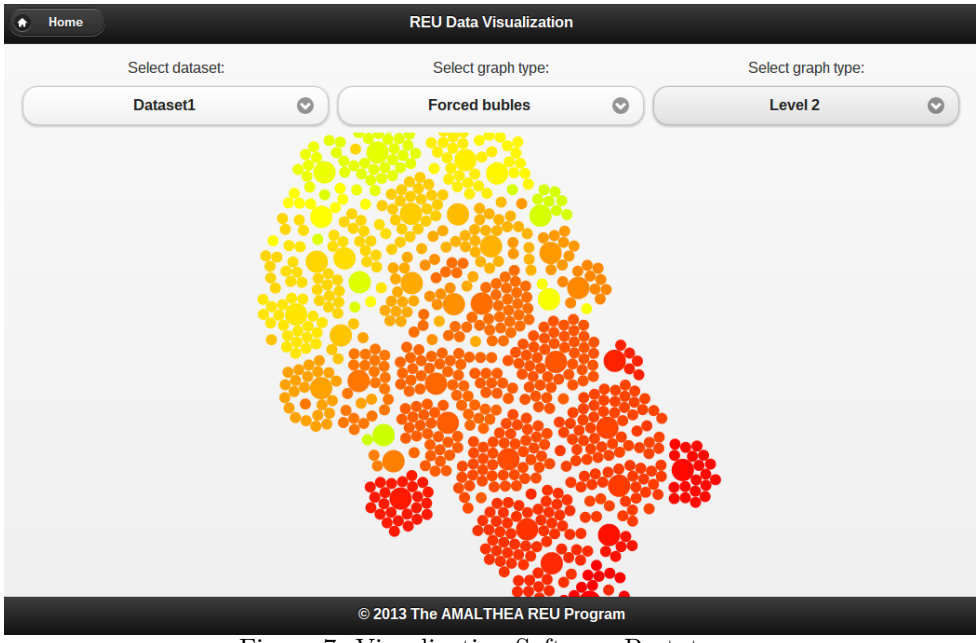


Figure 7: Visualization Software Prototype

REFERENCES

[1] Language manual - apache hive - apache software foundation, 2013. URL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual.

[2] C-T. Chu, S. K. Kim, Y-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Neural Information Processing Systems (NIPS)*, pages 281–288, 2007.

[3] Bi Ru Dai and I Chang Lin. Efficient map/reduce-based dbscan algorithm with optimized data partition. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 59–66, 2012. doi:10.1109/CLOUD.2012.42.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[5] Martin Ester, Hans Peter Kriegel, Jörg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.

[6] Rui Maximo Esteves, Thomas J. Hacker, and Chunming Rong. Cluster analysis for the cloud: Parallel competitive fitness and parallel k-means++ for large dataset analysis. In *CloudCom*, pages 177–184, 2012.

[7] The Apache Software Foundation. Apache mahout: Scalable machine learning and data mining, 2013. URL: http://mahout.apache.org.

[8] The Apache Software Foundation. Hadoop 1.1.2 documentation, 03 2013. URL: http://hadoop.apache.org/docs/r1.1.2.

[9] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315:2007, 2007.

[10] Aristides Gionis, Heikki Mannila, and Panayiotis Tsaparas. Clustering aggregation. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007. URL: http://doi.acm.org/10.1145/1217299.1217303, doi:10.1145/1217299.1217303.

[11] Inmar E. Givoni, Clement Chung, and Brendan J. Frey. Hierarchical affinity propagation. *CoRR*, abs/1202.3722, 2012. URL: http://uai.sis.pitt.edu/papers/11/p238-givoni.pdf.

[12] Renchu Guan, Xiaohu Shi, Maurizio Marchese, Chen Yang, and Yanchun Liang. Text clustering with seeds affinity propagation. *IEEE Trans. Knowl. Data Eng.*, 23(4):627–637, 2011. URL: http://dblp.uni-trier.de/db/journals/tkde/tkde23.html.

[13] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A k-means clustering algorithm. *Applied Statistics*, 28:100–108, 1978.

[14] S. Humbetov. Data-intensive computing with map-reduce and hadoop. In *Application of Information and Communication Technologies (AICT), 2012 6th International Conference on*, pages 1–5, 2012. doi:10.1109/ICAICT.2012.6398489.

[15] Alexa Internet Inc. Amazon site info, 07 2013. URL: http://www.alexa.com/siteinfo/amazon.com.

[16] Amazon Web Services Inc. Command line interface reference for amazon emr, 07 2013. URL: http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-cli-reference.html.

[17] L. Kaufman and P. Rousseeuw. *Clustering by Means of Medoids*. Reports of the Faculty of Mathematics and Informatics. Delft University of Technology. Fac., Univ., 1987. URL: http://books.google.com/books?id=HK-4GwAACAAJ.

[18] Haiping Lu, Konstantinos N. Plataniotis, and Anastasios N. Venetsanopoulos. A survey of multilinear subspace learning for tensor data. *Pattern Recogn.*, 44(7):1540–1551, July 2011. URL: http://dx.doi.org/10.1016/j.patcog.2011.01.004, doi:10.1016/j.patcog.2011.01.004.

[19] Geoffrey McLachlan and David Peel. *Finite Mixture Model*. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc, Portland, United States, 1 edition, 2000.

[20] T.R.G. Nair and K.L. Madhuri. Data mining using hierarchical virtual k-means approach integrating data fragments in cloud computing environment. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 230–234, 2011. doi:10.1109/CCIS.2011.6045065.

[21] Nachiketa Sahoo, Jamie Callan, Ramayya Krishnan, George Duncan, and Rema Padman. Incremental hierarchical clustering of text documents. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, CIKM '06, pages 357–366, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1183614.1183667, doi:10.1145/1183614.1183667.

[22] Hanli Wang, Yun Shen, Lei Wang, Kuangtian Zhufeng, Wei Wang, and Cheng Cheng. Large-scale multimedia data mining using mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 287–292, 2012. doi:10.1109/CloudCom.2012.6427595.

[23] Fei Wu, Wenhua Wang, Hanwang Zhang, and Yueting Zhuang. *Handbook of Research on Hybrid Learning Model: Advanced Tools, Technologies, and Applications*, chapter The Clustering of Large Scale E-Learning Resources, pages 94–104. IGI Global, 2010.

[24] Jianxiong Xiao, Jingdong Wang, Ping Tan, and Long Quan. Joint affinity propagation for multiple view segmentation. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–7, 2007. doi:10.1109/ICCV.2007.4408928.

[25] Cao Zewen and Zhou Yao. Parallel text clustering based on mapreduce. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 226–229, 2012. doi:10.1109/CGC.2012.128.

[26] Ying Zhao and George Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. *Mach. Learn.*, 55(3):311–331, June 2004. URL: http://dx.doi.org/10.1023/B:MACH.0000027785.44527.d6, doi:10.1023/B:MACH.0000027785.44527.d6.

APPENDIX A. HADOOP DOCUMENTATION

A.1 Installing Hadoop

A fresh copy of Hadoop can be acquired from the Apache Hadoop project website*. Alternatively, depending on your operating system, a Hadoop package might already be available in your software repositories. The version of Hadoop used in this documentation is 1.0.3. Specific details might change between different versions.

Once an up to date copy of the software has been acquired, untar the archive and enter it.

```
$ tar xvf hadoop-1.1.2.tar.gz
$ cd hadoop-1.1.2
```

Your distribution of Hadoop should contain many files and folders, several important ones will be highlighted below.

```
$ ls
...
drwxr-xr-x 2 jrouly users 4.0K Jun 6 17:06 bin
drwxr-xr-x 2 jrouly users 4.0K Jun 6 17:06 conf
drwxr-xr-x 6 jrouly users 4.0K Jun 6 17:06 docs
drwxr-xr-x 5 jrouly users 4.0K Jun 6 17:06 lib
drwxr-xr-x 16 jrouly users 4.0K Jun 6 17:06 src
...
```

- `hadoop/bin` contains the Hadoop executable, as well as commands to start and stop the Hadoop server.
- `hadoop/conf` contains server configuration files which are used to define the kind of server that will be running on your machine or cluster of machines.
- `hadoop/docs` contains useful documentation for the beginner.
- `hadoop/lib` contains several additional Jar libraries of code which will be useful when developing for Hadoop projects.
- `hadoop/src` contains the source code for the entire Hadoop project. Use this as reference or examples of programming style.

Your directory will also contain several important archive files.

```
-rw-r--r--  1 jrouly users 6.7K Jan 30 21:05 hadoop-core-1.0.3.jar
-rw-r--r--  1 jrouly users 140K Jan 30 21:05 hadoop-examples-1.0.3.jar
```

These and other `.jar` files are libraries of code which might be useful or even required when developing for Hadoop projects. They contain the class files necessary to seamlessly integrate with the Hadoop framework. Once you have a full copy of the Hadoop distributable, you must configure both your new Hadoop server and your existing system before moving on.

*http://hadoop.apache.org

A.2 Configuring Hadoop and its Cluster Machines

A.2.1 Allowing Local SSH Connections

Hadoop operates by dividing processing tasks among all available machines connected to the cluster. There are two main machine designations: NameNode and DataNode. DataNodes serve up the data stored on the Hadoop Distributed Filesystem (HDFS), while NameNodes facilitate and organize the operations of other machines. To allow this communication to take place, Hadoop must be able to access the other nodes in its cluster remotely. Specifically, your machine must allow remote SSH connections, with or without a password.

Install an SSH client and server on your machine, then configure your SSH server to accept local connections without a password (this is an optional step).

```
# apt-get install openssh
$ ssh-keygen t rsa -P
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

Make sure you can connect to your local machine.

```
$ ssh localhost
$ exit
```

A.2.2 Hadoop Server Configuration Files

There are four configuration files which need updating, and they are all located in `hadoop/conf`.

hadoop-env.sh

Uncomment the line defining your `JAVA_HOME` directory. This allows Hadoop to properly execute Java code.

```
export JAVA_HOME=/path/to/jvm
```

core-site.xml

This file contains several directives which will need to be defined.

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</name>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/path/to/hdfs</value>
  </property>
</configuration>
```

`fs.default.name` defines where the default Hadoop web server is located. `hadoop.tmp.dir` defines the local directory where the HDFS will be constructed*.

hdfs-site.xml

This file configures the Hadoop distributed file system. When scaling up the size of a cluster, this file will need to be updated. For the purposes of this document, a single-node cluster will be herein defined.

*Note: You must have read/write permissions on the `hadoop.tmp.dir` directory

```
  <configuration>
    <property>
      <name>dfs.replication</name>
      <value>1</value>
    </property>
  </configuration>
```

mapred-site.xml

This file controls overrides to the MapReduce system, setting upper limits on the number of map and reduce tasks allowed. These two directives are optional, but recommended.

```
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:9001</value>
  </property>
  <property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>4</value>
  </property>
  <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>2</value>
  </property>
</configuration>
```

A.2.3 Placing Hadoop on the System PATH

In order to access the Hadoop command from anywhere on your system, it is necessary to add the Hadoop binary to your System `PATH`. The `PATH` is a colon (":") separated list of directories which defines locations the terminal shell will look for commands. Depending on your system, the location of the file defining your `PATH` may vary. Defining the `PATH` for a single session will suffice for now.

```
$ export PATH=/path/to/hadoop/bin:$PATH
```

A.2.4 Initializing the Hadoop Cluster

Before you can begin using the HDFS, it must be formatted. Formatting the HDFS may also be useful later down the road during debugging or development.

```
$ hadoop namenode -format
```

A.3 Starting the Hadoop Cluster

Congratulations — your Hadoop cluster should now be set up and ready to run. Start up your Hadoop server.

```
$ start-all.sh
```

To check if everything started up successfully, the `jps` command lists all processes currently running under the Java Virtual Machine (JVM). Hadoop's five components should show up (with different process IDs).

```
$ jps
940 NameNode
1473 TaskTracker
11011 Jps
1210 SecondaryNameNode
1074 DataNode
1318 JobTracker
```

Hadoop offers several example MapReduce problems to illustrate the flexibility of the project. Try approximating the value of π to see your new Hadoop cluster in action.

```
$ hadoop jar /path/to/hadoop/hadoop-examples-*.jar pi 10 100
```

To stop your Hadoop cluster at any point, issue the stop command.

```
$ stop-all.sh
```

A.3.1 Troubleshooting

There are several possible sources of error when configuring a Hadoop cluster. Most stem from possible permissions errors wherein the Hadoop framework attempts to access a log or PID directory owned by the system. The following commands may be useful in such situations, but make sure to use them sparingly and carefully.

1. `chmod`: modify the permission settings of a file or folder
2. `chown`: change the ownership of a file or folder
3. `chgrp`: change the group membership of a file or folder

Alternatively, configure the Hadoop server to store its log and PID files in a directory already owned by the executing user.

A.4 Running Projects on Hadoop

Once the Hadoop cluster is up and running successfully, your Hadoop MapReduce project can now be executed. There are two steps to this: setting up the environment to run, and running the project

A.4.1 Configuring the HADOOP_CLASSPATH

In the `hadoop-env.sh` configuration file, uncomment the line containing `HADOOP_CLASSPATH` and define it to include any Hadoop projects you are currently developing and any Jar libraries containing code which will be used by your applications. The `HADOOP_CLASSPATH` is a colon (":") separated list of directories and `.jar` files in the same format as the System `PATH` described earlier.

```
export HADOOP_CLASSPATH=/path/to/project:$HADOOP_CLASSPATH
export HADOOP_CLASSPATH=/path/to/library.jar:$HADOOP_CLASSPATH
```

To check that `hadoop-env.sh` has been configured properly, the current `HADOOP_CLASSPATH` can always be viewed dynamically.

```
$ hadoop classpath
```

A.4.2 Running a Program under Hadoop

Once Hadoop has access to the root directory of a project, the project code can be executed under the Hadoop framework.

```
$ hadoop <Main class> <arguments>
```

For example, to run the provided source code:

```
$ hadoop root.Driver
```

In order to run a project from a Jar archive, first create the Jar archive*

```
$ jar cvf myproject.jar /path/to/project/*
```

then run the Jar archive using the Hadoop command. This is the same as before with the addition of the `jar` directive.

```
$ hadoop jar myproject.jar <Main class> <arguments>
```

For example, to run the provided distributable Jar:

```
$ hadoop jar distr/driver.jar
```

APPENDIX B. TIPS AND TRICKS

B.1 Amazon Elastic MapReduce (EMR) Tips and Tricks

One of the most common issues with running a custom JAR on EMR is premature failure. This can be caused by several possible issues:

- make sure all outside libraries have been properly imported and added to the Path using a bootstrap script,
- make sure all files referenced on an Amazon Simple Storage Service (S3) drive or other outside webhosting are publicly available,
- check the job's `stderr` log for any possible compilation issues,
- make sure the custom JAR was compiled against the same version of Hadoop as is on the EMR cluster.

Occasionally a custom JAR will fail on EMR and print a stack trace referencing a Google Cache compilation error. The cause of this error is unknown, but involves the EMR infrastructure and the libraries compiled into certain Amazon Elastic Compute Cloud (EC2) Virtual Machines (VMs). The solution is generally to attempt running on different instance types, specifically the Standard Small instance.

When creating an EMR cluster, it is sometimes useful to diversify the roles of VMs. The EMR Command Line Interface (CLI) gives the option of adding instances to an existing EMR cluster under certain groups. Specifically, instances can be a part of the Master, Core, and Task instance groups. The Master instance group contains a single instance serving as the Hadoop NameNode. This instance generally does not need to be significantly powerful. The Core instance group contains one or more instance serving as Hadoop DataNodes. These instances simultaneously store data, host the HDFS, and provide processing power as Map and Reduce slots. The Core instance group can only be grown during an application's runtime, as shrinking it would imply potential loss of data. Generally these instances should be the most powerful machines with the most storage and memory. Finally, the Task instance group contains zero or more instances serving as Hadoop TaskTrackers. These instances only provide Map and Reduce slots, and should generally be computationally powerful VMs if present.

*__Note__: If you are developing in an IDE like Eclipse, an entire project can sometimes be easily exported into a Jar archive using the graphical interface.

APPENDIX C. USEFUL SCRIPTS

C.1 Amazon Web Services (AWS) Scripts

C.1.1 bootstrap-install-mahout.sh

```
#!/bin/bash

set -e

# DOWNLOAD MAHOUT
echo "cd ~"
cd ~
echo "wget https://s3.amazonaws.com/YOUR-BUCKET/mahout-distribution-0.6.tar.gz"
wget https://s3.amazonaws.com/YOUR-BUCKET/mahout-distribution-0.6.tar.gz
echo "gunzip mahout-distribution-0.6.tar.gz"
gunzip mahout-distribution-0.6.tar.gz
echo "tar xvf mahout-distribution-0.6.tar"
tar xvf mahout-distribution-0.6.tar

# INSTALL MAHOUT SYMLINK
echo "cd /usr/lib"
cd /usr/lib
echo "sudo ln -s /home/hadoop/mahout-distribution-0.6 mahout"
sudo ln -s /home/hadoop/mahout-distribution-0.6 mahout
echo "cd ~"
cd ~

# INSTALL MAHOUT LIBRARY ON PATH
echo "export PATH=$PATH:/usr/lib/mahout/bin"
export PATH=$PATH:/usr/lib/mahout/bin

# INSTALL MAHOUT JARS ON HADOOP_CLASSPATH
echo "export from /usr/lib/mahout/*.jar"
for lib in /usr/lib/mahout/*.jar
do
  export MAHOUT_CLASSPATH=$lib:$MAHOUT_CLASSPATH
done

echo "export from /usr/lib/mahout/lib/*.jar"
for lib in /usr/lib/mahout/lib/*.jar
do
  export MAHOUT_CLASSPATH=$lib:$MAHOUT_CLASSPATH
done

echo "export HADOOP_CLASSPATH=$MAHOUT_CLASSPATH:$HADOOP_CLASSPATH"
export HADOOP_CLASSPATH=$MAHOUT_CLASSPATH:$HADOOP_CLASSPATH

echo "echo "export PATH=$PATH:/usr/lib/mahout/bin" >> ~/.bashrc"
echo "export PATH=$PATH:/usr/lib/mahout/bin" >> ~/.bashrc
```

C.1.2 hierarchicalKMeans.sh

```
rm -r dump
mkdir dump
mkdir dump/level1
mkdir dump/level2
mkdir dump/level3
```

```
6
7  WORKTAR=$1
8  FILENAME=$(echo $WORKTAR | sed 's/.*\///g')
9  FOLDER=$(echo $FILENAME | sed 's/\..*//g')
10
11 wget $WORKTAR
12 tar xvf $FILENAME
13
14 hadoop fs -mkdir /user/hadoop/
15 hadoop fs -put $FOLDER .
16
17 # LEVEL 1
18 mahout canopy -i $FOLDER/vectors -o $FOLDER/canopy -t1 20 -t2 10 -xm mapreduce -ow -cl
19 mahout kmeans -i $FOLDER/vectors -o $FOLDER/level1 -c $FOLDER/canopy/clusters-0-final
       -x 5 -xm mapreduce -ow -cl
20 mahout clusterdump -s $FOLDER/level1/clusters-*-final/ -p $FOLDER/canopy/
       clusteredPoints -o dump/level1/root
21
22 # LEVEL 2
23 hadoop fs -mkdir $FOLDER/level2
24 mahout clusterpp -i $FOLDER/level1 -o $FOLDER/level2/data -xm sequential
25
26 rm -r data
27 hadoop fs -get $FOLDER/level2/data .
28 for x in `ls data | grep -v SUCCESS`; do
29   echo
30   mahout canopy -i $FOLDER/level2/data/$x -o $FOLDER/level2/canopy/$x -t1 10 -t2 5 -xm
       mapreduce -ow -cl
31   mahout kmeans -i $FOLDER/level2/data/$x -o $FOLDER/level2/$x -c $FOLDER/level2/
       canopy/$x/clusters-0-final -x 1 -xm mapreduce -ow -cl
32 done
33 rm -r data
34
35 # LEVEL 3
36 hadoop fs -mkdir $FOLDER/level3
37 rm -r level2
38 hadoop fs -get $FOLDER/level2 .
39 for x in `ls level2/ | grep -v data | grep -v canopy`; do
40   echo
41
42   mahout clusterdump -s $FOLDER/level2/$x/clusters-*-final/ -p $FOLDER/level2/canopy/
       $x/clusteredPoints -o dump/level2/$x
43   mahout clusterpp -i $FOLDER/level2/$x -o $FOLDER/level3/data/$x -xm sequential
44
45 done
46 rm -r level2
47
48 rm -r data
49 hadoop fs -get $FOLDER/level3/data .
50 for x in `ls data`; do
51   echo
52   for y in `ls data/$x | grep -v SUCCESS`; do
53
54     mahout canopy -i $FOLDER/level3/data/$x/$y -o $FOLDER/level3/canopy/$x-$y -t1 5 -
       t2 2 -xm mapreduce -ow -cl
55     mahout kmeans -i $FOLDER/level3/data/$x/$y -o $FOLDER/level3/$x-$y -c $FOLDER/
       level3/canopy/$x-$y/clusters-0-final -x 1 -xm mapreduce -ow -cl;
```

```
  done
done
rm -r data

rm -r level3
hadoop fs -get $FOLDER/level3/ .
for x in `ls level3/ | grep -v data | grep -v canopy`; do

  mahout clusterdump -s $FOLDER/level3/$x/clusters-*-final/ -p $FOLDER/level3/canopy/
    $x/clusteredPoints -o dump/level3/$x

done
rm -r level3
```

APPENDIX D. SUPPORTING ONLINE MATERIAL

D.1 EMR Developer's Guide

The EMR Developer's Guide is available for download below.
http://awsdocs.s3.amazonaws.com/ElasticMapReduce/latest/emr-dg.pdf

D.2 Source Code

The source code for this project, in its entirety, is available for download at the mirror below.
http://michel.rouly.net/projects/amalthea2013/source.tar

D.3 Code JAVADOC

The complete code documentation is available online at the mirror below.
http://michel.rouly.net/projects/amalthea2013/docs/javadoc