On the Performance of High Dimensional Data Clustering and Classification Algorithms

Kathleen Ericson and Shrideep Pallickara
Computer Science Department
Colorado State University
Fort Collins, CO USA
{ericson,shrideep}@cs.colostate.edu

Abstract

There is often a need to perform machine learning tasks on voluminous amounts of data. These tasks have application in fields such as pattern recognition, data mining, bioinformatics, and recommendation systems. Here we evaluate the performance of 4 clustering algorithms and 2 classification algorithms supported by Mahout within two different cloud runtimes, Hadoop and Granules. Our benchmarks use the same Mahout backend code, ensuring a fair comparison. The differences between these implementations stem from how the Hadoop and Granules runtimes (1) support and manage the lifecycle of individual computations, and (2) how they orchestrate exchange of data between different stages of the computational pipeline during successive iterations of the clustering algorithm. We include an analysis of our results for each of these algorithms in a distributed setting, as well as a discussion on measures for failure recovery.

Keywords- Machine Learning; Distributed Stream Processing; Hadoop; Mahout; Clustering; Classification; Granules

1 Background and Related Work

As the rate at which we generate data increases, we find a greater and greater need to handle voluminous amounts of data within traditional machine learning algorithms. As a general rule of thumb, the more examples you can provide to a machine learning algorithm, the better it will be able to perform. The ability to quickly and efficiently process large amounts of data is necessary in order to effectively scale learning algorithms to match the growth of data available. Here we explore both clustering and classification algorithms, within the realm of distributed processing.

To facilitate distributed processing one approach is to use the MapReduce [1] framework. Here, a large input dataset is sliced into smaller datasets, each of which is then operated upon by a different computation – these separate computations are the *Mappers*. The results of the processing are then fed into a *Reducer* (or a set of reducers) which will account for boundary conditions and combine results for further processing. For example, it is possible that a cluster center residing in one map may have points within its purview that is part of the input data to another map function. The outputs from the reducer are then fed into the appropriate mappers to begin the next round of processing.

Mahout [2] is a library that implements several clustering and classification algorithms which have been modified to fit the Map-Reduce [1] model. The Mahout implementations have been deployed within Apache Hadoop [3] – a MapReduce based cloud runtime. While Mahout has been designed to work specifically with Hadoop, there is nothing to preclude using the Mahout library within another processing system which supports the MapReduce paradigm.

Clustering algorithms are an *unsupervised* machine learning technique that facilitates the creation of *clusters*, which allow us to group similar items (also called observations) together so that these clusters are similar in some quantifiable sense. Clustering has broad applications in areas such as data mining [4], recommendation systems [5], pattern recognition [6], identification of abnormal cell clusters for cancer detections, and bioinformatics [7] among others. Clustering algorithms have certain unique characteristics. First, the algorithms often involve multiple *rounds* (also called *iterations*) of execution, where the output of the previous round is the input to the subsequent round. Second, the number of iterations of the algorithm is determined by the *convergence* characteristics of the algorithm. This convergence is generally based on distance measures (in n-dimensional space) and also the movement of cluster centers in successive iterations of the algorithm. To account for cases where convergence may not occur for a very large number of iterations, it is also possible to specify an upper bound on the number of iterations. Finally, the algorithm may operate on n-dimensional data and will cluster along these dimensions. Beyond the first iteration the progress of the clustering computation depends on (1) the state that it has built up in previous iteration (2) the initial set of data points that it holds, and (3) the adjustments to the cluster centers that it receives from the previous iteration.

As data volumes increase, it quickly becomes untenable to perform this clustering over a single machine. One challenge in implementing distributed clustering algorithms is that it is possible that an algorithm will get stuck in local optima, never finding the optimal solution. Attempting to converge on an optimal solution can be even more difficult when data is distributed, where no single node is fully aware of all data points.

Classification algorithms, on the other hand, are a *supervised* machine learning technique. Where unsupervised learning techniques are not aware of the correct labels for data and need to repeatedly iterate through the inputs in order to refine observations; supervised machine learning algorithms are provided the correct answer with the training data, and only loop through the inputs once to create a statistical model. This model is then used to predict, or *classify* incoming data by determining the likelihood of the new, unseen data belonging to a class learned in the training phase. Classifiers can be used in BCI applications [8], bioinformatics [9], and spam filtering.

The task of training a classifier becomes more monumental as the number of training sets increases. Each input needs to be read in, processed, and then used to modify the prediction model. If a single node attempted to perform this task sequentially, the training time would quickly become unfeasible. Classification provides a challenge when moving to a distributed processing environment. Several stages of model creation require all data gathered so far be collected to a single node for further processing. While this is a natural fit for a reduce stage, it also means a mandated bottleneck in processing.

We have compared the efficiency of orchestrating the distributed executions of these machine learning algorithms within our distributed stream processing system, Granules [10, 11]. Our benchmarks compare the *same* Mahout code running inside Granules and Hadoop. We chose Hadoop and Granules for this comparison as they are representative of file processing and stream processing systems, respectively. As both support the MapReduce framework, we can use the Mahout codebase without modifications in either runtime. With the machine learning algorithms identical and unmodified, the only differences in computation speed should be a result of the lifecycle support for individual computations and the underlying communications framework.

We have implemented 4 clustering algorithms within Granules: K-means [12], Fuzzy k-means [13], Dirichlet [14] and Latent Dirichlet Allocation [15]. These algorithms are representative of two broad classes of clustering algorithms: (1) *discriminative*, where we are making decisions if a point belongs in a predefined set of clusters (k-means, fuzzy k-means) and (2) *generative*, where a model is tweaked to fit the data and we can even generate the data the model has been fit to using the model parameters (Dirichlet and Latent Dirichlet Allocation). We believe that these algorithms are also a good example of performance improvements that can be accrued by moving away from execute-once and stateless semantics in traditional MapReduce implementations such as Hadoop.

We also implemented 2 classification algorithms: Naïve Bayes and Complementary Bayes [16]. While Mahout supports other classification algorithms, such as Stochastic Gradient Descent (SGD) [2, 17], there is only a sequential implementation in Mahout since it scales linearly as the size of the training set grows. SGD is not a good choice for extremely large datasets, but it can be trained incrementally meaning that a very large dataset may be worked through piece-meal.

Naive Bayes and Complementary Bayes work well with textual classification tasks, unlike SGD, such as filtering emails or sorting documents. These approaches create slightly different models, whose performance can vary based on the dataset. In Mahout, once the data has been preprocessed it can be used in both Bayes and Complementary Bayes without further modifications. This means that if a model trained with Naïve Bayes isn't performing adequately, it is a simple task to switch to a Complementary Bayes model.

Distributed machine learning has been a big topic for several years, not only on multicore machines [18], but also GPUs [19]. While many works mention machine learning applications running in a distributed environment [1], [20] they do not go into depth about the details of their implementations, and have not made the libraries available to the public.

Mahout offers access to many varied machine learning algorithms, but it is geared towards developing enhanced recommenders [21] which can use multiple different clustering and classification algorithms to help generate recommendations. The Twister Iterative Map-Reduce runtime [22], on the other hand, has been developed to help biologists leverage the many parallel algorithms available for bioinformatics research. It allows biologists to specify a high-level workflow without needing a strong background in high performance computing.

1.1 Paper Extensions

From the initial publication of *On the Performance of Distributed Data Clustering Algorithms in File and Streaming Processing Systems* [23], we have added several extensions for this special edition issue. We have switched to the newer Mahout 0.5 and Hadoop 1.0.0 versions, both of which have been released since the original publication. All of our original benchmarks were redone in this particular setting. The clustering section has been expanded to cover the topic of fault-tolerance, which was not previously supported in our Granules implementation. We have shown that even with a basic fault-tolerance implementation – with current clusters written to file after every implementation, our Granules-based implementation can still outperform the original Hadoop implementation. We have further explored methods of reducing strain placed on the hard drives of our HDFS cluster by saving state to disk on a staggered timescale. Additionally, we have extended our evaluation to cover Mahout's supported distributed classification algorithms: Naïve Bayes and Complementary Bayes.

1.2 Paper Organization

The remainder of this paper is ordered as follows: In section **Error! Reference source not found.**, we provide a quick overview of Granules and Hadoop, the two cloud runtimes used in this work. In section 3 we explore clustering in a distributed setting, covering our experimental setup, dataset and actual experiments with clustering algorithms as well as an overview of fault-tolerance for clustering. We then move on to discuss classification as supported in Mahout in section 4, as well as outline our experimental setup and results. We then report our conclusions and describe future work in section 5.

2 Core Technology

2.1 Hadoop

Hadoop [3] is a Java-based cloud computing runtime which supports the Map-Reduce [1] paradigm. Hadoop has execute-once semantics, meaning that with iterative tasks all state information needs to be written to file and then read back in for every step of the computation.

Hadoop is open-source, and widely used for Map-Reduce computations. Mahout [2] has been built to run on top of Hadoop and the Hadoop Distributed File System (HDFS) [24]. HDFS is an implementation of the Google File System where a large file is broken into fixed size chunks each of which is then replicated. While processing the data, the runtime pushes computations to the machines where these blocks are hosted to maximize data locality during processing for faster executions.

When Hadoop is running with HDFS, Hadoop can take advantage of data locality and push computations to the data they are supposed to operate on, cutting down on the networking overhead which may be incurred when reading from HDFS. This is not supported in Granules, which may give the Hadoop based implementation an edge in processing overheads.

2.2 Granules

Granules [10, 11] is a lightweight distributed stream processing system (DSPS) and is designed to orchestrate a large number of computations on a set of available machines. The runtime is designed specifically to support processing of data streams. Granules supports two of the most dominant models for cloud computing: MapReduce and dataflow graphs [25]. In Granules individual computations have a finite state machine associated with them. Computations change state depending on the availability of data on any of their input datasets or as a result of external triggers. When the processing is complete, computations become dormant awaiting data on any of their input datasets.

In Granules, computations specify a scheduling strategy, which in turn govern their lifetimes. Computations specify their scheduling strategy along three dimensions: counts, data driven and periodicity. The counts axis specifies limits on the number of times a computation task needs to be executed. The data driven axis specifies that a computation task needs to be scheduled for execution whenever data is available on any one of its constituent datasets, which could be streams or files. The periodicity axis specifies that computations be scheduled for execution at predefined intervals. One can also specify a custom scheduling strategy that is a combination along these three dimensions; for example, limit a computation to be executed 500 times either when data is available or at regular intervals. A computation can change its scheduling strategy during execution, and Granules enforces the newly established scheduling strategy during the next round of execution. Computations in Granules can build state over successive rounds of execution, meaning we can break away from execute-once semantics. Though the typical CPU burst time for computations during a given execution is short (seconds to a few minutes), these computations can be long-running with computations toggling between activations and dormancy. Domains that Granules has been deployed include handwriting recognition [26], Brain Computer Interfaces [27], and epidemiological simulations.

3 Clustering in a Distributed Setting

Clustering is a machine learning algorithm in which the program is responsible for discovering commonalities across voluminous datasets, and finding appropriate groups to place, or *cluster*, all incoming data. Clustering is a very useful tool in unsupervised data mining and can help uncover relationships between data points which are not otherwise noticed. The classic example of this is the retailer who noticed that diapers and beer are often bought together. In our examples, we are not working with retailer information, but instead working to classify news articles under various topics.

An important aspect of determining clusters is defining how distance will be measured. The distance measure used may bias results, so it is important to try clustering with several different distance measures to determine the best approach for a given dataset. Mahout includes definitions of multiple types of distance measurements and also allows users to specify custom distance measures. In this paper we use the Euclidean distance measure to determine distances between points and cluster centers across all our tests as done in [2].

3.1 Clustering using Hadoop

Hadoop computations have execute-once semantics and are stateless. Clustering computations expressed in Hadoop need to account for these execute-once and statelessness constraints. At the end of an iteration, every computation must store the state such that the subsequent iterations can retrieve this information as part of its initialization. A new computation must be launched for each round of execution, and this computation must reconstruct the state saved by the previous iteration from disk, typically using HDFS. Often, the original data splits also need to be loaded into the computation. In the case of clustering algorithms which often have multiple, successive rounds of execution this can lead to overheads and increased execution times.

3.2 Clustering Using Granules

Depending on their specified scheduling strategy Granules computations stay dormant when conditions for their execution have not been satisfied. Computations are activated from dormancy once data is available on one or more of their input datasets. The activation overhead for computations once data is available for processing is in the order of 700 microseconds. Computations in Granules can have multiple rounds of execution and the runtime manages their lifecycles. Individual computations are able to retain state across these multiple rounds of execution.

Granules allows computations to enter a dormant state between rounds of execution. Due to this ability, running an iterative Map-Reduce application – such as the machine learning algorithms within the Mahout library – within Granules should be more efficient than in a runtime that requires all data to be written to and read from disk between rounds of execution.

In our setting involving Granules, each mapper works with a subset of the original dataset. The mapper is then responsible for clustering these points throughout the lifetime of the algorithm. For every iteration, the mapper loops through the points it is responsible for and aggregates all cluster information before sending this data on to the reducer. The reducer is activated when it receives outputs from individual mappers. Once the reducer has received inputs from all mappers, it is able to determine global adjustments to the clusters and send this information back to the mappers to start the next round of clustering. Implementing these distributed clustering algorithms as Granules computations have a few advantages that could translate into faster execution times. Computations can gain from:

(1) Not having to reinitialize state from the disk
(2) Streaming results between intermediate stages of a computation pipeline rather than having to perform disk I/O.
(3) Fast activation of dormant computations as data streams become available.

3.3 Code Modifications

To adapt Mahout code to run within the Granules runtime, we needed to modify the drivers for the clustering algorithm as well as some semantic changes to the map and reduce code. The actual clustering algorithms were not touched at all, meaning we will be seeing a fair comparison of execution times given different communications substrates. It is important to keep in mind that the bulk of our code modification, in the drivers, would need to be modified for something as small as a change in the type of data being clustered.

The I/O format of the code is similarly untouched. In our Granules runs, we kept the HDFS backend for initial loading of points and clusters – while we cannot take advantage of rack-locality in Granules, this did enable us to ensure the runtime overheads are comparable. The real changes were made to slightly tweak the map and reduce code, to fit the different programming paradigm of Granules. Hadoop demands run-once semantics – the map and reduce code is called for every line of data that is read by Hadoop. In Granules, computations can retain state during successive rounds of execution and multiple lines of data can be processed at a time.

Both Hadoop and Granules use different strategies to move data between different stages of a computation pipeline. In the case of Hadoop this involves disk I/O and polling to determine if the data is available within HDFS. In the case of Granules, data is streamed between the different stages and computations are activated from their dormancy when such data is available.

3.4 Experimental Setup for Clustering

Both the Hadoop and Granules-based implementations initially read data from an HDFS cluster. For our tests, we are using Hadoop version 1.0.0 and Mahout version 0.5. All tests are run on 2.4 GHz quad-core machines running Fedora 14 with 12GB RAM and a gigabit network connection. Each distributed run contains 25 mappers and a reducer (Mahout clustering algorithms are set to run with a single reducer, eliminating boundary conflicts). To properly compare Mahout performance across runtimes, when testing with Granules, the Granules resources are running on the HDFS worker machines.

Among the 25 machines, one was also responsible for acting as a NameNode, and TaskTracker, while five were acting as Brokers (for the Granules runs). For both approaches, the Mahout operation was submitted from a machine outside the cluster.

3.4.1 Clustering Setup

For each clustering method we analyze: k-means, fuzzy k-means, dirichlet, and latent dirichlet allocation, we first use Mahout to generate a random set of starting clusters. We use the same set of starting clusters when contrasting the performance of Hadoop and Granules. Canopy clustering [28] is a technique used to jumpstart clustering algorithms, and usually only runs for a limited number of iterations. Due to the limited iterations of computations involved in canopy clustering, we will not be analyzing canopy clustering in this work.

3.5 Dataset

For the clustering example, we used the Reuters-21578 text categorization collection data set [29]. This data set contains 21,578 documents which appeared on the Reuters newswire in 1987. This dataset was generated following the ACM SIGIR '96 conference when it was decided the Reuters-22173 dataset should be cleaned and standardized in order to achieve more comparable results across text categorization studies. We processed the dataset to convert it to a format that Mahout can handle, based on the guidelines in [2]. This produced vectors of normalized bigrams from the input data with *term frequency – inverse document frequency* (TF-IDF) weighting. The TF-IDF weighting helps to lower the importance of words which occur often across all documents, such as "the," "he," "she," or "a."

We are clustering across all news documents in the dataset, so we have 21,578 points to cluster. There are 95,000+ dimensions of bigrams, or unique pairs of words. Since no single document contains all possible bigrams, these are stored internally using Mahout's `SparseVector` format.

3.6 K-Means Clustering

Mahout supports several different clustering algorithms. We initially start with k-means clustering, a clustering algorithm where the user estimates how many clusters are required (*k*) to adequately group all data. The algorithm then runs with this number kept constant – no clusters are added or removed during the computation. This algorithm was first introduced as a technique for pulse-code modulation [12].

K-means is the most basic clustering algorithm supported by Mahout, and operates on the principle that all data can be separated into distinct clusters. K-means requires the user to specify a *k* value, and the output can vary drastically based on not only the number of clusters chosen, but the initial starting points of all clusters. With respect to our dataset, when looking for very broad topic categories, a small value of *k* would be chosen (10-20). When looking for very small and finely honed categories, we would need to drastically increase k (1,000).

K-means clustering is a good choice when it is believed that all points belong to distinct groups. It can also a good choice when initially approaching a new dataset. K-means runs quickly, and can find large distinctions within data.

In the Hadoop implementation, the input data is separated into a number of files. This data consists of the points which will be clustered. Each map process is responsible for looping through its assigned input file(s), and assigning the points to the nearest cluster. The mappers output a file which contains a cluster ID connected to the point which is assigned to it. The reducer will read in each file generated by the mappers, and move through the list of clusters assigning each point to it. Once this is complete, the reducer then computes new cluster centers, and generates a file containing the new clusters. The entire process then repeats: the mappers read in the new cluster data, as well as the points and begin processing again.

Our implementation in Granules follows the original Hadoop implementation. Each mapper node is responsible for loading a set of points into memory, and is responsible for clustering those points. In each iteration, a set of current clusters is made available to all mappers. Once each mapper has finished clustering their points, a set of `ClusterObversations` for each cluster is sent to the reducer. The reducer combines `ClusterObservations` from each mapper, and uses this information to update the cluster centers. This modified set of clusters is then sent to each mapper for the next iteration.

One major difference between the Hadoop and Granules versions is where the completion point is computed. The Hadoop version will calculate the maximum number of iterations in the mappers as well as in the reducer, while in Granules the mappers are unaware of the overall point in execution and the reducer is responsible for keeping track of rounds of execution.

3.6.1 K-Means Runtime Analysis

K-means is the simplest clustering algorithm we have benchmarked. In the Hadoop implementation, a mapper is responsible for loading the current set of clusters from disk each iteration. Once the clusters are in memory, the mapper then reads through the list of points assigned to it one at a time, and identifies which cluster the point belongs to. Once a point has been assigned to a cluster, it is written out to file. The overall cost of the Hadoop map operation is $CR_D + NCR_DW_D$ where C is the number of clusters, N is the number of points a given mapper is responsible for clustering and R_D and W_D are read and write times to disk. It is important to note that these values include seek time as well as the time to actually read and write the data.

The Hadoop reducer will first read in the current set of clusters from file then read in the outputs from all the mappers. As the reducer reads in data, it modifies the clusters in constant time and writes out a modified cluster once it has finished processing all points assigned to the cluster. The overall runtime of the reducer is $CR_D + MNR_DW_D$, where M is the number of mappers in the system.

In the Granules mapper, we can take advantage of state retention by keeping the points to be clustered in memory, so we only need to read in the new states for every round of execution. We can also use state to make sure we send less data to the Reducer, helping cut down on the amount of work the reducer needs to perform. The Granules mapper runtime is: $CR_S + NC + CW_S$ where R_S refers to the cost of reading streaming data over a socket while W_S refers to writing data to a socket. It is important to note that this includes the cost of the streaming substrate overhead.

The Granules reducer needs to read in the input from all the mappers, and send out the newly computed clusters for the next round of computations. The running time of this operation is $MCR_S + CW_S$.

Comparing the mapper runtimes for the Granules and Hadoop implementations, it is clear that both can be boiled down to an $O(NC)$ operation, and the major difference between them is simply the constants around that function. The reducers have a different overhead by an order of N, yet we're not seeing a commensurate speedup in our benchmarks. This is because the majority of the computation is spent in the mappers, while the role of the reducers is relatively small in the overall computation.

3.6.2 K-means Clustering Results

In both Hadoop and Granules, we ran 20 rounds of k-means on 88 clusters for 100 iterations. Both implementations used the same initial set of starting clusters. The results of these tests are displayed in Figure 1. and summarized in TABLE I.

Another observation is the difference in standard deviation of the running times, where Granules deviates by about 8 minutes, Hadoop varies by only a bit over 2 minutes. We are obviously seeing some networking congestion when relying on the Granules approach.

TABLE I. K-MEANS CLUSTERING IN SECONDS

	Mean	Min	Max	SD
Granules	1214.895	559.24	1938.766	484.175
Hadoop	5122.81	5002.732	5683.786	137.581

From these results, we can clearly see that the Granules implementation can outperform the Hadoop implementation by decreasing the amount of disk accesses necessary to complete the operation. Granules can even outperform the Hadoop version when both are running on top of HDFS, where the Hadoop workers can take advantage of data locality to speed up access times.

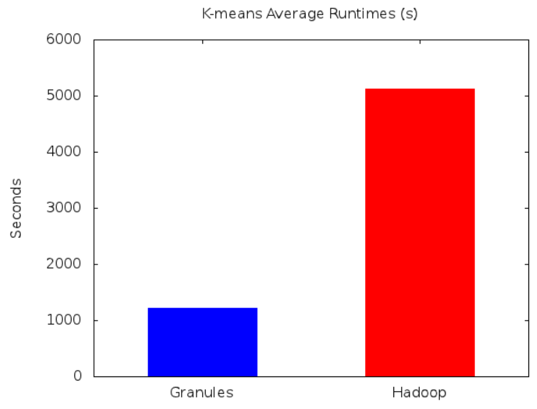

Figure 1. K-means Average runtime in Seconds

3.7 Fuzzy K-Means

Fuzzy K-Means [13] operates on a similar principle as k-means: The user chooses an initial set of *k* clusters, and allows the algorithm to run and adjust their centers as points are assigned to them. Fuzzy k-means allows an extra degree of freedom by allowing a point to belong to more than one cluster.

Using our dataset as an example, k-means is able to find the broad and overarching topics and group the articles accordingly; however, k-means cannot handle data points that span multiple topics. For example, a news article may discuss oil prices in the Middle East. With k-means, this article can either be clustered with articles about the Middle East, or articles discussing the prices of raw materials, but not both. Fuzzy k-means would allow the article to be associated with both topics, thus revealing a link between data that k-means could not show. Not only will fuzzy k-means show this overlapping of topics, it will also describe the degree to which the article is related to each topic.

Fuzzy k-means operates in roughly the same manner as k-means, with the modification that instead of each point belonging to a single cluster, each point is assigned a probability of belonging to every cluster. After this step, the reducer then goes through each probability, and adjusts cluster centers with respect to those points with the highest probability of belonging to the cluster.

3.7.1 Fuzzy K-Means Runtime Analysis

Fuzzy k-means is a slightly more complex clustering algorithm than k-means, and requires much more data to be sent between the Map and Reduce phases. As fuzzy k-means computes the probability that each point belongs to every node, the mapper now will pass *NC* information to the reducer instead of just *N*. The overall running time of the Hadoop mapper is $R_DC + R_DNCW_D$. Again, R_D and W_D refer to reading from and writing to disk – including seek time, *N* is the number of points a mapper is responsible for clustering, and *C* is the number of clusters.

The runtime for the fuzzy k-means Hadoop reducer is very similar to the k-means version – it simply has to handle more data. The runtime is: $MNCR_D + CW_D$, where *M* is the number of mappers in the system.

The Granules fuzzy k-means mapper has an overhead of $R_SC + NC + CW_S$, where R_S and W_S are the times to read stream data from and write it to sockets – including the streaming overheads. A major difference between this and the Hadoop mapper is that Granules can retain state information and can aggregate outputs, so it only needs to send *C* to the reducer, instead of *NC*.

The Granules reducer takes advantage of the partial aggregation done by the mappers, and needs to read in far less data than its Hadoop counterpart (by a factor of *N*). The runtime of the Granules reducer is: $MCR_S + CW_S$, again with M being the total number of reducers.

Both the Granules and Hadoop approaches are bounded by the *NC* computation to compute the probability of each point belonging to every cluster, essentially bounding the runtime at *O(N)*. While we do see a big difference in the reducer behavior, it is another computation where the work done by the reducers is insignificant when compared to the work performed by the mappers. Despite the Granules implementation having a quicker reducer runtime by a factor of *N*, the overall runtimes are still very similar.

3.7.2 Fuzzy K-Means Results

Since the fuzzy k-means algorithm allows points to span multiple clusters it is very long running. We ran 20 rounds of fuzzy k-means for 25 iterations before halting the computations. We set the number of clusters to 44 and used the same set of initial clusters was used for both Hadoop and Granules versions. In 0the mean execution times for Granules and Hadoop can be seen. Hadoop is taking just a bit more time than Granules to finish processing the data. More detailed results are shown below in TABLE II. We can see that on average, the Granules implementation is finishing over 700 seconds (almost 12 minutes) sooner than the Hadoop implementation.

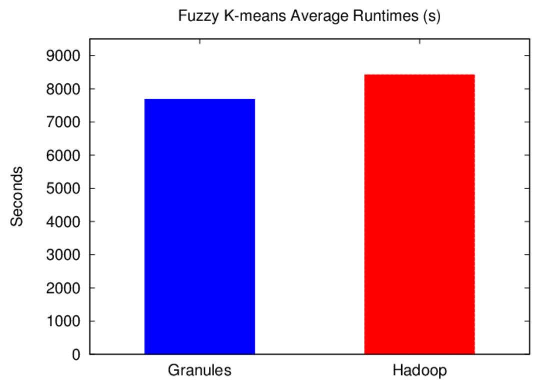

Figure 2. Fuzzy k-means Average Runtime in Seconds

While we do not see the same increase in speed we get with k-means, the Granules implementation still manages to outperform Hadoop implementation by a small margin simply by cutting out the need for repeated reads/writes from disk. From analyzing the source code, it appears that fuzzy k-means has a far higher ratio of CPU-to-I/O bound processing than the other clustering algorithms we discuss here, accounting for the difference in speedup. In order to isolate the bottleneck in the Granules implementation, we timed each step of the algorithm. Through this method, we found that the biggest bottleneck in our system was in the reducer. Each round of execution, several seconds were lost with output from mappers waiting in the reducer's queue while it was processing previous inputs. These slight delays add up over each iteration of processing, leading to a smaller increase in speed than we hoped for.

TABLE II. FUZZY K-MEANS CLUSTERING IN SECONDS

	Mean	**Min**	**Max**	**SD**
Granules	7685.74	7676.47	7695.79	5.567
Hadoop	8423.78	8414.22	8431.15	5.812

3.8 Dirichlet Clustering

Dirichlet clustering [14] differs drastically from k-means. Most notably, there is no *k*. Dirichlet clustering may add and remove clusters as it deems necessary, and additionally can support different shapes of models. K-means and fuzzy k-means both assume that all clusters have normal distributions around a central point (in the case of 2-dimensional data it is circular). They cannot handle a distribution where clusters match a different model. Mahout currently supports models such as `GaussianCluster`, `NormalModel`, and `SampledNormalDistribution`; also allowing the user to define more models as needed.

Because of its complexity, Dirichlet clustering can take far longer to run than k-means or fuzzy k-means. Due to the many iterations it may go through, Mahout allows the user to specify the number of iterations to move through before writing cluster information to file – though this is only available for local in-memory runs.

Dirichlet clustering is a good initial clustering algorithm as it can help to determine an appropriate k to give the faster running k-means algorithms, or even help show why k-means may be having problems e.g.: if the data does not fit the normal distribution model that k-means expects, Dirichlet should be able to cluster data where either k-means or fuzzy k-means fails.

3.8.1 Dirichlet Clustering Runtime Analysis

As mentioned above, Dirichlet clustering follows a different paradigm than k-means or fuzzy k-means. For Dirichlet, the number of models D is a parameter. This algorithm also requires state to be passed between each iteration. In the Hadoop Mapper the state is first read in, then the algorithm is run as the points to cluster, N, are read in. This leaves the overall mapper runtime at $DR_D + R_DNDW_D$, where R_D and W_D are read and write to disk respectively, and N is the number of points to cluster.

Each mapper writes data for every point, for every model, meaning that the Hadoop reducer then has to read in all this information from every mapper, as well as load state. Once the reducer has finished processing all data for a model, it then writes the model information to disk to be read in as state information for the next round of computation. The overall runtime of the reducer is $R_DD + R_DMND + W_DD$, where M is the number of mappers in the setup.

The Granules mapper follows the same approach as the Hadoop mapper, but manages to cut down on the overheads slightly by aggregating data, and sending a smaller amount of data to the Reducer. This also helps to cut down on the reducers' runtime. For the Granules mapper we see an overhead of $R_SD + ND + DW_S$, where R_S and W_S are read and write overheads for sending data to sockets, including the streaming overhead. Because of the partial aggregation of data at the mappers, the value associated with the write is only D instead of ND. Additionally, the Granules mapper does not need to read in the points, completely removing a read operation.

The Granules reducer takes advantage of the partial aggregation by having a much reduced read-in time: $R_SMD + DW_S$. Additionally, the Granules reducer has no need to read in the current state, since it saves the state generated in the previous iteration. This significantly cuts down the I/O time that Granules needs for this algorithm.

Both the map and reduce in the Hadoop implementation are bounded by *O(ND)*. In Granules, the Map has *O(ND)*, but the reducer only has *O(MD)*, and *M* is usually several orders of magnitude smaller than *N*. As we see in the next section, however, we are not seeing a speedup in Granules of several orders of magnitude. This is because the work done in the map and reduce portions of the algorithm are not balanced – the mapper does far more work than the reducer, so lowering the runtime of the reducer drastically does not have a great effect on overall runtime.

3.8.2 Dirichlet Clustering Results

We ran 20 iterations of Dirichlet clustering which ran for 40 iterations each. The results of these tests in both Granules and Hadoop are displayed below in TABLE III. Granules runs Dirichlet clustering to completion about five times faster than Hadoop, which is also visualized in Figure 3.

Dirichlet clustering relies heavily on state, and does not require as much processing of data points as fuzzy k-means. From these results, it seems clear that the majority of the processing time Hadoop spends is loading the state from file every step.

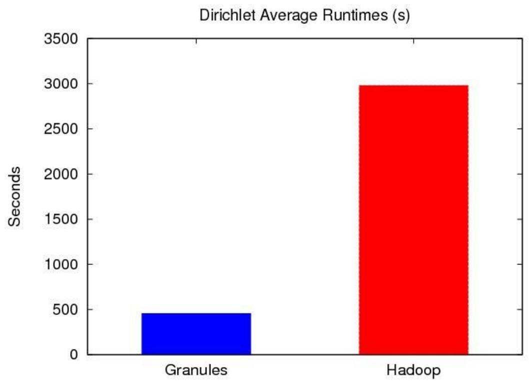

Figure 3. Dirichlet Average Runtime in Seconds

TABLE III. DIRICHLET CLUSTERING IN SECONDS

	Mean	Min	Max	SD
Granules	456.78	437.61	481.64	10.474
Hadoop	31933.61	31826.97	32332.79	131.5412

3.9 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) [15] is a clustering method similar to Dirichlet clustering. It is a generative model, so it starts off with a known model, and tweaks parameters to fit the model to the data. LDA can cluster words into "topics" by defining all documents as a mixture of all topics with a given probability. Much like k-means, LDA needs to be given a *k*, which identifies the number of topics in the dataset. The LDA classifier then attempts to discern the separate topics, and cluster each document into the appropriate topic. The algorithm reads through every word in every document, and calculates the probability that each word belongs to a topic. Based on the number of words in the document belonging to each topic, the overall topic of the document can be determined. LDA runs until the maximum number of iterations have been reached, or once the model has stabilized i.e. the amount of change between iterations has fallen below a given threshold.

Where algorithms such as k-means are very adept at grouping data with patterns not always apparent to humans, LDA can achieve results very similar to what would happen if we asked a human to cluster documents by topic [2]. The cost of this is that the algorithm takes many iterations to reach that level. LDA allows the process to be sped up by modifying a number of parameters which should help to cut down on the number of necessary iterations, such as automatically detecting stopwords and removing them from future calculations. LDA is a good clustering algorithm when one is looking for clustering that is human-understandable.

3.9.1 Latent Dirichlet Allocation Runtime Analysis

LDA runtimes depend on the number of topics to be clustered by (T), as well as the dimensionality of the data ($|P|$). LDA analyzes the number of times given bigrams appear in a document to determine the probability that the document belongs to a given group. This matrix of probabilities defining the relationship between all bigrams and each topic is passed between rounds of execution as part of the state information. This means that the mappers need to load a $T|P|$ size array into memory before every run. This array is changed by the reducer between every round of execution, so even the Granules approach incurs this cost.

After loading the state, the mapper then creates an inference for every point to be clustered – this involves looping through the dimensions of the point to be clustered, and assigning weights based off of the state information gathered in the first step. The Hadoop Mapper performs this step as it reads in points, and writes out information as soon as it has calculated adjusted probabilities for every topic. This results in a runtime of $R_DT|P| + R_DN|P|TW_D$, where R_D and W_D are read and write to disk (including seek time), T is the number of topics, N the nodes to be clustered, and $|P|$ the dimensionality of every point in N.

The Hadoop reducer has a far simpler task than the mappers, it simply needs to read in data output by the mappers and aggregate probabilities. The aggregated probabilities are then read in as the state table by the mappers in the next iteration. The Hadoop reduce runtime is $NMT|P|R_D + T|P|W_D$, with M being the number of mappers.

As mentioned above, the Granules mappers also need to load in state at the beginning, so it does not gain much improvement over the Hadoop implementation there. Granules can improve on the Hadoop implementation slightly, however, by again performing partial aggregation at the mapper. Instead of pushing out output immediately, the Granules mapper can hold the information in memory until the algorithm has completed and only needs to write $T|P|$ data to the reducer instead of $NT|P|$ data. The overall runtime of the Granules mapper is $T|P|R_S + N|P|T + T|P|W_S$, with R_S and W_S being overheads for reading and writing data to sockets, including the streaming overhead.

The Granules reducer can again take advantage of the decreased size of input and has a runtime of $MT|P|R_S + T|P|W_S$. While this is not as great of an increase as we saw with Dirichlet clustering, it still allows the Granules implementation to gain an edge over the original Hadoop runtimes.

The Hadoop mapper and reducer are both essentially bound by $O(NT|P|)$. The Granules mapper has almost the exact same runtime, but the Granules reducer is only bound by $O(T|P|)$. Again, this looks like it should lead to a much larger margin in performance between the Hadoop and Granules implementations, but the disparity between workloads holds true for LDA as well: even if we speed up the reducer, the mapper is still slowing us down too much for it to be noticeable.

3.9.2 Latent Dirichlet Allocation Results

In our tests, we ran LDA for 40 iterations clustering into 10 topics. While this was not enough iterations to allow the model to converge, this is enough iterations to give us a good idea of how the algorithm runs. We ran the full 40 iterations with both Granules and Hadoop versions, and compared the running time of each below in TABLE IV.

TABLE IV. LDA CLUSTERING IN SECONDS

	Mean	**Min**	**Max**	**SD**
Granules	1438.656	1402.428	1445.735	6.956
Hadoop	2266.084	2212.842	2294.902	25.081

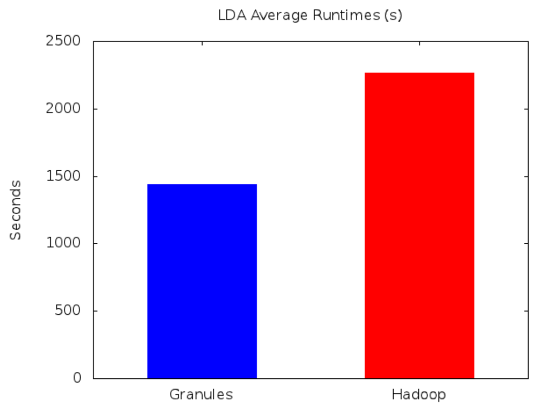

Figure 4. LDA Average Runtime in Seconds

Figure 4. shows a direct comparison of mean execution times of LDA for Granules and Hadoop. On average, Granules finished about 4 minutes earlier than the Hadoop implementation. Again, this seems to be a direct result of Granules' ability to stream data between stages, instead of needing to write to disk between every step.

Interestingly, the difference between standard deviations is again very high in our experiments with LDA. LDA requires a mixture of overhead to read in state, as well as significant processing time needed to build probability tables once the state has been read in. It is interesting to see that it has a very similar profile to fuzzy k-means, the other algorithm we examine with a heavy CPU processing load. Additionally, both feature relatively close execution times with a large difference in standard deviation.

3.10 Fault Tolerant Clustering

With the Hadoop based implementation of Mahout, fault-tolerance is obtained automatically with HDFS. After every iteration of clustering, the current clusters are written out to HDFS making it possible to recover from failure by simply starting from the last completed iteration.

In the previous sections, we showed how switching Mahout from Hadoop which has a run-once paradigm to Granules which allows state to be built up across iterations can lead to a processing speedup. While this speedup was less drastic in CPU bound computations, such as fuzzy k-means clustering, even a small speedup can become drastic as the number of iterations increases.

In the previous sections, we have shown that we can complete clustering more quickly, but this speedup was gained at the cost of fault-tolerance. Instead of performing a write and then read from disk after every iteration, we are passing this information across the network, and keeping needed information in memory. In the event of a failure, this approach loses all information, requiring the computation to be restarted from the beginning.

In this section we look at methods of reintroducing fault tolerance to our solution. While Granules does not yet support a method for automatically detecting and recovering from failures, we are able to take an approach similar to the original implementation and make use of HDFS as a fault-tolerant storage resource.

3.10.1 Adding Fault Tolerance to Granules Clustering

As an initial test of fault tolerance in Granules, we first worked with a naive checkpointing scheme. In this scheme, the set of current clusters is written to HDFS after every iteration. Essentially, we are ensuring fault tolerance levels equivalent to what is found in the original implementation, where we can recover from total failure by simply picking up from the last completed iteration of processing.

At first glance it appears we will be losing the processing speedups gained by switching from a Hadoop backend to Granules, but with Granules we can interleave writes with processing. For example, after the reducer finishes calculating a new set of clusters, it can immediately send on these new clusters to the mappers for the next phase of computations, and then orchestrate the write to HDFS – performing that operation when the reducer would otherwise be in a dormant state, waiting for the mappers to complete processing.

We reran all our base clustering algorithms with this naïve checkpointing scheme, and saw no difference in clustering overheads. This means we can recover from any failure to continue processing from the last iteration of clustering. The one additional overhead we incur is that of human intervention. Unlike Hadoop, Granules does not currently support automatic detection and recovery of failures. A user would be required to manually restart the cluster after a failure occurred. One upside of this approach, however, is that we can use this restart capability to pick up a completed clustering result and run the algorithm for more iterations to see if we can achieve better results.

3.10.2 Optimizing Checkpointing in Granules Clustering

From our previous tests, there is no loss of performance from even a naïve implementation of fault-tolerance. One disadvantage of this approach, however, is the stress placed on the hard drives in our HDFS cluster. While this is no greater stress than we see when using the original Mahout code, we should be able to do better.

In order to reduce stress on the cluster, we can introduce a new checkpointing scheme which balances the amount of writes we enforce with recovery time. We can balance the time to load an algorithm with the time it takes to complete a single iteration of clustering. From our previous benchmarks, we are able to calculate the average time it takes to run a single iteration of each algorithm. The optimal checkpointing scheme is one where we only checkpoint when the cost to recover from the last checkpoint outweighs the cost of loading a checkpoint.

4 Classification in a Distributed Setting

Unlike clustering techniques, classification is a *supervised* machine learning algorithm. This means there is a known, correct output which is provided to the cluster with the training data. Through the training process the algorithm builds a statistical model to predict which class a sample belongs to.

Since classification models are built as the training data is processed, it does not make sense to use an iterative approach such as we saw in the clustering algorithms. If we attempt to rerun training sets, we increase the chances of *overtraining*, where the classifier becomes unable to handle new inputs.

Classification algorithms generally require a large number of training inputs to accurately classify data, leading to a bottleneck in processing. Using the MapReduce framework, we can overcome this bottleneck by processing inputs in parallel.

4.1 Classification in Mahout

Since training is performed after reading in the datasets once, we do not expect to see a drastic difference in runtimes as we move from Hadoop's run-once paradigm to a Granules implementation which can build state. The only space for improvement is the ability to build state between inputs – Hadoop only maintains information about the current line of input, not any previously seen inputs. In order to gather statistical information, such as how often the word "apple" has been seen across all inputs, a Hadoop mapper would need to output "apple 1" every time it sees the word and then rely on a reducer to add together all the ones. This approach creates extra files in HDFS, leading to a potential strain on hard drives in the cluster.

4.2 Classification with Granules

In the Granules implementation, we followed the same outline as the Hadoop-based implementation, in order to keep the comparison as fair as possible. By streaming outputs between the classification steps, we are taking some load off of the HDFS cluster. Due to the speed with which we can train a Bayesian classifier, fault-tolerance in the middle of the training process is less of a priority then we saw in clustering.

4.3 Code Changes

In the Hadoop based implementation, there are 4 full map and reduce jobs needed to build a statistical model of the input data. This is due to Hadoop's inability to maintain state – it is difficult to perform a task such as summing up the number of times a word is seen across all test cases, requiring a chain of map and reduce tasks. Such a task is much simpler in the Granules framework, where each mapper maintains memory across more than one line of input at a time. Because of this, our processing pipeline is slightly shorter than the original Hadoop-based pipeline. Instead of 4 map-reduce tasks, we have two map-reduce pairs, and two combiner stages. These shortcuts were gained simply by being able to aggregate data in a single stage.

4.4 Experimental Setup for Classification

For our classification tests, we are using the same setup as the clustering tests. The only difference is the number of nodes we are using. Where the clustering dataset was simple to divide into 25 files for training, the classification dataset is more naturally divided into 20 input files – there are 20 classes we are training to predict. Because of this, we are starting with 20 mappers and a single reducer for the first stage of classification.

To properly compare Mahout performance across runtimes, when testing with Granules we again ensure that the Granules resources are running on the HDFS worker machines. Among these 20 machines, one is also responsible for acting as a NameNode, and TaskTracker, while five act as Brokers (for the Granules runs). For both approaches, the Mahout operation was submitted from a machine outside the cluster. For both the Hadoop and Granules based approaches we used the same pre-processed inputs. These inputs are hosted in HDFS, and each approach starts by reading these inputs from file.

4.5 Classification Dataset

For our clustering experiments, we used the 20 News Groups [30] dataset. This is a standardized dataset available from the UCI Machine Learning Library (http://archive.ics.uci.edu/ml/index.html), and is used as the sample dataset in *Mahout In Action* [2]. This dataset consists of emails sent to various interest groups, and has already been separated into training and test sets. The goal is to train a model to sort unlabeled emails based on the sender, content, and title of the email. It is important to remember that this is a valid real-world example. Classifiers are often used to filter emails, with the typical example being a spam filter – particular senders, subjects, and content can flag an incoming email as potential spam.

The training dataset consists of 11,314 individual emails, each of which becomes an input to our classification models. The training set splits the emails by mailing list, so each mapper in the first phase is responsible for reading all emails addressed to a given class. This means about 566 mails processed per mapper on average. With a larger dataset, we would want to split the inputs further, to help speed up this initial processing step.

4.6 Classification Algorithms Supported in Mahout

Mahout supports several classification algorithms including Stochastic Gradient Descent (SGD) [17], as well as Naïve and Complementary Bayes implementations. While support is planned for other classification approaches such as Artificial Neural Networks (ANN) [31] and Hidden Markov Models (HMM) [32], these algorithms are not yet fully supported.

Mahout's classification algorithms create logistic regression models, so instead of predicting a single value for a sample input, these algorithms will return a list of probabilities. Each probability corresponds to the likelihood that the input belonged to a class the model was trained to recognize. Logistic regression is often very useful since it is possible to see how close the model was to providing an alternate classification.

Stochastic gradient descent is a method of function optimization which may be used to support other machine learning algorithms such as neural networks. SGD determines a gradient between sample points, and can then adjust weights in the objective function in order to move along this gradient. Where many functions will simply determine which direction to move a weight in (adding or subtracting from the current weights) then modify the weights by a fixed value, SGD will actually determine the gradient between two sample points and can then also know by what magnitude weights should be modified. In Mahout, SGD is implemented as a stand-alone logistic regression technique. The model is modified after each input is processed, and never needs to maintain any information about previous inputs. This means that SGD scales linearly as the number of inputs grows – you still need to read in and process every input, but the computing overhead is negligible. Because of this, SGD only has a sequential implementation – there is no reason to distribute it. While SGD is beyond the scope of this paper, it is important to understand which situations SGD is the preferred choice of classification algorithm.

SGD works best when operating on samples with continuous fields – where valid values exist along a range of possible inputs. Categorical fields – ones where there are a finite set of options – can also be massaged to work with SGD. An example of how these differ would be to consider a classification problem involving people. Age can be considered a continuous field, while the test group number would be considered a categorical field. SGD does not perform well when the sample data contains fields of open-ended text – essentially anything which is not a number and does not have a finite number of possibilities (cannot be considered categorical). An example of data which does not work well with SGD is a problem involving email filtering – SGD cannot make use of the body of the email directly, only statistics built around the text.

Naïve Bayes and Complementary Bayes [16], on the other hand, work best when operating on purely textual data. These approaches can also make use of categorical fields, but cannot use continuous fields unless they can somehow be massaged to look like textual data. Along with the ability to easily handle textual data, these approaches are designed to be run in a distributed manner, so they can handle larger data sets more easily than SGD can. The Bayesian approaches use these text inputs to determine how likely it is that any word seen belongs to a specific class. For example, emails which reference Pentium processors are more likely to belong to a computer-themed mailing list than a gardening mailing list. In order to determine which words or phrases are most useful, these implementations make use of TF-IDF (as discussed above in section 3.5) to reduce the importance of words which occur across all inputs.

Naïve Bayes and Complementary Bayes classification are well-suited for distributed execution, as much of the training process may be performed on portions of the dataset, only synchronizing after a large amount of processing has already occurred. In the original Hadoop based implementation, there are 4 tasks which go into building these classifiers: First, every training sample must be read and processed, building a table of probabilities for each class. In the second step, all probabilities are normalized across the classes. Once that is done, another pass is made to build overall probabilities with respect to each class, and then in the last step these probabilities are normalized and prepared to be used for classification. It is only in this last stage where Naïve Bayes and Complementary Bayes differ – they each generate weights slightly differently, which may result in slightly different classifications. Complementary Bayes is a slightly more expensive operation, but it uses the same format for samples as Naïve Bayes – this means no extra preprocessing steps, so it is simple to train both and compare results to determine which should be used in a production environment.

4.6.1 Naïve and Complementary Bayes Experiments

Both the Naïve and Complementary Bayes classification approaches use the same inputs. For our experiments we used an n-gram size of one, meaning we are analyzing individual words and the probability of these words occurring in any given class. Additionally, we are using a smoothing parameter of one for both Naïve Bayes and Complementary Bayes.

In both the Naïve and Complementary Bayes algorithms, there are four stages to the computation. First, the inputs are processed and separated into n-grams associated with the correct label. Once this has been done, the TF-IDF is calculated for each term in each label. After this step, there is a final processing function where all the weights computed so far are summed together in order to normalize all the results across all labels.

Once these processing steps have been completed, the classifier finally enters specific code for Naïve and Complementary Bayes. In this step, the weights calculated previously are used to build the classification model. While we are not expecting to see a dramatic speedup due to the non-iterative behavior of classification algorithms, we may be able to see some by leveraging Granules' ability to aggregate data.

4.6.2 Classification Results

For both Naïve Bayes and Complementary Bayes, we trained the classifier 20 separate times and averaged the results here. TABLE V. and TABLE VI. below show these results for Naïve and Complementary Bayes respectively. The average runtimes were essentially the same across both the Granules and Hadoop implementations, following our initial belief that we would not see significant processing gains moving from a file to stream based implementation. One interesting trend we see is the increased standard deviation when we moved to the Granules implementation. With both Naïve Bayes and Complementary Bayes, we saw 5 times as much variation, though slightly more with Complementary Bayes. This variation is most likely a result of our approach of aggregating data and then pushing a larger object across the network to the next phase of execution. One avenue to help cut down on this variation may be to look into sending results on in a piecemeal fashion, reducing the sudden load on the network.

TABLE V. NAÏVE BAYES RUNTIMES IN SECONDS FOR GRANULES AND HADOOP IMPLEMENTATIONS

	Mean	**Min**	**Max**	**SD**
Granules	172.51	158.44	203.60	10.232
Hadoop	172.86	170.32	180.24	2.290

TABLE VI. COMPLEMENTARY BAYES RUNTIMES IN SECONDS FOR GRANULES AND HADOOP IMPLEMENTATIONS

	Mean	**Min**	**Max**	**SD**
Granules	184.09	163.07	203.89	11.932
Hadoop	190.37	188.24	196.17	2.308

4.7 Fault-tolerant Classification

Classification in Mahout is a very quick task – generally less than 3 minutes. Because of this, we have not worked to implement fault-tolerance in the Granules implementation. While this does place the Granules approach at a slight disadvantage, it is not clear that restarting the process mid-training would grant a noticeable improvement in overall training speed.

5 Conclusions and Future Work

Our results demonstrate the feasibility of using Granules to manage the orchestration of large clustering and classification operations. Since Granules supports computations that can execute multiple, successive rounds of execution while retaining state it is particularly well suited for clustering algorithms that are inherently iterative. The ability to stream results between stages of an execution pipeline and activating computations when such (intermediate result) streams are available allows us to support distributed implementations of the clustering algorithms in an efficient fashion. Our benchmarks show that switching to a stream-based approach can greatly improve the runtimes of iterative tasks. This streaming feature allows us to incorporate support for fault tolerance without incurring performance overhead by interleaving the processing and I/O operations concurrently.

In Mahout only Naïve and Complementary Bayes are currently the only fully implemented classification algorithms with distributed implementations. As more algorithms are added to Mahout, we plan to continue this analysis to determine the effects of moving from a file to streaming based framework.

We also plan to explore the suite of recommendation algorithms in Mahout. Granules' ability to enter a dormant state between rounds of execution should mean a drastic increase in performance for a recommender system. Recommender systems work by finding similar items a customer may be interested in. Examples include Netflix [33] and Amazon [34], where users get personalized recommendations on items to view/purchase based on their previous patterns of viewing/shopping. These algorithms need to operate quickly and efficiently in order to provide accurate recommendations in a timely matter – if processing takes too long, you may miss a customer. Recommendation systems are a natural extension of clustering and classification algorithms, building on the idea of finding similar items to present to customers.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," ACM Commun., vol. 51, pp. 107-113, Jan. 2008 2008.

[2] S. Owen, et al., Mahout in Action: Manning Publications, 2011 (est.).

[3] T. White, Hadoop: The Definitive Guide, 1 ed.: O'Reilly Media, 2009.

[4] P. Berkhin, "A Survey of Clustering Data Mining Techniques," in Grouping Multidimensional Data: Recent Advances in Clustering, J. Kogan and C. K. Nicholas, Eds., ed: Springer, 2006, pp. 25-83.

[5] L. Qing and K. Byeong Man, "Clustering approach for hybrid recommender system," in Web Intelligence, 2003. WI 2003. Proceedings. IEEE/WIC International Conference on, 2003, pp. 33-38.

[6] C. W. Anderson and J. A. Bratman, "Translating Thoughts into Actions by Finding Patterns in Brainwaves," in Fourteenth Yale Workshop on Adaptive and Learning Systems, New Haven, CT, 2008, pp. 1-6.

[7] K. Y. Yeung, et al., "Validating clustering for gene expression data," Bioinformatics, vol. 17, pp. 309-318, April 1, 2001 2001.

[8] E. M. Forney and C. W. Anderson, "Classification of EEG during imagined mental tasks by forecasting with Elman Recurrent Neural Networks," in Neural Networks (IJCNN), The 2011 International Joint Conference on, 2011, pp. 2749-2755.

[9] A. C. Tan and D. Gilbert, "An empirical comparison of supervised machine learning techniques in bioinformatics," presented at the Proceedings of the First Asia-Pacific bioinformatics conference on Bioinformatics 2003 - Volume 19, Adelaide, Australia, 2003.

[10] S. Pallickara, et al., "Granules: A Lightweight, Streaming Runtime for Cloud Computing With Support for Map-Reduce," in IEEE International Conference on Cluster Computing, New Orleans, LA., 2009.

[11] S. Pallickara, et al., "An Overview of the Granules Runtime for Cloud Computing," in IEEE International Conference on e-Science, Indianapolis, 2008.

[12] S. Lloyd, "Least squares quantization in PCM," Information Theory, IEEE Transactions on, vol. 28, pp. 129-137, 1982.

[13] J. C. Bezdek, Pattern Recognition with Fuzzy Objective Function Algorithms. Norwell, MA: Kluwer Academic Publishers, 1981.

[14] P. McCullagh and J. Yang, "How many clusters?," Bayesian Analysis, vol. 3, pp. 101-120, 2008.

[15] D. M. Blei, et al., "Latent dirichlet allocation," J. Mach. Learn. Res., vol. 3, pp. 993-1022, 2003.

[16] D. Lewis, "Naive (Bayes) at forty: The independence assumption in information retrieval

[17] W. A. Gardner, "Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique," Signal Processing, vol. 6, pp. 113-133, 1984.

[18] C.-T. Chu, et al., "Map-Reduce for Machine Learning on Multicore," in Advances in Neural Information Processing Systems (NIPS), Vancouver, Canada, 2006.

[19] B. Catanzaro, et al., "Fast support vector machine training and classification on graphics processors," presented at the Proceedings of the 25th international conference on Machine learning, Helsinki, Finland, 2008.

[20] Y. Yu, et al., "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," presented at the Proceedings of the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008.

[21] P. Resnick and H. R. Varian, "Recommender systems," Commun. ACM, vol. 40, pp. 56-58, 1997.

[22] C. Hemmerich, et al., "Map-Reduce Expansion of the ISGA Genomic Analysis Web Server," presented at the CloudCom 2010, Indianopolis, USA, 2010.

[23] K. Ericson and S. Pallickara, "Adaptive heterogeneous language support within a cloud runtime," Future Generation Computer Systems, vol. 28, pp. 128-135, 2012.

[24] D. Borthakur. (2007). The Hadoop Distributed File System: Architecture and Design. Available: http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf

[25] M. Isard, et al., "Dryad: distributed data-parallel programs from sequential building blocks," in 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Lisbon, Poutugal, 2007.

[26] K. Ericson, et al., "Handwriting Recognition using a Cloud Runtime," in Colorado Celebration of Women in Computing, Golden, 2010.

[27] K. Ericson, et al., "Analyzing Electroencephalograms Using Cloud Computing Techniques," in IEEE Conference on Cloud Computing Technology and Science, Indianapolis, USA, 2010.

[28] A. McCallum, et al., "Efficient clustering of high-dimensional data sets with application to reference matching," presented at the Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, Massachusetts, United States, 2000.

[29] D. D. Lewis, "Reuters-21578 text categorization test collection, Distribution 1.0," A. T. L.-. Research, Ed., 1.0 ed: UCI Machine Learning Repository, 1997.

[30] T. Mitchell, "Twenty Newsgroups Data Set ", ed: UCI Machine Learning Repository, 1999.

[31] B. Yegnanarayana, Artificial Neural Networks: Prentice-Hall of India, 2004.

[32] B.-H. Juang, "Hidden Markov Models," in Wiley Encyclopedia of Telecommunications, ed: John Wiley & Sons, Inc., 2003.

[33] J. Bennet and S. Lanning, "Title," unpublished|.

[34] G. Linden, et al., "Amazon.com recommendations: item-to-item collaborative filtering," Internet Computing, IEEE, vol. 7, pp. 76-80, 2003.

[1]