Gaussian Mixture Models Use-Case: In-Memory Analysis with Myria

Ryan Maas[1], Jeremy Hyrkas[1], Olivia Grace Telford[2], Magdalena Balazinska[1], Andrew Connolly[2], and Bill Howe[1]
[1]Department of Computer Science & Engineering, University of Washington
[2]Astronomy Department, University of Washington

ABSTRACT

In our work with scientists, we find that Gaussian Mixture Modeling is a common type of analysis applied to increasingly large datasets. We implement this algorithm in the Myria shared-nothing relational data management system, which performs the computation in memory. We study resulting memory utilization challenges and implement several optimizations that yield an efficient and scalable solution. Empirical evaluations on large astronomy and oceanography datasets confirm that our Myria approach scales well and performs up to an order of magnitude faster than Hadoop.

1. INTRODUCTION

Big data analytics is a memory-intensive process because of the size of the datasets and the growing complexity of modern analytics (e.g., iterative machine learning algorithms). In addition, the size of scientific datasets is increasing past the point where in-memory computation can be done on a single node. Most scientists are proficient at single-node data analysis environments like Python, but must move to other tools for large-scale computation.

At the other extreme, Hadoop [10] scales well by breaking the analysis into small tasks, which can be executed independently. Analysis problems framed this way can scale to hundreds or thousands of nodes, but the overhead costs of Hadoop on a few nodes are often not competitive with other methods. Hadoop is also limited in that intermediate steps are materialized to disk, preventing effective use of in-memory data processing beyond the boundaries of one task.

Modern big data analytic engines such as Spark [25], Myria [20], and others focus on doing the analysis in memory in a shared-nothing cluster without going to disk. For these systems it is crucial that the implemented algorithms make efficient use of memory, as an inefficient implementation could quickly run into the memory limits of the cluster.

One such algorithm is a well known clustering method in the statistical literature known as Gaussian Mixture Modeling (GMM) [19], the implementation of which we explore here. We are motivated by use-cases in two fields, astronomy and oceanography, where a fast, in-memory implementation of GMM has immediate applications. The scientists we work with typically use single-noded systems such as Python and R to run GMM, but these approaches fail on datasets large than a few million rows. Two such datasets from both fields are presented here as use-cases.

In this paper, we implement GMM in the parallel, shared-nothing data management system Myria, which we choose as representative of the above in-memory distributed systems. We compare the performance of the Myria implementation with that of a known Python implementation [21] and a recently published Hadoop implementation [13].

We discuss several important optimizations and their effect on memory utilization and overall performance. These include the following:

- A primitive matrix data type for linear algebra.
- Optimizing the algorithm for efficient memory usage.
- In-memory iteration.

In summary, this paper makes three contributions: (1) We present the implementation of the GMM algorithm in the Myria system, where the input data starts from disk but the GMM analysis is performed in-memory. (2) We discuss the impact of several optimizations on the performance of the analysis. (3) We compare the performance of the implementation against Python and Hadoop implementations of the same algorithm on two real datasets from the astronomy and oceanography domains.

2. MOTIVATING APPLICATIONS

Many application domains make use of clustering algorithms such as Gaussian Mixture Models. This includes astronomy, oceanography, topic modeling, and others. We focus on two specific applications in this paper and describe them here.

2.1 Astronomy application

Large-scale astronomical imaging surveys (*e.g.*, Sloan Digital Sky Survey [22]) collect databases of telescope images. The key analysis is to extract sources (galaxies, stars, quasars, etc.) from these images. While extracting sources is a simple process, classifying these sources into object types is difficult. An example of this type of classification is the use of multiple wavelength observations in separating high redshift quasars (*i.e.*, galaxies powered by a central black hole) from stars within our Galaxy. Given that both quasars and stars are point sources (*i.e.*, they cannot be distinguished by data from a single image alone) and that there are 400-times more stars than quasars within a data set, the accuracy of this classification determines the success of finding some of the highest redshift sources within the universe. The GMM algorithm can serve to perform this classification.

IMDM '15, August 31 2015, Kohala Coast, HI, USA

DOI: http://dx.doi.org/10.1145/2803140.2803143

The data used in this paper is derived from two astronomical surveys: the Sloan Digital Sky Survey (SDSS) [22] and the Wide-field Infrared Survey Explorer (WISE) [24]. These imaging surveys cover over 10,000 square degrees (about one quarter of the sky) and have detected 495 million stars, galaxies, and asteroids. The measurements used to separate stellar from quasar sources are a mix of measurements at multiple wavelengths including:

- Optical fluxes from SDSS detected sources measured at five wavelengths from ultraviolet to near-infrared wavelengths (i.e. the u, g, r, i, and z passbands) with apertures optimized for point sources.
- Optical fluxes from SDSS detected sources measured at five wavelengths (i.e. the u, g, r, i, and z passbands) with apertures optimized for extended sources.
- Mid-infrared fluxes measured from the WISE survey (i.e. $w1$, $w2$, and $w3$) with apertures optimized for point sources

2.2 Oceanography application

The SeaFlow [23] cytometer, developed at the University of Washington, is a novel method for measuring the abundance of phytoplankton in oceans. Attached to a vessel, the machine continuously measures optical properties of microscopic particles in the water for days or weeks at a time. Measurements are obtained by pushing particles through a small capillary one-at-a-time, shining a laser on the particles, and measuring properties such as light scatter and fluorescence. The resulting data sets consist of millions of measurements of particles in the water. Each particle may be a member of a population of phytoplankton. The GMM algorithm can serve to cluster the data and identify these populations. Cluster analysis is a common method for analyzing cytometry data from other fields. However, SeaFlow data sets are substantially larger than most other cytometry data sets and therefore require more sophisticated methods to manage the scale of the data.

Data points in SeaFlow datasets contain a number of measurements. Only four are typically used when classifying the particles present in the data:

- Forward scatter small (*fsc_small*): the forward scatter of light from the laser as it passes over and through the particle. This measurement is proportional to particle size.
- Perpendicular scatter (*fsc_perp*): the light that scatters perpendicular to the laser. This measurement is somewhat representative of the complexity of the cell.
- Chlorophyll small (*chl_small*): red fluorescence measured from the particle, which is indicative of chlorophyll content.
- Phycoerythrin (*pe*): orange fluorescence measured from the particle, which is indicative of phycoerythrin, a protein found in some algaes.

For a more in depth description of the data and the challenge of classification in this application, see [13].

3. THE GMM ALGORITHM

In statistics, complex distributions can often be modeled by a mixture of canonical distributions, which are well characterized. A Gaussian mixture model (GMM) [19] is a model of a distribution as a mixture of K separate multivariate normal distributions, each with individual parameters represented collectively by θ. The probability density function of the model given the parameters is thus:

$$p(\mathbf{x}|\theta) = \sum_{k=1}^{K} \alpha_k \mathcal{N}(\mathbf{x}_i|\mu_k, \mathbf{\Sigma}_k),$$

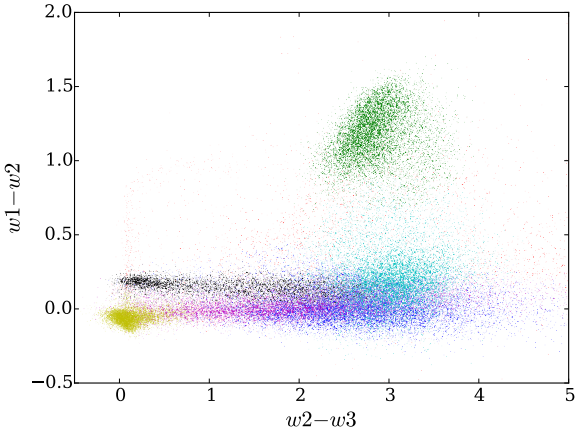

Figure 1: An example GMM clustering of astronomical point sources from the dataset in Section 2.1. Points are colored by the component of maximum responsibility for that point. Points clustered in yellow near $(0, 0)$ are likely to be stars, as opposed to quasars, which are colored in green.

where $\mathbf{x}_i$ is a D dimensional point, $\mathcal{N}$ denotes the normal probability distribution of an observation, and α_k, μ_k, and $\mathbf{\Sigma}_k$ are the amplitude, mean, and covariance matrix of each Gaussian component. In this model, each point observed has some probability of having been generated by each component. These are called the responsibilities r_{ik} of the point $\mathbf{x}_i$ for each of the K components. Having fit a GMM to a data set and deriving the responsibilities, a hard clustering for each point can be found by assigning that point to the component whose responsibility is highest. The result of running GMM on a small dataset from Section 2.1 is shown in Figure 1.

There is no analytic solution to estimate the parameters of a mixture of Gaussians from a sample data set. The standard approach is to iteratively maximize the likelihood function of the mixture model in an algorithm called Expectation Maximization (EM). The algorithm works as follows:

- Choose a number of clusters K and initialize their parameters.
- Expectation step: calculate the responsibility of each point to each Gaussian component given the current parameters:

$$r_{ik} = \frac{\alpha_k \mathcal{N}(\mathbf{x}_i|\mu_k, \mathbf{\Sigma}_k)}{\sum_{k'}^{K} \alpha_{k'} \mathcal{N}(\mathbf{x}_i|\mu_{k'}, \mathbf{\Sigma}_{k'})}$$

- Maximization: using the new responsibilities from the previous E step, re-estimate the parameters of the components. N is the number of points:

$$\alpha_k = \frac{1}{N} \sum_{i}^{N} r_{ik} = \frac{r_k}{N}$$

$$\mu_k = \frac{\sum_i r_{ik} \mathbf{x}_i}{r_k}$$

$$\mathbf{\Sigma}_k = \frac{\sum_i r_{ik} \mathbf{x}_i \mathbf{x}_i^T}{r_k} - \mu_k \mu_k^T$$

- Iterate subsequent EM steps until convergence of the likelihood function.

To implement Gaussian mixture models, one needs the ability to compute the above linear algebra statements, including the evaluation of a point $\mathbf{x}$ at a Gaussian distribution,

$$p(\mathbf{x}|\mu, \mathbf{\Sigma}) = \frac{1}{(2\pi)^{D/2}|\mathbf{\Sigma}|^{1/2}} \exp\{-\frac{1}{2}(x-\mu)^T \mathbf{\Sigma}^{-1}(x-\mu)\},$$

```
import numpy as np
from sklearn import GMM

rawData = np.genfromtxt(filename, delimiter=',',
        names=['g', 'i', 'w1', 'w2', 'w3'])

# Generate feature vectors from data
features[:,0] = rawData['g'] - rawData['i']
features[:,1] = rawData['i'] - rawData['w1']
...

# Create a GMM model with K components
gmm = GMM(n_components=K, n_init=1, n_iter=20)
# Fit the model to the feature data
gmm.fit(features)

# Get cluster assignments for each point
clusters = gmm.predict(features)
```

Figure 2: An example of using `sklearn.mixture.GMM` to cluster astronomy data. The data is loaded in to memory with Numpy, then a GMM model object is created. Calling `gmm.fit(features)` begins running the EM algorithm, ending either in convergence of after a maximum number of iterations.

and thus take the determinant and inverse of the covariance matrix $\mathbf{\Sigma}$.

Typically in the clustering use-cases considered here, the number of points N is in the tens or hundreds of millions and the dimensionality of the data D and the number of clusters K is on the order of 5 to 10. Thus calculating GMM involves millions of linear algebra operations on small matrices, differentiating it from some other machine learning algorithms that operate on large matrices.

4. PYTHON IMPLEMENTATION

For the single-threaded Python implementation, we make use of the scikit-learn module [21]. The `sklearn.mixture.GMM` object in this module is an implementation of the expectation-maximization (EM) algorithm for fitting Gaussian Mixture Models. The `fit` method serves to fit a mixture model with a user-specified number of components to data, and the `predict` method assigns a test sample to the Gaussian component to which it most likely belongs. The Python programs that we consider in this paper use the default amount of memory for a Linux process, as we describe further in Section 7. Figure 2 shows a snippet of Python code calling GMM in our astronomy use case.

The in-memory data representation used by `scikit-learn` routines is the Numpy `ndarray`, which is a wrapper around a multi-dimensional array in C. In our case, the array of data is N by D, the number of observations and number of features respectively. Numpy arrays take up little more memory than the C array they wrap. For instance, one of our `ndarray` data objects is 7.78 million rows and 4 columns, and each entry is an 8 byte float. The minimum size C array is 237.6 MB, and is equal to the size of our `ndarray` while loaded in memory.

5. HADOOP IMPLEMENTATION

GMMs based on Expectation Maximization can be parallelized and scaled up using a straightforward implementation in the MapReduce programming paradigm. Here, we explore the algorithm suggested by [7]. The algorithm works as follows:

- Map phase: each Mapper handles a subset of the points. Using the current parameters for all Gaussians, the mapper computes for each point the responsibility of each Gaussian (r_{ik} for all k Gaussians) and outputs r_{ik} as well as the point's observed values and observed variance scaled by r_{ik}, which are later used in the M step. The responsibilities are computed using the logsumexp [1] method to avoid underflow. Matrix operations are performed using the Jama Matrix library [12].
- Combiner phase: for each Gaussian, the mapper sums up the scaled point values, scaled variances, and r_{ik} values. These are then passed on to the Reducers.
- Reducer phase: Each reduce phase handles one Gaussian. The scaled values and variances are summed together, and divided by the sum of all r_{ik} values. This gives each Gaussian its updated mean and variance. The sum of all r_{ik} values is also divided by N, the number of input particles (this is also passed through the Map and Combine phases), which gives the new mixture weight. Each reducer outputs the new parameters for each Gaussian.

Although [7] only describes the naïve Map and Reduce steps to implementing EM, we found that writing a Combiner was necessary for achieving reasonable efficiency, since the naïve implementation requires sending data from every measurement to all K reducers (essentially, the size of the data to be reduced is K times larger than the data to be mapped).

6. MYRIA IMPLEMENTATION

In this section, we present the implementation of the GMM algorithm in the Myria big data system.

6.1 Myria system

Myria [20, 11] is a stack for big data management and analytics developed by the database group and eScience Institute at the University of Washington. It runs as a production service in the private cluster of the database group and supports several domain scientists on the University of Washington campus.

Queries in Myria are expressed as scripts in the MyriaL declarative language [11]. Myria's optimizer translates these scripts into MyriaX query plans. MyriaX is Myria's parallel, shared-nothing query execution engine. MyriaX uses PostgreSQL internally for data storage. Once it reads data out of PostgreSQL, however, it performs all subsequent processing in memory. It pipelines data across operators on the same machine and on different machines. Across machines, MyriaX uses data shuffling operators to re-hash the data as necessary. Each operator partition consumes batches of tuples and output batches of tuples. A TupleBatch object holds one batch of tuples. It uses an internal representation based on PAX [2], where each batch is a horizontal partition of a relation with a column-store internal representation. Myria's TupleBatch objects do not currently use compression.

Myria supports both synchronous and asynchronous iterations. For the GMM application, we use synchronous iterations, as the output `Components` table of every EM step must be fed in to the input of the next step. As these are small tables, we evaluate both execution models where the data goes to disk after each iteration and models where data streams from one iteration directly to the next.

6.2 Data model

Myria is a relational DBMS, and Gaussian Mixture Modeling fits well into the relational framework. We store the data and model parameters in two relations: `Points` and `Components`.

The `Components` relation has the following schema: `Components(gid,` $\texttt{amp}^{(t)}$`,` $\mu^{(t)}$`,` $\sigma^{(t)}$`)`. It contains one tuple per Gaussian component, whose fields are a unique identifier,

Input size (million points)	0.24	0.48	0.97
Ratio of runtimes:			
Matrix type / No type	1.01	1.05	1.03

Table 1: E step runtime performance with Matrix data type.

gid, and the parameters of that component at time (t). These parameters are the amplitude, mean vector, and covariance matrix of the Gaussian, stored as individual floating point values in row major order.

The Points relation has the following schema: Points(pid, x, $r^{(t)}$). It contains one tuple per point, whose fields are a unique point ID, pid, a vector, x, of $|x| = D$ features of the point, and a vector $r^{(t)}$, of $|r^{(t)}| = K$ responsibilities of that point to each Gaussian component. Each vector is stored as a sequence of floating points values. The K responsibility values are part of the mixture model, but we choose to store them in the data relation because they are frequently accessed with the feature values in the GMM use-cases that we consider. Storing the responsibilities together with the input data avoids this frequent join.

6.3 Linear algebra in Myria

As described in Section 3, the EM algorithm requires operations on vectors and covariance matrices using linear algebra. Myria does not yet support a matrix object as a primitive type nor linear algebra operations natively. Therefore, for our implementation, we create new, user-defined operators in the Myria source code that utilize the matrix library Jama [12]. This is the same matrix library used in the Hadoop implementation, allowing for a fair comparison across the two systems.

While Jama implements linear algebra routines entirely within Java, we also tested a linear algebra library, which calls out to LAPACK[15] libraries outside of the Java environment. One such library is jblas [14]. We compared jblas to Jama by measuring the cost of a matrix inversion, which is by far the most expensive matrix operation in the GMM algorithm. We found that, for matrices with dimension less than 5 by 5, Jama was faster for this key operation. Since our use-cases use covariance matrices of dimension 4 by 4, we decided to use the Jama libary.

We also explore whether a primitive matrix type in Myria would provide increased performance as opposed to unrolling the cells of a matrix in relational format. We implement a MyriaMatrix type, which is a wrapper around the Jama Matrix object, and use that type to pass vectors and matrices in the intermediate relations. Runtime analysis on the optimized Myria implementation shows that the Matrix wrapper type does not have a significant effect on runtimes. Serializing Jama matrices into arrays of floats and back is simply not a bottleneck operation in this computation. Additionally, our Myria Matrix type does not perform any compression, which could reduce the total amount of data shuffled. In fact, the extra steps of wrapping each Jama array in a Myria matrix object when passing between operators slightly adds to the runtime, as shown in Table 1. Nonetheless, a primitive matrix type makes the implementation significantly easier to write.

6.4 Naïve implementation

Similar to the MapReduce implementation, the EM algorithm for GMM can be expressed naturally with relational operators. Our first implementation seeks to change as little as possible from the standard relational algebra, adding only the functionality required for the linear algebra functions inside user-defined aggregate operators. Figure 3 shows the query plan for one iteration, which proceeds as follows from timestep (t):

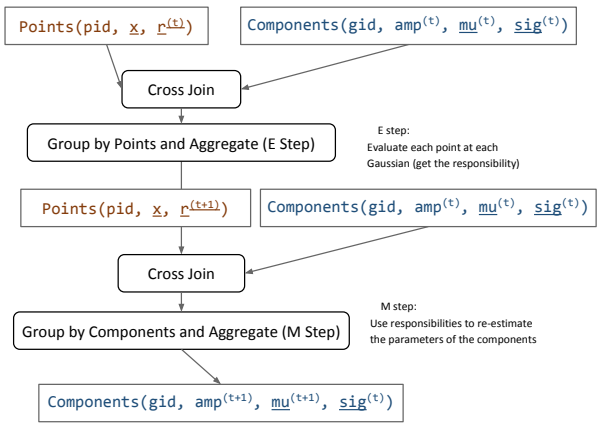

Figure 3: Query plan showing the naïve implementation of an EM algorithm for GMM in Myria. The relations are represented by sharp cornered boxes, and the operators by rounded boxes. This query plan performs one EM step, updating the parameters of the Gaussian components from time (t) to $(t+1)$.

E step:

- Cross product between the Points and Components relations to produce a relation with one tuple for every point-component pair.
- Group-by aggregate to evaluate each point at each component and aggregate the result by point ID. Return a new Points relation with the updated responsibilities for time $(t+1)$.

M step:

- Cross product between the updated Points and original Components relations to produce a relation with one tuple for every point-component pair, with the new responsibilities.
- Group by gid and aggregate to recompute the parameters of the Gaussian components using the updated responsibilities for each point.

The output of the query plan is the Components relation at time $(t+1)$, representing one iteration of the EM algorithm. The intermediate output of the E step can also serve to obtain the responsibility values from each point to each Gaussian. The query plan can be executed an arbitrary number of times to get as many iterations of the algorithm as desired.

6.5 Memory utilization challenges and optimizations

The cross products in the naïve query plan pose obvious problems for resource utilization. The number of tuples generated before the aggregates is equal to $N \cdot K$, where N is the number of data points and K is the number of components in the mixture model. While K is generally less than 10 in our applications, any multiplier is significant because the Points relation is in the tens to hundreds of millions. This overhead costs more in CPU than memory because these intermediate tuples are directly aggregated as they are produced. The group-by aggregate over Points, however, uses a large amount of memory because each point in Points has its own group and its own aggregate. The sum of the memory utilization of this group-by aggregate can easily cause out-of-memory errors. Overall, these cross products thus cost significantly in both CPU and memory.

In our evaluation of the naïve algorithm, these problems became immediately apparent. As we increased the input data size, a 16-node cluster running the naïve algorithm quickly dropped in points processed per second, and failed to process the full astronomy data due to out-of-memory errors. The results are shown in Section 7.1.

Our solution to this problem is to rewrite the join and aggregate operators, taking advantage of what we know about the EM algorithm. The result, described in the next section, is closer to the MapReduce style implementation of Hadoop.

6.6 Optimized implementation

The first observation is that the join followed by the group-by aggregate in the E step together produce one output `Point` tuple per input `Point` tuple. Since $K < 10$, the components table is less than 10 tuples. We already use a broadcast join in the naïve plan, but now we can write the join and aggregate as a single operator to avoid generating the large intermediate result.

The second observation of the EM algorithm is that no `Point` data needs to be shuffled between workers, only partial results of the M step calculation of Gaussian parameters. The calculations of means and covariance matrices are associative, so their partial results can be computed on local data and then aggregated by Gaussian component. This is similar to the combiner optimization discussed in the Hadoop implementation.

The final observation is that the M step does not require the Component parameters from time (t) to calculate them for time $(t+1)$, only the Point data and responsibilities are required. In the naïve plan, we use a cross product to replicate the `Point` data for each component before doing an aggregate. Since the `Point` relation contains the responsibilities, we can replace the cross product with a local group by and aggregate that outputs partial results of the M step, followed by a global group-by and aggregate to get the final, global result. The partial results get shuffled across machines but they represent a small amount of data.

The query plan for the optimized algorithm is shown in Figure 4, and the algorithm proceeds as follows from timestep (t):

E step:

- Broadcast `Component` to all workers and perform a local join with the local partition of the `Points` relation. For each point tuple, evaluate it at every component and aggregate the results directly, returning a new point tuple with the updated responsibilities.

M step:

- Aggregate the `Points` relation grouping by component ID locally, producing a tuple of partial results for each component.
- Group by component ID globally and combine the partial results for every component to calculate the final values of the parameters.

7. EVALUATION

We comparatively evaluate the three GMM implementations (Python, Hadoop, and Myria) on datasets from the astronomy and oceanography domains. We run all experiments on Amazon EC2 using *m1.large* instances. We measure the runtime of the Python implementation on only one instance and the performance of Myria and Hadoop deployments up to 16 instances. In all tests, the number of clusters is fixed ($K = 7$) and, unless otherwise noted, the runtime is measured for one iteration of GMM. In all distributed cases, the data is horizontally partitioned evenly across nodes.

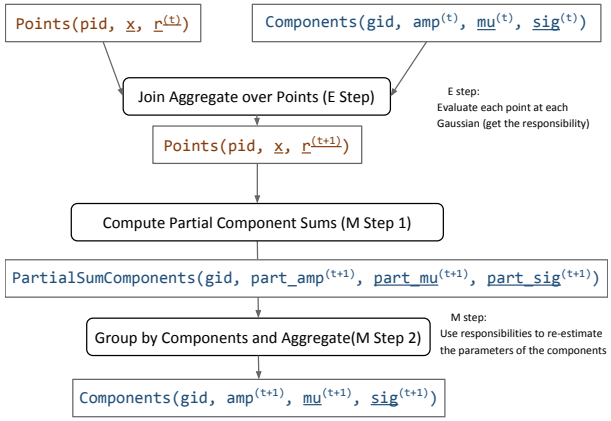

Figure 4: Query plan showing the optimized implementation of an EM algorithm for GMM in Myria. The relations are represented by sharp cornered boxes, and the operators by rounded boxes. This query plan performs one EM step, updating the parameters of the Gaussian components from time (t) to $(t+1)$.

In all experiments, the Python and Myria results are the median of three runs. Runtimes have negligible variance across the three runs. The Hadoop experiments are executed once.

Astronomy Classification Experimental Setup We make use of photometric data from the Sloan Digital Sky Survey (SDSS) and the Wide-field Infrared Survey Explorer (WISE). Only objects that are detected in both surveys are selected, giving a total of 495 million data points in the initial catalog. We then perform a standard quality cut and select objects within a range of r-band magnitude: $15 < r < 20$ to produce a high quality sample of 197 million points. We calculate four colors for each object: $g - i$, $i - w1$, $w1 - w2$, and $w2 - w3$; these are the features input into the GMM clustering analysis. We initialize the Gaussian components by the default method for `scikit-learn`, which is to assign random responsibilities to the points and perform one M step. This produces a set of initial Gaussian parameters, which we use in our experiments on all datasets. This calculation is done offline and not included in the experimental runtimes.

SeaFlow Classification Experimental Setup We analyze the Thompson 11 data set, one of three carefully curated SeaFlow data sets available [13]. The data set consists of approximately 22.6 million data points, each measured in four dimensions.

Before clustering, the data is filtered by one column representing chlorophyll content; this represents a real step in the analysis pipeline where oceanographers drop data points that they believe do not correspond to phytoplankton. The filter leaves approximately 15 million data points to cluster using GMMs. Since the filtering step is part of the full pipeline that the oceanographers follow to classify SeaFlow data, we present runtimes for both filtering the SeaFlow data set and for running one iteration of GMM on the filtered data set. Filtering the data in Myria is trivial and equivalent to a selection query (`SELECT * FROM thompson_11 WHERE chl_small < (const)`). A Hadoop implementation is only slightly more work. Only a Mapper is needed, which materializes a row and passes it along if the chlorophyll column is below the threshold. The default IdentityReducer in Hadoop is used to pass values through.

We initialize the GMM components on the filtered SeaFlow data set using a method similar to the k-means++ algorithm [3]. This initialization was done offline on a pre-filtered Thompson 11 data

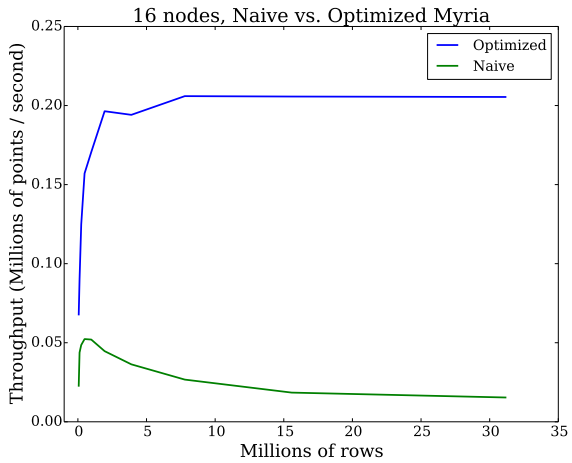

Figure 5: Throughput of naïve and optimized GMM implementations on a single Myria instance for varying dataset sizes.

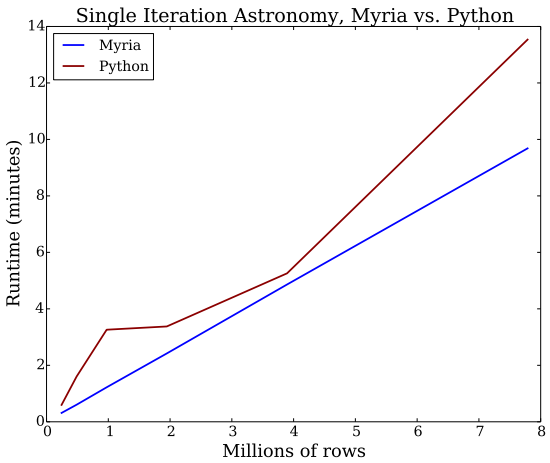

Figure 6: Runtime of GMM algorithm on a single node using either the Myria or Python implementations. Runtimes are similar with Myria slightly out-performing Python and displaying overall better scalability.

set, so no runtime for computing these initial points is included in the evaluation.

7.1 Naïve vs. Optimized Myria Implementation

We first compare the performance of the naïve algorithm, which uses cross products, and the optimized algorithm, which eliminates the cross products and performs the other optimizations described in Section 6.6. We first note that, on 16 nodes, the largest dataset that we can process with the naïve algorithm contains 30 million rows (1.5 GB). Any larger data sets fails due to memory errors. Figure 5 shows the number of points processed per second (throughput) as a function of input size, for both the naïve and optimized algorithms on 16 nodes. The plot shows that the throughput of the naïve algorithm peaks at 400,000 points/sec (24 MB), and subsequently drops for all larger datasets. In contrast, the throughput of the optimized algorithm shows no falloff for the largest datasets. On the largest dataset the optimized algorithm achieves 13 times higher throughput than the naïve one.

We also take memory profiles of the worker nodes during the EM steps of both algorithms, including the specific workers that run out of heap space and fail in the naïve algorithm. As expected, for the naïve algorithm, the worker's memory usage quickly reaches the maximum heap size for large datasets. When the garbage collector can no longer make space, the worker fails and halts the EM step.

On datasets that the naïve algorithm can process, the main difference between the two implementations is that the optimized algorithm garbage-collects more evenly than the naïve algorithm. Workers in the optimized algorithm rarely do more than one garbage-collection per second, while the naïve algorithm must do more frequent garbage collections even on the datasets for which it achieves maximum throughput.

7.2 Quantitative Evaluation

Our quantitative evaluation comprises three categories for each system: How runtime grows with input size, how runtime drops with the number of worker nodes, and how runtime grows with number of iterations of the algorithm. Where possible, we evaluate all systems on the astronomy and oceanography datasets. However, the oceanography data set (15 million rows, 472 MB) is too large for the Python implementation to run in memory, so for Python we only compare on astronomy results.

First we consider a single iteration of the algorithm and vary the input size. The Python implementation only runs on a single node, so we compare that to the Myria implementation on a single node. Figure 6 shows the runtimes for the Myria implementation and the Python implementation as a function of input size in rows. For all sizes of input, Myria is faster than Python for one iteration. The Myria runtimes also scale linearly, while the Python runtimes plateau between one and two million rows before continuing to increase. We believe this is due to the Python in-memory data structures fitting into the CPU cache. The data in Python is an N by D array, where N is the number of points and D is the number of dimensions. Whether that array can fit into the cache when first loaded will determine how long it takes to process the entire data set. On the other hand, in Myria each point is processed one by one, with no concept of a data array.

The Hadoop implementation scales to as many nodes as desired, though presumably Hadoop should be more efficient the more nodes it uses. We compare the runtime of Myria and Hadoop on 16 nodes as a function of input size in Figure 7. Myria's runtimes scale near perfectly linear. Hadoop's runtimes plateau between 5 and 20 million rows, then continue to increase. In this case it appears that for smaller input sizes the cost of Hadoop's startup is the main source of overhead. Beyond 20 million rows (50 MB per worker), the computation cost begins to dominate and runtimes start to grow linearly with input data size.

Next, we examine how runtime scales as the number of worker nodes increases. Figures 8 and 9 show runtimes plotted for varying numbers of nodes for the astronomy and oceanography datasets respectively. Table 2 summarizes the speedup of Myria and Hadoop on each dataset. Hadoop was only run on two nodes or more, so its speedup is normalized with respect to two nodes. Both systems show close to linear speed-ups on both datasets.

Finally, we examine runtimes of the algorithm for more than one iteration. Since Hadoop is significantly slower than Myria already for one iteration, we focus our evaluation on the comparison between Python and Myria. In the real astronomy and oceanography use-cases, the GMM algorithm needs to be run for upwards of 10 iterations before convergence.

We consider the astronomy use case of 7 million points (374

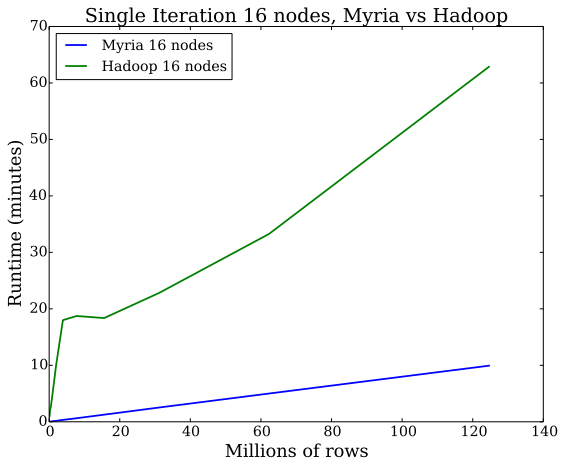

Figure 7: Runtime of GMM algorithm on astronomy datasets and a 16-node Hadoop or Myria cluster as we vary the input data size.

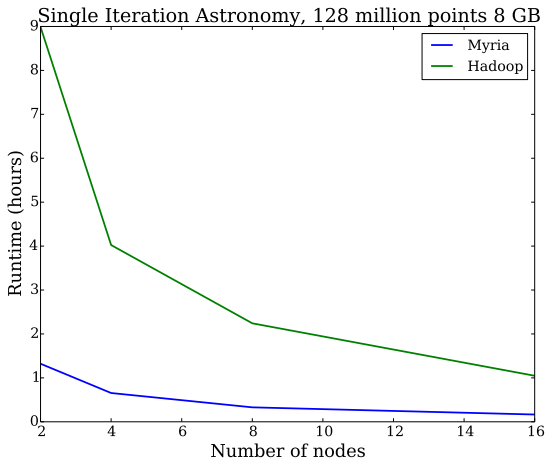

Figure 8: Runtime of GMM algorithm on astronomy data for varying cluster sizes using Hadoop or Myria.

Number of nodes	2	4	8	16
Myria Astronomy	1.97	3.97	7.91	15.72
Hadoop Astronomy	N/A	4.47	8.02	17.16
Myria Oceanography	1.93	3.90	7.36	15.27
Hadoop Oceanography	N/A	3.77	6.23	16.32

Table 2: Speed-ups for Myria and Hadoop on astronomy and oceanography data.

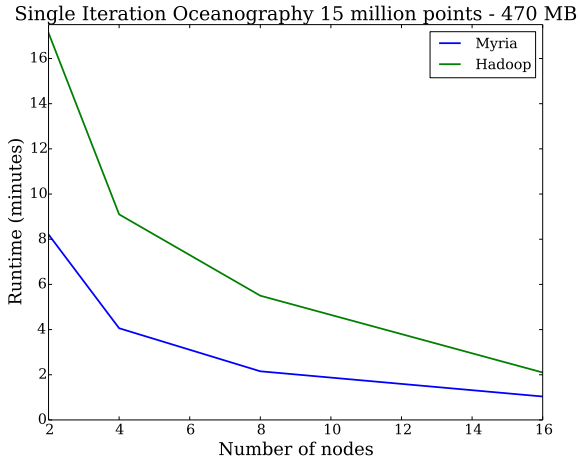

Figure 9: Runtime of GMM algorithm on oceanography data for varying cluster sizes using Hadoop or Myria.

MB), the largest data set which Python could fit in memory before crashing. Both the Myria and Python runtimes grow linearly with the number of iterations, as expected for a synchronous iterative algorithm. While Python is slower than Myria in the first iteration, Python is faster in subsequent iterations because Myria must repeatedly scan the `Points` table. As a result, on a single node Python is faster for more than a few iterations. To achieve better times than Python, more Myria nodes can be used. For ten EM steps on the 7 million point data set, Python takes 16.4 minutes, single node Myria takes 79.2 minutes, and 16 node Myria takes 5.3 minutes.

We also experiment with chaining together multiple EM steps in the same query plan, keeping the intermediate `Components` table results in memory. The result is that runtimes were approximately equal to the tests where `Components` tables were saved to disk. This is because with each iteration the `Points` table must still be scanned from disk, which is orders of magnitude larger in size than `Components`. Even though multiple iterations are chained together in this method, we do not see the speedup one might expect from the `Points` table being cached in memory. The difference in speed per iteration represents the tradeoff of our Myria implementation versus Python. Multiple iterations of Python will be faster than Myria on a single node, but Myria's streaming scan of the data lets it process hundreds of millions of points on a single node.

The full power of the Myria implementation is in taking advantage of multiple nodes and the largest datasets. On the 128 million row dataset in Figure 8, 25 iterations on a single Myria node would take approximately three days, but on sixteen nodes would complete in just over four hours.

7.3 Qualitative Evaluation

The GMM implementation in the `scikit-learn` package is approximately 800 lines of code, though in practice scientists would not write their own implementation. Using the prepackaged algorithm involves a significant number of data-munging steps, since the data must be manually loaded from csv files and code must be written to extract the features.

In Myria, the data from the astronomy use-case was processed through a series of MyriaL queries to produce the features used by the algorithm. Running these queries in Myria is trivial, while for Python and Hadoop the data needed to be processed separately before running the GMM algorithm. Myria's query language, MyriaL, does not yet support GMM as a primitive operator, so queries must be sent directly to the query execution backend in the form of JSON physical query plans. We plan to expose the GMM algorithm at the language layer in future work, both as a convenience for users and to allow for interaction with Myria's query optimizer.

The Myria implementation comprises three hand-coded operators in the database system, each with approximately 1000 lines of code. While this would seem daunting for a user to write, these operators are just modified versions of standard database operators join, apply, and group-by aggregate. The actual lines of code added and modified total approximately 500 lines across all three operators. In future work, we plan to extend MyriaL to allow the same code to be written

by users without modifying the source code of the system.

Our Hadoop implementation of GMMs contains Mapper, Combiner, and Reducer classes as described in Section 5. In addition, we implement a Gaussian class to evaluate points in the distribution and keep track of the Gaussian parameters, a class to represent each row of data (the input to the mappers), and a class to represent the input to the Reducers. This is a simple, if inefficient, implementation. In addition, the current parameters for each Gaussian must be added to the cache path of every Mapper so they can construct Gaussian classes. Each application has a driver class to setup the Hadoop jobs and run them. All things considered, the amount of code for the Hadoop implementation is only slightly above 1000 lines of code. The EM evaluation of GMMs maps nicely to the MapReduce programming paradigm, and no mathematical/programmatic difficult arose other than those that would be common to any implementation of GMM.

Comparing the implementation of the three algorithms qualitatively, the Python implementation is the most straightforward adaptation of the EM equations in Section 3: The primitive operations in the language resemble linear algebra. The Hadoop implementation requires the most original code because of the many classes that must be hand-written. The Myria implementation requires less code than the Hadoop implementation and is simpler to understand because of its adaptation of relational operators.

8. RELATED WORK

There has been a significant amount of work on distributed analytic systems. These systems provide tools for writing machine learning algorithms, often including linear algebra. Apache Mahout [17] is a library of machine-learning algorithms implemented in Hadoop and Spark. While some of the linear algebra functionality needed for GMM exists, as of Mahout 0.10.0 there is no implementation of Gaussian Mixture Modeling.

SciDB [6] is built around an array data model, and supports distributed linear algebra through calls to ScaLAPACK [5]. No GMM implementation is currently available in SciDB, though we initially experimented with implementing the algorithm in that system. At that time, we found that SciDB's linear algebra support was best suited for large matrices with dimensions upwards of hundreds or thousands, whereas our algorithm requires millions of linear algebra operations on small matrices of around 5 dimensions.

GraphLab [16] is a graph-based engine, which contains implementations for clustering algorithms such as k-means. Gaussian Mixture Modeling is not yet supported, though the linear algebra routines needed to support the algorithm are currently being added to the GraphChi version of the system.

MLlib is a machine learning library for the Spark engine [18], an in-memory system built on the MapReduce framework. Very recently a GMM algorithm was added to MLlib, though not at the time of writing of this paper. Preliminary results comparing the GMM algorithm of MLlib show that the Spark implementation is faster but overall comparable to Myria on a 16-node cluster for one iteration. Myria is faster on smaller data sizes because it leverages all nodes of the cluster for the smallest data sizes. Spark is faster for multiple iterations because the data is only read into main memory once. A more in-depth comparison would be valuable future work.

There also exist frameworks for parallelizing workflows in existing analytic systems such as Python and R. Distributed R [8] is one such system. Generally, these frameworks do not include implementations of specific algorithms such as GMM. Without reimplementing GMM, we were not able to test the efficiency of those frameworks. However, if the goal is to write the fastest implementation of GMM regardless of system, these would be a reasonable starting point for future work.

In short, few distributed analytics systems implement support for GMM, which is unusual given its utility. We expect to see more systems provide GMM out-of-the-box in the future. Most commonly, GMM is implemented in analytic engines on a single node. In this paper, we use scikit-learn [21], and other implementations can be found in the R programming language [9] [4].

9. CONCLUSION

We implemented Gaussian Mixture Modeling (GMM) in the Myria shared-nothing relational data management system, and evaluated the performance on real use-cases from astronomy and oceanography. We compared the distributed in-memory results to a Python implementation on a single node, and Hadoop. We presented both naïve and memory-efficient implementations of GMM, and saw how optimizations for in-memory processing enabled Myria to process much larger data sets than Python, even on a single node.

Acknowledgments

This work is supported in part by NSF IGERT grant DGE-1258485, NSF grant IIS-1247469, the Intel Science and Technology Center for Big Data, and a gift from Amazon.

10. REFERENCES

[1] R. Adams. Computing log-sum-exp, Jan. 2013.
[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proc. of VLDB*, pages 169–180, 2001.
[3] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *Proc. of SODA*, pages 1027–1035, 2007.
[4] T. Benaglia, D. Chauveau, D. R. Hunter, and D. Young. mixtools: An R package for analyzing finite mixture models. *Journal of Statistical Software*, 32(6):1–29, 2009.
[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.
[6] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proc. of SIGMOD*, pages 963–968, 2010.
[7] C. Chu et al. Map-reduce for machine learning on multicore. In *Proc. of NIPS*, pages 281–288, 2006.
[8] HP Distributed R. http://www.distributedr.org.
[9] C. Fraley and A. E. Raftery. Model-based clustering, discriminant analysis and density estimation. *Journal of the American Statistical Association*, 97:611–631, 2002.
[10] Apache Hadoop. http://hadoop.apache.org.
[11] D. Halperin et al. Demonstration of the Myria big data management service. In *Proc. of SIGMOD*, pages 881–884, 2014.
[12] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. Jama: A Java matrix package. *URL: http://math. nist. gov/javanumerics/jama*, 2000.
[13] J. Hyrkas, D. Halperin, and B. Howe. Time-varying clusters in large-scale flow cytometry. In *Proc. of AAAI*, pages 4022–4023, 2015.
[14] jblas: Linear Algebra for Java. http://jblas.org.
[15] LAPACK :âĂL' Linear Algebra PACKage. http://www.netlib.org/lapack.
[16] Y. Low et al. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.
[17] Apache Mahout. http://mahout.apache.org.
[18] Spark Machine Learning Library (MLlib). http://spark.apache.org/mllib.
[19] K. P. Murphy. *Machine Learning : A Probabilistic Perspective*. The MIT Press, 2012.
[20] Myria: Big Data as a Service. http://myria.cs.washington.edu.
[21] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[22] Sloan Digital Sky Survey III: DR 10. http://www.sdss3.org/dr10/.
[23] J. Swalwell, F. Ribalet, and E. Armbrust. Seaflow: A novel underway flow-cytometer for continuous observations of phytoplankton in the ocean. *Limnology & Oceanography Methods*, 9:466–477, 2011.
[24] Wide-field Infrared Survey Explorer. http://www.nasa.gov/mission_pages/WISE/main/index.html.
[25] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of USENIX*, pages 15–28, 2012.