Gaussian Mixture Models on Myria and Spark

Parmita Mehta

CSE 599c, Spring-2017

June 6, 2017

In this project we compare performance of Spark and Myria on clustering algorithm, Gaussian Mixture Models(GMM) on an astronomy catalog dataset. We compare python user defined functions based implementations on both systems, using python computing libraries SciPy and NumPy. We compare performance per iteration as well as the performance impact of change in datasize. We fins Myria and spark perform similarly.

1 Introduction

Large-scale astronomical imaging surveys (e.g., Sloan Digital Sky Survey [SDSS]) collect databases of telescope images. Analysis of these images extracts sources e.g., galaxies, stars, quasars, etc. GMM is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. Thus, GMM can be used to classify the sources extracted from the images to classify these sources into different categories. The data used in this paper is derived from two astronomical surveys: the Sloan Digital Sky Survey and the Wide-field Infrared Survey Explorer [WISE]. The native implementation on Myria was compared to Hadoop in [RM15] for this dataset. In this project we compare python UDF based implementations on Apache Spark [MZ12] and Myria [JW16], examining the system tuning and setup with the same python implementation.

2 Gaussian Mixture Models and Expectation Maximization

In statistics a mixture model is a probabilistic model representing the presense of subcomponents in the overall population. A Gaussian Mixture Model (GMM) assumes all the data points are generated from a mixture of finite number of Gaussian distributions with unknown parameters. A random variable X is Gaussian if it has the following probability distribution function:

$$p_x(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{(x-\mu)^2/2\sigma^2}$$

The two parameters are, μ the mean and, σ^2 the variance.

The main difficulty in learning GMMs from unlabeled data is that we do not know which points belong to which component. Expectation-Maximization(EM)[AD77] is a statistical algorithm to get around this problem via an iterative process. It constitues of two steps, E-step and M-step, which are repeated unitl parameters converge to maximize the likelihood of the observations. EM starts by assuming random K components and computes for each observation the probability of belonging to each component of the model. This is called the E-step. The second step in the iterative process, called the M-step, uses the points associated with each component to estimate better parameter for each of the K components. The E-step and the M-step are repeated until convergence.

3 Systems

This section we briefly describe the two systems we evaluate.

Myria is a shared-nothing DBMS developed at the University of Washington. Myria uses the relational data model and PostgreSQL as its node-local storage subsystem. Users write queries in MyriaL, an imperative-declarative hybrid language, with SQL-like declarative query constructs and imperative statements such as loops. Myria query plans are represented as graphs of operators and may include cycles. During execution, operators pipeline data without materializing it to disk. Myria supports Python user-defined functions and a BLOB data type. The BLOB data type allows users to write queries that directly manipulate objects like NumPy arrays. For our experiments we implement EM algorithm as python user defined function and aggregates.

Spark supports a dataflow programming paradigm. Spark's primary abstraction is a distributed, fault-tolerant collection of elements called Resilient Distributed Datasets (RDDs), which are processed in parallel. RDDs can be created from files in HDFS or by transforming existing RDDs. RDDs are partitioned and Spark executes separate tasks for different RDD partitions. RDDs support two types of operations: transformations and actions. Transformations create a new dataset from an exisiting one, e.g., map is a transformation that passes each element through a function and returns a new RDD representing the results. Actions return a value after running a computation: e.g., reduce is an action that aggregates all the elements of the RDD using a function and returns the final result. All transformations are lazy and are executed only when an action is called. In addition to map and reduce, Spark's API supports relational algebra operations: e.g., distinct, union, join, groupby, etc. In our experiments we use the Python interface **PySpark**, and use user defined functions and aggregates to implement GMM. Spark's machine learning library (SparkML) supports EM natively, we chose to implement the functions to ensure equitable comparison with Myria.

4 Use-case

4.1 Data

Our data comprises of extracted sources from large scale astronomical imaging surveys. Each of the sources extracted could be a galaxy, star, quasar etc. The source information consists of optical fluxes (brightness) measured a different wavelengths. The assumption is that the optical fingerprint of stars, quasars, etc. is identical and can be determined by the flux values at various wavelenghts. Fig.1 shows the raw data plotted (X,Y and V,W) The Points relation and RDD have the following schema:
Points: (PointID, X,Y,V,W, Weights,Label). PointsID is an integer, X,Y,V and W are floats and Weights is a 7×1 array of floats, The Components relation and RDD have the following schema:

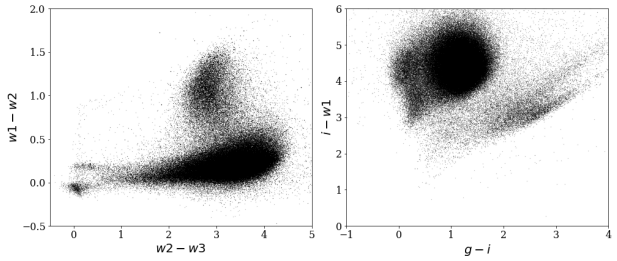

Figure 1: ***Raw data, 200K data points.***

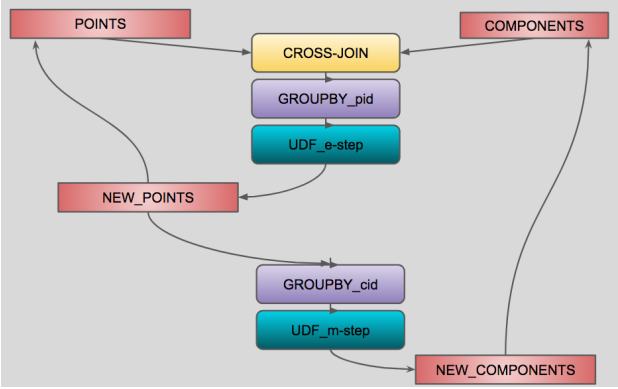

Figure 2: ***Query plan.***

Components: (ComponentsID, Mean, Variance), where ComponentsID is an integer, Mean is a 4×1 array of floats and variance is 4×4 matrix of floats. For the purpose of this evaluation we assume diagonal variance. We have 200K point sources, for each source there are four flux values. We also start with seven components. Initial parameters for these components are pre-generated.

The processing pipeline consists of two steps, E-step and M-step. After components and points relations/RDDs are initialized the processing pipeline iterates over these two steps, checking the log-likehood value after each iteration to determine convergence. This is depicted as a query plan in Fig.2.

5 Experiments

We evaluate the performance of the implemented use cases and the system tunings necessary for successful and efficient execution. All experiments are performed on a two node cluster in the Amazon Cloud. Each of the instances is m3.large type, with 2 vCPU, 7.5GB of memory, and 64GB SSD storage.
We ran three set of experiments. For the first experiment we ran the entire pipeline end-to-end on a two node cluster, measuring the time for each iteration. Next we varied the data on both systems for a single iteration at three sizes of data at 200K, 100k, 50K points respectively. Finally, we varied number of partitions in spark to assess the variation in degree of parallelism for Spark.

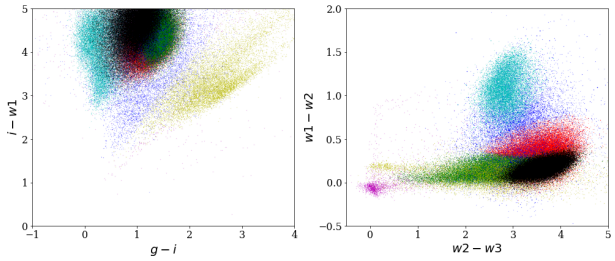

Figure 3: ***Seven components detected via GMM.***

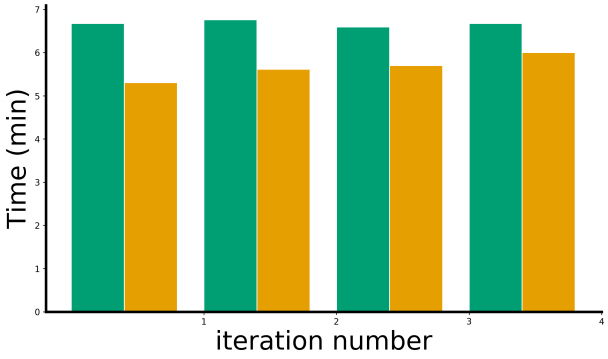

Figure 4: ***End to end time for iterations [1,4]***

We attempted to follow the query plan from Fig.2 for both Spark and Myria with minor differences. Myria does not support tuple tearing so after the weights have been calculated in the E-step a map -like UDF operation is required to assign each point to a component.

Fig.3 shows the final results for GMM, with the seven components represented as different colors for the points. The two views are (x,y) and (v,w) for each point.

6 Evaluation

Fig.4 shows the end-to-end time for first 4 iterations of the EM algorithm on both Myria and Spark. Over all timings for each iteration are similar, with Myria slower than Spark. Time for each iteration is similar (given

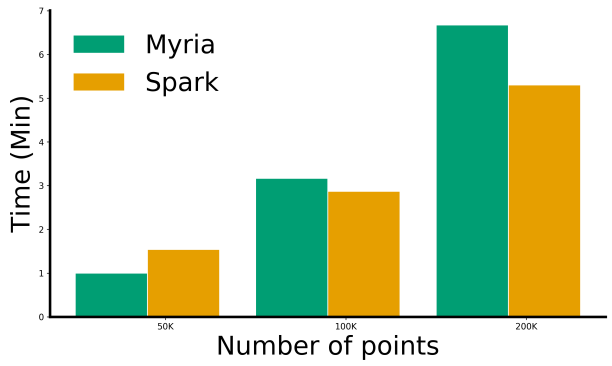

Figure 5: ***Time for one iteration, with varying number of points to classify.***

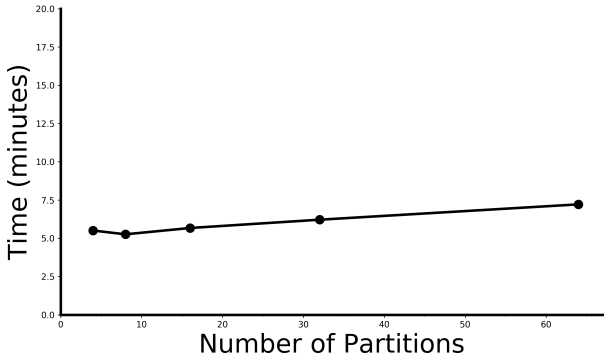

Figure 6: ***Time for one iteration, 200K points, varying number of partitionsin Spark.***

variance of running in AWS) for each iteration in Myria but increases a little for each iteration in Spark. This seems to be due less efficient garabage collection for spark cache (not jvm), when compared to JVM garbage collection in myria.

Fig.5 shows the variation in time for a single iteration for varying datasize. For this experiment we kept the number of partitions equal to default partitions(4) in Spark. For Myria we utilized with 1 worker per compute node. This demonstrates good scalability for both systems as long as the entire data set is smaller than the memory available to the cluster. Myria is slower than Spark for a couple reasons. Firstly, the extra operation to assign a label to each point based on weight incurs high overhead as the data has to be streamed to a python process and results back for each point. Secondly, if a tuple contains a BLOB type, which is the format we use for storing numpy arrays (component mean, component variance and point weights are all stored as BLOBs), the tuple batch size is one. Which leads to higher runtime.

Finally Fig.6 shows the variation in runtime as number of partitions is changed for Spark. For Spark performance for 4-16 partitions is similar and it grows after 16 partititons. This is to be expected total runtime is dependent upon degree of parallelism, and the time for the actual computation. However, parallelism has an overhead due to scheduling and communication. Thus increasing the degree of parallelism beyond the point where the cost of parallelism is offset by parallizing the computation will result in increase in runtime, which is exactly what we observe for number of partitions for Spark.

7 Conclusion

In this project we implemented GMM on Myria and Spark using python UDFs. Both systems presented a similar degree of difficulty and presented similar performance.

References

[spark] http://spark.apache.org/

[SDSS] http://www.sdss3.org/dr10/

[WISE] http://www.nasa.gov/mission pages/WISE/main/index.html

[myria] J. Wang et al., The myria big data management and analytics system and cloud services., *CIDR.*, (2017)

[MA15] Ryan Maas et al., Gaussian Mixture Models Use-Case: In-Memory Analysis with Myria, *VLDB.*, (2015)

[MZ12] Matei Zaharia et al., Resilient Distributed Datasets: A Fault-Tolerant Abstraction for
In-Memory Cluster Computing, *NSDI.*, (2012)

[AD77] A. P. Dempsteret al. ,Maximum likelihood from incomplete data via the EM algorithm, *JRSS*,(1977)