# Cognitive Impairment Gene Expression Distributed Computation

By Wei Shi

## 1. INTRODUCTION

The map-reduce program is written in PySpark for selecting top k t-scores between patients diagnosed with Alzheimer's Disease (AD) and those who have no cognitive impairment (NCI) using *ROSMAP_RNASeq_Entrez.csv* and *gene_cluster.csv* files. I will refer to *ROSMAP_RNASeq_Entrez.csv* as the "Rosmap" file, and *gene_cluster.csv* as the "gene cluster" file. Each *clusterID* in the *gene_cluster.csv* is assigned a group of Entrez IDs [1] whose gene values are summed per *clusterID*. There are 16,380 Entrez IDs, the numerical fields left of "DIAGNOSIS" column in the Rosmap table below:

| PATIENT_ID | DIAGNOSIS | 1 | 2 | 3 | … |
|---|---|---|---|---|---|
| X764_130520 | NA | 0.35 | 9.68 | 0.11 | … |
| X800_130701 | NA | 2.24 | 32.04 | 0.13 | … |
| X164_120423 | 6 | 3.12 | 24.08 | 0.04 | … |
| … | … | … | … | … | … |
| X954_131107 | 1 | 0.39 | 36 | 0.04 | … |

Table 1

The gene cluster file looks like the table below. The irrelevant columns are not shown.

| ClusterID | … | subunits(Entrez IDs) | … |
|---|---|---|---|
| 1 | … | 604;9759 | … |
| 2 | … | 604;10014 | … |
| 3 | … | 2033;1387;8850;8202 | … |
| … | … | … | … |
| 6375 | … | 10016;553115 | … |

Table 2

The null hypothesis for our t-test is that there is no difference between the means of clusters of gene expression values between patients diagnosed with AD, diagnosis of 4 or 5, and NCI, diagnosis of 1.

## 2. METHODS

After the Rosmap and gene cluster files are read into a Spark Resilient Distributed Datasets (RDD). The RDD is transformed until the top-k t-scores are calculated. Each step of the algorithm will take input from the previous step unless specified otherwise. The goal is to calculate the top-k t-score between two diagnosis groups AD and NCI. The format of each section begins with the initial input, the intermediate numbered steps required to generate the final output of the section, and then the schema of the final output of that section. You can skip the intermediate steps of each section and look only at the initial inputs and final output schemas to get the general idea.

## 2.1. BROADCAST ENTREZ IDS

**Input Rosmap file schema**: rosmap_rdd ← (patient ID, diagnosis, g1, g2, g3, …)

The *g1, g2, g3, …* of the input are instances of the patient's gene expression values. For this section we will process the Rosmap file. First we need to extract the Entrez IDs from the header and broadcast it to every node in the cluster.

**1**. Filter: read the Rosmap file into an RDD and get the header of the Rosmap file.

header_rdd ← [“PATIENT_ID”, “DIAGNOSIS”, 1, 2, …]

Then collect the *header_rdd* and delete “PATIENT_ID” and “DIAGNOSIS” from the header.

header_rdd ← [1, 2, 3, 9, 10, …]

Then broadcast the header_rdd to each node in our cluster.

**2**. Filter: remove the header from *rosmap_rdd*.

**3**. Filter: select only patients with diagnosis 1 for NCI, and 4 or 5 for AD from *rosmap_rdd*.

**4**. Map: associate patient ID and diagnosis of each row with each gene expression value. A sample row becomes:

[(PID, D, g1), (PID, D, g1), …]

Where *PID* is an instance of a patient ID, *D* is an instance of diagnosis, and *g1, g2, …* are instances of gene expression values. Each row of these tuples contain a single patient's gene expression values.

**5**. Flatmap: zip the *header_rdd* with each row of the *rosmap_rdd*. A sample row becomes:

(1, (PID, D, g1)), (2, (PID, D, g1)), (3, (PID, D, g1)), (9, (PID, D, g1)), …

**Output schema**: (Entrez ID, (patient ID, diagnosis, gene value))

## 2.2 MAPPING GENE CLUSTER FILE

**Input gene cluster file schema**: (cluster ID, …, Organism, …, Entrez IDs, …)

Input is the gene cluster file. This section converts the gene cluster file to an RDD of key-value tuples of the form (Entrez ID, cluster ID).

**1**. Map: the input is the gene cluster file. The gene cluster file is read in using comma as delimiter producing the following schema:

[ cluster ID, …, Organism, …, Entrez IDs, ... ]

where the Entrez IDs are under one column and are delimited by semicolons “;”.

**2**. Filter: we want to extract only those rows for humans:

[ cluster ID, …, Human, …, Entrez IDs, …]

**3**. Map: we are only interested in cluster ID and Entrez IDs:

[ cluster ID, …, Human, …, Entrez IDs, …] → [ cluster ID,  [EID1, EID2, ...]]

Here I split the Entrez IDs by their semicolon delimiter.  *EIDn* is an instance of an Entrez ID. The data from other columns are omitted since they're not need.

**4**. Flat-map: flat-map each row to tuples of the form:

[ CID1, [EID1;EID2; …] ] → (EID1, CID1), (EID2, CID1), ...

[ CID2, [EID5;EID3; …] ] → (EID3, CID2), (EID5, CID2), …

Here *CIDn* is an instance of cluster ID. This results in an RDD with tuples of schema (Entrez ID, cluster ID) mixed together:

(EID1, CID1), (EID2, CID1), (EID3, CID2), (EID5, CID2), …

**Output schema**: (Entrez ID, cluster ID)

## 2.3  GENE VALUE SUMS BY ENTREZ ID PER CLUSTER ID

**Input**:

**2.1 output schema**: (Entrez ID, (patient ID, diagnosis, gene value))

**2.2 output schema**: (Entrez ID, cluster ID)

The goal of this section is to sum every gene expression value assigned to each cluster ID. For example, if cluster ID *CID* is assigned Entrez IDs 1, 2, and 11, then the sum of all patient's gene expression values for gene Entrez IDs 1, 2, and 11 are assigned to *CID*.

**1**. Join: The RDD from result of **2.1** and **2.2** has the schema, respectively:

**2.1**: (Entrez ID, (patient ID, diagnosis, gene value))

**2.2**: (Entrez ID, cluster ID)

Join these two RDDs to get a new RDD with tuples schema:

(Entrez ID, ((patient ID, diagnosis, gene value), cluster ID))

**2**. Map: discard the Entrez ID and concatentate patient ID with cluster ID:

(patient ID + ";" + cluster ID, (diagnosis, gene value))

the new key here is a string concatenation of the patient ID and cluster ID, delimited by a semicolon.

**3**. Filter: remove tuples with missing gene expression value (values that are the empty string when read in from the file). The tuple schema becomes:

(patient ID + ";" + cluster ID, (diagnosis, gene value that exists))

**4**. Map: convert string gene values to float:

(patient ID + “;” + cluster ID, (diagnosis, float gene value that exists))

**5**. ReduceByKey: sum all gene values with same *(patient ID + “;” + cluster ID)* key to get:

(patientID + “;” + cluster ID, (diagnosis, gene value sum))

**6**. Map: remove patient ID, since they are no longer required:

(diagnosis, (cluster ID, gene value sum))

The new key is the diagnosis, with a value of tuple (cluster ID, gene value sum).

**7**. Map: concatenate diagnosis with cluster ID to create the new RDD:

ad_nci_rdd ← (“ad;4”, gene value sum), (“nci;5”, gene value sum), ...

**Output schema**:

ad_nci_rdd ← (Diagnosis + “;” + cluster ID, gene sum for a patient), ...

## 2.4 CLUSTER GENE MEAN

**Input**:

**2.3 output schema**:

ad_nci_rdd ← (Diagnosis + “;” + cluster ID, gene sum for a patient), …

For this section, we calculate the mean across different rows or patients of the same cluster ID. There should be at least one patient with AD diagnosis and one with NCI diagnosis, otherwise the program will exit without calculating top-k t-scores. The mean calculation uses the two-pass algorithm for simplicity, and because the mean is a required output.

**1**. Map: we want to count the size of each cluster ID. So we assign each cluster ID a value 1. We do this for both *ad_rdd* and *nci_rdd*. The result looks somewhat like this:

(Diagnosis + “;” + cluster ID, gene sum for a patient), … → (Diagnosis + “;” + cluster ID, 1), ...

**2**. ReduceByKey: we want to count the size of each cluster ID.

(Diagnosis + “;” + cluster ID, 1), ... → (Diagnosis + “;” + cluster ID, size), …

where *size* is the number of tuples with key *cluster ID.* We create two new RDDs *ad_size* and *nci_size* with following tuple schema:

ad_nci_size_rdd ← (Diagnosis + “;” + cluster ID, size), …

**3**. ReduceByKey: the input to this step is the *ad_nci_rdd* from section **2.3**. Accumulate the sums for each cluster ID, which will be used to calculate the mean along with RDD for cluster sizes from step **2**.

ad_nci_sum_rdd ← (Diagnosis + “;” + cluster ID, sum of cluster sums), ...

where *sum of cluster sums* is the sum of all gene expression value sums for all patients or rows for that cluster ID.

**4**. Join: input is RDD from step **2** and step **3**, which we join together. The schema of resulting RDD looks like this:

(Diagnosis + “;” + cluster ID, (sum of cluster sums, size)), …

**5**. Map: finally we calculate the mean.

ad_nci_mean_rdd ← (Diagnosis + “;” + cluster ID, sum of cluster sums/size), ...

**Output schema**:

ad_nci_size_rdd ← (Diagnosis + “;” + cluster ID, size), …

ad_nci_mean_rdd ← (Diagnosis + “;” + cluster ID, mean), ...

## 2.5 AD AND NCI T-TEST

**Input**:

**2.3 output schema:**

ad_nci_rdd ← (Diagnosis + “;” + cluster ID, gene sum for a patient), ...

**2.4 output schemas**:

ad_nci_size_rdd ← (Diagnosis + “;” + cluster ID, size), …

ad_nci_mean_rdd ← (Diagnosis + “;” + cluster ID, mean), …

This section calculates the t-test between two diagnosis groups, AD and NCI. First we calculate the population variance. The input is the output from **2.3**, where the key is the concatenation of diagnosis and cluster ID, delimited by a semicolon, and the value is the *gene sum for a patient,* the sum of gene values columns for a row or patient for that cluster ID, which I will call *GS* for simplicity. The other input is the output from **2.4** with values containing the size and mean of each cluster and diagnosis, with the same key as the output of **2.3**.

**1**. Join: join the RDD from **2.3** with the cluster mean RDD from **2.4**. And instance of the RDDs looks like this:

(D_CID, (GS, mean)), …

where *D_CID* is the concatenated diagnosis and cluster ID.

**2**. Map: perform the subtraction (GS – mean). The schema of the two RDDs become:

(D_CID, (GS, mean)), … → (D_CID, (GS - mean)), …

**3**. Map: square the difference from step **2**. The schema of the two RDDs become:

(D_CID, (GS - mean)), … → (D_CID, (GS – mean)^2), …

**4**. ReduceByKey: sum the difference squared, giving us the numerator of the variance per cluster ID:

(D_CID, Sum((GS – mean)^2)), …

**5**. Join: join the RDD from step **4** with *ad_nci_size* from step **2** of section **2.4**.

(D_CID, (Sum((GS – mean)^2), size)), …

This RDD essentially has the schema:

(D_CID, (variance numerator, variance denominator)), …

**6**. Map: divide the variance numerator by (variance denominator)^2 to get schema:

(D_CID, variance/size), …

**7**. Filter: remove diagnosis part of key, leaving only the cluster ID:

(CID, variance/size), …

**8**. ReduceByKey: calculate the denominator of t-test score by joining and adding corresponding *variance/size* and square rooting:

(CID, sqrt(ad_variance/ad_size + nci_variance/nci_size)), …

**9**. Filter: we want to eliminate any tuple containing denominator radicant of 0 to avoid divide-by-zero error.

(cluster ID, t-score denominator radicant) → (cluster ID, non-zero t-score denominator radicant)

tscore_denominator_radicant_rdd ← (cluster ID, non-zero t-score denominator radicant)

**10**. Filter: get the numerator of the t-score by splitting the *ad_nci_mean_rdd*:

ad_mean_rdd ← (AD_CID, ad_mean)

nci_mean_rdd ← (NCI_CID, nci_mean)

**11**. Map: remove diagnosis from key:

ad_mean_rdd ← (CID, ad_mean)

nci_mean_rdd ← (CID, nci_mean)

**12**. Join: join *ad_mean_rdd* and *nci_mean_rdd* together:

(CID, (ad_mean, nci_mean))

**13**. Map: take the difference of ad_mean and nci_mean:

numerator_rdd ← (CID, (ad_mean – nci_mean))

The above has schema:

(cluster ID, t-score numerator)

**13**. Join: join the two RDDs from step **8** and step **13**:

(CID, (ad_mean – nci_mean, sqrt((ad_variance/size) + (nci_variance/size))))

which essentially has the schema:

(CID, (t-test numerator, t-test denominator))

**14**. Map: finally we divide the t-test numerator by the t-test denominator:

(CID, (t-test numerator, t-test denominator)) → (CID, t-test numerator/t-test denominator)

The output is the t-test across patients diagnosed with AD and patients diagnosed with NCI for each cluster ID, with final schema (CID, t-score).

**Output schema**: (cluster ID, t-score between AD and NCI)

## 2.6 TOP-K T-SCORES

**Input**:

**2.5 output schema:** (cluster ID, t-score between AD and NCI)

Here we select the top-k t-score out of all cluster IDs.

**1**. Map: the input is the output RDD from section **2.4**. Here I reverse key-value order:

(CID, t-score) → (t-score, CID)

**2**. SortByKey: sort by key in descending order. One machine's RDD would have:

(t-score1, CID1),

(t-score2, CID2), …

where t-score1 > t-score2.

**3**. ZipWithIndex: assign each tuple an index after being sorted.

(1, (t-score1, CID1)),

(2, (t-score2, CID2)), …

**4**. Filter: filter for top-k.

(1, (t-score1, CID1)), …

(k, (t-scorek, CIDk))

**5**. Map: finally get rid of row index:

(t-score1, CID1), …

(t-scorek, CIDk)

The final output is an RDD of top-k t-scores.

**Output schema**: (t-score, CID), …

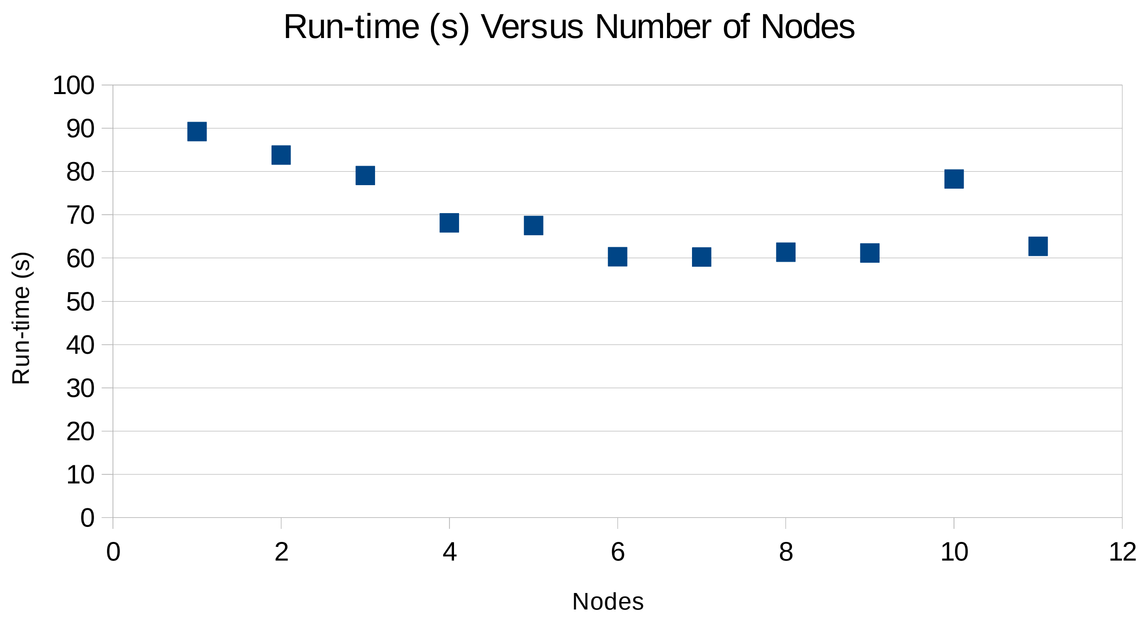
## 3. RESULT

| cluster ID | T-score | AD mean | NCI mean | AD pop. std | NCI pop. std |
|---|---|---|---|---|---|
| '5401' | 7.4232190278 | 148.8417 | 129.3198333333 | 29.0072360555 | 23.7849190893 |
| '2762' | 6.3547837592 | 37.95148 | 32.4513333333 | 10.4206569196 | 6.8694271474 |
| '3025' | 5.6588419892 | 13.69632 | 11.0393333333 | 5.3041291068 | 4.1198829002 |
| '2378' | 5.642089815 | 25.27054 | 22.4927222222 | 5.5093239904 | 4.3739605534 |
| '5467' | 5.3734683905 | 62.57063 | 52.3986111111 | 23.3419782086 | 14.3261454366 |
| '2842' | 5.3276699136 | 59.82327 | 52.8748888889 | 14.5506405805 | 11.6308252544 |
| '5464' | 5.3213614436 | 64.96345 | 54.6193888889 | 23.965165793 | 14.7164280148 |
| '3028' | 5.254801415 | 16.09206 | 13.3902777778 | 5.7490049251 | 4.5723159012 |
| '3532' | -4.9751818134 | 21.5491 | 25.2412222222 | 7.2310705472 | 7.544821672 |
| '5475' | 4.9102767009 | 81.88296 | 72.4663888889 | 23.4334015995 | 14.7899477786 |

Table 3

The top 10 ranked t-test scores, the AD and NCI means and their population standard deviations are shown in table 3, ordered by the magnitude of the t-test scores.

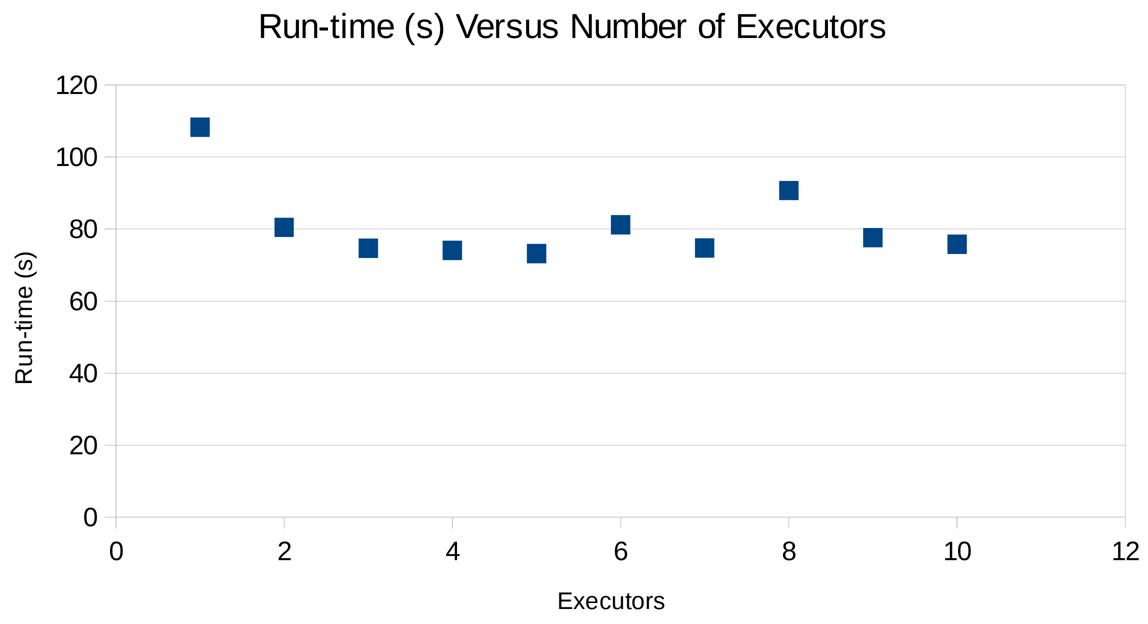
The following graphs represent general behavior of the run-time versus different parameters, and not as the actual performance of the application, since the run-time depends on machine specifications, and the program has since been better optimized since these data were collected. All of the measurements uses an HDFS Rosmap file and gene cluster file of block size 1.



Graph 1: running on Amazon Web Service ElasticMapReduce cluster.

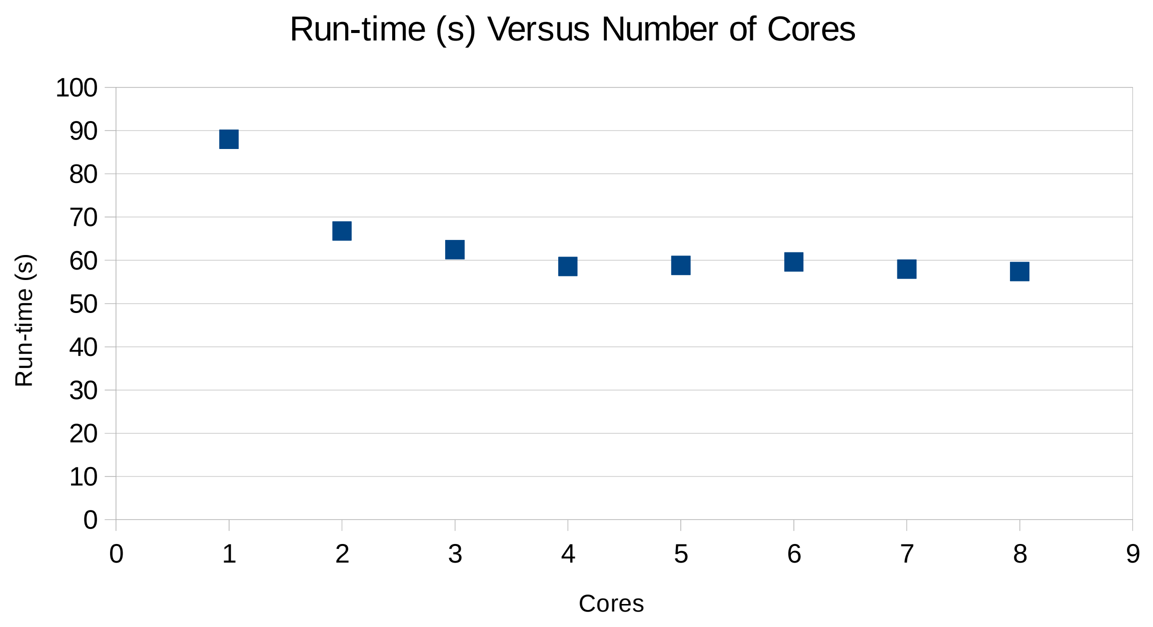
Graph 1 shows the run time per node with YARN master in cluster deploy mode, including the master node, on a Amazon Web Service (AWS) ElasticMapReduce cluster (EMR) [2]. Here the number of executors and its cores are not manually set. Increasing the number of executors don't always give optimal results. The nodes are *m3.xlarge* instances [3] with 8 virtual CPU cores, 15GB memory, and 80GB SSD. The dynamic allocation feature of Spark [4], enabled by default in an AWS EMR cluster, is disabled here because it is into required for this problem. Since with it enabled the number of executors scale up by workload and not number of nodes, but the Rosmap file, the largest file, is only 46.2MB. The dynamic allocation property is only useful when multiple Spark applications are running.

Sometimes using the same number of nodes gives run time differences as large as 10 seconds, such as sudden increase for the run-time for 10 nodes from Graph 1. Increasing the number of nodes can increase node communication overhead, and you would see a run time increase. YARN itself also introduces several seconds of overhead compared to running the Spark application using Spark's local master. Ideally, we shouldn't see any significant improvements for such a small dataset in a distributed environment, and so local master performed better due to less communication overhead. Eventually, as the number of nodes keep increasing, the performance stagnates and may actually sporadically increase due to node communication overhead, especially as when each map steps start to take less time then the time to communicate between nodes, creating a bottleneck where you no longer see performance gains. The greatest performance increase for this experiment occurs between 1 and 4 nodes. Furthermore, as I increased the number of nodes, the number of YARN containers did not increase beyond 3, since the dataset is relatively small and not by any means would it be considered big data.



Graph 2: using YARN pseudo-distributed mode.

Graph 2 shows the number of one-core executors versus run-time in pseudo-distributed mode on a laptop with Intel core i5-5300U 2.3GHz processor with 2 physical cores and 4 threads (8 virtual cores). The run-time seems to stagnates earlier, similar to experiment on AWS EMR clusters. Most noteworthy is how much longer the average run-time becomes in a pseudo-distributed environment.



Graph 3: running on local master.

Graph 3 shows run-time performance using local master on the same laptop from Graph 2. Running on local master gives similar eventual performance compared to running the application on an AWS EMR cluster.

## 3.1. MACHINE LEARNING DIAGNOSIS

The machine learning part is contained in the *diagnose.py* module and uses gradient boosted trees binary classifier from Spark's RDD-based API [5] to diagnose whether a person, given their gene expression profile with Entrez IDs, is diagnosed as no-cognitive-impairment or has Alzheimer's Disease. The input is a CSV file of the schema (patient ID, g1, g2, g3, …), where *g1, g2, g3, …* are instances of the patient's gene expression values ordered by Entrez IDs. This file is then read into an RDD and processed into a libSVM file. Currently, the model supports only libSVM files, and there is a module for converting to libSVM file called "*labeled_to_libsvm.py*" in the project directory. The choice of model is under the assumption that there is no relationship between how the clusters of Entrez IDs are formed in the gene cluster file, so the features can be treated as heterogeneous. The other reason is that the available dataset is labeled, so supervised learning is a good choice.

Initial tests using 3-fold cross validation and top 1 cluster as feature obtained 66% accuracy for 100 tree iterations with 149 true positives, 117 true negatives, 63 false positives, and 74 false negatives. However, increasing to more iterations did not seem to improve results. After the data set has been split up into two groups of patients, those diagnosed with AD and those diagnosed with NCI, the size of AD clusters are 223 and size of NCI clusters are 180, which gives a degree of freedom of 401 using pooled degrees of freedom formula for independent means. Using 401 degrees of freedom and a t-score of 3.5, the two-tailed p-value is 0.000517, which is statistically significant for rejecting the null hypothesis stating that there is no difference between the selected cluster means between AD and NCI. In other words, these clusters of genes act as good indicators of diagnosis for our binary classification of either AD or NCI. Of the 1176 cluster t-score tuples between AD and NCI, 153 has a t-score greater than 3.5. For the 153 clusters, the model hovers around a 63% 3-fold cross-validation accuracy with 160 true positives, 91 true negatives, 89 false positives, and 63 false negatives with 100 tree iterations. Even with more iterations, the 3-fold cross validation accuracy does not get any better. With 400 clusters and 100 iterations, one 3-fold cross validation obtained an accuracy of 62% with 146 true positives, 102 true negatives, 78 false positives, and 76 false negatives. So it seems increasing clusters did not improve accuracy. Different pre-computed models reside in "*models/*" folder with naming schema "*gbt_xc_yi*" for *x* clusters and *y* iterations. E.g., "*gbt_200c_50i*" was built using 200 feature clusters and 50 tree iterations. Increasing to too many iterations may cause stack overflow error on less advanced machines. Although the experiments seem to show fewer clusters give better accuracy, but when the model is tested on the same data it was trained on, the models trained on more clusters were far more accurate. In general, it is safer to say that a few hundred top-k t-score feature clusters, which are all statistically significant, should provide good results.

## 4. DISCUSSION

The program does error checking, such as filtering for missing data, which can be removed to improve performance only by a constant amount if we can assume those errors will never occur. There is also an attempt to place code modularity ahead of run-time when plausible. The program currently uses the two-pass algorithm to calculate the variance, which may be less numerically stable than other algorithms for calculating variance. I have tried implementing the computational formula that separates the one sum into two sums, but abandoned it since it caused a positive infinity error when calculating the sum of all samples squared. I chose the two pass algorithm also for simplicity, since the mean is also a required output and needs to be calculated anyway. The program also doesn't set the minimum partition size. Setting it would further improve performance somewhat, but not by much. The algorithm itself may be further optimized if mean and variance outputs can be ignored because the run-time measurements include the time taken to calculate these output RDDs. The program assumes the top-k value may be large, so instead of using Spark's *take()* method, the top-k cluster IDs are selected using map-reduce. Performance can be improved if we can assume top-k is small by simply using Spark's *take()*. I also originally thought of breaking up the Rosmap RDD row-wise and instead of broadcasting the Entrez IDs and perform a transpose operation of the RDD pieces of Rosmap such that the width of each RDD's isn't allowed to grow. But this essentially imitates partitioning the RDD, which can be easily set through the block size of the Rosmap HDFS file. Since block size is fixed for the run-time measurements, I decided to broadcast the Entrez IDs instead. Finally, the application can be improved in certain areas by using better optimized Spark Dataframes API for certain queries rather than the Spark RDD API that the program currently uses, or simply switching to Java or Scala instead of using Python.

## 5. CONCLUSION

Similar performances drop-offs are seen between AWS EMR cluster, and testing locally using pseudo-distributed mode and Spark's local master. Due to the small size of the data set, distributed computing does not provide large performance gains, but nonetheless the behavior of how distributed computing performs can be seen. Furthermore, the run-time graphs here should be used as general behavior overview of how the program performs and not as its actual run-time, since run-time varies depending on the machine and the program has been further optimized since this paper was written and the data collected.

The Gradient Boosted Trees machine learning model was chosen since labeled data is available and the clusters of genes are treated as heterogeneous. Model 3-fold cross-validation accuracy did not seem to improve with more clusters and tree iterations. The 3-fold cross validation results for different amount of clusters and tree iterations hover around 61% accuracy, and the number of clusters chosen as features can be extended to a few hundred without sacrificing statistical significance of the p-value.

## REFERENCES

[1] National Center for Biotechnology Information. [Online].
Available: https://www.ncbi.nlm.nih.gov/gene. [Accessed: 18-May-2017].

[2] Amazon. Amazon EMR. [Online]. Available: https://aws.amazon.com/emr/.
[Accessed: 18-May-2017]

[3] Amazon. Amazon EC2 Instance Types. Avalaible: https://aws.amazon.com/ec2/instance-types/
[Accessed: 18-May-2017]

[4] Apache. Spark Configuration [Online].
Available: https://spark.apache.org/docs/latest/configuration.html. [Accessed 18-May-2017]

[5] Apache. Ensembles – RDD-based API [Online].
Available: https://spark.apache.org/docs/2.1.0/mllib-ensembles.html. [Accessed 20-May-2017]