Multinomial Logistic Regression with Apache Spark

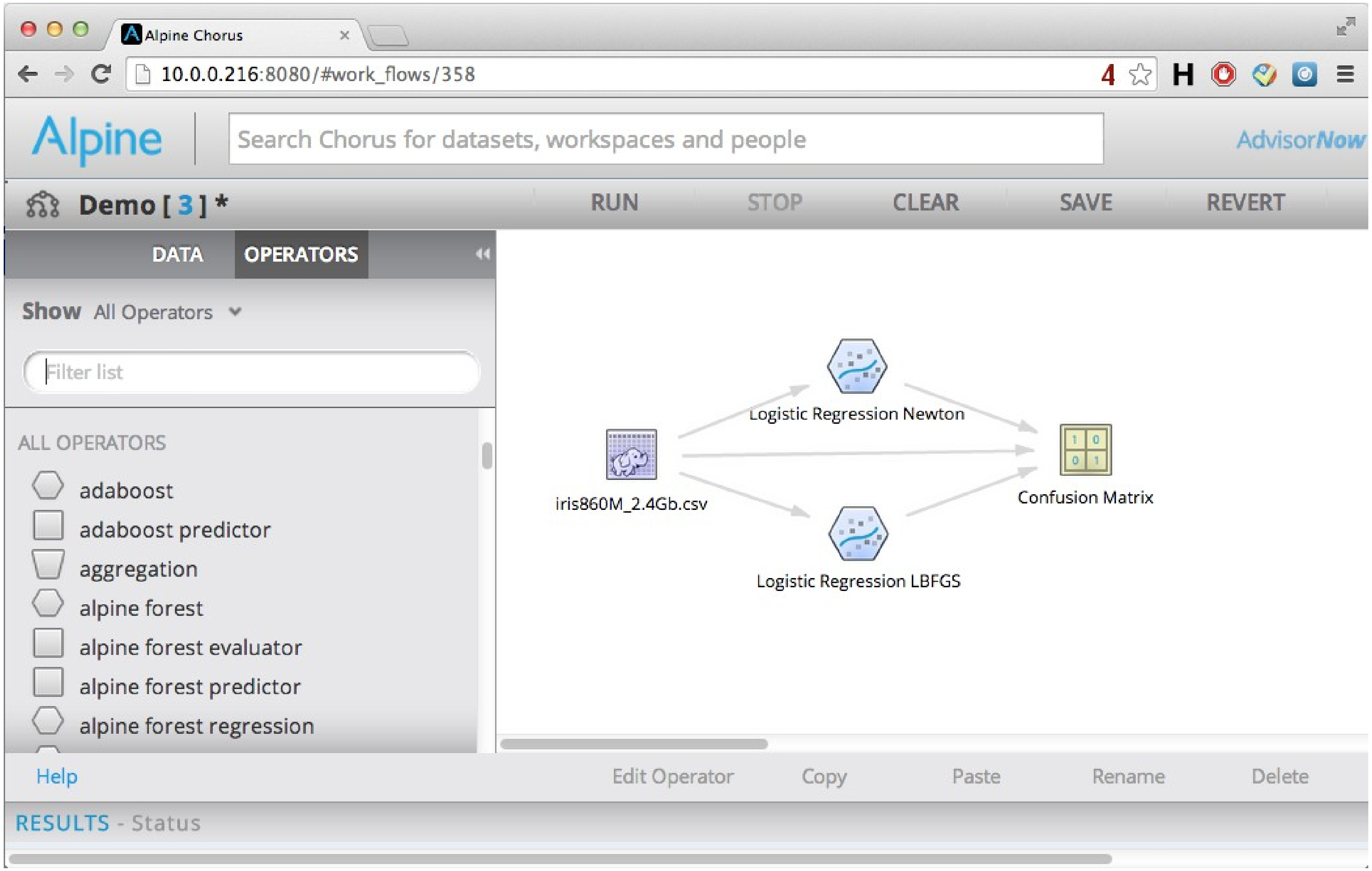
DB Tsai
dbtsai@alpinenow.com
Machine Learning Engineer

June 20th, 2014

Silicon Valley Machine Learning Meetup

Alpine

What is Alpine Data Labs doing?



- Collaborative, Code-Free, Advanced Analytics Solution for Big Data

We're open source friendly!

- Technology we're using: Scala/Java, Akka, Spray, Hadoop, Spark/SparkSQL, Pig, Sqoop
- Our platform runs against different flavors of Hadoop distributions such as Cloudera, Pivotal Hadoop, MapR, HortonWorks and Apache.
- Actively involved in the open source community: almost of all our newly developed algorithms in Spark will be contributed back to MLLib.
- Already committed a L-BFGS optimizer to Spark, and helped fix couple bugs. Working on multinational logistic regression, GLM, Decision Tree and a Random Forest
- In addition we are the maintainer of several open source projects including Chorus, SBT plugin for JUnit test Listener and several other projects.

We're hiring!

- Machine Learning Engineer
- Data Scientist
- UI/UX Engineer
- Front-end Engineer
- Infrastructure Engineer
- Back-end Engineer
- Automation Test Engineer

Shoot me an email at
dbtsai@alpinenow.com

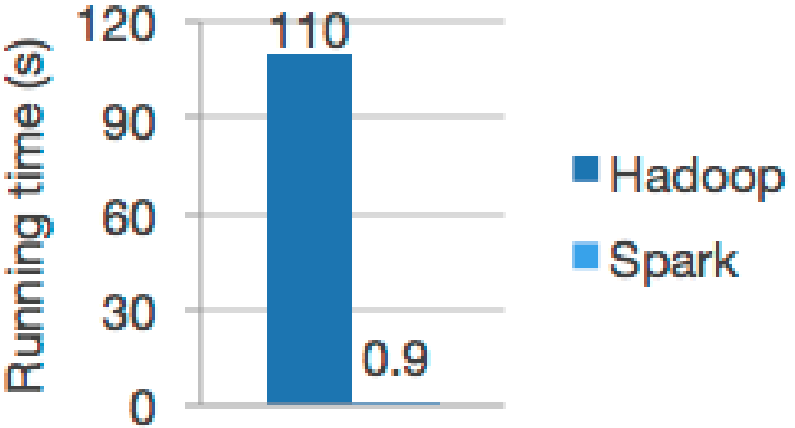
Machine Learning with Big Data

- Hadoop MapReduce solutions

- MapReduce is scaling well for batch processing
- Lots of machine learning algorithms are iterative by nature.
- There are lots of tricks people do, like training with subsamples of data, and then average the models. Why big data with approximation?

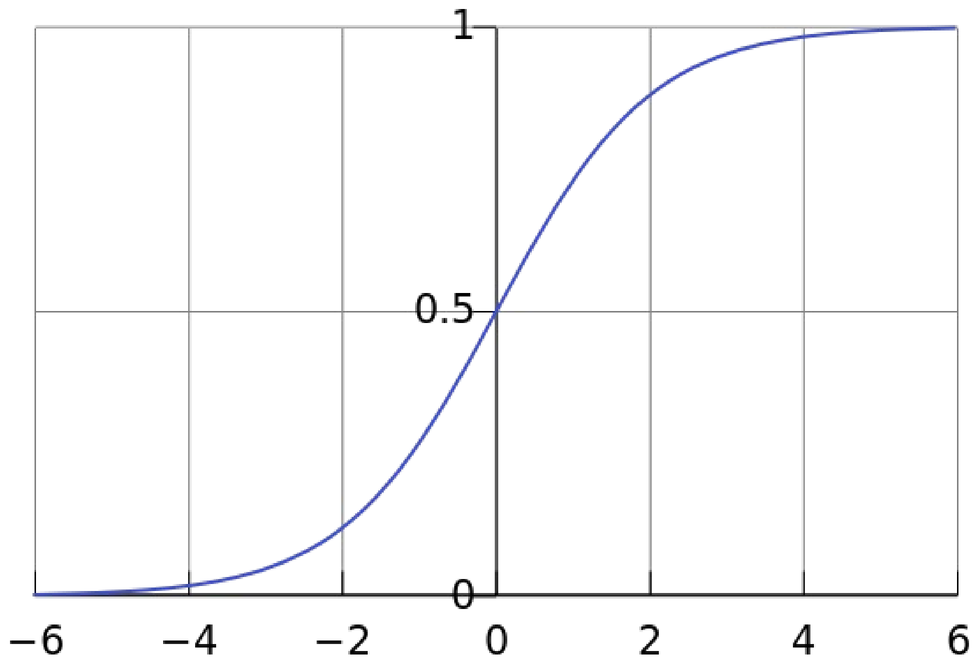
Spark Lightning-fast cluster computing

- Empower users to iterate through the data by utilizing the in-memory cache.
- Logistic regression runs up to 100x faster than Hadoop M/R in memory.

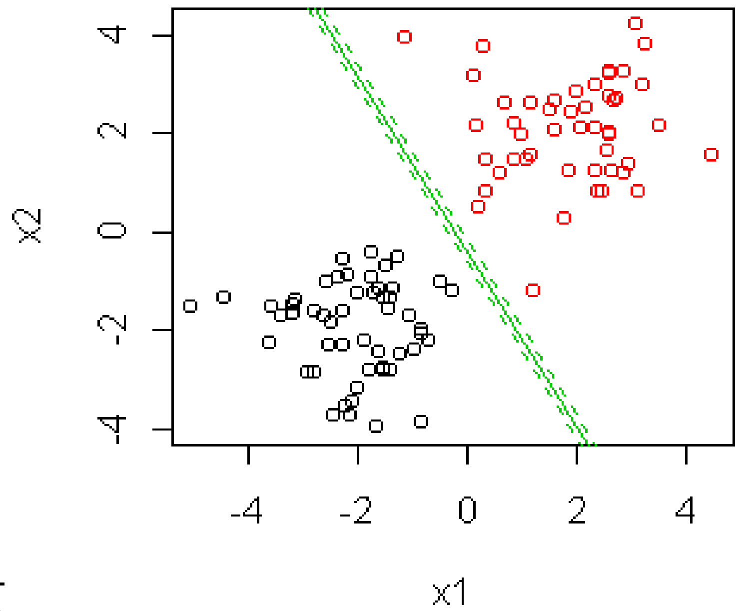


- We're able to train
  exact model without doing any approximation.

Binary Logistic Regression



$$d = \frac{ax_1 + bx_2 + cx_0}{\sqrt{a^2+b^2}} \quad where\ x_0 = 1$$



$$P(y=1|\vec{x},\vec{w}) = \frac{\exp(d)}{1+\exp(d)} = \frac{\exp(\vec{x}\,\vec{w})}{1+\exp(\vec{x}\,\vec{w})}$$

$$P(y=0|\vec{x},\vec{w}) = \frac{1}{1+\exp(\vec{x}\,\vec{w})}$$

$$\log\frac{P(y=1|\vec{x},\vec{w})}{P(y=0|\vec{x},\vec{w})} = \vec{x}\,\vec{w}$$

$$w_0 = \frac{c}{\sqrt{a^2+b^2}} \quad where\ w_0\ is\ called\ as\ intercept$$

$$w_1 = \frac{a}{\sqrt{a^2+b^2}}$$

$$w_2 = \frac{b}{\sqrt{a^2+b^2}}$$

Training Binary Logistic Regression

- Maximum Likelihood estimation
  From a training data $X = (\vec{x}_1, \vec{x}_2, \vec{x}_3, \ldots)$
  and labels $Y = (y_{1,} y_{2,} y_{3,} \ldots)$
- We want to find $\vec{w}$ that maximizes the likelihood of data defined by

$$L(\vec{w}, \vec{x}_{1,} \ldots, \vec{x}_N) = P(y_1 | \vec{x}_1, \vec{w}) P(y_2 | \vec{x}_2, \vec{w}) \ldots P(y_N | \vec{x}_N, \vec{w})$$

- We can take log of the equation, and minimize

$$l(\vec{w}, \vec{x}) = \log P(y_1 | \vec{x}_1, \vec{w}) + \log P(y_2 | \vec{x}_2, \vec{w}) \ldots + \log P(y_N | \vec{x}_N, \vec{w})$$

  it instead. The Log-Likelihood becomes the loss function.

Optimization

- First Order Minimizer
  Require loss, gradient of loss function
  - Gradient Decent $\vec{w}_{n+1} = \vec{w}_n - \gamma \vec{G}$, γ is step size
  - Limited-memory BFGS (L-BFGS)
  - Orthant-Wise Limited-memory Quasi-Newton (OWLQN)
  - Coordinate Descent (CD)
  - Trust Region Newton Method (TRON)
- Second Order Minimizer
  Require loss, gradient and hessian of loss function
  - Newton-Raphson, quadratic convergence. Fast!

    $$\vec{w}_{n+1} = \vec{w}_n - H^{-1} \vec{G}$$
- Ref: Journal of Machine Learning Research 11 (2010) 3183-3234, Chih-Jen Lin et al.

Problem of Second Order Minimizer

- Scale horizontally (the numbers of training data) by leveraging Spark to parallelize this iterative optimization process.
- Don't scale vertically (the numbers of training features). Dimension of Hessian is

$$dim(H)=[(k-1)(n+1)]^2 \quad where\ k\ is\ num\ of\ class\ ,\ n\ is\ num\ of\ features$$

- Recent applications from document classification and computational linguistics are of this type.

L-BFGS

- It's a quasi-Newton method.
- Hessian matrix of second derivatives doesn't need to be evaluated directly.
- Hessian matrix is approximated using gradient evaluations.
- It converges a way faster than the default optimizer in Spark, Gradient Decent.
- We love open source! Alpine Data Labs contributed our L-BFGS to Spark, and it's already merged in Spark-1157.

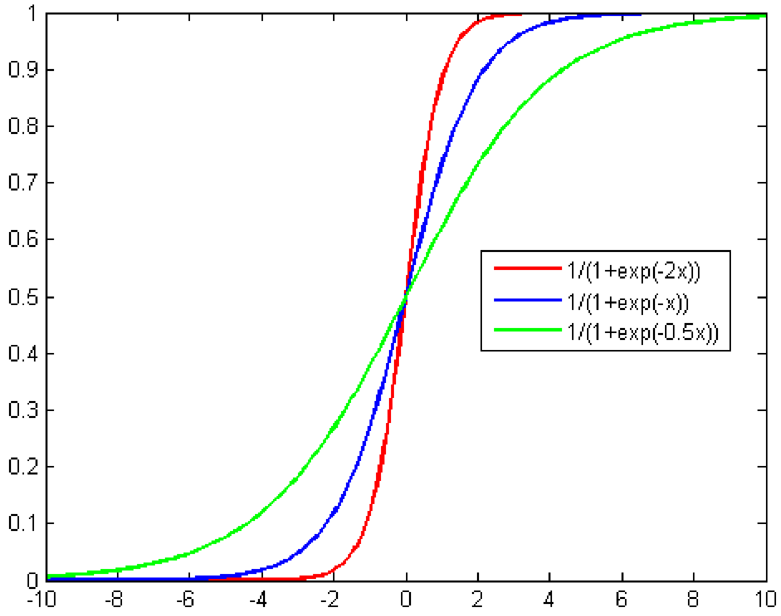
Training Binary Logistic Regression

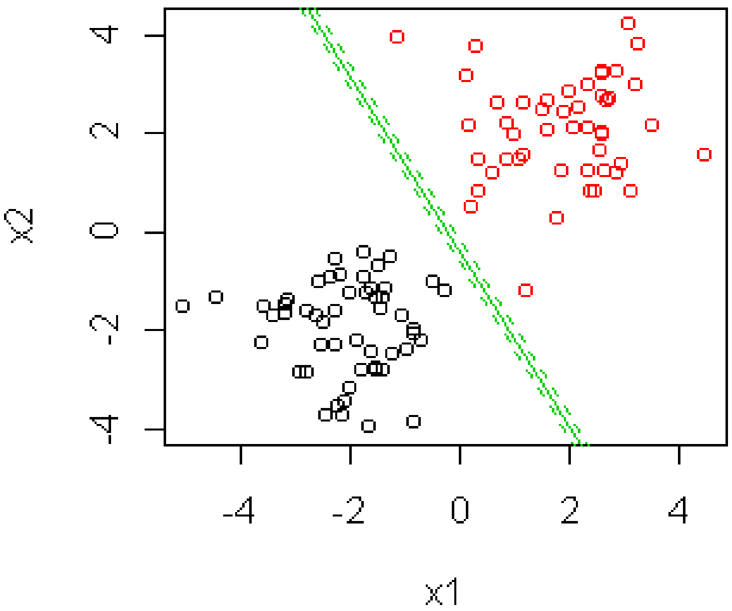
$$
\begin{aligned}
l(\vec{w},\vec{x}) &= \sum_{k=1}^{N} \log P(y_k|\vec{x}_k,\vec{w}) \\
&= \sum_{k=1}^{N} y_k \log P(y_k=1|\vec{x}_k,\vec{w}) + (1-y_k)\log P(y_k=0|\vec{x}_k,\vec{w}) \\
&= \sum_{k=1}^{N} y_k \log \frac{\exp(\vec{x}_k\vec{w})}{1+\exp(\vec{x}_k\vec{w})} + (1-y_k)\log \frac{1}{1+\exp(\vec{x}_k\vec{w})} \\
&= \sum_{k=1}^{N} y_k \vec{x}_k \vec{w} - \log(1+\exp(\vec{x}_k\vec{w}))
\end{aligned}
$$

$$
Gradient:\ G_i(\vec{w},\vec{x}) = \frac{\partial l(\vec{w},\vec{x})}{\partial w_i} = \sum_{k=1}^{N} y_k x_{ki} - \frac{\exp(\vec{x}_k\vec{w})}{1+\exp(\vec{x}_k\vec{w})} x_{ki}
$$

$$
Hessian:\ H_{ij}(\vec{w},\vec{x}) = \frac{\partial\partial l(\vec{w},\vec{x})}{\partial w_i \partial w_j} = -\sum_{k=1}^{N} \frac{\exp(\vec{x}_k\vec{w})}{(1+\exp(\vec{x}_k\vec{w}))^2} x_{ki} x_{kj}
$$

Overfitting





$$P(y=1|\vec{x},\vec{w})=\frac{\exp(zd)}{1+\exp(zd)}=\frac{\exp(\vec{x}\,\vec{w})}{1+\exp(\vec{x}\,\vec{w})}$$

Regularization

- The loss function becomes

$$l_{total}(\vec{w}, \vec{x}) = l_{model}(\vec{w}, \vec{x}) + l_{reg}(\vec{w})$$

- The loss function of regularizer doesn't depend on data. Common regularizers are
  - L2 Regularization: $l_{reg}(\vec{w}) = \lambda \sum_{i=1}^{N} w_i^2$
  - L1 Regularization: $l_{reg}(\vec{w}) = \lambda \sum_{i=1}^{N} |w_i|$

$$\vec{G}(\vec{w}, \vec{x})_{total} = \vec{G}(\vec{w}, \vec{x})_{model} + \vec{G}(\vec{w})_{reg}$$

$$\bar{H}(\vec{w}, \vec{x})_{total} = \bar{H}(\vec{w}, \vec{x})_{model} + \bar{H}(\vec{w})_{reg}$$

- L1 norm is not differentiable at zero!

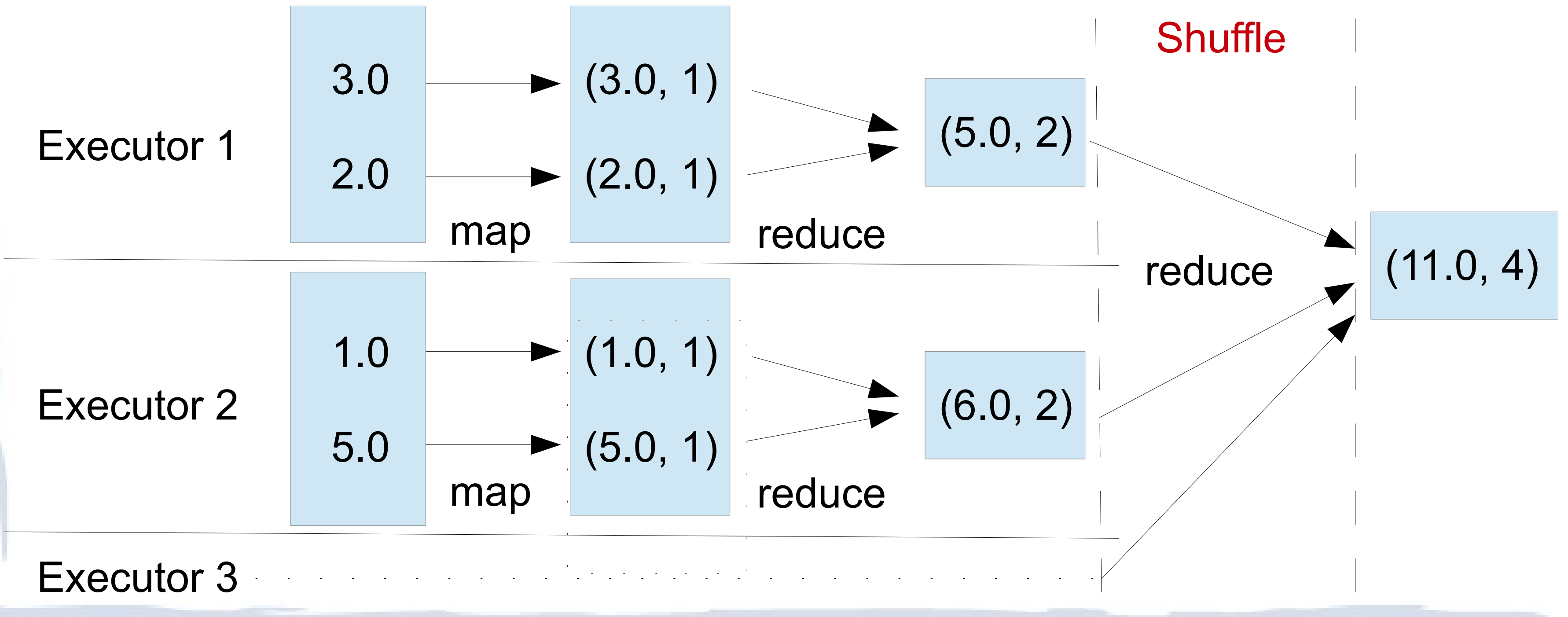
Mini School of Spark APIs

- map(func) : Return a new distributed dataset formed by passing each element of the source through a function func.
- reduce(func) : Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

  This is O(n) operation where n is number of partition. In SPARK-2174, treeReduce will be merged for O(log(n)) operation.

Example – compute the mean of numbers.

```
val input = sc.textFile("hdfs://...")
val cachedRDD = input.cache()
val (sum, counts) = cachedRDD.map(
  line => (line.toDouble, 1)
).reduce(
  (a, b) => (a._1 + b._1, a._2 + b._2)
)
val mean = sum / counts
```

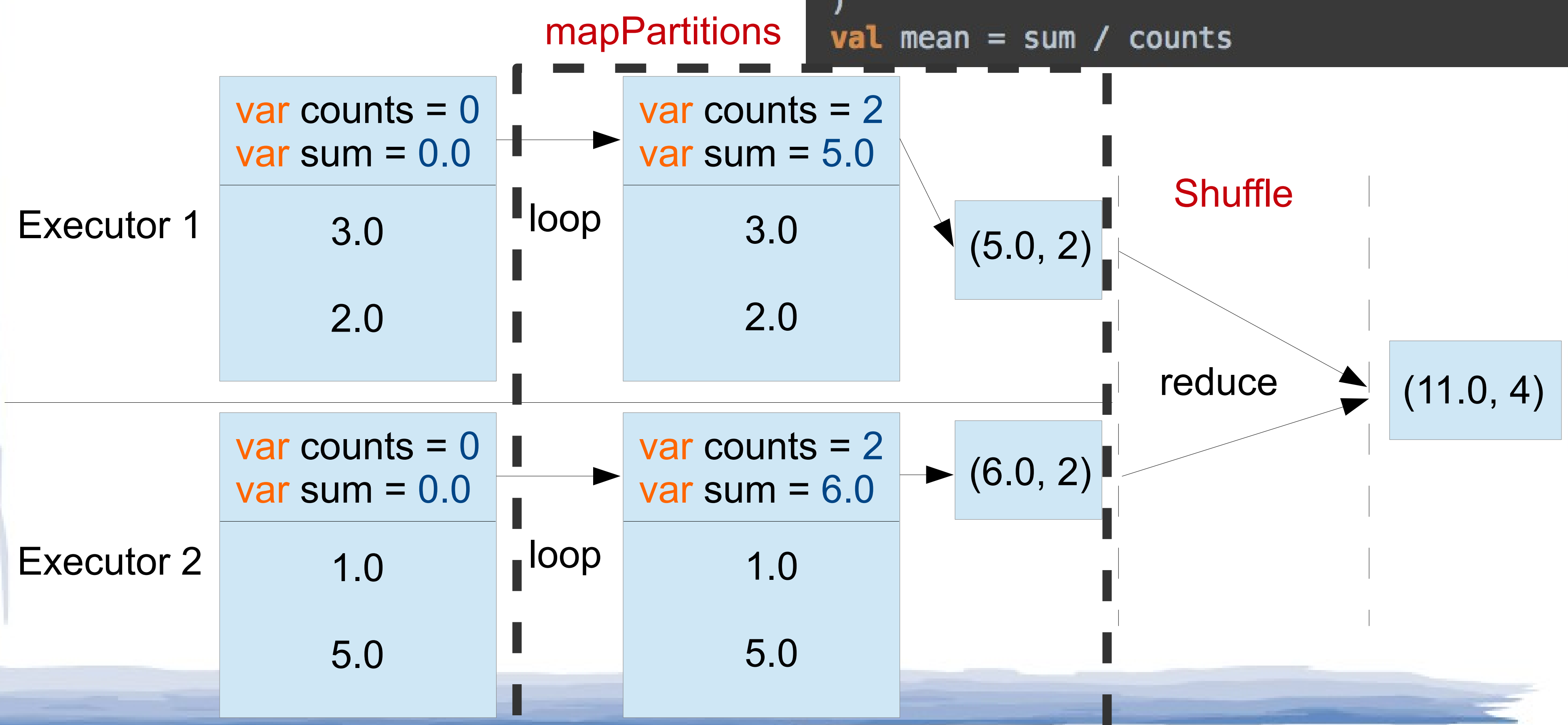


Mini School of Spark APIs

- mapPartitions(func) : Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator[T] => Iterator[U] when running on an RDD of type T.
- This API allows us to have global variables on entire partition to aggregate the result locally and efficiently.

Better Mean of Numbers Implementation

```
val input = sc.textFile("hdfs://...")
val cachedRDD = input.cache()
val (sum, counts) = cachedRDD.mapPartitions(
  iter => {
    var counts = 0
    var sum = 0.0
    for(x <- iter) {
      sum += x
      counts += 1
    }
    Iterator((sum, counts))
  }
).reduce(
  (a, b) => (a._1 + b._1, a._2 + b._2)
)
val mean = sum / counts
```



More Idiomatic Scala Implementation

- aggregate(zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U

```scala
val input = sc.textFile("hdfs://...")
val cachedRDD = input.cache()

case class Aggregator(var sum: Double, var counts: Int)

val aggregator = cachedRDD.aggregate(Aggregator(0.0, 0))(
  seqOp = (aggregator, value) => {
    aggregator.sum += value.toDouble
    aggregator.counts += 1
    aggregator
  },
  combOp = (aggregatorA, aggregatorB) => {
    aggregatorA.sum += aggregatorB.sum
    aggregatorA.counts += aggregatorB.counts
    aggregatorA
  }
)
val mean = aggregator.sum / aggregator.counts
```

zeroValue is a neutral "zero value" with type U for initialization. This is analogous to var counts = 0
var sum = 0.0
in previous example.

seqOp is function taking (U, T) => U, where U is aggregator initialized as zeroValue. T is each line of values in RDD. This is analogous to mapPartition in previous example.

combOp is function taking (U, U) => U. This is essentially combing the results between different executors. The functionality is the same as reduce(func) in previous example.

Approach of Parallelization in Spark

Time

Spark Driver JVM	Spark Executor 1 JVM	Spark Executor 2 JVM
1) Find available resources in cluster, and launch executor JVMs. Also initialize the weights.		
2) Ask executors to load the data into executors' JVMs	2) Trying to load the data into memory. If the data is bigger than memory, it will be partial cached. (The data locality from source will be taken care by Spark)	
3) Ask executors to compute loss, and gradient of each training sample (each row) given the current weights. Get the aggregated results after the Reduce Phase in executors.	3) Map Phase: Compute the loss and gradient of each row of training data locally given the weights obtained from the driver. Can either emit each result or sum them up in local aggregators. 4) Reduce Phase: Sum up the losses and gradients emitted from the Map Phase	
5) If the regularization is enabled, compute the loss and gradient of regularizer in driver since it doesn't depend on training data but only depends on weights. Add them into the results from executors.	Taking a rest!	
6) Plug the loss and gradient from model and regularizer into optimizer to get the new weights. If the differences of weights and losses are larger than criteria, GO BACK TO 3)	Taking a rest!	
7) Finish the model training!	Taking a rest!	

Step 3) and 4)

- This is the implementation of step 3) and 4) in MLlib before Spark 1.0

```
val (gradientSum, lossSum) = data.map {
  case (y, features) =>
    val featuresCol = new DoubleMatrix(features.length, 1, features:_*)
    val (grad, loss) = gradient.compute(featuresCol, y, weights)
    (grad, loss)
}.reduce((a, b) => (a._1.addi(b._1), a._2 + b._2))
```

- gradient can have implementations of Logistic Regression, Linear Regression, SVM, or any customized cost function.
- Each training data will create new "grad" object after gradient.compute.

Step 3) and 4) with mapPartitions

```scala
val (gradientSum, lossSum) = data.mapPartitions {
  xIterator => {
    var lossPartitionSum = 0.0
    var gradientPartitionSum = None : Option[DoubleMatrix]
    var featuresVector = None : Option[DoubleMatrix]

    for(x <- xIterator) {
      x match {
        case (y, features) => {
          featuresVector.getOrElse {
            // Initialize for the first access, and assign it to featuresVector.
            featuresVector = Some(new DoubleMatrix(features.length, 1))
            featuresVector.get
          }.data = features

          val (grad, loss) = gradient.compute(featuresVector.get, y, weights)

          lossPartitionSum += loss

          gradientPartitionSum.getOrElse {
            // Initialize for the first access, and assign it to gradientPartitionSum.
            gradientPartitionSum = Some(new DoubleMatrix(features.length, 1))
            gradientPartitionSum.get
          }.addi(grad)
        }
      }
    }
    Iterator((gradientPartitionSum.get, lossPartitionSum))
  }
}.reduce((a, b) => (a._1.addi(b._1), a._2 + b._2))
```

Step 3) and 4) with aggregate

- This is the implementation of step 3) and 4) in MLlib in coming Spark 1.0

```
val (gradientSum, lossSum) = data.aggregate((BDV.zeros[Double](weights.size), 0.0))(
    seqOp = (c, v) => (c, v) match { case ((grad, loss), (label, features)) =>
      val l = gradient.compute(features, label, weights, Vectors.fromBreeze(grad))
      (grad, loss + l)
    },
    combOp = (c1, c2) => (c1, c2) match { case ((grad1, loss1), (grad2, loss2)) =>
      (grad1 += grad2, loss1 + loss2)
    })
```

- No unnecessary object creation! It's helpful when we're dealing with large features training data. GC will not kick in the executor JVMs.

Extension to Multinomial Logistic Regression

- In Binary Logistic Regression

$$\log \frac{P(y=1|\vec{x},\vec{w})}{P(y=0|\vec{x},\vec{w})} = \vec{x}\vec{w}$$

- For K classes multinomial problem where labels ranged from [0, K-1], we can generalize it via

$$\log \frac{P(y=1|\vec{x},\bar{w})}{P(y=0|\vec{x},\bar{w})} = \vec{x}\vec{w}_1$$
$$\log \frac{P(y=2|\vec{x},\bar{w})}{P(y=0|\vec{x},\bar{w})} = \vec{x}\vec{w}_2$$
$$\dots$$
$$\log \frac{P(y=K-1|\vec{x},\bar{w})}{P(y=0|\vec{x},\bar{w})} = \vec{x}\vec{w}_{K-1}$$

$$\Longrightarrow$$

$$P(y=0|\vec{x},\bar{w}) = \frac{1}{1+\sum_{i=1}^{K-1}\exp(\vec{x}\vec{w}_i)}$$
$$P(y=1|\vec{x},\bar{w}) = \frac{\exp(\vec{x}\vec{w}_2)}{1+\sum_{i=1}^{K-1}\exp(\vec{x}\vec{w}_i)}$$
$$\dots$$
$$P(y=K-1|\vec{x},\bar{w}) = \frac{\exp(\vec{x}\vec{w}_{K-1})}{1+\sum_{i=1}^{K-1}\exp(\vec{x}\vec{w}_i)}$$

- The model, weights $\bar{w} = (\vec{w}_1, \vec{w}_2, \dots, \vec{w}_{K-1})^T$ becomes (K-1)(N+1) matrix, where N is number of features.

Training Multinomial Logistic Regression

Define:
$$\alpha(y_k)=1 \quad if\ y_k=0$$
$$\alpha(y_k)=0 \quad if\ y_k\neq 0$$

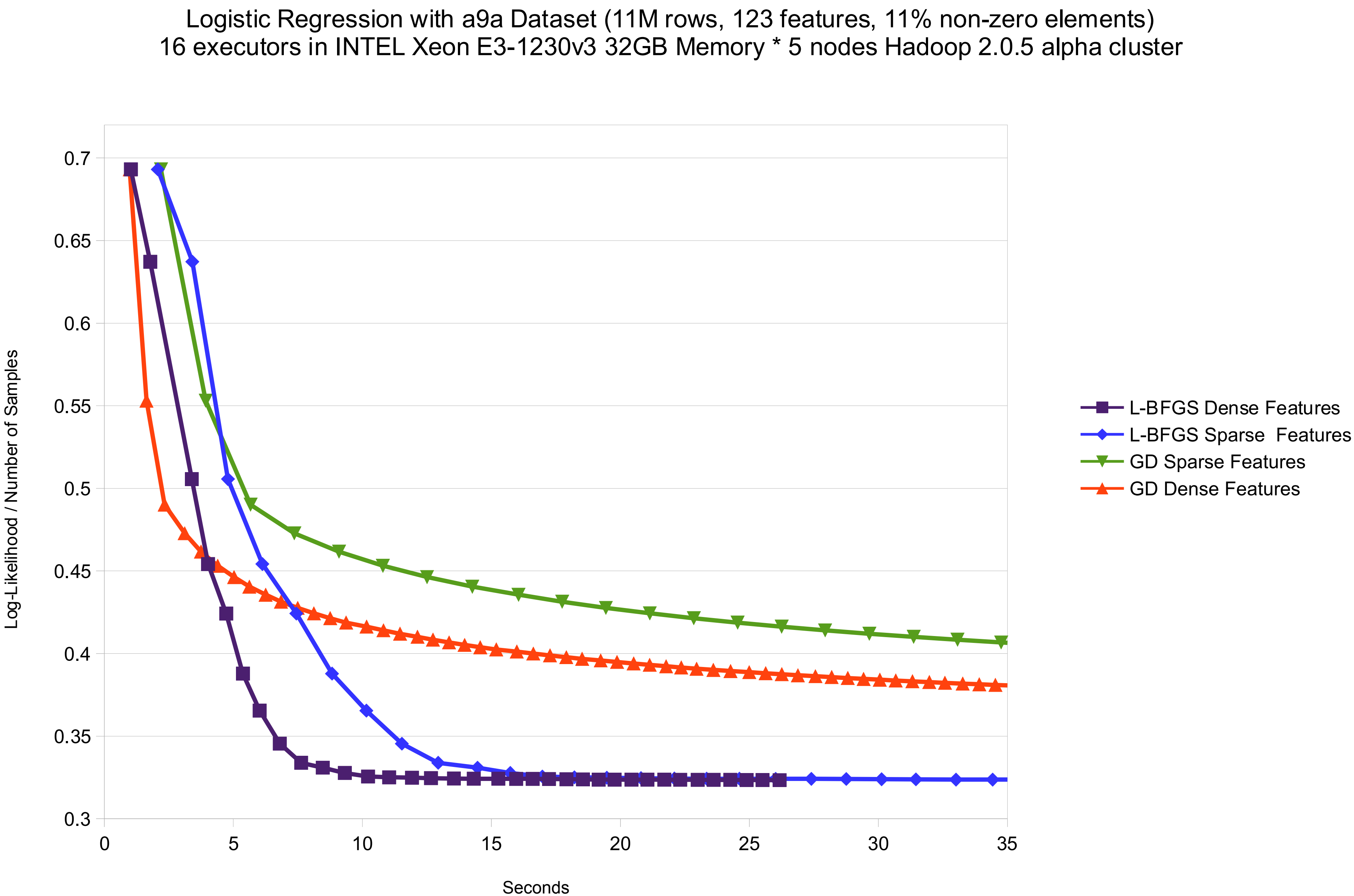
$$
\begin{aligned}
l(\bar{w},\vec{x}) &= \sum_{k=1}^{N} \log P(y_k|\vec{x}_k,\bar{w}) \\
&= \sum_{k=1}^{N} \alpha(y_k)\log P(y=0|\vec{x}_k,\bar{w}) + (1-\alpha(y_k))\log P(y_k|\vec{x}_k,\bar{w}) \\
&= \sum_{k=1}^{N} \alpha(y_k)\log \frac{1}{1+\sum_{i=1}^{K-1}\exp(\vec{x}\,\vec{w}_i)} + (1-\alpha(y_k))\log \frac{\exp(\vec{x}\,\vec{w}_{y_k})}{1+\sum_{i=1}^{K-1}\exp(\vec{x}\,\vec{w}_i)} \\
&= \sum_{k=1}^{N} (1-\alpha(y_k))\vec{x}\,\vec{w}_{y_k} - \log\left(1+\sum_{i=1}^{K-1}\exp(\vec{x}\,\vec{w}_i)\right)
\end{aligned}
$$

Note that the first index "i" is for classes, and the second index "j" is for features.

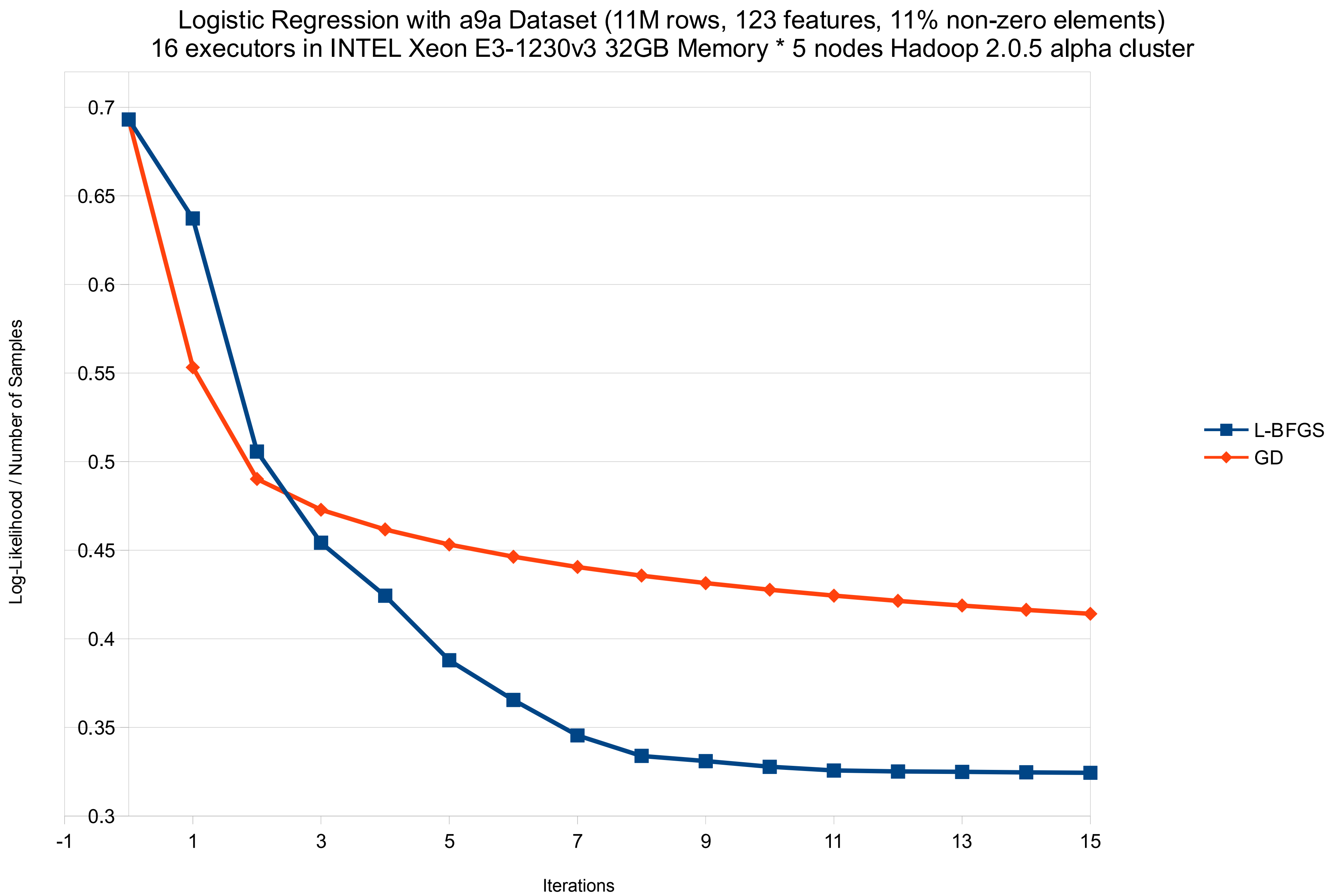
$$Gradient:\ G_{ij}(\bar{w},\vec{x}) = \frac{\partial l(\bar{w},\vec{x})}{\partial w_{ij}} = \sum_{k=1}^{N} (1-\alpha(y_k))\, x_{kj}\,\delta_{i,y_k} - \frac{\exp(\vec{x}_k\,\vec{w})}{1+\exp(\vec{x}_k\,\vec{w})}\, x_{kj}$$

$$Hessian:\ H_{ij,lm}(\bar{w},\vec{x}) = \frac{\partial\partial l(\bar{w},\vec{x})}{\partial w_{ij}\,\partial w_{lm}}$$

a9a Dataset Benchmark

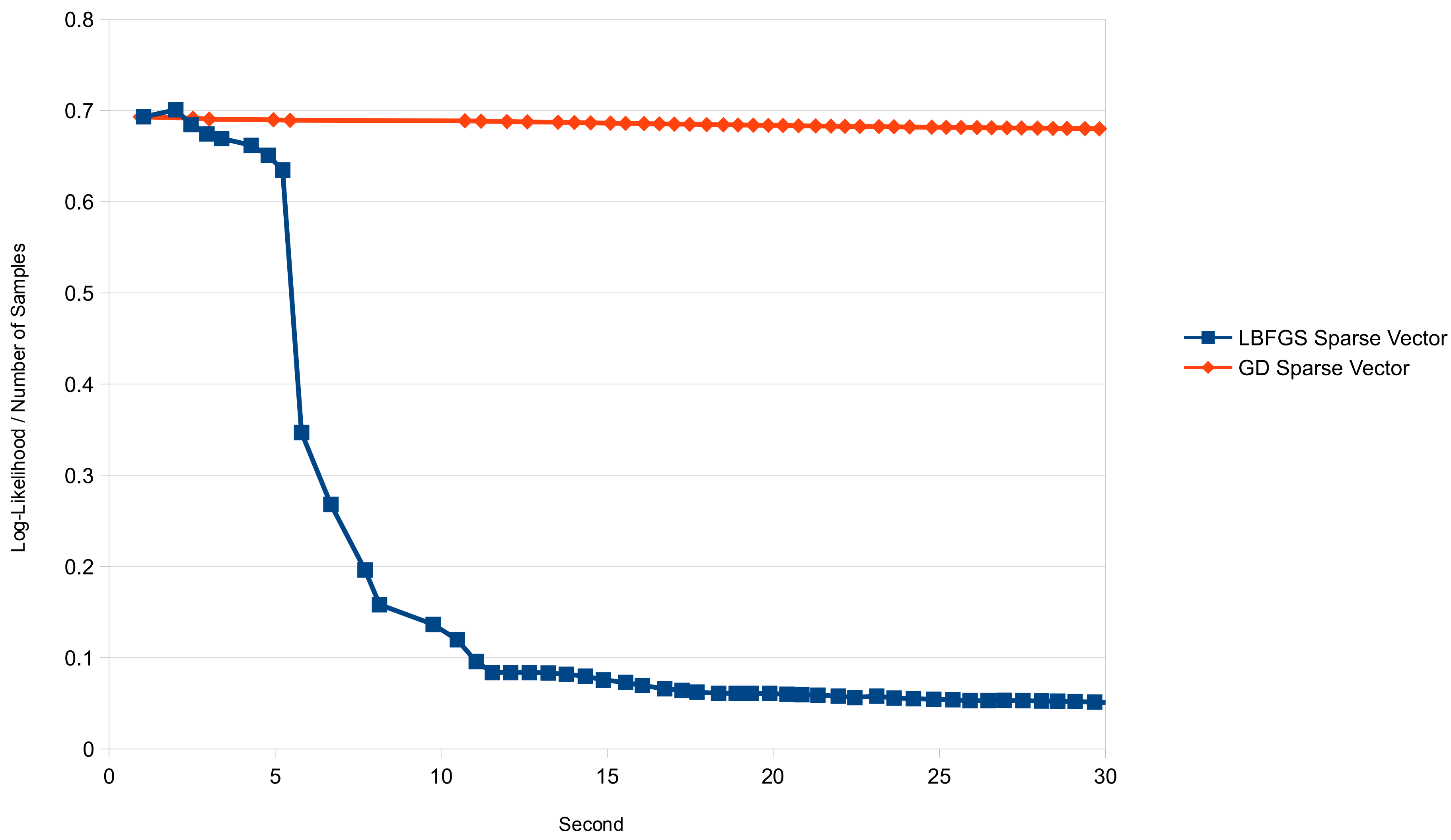


a9a Dataset Benchmark

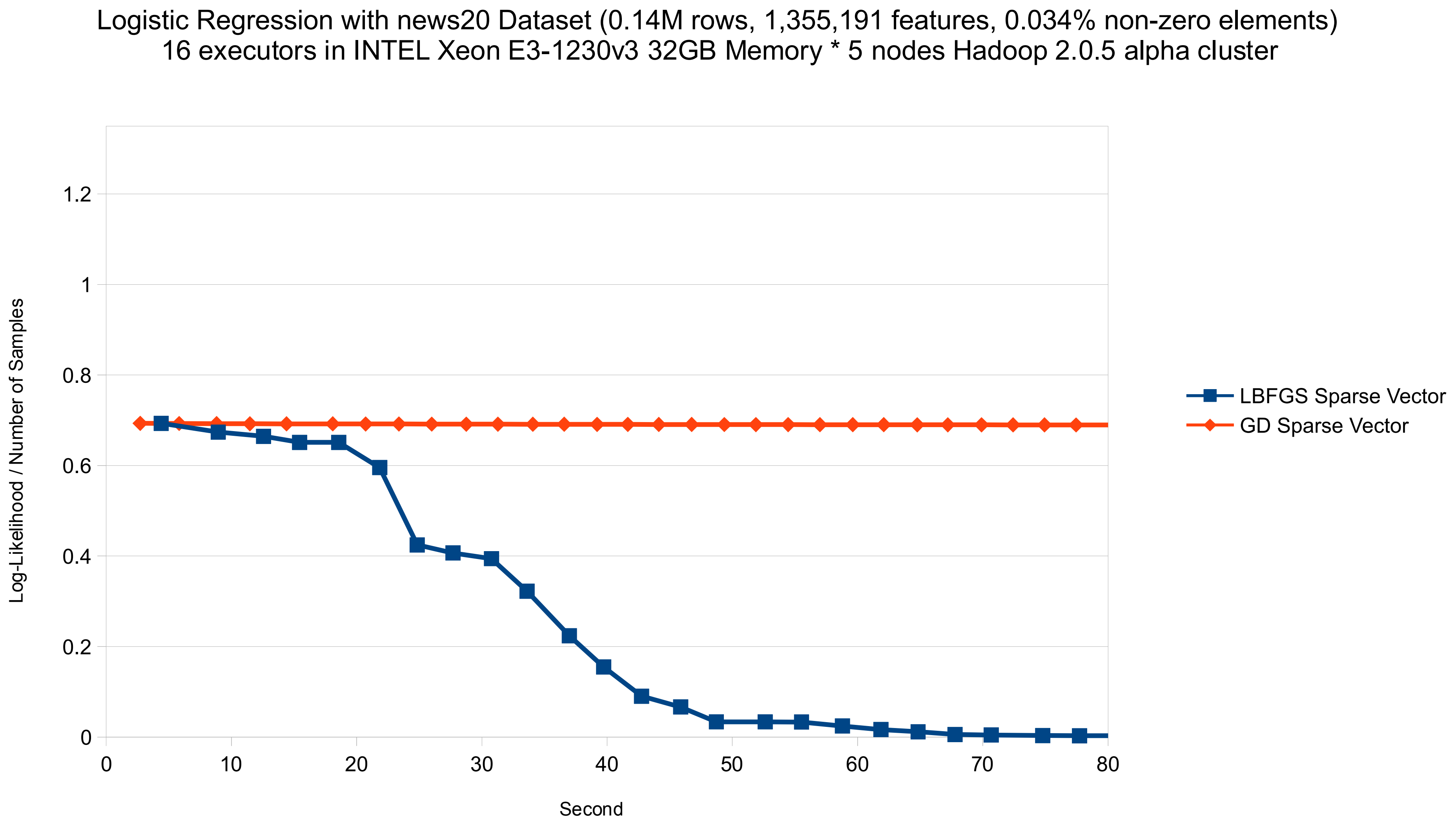


rcv1 Dataset Benchmark

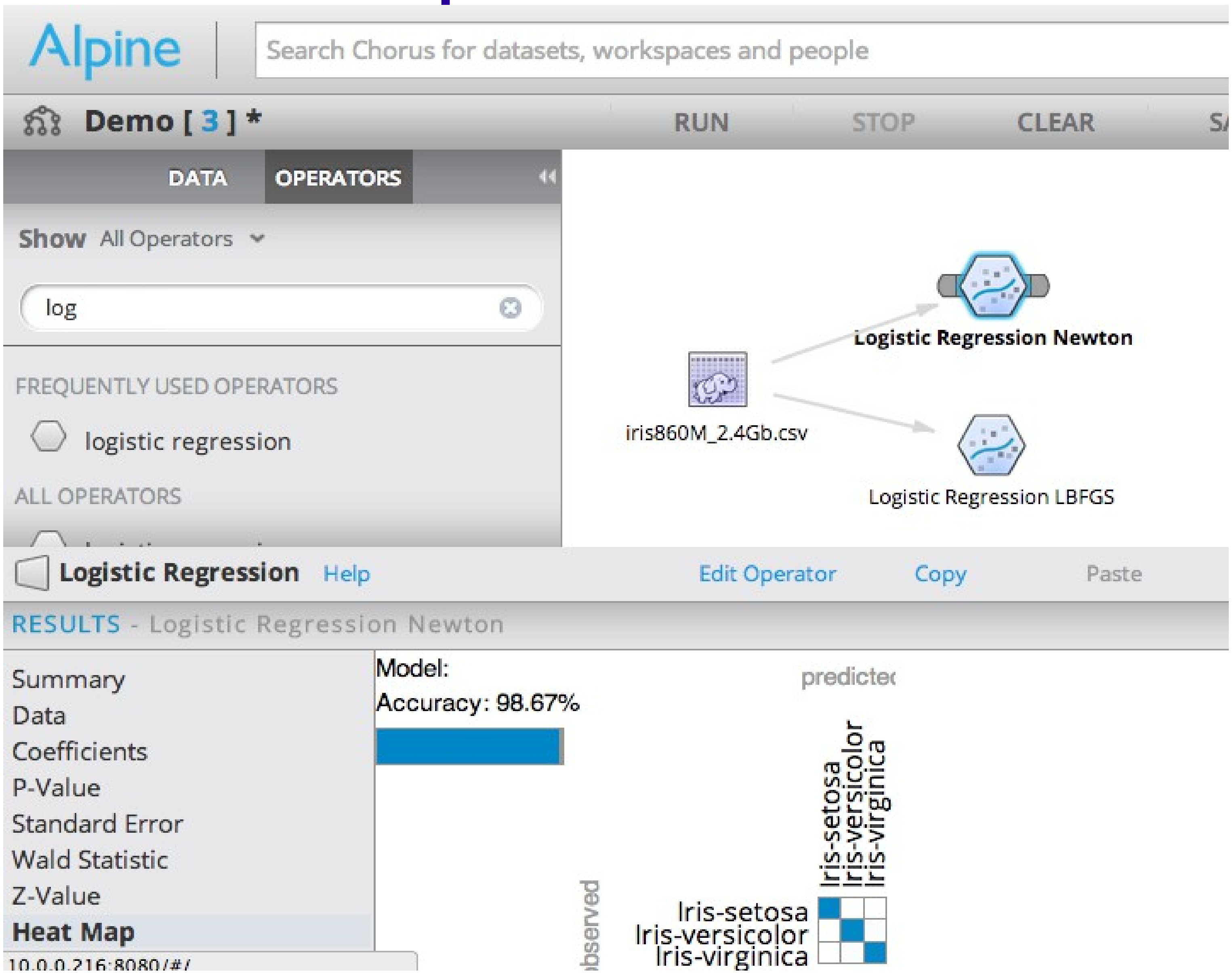
Logistic Regression with rcv1 Dataset (6.8M rows, 677,399 features, 0.15% non-zero elements)
16 executors in INTEL Xeon E3-1230v3 32GB Memory * 5 nodes Hadoop 2.0.5 alpha cluster



news20 Dataset Benchmark



Alpine Demo



Alpine
Search Chorus for datasets, workspaces and people
Demo [3] *
RUN
STOP
CLEAR
DATA
OPERATORS
Show All Operators
log
FREQUENTLY USED OPERATORS
logistic regression
ALL OPERATORS
Logistic Regression Newton
iris860M_2.4Gb.csv
Logistic Regression LBFGS
Logistic Regression
Help
Edit Operator
Copy
Paste
RESULTS - Logistic Regression Newton
Summary
Data
Coefficients
P-Value
Standard Error
Wald Statistic
Z-Value
Heat Map
Model:
Accuracy: 98.67%
Iris-setosa
Iris-versicolor
Iris-virginica
Iris-setosa
Iris-versicolor
Iris-virginica

Alpine Demo

Spark Stages Storage Environment Executors **Spark_Logistic_Regression_Train_Job** application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
3	Memory Deserialized 1x Replicated	39	100%	6.6 GB	0.0 B

Alpine Demo

Spark | Stages | Storage | Environment | Executors | **Spark_Logistic_Regression_Train_Job** application UI

Spark Stages

Total Duration: 37.5 s
Scheduling Mode: FIFO
Active Stages: 1
Completed Stages: 4
Failed Stages: 0

Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
6	reduce at MultiLogisticRegression.scala:357	2014/05/01 13:04:14	2.0 s	24/39		

Completed Stages (4)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
5	reduce at MultiLogisticRegression.scala:357	2014/05/01 13:04:11	3.1 s	39/39		
4	reduce at MultiLogisticRegression.scala:357	2014/05/01 13:04:08	2.9 s	39/39		
3	reduce at MultiLogisticRegression.scala:357	2014/05/01 13:03:49	19.2 s	39/39		
0	reduce at DistinctValueCounter.scala:66	2014/05/01 13:03:42	6.5 s	39/39		

Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write

Conclusion

- We're hiring!
- Spark runs programs 100x faster than Hadoop
- Spark turns iterative big data machine learning problems into single machine problems
- MLlib provides lots of state of art machine learning implementations, e.g. K-means, linear regression, logistic regression, ALS, collaborative filtering, and Naive Bayes, etc.
- Spark provides ease of use APIs in Java, Scala, or Python, and interactive Scala and Python shell
- Spark is Apache project, and it's the most active big data open source project nowadays.