An Introduction to Using Spark for Machine Learning

Stephen Boesch
Silicon Valley Code Camp '14
javadba@gmail.com

Warmup: Local Linear Regression

Vanilla Scala

```
  case class DataPoint(x: Vector[Double], y: Double)

  def generateData = {
    def generatePoint(i: Int) = {
      val y = if(i % 2 == 0) -1 else 1
      val x = DenseVector.fill(D){rand.nextGaussian + y * R}
      DataPoint(x, y)
    }
    Array.tabulate(N)(generatePoint)
  }


    val data = generateData
    // Initialize w to a random value
    var w = DenseVector.fill(D){2 * rand.nextDouble - 1}
    println("Initial w: " + w)

    for (i <- 1 to ITERATIONS) {
      println("On iteration " + i)
      var gradient = DenseVector.zeros[Double](D)
      for (p <- data) {
        val scale = (1 / (1 + math.exp(-p.y * (w.dot(p.x)))) - 1) * p.y
        gradient +=  p.x * scale
      }
      w -= gradient
    }

    println("Final w: " + w)
```

```
Initial w: DenseVector(0.26984398152220623, 0.2141701140700476, -0.2947345596068882, 0.4786029732880166, -0.893023573769929,
-0.8314545701477489, -0.14399803242006537, -0.11886354990138681, 0.2546115277961041, -0.2531907652708987)
On iteration 1
On iteration 2
On iteration 3
On iteration 4
On iteration 5
Final w: DenseVector(4104.954211961398, 4124.402892032297, 4973.560861262447, 3873.9421188094493, 5994.273235277295,
5765.45578236057, 4797.932213685545, 4879.9863549280135, 4212.584699094446, 5029.314517604473)
```

Warmup: Python

```
Using Python version 2.7.5 (default, Aug 25 2013 00:04:04)

SparkContext available as sc.

>>> from pyspark.mllib.random import RandomRDDs

sc = ... # SparkContext

# Generate a random double RDD that contains 1 million i.i.d. values drawn from the
# standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
u = RandomRDDs.uniformRDD(sc, 1000000L, 10)
# Apply a transform to get a random double RDD following `N(1, 4)`.
v = u.map(lambda x: 1.0 + 2.0 * x)
from pyspark.mllib.stat import Statistics
summary = Statistics.colStats(mat)
print summary.mean()
print summary.variance()
print summary.numNonzeros()

# Dense vectors are simply represented as NumPy array objects, so there is no need to covert them for use in
MLlib

# shamelessly lifted from SOF:  http://stackoverflow.com/questions/312443/how-do-you-split-a-list-into-evenly-
sized-chunks-in-python
def chunk(it, size):
    it = iter(it
    return iter(lambda: tuple(islice(it, size)), ())
```

Warmup: Hypothesis Testing in Java

```
// Compute Chi Squared against vector or matrix

JavaSparkContext jsc =  new JavaSparkContext("local", "Simple App",
      "$SPARK_HOME", new String[]{"target/svcc-1.0.jar"});
```

```
Vector vec = ... // a vector composed of the frequencies of events

// compute the goodness of fit. If a second vector to test against is not supplied as a parameter,
// the test runs against a uniform distribution.
ChiSqTestResult goodnessOfFitTestResult = Statistics.chiSqTest(vec);
// summary of the test including the p-value, degrees of freedom, test statistic, the method used,
// and the null hypothesis.
System.out.println(goodnessOfFitTestResult);

Matrix mat = ... // a contingency matrix

// conduct Pearson's independence test on the input contingency matrix
ChiSqTestResult independenceTestResult = Statistics.chiSqTest(mat);
// summary of the test including the p-value, degrees of freedom...
System.out.println(independenceTestResult);

JavaRDD<LabeledPoint> obs = ... // an RDD of labeled points

// The contingency table is constructed from the raw (feature, label) pairs and used to conduct
// the independence test. Returns an array containing the ChiSquaredTestResult for every feature
// against the label.
ChiSqTestResult[] featureTestResults = Statistics.chiSqTest(obs.rdd());
int i = 1;
for (ChiSqTestResult result : featureTestResults) {
    System.out.println("Column " + i + ":");
    System.out.println(result); // summary of the test
    i++;
}
```

Short Rest after Warmup .. now why Spark?

All of the prior examples could have been as easily performed directly in python. And much easier in R/Matlab/other stats packages.

So let us explore the capabilities of Spark that make it worthwhile to consider as a Machine Learning development and execution platform.

Spark Architecture (Briefly..)

From the Spark Documentation http://spark.apache.org/docs/latest

Spark

- Is a Fast and general-purpose cluster computing system.
- Provides high-level APIs in Java, Scala and Python
- Has an Optimized engine that supports general execution graphs
- Supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing
  - MLlib for machine learning
  - GraphX for graph processing
  - Spark Streaming.
- Integrated with the Hadoop Ecosystem
  - Can run on Map/Reduce V1 (MRV1) as well as YARN
  - Able to read all Hadoop file formats
  - Integrated Hive query support
- Interoperability with clustering frameworks
  - Mesos - distributed app management
  - Tachyon -in-memory filesystem

RDD's: the Heart of Spark DataSets

Resilient Distributed Datasets - RDD's - are the core of the Spark Framework.

- In-memory Immutable Data Structure
- Supports a rich set of functional operators. E.g. map/filter/reduce/partition/aggregate operators including count/groupBy.
- Supports DAG/Graph of dependencies
- Tolerant to failures: data lineage is tracked so the data can be rebuilt from upstream RDD's
- Supports parallel operations
- Data Partitioning is supported and exposed: properly used this leads to reduced network traffic during shuffle stages
- Transformations are *lazy:* a pipeline/series of operations are defined programmatically but not executed .. until ..
- Actions: return data to the driver program. e.g count/collect/take(N)
- Persistence: cache(), saveAsNewAPIHadoopFile, saveAsXXX. etc

Summary: Why Spark for ML?

Scalability: Similar to Hadoop in theory - though Spark is "young" in terms of supporting truly massive datasets. Hadoop has the edge on maturity there. But this is changing rapidly as companies start to throw everything they have onto Spark.

- Alternatives:
  - *RevolutionAnalytics*: custom cluster computing backend for R

Performance: 10X+ over Hadoop for on-disk, up to 100X on datasets fitting in memory. *(Note: 10/10: An impressive announcement was made today, stay tuned ..!)*

In-memory support makes Spark ideal for iterative algorithms - avoid re-reading model/data on each iteration

Functional programming style fits well with data manipulations inherent in ML.

Operator overloading allows creation of Domain Specific Languages that give Scala ability to come closer to the concise syntax of Statistical languages R, Matlab, etc. (But not *as* succint - those do win for equation definition).

Compared to Stats languages:

- Much much less availability of libraries
  - But growing quickly
- Visualization is not built-in.
  - And Scala overall is light on Vis
  - (Though once again that is evolving as new contribs happening)

What About SparkR?

Direct quote from DataBricks (Spark supporter company) 1.1.0 release announcement:

> At this point you may be asking why we're providing native support for statistics functions inside Spark given the existence of the SparkR project. As an R package, SparkR is a great lightweight solution for empowering familiar R APIs with distributed computation support. What we're aiming to accomplish with these built-in Spark statistics APIs is cross language support as well as seamless integration with other components of Spark, such as Spark SQL and Streaming, for a unified data product development platform. We expect these features to be callable from SparkR in the future.

Algorithms supported by Spark in 1.1.0 Release

```
Statistics:
   Correlations: data dependence analysis
   hypothesis testing: goodness of fit; independence test (ChiSquare)
   Stratified sampling: scaling training set with
      controlled label distribution
Random data generation: randomized algorithms; performance tests
Classification and regression
   linear models (SVMs, logistic regression, linear regression)
   decision trees
   naive Bayes
Collaborative filtering
   alternating least squares (ALS)
Clustering
   k-means, KMeans||
Dimensionality reduction
   singular value decomposition (SVD)
   principal component analysis (PCA)
Feature extraction and transformation
Optimization (developer)
   stochastic gradient descent
   limited-memory BFGS (L-BFGS)
```

Data Types: Matrices

Local Matrix
Integer-typed row and column indices => Double-typed values
stored on a single machine
Entries are stored in single double array in column major.

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices}

// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
```

Java

```
import org.apache.spark.mllib.linalg.Matrix;
import org.apache.spark.mllib.linalg.Matrices;

// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Matrix dm = Matrices.dense(3, 2, new double[] {1.0, 3.0, 5.0, 2.0, 4.0, 6.0});
```

Python!

```
import numpy as np
import scipy.sparse as sps
from pyspark.mllib.linalg import Vectors

# Use a NumPy array as a dense vector.
dv1 = np.array([1.0, 0.0, 3.0])
# Use a Python list as a dense vector.
dv2 = [1.0, 0.0, 3.0]
# Create a SparseVector.
sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])
# Use a single-column SciPy csc_matrix as a sparse vector.
sv2 = sps.csc_matrix((np.array([1.0, 3.0]), np.array([0, 2]), np.array([0, 2])), shape =
(3, 1))
```

Local vector

- Integer or Double types on local machine
- Dense or Sparse
- Eg. vector (1.0, 0.0, 3.0):
  - Dense: [1.0, 0.0, 3.0]
  - Sparse: (3, [0, 2], [1.0, 3.0]), 3 = size of the vector.
- *Python*:
  - Dense:
    - NumPy's array
    - Python's list, e.g., [1, 2, 3]
  - Sparse:
    - MLlib's SparseVector.
    - SciPy's csc_matrix with a single column

Further Examples

- Feature Reduction: PCA and SVD
- Classification
  - Decision Trees:
    - Hierarchical
    - Random Forests
  - Regularization
- Linear Regression
  - SVM
  - Streaming LR
- Collaborative Filtering via ALS - MovieLens
- Clustering
  - KMeans, KMeans||
- More Stats
  - Scaling - Standard Scaler
  - Distributions
  - Feature Extraction

Dimensionality Reduction: SVD

- *wikipedia*: The singular value decomposition (SVD) is a factorization of a real or complex matrix, with many useful applications in signal processing and statistics.

In any singular value decomposition

$$\mathbf{M} = \mathbf{U\Sigma V}^*$$

the diagonal entries of $\mathbf{\Sigma}$ are equal to the singular values of $\mathbf{M}$. The columns of $\mathbf{U}$ and $\mathbf{V}$ are, respectively, left- and right-singular vectors for the corresponding singular values. Consequently, the above theorem implies that:

- An $m \times n$ matrix $\mathbf{M}$ has at most $p = \min(m, n)$ distinct singular values.
- It is always possible to find an orthogonal basis $\mathbf{U}$ for K^m consisting of left-singular vectors of $\mathbf{M}$.
- It is always possible to find an orthogonal basis $\mathbf{V}$ for K^n consisting of right-singular vectors of $\mathbf{M}$.

If we keep the top k singular values, then the dimensions of the resulting low-rank matrix will be: U: m×k, Σ: k×k, V: n×k.

Performance

We assume n is smaller than m. The singular values and the right singular vectors are derived from the eigenvalues and the eigenvectors of the Gramian matrix ATA. The matrix storing the left singular vectors U, is computed via matrix multiplication as U=A(VS−1), if requested by the user via the computeU parameter. The actual method to use is determined automatically based on the computational cost:

If n is small (n<100) or k is large compared with n (k>n/2), we compute the Gramian matrix first and then compute its top eigenvalues and eigenvectors locally on the driver. This requires a single pass with O(n2) storage on each executor and on the driver, and O(n2k) time on the driver.

Otherwise, we compute (ATA)v in a distributive way and send it to **ARPACK** to compute (ATA)'s top eigenvalues and eigenvectors on the driver node. This requires O(k) passes, O(n) storage on each executor, and O(nk) storage on the driver.

Dimensionality Reduction: SVD

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.mllib.linalg.SingularValueDecomposition
```

```
val nPoints = 50000
val mat: RowMatrix = getRowMatrix(sc, nPoints)  // getRowMatrix is custom helper

// Compute the top 20 singular values and corresponding singular vectors.
val svd: SingularValueDecomposition[RowMatrix, Matrix] = mat.computeSVD(20, computeU = true)
val U: RowMatrix = svd.U // The U factor is a RowMatrix.
val s: Vector = svd.s // The singular values are stored in a local dense vector.
val V: Matrix = svd.V // The V factor is a local dense matrix.
```

```
scala> val svd: SingularValueDecomposition[RowMatrix, Matrix] = mat.computeSVD(2, computeU = true)
14/10/11 08:17:15 INFO SparkContext: Starting job: reduce at RDDFunctions.scala:111
14/10/11 08:17:15 INFO DAGScheduler: Got job 8 (reduce at RDDFunctions.scala:111) with 2 output partitions (allowLocal=false)
14/10/11 08:17:15 INFO DAGScheduler: Final stage: Stage 8(reduce at RDDFunctions.scala:111)
14/10/11 08:17:15 INFO DAGScheduler: Parents of final stage: List()
14/10/11 08:17:15 INFO DAGScheduler: Missing parents: List()
14/10/11 08:17:15 INFO DAGScheduler: Submitting Stage 8 (MapPartitionsRDD[8] at mapPartitions at RDDFunctions.scala:100), which has no missing parents
14/10/11 08:17:15 INFO MemoryStore: ensureFreeSpace(2672) called with curMem=16912, maxMem=286300569
14/10/11 08:17:15 INFO MemoryStore: Block broadcast_8 stored as values in memory (estimated size 2.6 KB, free 273.0 MB)
14/10/11 08:17:15 INFO DAGScheduler: Submitting 2 missing tasks from Stage 8 (MapPartitionsRDD[8] at mapPartitions at RDDFunctions.scala:100)
14/10/11 08:17:15 INFO TaskSchedulerImpl: Adding task set 8.0 with 2 tasks
14/10/11 08:17:15 INFO BlockManager: Removing broadcast 7
14/10/11 08:17:15 INFO BlockManager: Removing block broadcast_7
..

14/10/11 08:17:15 INFO MemoryStore: ensureFreeSpace(88) called with curMem=17840, maxMem=286300569
14/10/11 08:17:15 INFO MemoryStore: Block broadcast_9 stored as values in memory (estimated size 88.0 B, free 273.0 MB)
svd: org.apache.spark.mllib.linalg.SingularValueDecomposition[org.apache.spark.mllib.linalg.distributed.RowMatrix,org.apache.spark.mllib.linalg.Matrix] =
SingularValueDecomposition(org.apache.spark.mllib.linalg.distributed.RowMatrix@6e8aa44a,[3016.2683309146237,1790.8648037468352],-0.27004977268916075
0.026303436294853808
-0.5352393008799868   -0.835044797786914
-0.8003699213888763   0.5495528317897149    )

scala> val U: RowMatrix = svd.U // The U factor is a RowMatrix.
U: org.apache.spark.mllib.linalg.distributed.RowMatrix = org.apache.spark.mllib.linalg.distributed.RowMatrix@6e8aa44a

scala> val s: Vector = svd.s // The singular values are stored in a local dense vector.
s: org.apache.spark.mllib.linalg.Vector = [3016.2683309146237,1790.8648037468352]

scala> val V: Matrix = svd.V // The V factor is a local dense matrix.
V: org.apache.spark.mllib.linalg.Matrix =
-0.27004977268916075  0.026303436294853808
-0.5352393008799868   -0.835044797786914
-0.8003699213888763   0.5495528317897149
```

Simple Distributions and Helpers

```scala
// Generate a random double RDD that contains 1 million i.i.d. values drawn from the
// standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
// Then apply a transform to get a random double RDD following `N(1, 4)`.
def normal(sc: SparkContext, N: Long) = normalRDD(sc, N, 10).map{ x =>
  1.0 + 2.0 * x
}

import org.apache.spark.mllib.stat._
import org.apache.spark.rdd.RDD

def getSummary(rdd: RDD[Vector]): MultivariateStatisticalSummary = {
  // Coming in 1.2.0 !  rdd.treeAggregate(new MultivariateOnlineSummarizer)(
  rdd.aggregate(new MultivariateOnlineSummarizer)(
    (aggregator, data) => aggregator.add(data),
    (aggregator1, aggregator2) => aggregator1.merge(aggregator2))
}

def printSummary(summ: MultivariateStatisticalSummary) = {
  println(s"Count: ${summ.count}\nMean: ${summ.mean}\nMax: ${summ.max}\nVar: ${summ.variance}")
}
```

```scala
 // Custom distribution
  def genVectorRdd(sc: SparkContext, n: Int) = {
    import java.util.Random
    val xrand = new Random(37)
    val yrand = new Random(41)
    val zrand = new Random(43)
    val SignalScale = 5.0
    val NoiseScale = 8.0
    val npoints = 50000
    val XScale = SignalScale / npoints
    val YScale = 2.0 * SignalScale / npoints
    val ZScale = 3.0 * SignalScale / npoints
    val randRdd = sc.parallelize({
      for (p <- Range(0, npoints))
      yield
        (NoiseScale * xrand.nextGaussian + XScale * p,
          NoiseScale * yrand.nextGaussian + YScale * p,
          NoiseScale * zrand.nextGaussian + ZScale * p)
    }.toSeq, 2)
    val vecs = randRdd.map { case (x, y, z) =>
      Vectors.dense(Array(x, y, z))
    }
    val summary = getSummary(vecs)
    printSummary(summary)
    vecs
  }
def getRowMatrix(sc: SparkContext, n: Int) = {
  new RowMatrix(genVectorRdd(sc, n))
}
```

Dimensionality Reduction

Principal Component Analysis (PCA)

- PCA computes eigenvalues/vectors and sorts by highest to lowest

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix

val mat: RowMatrix = ...

// Compute the top 10 principal components.
val pc: Matrix = mat.computePrincipalComponents(10) // Principal components are stored in a local dense
matrix.

// Project the rows to the linear space spanned by the top 10 principal components.
val projected: RowMatrix = mat.multiply(pc)
```

```
scala> val pc: Matrix = mat.computePrincipalComponents(2)
14/10/11 08:25:54 INFO SparkContext: Starting job: reduce at RDDFunctions.scala:111
14/10/11 08:25:54 INFO DAGScheduler: Got job 9 (reduce at RDDFunctions.scala:111) with 2 output partitions (allowLocal=false)
14/10/11 08:25:54 INFO DAGScheduler: Final stage: Stage 9(reduce at RDDFunctions.scala:111)
14/10/11 08:25:54 INFO DAGScheduler: Parents of final stage: List()
14/10/11 08:25:54 INFO DAGScheduler: Missing parents: List()
14/10/11 08:25:54 INFO DAGScheduler: Submitting Stage 9 (MapPartitionsRDD[10] at mapPartitions at RDDFunctions.scala:100), which has no missing parents
14/10/11 08:25:54 INFO MemoryStore: ensureFreeSpace(2608) called with curMem=17928, maxMem=286300569
14/10/11 08:25:54 INFO MemoryStore: Block broadcast_10 stored as values in memory (estimated size 2.5 KB, free 273.0 MB)
14/10/11 08:25:54 INFO DAGScheduler: Submitting 2 missing tasks from Stage 9 (MapPartitionsRDD[10] at mapPartitions at RDDFunctions.scala:100)
14/10/11 08:25:54 INFO TaskSchedulerImpl: Adding task set 9.0 with 2 tasks
14/10/11 08:25:54 WARN TaskSetManager: Stage 9 contains a task of very large size (1173 KB). The maximum recommended task size is 100 KB.
14/10/11 08:25:54 INFO TaskSetManager: Starting task 0.0 in stage 9.0 (TID 16, localhost, PROCESS_LOCAL, 1201285 bytes)
..

14/10/11 08:25:54 INFO Executor: Finished task 1.0 in stage 10.0 (TID 19). 806 bytes result sent to driver
14/10/11 08:25:54 INFO Executor: Finished task 0.0 in stage 10.0 (TID 18). 806 bytes result sent to driver
14/10/11 08:25:54 INFO TaskSetManager: Finished task 1.0 in stage 10.0 (TID 19) in 61 ms on localhost (1/2)
14/10/11 08:25:54 INFO TaskSetManager: Finished task 0.0 in stage 10.0 (TID 18) in 98 ms on localhost (2/2)
14/10/11 08:25:54 INFO DAGScheduler: Stage 10 (reduce at RDDFunctions.scala:111) finished in 0.098 s
14/10/11 08:25:54 INFO TaskSchedulerImpl: Removed TaskSet 10.0, whose tasks have all completed, from pool
14/10/11 08:25:54 INFO DAGScheduler: Job 10 finished: reduce at RDDFunctions.scala:111, took 0.128423 s
pc: org.apache.spark.mllib.linalg.Matrix =
-0.27855310208036344  0.031048647935745476
-0.5402248784021993   -0.8314387702591389
-0.7940813875647355   0.5547481886147375
```

Classification: Decision Trees
Hierarchical and Random Forests

- Docs: http://spark.apache.org/docs/latest/mllib-decision-tree.html

Node impurity and information gain

The *node impurity* is a measure of the homogeneity of the labels at the node. The current implementation provides two impurity measures fo classification (Gini impurity and entropy) and one impurity measure for regression (variance).

Impurity	Task	Formula	Description
Gini impurity	Classification	$\sum_{i=1}^{M} f_i(1-f_i)$	f_i is the frequency of label i at a node and M is the number of unique labels.
Entropy	Classification	$\sum_{i=1}^{M} -f_i log(f_i)$	f_i is the frequency of label i at a node and M is the number of unique labels.
Variance	Regression	$\frac{1}{n}\sum_{i=1}^{N}(x_i-\mu)^2$	y_i is label for an instance, N is the number of instances and μ is the mean given by $\frac{1}{N}\sum_{i=1}^{n} x_i$.

The *information gain* is the difference between the parent node impurity and the weighted sum of the two child node impurities. Assuming th split s partitions the dataset D of size N into two datasets D_{left} and D_{right} of sizes N_{left} and N_{right}, respectively, the information gain is:

$$IG(D,s) = Impurity(D) - \frac{N_{left}}{N} Impurity(D_{left}) - \frac{N_{right}}{N} Impurity(D_{right})$$

Classification: Decision Trees

Hierarchical and Random Forests

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
// Cache the data since we will use it again to compute training error.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt").cache()

// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 100

val model = DecisionTree.trainClassifier(data, numClasses, categoricalFeaturesInfo, impurity,
  maxDepth, maxBins)

// Evaluate model on training instances and compute training error
val labelAndPreds = data.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val trainErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / data.count
println("Training Error = " + trainErr)
println("Learned classification tree model:\n" + model)
```

```
14/10/11 08:34:48 INFO TaskSetManager: Finished task 0.0 in stage 20.0 (TID 37) in 118 ms on localhost (2/2)
14/10/11 08:34:48 INFO DAGScheduler: Stage 20 (collectAsMap at DecisionTree.scala:573) finished in 0.119 s
14/10/11 08:34:48 INFO TaskSchedulerImpl: Removed TaskSet 20.0, whose tasks have all completed, from pool
14/10/11 08:34:48 INFO DAGScheduler: Job 17 finished: collectAsMap at DecisionTree.scala:573, took 0.296066 s
14/10/11 08:34:48 INFO RandomForest: Internal timing for DecisionTree:
14/10/11 08:34:48 INFO RandomForest:   init: 0.499509
  total: 2.262138
  findSplitsBins: 0.326065
  findBestSplits: 1.753114
  chooseSplits: 1.747873
model: org.apache.spark.mllib.tree.model.DecisionTreeModel = DecisionTreeModel classifier of depth 2 with 5 nodes
```

Classification: Naive Bayes

- From the Mllib docs:
  - MLlib supports multinomial naive Bayes, which is typically used for document classification. Within that context, each observation is a document and each feature represents a term whose value is the frequency of the term. Feature values must be nonnegative to represent term frequencies. Additive smoothing can be used by setting the parameter λ (default to 1.0). F

Multinomial naive Bayes [edit]

With a multinomial event model, samples (feature vectors) represent the frequencies with which certain events have been generated by a multinomial $(p_1, \ldots, p_n)$ where p_i is the probability that event i occurs (or k such multinomials in the multiclass case). This is the event model typically used for document classification; the feature values are then term frequencies, generated by a multinomial that produces some number of words (see bag of words assumption). The likelihood of observing a feature vector (histogram) F is given by

$$p(F|C) = \frac{(\sum_i F_i)!}{\prod_i F_i!} \prod_i {p_i}^{F_i}$$

Classification: Naive Bayes

```
import org.apache.spark.mllib.classification.NaiveBayes
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
```

```
val data = sc.textFile("data/mllib/sample_naive_bayes_data.txt")
val parsedData = data.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}
// Split data into training (60%) and test (40%).
val splits = parsedData.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0)
val test = splits(1)

val model = NaiveBayes.train(training, lambda = 1.0)
```

```
 val predictionAndLabel = test.map(p => (model.predict(p.features), p.label))
 val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count() / test.count()
```

```
scala> val splits = parsedData.randomSplit(Array(0.6, 0.4), seed = 11L)
splits: Array[org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint]] = Array(PartitionwiseSampledRDD[37] at randomSplit at
<console>:40, PartitionwiseSampledRDD[38] at randomSplit at <console>:40)
scala> val training = splits(0)
training: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = PartitionwiseSampledRDD[37] at randomSplit at <console>:40
scala> val test = splits(1)
test: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = PartitionwiseSampledRDD[38] at randomSplit at <console>:40
14/10/11 08:42:58 INFO Executor: Finished task 0.0 in stage 24.0 (TID 45). 1329 bytes result sent to driver
14/10/11 08:42:58 INFO TaskSetManager: Finished task 0.0 in stage 24.0 (TID 45) in 18 ms on localhost (2/2)
14/10/11 08:42:58 INFO DAGScheduler: Stage 24 (collect at NaiveBayes.scala:120) finished in 0.019 s
14/10/11 08:42:58 INFO TaskSchedulerImpl: Removed TaskSet 24.0, whose tasks have all completed, from pool
14/10/11 08:42:58 INFO DAGScheduler: Job 20 finished: collect at NaiveBayes.scala:120, took 0.047686 s
model: org.apache.spark.mllib.classification.NaiveBayesModel = org.apache.spark.mllib.classification.NaiveBayesModel@338da3a0
scala> val predictionAndLabel = test.map(p => (model.predict(p.features), p.label))
predictionAndLabel: org.apache.spark.rdd.RDD[(Double, Double)] = MappedRDD[41] at map at <console>:48
scala> val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count() / test.count()
14/10/11 08:43:00 INFO ContextCleaner: Cleaned shuffle 3
14/10/11 08:43:00 INFO BlockManager: Removing broadcast 29
4/10/11 08:43:00 INFO TaskSchedulerImpl: Removed TaskSet 26.0, whose tasks have all completed, from pool
14/10/11 08:43:00 INFO DAGScheduler: Job 22 finished: count at <console>:50, took 0.016937 s
accuracy: Double = 1.0
```

Collaborative Filtering: Alternating Least Squares (ALS)

MovieLens Data

- Goal: predict the ratings of u users for m movies
- Starting with: a partially filled matrix R containing the known ratings for some user-moviepairs.

- ALS models R as the product of two matrices Ma nd U of dimensions m×k and k×u respectively
- Each user and each movie has a k-dimensional “featurevector” describing its characteristics
- a user’s rating for a movie is the dot product of its feature vector and the movie’s.
- ALS solves for M and U using the known ratings and then computes M×U to predict the unknown ones.

Collaborative Filtering: Alternating Least Squares (ALS)

MovieLens Data

Alternate Least Squares Optimizes for object MovieLensALS {

```
// Register custom structures into the Serializer

  class ALSRegistrator extends KryoRegistrator {
    override def registerClasses(kryo: Kryo) {
      kryo.register(classOf[Rating])
      kryo.register(classOf[mutable.BitSet])
    }
  }
    ❖  // Parse out th ratings

   val ratings = sc.textFile(params.input).map { line =>
     val fields = line.split("::")
    if (params.implicitPrefs) {
      /*
       * MovieLens ratings are on a scale of 1-5:
       * 5: Must see
       * 4: Will enjoy
       * 3: It's okay
       * 2: Fairly bad
       * 1: Awful
       * So we should not recommend a movie if the predicted rating is less than 3.
       * To map ratings to confidence scores, we use
       * 5 -> 2.5, 4 -> 1.5, 3 -> 0.5, 2 -> -0.5, 1 -> -1.5. This mappings means unobserved
       * entries are generally between It's okay and Fairly bad.
       * The semantics of 0 in this expanded world of non-positive weights
       * are "the same as never having interacted at all".
       */
      Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble - 2.5)
    } else {
      Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
    }
  }.cache()

  val numRatings = ratings.count()
  val numUsers = ratings.map(_.user).distinct().count()
  val numMovies = ratings.map(_.product).distinct().count()

  println(s"Got $numRatings ratings from $numUsers users on $numMovies movies.")
```

Collaborative Filtering: Alternating Least Squares (ALS)

MovieLens Data

```
// Partition the dataset into training and test and cache in the cluster memory
   val splits = ratings.randomSplit(Array(0.8, 0.2))
    val training = splits(0).cache()
    val test = if (params.implicitPrefs) {
      /*
       * 0 means "don't know" and positive values mean "confident that the prediction should be 1".
       * Negative values means "confident that the prediction should be 0".
       * We have in this case used some kind of weighted RMSE. The weight is the absolute value of
       * the confidence. The error is the difference between prediction and either 1 or 0,
       * depending on whether r is positive or negative.
       */
      splits(1).map(x => Rating(x.user, x.product, if (x.rating > 0) 1.0 else 0.0))
    } else {
      splits(1)
    }.cache()

    val numTraining = training.count()
    val numTest = test.count()
    println(s"Training: $numTraining, test: $numTest.")

    ratings.unpersist(blocking = false)

// Set up the ALS model and run against the training set
    val model = new ALS()
      .setRank(params.rank)
      .setIterations(params.numIterations)
      .setLambda(params.lambda)
      .setImplicitPrefs(params.implicitPrefs)
      .setUserBlocks(params.numUserBlocks)
      .setProductBlocks(params.numProductBlocks)
      .run(training)

    val rmse = computeRmse(model, test, params.implicitPrefs)

    println(s"Test RMSE = $rmse.")
```

Collaborative Filtering: KMeans Clustering

Description [edit]

Given a set of observations ($\mathbf{x}_1$, $\mathbf{x}_2$, …, $\mathbf{x}_n$), where each observation is a *d*-dimensional real vector, *k*-means clustering aims to partition the *n* observations into *k* (≤ *n*) sets $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$ so as to minimize the within-cluster sum of squares (WCSS). In other words, its objective is to find:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2$$

where $\boldsymbol{\mu}_i$ is the mean of points in S_i.

History [edit]

Standard algorithm [edit]

The most common algorithm uses an iterative refinement technique. Due to its ubiquity it is often called the ***k*-means algorithm**; it is also referred to as **Lloyd's algorithm**, particularly in the computer science community.

Given an initial set of *k* means $m_1^{(1)},\ldots,m_k^{(1)}$ (see below), the algorithm proceeds by alternating between two steps:[7]

Assignment step: Assign each observation to the cluster whose mean yields the least within-cluster sum of squares (WCSS). Since the sum of squares is the squared Euclidean distance, this is intuitively the "nearest" mean.[8] (Mathematically, this means partitioning the observations according to the Voronoi diagram generated by the means).

$$S_i^{(t)} = \{x_p : \|x_p - m_i^{(t)}\|^2 \leq \|x_p - m_j^{(t)}\|^2 \ \forall j, 1 \leq j \leq k\},$$

where each x_p is assigned to exactly one $S^{(t)}$, even if it could be is assigned to two or more of them.

Update step: Calculate the new means to be the centroids of the observations in the new clusters.

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$

Collaborative Filtering: KMeans Clustering

```scala
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors

// Load and parse the data
val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))

// Cluster the data into two classes using KMeans
val numClusters = 2
val numIterations = 20
val clusters = KMeans.train(parsedData, numClusters, numIterations)

// Evaluate clustering by computing Within Set Sum of Squared Errors
val WSSSE = clusters.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " + WSSSE)
```

Backup Slides

PySpark

- Python was added Spark Version 0.7 in 2013.
- Write Spark jobs in Python
- Supports Python C extensions; not jython
- Interactive use through the Python REPL
- Well documented: https://spark.apache.org/docs/latest/api/python/index.html
- Built on top of the the Java API
- Communicates with a local Java Process using Py4J
- Python RDD's are stored as Spark as RDD[Array[Byte]]
- of serialized Python objects
- Functions are executed in Python worker processes that communicate with Spark Worker
- Communicate with Spark over local Pipes

Multivariate Stats in Python

```
vectors = list(chunks(varr))

rdd = sc.parallelize(vectors)

from pyspark.mllib.stat import Statistics
summary = Statistics.colStats(rdd)
print summary.mean()
print summary.variance()
print summary.numNonzeros()
```

```
>>> print summary.mean()
[ 2.03282035  1.95277754  2.05928149  1.96758764  2.0091106   2.01891858
  1.99522667  1.87768044  2.01224161  2.03110039  1.97562606  1.93322121
  1.96990131  1.91329294  1.9568143   1.98735053  1.95354473  1.93334595
  2.07542202  2.0167235   1.95672588  2.0006726   2.0505268   1.92291114
 ..
 1.99149066  1.9877257   1.98624929  2.02737063  2.00125737  2.10034423
  1.95524569  1.93951363  2.02638264  2.0998278   2.05584373  1.88265087
  2.03698843  2.0547556   2.02145163  2.03701662  1.88289513  1.88455293
  2.05633145  1.97806028  2.02746833  1.97435185]
>>> print summary.variance()
[ 0.27623369  0.3471978   0.37262464  0.34692731  0.34389388  0.36666851
  0.29865858  0.30273665  0.32713072  0.31498331  0.35375273  0.3526142
  0.34769931  0.33265259  0.31073556  0.31657906  0.32220851  0.31881301
  0.37935283  0.37326062  0.36286453  0.28700645  0.34463983  0.30170693
..

  0.31023702  0.36232397  0.32956356  0.30647882  0.31151803  0.29240776
  0.2839644   0.34041345  0.32836504  0.41981734  0.36547536  0.31959335
  0.37229229  0.32639899  0.35957478  0.38471713]
>>> print summary.numNonzeros()
[ 100.  100.  100.  100.  100.  100.  100.  100.  100.  100.  100.  100.
..

  100.  100.  100.  100.  100.  100.  100.  100.  100.  100.  100.  100.
  100.  100.  100.  100.]
```

Amazon EC2 Spark Console

Following is the console visible on the web at port 5080 on your ec2 master instance

http://ec2-54-234-7-59.compute-1.amazonaws.com:5080/ganglia/

Main | Search | Views | Aggregate Graphs | Compare Hosts | Events | Automatic Rotation | Mobile

spark Cluster Report for Sun, 12 Oct 2014 17:37:52 +0000 — Get Fresh Data

Last hour | 2hr | 4hr | day | week | month | year — or from UNKNOWN_TYPE to UNKNOWN_TYPE — Go | Clear — Physical View

Metric load_one — **Sorted** ascending | descending | by name

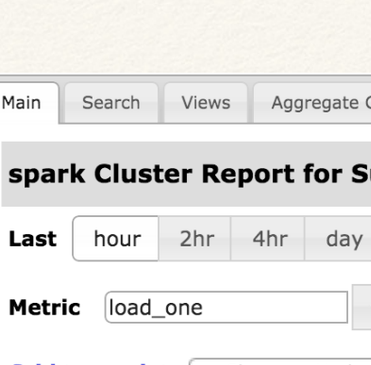
Grid > **spark** > --Choose a Node — Show only nodes matching UNKNOWN_TYPE — Filter — Max graphs to show all

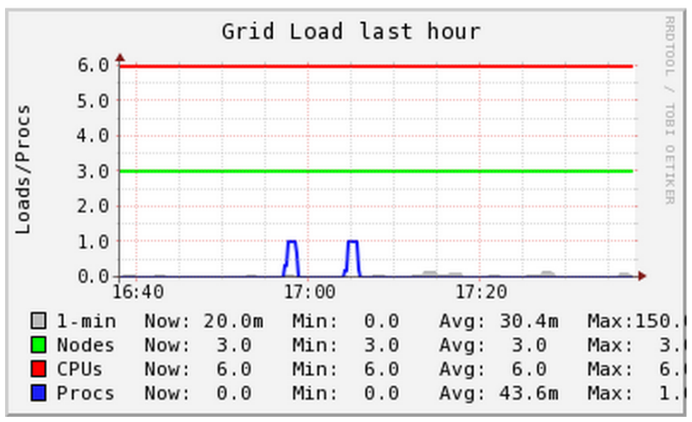
Overview of spark @ 2014-10-12 17:37

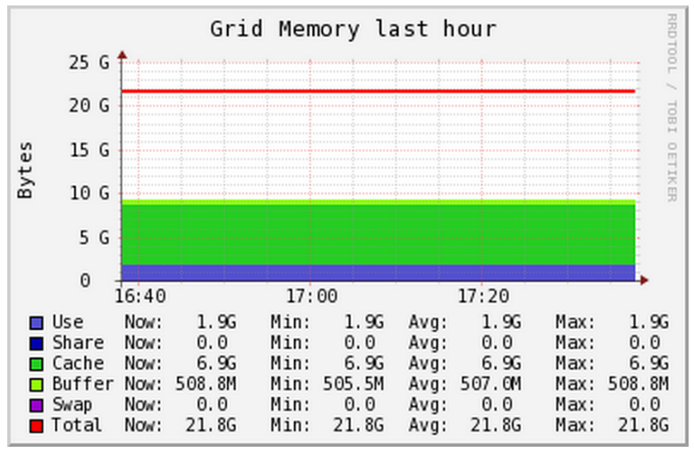
CPUs Total: **6**
Hosts up: **3**
Hosts down: **0**

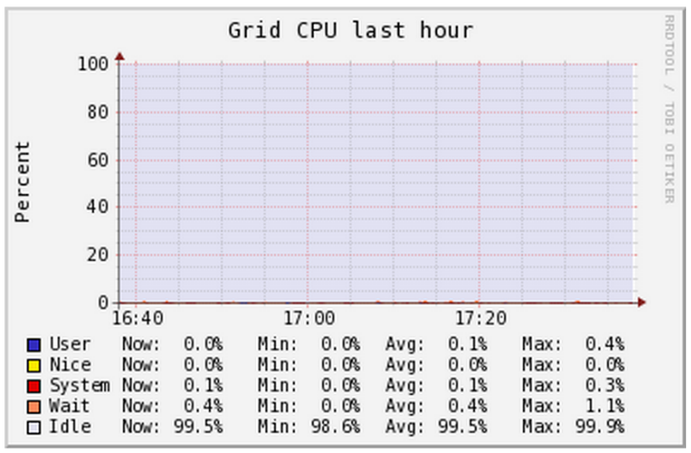
Current Load Avg (15, 5, 1m):
3%, 1%, 0%
Avg Utilization (last hour):
0%

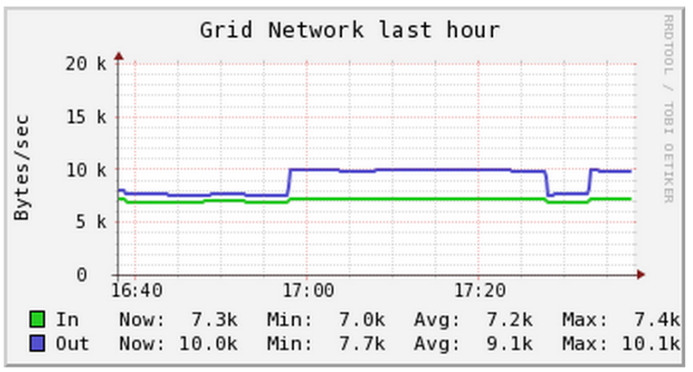
Utilization heatmap











Stacked Graph - load_one