Document Classification with Apache Spark

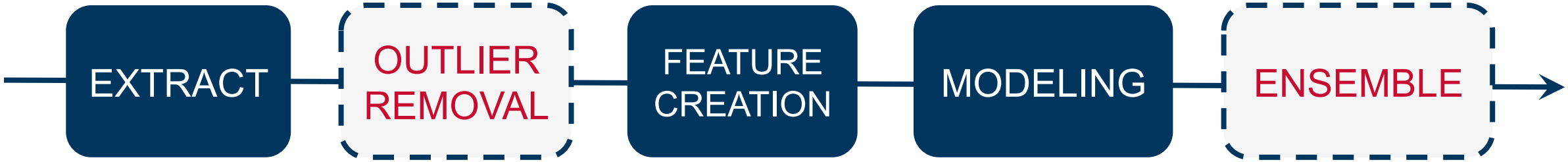
Joseph Blue, @joebluems

August 19, 2015

Background survey

- Data Science
  - Supervised vs. Unsupervised Scenarios
  - Classification Algorithms: Naïve Bayes, Linear, Decision Trees, etc.
  - Model metrics: KS, AuROC, etc.
  - Boosting, Stacking, Bagging, etc.
- TF-IDF Feature Extraction
- Apache Spark: RDD, DAG, Scala shell, MLlib
- Applying Machine Learning to Business Problems

Big Data Solution Workflow



Import data into Hadoop and transform into format appropriate for solution

Identify and remove / adjust records that negatively affect the ability to achieve good performance

Extract from the raw data inputs that the ML algorithms will use for pattern detection

Application of the appropriate machine learning algorithms, such as Naïve Bayes, Linear, Tree-Based methods

Combining multiple models to increase the performance of the ultimate solution

Resources

- GITHUB – code & instructions

  https://github.com/joebluems/Mockingbird

- Kaggle – data science competitions, code, message boards

  https://www.kaggle.com/competitions

- Spark MLLib

  http://spark.apache.org/docs/1.3.0/mllib-guide.html

Part 1: Working with Text

Raw “documents” corpus

- Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.
- Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.
- But, in a larger sense, we can not dedicate -- we can not consecrate -- we can not hallow -- this ground. The brave men, living and dead, who struggled here, have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us -- that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion -- that we here highly resolve that these dead shall not have died in vain -- that this nation, under God, shall have a new birth of freedom -- and that government of the people, by the people, for the people, shall not perish from the earth.

Tokenized* documents

- ArrayBuffer(four, score, seven, year, ago, our, father, brought, forth, contin, new, nation, conceiv, liberti, dedic, proposit, all, men, creat, equal)
- ArrayBuffer(now, we, engag, great, civil, war, test, whether, nation, ani, nation, so, conceiv, so, dedic, can, long, endur, we, met, great, battl, field, war, we, have, come, dedic, portion, field, final, rest, place, those, who, here, gave, live, nation, might, live, altogeth, fit, proper, we, should, do)
- ArrayBuffer(larger, sens, we, can, dedic, we, can, consecr, we, can, hallow, ground, brave,men, live, dead, who,struggl, here, have, consecr, far, abov, our, poor, power, add, detract, world, littl, note, nor, long, rememb, what, we, sai, here, can, never, forget, what, did, here, us, live, rather, dedic, here, unfinish, work, which, who, fought, here, have, thu, far, so, nobli, advanc, rather, us, here, dedic, great, task, remain, befor, us, from, honor, dead, we, take, increas, devot, caus, which, gave, last, full, measur, devot, we, here, highli, resolv, dead, shall, have, di, vain, nation, under, god, shall, have, new, birth, freedom, govern, peopl, peopl, peopl, shall, perish, from, earth)

*Using Apache Lucene's Standard English Analyzer

TF* Vectors – Total Frequency (i.e. word counts)

- (1000, [17,63,94,197,234,335,412,437,445,521,530,556,588,673,799,893 ,937,960,990], [1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0, 1.0,1.0,1.0])
- (1000, [17,21,22,37,63,92,167,211,240,256,270,272,393,395,445,449, 460,472,480,498,535,612,676,688,694,706,724,732,790,909,916,939,960, 965,996], [1.0,2.0,1.0,1.0,3.0,2.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,2.0,1.0,1.0,1.0,2.0,1.0,1.0,1.0,2.0, 1.0,1.0,2.0,1.0,1.0,4.0,1.0,1.0,1.0,1.0,1.0,1.0])
- (1000,[21,63,92,131,143,147,196,205,208,240,250,256,265,268,296,312,326,340, 341,367,378,391,399,400,412,417,441,445,449,455,464,483,494,503,515,524,526, 539,551,575,602,612,627,641,645,656,676,694,721,742,757,767,780,786,790,802, 807,817,818,844,852,920,946,951,960,983,985,990], [1.0,1.0,2.0,1.0,2.0,1.0,2.0,1.0,1.0,4.0,1.0,4.0,1.0,4.0,1.0,1.0,2.0,1.0,3.0,1.0,1.0,1.0,1.0, 2.0,1.0,1.0,1.0,3.0,1.0,1.0,1.0,2.0,3.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0, 1.0,2.0,1.0,4.0,1.0,1.0,1.0,2.0,6.0,1.0,1.0,1.0,1.0,1.0,3.0,1.0,2.0,1.0,8.0,1.0,1.0,1.0])

*__Size of Hash__ = 1,000 (any token will be hashed to an integer 0-999)

TF-IDF* Vectors – Word weights

- (1000,[17,63,94,197,234,335,412,437,445,521,530,556,588,673,799,893,937,960,990], [0.287,0.0,0.0,0.0,0.0,0.0,0.287,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.287])
- (1000,[17,21,22,37,63,92,167,211,240,256,270,272,393,395,445,449,460,472,480, 498,535,612,676,688,694,706,724,732,790,909,916,939,960,965,996], [0.287,0.575,0.0,0.0,0.0,0.575,0.0,0.0,0.287,0.287,0.0,0.0,0.0,0.0,0.0,0.287,0.0,0.0,0.0,0.0,0.0,0.287,0.575, 0.0,0.287,0.0,0.0,0.0,1.15,0.0,0.0,0.0,0.0,0.0,0.0])
- (1000,[21,63,92,131,143,147,196,205,208,240,250,256,265,268,296,312,326,340, 341,367,378,391,399,400,412,417,441,445,449,455,464,483,494,503,515,524,526, 539,551,575,602,612,627,641,645,656,676,694,721,742,757,767,780,786,790,802, 807,817,818,844,852,920,946,951,960,983,985,990], [0.287,0.0,0.575,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.150,0.0,1.150,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0, 0.0,0.0,0.0,0.0,0.287,0.0,0.0,0.0,0.287,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0, 0.287,0.0,0.0,0.0,0.0,0.0,0.287,0.575,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.726,0.0,0.0,0.0,0.0,0.0,0.0,0.0, 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.287])

*__Minimum Document Frequency__ = 2 (all other tokens have TF-IDF = 0)

Transforming Text into Numeric Features

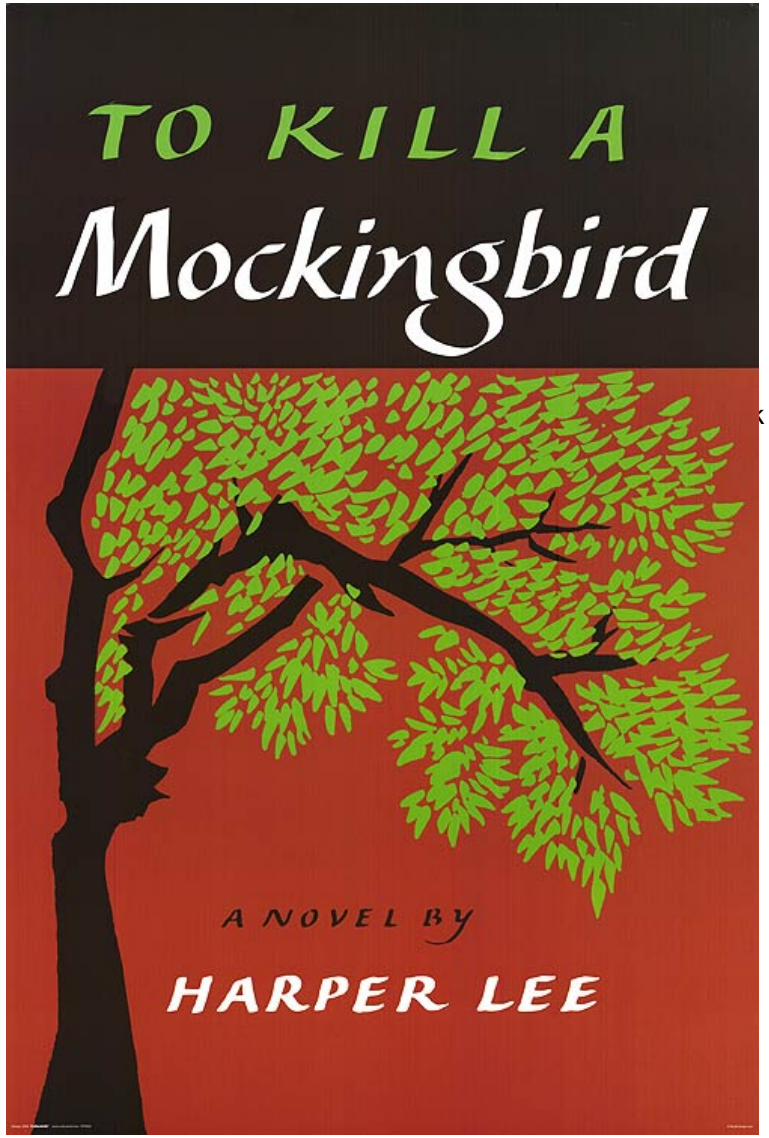
1. Use Lucene Analyzer to tokenize documents
2. Hash the tokens into sparse vectors with TF
    1. Control the vector size
    2. Smaller vectors require less memory; Larger vectors have fewer collisions
    3. Note: hashing is **one-way** (cannot convert back to tokens)
3. Build an IDF dictionary from the **training** vectors
    1. Can limit size by including *minimum document frequency*
    2. $\sqrt{(TF_w)}*\ln[(\#\ docs +1) /(doc\ freq_w+1)]$
4. Transform the TF vectors into TF-IDF Vectors

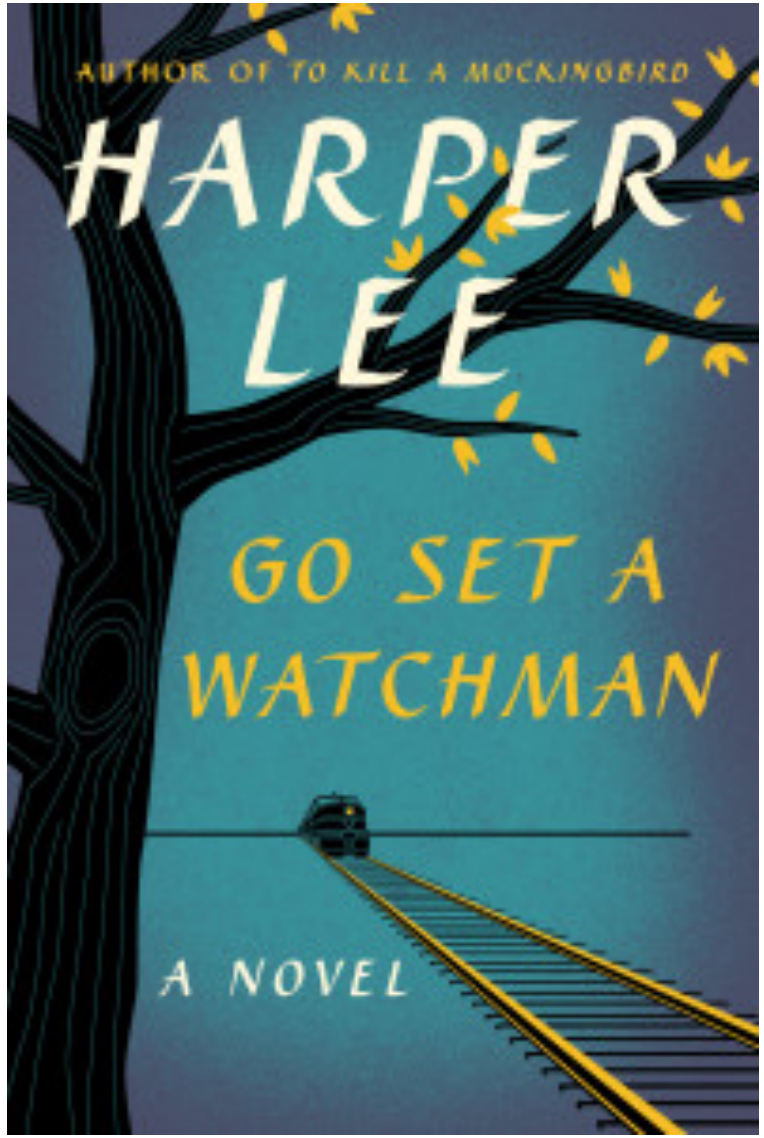
Spark Shell commands to transform text

```
import statements …
object Stemmer { …}

val getty = sc.textFile(”gettys.txt")
val stemmed = getty.map{x=>  Stemmer.tokenize(x)}
getty.collect().foreach(println)
stemmed.collect().foreach(println)

val tf = new HashingTF(1000) //size impacts memory and collisions
val tfdocs = stemmed
tfdocs.collect().foreach(println)

val idfModel = new IDF(minDocFreq = 2).fit(tfdocs)
val idfDocs = idfModel.transform(tfdocs)
idfDocs.collect().foreach(println)
```

STEP 2: Model Building

A classification problem…





Create the training and evaluation sets

```
val mock = sc.textFile(”mock.tokens")
val watch = sc.textFile(”watch.tokens”)
/// convert data to numeric features with TF
val tf = new HashingTF(10000)
val mockData = mock.map { line =>
  var target = "1"
  LabeledPoint(target.toDouble, tf.transform(line.split(",")))
}
val watchData = watch.map { line =>
  var target = "0"
  LabeledPoint(target.toDouble, tf.transform(line.split(",")))
}
/// build IDF model and transform data into modeling sets
val data = mockData.union(watchData)
val splits = data.randomSplit(Array(0.7, 0.3)) // prepare train and test sets
val trainDocs = splits(0).map{ x=>x.features}
val idfModel = new IDF(minDocFreq = 3).fit(trainDocs) // build on training data only

val train = splits(0).map{ point=>
  LabeledPoint(point.label,idfModel.transform(point.features))
}
val test = splits(1).map{ point=>
  LabeledPoint(point.label,idfModel.transform(point.features))
}
```

Naïve Bayes Algorithm

classes (aka data sources)

Likelihood calculations:

Tokens	$P(T_i \mid C_1)$	$P(T_i \mid C_2)$	…	$P(T_i \mid C_k)$
T_1	0.004	0.001		0.010
T_2	0.002	0.008		0.021
…			…	
T_n	0.014	0.002		0.003

Classification:

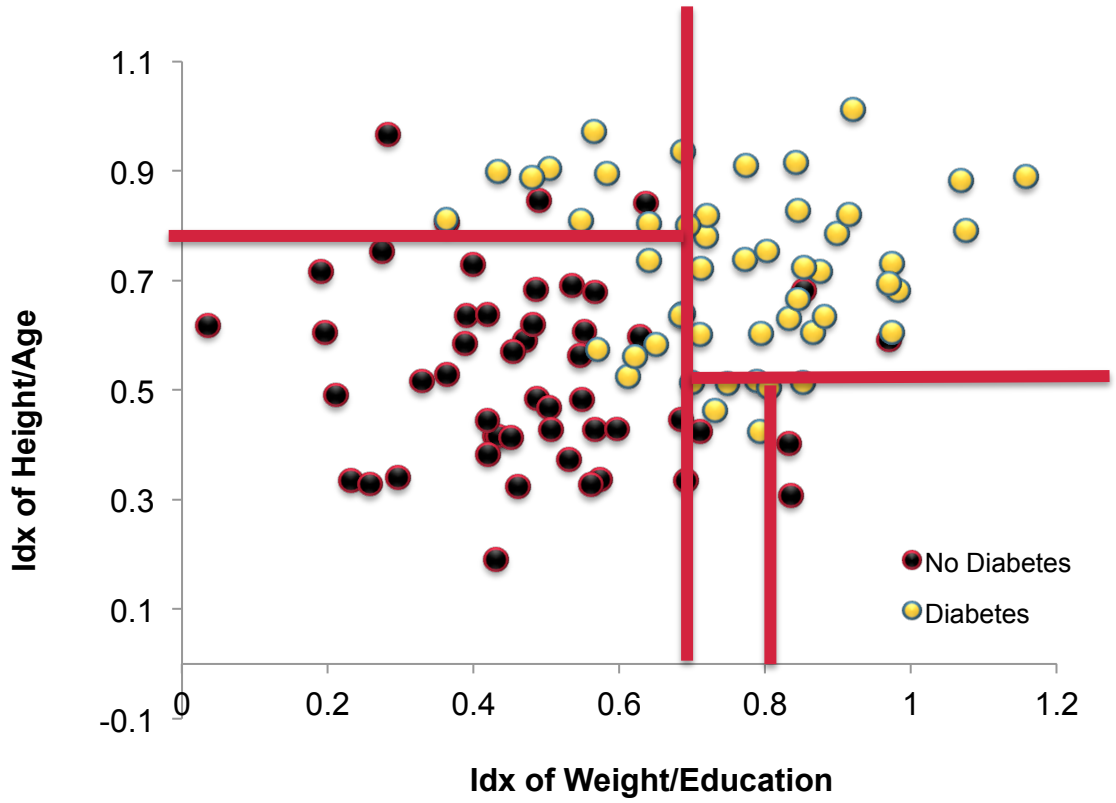
ignore prior * likelihoods

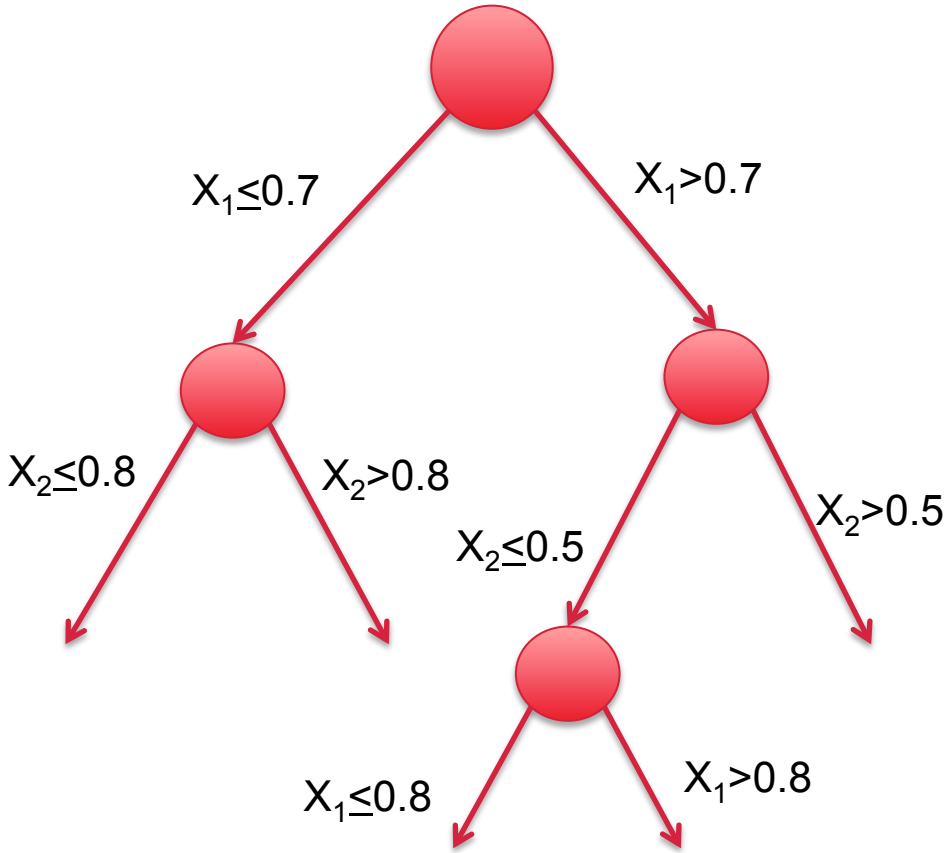
$$p(C_j \mid T_1, T_2, \ldots) = \frac{1}{Z} * p(C_j) * \prod_{i=1}^{n} p(T_i \mid C_j)$$

Run the Naïve Bayes Algorithm in Spark Shell

```
import org.apache.spark.mllib.classification.{NaiveBayes, NaiveBayesModel}

val nbmodel = NaiveBayes.train(train, lambda = 1.0)
val bayesTrain = train.map(p => (nbmodel.predict(p.features), p.label))
val bayesTest = test.map(p => (nbmodel.predict(p.features), p.label))
println("Training accuracy", bayesTrain.filter(x => x._1 == x._2).count() /
bayesTrain.count().toDouble)
println("Test accuracy    ", bayesTest.filter(x => x._1 == x._2).count() /
bayesTest.count().toDouble)

// print confusion matrix
println("Predict:mock,label:mock -> ",bayesTest.filter(x => x._1 == 1.0 & x._2==1.0).count())
println("Predict:watch,label:watch -> ",bayesTest.filter(x => x._1 == 0.0 & x._2==0.0).count())
println("Predict:mock,label:watch -> ",bayesTest.filter(x => x._1 == 1.0 & x._2==0.0).count())
println("Predict:watch,label:mock -> ",bayesTest.filter(x => x._1 == 0.0 & x._2==1.0).count())
```

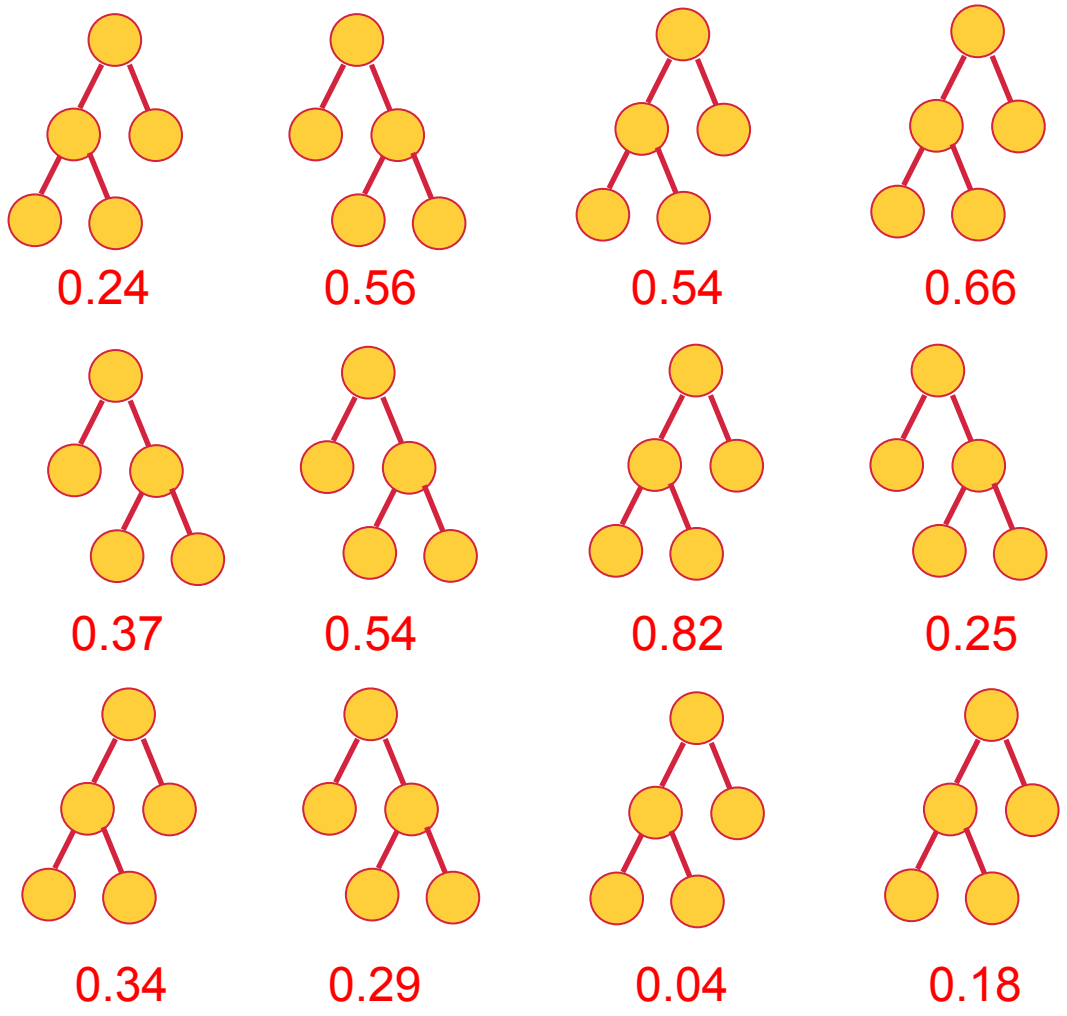
Decision Tree for Classification





Random Forest

- Aggregate estimates from many independent trees
- Key parameters
  - number of trees
  - maximum tree depth
- Notes
  - Randomness in training and feature subsets
  - more trees decreases over-fitting
  - trees are built in parallel
  - depth is generally ***larger*** than GBT



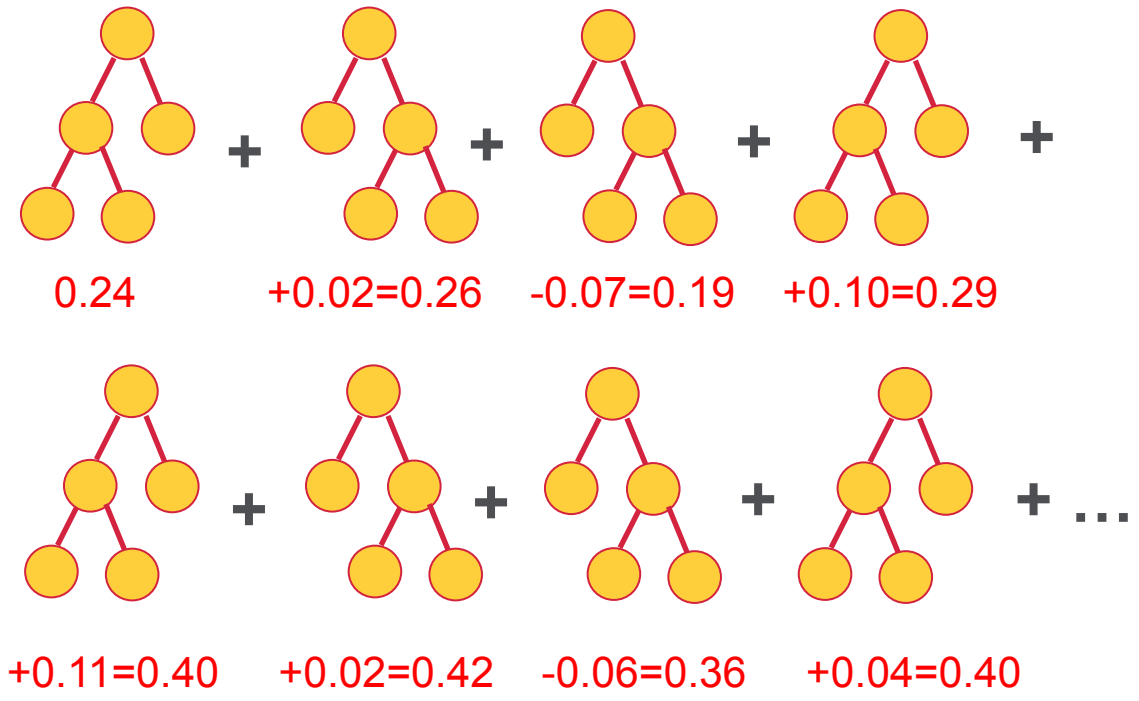
estimate = avg(T_i) = 0.40

Train a Random Forest Model in Spark

```
import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.model.RandomForestModel

val categoricalFeaturesInfo = Map[Int, Int]()
val numClasses = 2
val featureSubsetStrategy = "auto”.
val impurity = "variance” // tells Spark we want regression, not classification
val maxDepth = 10
val maxBins = 32
val numTrees = 50

val modelRF = RandomForest.trainRegressor(train, categoricalFeaturesInfo, numTrees,
featureSubsetStrategy, impurity, maxDepth, maxBins)
val trainScores = train.map { point =>
  val prediction = modelRF.predict(point.features)
  (prediction, point.label)
}
val testScores = test.map { point =>
  val prediction = modelRF.predict(point.features)
  (prediction, point.label)
}
```

Gradient Boosted Trees

- Successive trees attempt to minimize error by focusing on large residuals
- Key parameters
  - number of trees
  - maximum tree depth
- Notes
  - more trees ***increases*** over-fitting
  - trees are ***not*** built in parallel
  - depth is generally ***smaller*** than Random Forest



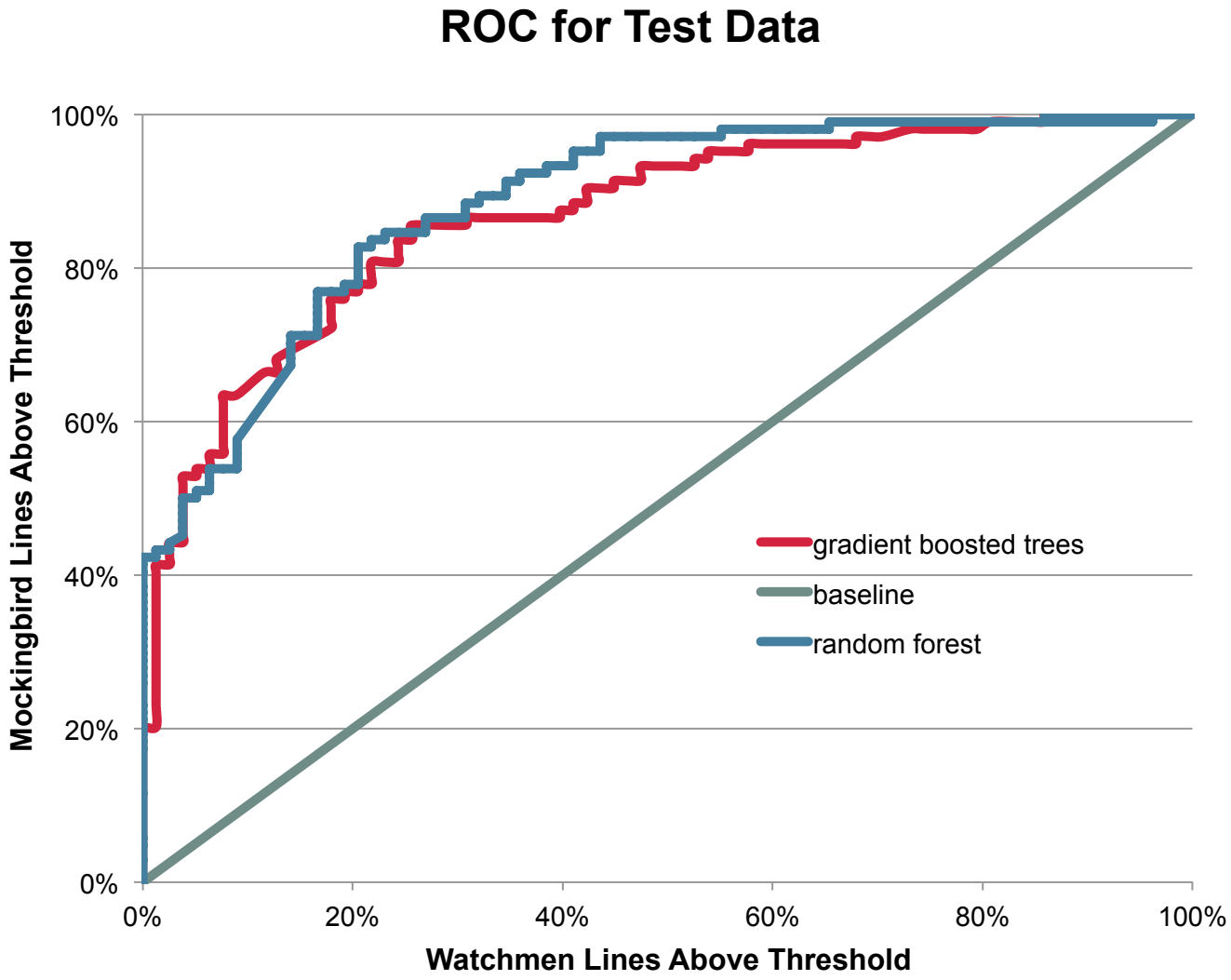
Train a GBTree Model in Spark

```
import org.apache.spark.mllib.tree.GradientBoostedTrees
import org.apache.spark.mllib.tree.configuration.BoostingStrategy
import org.apache.spark.mllib.tree.model.GradientBoostedTreesModel

val boostingStrategy = BoostingStrategy.defaultParams("Regression") // squared-error loss
boostingStrategy.numIterations = 50
boostingStrategy.treeStrategy.maxDepth = 5
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]()

val modelGB = GradientBoostedTrees.train(train, boostingStrategy)

val trainScores = train.map { point =>
  val prediction = modelGB.predict(point.features)
  (prediction, point.label)
}
val testScores = test.map { point =>
  val prediction = modelGB.predict(point.features)
  (prediction, point.label)
}
```

Looking at the ROC

1. Order results by descending score (i.e. threshold), optionally put into ordered bins
2. Calculate cumulative percent of targets
3. Calculate cumulative percent of non-targets

$AuROC_{RF}$= 0.884
$AuROC_{GB}$= 0.867
KS_{RF} = 0.62
KS_{GB} = 0.60

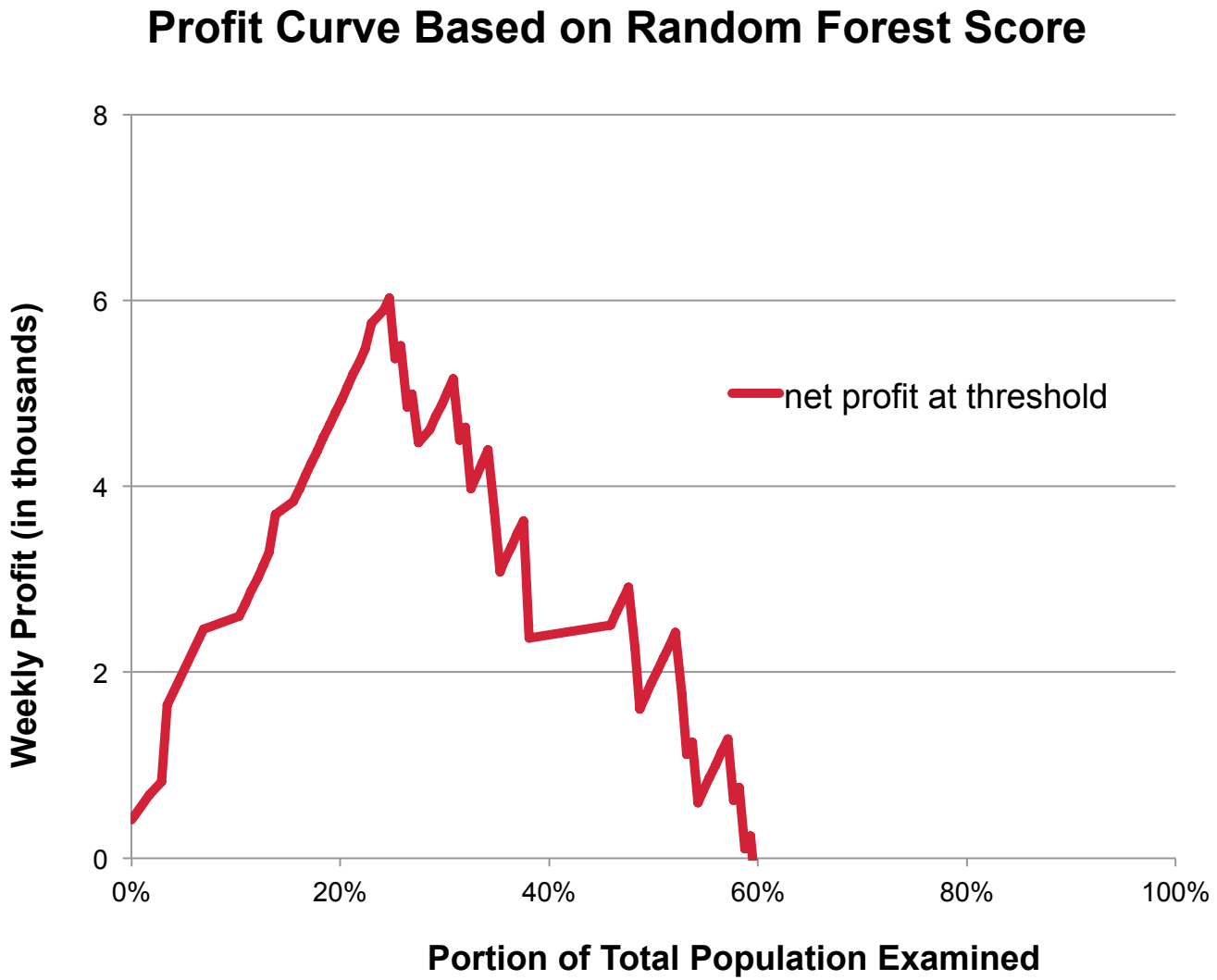


STEP 3: Value from Operational Constraints

From ROC to Value

- Start with highest scores
- Assign $ values
  - Target identification
  - Operational cost
- Display profit @ each threshold

Example:
Identify target: $950
Operational cost: $800



Find my presentation and other related resources here:

http://events.mapr.com/AtlantaHUG155

(you can find this link in the event's page at meetup.com)


Today's Presentation


Free On-Demand Training


Whiteboard & demo videos


Free eBooks


Free Hadoop Sandbox


And more…

MAPR®

Q&A

Engage with us!

@mapr   maprtech

mapr-technologies   MapR

sales@mapr.com   maprtech

Appendix

Glossary

- **RDD (Resilient Distributed Dataset)**
  - in Apache Spark, an immutable, partitioned collection of elements that can be operated on in parallel
- **Supervised Modeling**
  - family of modeling algorithms which use labeled data and thus have an error to minimize. Contrast with unsupervised methods
- **Overtraining**
  - Extra performance on the training set that is due to memorization of the training data rather than learning the true patterns
- **ROC**
  - Receiver Operating Characteristic. Measure performance by plotting cumulative false positives vs cumulative true positives over score bins
- **KS**
  - Kolmogorov-Smirnov coefficient = maximum separation of the ROC with baseline, usually reported * 100.

“Stemmer” (Tokenizer) object

```
import org.apache.lucene.analysis.en.EnglishAnalyzer
import org.apache.lucene.analysis.tokenattributes.CharTermAttribute
import scala.collection.mutable.ArrayBuffer
object Stemmer {
  def tokenize(content:String):Seq[String]={
    val analyzer=new EnglishAnalyzer()
    val tokenStream=analyzer.tokenStream("contents", content)
    val term=tokenStream.addAttribute(classOf[CharTermAttribute])
    tokenStream.reset()

    var result = ArrayBuffer.empty[String]
    while(tokenStream.incrementToken()) {
        val termValue = term.toString
        if (!(termValue matches ".*[\\d\\.].*")) {
          result += term.toString
        } }
    tokenStream.end()
    tokenStream.close()
    result
  } }
```

Launch the shell with this command:
./bin/spark-shell --packages "org.apache.lucene:lucene-analyzers-common:5.1.0”

This code originally appeared here:
https://chimpler.wordpress.com/2014/06/11/classifiying-documents-using-naive-bayes-on-apache-spark-mllib/

Code to produce and write an ROC

```
//// create RDD's of predictions and labels
val trainScores = train.map { point =>
  val prediction = modelGB.predict(point.features)
  (prediction, point.label)
}
val testScores = test.map { point =>
  val prediction = modelGB.predict(point.features)
  (prediction, point.label)
}

//// generate ROC's and write to file - will produce error if destination exists
val metricsTrain = new BinaryClassificationMetrics(trainScores,100)
val metricsTest = new BinaryClassificationMetrics(testScores,100)
val trainroc= metricsTrain.roc()
val testroc= metricsTest.roc()
trainroc.saveAsTextFile("./ROC/gbtrain")
testroc.saveAsTextFile("./ROC/gbtest")
```