Prediction model with decision tree

TANG Gen

Outline

- Preparing the data
- A first decision tree
- Grid search
  - Decision tree hyper - parameters
  - Tuning decision trees
- Decision tree with categorical variable

Preparing the data

/Introduction to dataset

- The dataset that we used is available at
https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/

- The data set records the types of forest covering parcels of land in Colorado, USA

- The forest cover type is to be predicted from the rest of the features, of which there are 54 in total

- The forest features are in 1 - 54 column, containing both categorical and numeric features

- The forest cover type is in the last column

Preparing the data

/Categorical variable vs. numerical variable

- A numerical variable is an observed response that is a numerical value
- A categorical variable is a variable that can take on one of a limited, and usually fixed, number of possible values, thus assigning each individual to a particular group or “category.”
  - For example, sex, city and etc.
  - A categorical variable can not be used without appropriate encoding
  - The most common used method is one-hot encoding

Preparing the data

/Categorical variable vs. numerical variable

	Sex
A	M
B	F
C	M
D	F

one-hot encoding

	Is_man	Is_female
A	1	0
B	0	1
C	1	0
D	0	1

Preparing the data

/Get familiar with data

- The column 11 - 14 contains the one-hot encoding of variable wilderness area designation

- The column 15 - 54 contains the one-hot encoding of variable soil type designation

```
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.regression._


val rawData = sc.textFile("file:/Users/tgbaggio/AAspark/chapter4/data/covtype.data")

rawData.first()
rawData.count()

// Take a look at the label variable
val label = rawData.map(line => line.split(",")).map(_.last)
label.distinct().collect()
```

Preparing the data

/Transformation to labeled data

- The Spark MLlib abstraction for a feature vector is known as a **LabeledPoint**, which consists of a Spark MLlib Vector of features, and a target value, here called the **label**

```
val data = rawData.map { line =>
  val values = line.split(',').map(_.toDouble)
  val featureVector = Vectors.dense(values.init)
  val label = values.last - 1
  LabeledPoint(label, featureVector)
}
```

- We split the data into three subsets: training, cross-validation and test

```
val Array(trainData, cvData, testData) = data.randomSplit(Array(0.8, 0.1, 0.1))

trainData.cache()
cvData.cache()
testData.cache()
```

A first decision tree

/Model and evaluation metrics

- **MulticlassMetrics** computes standard metrics that in different ways measure the quality of the predictions from a classifier, which here has been run on the CV set

```
import org.apache.spark.mllib.evaluation._
import org.apache.spark.mllib.tree._
import org.apache.spark.mllib.tree.model._
import org.apache.spark.rdd.RDD

// A first decision tree
val model = DecisionTree.trainClassifier( trainData, 7, Map[Int,Int](), "gini", 4, 100)

def getMetrics(model: DecisionTreeModel, data: RDD[LabeledPoint]): MulticlassMetrics = {
  val predictionsAndLabels = data.map(example =>
      (model.predict(example.features), example.label)
      )
  new MulticlassMetrics(predictionsAndLabels)
}

val metrics = getMetrics(model, cvData)
```

A first decision tree

/Model and evaluation metrics

- The details of metrics of the model

```
// Take a look at the metrics of model
metrics.confusionMatrix
metrics.precision

(0 until 7).map(
  cat => metrics.precision(cat)
).foreach(println)
```

```
scala> metrics.confusionMatrix
14010.0  6784.0   4.0     0.0   0.0  1.0   325.0
5448.0   22846.0  323.0   12.0  0.0  12.0  27.0
0.0      766.0    2684.0  66.0  0.0  13.0  0.0
0.0      1.0      155.0   98.0  0.0  0.0   0.0
0.0      927.0    4.0     3.0   0.0  0.0   0.0
0.0      532.0    1093.0  38.0  0.0  59.0  0.0
1149.0   31.0     0.0     0.0   0.0  0.0   898.0
```

A first decision tree

/Baseline model - random guess

- The first decision tree shows that about 70% of examples were classified correctly
- Although 70% accuracy sounds decent, it’s not immediately clear whether it is out- standing or poor accuracy
- Therefore, we need to establish a simplistic approach, a baseline to compare this model
- Consider a random guess classifier
    - A class that makes up 20% of the training set and 10% of the CV set will contribute 20% of 10%, or 2%, to the overall accuracy

A first decision tree

/Baseline model - random guess

```
//Baseline model
import org.apache.spark.rdd._

def classProbabilities(data: RDD[LabeledPoint]): Array[Double] = {
  val countsByCategory = data.map(_.label).countByValue()
  val counts = countsByCategory.toArray.sortBy(_._1).map(_._2)
  counts.map(_.toDouble / counts.sum)
}

val trainPriorProbabilities = classProbabilities(trainData)
val cvPriorProbabilities = classProbabilities(cvData)

trainPriorProbabilities.zip(cvPriorProbabilities).map {
  case (trainProb, cvProb) => trainProb * cvProb
}.sum
```

Grid search

/Decision tree hyper-parameters

- Maximum depth simply limits the number of levels in the decision tree. It is useful to avoid overfitting problem

- At the each node of decision tree, the algorithm search for an optimal decision rule, such as weight >= 100 or weight >= 500 etc.

    - Maximum bins limits the number of tries at each node

- In decision tree, the parameter “impurity” is to measure the quality of a decision rule.

$$I_G(p) = 1 - \sum_{i=1}^{N} p_i^2$$

Gini Impurity equation

$$I_E(p) = \sum_{i=1}^{N} p_i \log\left(\frac{1}{p}\right) = - \sum_{i=1}^{N} p_i \log(p_i)$$

Entropy

Grid search

/Tuning decision trees

```
// Tuning decision trees

val evaluations =
  for (impurity <- Array("gini", "entropy");
       depth <- Array(1, 20);
       bins <- Array(10, 300)) yield {
         val model = DecisionTree.trainClassifier(
           trainData, 7, Map[Int,Int](), impurity, depth, bins)
         val predictionsAndLabels = cvData.map(example =>
             (model.predict(example.features), example.label))
         val accuracy =
           new MulticlassMetrics(predictionsAndLabels).precision
         ((impurity, depth, bins), accuracy)
       }

evaluations.sortBy(_._2).reverse.foreach(println)
```

Decision trees with categorical variable

/Categorical features revisited

- As we discussed before, we can use categorical variable after one-hot encoding it
- However, with N-vlaued categorical variable, we need to create N numeric variable, which would increases memory usage and slows things down
- In MLlib, some algorithms can handle categorical variable directly, including decision trees
- In the following slide, we will show how to recreate “wilderness” and “soil” these two categorical variables and use them in decision trees

Decision trees with categorical variable

/Categorical features revisited

```
// Revisited categorical variable

val data = rawData.map { line =>
  val values = line.split(',').map(_.toDouble)
  val wilderness = values.slice(10, 14).indexOf(1.0).toDouble
  val soil = values.slice(14, 54).indexOf(1.0).toDouble
  val featureVector = Vectors.dense(values.slice(0, 10) :+ wilderness :+ soil)
  val label = values.last - 1
  LabeledPoint(label, featureVector)
}
```

	Is_man	Is_female
A	1	0
B	0	1
C	1	0
D	0	1

	Sex
A	0
B	1
C	0
D	1

Decision trees with categorical variable

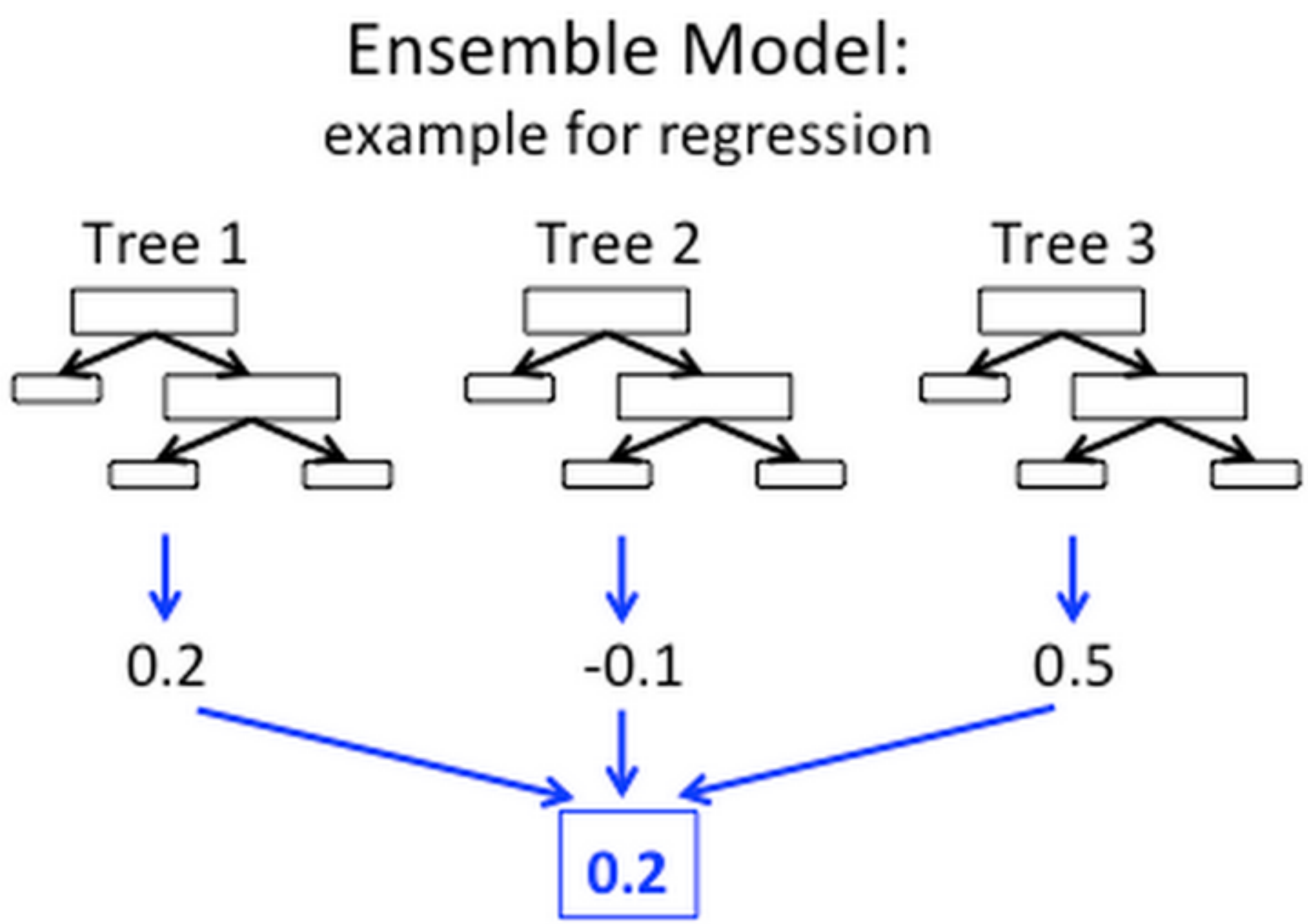
/Categorical features revisited

```
val evaluations =
  for (impurity <- Array("gini", "entropy");
       depth <- Array(10, 20);
       bins <- Array(10, 30))
yield {
  val model = DecisionTree.trainClassifier( trainData, 7, Map(10 -> 4, 11 -> 40), impurity, dept
h, bins)
  val trainAccuracy = getMetrics(model, trainData).precision val cvAccuracy = getMetrics(model,
cvData).precision ((impurity, depth, bins), (trainAccuracy, cvAccuracy))
}
```

- Where Map(column -> value) indicates the categorical variable; "column" indicates the position of categorical variable and "value" indicates how many value in this variable

Case studies

/Spark at carrefour



Case studies

/Spark at carrefour

- 14 millions clients and 30 000 products
- About 10T data and hundreds tables to join
- Core recommendation system built on Spark is back for all marketing operations