Big Data Analytics

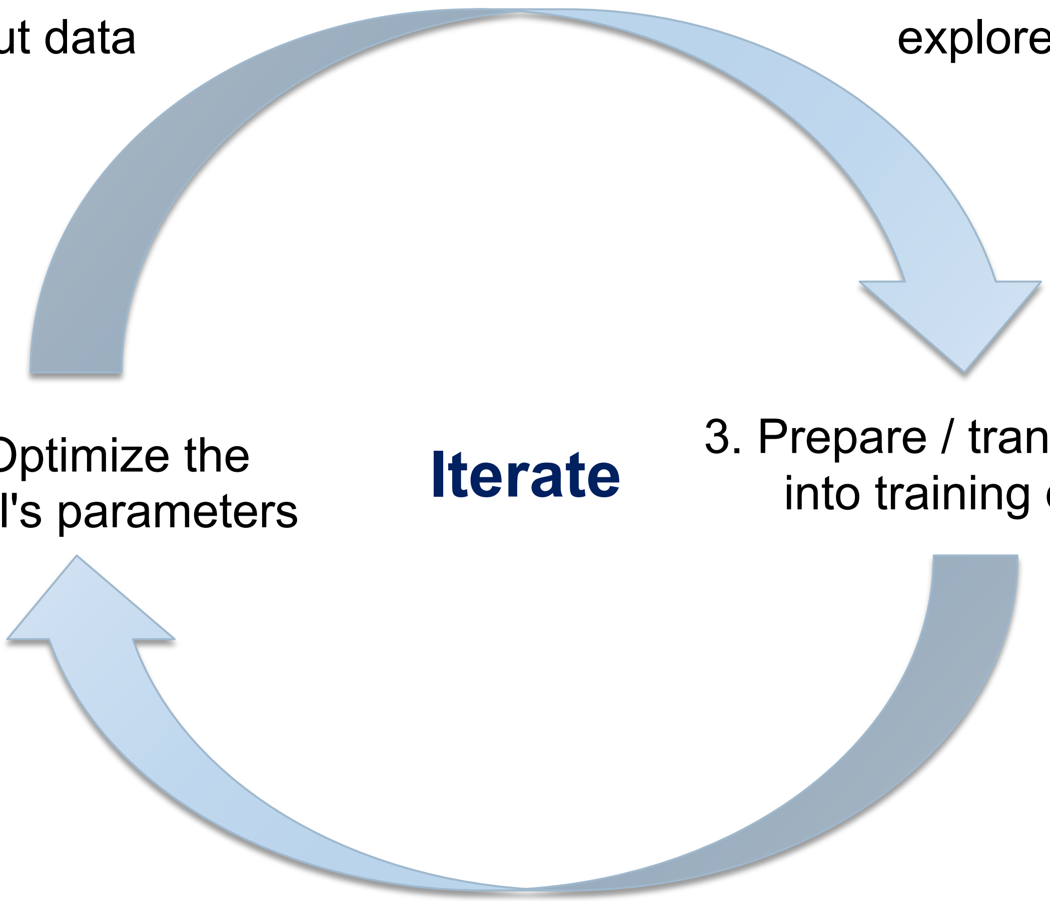
Chapter 2: Predictive Data Analysis with Decision Trees & Random Forests

Following: [3] "**Advanced Analytics with Spark**", Chapter 4

Prof. Dr. Martin Theobald
Faculty of Science, Technology & Communication
University of Luxembourg

uni.lu
UNIVERSITÉ DU LUXEMBOURG

Typical Data Science Workflow


1. Load the raw input data
2. Manually inspect / explore the raw data
3. Prepare / transform raw data into training data format
4. Train & verify the prediction model
5. Make prediction / perform advanced analytical tasks
6. Optimize the model's parameters
Iterate

Fast-Forward from Regression to Classification

Regression denotes the broad task of analyzing the dependencies among a **number of input variables** (either categorical or numerical), typically in order to predict the value (then also called the "label") of a **single output variable**.
(see the `LinearRegression` example from Chapter 1.2)

More specifically, the task of predicting the **label of a new object** based on the **labels of a set of training objects** belongs to the class of **supervised learning techniques**.

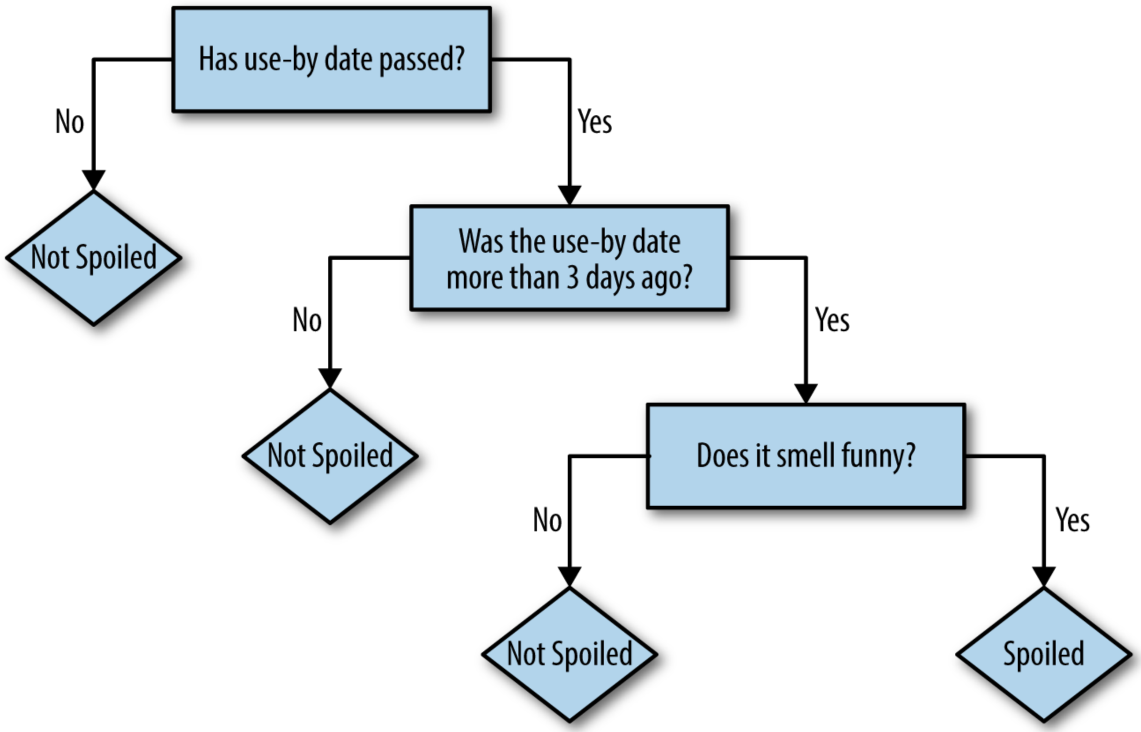
Classification is a supervised learning technique that usually **predicts a categorical label**, such as the type or class that a new object may belong to.

Decision trees are a simple (both intuitive and effective) form of classification which can handle both numerical and categorical data, typically by imposing a

- a **binary decision** at each inner node of the tree, thus predicting
- a **single label** for each object at a leaf node of the tree.

Random forests are sets of (more or less "randomly" generated) decision trees. The prediction then is usually made by a **majority vote** among the individual trees.

First Decision Tree Example



- Here, the products are supposed to be classified into the two categories "**Spoiled**" and "**Not Spoiled**" based on their "use-by date" and whether they "smell funny".
- However, this may not be the best-possible decision tree for this kind of prediction:
  - "Was the use-by date more than 3 days ago?" condition completely subsumes the "Has the use-by date passed?" condition.
  - But what if something smells funny even before the use-by date has passed?

The CoverType Data Set

The **CoverType data set** records the various types of forests that cover parcels of land in Colorado (as of 1998).

https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/

The main data file, covtype.data.gz (11 MB), consists of 518,012 parcels with 54 attributes (both numerical and categorical) that describe each parcel, including its *elevation*, *slope*, *distance to water*, *shade*, *soil type*, etc., along with the *type of forest* that covers the parcel.

The second file, covtype.info, contains a description of the attributes and their measurements in covtype.data.gz.

- The first 10 columns are **quantitative measurements** (of various units).
- The following 4 columns are (binary) **qualitative measurements** that describe the location of the wilderness area.
- The following 40 columns are (binary) **qualitative measurements** that describe the soil type.

CoverType is a very well-studied data set in many predictive-analysis and machine-learning applications.

- Kaggle includes a separate competition based on the CoverType data set.

Outline of Chapter 2

2.1 Decision Trees & Random Forests

Labeled Feature Vectors

- Consider the following table of **labeled training data** about pets for kids:

ID	Pet Name	Weight (kg)	Number of Legs	Color	Suitable for Kids?
1	Fido	20.5	4	Brown	**Yes**
2	Mr. Slither	3.1	0	Green	**No**
3	Nemo	0.2	0	Tan	**Yes**
4	Dumbo	1,390.8	4	Grey	**No**
5	Kitty	12.1	4	Grey	**Yes**
6	Jim	150.9	2	Tan	**No**
7	Millie	0.1	100	Brown	**No**
8	McPigeon	1.0	2	Grey	**No**
9	Spot	10.0	4	Brown	**Yes**

- Every row encodes a **vector of attribute values** that describe the pet.
- "**Suitable for Kids?**" is a distinguished attribute (i.e., the **label**) whose value we would like to predict also for new pets we have not seen yet.

Decision Tree

A **decision tree** is a tree-structured classifier whose inner nodes (including the root) encode **decision rules** and whose leafs represent the **labels** we wish to assign to the **conjunction of decisions** made under each root-to-leaf path in the tree.

A **binary decision tree** is a decision tree whose decision rules are all binary.

Both the decision rules and the labels at the leafs of the tree are not necessarily disjoint. Thus, sometimes decision trees are compacted into a **directed-acyclic graph** (DAG), then called a **decision diagram**.

<u>Goals in learning a decision-tree classifier:</u>

- A decision tree should **separate the training data well** according to the provided labels at each of its leaf nodes.
- Decision trees should **not be too deep** nor contain too many **redundant** subtrees (and hence avoid "overfitting" to the training data).
- Decision trees should usually be **balanced**.

Precision, Recall & Accuracy (Binary Classification)

For **binary classification** tasks, i.e., when all items to be classified carry only one out of **two possible labels** (such as Yes/No), we distinguish among the following **four classes of predictions** made by the classifier:

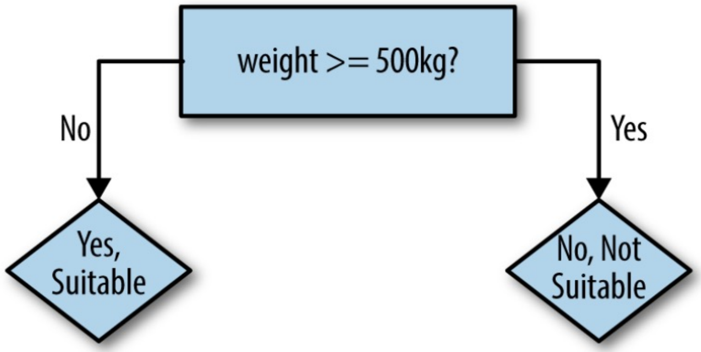
		Actual (Given) Label	
		Yes	No
Predicted Label	Yes	True Positive (TP)	False Positive (FP)
	No	False Negative (FN)	True Negative (TN)

Explanation:

- **True**: correct prediction by the system
- **False**: wrong prediction by the system
- **Positive**: an item predicted by the system
- **Negative**: an item missed by the system

From the above definitions of TP, TN, FP & FN, we can define the three principal quality measures **precision**, **recall** & **accuracy** (see next slides).

Small Decision Tree Example



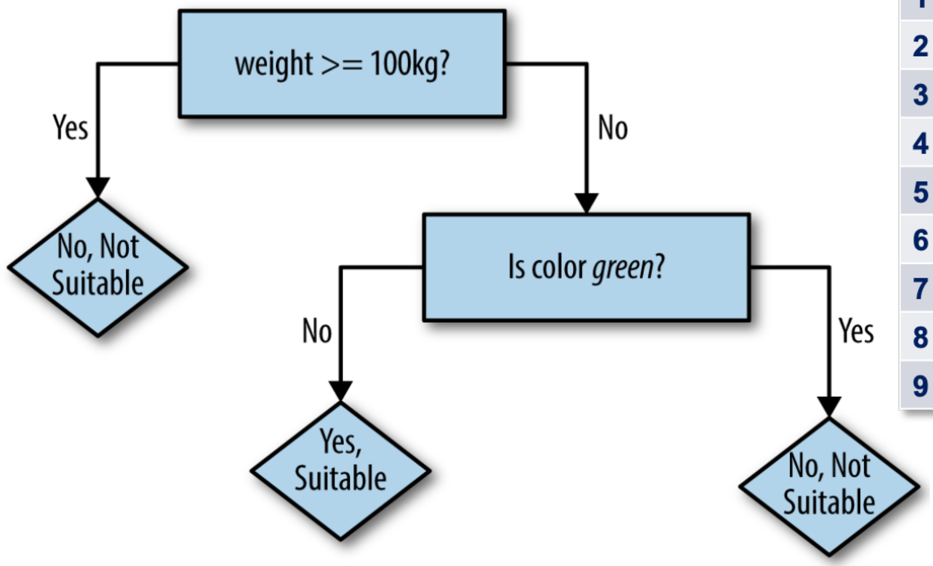
ID	Pet Name	Weight (kg)	Number of Legs	Color	Suitable for Kids?
1	Fido	20.5	4	Brown	Yes
2	Mr. Slither	3.1	0	Green	No
3	Nemo	0.2	0	Tan	Yes
4	Dumbo	1,390.8	4	Grey	No
5	Kitty	12.1	4	Grey	Yes
6	Jim	150.9	2	Tan	No
7	Millie	0.1	100	Brown	No
8	McPigeon	1.0	2	Grey	No
9	Spot	10.0	4	Brown	Yes

- The above decision tree classifies the data of Slide 8 with respect to being in the class "**Yes, Suitable**" into the following four categories:
  - **True Positives** (TP): 1, 3, 5, 9
  - **True Negatives** (TN): 4
  - **False Positives** (FP): 2, 6, 7, 8
  - **False Negatives** (FN): none
- Thus, we obtain a **precision** of $prec = \frac{|TP|}{|TP|+|FP|} = \frac{1}{2}$,

  a **recall** of $rec = \frac{|TP|}{|TP|+|FN|} = 1$,

  and an **accuracy** of $acc = \frac{|TP|+|TN|}{|TP|+|TN|+|FP|+|FN|} = \frac{5}{9}$.

Larger Decision Tree Example



ID	Pet Name	Weight (kg)	Number of Legs	Color	Suitable for Kids?
1	Fido	20.5	4	Brown	Yes
2	Mr. Slither	3.1	0	Green	No
3	Nemo	0.2	0	Tan	Yes
4	Dumbo	1,390.8	4	Grey	No
5	Kitty	12.1	4	Grey	Yes
6	Jim	150.9	2	Tan	No
7	Millie	0.1	100	Brown	No
8	McPigeon	1.0	2	Grey	No
9	Spot	10.0	4	Brown	Yes

- This decision tree classifies the data of Slide 8 with respect to being "**Suitable for Kids?**" into the following categories:
  - **True Positives** (TP): 1, 3, 5, 9
  - **True Negatives** (TN): 2, 4, 6
  - **False Positives** (FP): 7, 8
  - **False Negatives** (FN): none
- This time, we obtain a **precision** of $prec = \frac{|TP|}{|TP|+|FP|} = \frac{2}{3}$, a **recall** of $rec = \frac{|TP|}{|TP|+|FN|} = 1$, and an **accuracy** of $acc = \frac{|TP|+|TN|}{|TP|+|TN|+|FP|+|FN|} = \frac{7}{9}$.

Basic Issues

- For a categorical attribute with n possible values, we might have to consider $\mathbf{2^n - 1}$ **possible binary decision rules** for learning the best decision tree, i.e., the one that best separates the training data according to the given labels.

- For m categorical attributes, each with n possible values, we might even obtain $\mathbf{2^{m+n} - 1}$ **possible binary decision rules**.

- For numerical attributes, there might be **uncountably many** such rules.

- Even if we managed to find the best decision tree, it might end up to be completely **overfitted to the training data** but does not well classify new objects with an unknown label.

Finding a Good Splitting Condition

- Suppose that we split a **set of labeled training vectors** D vectors into **two subsets** D_{left} and D_{right}.
- Ideally, each of the two subsets would contain just one distinct label, i.e., they would be completely "**pure**".
- We can thus measure the "**gain in purity**" ΔI based on the (im-)purity of the sets before and after the splits:

$$\Delta I = I(D) - I(D_{left}) - I(D_{right})$$

- Since the three sets are not of equal size, we should **normalize** their impact on this gain in purity:

$$I(D) = \frac{|D|}{N} I_{G|E}(D)$$

Where N is the total number of labeled feature vectors in the entire training set.

- Usual choices for **measuring purity** which applied in decision-tree learning are:

Gini purity: $I_G(D) = 1 - \sum_{i=1}^{n} f_i^2$

Entropy: $I_E(D) = -\sum_{i=1}^{n} f_i \log f_i$

Where n is the number of distinct labels and f_i is the relative frequency of label i in D.

Learning a Decision Tree

Greedy algorithm for learning a **binary decision tree** (aka. "**C4.5**"):

1. Let D denote the set of **labeled training vectors** (having n distinct labels)
2. For each **categorical attribute** A of vectors in D
   - For each distinct value v of A
     - Compute the gain ΔI in Gini purity or entropy by splitting D into D_{left} and D_{right} using $A = v$ as decision rule
3. For each **numerical attribute** A of vectors in D
   - Discretize the range of A into m uni-width bins
   - For each center v of a uni-width bin for A
     - Compute the gain ΔI in Gini purity or entropy by splitting D into D_{left} and D_{right} using $A < v$ as decision rule
4. Let $(A, v)^{best}$ denote the splitting condition that maximizes the gain ΔI
5. Repeat from step 1 for both D_{left} and D_{right} based on $(A, v)^{best}$ as splitting condition until $maxdepth$ or a gain of $\Delta I \leq 0$ is reached
6. Add the **most common label** in D **as leaf**

Example: Splitting Conditions

Let $I(D) = \frac{9}{9}\left(1 - \left(\left(\frac{4}{9}\right)^2 + \left(\frac{5}{9}\right)^2\right)\right) \approx 0.494$

- Consider a split on $weight\ < \ 500$:

$$I(D_{left}) = \frac{8}{9}\left(1 - \left(\left(\frac{4}{8}\right)^2 + \left(\frac{4}{8}\right)^2\right)\right) \approx 0.444$$

$$I(D_{right}) = \frac{1}{9}\left(1 - \left(\frac{1}{1}\right)^2\right) = 0$$

That is: $\Delta I \approx 0.05$

- Consider a split on $weight\ < 100$:

$$I(D_{left}) = \frac{7}{9}\left(1 - \left(\left(\frac{3}{7}\right)^2 + \left(\frac{4}{7}\right)^2\right)\right) \approx 0.381$$

$$I(D_{right}) = \frac{2}{9}\left(1 - \left(\frac{2}{2}\right)^2\right) = 0$$

That is: $\Delta I \approx 0.113$

ID	Weight (kg)	Suitable for Kids?
1	20.5	Yes
2	3.1	No
3	0.2	Yes
4	1,390.8	No
5	12.1	Yes
6	150.9	No
7	0.1	No
8	1.0	No
9	10.0	Yes

Example: Splitting Conditions (ct'd)

Let $I(D') = \frac{7}{9}\left(1 - \left(\left(\frac{3}{7}\right)^2 + \left(\frac{4}{7}\right)^2\right)\right) \approx 0.381$

- Consider a split on $color = Green$:

$$I(D'_{left}) = \frac{1}{9}\left(1 - \left(\frac{1}{1}\right)^2\right) = 0$$

$$I(D'_{right}) = \frac{6}{9}\left(1 - \left(\left(\frac{4}{6}\right)^2 + \left(\frac{2}{6}\right)^2\right)\right) \approx 0.296$$

That is: $\Delta I \approx 0.085$

- Consider a split on $color = Brown$:

$$I(D'_{left}) = \frac{3}{9}\left(1 - \left(\left(\frac{2}{3}\right)^2 + \left(\frac{1}{3}\right)^2\right)\right) \approx 0.148$$

$$I(D'_{right}) = \frac{4}{9}\left(1 - \left(\left(\frac{2}{4}\right)^2 + \left(\frac{2}{4}\right)^2\right)\right) = 0.222$$

That is: $\Delta I \approx 0.011$

ID	Color	Suitable for Kids?
1	Brown	Yes
2	Green	No
3	Tan	Yes
5	Grey	Yes
7	Brown	No
8	Grey	No
9	Brown	Yes

Random Forests

- When initialized on the complete set of labeled feature vectors as input, the C4.5 algorithm of Slide 15 behaves in a **completely deterministic** manner.
- Thus, although it is a greedy algorithm, i.e., it may not find the truly best decision tree that minimizes classification mistakes, it will always return the same tree with the same decision rules.

- There are two basic ways how to **randomize C4.5**:
  1. Consider a **random** subset A' of **attributes** in D (with replacement) for finding the splitting conditions.
  2. Consider a **random** subset of **labeled feature vectors** D (with replacement) for learning the decision tree.

- **Label prediction** with a random forest then simply works by choosing the
  - **arithmetic mean** of the individual tree predictions for a numerical label, or by
  - **majority vote** among the individual tree predictions for a categorical label.

- Randomization helps us to **avoid overfitting**!

- Multiple decision trees can be learned **in parallel**!

Decision Trees for Regression

- Decision Trees and Random Forests can be used to **approximate an arbitrary regression function** that then predicts a **numerical label** instead of a categorical one.

- Only **numerical attributes** are allowed in this case as input. However, a same attribute may **occur repeatedly** at different levels or in different subtrees of each of the learned decision trees.

- The main difference to classification is the choice of the splitting condition for each numerical attribute, which usually is based on **minimizing the sample variance**:

$$\mathrm{Var}(D) = \frac{1}{|D|}\sum_{i=1}^{|D|}(y_i - \mu)^2$$

  Where y_i is the label of an instance in D and $\mu = \frac{1}{|D|}\sum_{i=1}^{|D|} y_i$ is the **sample mean** of all instances in D.

- **Prediction** in a random forest then again calculates the average label (i.e., the arithmetic mean) among the individual tree predictions.

2.2 Decision Trees in Spark's MLlib

Prepare the Training Data (I)

- Download and copy the covtype.data file into a local directory and/or your home directory on the IRIS cluster.
- The file covtype.info provides a concise description of the data, so we may refrain from collecting more statistics about the data at this point.
- Instead, directly load the data file into an RDD and transform the data into an array of `LabeledPoint` objects (much like in the `LinearRegression` examples shown earlier).

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

val rawData = sc.textFile("./covtype.data")
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) }
```

Prepare the Training Data (II)

- `values.init` returns all but the last values of the array.
- `values.last` returns the last value of the array. We subtract 1 from this value, since the MLlib decision-tree implementation needs numerical labels that start at 0.
- Note that the data set contains 4 binary attributes for the wilderness area and 40 binary attributes for the soil type, of which only 1 each is active at a time (this is also called a "**1-hot encoding**").
- This encoding may or may not be a good choice for learning decision trees, as it effectively turns these numerical attributes into categorical ones. We will revisit this issue later…

Prepare the Training Data (III)

- We will work on three splits of sizes 80%/10%/10% of the raw data, called **training set**, **validation set**, and actual **testing set**, in the following.

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

- We will train the model on one fixed subset, called `trainData`, and optimize the basic parameters of the model on the second set, called `valData`.
- The third set, called `testData`, will be used for the final evaluation.
- All the three RDD's are cached for later re-use.

Build a First Model

- Now we are already good to build our first decision tree in Spark:

```
import org.apache.spark.mllib.tree._
import org.apache.spark.mllib.tree.model._

val model = DecisionTree.trainClassifier(
   trainData, 7, Map[Int,Int](), "gini", 4, 100)
```

- Here, we must already specify the following parameters:
  - `7` is the number of prediction labels to expect
  - `Map[Int,Int]()` holds data about the number of distinct categories per feature (will be determined automatically if left empty)
  - `gini` is the desired Gini purity measure
  - `4` is the maximum depth of the tree
  - `100` is the maximum bin count for discretizing numerical features

Evaluate the Model

- The evaluation package of Spark's MLlib contains implementations of various **evaluation metrics**.
- We thus connect our trained model with Spark's evaluation metrics package for multi-class labels, called MulticlassMetrics.

```
import org.apache.spark.rdd._
import org.apache.spark.mllib.evaluation._

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, valData)
```

Show Precision & Recall

- Get a glance at the overall accuracy across all the 7 labels:

```
metrics.accuracy
```

- Our basic model thus already achieves an **accuracy of about 70%**.

- To get a detailed view on precision and recall for each of the 7 labels, we can do the following:

```
(0 until 7).map(
  label => (metrics.precision(label), metrics.recall(label))
).foreach(println)
```

- Note that precision, recall and accuracy are actually defined for binary classification tasks only.
- Accuracy (at least as defined in Spark's `MulticlassMetrics` package) simply counts for how many elements in `valData` the predicted label indeed matches the actual (i.e., given) label and then normalizes this count by the number of elements in `valData`.

Compare to a Reasonable Baseline

- Consider a classifier that randomly—but proportionally—picks a label according to the relative frequency of labels in the training and validation sets.

```
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 valPriorProbabilities = classProbabilities(valData)
trainPriorProbabilities.zip(valPriorProbabilities).map {
  case (trainProb, valProb) => trainProb * valProb
}.sum
```

- Random guessing achieves an **accuracy of only 37%** according to the above calculation, so our basic model already performs much better than random guessing!

Optimizing the Hyper-Parameters of the Model

```
val evaluations =
  for (impurity <- Array("gini", "entropy");
    depth <- Array(10, 20, 30);
    bins <- Array(50, 100, 300))
  yield {
    val model = DecisionTree.trainClassifier(
      trainData, 7, Map[Int,Int](), impurity, depth, bins)
    val predictionsAndLabels = valData.map(example =>
      (model.predict(example.features), example.label)
    )
    val accuracy =
      new MulticlassMetrics(predictionsAndLabels).accuracy
    ((impurity, depth, bins), accuracy) }

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

- The result is a **best accuracy value of about 91.2%** across all labels.

Transform Binary Categorical Attributes into Multivalued Ones

- Using 4 and 40 binary attributes to encode a two categorical attributes may be wasteful after all and consume a lot of time for training.
- So let's transform these into two multivalued categorical attributes as follows:

```
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)
}

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

Revalidate the Model

- Once more optimize the parameters of the model. Also explicitly provide the number of categorical values for the two new attributes:

```
val evaluations =
  for (impurity <- Array("gini", "entropy");
       depth <- Array(10, 20, 30);
       bins <- Array(50, 100, 300))
  yield {
    val model = DecisionTree.trainClassifier(
       trainData, 7, Map(10 -> 4, 11 -> 40),
       impurity, depth, bins)
    val trainAccuracy = getMetrics(model, trainData).accuracy
    val valAccuracy = getMetrics(model, valData).accuracy
    ((impurity, depth, bins), (trainAccuracy, valAccuracy))
  }
```

- This steps even yields a **final accuracy of about 94.5%** when evaluated over the `valData` split, which is a gain of 3% over the binary encoding!

Train a Random Forest

- After having optimized our data encoding and the model parameters, we finally merge the `trainData` and `valData` splits to train a **Random Forest** based on 90% of the available covtype data set.

- Based on the afore defined parameters, we may thus train an entire forest of 10 binary decision trees as follows:

```
val forest = RandomForest.trainClassifier(
  trainData.union(valData), 7, Map(10 -> 4, 11 -> 40), 10,
  "auto", "entropy", 30, 300)
```

- The `RandomForest` implementation in Spark's MLlib involves two new parameters:
  - `10` is the number of randomly generated decision trees
  - `"auto"` denotes an automatic choice of training subsets and attributes

Make a Prediction

- Finally, we can also "manually" classify a feature vector:

```
val input = "2709,125,28,67,23,3224,253,207,61,6094,0,29"
val vector = Vectors.dense(input.split(',').map(_.toDouble))

forest.predict(vector)
```

Using Random Forests for Regression

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

// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "./sample_libsvm_data.txt")
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

// Train a RandomForest model. Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val featureSubsetStrategy = "auto" // Let the algorithm choose the attributes
val impurity = "variance"; val maxDepth = 4; val maxBins = 32; val numTrees = 3

val model = RandomForest.trainRegressor(trainingData, categoricalFeaturesInfo,
  numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)

// Evaluate model on test instances and compute test error
val labelsAndPredictions = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction) }
val testMSE = labelsAndPredictions.map{ case(v, p) => math.pow((v - p), 2)}.mean()
```

See: https://spark.apache.org/docs/2.2.0/mllib-ensembles.html

Extensions

- **More Tuning**
  - Systematically explore a larger parameter space.
  - Transform dependent features into a more expressive form (e.g., avoid "1-hot encodings").
  - Perform more thorough cross-validations (instead of using just a single 80/10/10 split).

- **Compare to other classification techniques** (e.g., in Spark's MLlib)
  - Linear Regression / Logistic Regression
  - Naïve Bayes
  - Support Vector Machines (SVMs)