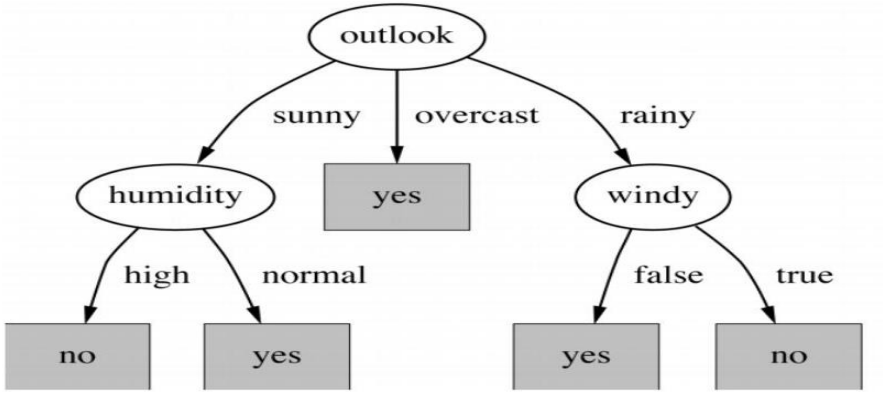
Decision Trees for Big Data Analytics

Dr. Rajiv Misra
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

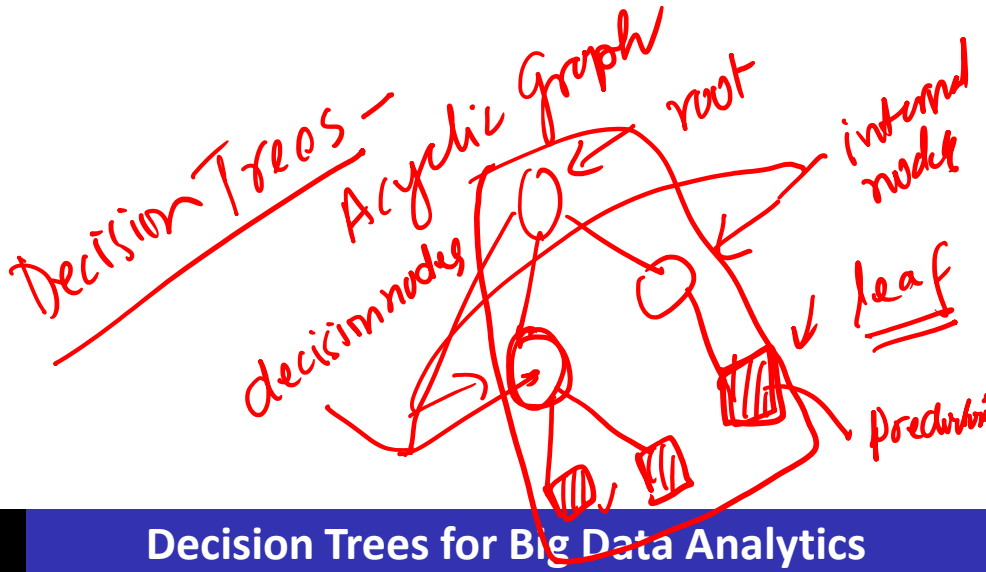
Preface

Content of this Lecture:

- In this lecture, we will discuss Decision Trees for Big Data Analytics and also discuss a case study of medical application using a Decision Tree in Spark ML



Decision Trees



Predict if Sachin will play cricket

Consider the following data set

our task is to predict if Sachin is going to play cricket on a given day.

We have observed Sachin over a number of days and recorded various things that might influence his decision to play cricket so we looked at what kind of weather it is.

Is it sunny or is it raining humidity is it high as normal is it windy

So this is our training set and these are the examples that we are going to build a classifier from.

Training examples: 9 yes / 5 no

Day	Outlook	Humidity	Wind	Play
D1	Sunny	High	Weak	No
D2	Sunny	High	Strong	No
D3	Overcast	High	Weak	Yes
D4	Rain	High	Weak	Yes
D5	Rain	Normal	Weak	Yes
D6	Rain	Normal	Strong	No
D7	Overcast	Normal	Strong	Yes
D8	Sunny	High	Weak	No
D9	Sunny	Normal	Weak	Yes
D10	Rain	Normal	Weak	Yes
D11	Sunny	Normal	Strong	Yes
D12	Overcast	High	Strong	Yes
D13	Overcast	Normal	Weak	Yes
D14	Rain	High	Strong	No

14 Days.

Data-set

Predict if Sachin will play cricket

So on day 15 if it's raining the humidity is high it's not very windy so is Sachin going to play or not and just by looking at the data it's kind of hard to it's it's kind of hard to decide right because it's you know some days when it's raining Sachin is playing other days Sachin is not playing sometimes he plays with strong winds sometimes he doesn't play with strong winds and with weak winds life so what do you do how do you predict it and the basic idea behind decision trees is try to at some level understand why Sachin plays. so this is the only classifier that that will cover that tries to predict Sachin playing in this way.

Training examples: **9 yes / 5 no**

Day	Outlook	Humidity	Wind	Play
D1	Sunny	High	Weak	No
D2	Sunny	High	Strong	No
D3	Overcast	High	Weak	Yes
D4	Rain	High	Weak	Yes
D5	Rain	Normal	Weak	Yes
D6	Rain	Normal	Strong	No
D7	Overcast	Normal	Strong	Yes
D8	Sunny	High	Weak	No
D9	Sunny	Normal	Weak	Yes
D10	Rain	Normal	Weak	Yes
D11	Sunny	Normal	Strong	Yes
D12	Overcast	High	Strong	Yes
D13	Overcast	Normal	Weak	Yes
D14	Rain	High	Strong	No

New data:

D15	Rain	High	Weak	?

Predict if Sachin will play cricket

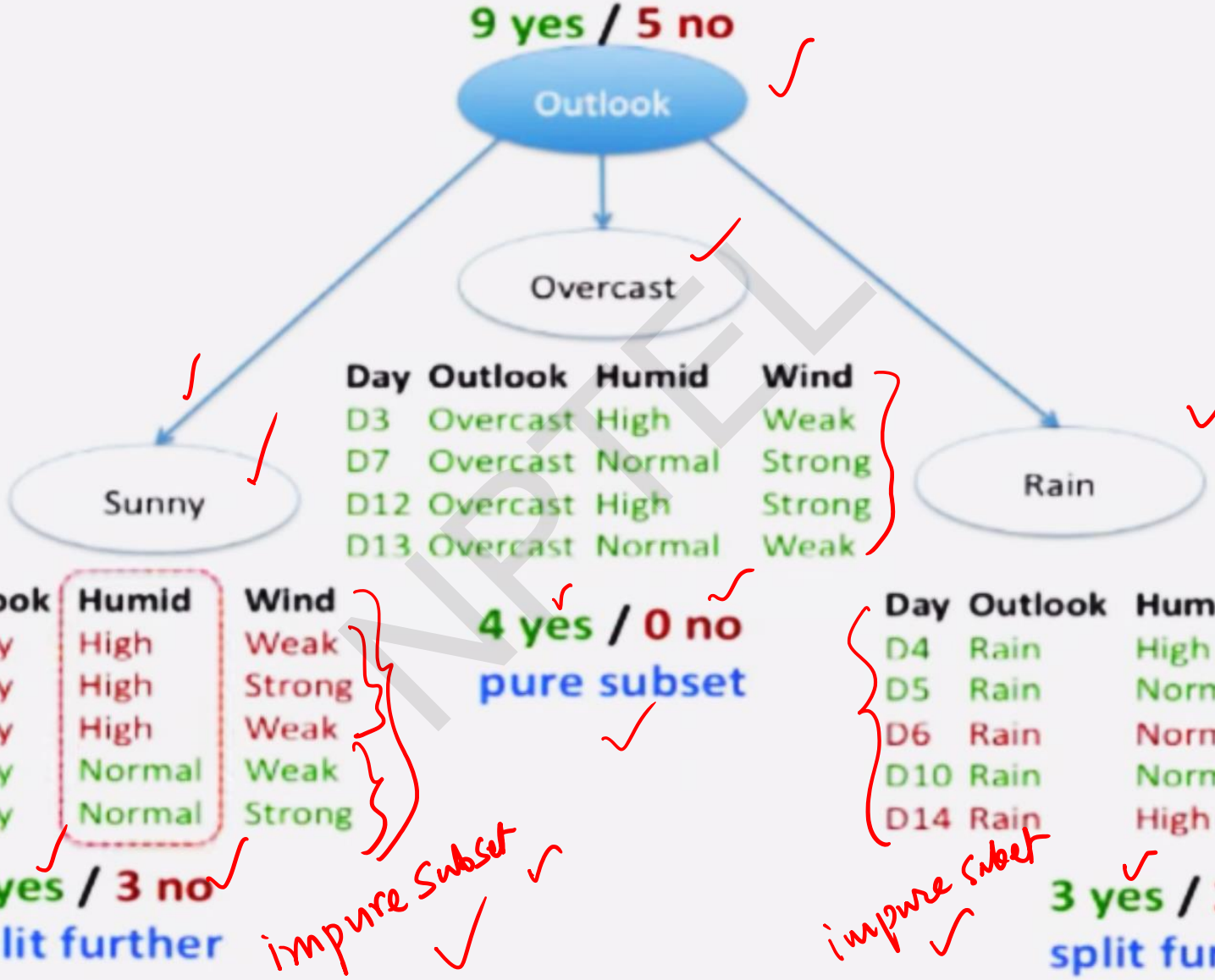
- Hard to guess
- Try to understand when Sachin plays
- Divide & conquer:
  - Split into subsets
  - Are they pure ? (all yes of all no)
  - If yes: stop
  - If not: repeat
- See which subset new data falls into

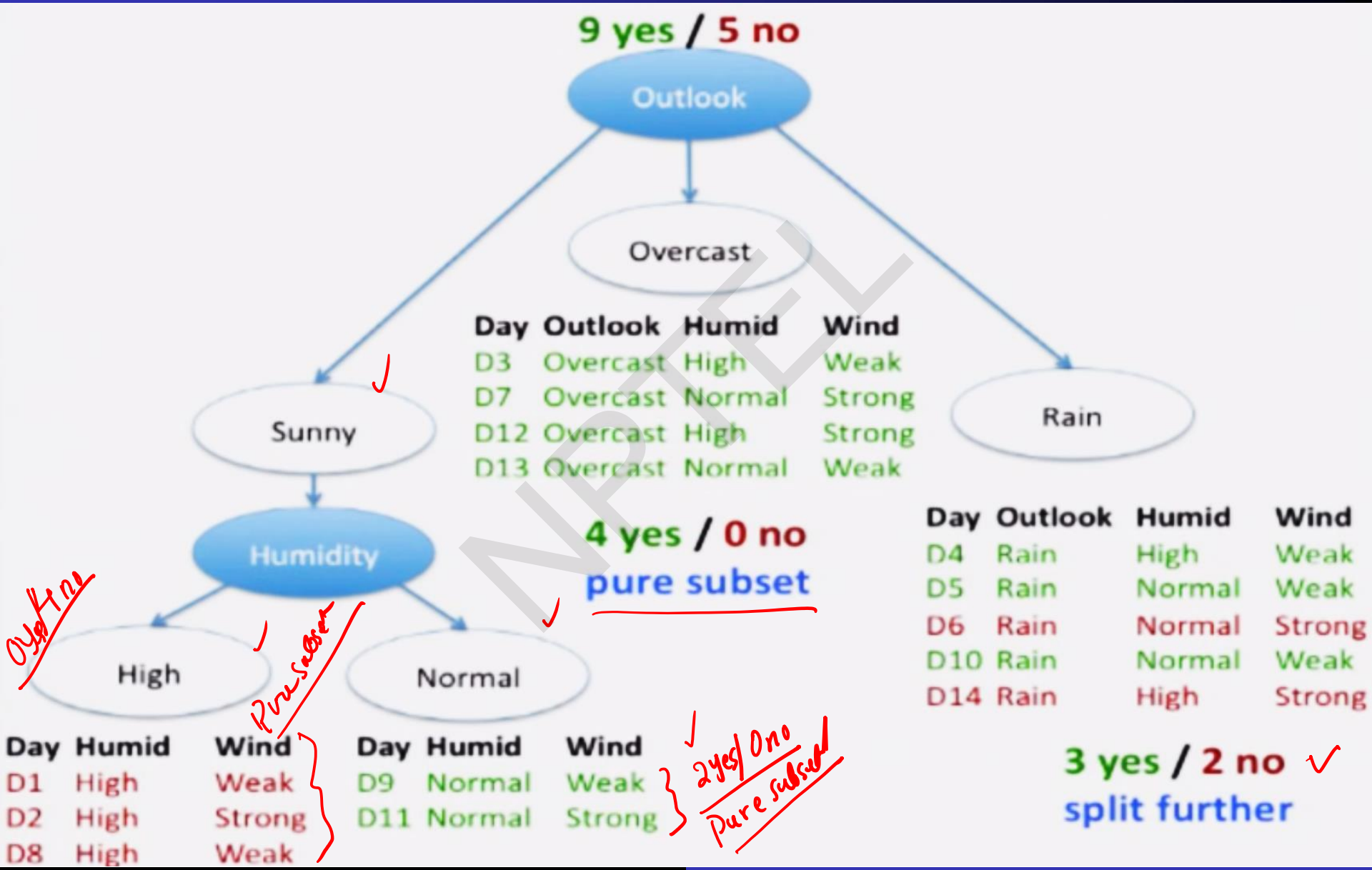
Training examples: **9 yes / 5 no**

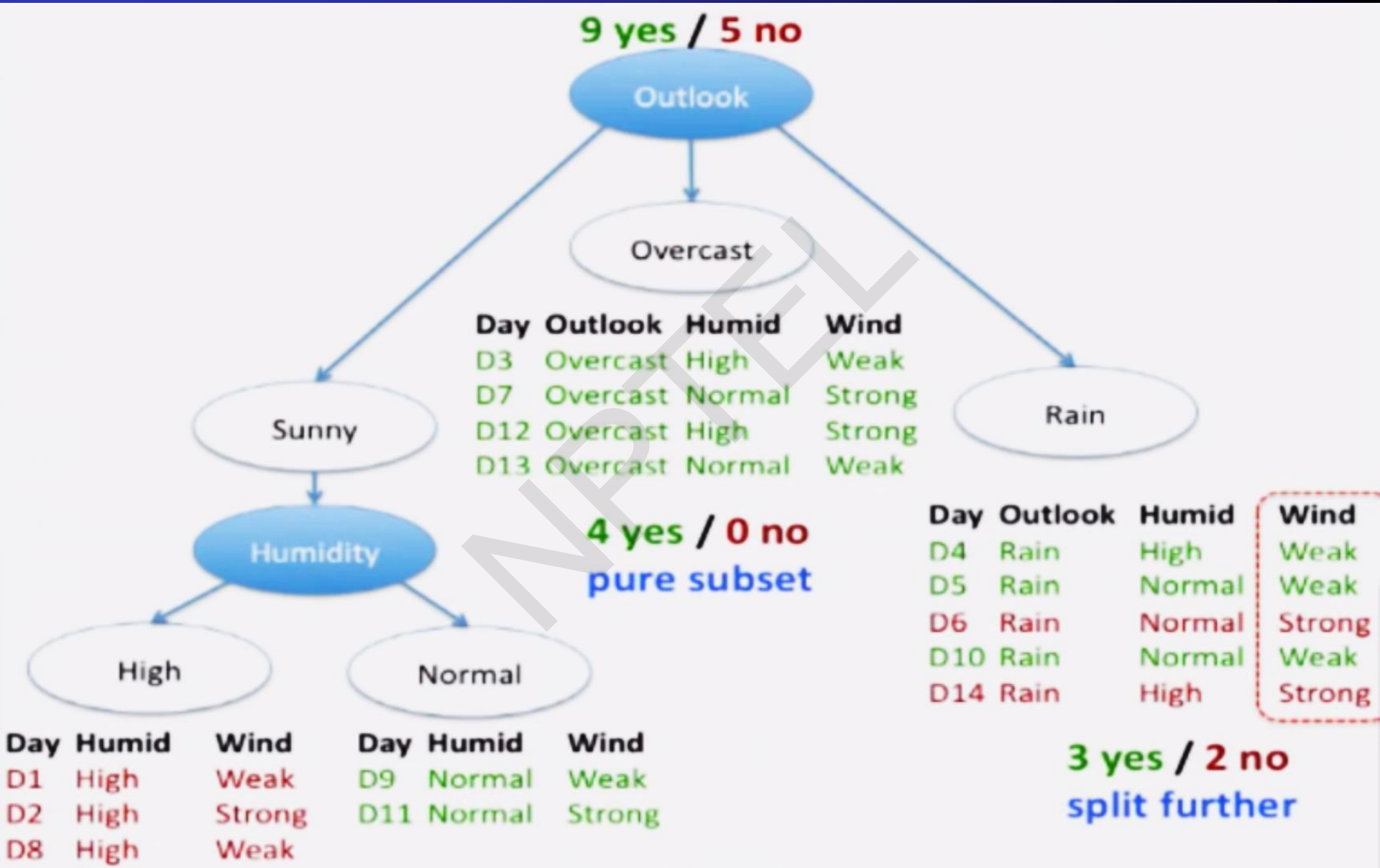
Day	Outlook	Humidity	Wind	Play
D1	Sunny	High	Weak	No
D2	Sunny	High	Strong	No
D3	Overcast	High	Weak	Yes
D4	Rain	High	Weak	Yes
D5	Rain	Normal	Weak	Yes
D6	Rain	Normal	Strong	No
D7	Overcast	Normal	Strong	Yes
D8	Sunny	High	Weak	No
D9	Sunny	Normal	Weak	Yes
D10	Rain	Normal	Weak	Yes
D11	Sunny	Normal	Strong	Yes
D12	Overcast	High	Strong	Yes
D13	Overcast	Normal	Weak	Yes
D14	Rain	High	Strong	No

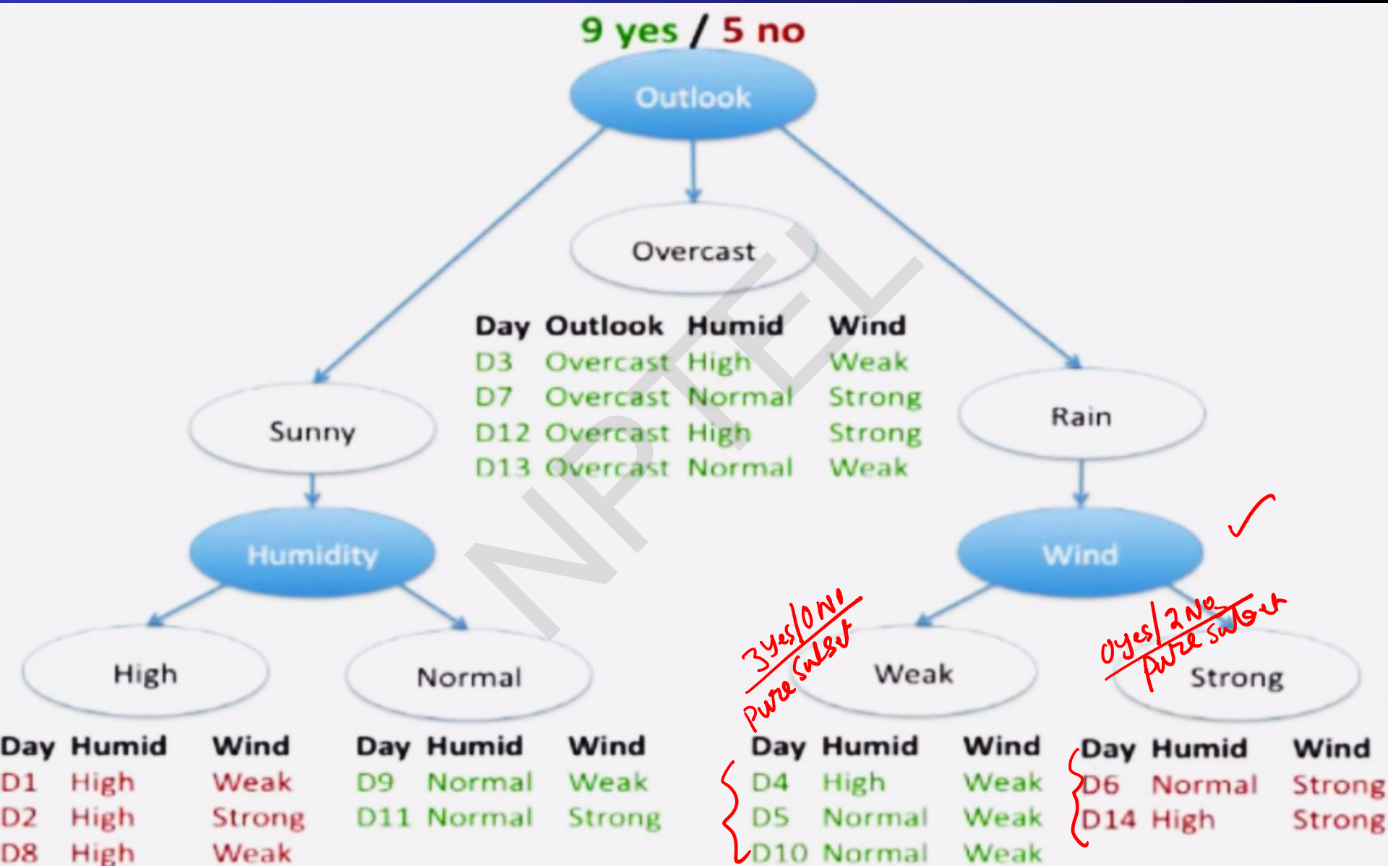
New data:

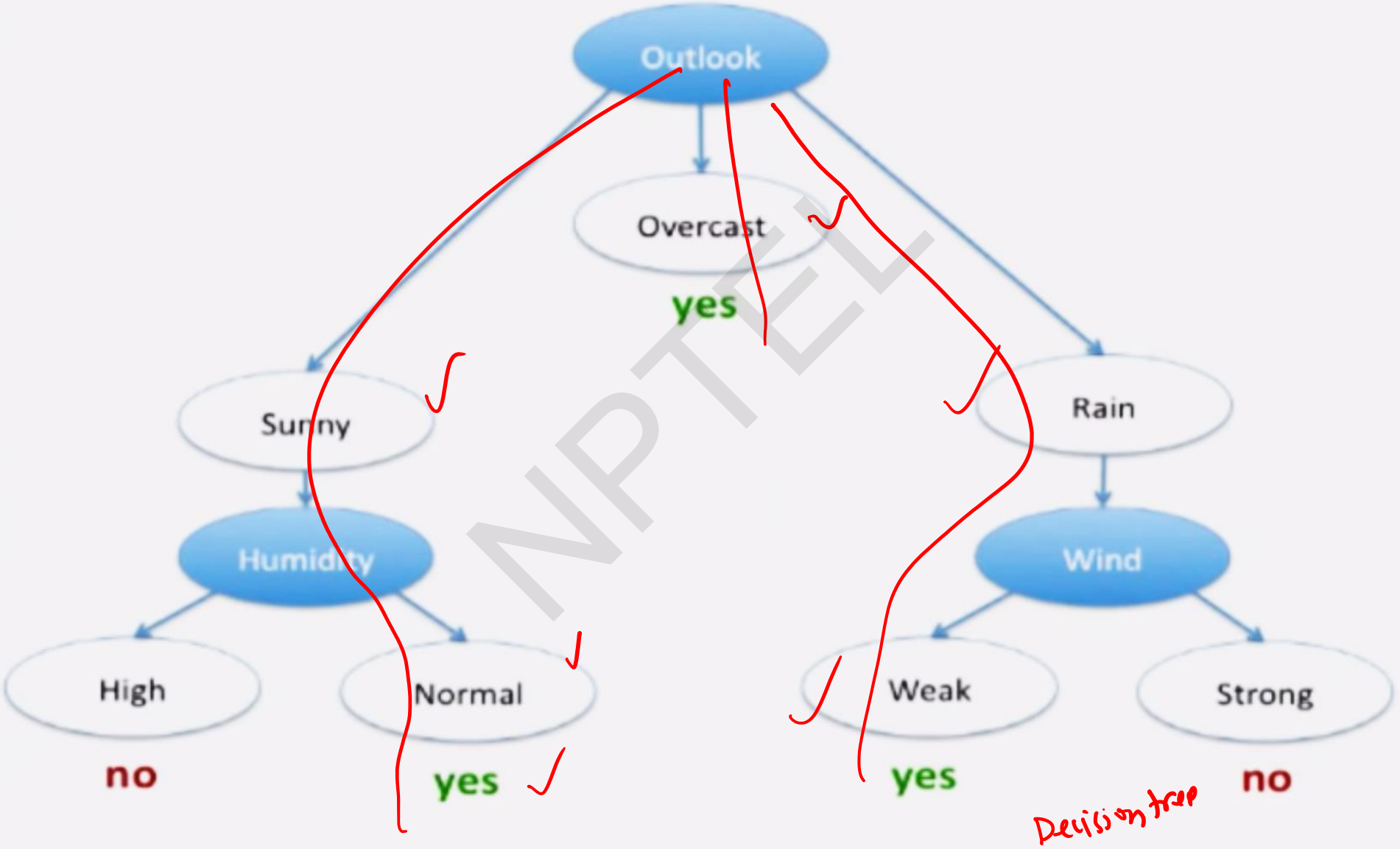
D15	Rain	High	Weak	?

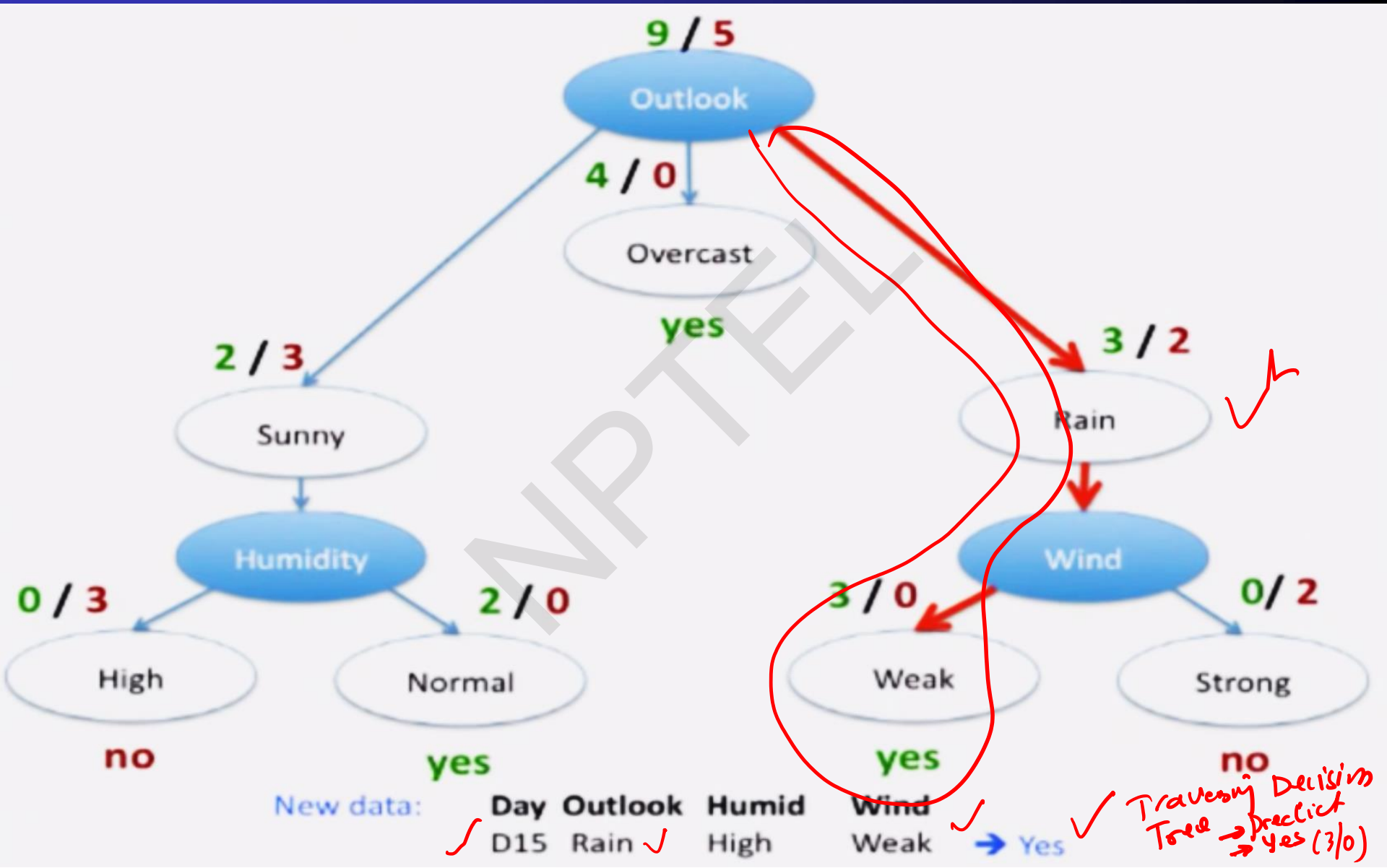

9 yes / 5 no
Outlook
Overcast
Sunny
Rain
Day Outlook Humid Wind
D3 Overcast High Weak
D7 Overcast Normal Strong
D12 Overcast High Strong
D13 Overcast Normal Weak
4 yes / 0 no
pure subset
Day Outlook Humid Wind
D1 Sunny High Weak
D2 Sunny High Strong
D8 Sunny High Weak
D9 Sunny Normal Weak
D11 Sunny Normal Strong
2 yes / 3 no
split further
impure subset
Day Outlook Humid Wind
D4 Rain High Weak
D5 Rain Normal Weak
D6 Rain Normal Strong
D10 Rain Normal Weak
D14 Rain High Strong
impure subset
3 yes / 2 no
split further


9 yes / 5 no
Outlook
Overcast
Sunny
Rain
Humidity
High
Normal
Day Outlook Humid Wind
D3 Overcast High Weak
D7 Overcast Normal Strong
D12 Overcast High Strong
D13 Overcast Normal Weak
4 yes / 0 no
pure subset
Day Outlook Humid Wind
D4 Rain High Weak
D5 Rain Normal Weak
D6 Rain Normal Strong
D10 Rain Normal Weak
D14 Rain High Strong
3 yes / 2 no
split further
Day Humid Wind
D1 High Weak
D2 High Strong
D8 High Weak
Day Humid Wind
D9 Normal Weak
D11 Normal Strong
2yes/0no
Pure subset
NPTEL


9 yes / 5 no
Outlook
Overcast
Sunny
Rain
Humidity
High
Normal
Day Outlook Humid Wind
D3 Overcast High Weak
D7 Overcast Normal Strong
D12 Overcast High Strong
D13 Overcast Normal Weak
4 yes / 0 no
pure subset
Day Outlook Humid Wind
D4 Rain High Weak
D5 Rain Normal Weak
D6 Rain Normal Strong
D10 Rain Normal Weak
D14 Rain High Strong
3 yes / 2 no
split further
Day Humid Wind
D1 High Weak
D2 High Strong
D8 High Weak
Day Humid Wind
D9 Normal Weak
D11 Normal Strong


9 yes / 5 no
Outlook
Overcast
Sunny
Rain
Humidity
Wind
High
Normal
Weak
Strong
Day Outlook Humid Wind
D3 Overcast High Weak
D7 Overcast Normal Strong
D12 Overcast High Strong
D13 Overcast Normal Weak
Day Humid Wind
D1 High Weak
D2 High Strong
D8 High Weak
Day Humid Wind
D9 Normal Weak
D11 Normal Strong
Day Humid Wind
D4 High Weak
D5 Normal Weak
D10 Normal Weak
Day Humid Wind
D6 Normal Strong
D14 High Strong
3 yes/0 no
Pure subset
0 yes/2 no
Pure subset
NPTEL

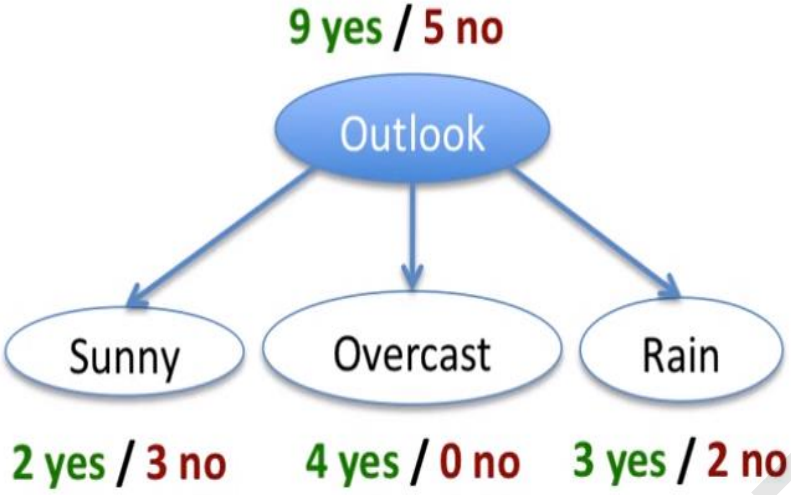

Outlook
Overcast
yes
Sunny
Rain
Humidity
Wind
High
Normal
Weak
Strong
no
yes
yes
no
Decision tree


9 / 5
Outlook
4 / 0
Overcast
yes
2 / 3
Sunny
3 / 2
Rain
Humidity
Wind
0 / 3
2 / 0
3 / 0
0 / 2
High
Normal
Weak
Strong
no
yes
yes
no
New data:
Day Outlook Humid Wind
D15 Rain High Weak → Yes
Traversing Decision Tree → Predict yes (3/0)

ID3 Algorithm for Decision Tree

- In decision tree learning, ID3 (Iterative Dichotomiser 3) is an algorithm invented by Ross Quinlan used to generate a decision tree from a dataset. ID3 is the precursor to the C4.5 algorithm, and is typically used in the machine learning and natural language processing domains.
- Split (node, {examples}):
  1. A← the best attribute for splitting the {examples}
  2. Decision attribute for this node ← A
  3. For each value of A, create new child node
  4. Split training {examples} to child nodes
  5. For each child node / subset:
     - if subset is pure: STOP
     - else: Split (child_node, {subset})
- Ross Quinlan (ID3: 1986), (C4.5:1993)
- Breimanetal (CaRT:1984) from statistics

Which attribute to split on ?



- Want to measure "purity" of the split
  - More certain about Yes/No after the split
    - pure set (**4 yes**/ **0 no**)➔ completely certain (100%)
    - impure (**3 yes**/**3 no**)➔ completely uncertain (50%)
  - Can't use P("yes" | set):
    - must be symmetric: **4 yes**/**0 no** as pure as **0 yes** / **4 no**

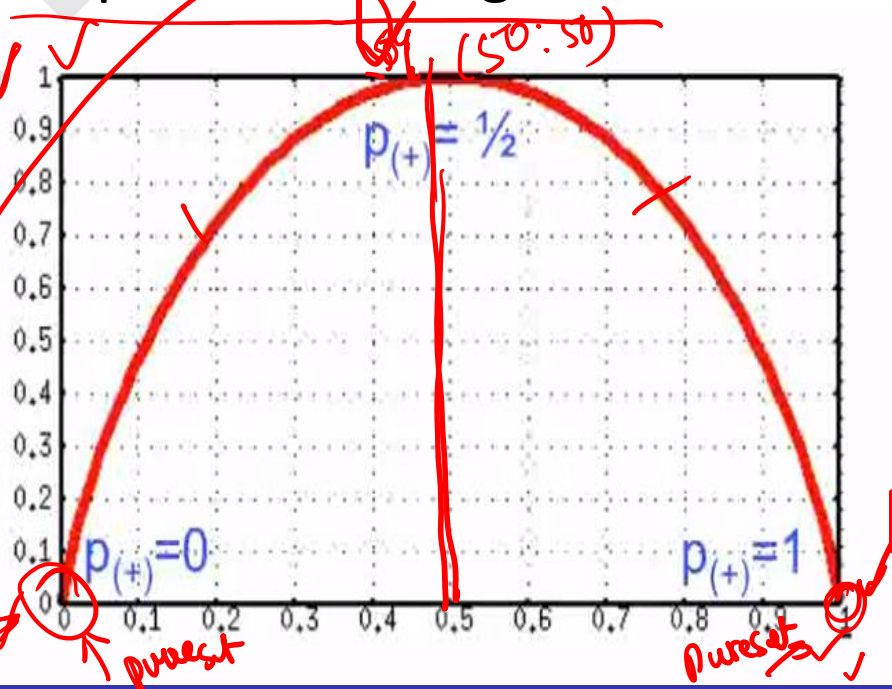
Entropy

- Entropy: $H(S) = - p_{(+)} \log_2 p_{(+)} - p_{(-)} \log_2 p_{(-)}$ bits
  - S … subset of training examples
  - $p_{(+)}$ / $p_{(-)}$…% of positive/negative examples in S
- Interpretation: assume item X belongs to S
  - How many bits need to tell if X positive or negative
- Impure (3 yes / 3 no):

$$H(S) = -\frac{3}{6}\log_2\frac{3}{6} - \frac{3}{6}\log_2\frac{3}{6} = 1 \text{ bits}$$

- Pure set (4 yes / 0 no):

$$H(S) = -\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{0}{4} = 0 \text{ bits}$$



Information Gain

- Want many items in pure sets
- Expected drop in entropy after split:

$$Gain(S,A) = H(S) - \sum_{V \in Values(A)} \frac{|S_V|}{|S|} H(S_V)$$

V … possible values of A
S … set of examples {X}
S_v … subset where X_A = V

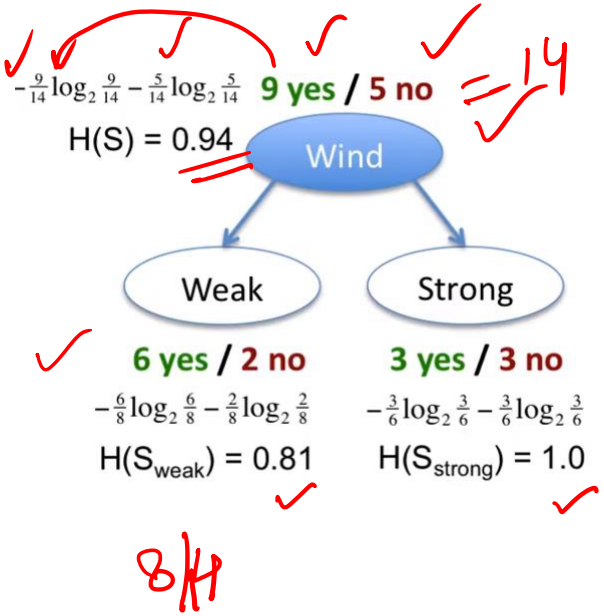
- Mutual Information
  -between attribute A
  and class labels of S

Gain (S, Wind)

= H(S) - 8/14 * H(Sweak) - 6/14 * H(Sstrong)

= 0.94- 8/14 * 0.81 - 6/14 * 1.0

= 0.049



Decision Trees for Regression

How to grow a decision tree

- The tree is built **greedily** from top to bottom
- Each split is selected to **maximize information gain (IG)**

$$IG = \underbrace{\text{Impurity}(Z)}_{\text{Error before split}} - \underbrace{\left(\frac{|Z_L|}{|Z|}\text{Impurity}(Z_L) + \frac{|Z_R|}{|Z|}\text{Impurity}(Z_R)\right)}_{\text{Error after split}}$$

Decision Tree for Regression

- Given a training set: Z= $\{(x_1,y_1),\ldots,(x_n,y_n)\}$

  y_i-real values

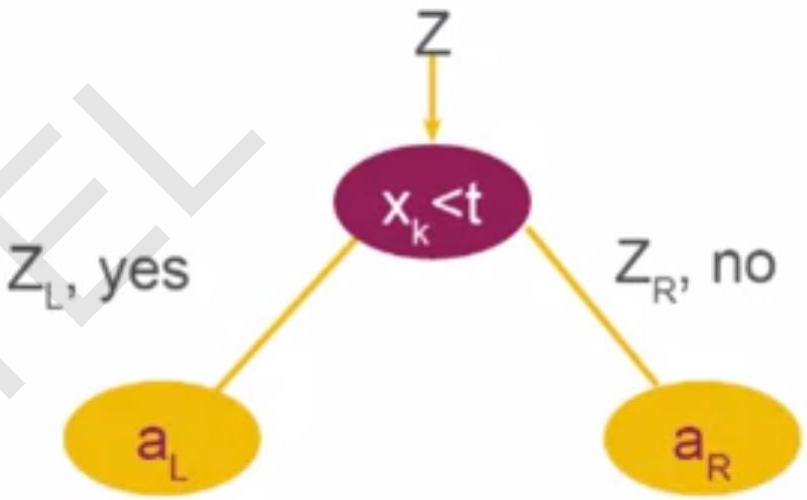
- Goal is to find f(x) (a tree) such that

$$\min \sum_{i=1}^{n} (f(\boldsymbol{x}_i) - y_i)^2$$

- How to grow a decision tree for regression ?

How to find the best split

- A tree is built from top to bottom. And at each step, you should find the best split in the internal node. You get a test dataset Z. And here is a splitting criteria xk less than t or xk is greater or equal than a threshold t. The problem is how to find k, the number of feature and the threshold t. And also you need to find values in leaves aL and aR.



What happens without a split?

- **What happens without a split?**
- Without a split, the best you can do is to predict one number, ‘a’. It makes sense to select such a number ‘a’ to minimize the squared error.
- It is very easy to prove that such variable a equals an average of all yi.
- Thus you need to calculate the average value of all the targets.
- Here ā (a-hat)denotes the average value.
- Impurity z equals the mean squared error if we use ā (a-hat) as a prediction.

Without split: predict one number **a**

$$\hat{a} = \min_a \sum_{i \in Z} (a - y_i)^2$$

$$\hat{a} = \frac{1}{|Z|} \sum_{i \in Z} y_i$$

|Z| - number of elements in Z

$$\text{Impurity}(Z) = \frac{1}{|Z|} \sum_{i \in Z} (\hat{a} - y_i)^2$$

Find the best split ($x_k < t$)

- What happens if you make some split by a condition xk < t? For some part of training objects ZL, you have x_k < t. For other part ZR holds xk ≥ t.
- The error consists of two parts for ZL and ZR respectively. So here we need to find simultaneously k, t, aL and aR.
- We can calculate the optimal aL and aR exactly the same way as we did for the case without a split. We can easily prove that the optimal aL equals an average of all targets which are the targets of objects which get to the leaf. And we will denote these values by a-hat L and a-hat R respectively.

$$\min_{k,\,t,\,a_L,a_R} \sum_{i \in Z_L} (a_L - y_i)^2 + \sum_{i \in Z_R} (a_R - y_i)^2$$

Values in leaves

$$\widehat{a_L} = \frac{1}{|Z_L|} \sum_{i \in Z_L} y_i, \qquad \widehat{a_R} = \frac{1}{|Z_R|} \sum_{i \in Z_R} y_i$$

$|Z_L|$ - number of elements in Z_L,
$|Z_R|$ - number of elements in Z_R

$$\min_{k,\,t} \sum_{i \in Z_L} (\widehat{a_L} - y_i)^2 + \sum_{i \in Z_R} (\widehat{a_R} - y_i)^2$$

Find the best split

- After this step we only need to find optimal k and t. We have formulas for impurity, for objects which get to the left branch of our splitting condition and to the right. Then you can find the best splitting criteria which maximizes the information gain. This procedure is done iteratively from top to bottom.

$$\text{Impurity}(Z_L) = \frac{1}{|Z_L|} \sum_{i \in Z_L} (\widehat{a_L} - y_i)^2$$

$$\text{Impurity}(Z_R) = \frac{1}{|Z_R|} \sum_{i \in Z_R} (\widehat{a_R} - y_i)^2$$

Maximize the information gain (IG):

$$IG = \text{Impurity}(Z) - \left(\frac{|Z_L|}{|Z|} \text{Impurity}(Z_L) + \frac{|Z_R|}{|Z|} \text{Impurity}(Z_R) \right)$$

with respect to k, t (splitting criteria $x_k < t$)

Stopping rule

- The node depth is equal to the **maxDepth** training parameter.
- No split candidate leads to an information gain greater than **mininfoGain**.
- No split candidate produces child nodes which have at least **minInstancesPerNode** training instances ($|Z_L|, |Z_R| <$ **minInstancesPerNode**) each.

Summary: Decision Trees

- **Automatically handling interactions of features:** The benefits of decision tree is that this algorithm can automatically handle interactions or features because it can combine several different features in a single decision tree. It can build complex functions involving multiple splitting criteria.

- **Computational scalability:** The second property is a computational scalability. There exists effect of algorithms for building decision trees for the very large data sets with many features.

- **Predictive power:** Single decision tree actually is not a very good predictor. The predictive power of a single tree is typically not so good.

- **Interpretability:** You can visualize the decision tree and analyze this splitting criteria in nodes, the values in leaves, and so one. Sometimes it might be helpful.

Building a tree using MapReduce

Problem: Building a tree

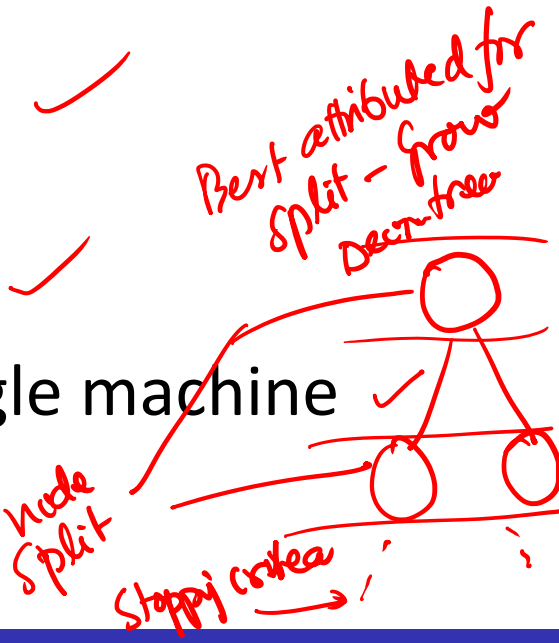
- Given a large dataset with hundreds of attributes
- **Build a decision tree!**

Algorithm 1 **FindBestSplit**

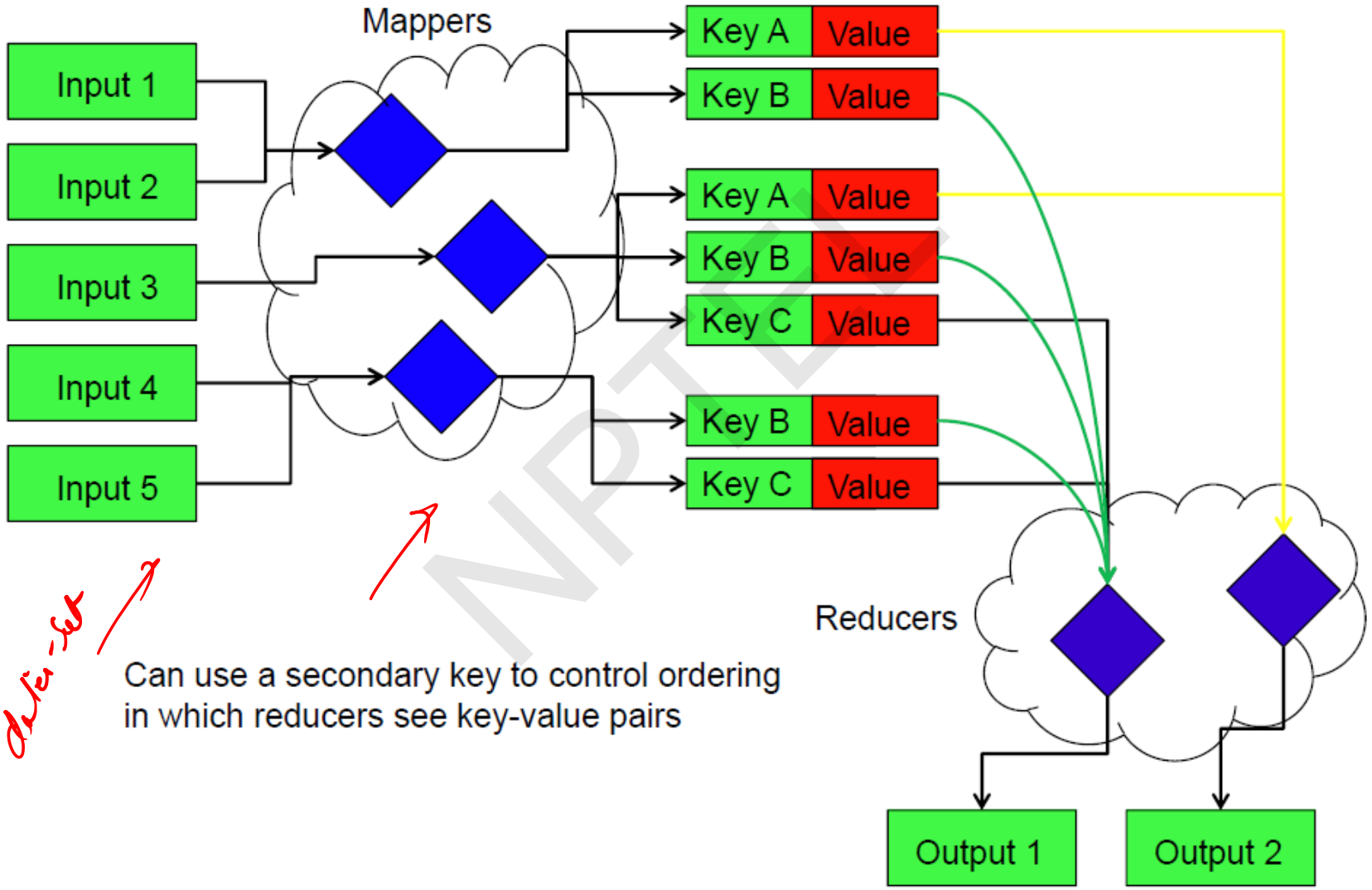
Require: Node n, Data $D \subseteq D^*$

1: $(n \rightarrow \text{split}, D_L, D_R) =$ **FindBestSplit**(D)
2: **if** StoppingCriteria(D_L) **then**
3: $n \rightarrow$left_prediction=FindPrediction(D_L)
4: **else**
5: **FindBestSplit**$(n \rightarrow \text{left}, D_L)$
6: **if** StoppingCriteria(D_R) **then**
7: $n \rightarrow$right_prediction=FindPrediction(D_R)
8: **else**
9: **FindBestSplit**$(n \rightarrow \text{right}, D_R)$

- **General considerations:**
  - Tree is small (can keep it memory):
    - Shallow (~10 levels)
  - Dataset too large to keep in memory
  - Dataset too big to scan over on a single machine
  - **MapReduce to the rescue!**



MapReduce

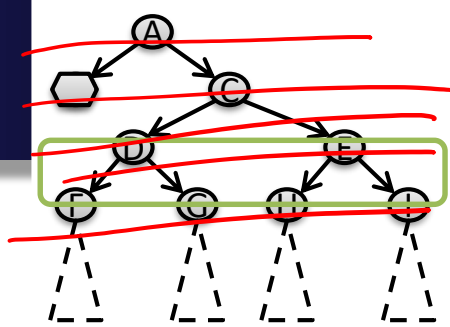


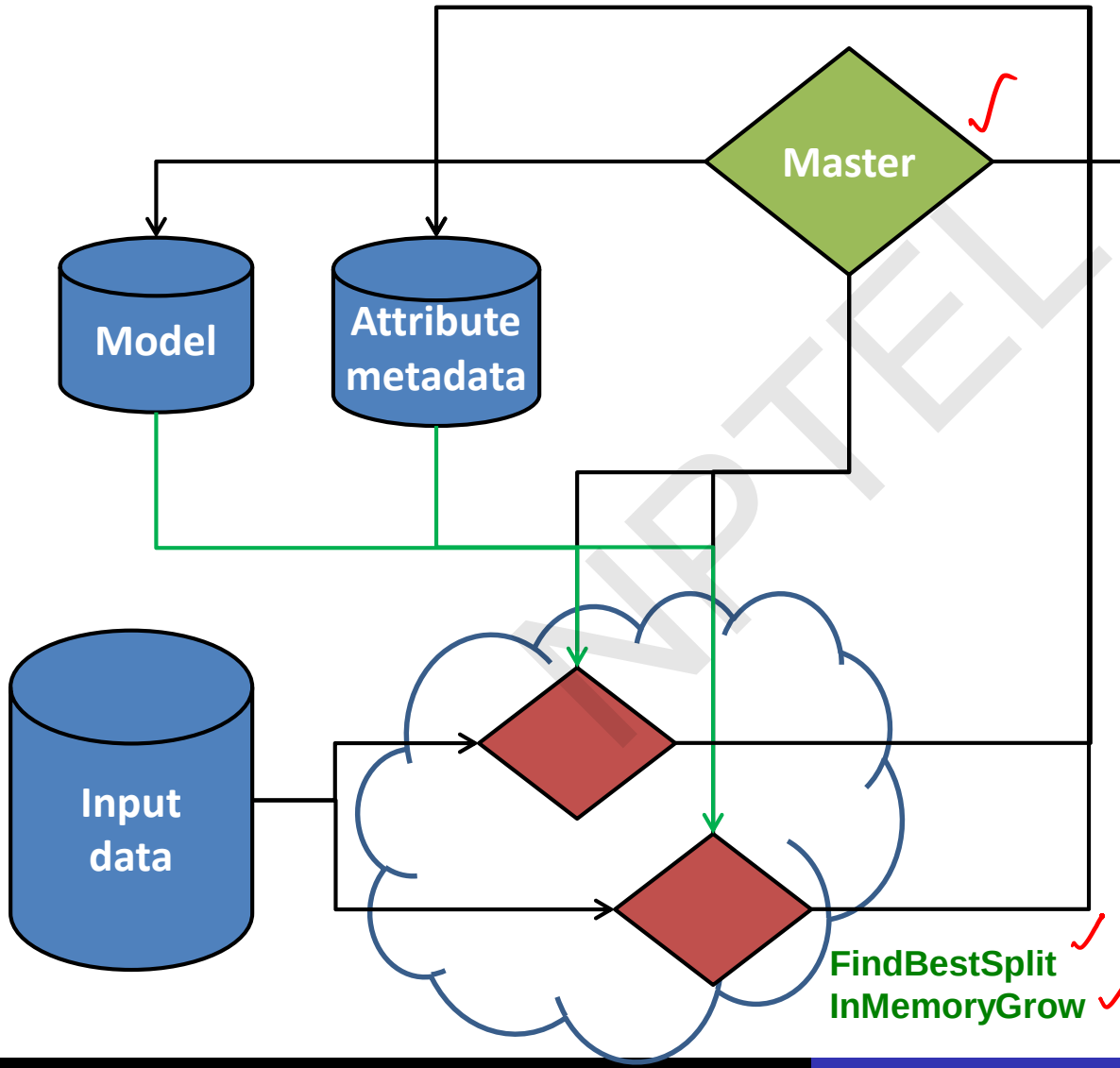
PLANET

Parallel **L**earner for **A**ssembling **N**umerous **E**nsemble **T**rees [Panda et al., VLDB '09]

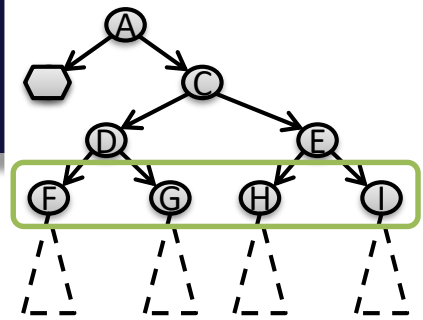
- A sequence of MapReduce jobs that build a decision tree
- **Setting:**
  - Hundreds of numerical (discrete & continuous) attributes
  - Target (class) is numerical: **Regression**
  - Splits are binary: $X_j < v$
  - Decision tree is small enough for each Mapper to keep it in memory
  - Data too large to keep in memory

PLANET Architecture



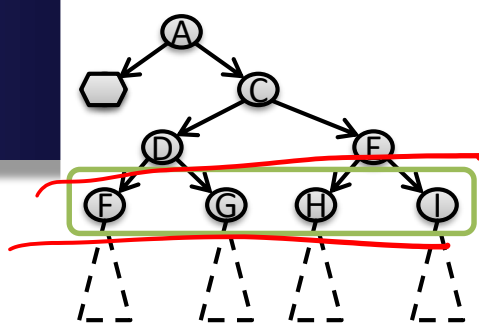


PLANET Overview



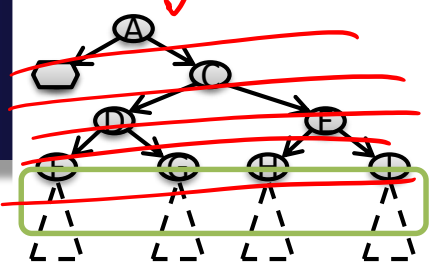
- We build the tree level by level
  - One MapReduce step builds one level of the tree
- **Mapper**
  - Considers a number of possible splits **(Xi, v)** on its subset of the data
  - For each split it stores partial statistics
  - Partial split-statistics is sent to **Reducers**
- **Reducer**
  - Collects all partial statistics and determines best split.
- **Master** grows the tree for one level

PLANET Overview



- **Mapper** loads the **model** and info about which **attribute splits** to consider
- Each mapper sees a subset of the data **D***
- Mapper "drops" each datapoint to find the appropriate leaf node ***L***
- For each leaf node ***L*** it keeps statistics about
  - 1) the data reaching ***L***
  - 2) the data in left/right subtree under split ***S***
- **Reducer** aggregates the statistics (1) and (2) and determines the best split for each node

PLANET: Components



- **Master**
  - Monitors everything (runs multiple MapReduce jobs)
- **Three types of MapReduce jobs:**

(1) MapReduce Initialization (run once first)

- For each attribute identify values to be considered for splits

(2) MapReduce FindBestSplit (run multiple times)

- MapReduce job to find best split when there is too much data to fit in memory

(3) MapReduce InMemoryBuild (run once last)

- Similar to FindBestSplit (but for small data)
- Grows an entire sub-tree once the data fits in memory

Reference

- B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. *VLDB* 2009.

- J. Ye, J.-H. Chow, J. Chen, Z. Zheng. Stochastic Gradient Boosted Distributed Decision Trees. CIKM 2009.

Example: Medical Application using a Decision Tree in Spark ML

Create SparkContext and SparkSession

- This example will be about using Spark ML for doing classification and regression with decision trees, and in samples of decision trees.
- First of all, you are going to create SparkContext and SparkSession, and here it is.

Consider Medical application using Decision trees for Big data Analytics

```
In [ ]: from pyspark import SparkContext
        from pyspark.sql import SparkSession

In [ ]: sc = SparkContext(appName = "module3_week4")

In [ ]: ! echo $PYSPARK_SUBMIT_ARGS

In [ ]: spark = SparkSession.Builder().getOrCreate() # required for dataframes
```

1. Create SparkContext

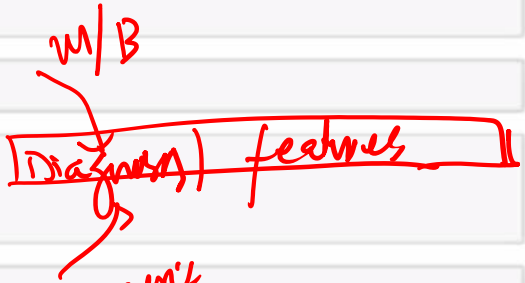
2. build spark sess

Download a dataset

- Now you are downloading a dataset which is about breast cancer diagnosis, here it is.

Download a dataset (breast cancer diagnosis)

```
In [ ]: !wget https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wis

In [ ]: !head -n 3 wdbc.data

In [ ]: !wc -l wdbc.data

In [ ]: #1) ID number
        #2) Diagnosis (M = malignant, B = benign)
        #3) Features

In [ ]: from pyspark.ml.linalg import Vectors
        from pyspark.ml.feature import StringIndexer

In [ ]: # Load a text file and convert each line to a Row.

        data = []

        with open("wdbc.data") as infile:
```



Exploring the Dataset

- Let's explore this dataset a little bit. The first column is ID of observation. The second column is the diagnosis, and other columns are features which are comma separated. So these features are results of some analysis and measurements. There are total 569 examples in this dataset. And if the second column is M, it means that it is cancer. If B, means that there is no cancer in a particular woman.

```
In [6]: !head -n 3 wdbc.data

842302,M,17.99,10.38,122.8,1001,0.1184,0.2776,0.3001,0.1471,0.2419,0.07871,1.09
5,0.9053,8.589,153.4,0.006399,0.04904,0.05373,0.01587,0.03003,0.006193,25.38,1
7.33,184.6,2019,0.1622,0.6656,0.7119,0.2654,0.4601,0.1189
842517,M,20.57,17.77,132.9,1326,0.08474,0.07864,0.0869,0.07017,0.1812,0.05667,
0.5435,0.7339,3.398,74.08,0.005225,0.01308,0.0186,0.0134,0.01389,0.003532,24.9
9,23.41,158.8,1956,0.1238,0.1866,0.2416,0.186,0.275,0.08902
84300903,M,19.69,21.25,130,1203,0.1096,0.1599,0.1974,0.1279,0.2069,0.05999,0.74
56,0.7869,4.585,94.03,0.00615,0.04006,0.03832,0.02058,0.0225,0.004571,23.57,25.
53,152.5,1709,0.1444,0.4245,0.4504,0.243,0.3613,0.08758

In [ ]: !wc -l wdbc.data

In [ ]: #1) ID number
        #2) Diagnosis (M = malignant, B = benign)
        #3) Features

In [ ]: from pyspark.ml.linalg import Vectors
        from pyspark.ml.feature import StringIndexer

In [ ]: # Load a text file and convert each line to a Row.
```

Exploring the Dataset

- First of all you need to transform the label, which is either M or B from the second column. You should transform it from a string to a number. We use a StringIndexer object for this purpose, and first of all you need to load all these datasets. Then you create a Spark DataFrame, which is stored in the distributed manner on cluster.

In [ ]:
```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import StringIndexer
```

In [ ]:
```
# Load a text file and convert each line to a Row.

data = []

with open("wdbc.data") as infile:
    for line in infile:
        tokens = line.rstrip("\n").split(",")
        y = tokens[1]
        features = Vectors.dense([float(x) for x in tokens[2:]])

        data.append((y, features))
```

In [ ]:
```
inputDF = spark.createDataFrame(data, ["label", "features"])
```

In [ ]:
```
inputDF.show()
```

In [ ]:
```
stringIndexer = StringIndexer(inputCol = "label", outputCol = "labelIndexed")
si_model = stringIndexer.fit(inputDF)
inputDF2 = si_model.transform(inputDF)
```

Exploring the Dataset

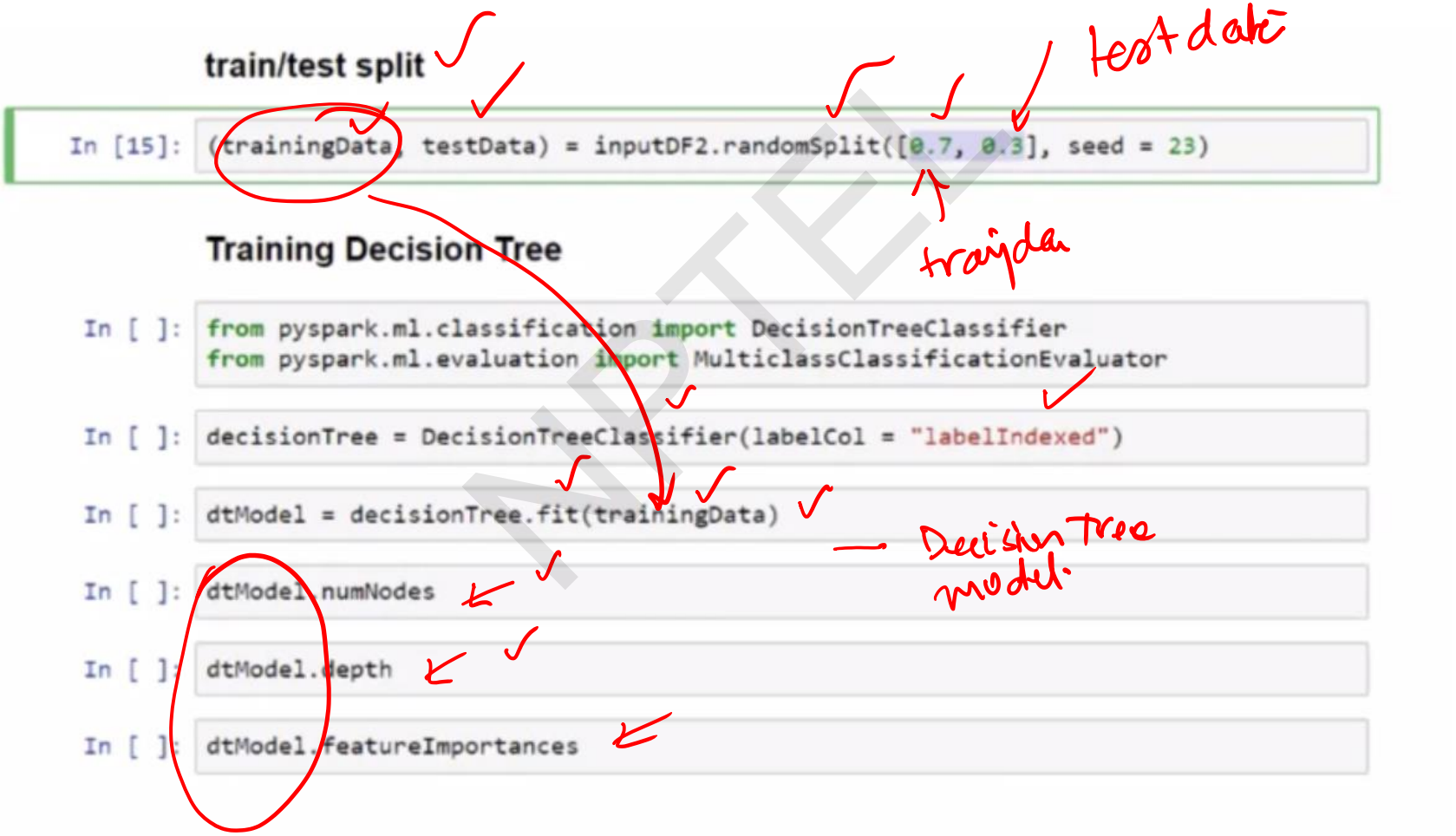
- inputDF DataFrame has two columns, label and features. Okay, you use an object vector for creating a vector column in this dataset, and then you can do string indexing. So Spark now enumerates all the possible labels in a string form, and transforms them to the label indexes. And now label M is equivalent 1, and B label is equivalent 0.

```
In [11]: inputDF = spark.createDataFrame(data, ["label", "features"])

In [12]: inputDF.show()
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|    M|[17.99,10.38,122....|
|    M|[20.57,17.77,132....|
|    M|[19.69,21.25,130....|
|    M|[11.42,20.38,77.5...|
|    M|[20.29,14.34,135....|
|    M|[12.45,15.7,82.57...|
|    M|[18.25,19.98,119....|
|    M|[13.71,20.83,90.2...|
|    M|[13.0,21.82,87.5,...|
|    M|[12.46,24.04,83.9...|
|    M|[16.02,23.24,102....|
|    M|[15.78,17.89,103....|
|    M|[19.17,24.8,132.4...|
|    M|[15.85,23.95,103....|
|    M|[13.73,22.61,93.6...|
|    M|[14.54,27.54,96.7...|
|    M|[14.68,20.13,94.7...|
```

Train/Test Split

- We can start doing the machine learning right now. First of all, you make training test splitting in the proportion 70% to 30%. And the first model you are going to evaluate is one single decision tree.



Train/Test Split

- We are making import DecisionTreeClassifier object. We create a class which is responsible for training, and we call the method fit to the training data, and obtain a decision tree model.

```
In [17]: decisionTree = DecisionTreeClassifier(labelCol = "labelIndexed")

In [18]: dtModel = decisionTree.fit(trainingData)

In [19]: dtModel.numNodes
Out[19]: 29

In [20]: dtModel.depth
Out[20]: 5

In [21]: dtModel.featureImportances
Out[21]: SparseVector(30, {1: 0.0589, 6: 0.0037, 10: 0.0112, 13: 0.0117, 20: 0.0324, 21:
         0.0302, 22: 0.7215, 24: 0.01, 26: 0.0191, 27: 0.1013})

In [22]: dtModel.numFeatures
Out[22]: 30

In [ ]: print dtModel.toDebugString
```

Train/Test Split

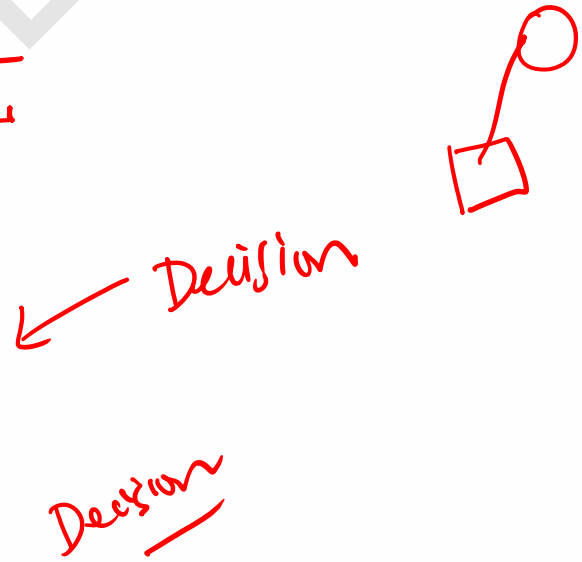
- Number of nodes and depths of decision tree, feature importances, total number of features used in this decision tree, and so on.
- We can even visualize this decision tree and explore it, and here is a structure of this decision tree. Here are splitting conditions If and Else, which predict values and leaves of our decision trees.

```
Out[21]: SparseVector(30, {1: 0.0589, 6: 0.0037, 10: 0.0112, 13: 0.0117, 20: 0.0324, 21:
         0.0302, 22: 0.7215, 24: 0.01, 26: 0.0191, 27: 0.1013})

In [22]: dtModel.numFeatures

Out[22]: 30

In [ ]: print dtModel.toDebugString

In [ ]: predictions = dtModel.transform(testData)

In [ ]: predictions.select('label', 'labelIndexed', 'probability', 'prediction').show()

In [ ]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
        accuracy = evaluator.evaluate(predictions)

        print("Test Error = %g" % (1.0 - accuracy))
```

Train/Test Split

- Now we are applying a decision tree model to the test data, and obtain predictions. And you can explore these predictions. The predictions are in the last column.
- And in this particular case, our model always predicts zero class.

```
In [23]: print dtModel.toDebugString

DecisionTreeClassificationModel (uid=DecisionTreeClassifier_4653be4ce1bd9e589b6
4) of depth 5 with 29 nodes
  If (feature 22 <= 114.2)
   If (feature 27 <= 0.1613)
    If (feature 20 <= 16.57)
     If (feature 27 <= 0.1258)
      If (feature 10 <= 0.9289)
       Predict: 0.0
      Else (feature 10 > 0.9289)
       Predict: 1.0
     Else (feature 27 > 0.1258)
      If (feature 21 <= 32.85)
       Predict: 0.0
      Else (feature 21 > 32.85)
       Predict: 1.0
    Else (feature 20 > 16.57)
     If (feature 1 <= 16.54)
      Predict: 0.0
     Else (feature 1 > 16.54)
      If (feature 24 <= 0.1084)
       Predict: 0.0
      Else (feature 24 > 0.1084)
       Predict: 1.0
```



Predictions

```
In [ ]: predictions = dtModel.transform(testData)

In [ ]: predictions.select('label', 'labelIndexed', 'probability', 'prediction').show()

In [ ]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
        accuracy = evaluator.evaluate(predictions)

        print("Test Error = %g" % (1.0 - accuracy))
```

Predictions

```
In [24]: predictions = dtModel.transform(testData)

In [25]: predictions.select('label', 'labelIndexed', 'probability', 'prediction').show()
```

```
+-----+------------+--------------------+----------+
|label|labelIndexed|         probability|prediction|
+-----+------------+--------------------+----------+
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
```

Accuracy

```
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
|    B|          0.0|[0.99086757990867...|          0.0|
+-----+-------------+--------------------+-------------+
only showing top 20 rows
```

In [26]:
```
evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
accuracy = evaluator.evaluate(predictions)

print("Test Error = %g" % (1.0 - accuracy))
```

```
Test Error = 0.0451977
```

96%

Conclusion

- In this lecture, we have discussed Decision Trees for Big Data Analytics.

- We have also discussed a case study of Breast Cancer Diagnosis using a Decision Tree in Spark ML.

Big Data Predictive Analytics

Dr. Rajiv Misra
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the fundamental techniques of predictive analytics.

- We will mainly cover Random Forest, Gradient Boosted Decision Trees and a Case Study with Spark ML Programming, Decision Trees and Ensembles.

Decision Trees

Summary: Decision Trees

- **Automatically handle interactions of features:** It can combine several different features in a single decision tree. It can build complex functions involving multiple splitting criteria.

- **Computational scalability:** There exists effect of algorithms for building decision trees for the very large data sets with many features. But unfortunately, single decision tree actually is not a very good predictor.

- **Predictive Power:** The predictive power of a single tree is typically not so good.

- **Interpretability:** We can visualize the decision tree and analyze this splitting criteria in nodes, the values in leaves, and so one.

Bootstrap and Bagging

Bootstrap

- Bootstrapping is an algorithm which produces replicas of a data set by doing random sampling with replacement. This idea is essential for the random forest algorithm.
- Consider a dataset Z={(x1, y1),…,(xn,yn)}
- Bootstrapped dataset Z*- It is a modification of the original dataset Z, produced by random sampling with replacement.

Sampling with Replacement

- Each iteration pick an object at random, and there is no correlation with the previous step. Consider a data set, Z, for example, with five objects.

Random sampling with replacement Example

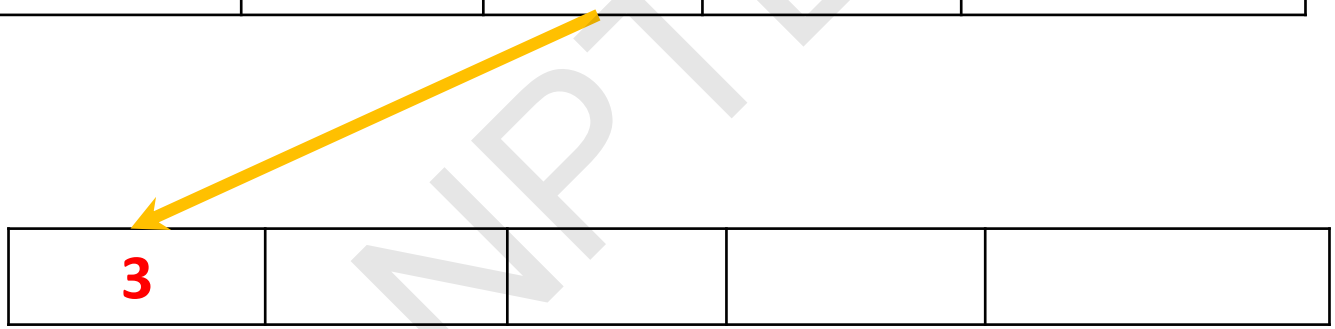
Z

1	2	3	4	5

← Data set

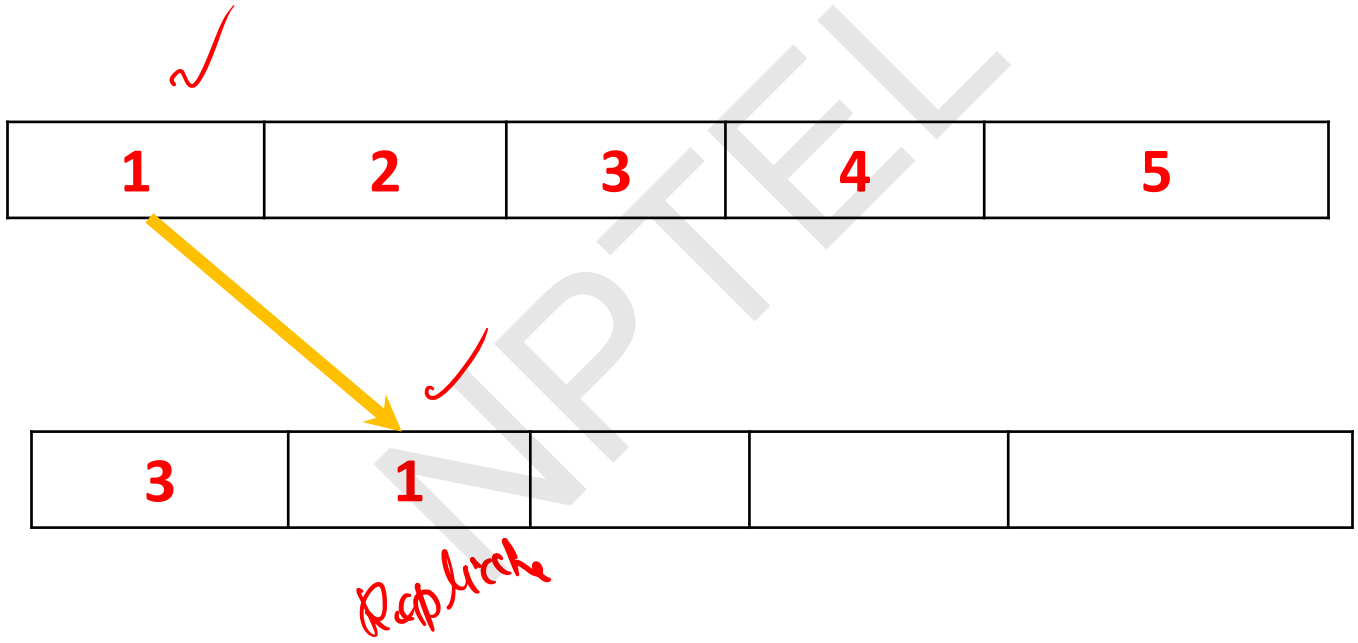
Sampling with Replacement

- At the first step, you pick at random an object, for example, the object number three.



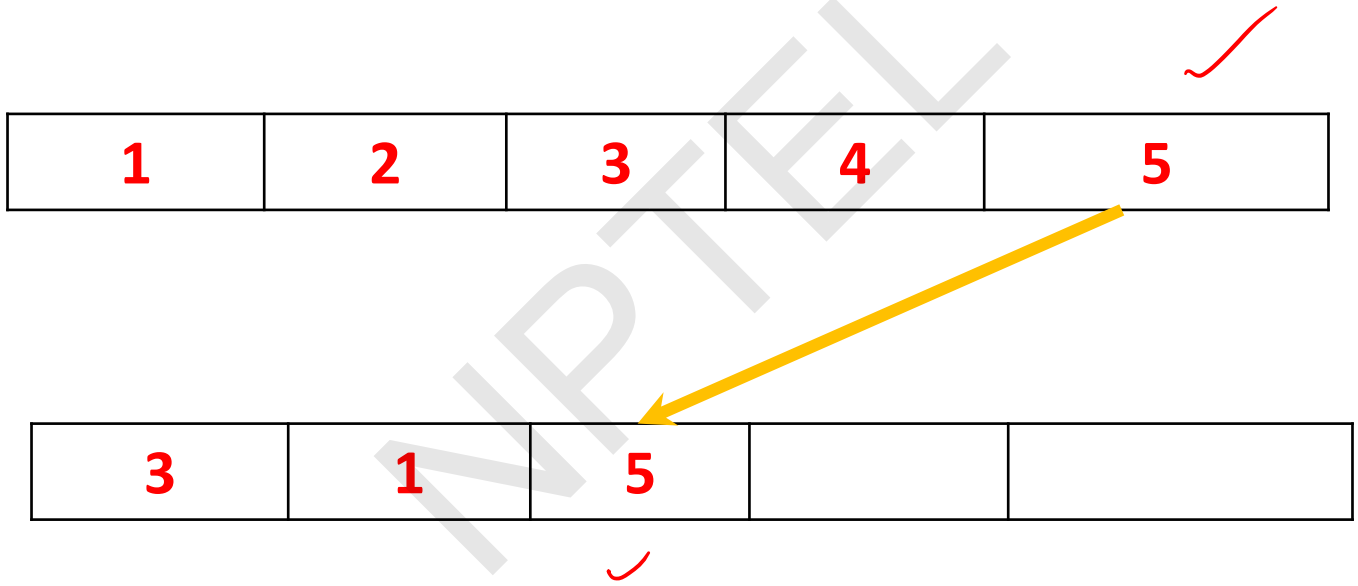
Sampling with Replacement

- Then you repeat it and pick the object number one.



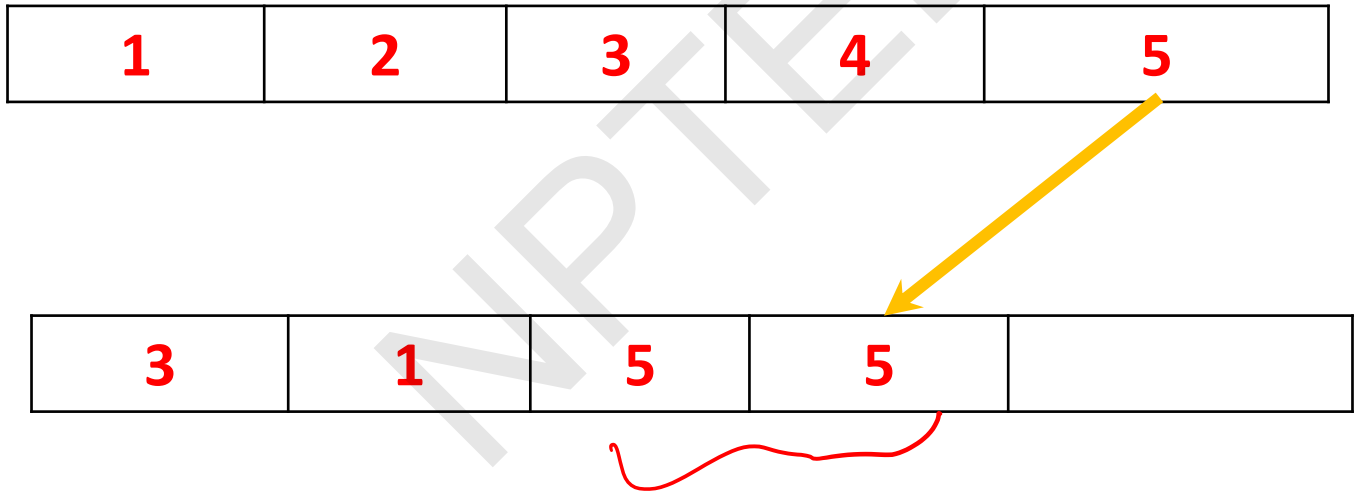
Sampling with Replacement

- Then the object number five.



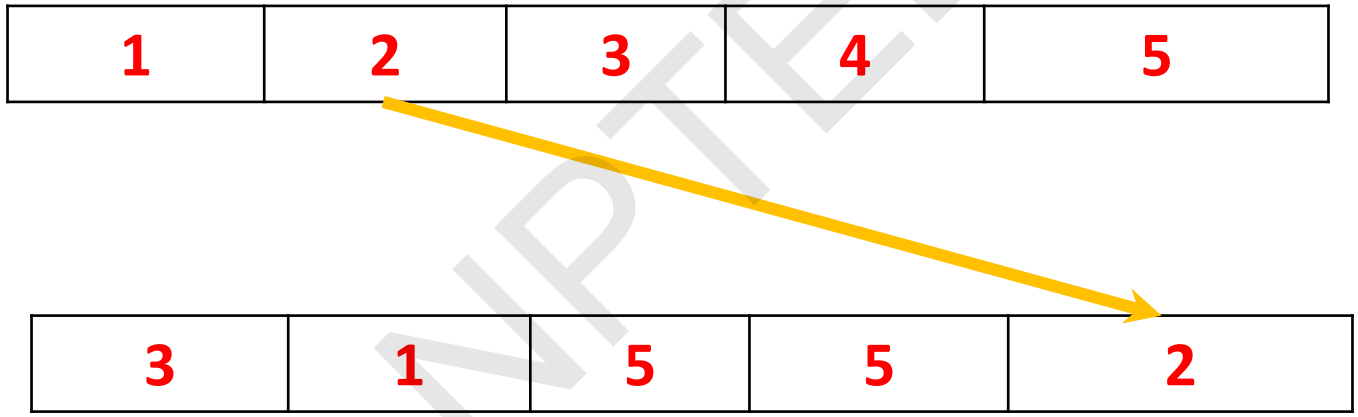
Sampling with Replacement

- Then possibly you can pick again the object number five, because at each iteration you pick an object at random, and there is no correlation with the previous step.



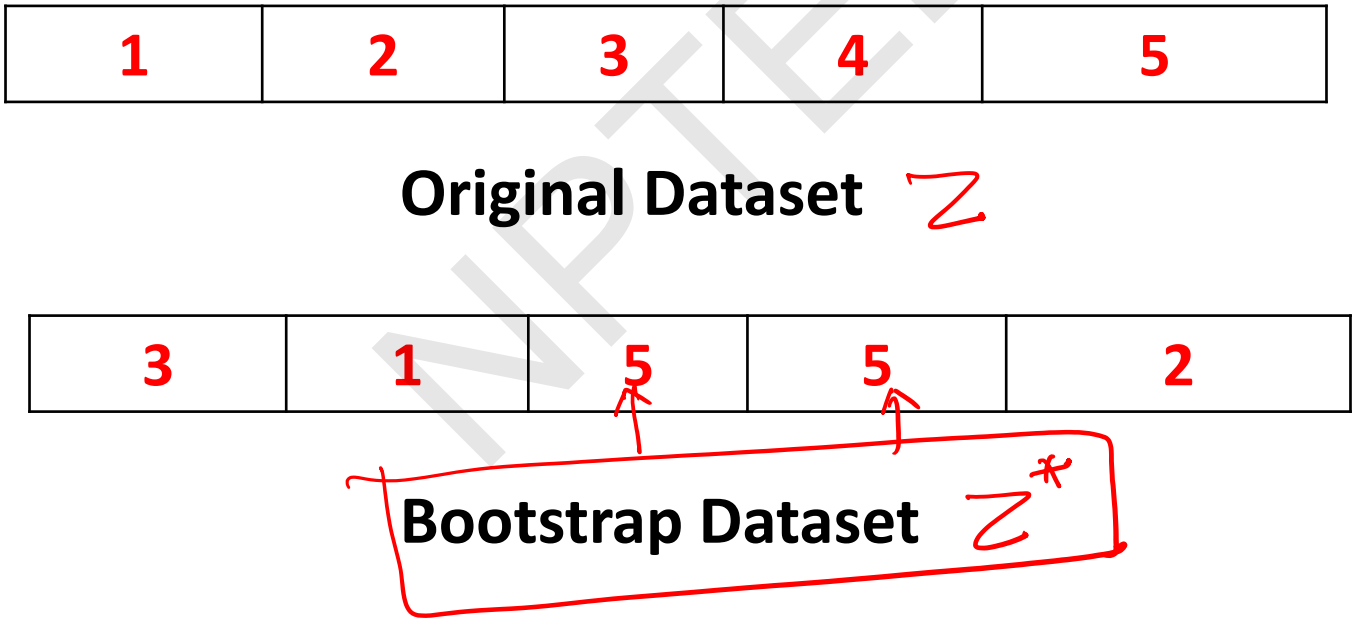
Sampling with Replacement

- And finally, you pick the object number two.



Sampling with Replacement

After bootstrapping we have a new data set. The size of this data set is the number of elements in the original data set. But its content, as you can see, is slightly different. Some objects may be missing and other objects may be present several times, more than once.



Bagging

- It was the second idea essential for understanding of the random forest algorithm.
- Bagging (Bootstrap Aggregation): It is a general method for averaging predictions of other algorithms, not decision trees, but any other algorithm in general.
- Bagging works because it reduces the variance of the prediction.

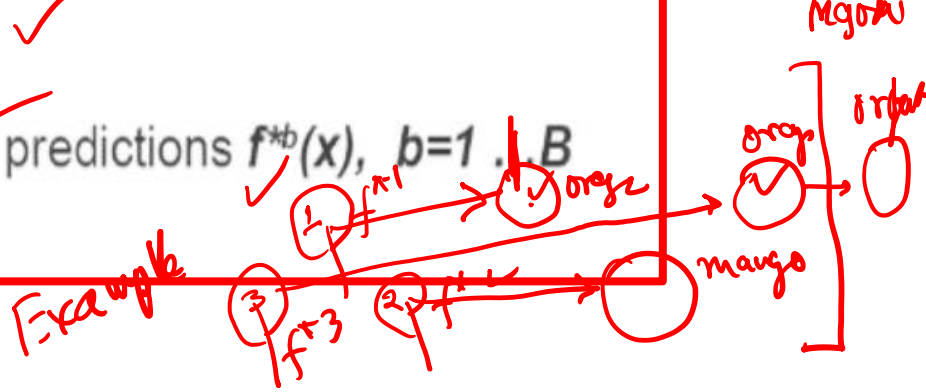
Algorithm: Bagging

Input: training set $Z=\{(x_1, y_1),\ldots,(x_n, y_n)\}$,
B – number of iterations
Machine learning method M

1. For b=1…B:
2. Draw a bootstrap sample Z^{*b} of size n from training data
3. Apply method M to the dataset Z^{*b} and obtain a model $f(x)^{*b}$
4. Return: ensemble $\{f^{*1}\ldots f^{*B}\}$

Prediction with ensemble:

- Regression: $f(x)=\frac{1}{n}\sum_{b=1}^{B} f^{*b}(x)$
- Classification: majority vote of all predictions $f^{*b}(x)$, $b=1\ldots B$



Why does Bagging work ?

- Model f(x) has higher predictive power than any single $f^{xb}(x)$, b=1,…,B

- Most of situations with any machine learning method in the core, the quality of such aggregated predictions will be better than of any single prediction.

Why does bagging works?

- This phenomenon is based on a very general principle which is called the bias variance trade off. You can consider the training data set to be random by itself.

Why does Bagging work ?

- **Why is it so? What is the training data set?**

- In the real situation, the training data set may be a user behavior in Internet, for example, web browsing, using search engine, doing clicks on advertisement, and so on.

- Other examples of training data sets are physical measurements. For example, temperature, locations. date, time, and so on. And all these measurements are essentially stochastic.

- If you can repeat the same experiment in the same conditions, the measurements actually will be different because of the noise in measurements, and since user behavior is essentially stochastic and not exactly predictable. Now, you understand that the training data set itself is random.

Why does Bagging work ?

- **Bagging:** It is an averaging over a set of possible datasets, removing noisy and non-stable parts of models.

- After averaging, the noisy parts of machine learning model will vanish out, whereas stable and reliable parts will remain. The quality of the average model will be better than any single model.

Summary

- **Bootstrap:** A method for generating different replicas of the dataset
- **Bagging (Bootstrap Aggregation):** A method for averaging predictions and reducing prediction's variance
- Bagging improves the quality of almost any machine learning method.
- Bagging is very time consuming for large data sets.

Random Forest

Random Forest

- **Random Forest** Algorithm is a bagging of de-correlated decision trees.

Algorithm: Random Forest

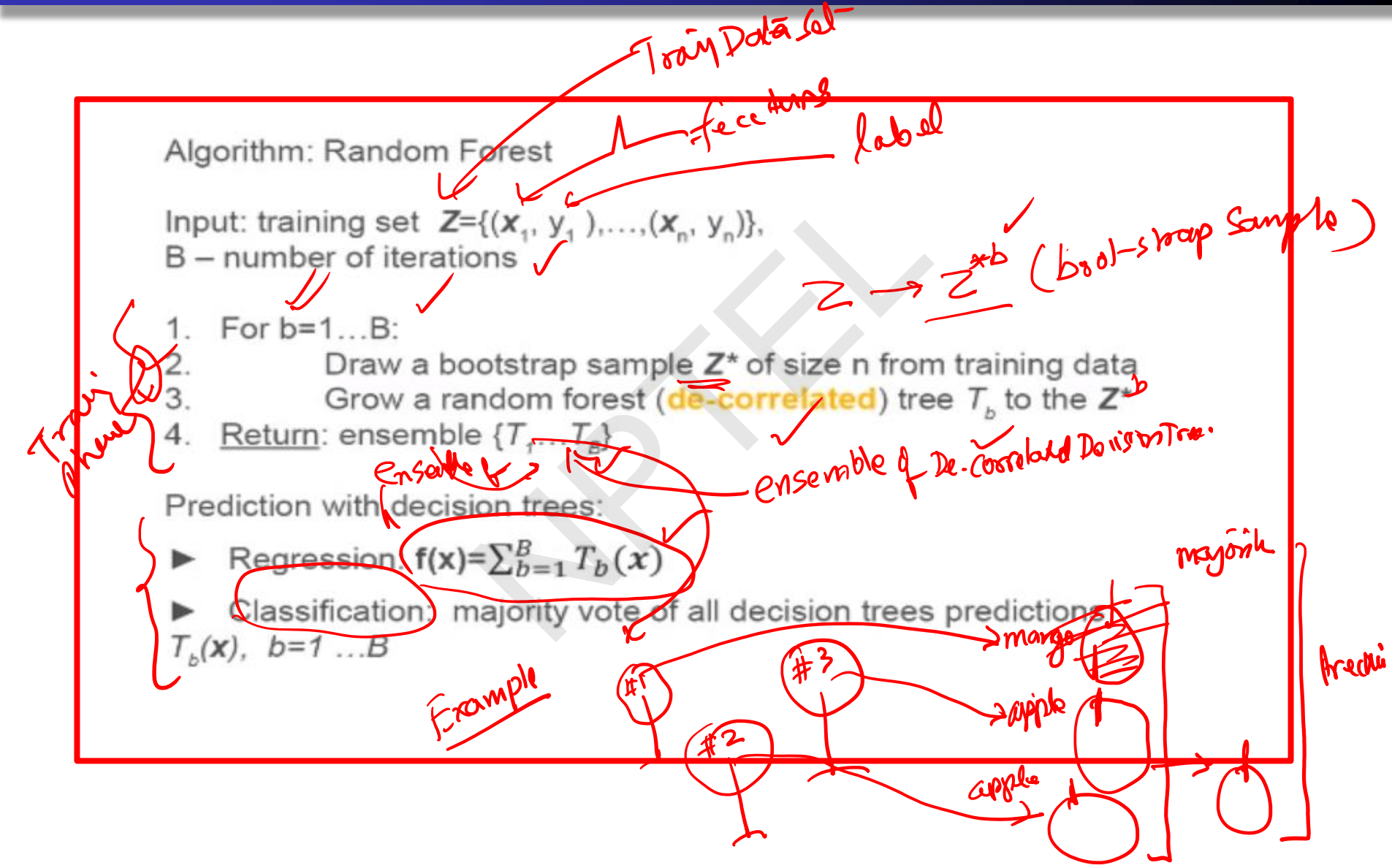
Algorithm: Random Forest

Input: training set $Z=\{(x_1, y_1),\ldots,(x_n, y_n)\}$,
B – number of iterations

1. For b=1…B:
2. Draw a bootstrap sample Z^* of size n from training data
3. Grow a random forest (de-correlated) tree T_b to the Z^{*b}
4. Return: ensemble $\{T_1 \ldots T_B\}$

Prediction with decision trees:

- Regression: $f(x)=\sum_{b=1}^{B} T_b(x)$
- Classification: majority vote of all decision trees predictions $T_b(x)$, b=1 …B



How to grow a random forest decision tree

- The tree is built **greedily** from top to bottom
- Select $m \leq p$ of the input variables at random as candidates for splitting
- Each split is selected to **maximize information gain (IG)**

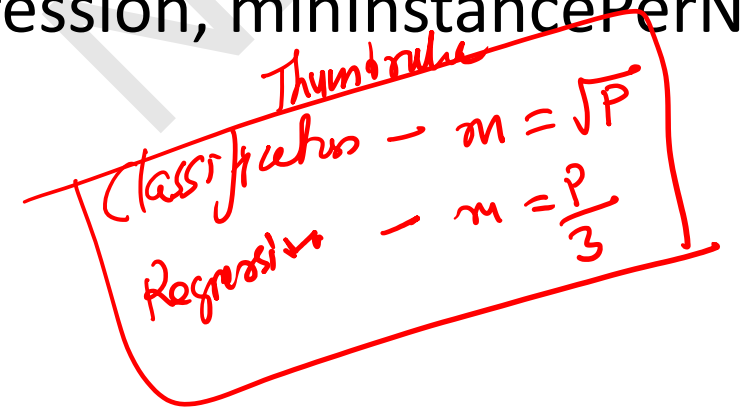
$$IG = \underbrace{\text{Impurity}(Z)}_{\text{Error before split}} - \underbrace{\left(\frac{|Z_L|}{|Z|}\text{Impurity}(Z_L) + \frac{|Z_R|}{|Z|}\text{Impurity}(Z_R)\right)}_{\text{Error after split}}$$

How to grow a random forest decision tree

- Select m ≤ p of the input variables at random as candidates for splitting

- **Recommendations from inventors of Random Forests:**

- m=√p for classification, minInstance PerNode =1
- m= p/3for regression, minInstancePerNode=5

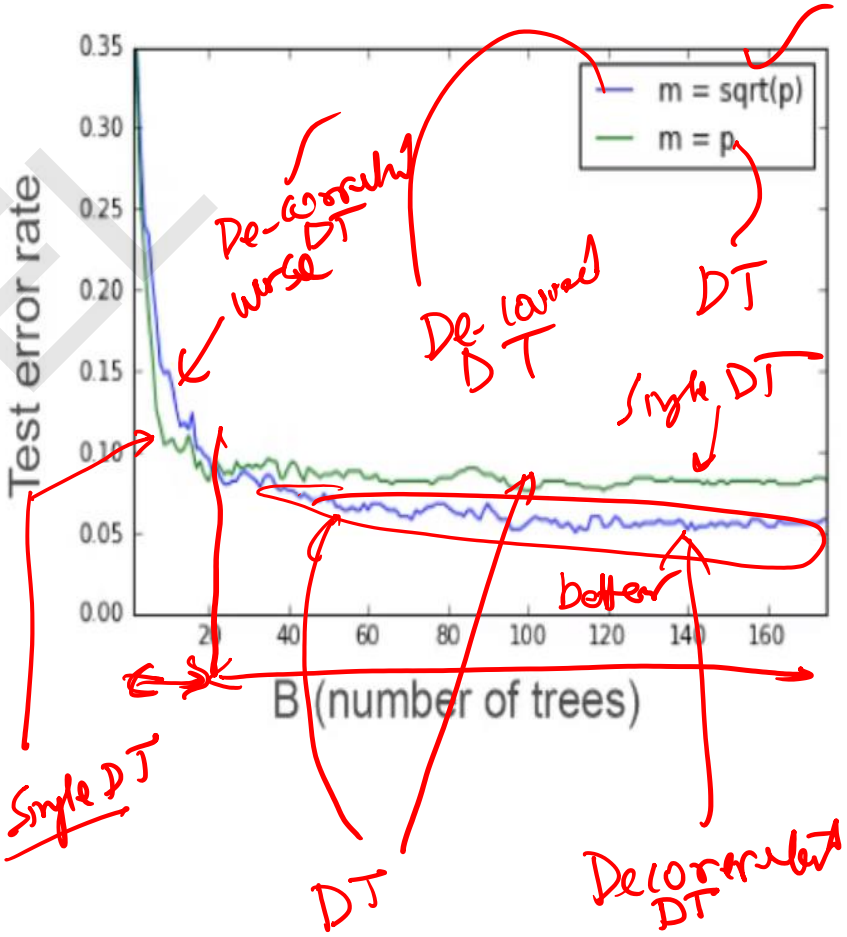


Random forest

Here are the results of training of two random force. The first variant is marked with green here, and either the variant were at each step m equals speed.

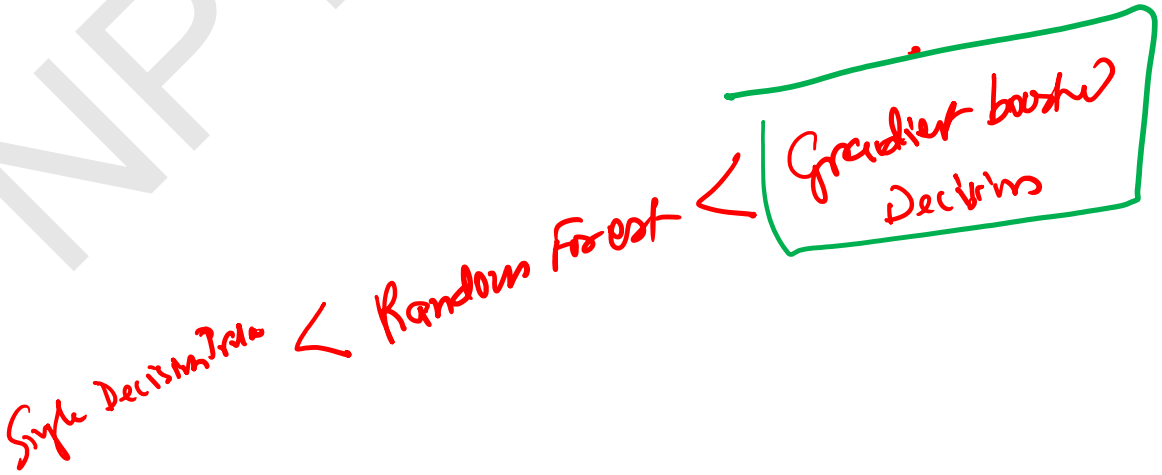
It means that at each step, we grow a regular decision tree and you find the best split among all the variables. And the blue line, it is a de-correlated decision tree.

In this situation, we randomly pick m equals square root of b. And all the trees can be built using different subsets of variables. As we can see at this diagram, at the initial stage, the variant is m equals square root b is worse before 20 iterations. But eventually, this variant of the Random Forest algorithm converges to the better solution.



Summary

- Random Forest is a good method for a **general purpose classification/regression problems** (typically slightly worse than gradient boosted decision trees)



Summary

- **Automatically handle interactions of features:** Of course, this algorithm can automatically handle interactions of features because this could be done by a single decision tree.

- **Computational scalability:** Of course, this algorithm can automatically handle interactions of features because this could be done by a single decision tree.

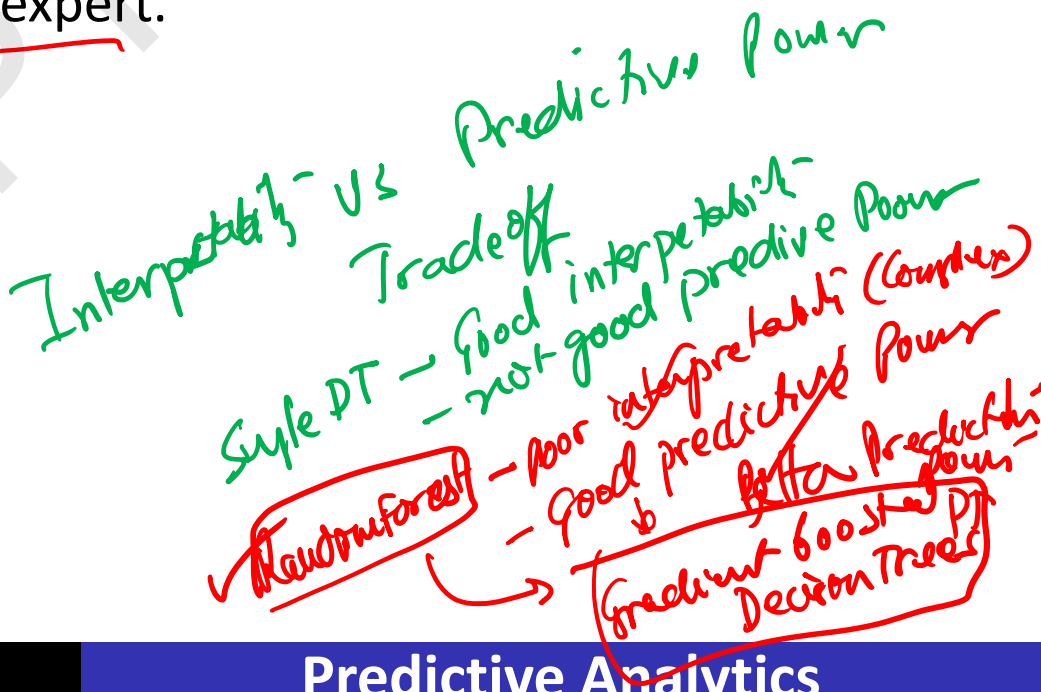
- In the Random Forest algorithm, each tree can be built independently on other trees. It is an important feature and I would like to emphasize it. That is why the Random Forest algorithm east is essentially parallel.

- The Random Forest could be trained in the distributed environment with the high degree of parallelization.

Summary

- **Predictive Power:** As far as predictive power of the Random Forest is, on the one hand better than a single decision tree, but it is slightly worse than gradient boosted decision trees.

- **Interpretability:** Here you'll lose the interpretability because their composition of hundreds or thousands Random Forest decision trees cannot be analyzed by human expert.



Gradient Boosted Decision Trees

Regression

Boosting

- **Boosting:** It is a method for combining outputs of many weak classifiers to produce a powerful ensemble.

- There are several variants of boosting algorithms, AdaBoost, BrownBoost, LogitBoost, and Gradient Boosting.

Big Data

- Large number of training examples.
- Large number of features describing objects.
- In this situation, It is very natural to assume that you would like to train a really complex model, even having such a great amount of data and hopefully, this model will be accurate.
- There are two basic ways, in machine learning, to build complex models.
- The first way is to start with a complex model from the very beginning, and fit its parameters. This is exactly the way how neural network operates.
- And the second way is to build a complex model iteratively. You can build a complex model iteratively, where each step requires training of a simple model. In context of boosting, these models are called weak classifiers, or base classifiers.

Regression

Given a training set: Z={(x1,y1),…,(xn,yn)}

Xi- features, yi-targets (real values)

Goal is to find f(x) using training set, such as

$$\min \sum_{(x,y) \in T} (f(x) - y)^2$$

At test set T={(x1,y1),…,(xn,yn)}

How to build f(x)

Gradient Boosted Trees for Regression

How to build such f(x)?

In boosting, our goal is to build the function f(x) iteratively. We suppose that this function f(x) is just a sum of other simple functions, $h_m(x)$.

And particular, you assume that each function $h_m(x)$ is a decision tree.

$$f(\boldsymbol{x}) = \sum_{m=1}^{M} h_m(\boldsymbol{x})$$

$h_m(\boldsymbol{x})$ - a decision tree

Algorithm: Gradient Boosted Trees for Regression

Algorithm: Gradient Boosted Trees for Regression

Input: training set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$

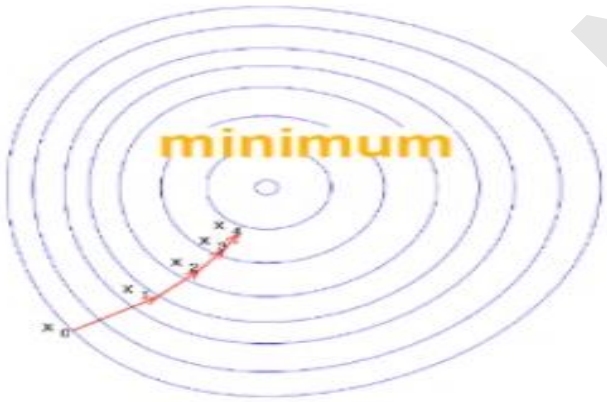
M – number of iterations

1. $f_0(x) = \frac{1}{n}\sum_{i=1}^{n} y_i$
2. For m=1…M:
3. $\hat{y}_i = y_i - f_{m-1}(x_i)$ (residual)
4. Fit a decision tree $h_m(x)$ to the targets $\hat{y}_i$
   (auxiliary training set $\{(x_1, \hat{y}_1), \ldots, (x_n, \hat{y}_n)\}$)
5. $f_m(x) = f_{m-1}(x) + \nu h_m(x)$
6. Return $f_m(x)$

ν - regularization (learningRate), recommended ≤ 0.1

Optimization Theory

- You may have noticed that **gradient boosting is somewhat similar to the gradient descent in the optimization theory.** If we want to minimize a function in the optimization theory using the gradient descent, we make a small step in the direction opposite to the gradient.
- Gradient of the function, by definition, is a vector which points to the direction with the fastest increase. Since we want to minimize the function, we must move to the direction opposite to the gradient. To ensure convergence, we must make very small steps. So you're multiplying each gradient by small constant, which is called step size. It is very similar to what we do in gradient boosting.
- And gradient boosting is considered to be a minimization in the functional space.



$$f_m(x) = f_0(x) + vh_1(x) + vh_2(x) + \ldots$$

Boosting-Minimization in the functional space

Summary

- **Boosting** is a method for combining outputs of many weak classifiers or regressors to produce a powerful ensemble.
- **Gradient Boosting** is a gradient descent minimization of the target function in the functional space.
- **Gradient Boosting with Decision Trees** is considered to be the best algorithm for general purpose classification or regression problems.

Gradient Boosted Decision Trees

Classification

Classification

Given a training set: Z={(x1,y1),…,(xn,yn)}

Xi- features, yi-class labels (0,1)

Goal is to find f(x) using training set, such as

$$\min \sum_{(x,y) \in T} [f(x) \neq y]$$

At test set T={(x1,y1),…,(xn,yn)}

How to build f(x) ?

Gradient Boosted Trees for Classification

How we are going to build such the function, f(x)?

We use a probabilistic model by using the following expression.

$$P(y = 1|\boldsymbol{x}) = \frac{1}{1 + \exp(-\sum_{m=1}^{M} h_m(\boldsymbol{x}))}$$

$h_m(\boldsymbol{x})$ - a decision tree

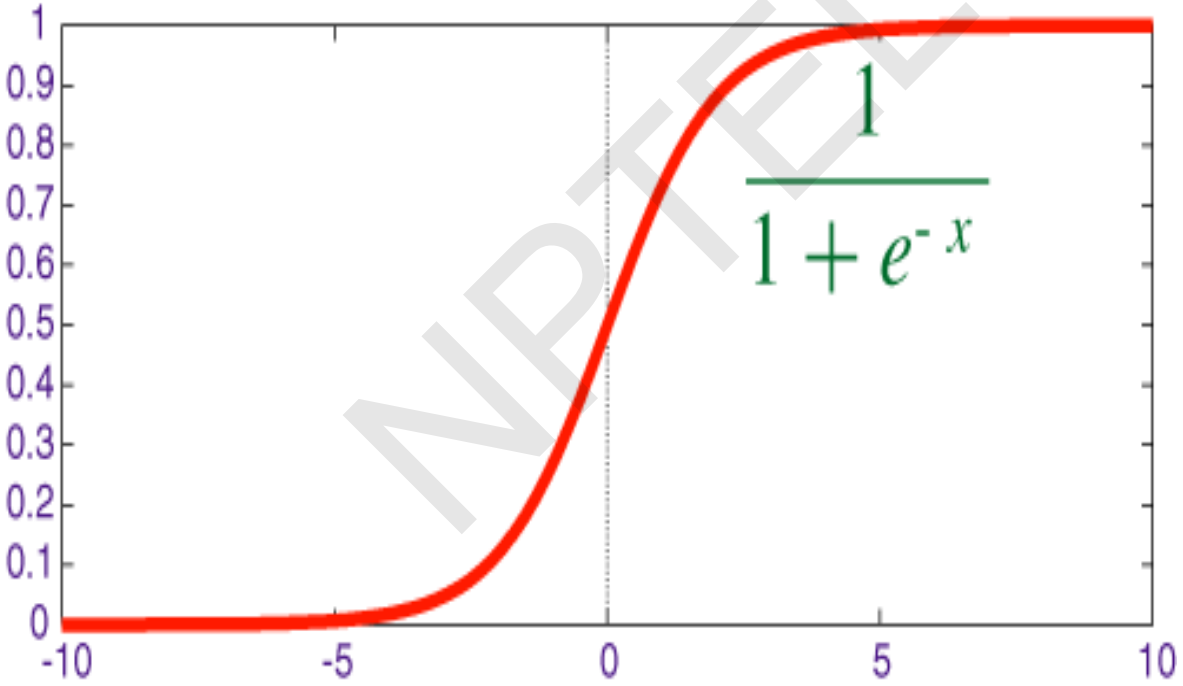
$$0 < P(y = 1|\boldsymbol{x}) < 1$$

We model the probability of belonging of an object to the first class. And here inside the exp, there is the sum of hm(x), and each hm(x) is a decision tree.

We can easily check that such expression for probability will be always between zero and one, so it is normal regular probability.

Sigmoid Function

This function is called the sigmoid function, which maps all the real values into the range between zero and one.



Let us denote the sum of all hm(x) by f(x). It is an ensemble of decision trees. Then you can write the probability of belonging to the first class in a simple way using f(x). And the main idea which is used here is called the principle of maximum likelihood.

What is it? First of all, what is the likelihood?

Likelihood is a probability of absorbing some data given a statistical model. If we have a data set with an objects from one to n, then the probability of absorbing such data set is the multiplication of probabilities for all single objects. This multiplication is called the likelihood.

$$f(\boldsymbol{x}) = \sum_{m=1}^{M} h_m(\boldsymbol{x})$$

$$P(y = 1|\boldsymbol{x}) = \frac{1}{1 + \exp(-f(\boldsymbol{x}))}$$

Likelihood:

$$\prod_{i=1}^{n} P(y_i|\boldsymbol{x}_i) = P(y_1|\boldsymbol{x}_1) \cdot \ldots \cdot P(y_n|\boldsymbol{x}_n)$$

The principle of maximum likelihood

- **Algorithm:** find a function f(x) maximizing the likelihood
- **Equivalent:** find a function f(x) maximizing the logarithm of the likelihood
- (since logarithm is a monotone function)

$$Q[f] = \sum_{i=1}^{n} \log(P(y_i|\boldsymbol{x}_i))$$

$$\max Q[f]$$

The principle of maximum likelihood

We will denote by Q[f] the logarithm of the likelihood, and now, it is sum of all logarithms of probabilities and you are going to maximize this function.

We will use shorthand for this logarithm L(yi, f(x)i). It is the logarithm of probability. And here, we emphasize that this logarithms depend actually on the true label, yi and our prediction, f(x)i. Now, Q[f] is a sum of L(yi, f(x)i).

$$L(y_i, f(\boldsymbol{x}_i)) = \log(P(y_i|\boldsymbol{x}_i))$$

$$Q[f] = \sum_{i=1}^{n} L(y_i, f(\boldsymbol{x}_i))$$

Prediction label

True label for i th data object

Algorithm: Gradient Boosted Trees for Classification

Input: training set $Z=\{(x_1, y_1), \ldots, (x_n, y_n)\}$,
M – number of iterations

1. $f_0(x) = log \frac{p_1}{1-p_1}$ p_1- part of objects of first class
2. For m=1…M:
3. $g_i = \frac{dL(y_i, f_m(x_i))}{df_m(x_i)}$
4. Fit a decision tree $h_m(x_i)$ to the target g_i
   (auxiliary training set $\{(x_1, g_i), \ldots, (x_n, g_n)\}$)
5. $\rho_m = \underset{\rho}{\text{argmax}}\, Q[f_{m-1}(x) + \rho h_m(x)]$
6. $f_m(x) = f_{m-1}(x) + v\rho_m h_m(x_i)$
7. Return: $f_M(x)$

v - regularization (learningRate), recommended ≤ 0.1

Stochastic Boosting

Algorithm: Gradient Boosted Trees for Classification

Input: training set $Z=\{(x_1, y_1), \ldots, (x_n, y_n)\}$,
M – number of iterations

1. $f_0(x) = log \frac{p_1}{1-p_1}$ p_1- part of objects of first class
2. For m=1…M:
3. $g_i = \frac{dL(y_i, f_m(x_i))}{df_m(x_i)}$
4. Fit a decision tree $h_m(x_i)$ to the target g_i
   (auxiliary training set $\{(x_1, g_i), \ldots, (x_n, g_n)\}$)
5. $\rho_m = \underset{\rho}{\text{argmax}}\, Q[f_{m-1}(x) + \rho h_m(x)]$
6. $f_m(x) = f_{m-1}(x) + v\rho_m h_m(x_i)$
7. Return: $f_M(x)$

v - regularization (learningRate), recommended ≤ 0.1

Algorithm: Gradient Boosted Trees for Classification + Stochastic Boosting

Input: training set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, M – number of iterations

1. $f_0(x) = \frac{1}{n}\sum_{i=1}^{n} y_i$
2. For m=1…M:
3. $g_i = \frac{dL(y_i, f_m(x_i))}{df_m(x_i)}$
4. Fit a decision tree $h_m(x_i)$ to the target g_i
   (auxiliary training set $\{(x_1, g_1), \ldots, (x_k, g_k)\}$), *k=0.5n*
   created by random sampling with replacement
5. $\rho_m = \underset{\rho}{\text{argmax}}\, Q[f_{m-1}(x) + \rho h_m(x)]$
6. $f_m(x) = f_{m-1}(x) + v\rho_m h_m(x_i)$
7. Return $f_M(x)$

v - regularization (learningRate), recommended ≤ 0.1

Sampling with Replacement

n=8

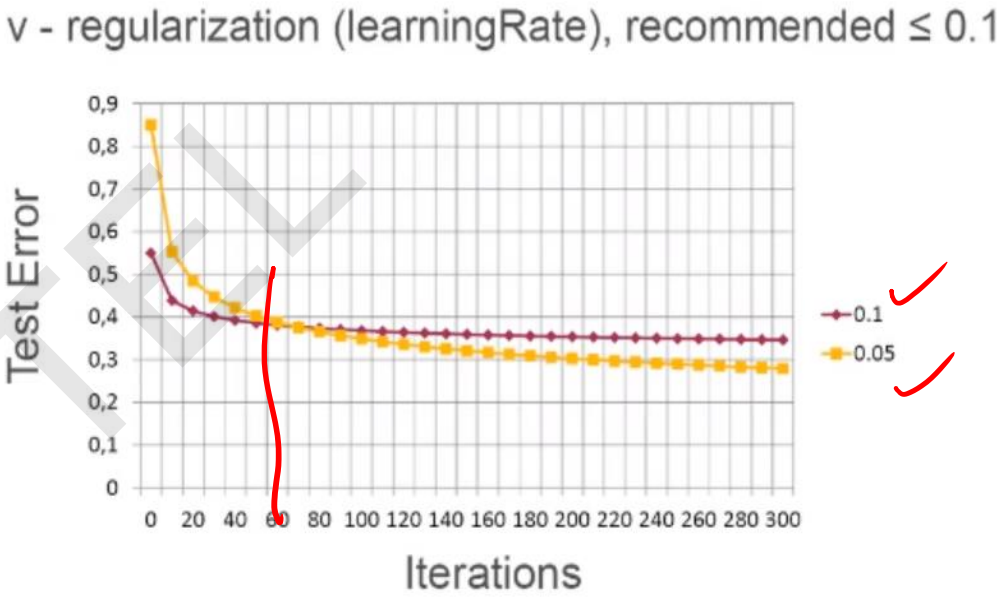
1	2	3	4	5	6	7	8

k=4

7	3	1	3

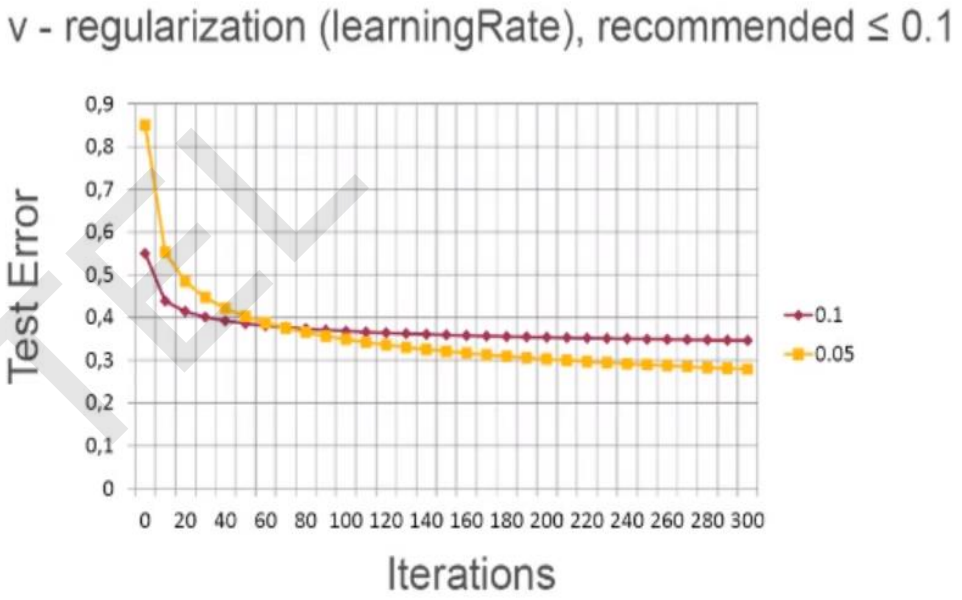
Tips for Usage

- First of all, it is important to understand how the regularization parameter works. In this figure, you can see the behavior of the gradient boosted decision trees algorithm with two variants of this parameter, 0.1 and 0.05.
- What happens here, at the initial stage of learning the variant with parameter 0.1 is better because it has lower testing error.



Tips for Usage

- At each iteration, you measure a testing error of our ensemble on the hold out data set.
- But eventually, the variant with lower regularization, 0.05, reach lower testing error. Finally, this variant turns out to be superior.
- It is a very typical behavior, and you should not stop your algorithm after several dozen iterations. You should proceed until convergence.
- Convergence happens when your testing error doesn't change a lot. The variant with lower regularization converges more slowly, but eventually it builds a better model.



Tips for Usage

- The recommended **learningRate** should be less or equal than 0.1.
- The bigger your data set is, the larger **number of iterations** should be.
- The recommended **number of iterations** ranges from several hundred to several thousand.
- Also, the more features you have in your data set, the deeper your decision tree should be.
- These are very general rules because the bigger your data set is, the more features you have, the more complex model you can build without overfitting.

Summary

- It is a **best method** for a general purpose classification and regression problems.
- It **automatically handles interactions** of features, because in the core, it is based on decision trees, which can combine several features in a single tree.
- But also, this algorithm is **computationally scalable**. It can be effectively executed in the distributed environment, for example, in Spark. So it can be executed at the top of the Spark cluster.

Summary

- But also, this algorithm has a very **good predictive power**. But unfortunately, the models are not interpretable.
- The **final ensemble effects is very large** and cannot be analyzed by a human expert.
- There is always a tradeoff in machine learning, between predictive power and interpretability, because the more complex and accurate your model is, the harder is the analysis of this model by a human.

Spark ML, Decision Trees and Ensembles

Introduction

- This lesson will be about using Spark ML for doing classification and regression with decision trees, and in samples of decision trees.

- First of all, you are going to create SparkContext and SparkSession

```
In [ ]: from pyspark import SparkContext
        from pyspark.sql import SparkSession

In [ ]: sc = SparkContext(appName = "module3_week4")

In [ ]: ! echo $PYSPARK_SUBMIT_ARGS

In [ ]: spark = SparkSession.Builder().getOrCreate() # required for dataframes
```

- Now you are downloading a dataset.

```
In [*]: !wget https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wis

--2017-07-31 11:09:06--  https://archive.ics.uci.edu/ml/machine-learning-databa
ses/breast-cancer-wisconsin/wdbc.data
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.249
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.249|:443...
 connected.
HTTP request sent, awaiting response... 200 OK
Length: 124103 (121K) [text/plain]
Saving to: 'wdbc.data.11'

100%[======================================>] 124,103      158KB/s   in 0.8s

2017-07-31 11:09:07 (158 KB/s) - 'wdbc.data.11' saved [124103/124103]

In [ ]: !head -n 3 wdbc.data

In [ ]: !wc -l wdbc.data
```

- Let's explore this dataset a little bit. The first column is ID of observation. The second column is the diagnosis, and other columns are features which are comma separated.

```
In [6]: !head -n 3 wdbc.data

842302,M,17.99,10.38,122.8,1001,0.1184,0.2776,0.3001,0.1471,0.2419,0.07871,1.09
5,0.9053,8.589,153.4,0.006399,0.04904,0.05373,0.01587,0.03003,0.006193,25.38,1
7.33,184.6,2019,0.1622,0.6656,0.7119,0.2654,0.4601,0.1189
842517,M,20.57,17.77,132.9,1326,0.08474,0.07864,0.0869,0.07017,0.1812,0.05667,
0.5435,0.7339,3.398,74.08,0.005225,0.01308,0.0186,0.0134,0.01389,0.003532,24.9
9,23.41,158.8,1956,0.1238,0.1866,0.2416,0.186,0.275,0.08902
84300903,M,19.69,21.25,130,1203,0.1096,0.1599,0.1974,0.1279,0.2069,0.05999,0.74
56,0.7869,4.585,94.03,0.00615,0.04006,0.03832,0.02058,0.0225,0.004571,23.57,25.
53,152.5,1709,0.1444,0.4245,0.4504,0.243,0.3613,0.08758

In [ ]: !wc -l wdbc.data

In [ ]: #1) ID number
        #2) Diagnosis (M = malignant, B = benign)
        #3) Features

In [ ]: from pyspark.ml.linalg import Vectors
        from pyspark.ml.feature import StringIndexer

In [ ]: # Load a text file and convert each line to a Row.
```

- So these features are results of some analysis and measurements. There are total 569 examples in this dataset.

```
In [6]: !head -n 3 wdbc.data

842302,M,17.99,10.38,122.8,1001,0.1184,0.2776,0.3001,0.1471,0.2419,0.07871,1.09
5,0.9053,8.589,153.4,0.006399,0.04904,0.05373,0.01587,0.03003,0.006193,25.38,1
7.33,184.6,2019,0.1622,0.6656,0.7119,0.2654,0.4601,0.1189
842517,M,20.57,17.77,132.9,1326,0.08474,0.07864,0.0869,0.07017,0.1812,0.05667,
0.5435,0.7339,3.398,74.08,0.005225,0.01308,0.0186,0.0134,0.01389,0.003532,24.9
9,23.41,158.8,1956,0.1238,0.1866,0.2416,0.186,0.275,0.08902
84300903,M,19.69,21.25,130,1203,0.1096,0.1599,0.1974,0.1279,0.2069,0.05999,0.74
56,0.7869,4.585,94.03,0.00615,0.04006,0.03832,0.02058,0.0225,0.004571,23.57,25.
53,152.5,1709,0.1444,0.4245,0.4504,0.243,0.3613,0.08758

In [7]: !wc -l wdbc.data

569 wdbc.data

In [ ]: #1) ID number
        #2) Diagnosis (M = malignant, B = benign)
        #3) Features

In [ ]: from pyspark.ml.linalg import Vectors
        from pyspark.ml.feature import StringIndexer
```

- First of all you need to transform the label, which is either M or B from the second column. You should transform it from a string to a number.
- You use a StringIndexer object for this purpose, and first of all you need to load all these datasets.

```
In [6]: !head -n 3 wdbc.data

842302,M,17.99,10.38,122.8,1001,0.1184,0.2776,0.3001,0.1471,0.2419,0.07871,1.09
5,0.9053,8.589,153.4,0.006399,0.04904,0.05373,0.01587,0.03003,0.006193,25.38,1
7.33,184.6,2019,0.1622,0.6656,0.7119,0.2654,0.4601,0.1189
842517,M,20.57,17.77,132.9,1326,0.08474,0.07864,0.0869,0.07017,0.1812,0.05667,
0.5435,0.7339,3.398,74.08,0.005225,0.01308,0.0186,0.0134,0.01389,0.003532,24.9
9,23.41,158.8,1956,0.1238,0.1866,0.2416,0.186,0.275,0.08902
84300903,M,19.69,21.25,130,1203,0.1096,0.1599,0.1974,0.1279,0.2069,0.05999,0.74
56,0.7869,4.585,94.03,0.00615,0.04006,0.03832,0.02058,0.0225,0.004571,23.57,25.
53,152.5,1709,0.1444,0.4245,0.4504,0.243,0.3613,0.08758

In [7]: !wc -l wdbc.data

569 wdbc.data

In [ ]: #1) ID number
        #2) Diagnosis (M = malignant, B = benign)
        #3) Features

In [ ]: from pyspark.ml.linalg import Vectors
        from pyspark.ml.feature import StringIndexer
```

- Then you create a Spark DataFrame, which is stored in the distributed manner on cluster.

```
In [9]: from pyspark.ml.linalg import Vectors
        from pyspark.ml.feature import StringIndexer

In [10]: # Load a text file and convert each line to a Row.

         data = []

         with open("wdbc.data") as infile:
             for line in infile:
                 tokens = line.rstrip("\n").split(",")
                 y = tokens[1]
                 features = Vectors.dense([float(x) for x in tokens[2:]])

                 data.append((y, features))

In [11]: inputDF = spark.createDataFrame(data, ["label", "features"])

In [*]: inputDF.show()

In [ ]: stringIndexer = StringIndexer(inputCol = "label", outputCol = "labelIndexed")
        si_model = stringIndexer.fit(inputDF)
        inputDF2 = si_model.transform(inputDF)
```

- inputDF DataFrame has two columns, label and features. We use an object vector for creating a vector column in this dataset.

```
In [11]: inputDF = spark.createDataFrame(data, ["label", "features"])

In [12]: inputDF.show()
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|    M|[17.99,10.38,122....|
|    M|[20.57,17.77,132....|
|    M|[19.69,21.25,130....|
|    M|[11.42,20.38,77.5...|
|    M|[20.29,14.34,135....|
|    M|[12.45,15.7,82.57...|
|    M|[18.25,19.98,119....|
|    M|[13.71,20.83,90.2...|
|    M|[13.0,21.82,87.5,...|
|    M|[12.46,24.04,83.9...|
|    M|[16.02,23.24,102....|
|    M|[15.78,17.89,103....|
|    M|[19.17,24.8,132.4...|
|    M|[15.85,23.95,103....|
|    M|[13.73,22.61,93.6...|
|    M|[14.54,27.54,96.7...|
|    M|[14.68,20.13,94.7...|
```

- Then we can do string indexing. So Spark now enumerates all the possible labels in a string form, and transforms them to the label indexes.

```
|    M|[13.0,21.82,87.5,...|
|    M|[12.46,24.04,83.9...|
|    M|[16.02,23.24,102....|
|    M|[15.78,17.89,103....|
|    M|[19.17,24.8,132.4...|
|    M|[15.85,23.95,103....|
|    M|[13.73,22.61,93.6...|
|    M|[14.54,27.54,96.7...|
|    M|[14.68,20.13,94.7...|
|    M|[16.13,20.68,108....|
|    M|[19.81,22.15,130....|
|    B|[13.54,14.36,87.4...|
+-----+--------------------+
only showing top 20 rows
```

```
In [*]: stringIndexer = StringIndexer(inputCol = "label", outputCol = "labelIndexed")
        si_model = stringIndexer.fit(inputDF)
        inputDF2 = si_model.transform(inputDF)

In [ ]: inputDF2.show()
```

- And now label M is equivalent 1, and B label is equivalent 0.

```
In [14]: inputDF2.show()
+-----+--------------------+------------+
|label|            features|labelIndexed|
+-----+--------------------+------------+
|    M|[17.99,10.38,122....|         1.0|
|    M|[20.57,17.77,132....|         1.0|
|    M|[19.69,21.25,130....|         1.0|
|    M|[11.42,20.38,77.5...|         1.0|
|    M|[20.29,14.34,135....|         1.0|
|    M|[12.45,15.7,82.57...|         1.0|
|    M|[18.25,19.98,119....|         1.0|
|    M|[13.71,20.83,90.2...|         1.0|
|    M|[13.0,21.82,87.5,...|         1.0|
|    M|[12.46,24.04,83.9...|         1.0|
|    M|[16.02,23.24,102....|         1.0|
|    M|[15.78,17.89,103....|         1.0|
|    M|[19.17,24.8,132.4...|         1.0|
|    M|[15.85,23.95,103....|         1.0|
|    M|[13.73,22.61,93.6...|         1.0|
|    M|[14.54,27.54,96.7...|         1.0|
|    M|[14.68,20.13,94.7...|         1.0|
|    M|[16.13,20.68,108....|         1.0|
|    M|[19.81,22.15,130....|         1.0|
|    B|[13.54,14.36,87.4...|         0.0|
```

- First of all, you make training test splitting in the proportion 70% to 30%. And the first model you are going to evaluate is one single decision tree.

train/test split

```
In [15]: (trainingData, testData) = inputDF2.randomSplit([0.7, 0.3], seed = 23)
```

Training Decision Tree

```
In [ ]: from pyspark.ml.classification import DecisionTreeClassifier
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator

In [ ]: decisionTree = DecisionTreeClassifier(labelCol = "labelIndexed")

In [ ]: dtModel = decisionTree.fit(trainingData)

In [ ]: dtModel.numNodes

In [ ]: dtModel.depth

In [ ]: dtModel.featureImportances
```

- Here we are making import DecisionTreeClassifier object.
- We create a class which is responsible for training, and
- Then call the method fit to the training data, and obtain a decision tree model.

train/test split

```
In [15]: (trainingData, testData) = inputDF2.randomSplit([0.7, 0.3], seed = 23)
```

Training Decision Tree

```
In [ ]: from pyspark.ml.classification import DecisionTreeClassifier
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator

In [ ]: decisionTree = DecisionTreeClassifier(labelCol = "labelIndexed")

In [ ]: dtModel = decisionTree.fit(trainingData)

In [ ]: dtModel.numNodes

In [ ]: dtModel.depth

In [ ]: dtModel.featureImportances
```

- So the training was quite fast, and here are some results.
- Number of nodes and depths of decision tree, feature importance, total number of features used in this decision tree, and so on.

```
In [17]: decisionTree = DecisionTreeClassifier(labelCol = "labelIndexed")

In [18]: dtModel = decisionTree.fit(trainingData)

In [19]: dtModel.numNodes
Out[19]: 29

In [20]: dtModel.depth
Out[20]: 5

In [21]: dtModel.featureImportances
Out[21]: SparseVector(30, {1: 0.0589, 6: 0.0037, 10: 0.0112, 13: 0.0117, 20: 0.0324, 21:
         0.0302, 22: 0.7215, 24: 0.01, 26: 0.0191, 27: 0.1013})

In [22]: dtModel.numFeatures
Out[22]: 30

In [ ]: print dtModel.toDebugString
```

- We can even visualize this decision tree and explore it, and here is a structure of this decision tree.
- Here are splitting conditions If and Else, which predict values and leaves of our decision trees.

```
In [23]: print dtModel.toDebugString

DecisionTreeClassificationModel (uid=DecisionTreeClassifier_4653be4ce1bd9e589b6
4) of depth 5 with 29 nodes
  If (feature 22 <= 114.2)
   If (feature 27 <= 0.1613)
    If (feature 20 <= 16.57)
     If (feature 27 <= 0.1258)
      If (feature 10 <= 0.9289)
       Predict: 0.0
      Else (feature 10 > 0.9289)
       Predict: 1.0
     Else (feature 27 > 0.1258)
      If (feature 21 <= 32.85)
       Predict: 0.0
      Else (feature 21 > 32.85)
       Predict: 1.0
    Else (feature 20 > 16.57)
     If (feature 1 <= 16.54)
      Predict: 0.0
     Else (feature 1 > 16.54)
      If (feature 24 <= 0.1084)
       Predict: 0.0
      Else (feature 24 > 0.1084)
       Predict: 1.0
```

- Here we are applying a decision tree model to the test data, and obtain predictions.

```
In [*]: predictions = dtModel.transform(testData)

In [ ]: predictions.select('label', 'labelIndexed', 'probability', 'prediction').show()

In [ ]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
        accuracy = evaluator.evaluate(predictions)

        print("Test Error = %g" % (1.0 - accuracy))
```

- Here we can explore these predictions. The predictions are in the last column. And in this particular case, our model always predicts zero class.

```
In [25]: predictions.select('label', 'labelIndexed', 'probability', 'prediction').show()

+-----+------------+--------------------+----------+
|label|labelIndexed|         probability|prediction|
+-----+------------+--------------------+----------+
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
|    B|         0.0|[0.99086757990867...|       0.0|
+-----+------------+--------------------+----------+
```

- Now we can evaluate the accuracy of model.
- For this purpose, we use a MulticlassClassificationEvaluator with a metric named accuracy.
- The testing error is 3%, the model is quite accurate.

```
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
|    B|       0.0|[0.99086757990867...|       0.0|
+-----+----------+--------------------+----------+
only showing top 20 rows
```

```
In [26]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
         accuracy = evaluator.evaluate(predictions)

         print("Test Error = %g" % (1.0 - accuracy))
```

```
Test Error = 0.045197
```

Gradient Boosted Decision Trees

- First of all we import it, we create an object which will do this classification. Here you are specifying labelCol = "labelIndexed", and featuresCol = "features". Actually, it is not mandatory to specify feature column, because its default name is features. We can do it either with this argument or without this argument.

```
In [27]: from pyspark.ml.classification import GBTClassifier

In [ ]: gbdt = GBTClassifier(labelCol = "labelIndexed", featuresCol = "features", maxIter

In [ ]: gbdtModel = gbdt.fit(trainingData)

In [ ]: gbdtModel.featureImportances

In [ ]: gbdtModel.toDebugString

In [ ]: predictions = gbdtModel.transform(testData)
        predictions.select('label', 'labelIndexed', 'prediction').show()

In [ ]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
        accuracy = evaluator.evaluate(predictions)

        print("Test Error = %g" % (1.0 - accuracy))
```

Gradient Boosted Decision Trees

- We are going to do 100 iterations, and the default stepSize is 0.1.

GBDT

```
In [27]: from pyspark.ml.classification import GBTClassifier

In [28]: abelCol = "labelIndexed", featuresCol = "features", maxIter = 100, stepSize = 0.1)

In [29]: gbdtModel = gbdt.fit(trainingData)

In [30]: gbdtModel.featureImportances

Out[30]: SparseVector(30, {0: 0.0176, 1: 0.0631, 2: 0.0012, 3: 0.0066, 4: 0.0096, 5: 0.0
         068, 6: 0.0045, 7: 0.0106, 8: 0.0016, 9: 0.0, 10: 0.006, 11: 0.0025, 12: 0.003
         9, 13: 0.0077, 14: 0.0032, 15: 0.0088, 16: 0.0057, 17: 0.0007, 18: 0.0034, 19:
          0.0023, 20: 0.1718, 21: 0.0266, 22: 0.4246, 23: 0.0839, 24: 0.0214, 25: 0.012
         6, 26: 0.0476, 27: 0.0419, 28: 0.0011, 29: 0.0026})

In [ ]: gbdtModel.toDebugString

In [ ]: predictions = gbdtModel.transform(testData)
        predictions.select('label', 'labelIndexed', 'prediction').show()
```

Gradient Boosted Decision Trees

- The model is ready, and we can explore featureImportances. We can even visualize the sample of decision trees, and it is quite large and long.
- It is not interpretable by a human expert.

```
9, 13: 0.0077, 14: 0.0032, 15: 0.0088, 16: 0.0057, 17: 0.0007, 18: 0.0034, 19:
 0.0023, 20: 0.1718, 21: 0.0266, 22: 0.4246, 23: 0.0839, 24: 0.0214, 25: 0.012
6, 26: 0.0476, 27: 0.0419, 28: 0.0011, 29: 0.0026})
```

```
In [31]: gbdtModel.toDebugString
```

```
Out[31]: u'GBTClassificationModel (uid=GBTClassifier_4a02b2ac733bb03672df) with 100 tre
es\n  Tree 0 (weight 1.0):\n    If (feature 22 <= 114.2)\n     If (feature 27
 <= 0.1613)\n      If (feature 20 <= 16.57)\n       If (feature 27 <= 0.1258)
\n        If (feature 10 <= 0.9289)\n         Predict: -0.9817351598173516\n
      Else (feature 10 > 0.9289)\n         Predict: 1.0\n       Else (feature
 27 > 0.1258)\n        If (feature 21 <= 32.85)\n         Predict: -0.78947368
42105263\n        Else (feature 21 > 32.85)\n         Predict: 1.0\n      Else
(feature 20 > 16.57)\n       If (feature 1 <= 16.54)\n        Predict: -1.0\n
      Else (feature 1 > 16.54)\n        If (feature 24 <= 0.1084)\n         Pr
edict: -1.0\n        Else (feature 24 > 0.1084)\n         Predict: 1.0\n     E
lse (feature 27 > 0.1613)\n      If (feature 13 <= 17.67)\n       If (feature
 1 <= 17.53)\n        Predict: -1.0\n       Else (feature 1 > 17.53)\n
 Predict: 1.0\n      Else (feature 13 > 17.67)\n       Predict: 1.0\n    Else
 (feature 22 > 114.2)\n     If (feature 26 <= 0.1904)\n      If (feature 1 <=
 19.63)\n       Predict: -1.0\n      Else (feature 1 > 19.63)\n       Predict:
1.0\n     Else (feature 26 > 0.1904)\n      If (feature 6 <= 0.05539)\n
 If (feature 1 <= 14.93)\n        Predict: -1.0\n       Else (feature 1 > 14.9
3)\n        Predict: 1.0\n      Else (feature 6 > 0.05539)\n       Predict: 1.
0\n  Tree 1 (weight 0.1):\n    If (feature 22 <= 102.3)\n     If (feature 27 <
```

Gradient Boosted Decision Trees

- Now you are doing predictions at testing data. And finally, we evaluate the accuracy.

```
 19.63)\n       Predict: -1.0\n      Else (feature 1 > 19.63)\n       Predict:
1.0\n     Else (feature 26 > 0.1904)\n      If (feature 6 <= 0.05539)\n
 If (feature 1 <= 14.93)\n        Predict: -1.0\n       Else (feature 1 > 14.9
3)\n        Predict: 1.0\n      Else (feature 6 > 0.05539)\n       Predict: 1.
0\n  Tree 1 (weight 0.1):\n    If (feature 22 <= 102.3)\n     If (feature 27 <
= 0.1825)\n      If (feature 21 <= 32.85)\n      If (feature 12 <= 5.383)\n
```

In [32]:
```
predictions = gbdtModel.transform(testData)
predictions.select('label', 'labelIndexed', 'prediction').show()
```

```
+-----+------------+----------+
|label|labelIndexed|prediction|
+-----+------------+----------+
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
|    B|         0.0|       0.0|
```

Gradient Boosted Decision Trees

- In this case, accuracy is a bit lower than the one of the single decision tree.

```
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
|    B|          0.0|          0.0|
+-----+-------------+-------------+
only showing top 20 rows
```

```
In [33]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
         accuracy = evaluator.evaluate(predictions)

         print("Test Error = %g" % (1.0 - accuracy))
```

```
Test Error = 0.0451977
```

Random Forest

- We are importing the classes which are required for evaluating random forest.
- We are creating an object, and we are fitting this method.

```
In [34]: from pyspark.ml.classification import RandomForestClassifier, RandomForestClassif

In [35]: rfClassifer = RandomForestClassifier(labelCol = "labelIndexed", numTrees = 100)

In [ ]: rfModel = rfClassifer.fit(trainingData)

In [ ]: rfModel.featureImportances

In [ ]: rfModel.toDebugString

In [ ]: predictions = rfModel.transform(testData)
        predictions.select('label', 'labelIndexed', 'prediction').show()

In [ ]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
        accuracy = evaluator.evaluate(predictions)

        print("Test Error = %g" % (1.0 - accuracy))
```

Random Forest

- Here are featureImportances, and here is an example. Again, it is quite large.

```
In [34]: from pyspark.ml.classification import RandomForestClassifier, RandomForestClassif

In [35]: rfClassifer = RandomForestClassifier(labelCol = "labelIndexed", numTrees = 100)

In [36]: rfModel = rfClassifer.fit(trainingData)

In [37]: rfModel.featureImportances

Out[37]: SparseVector(30, {0: 0.0466, 1: 0.0242, 2: 0.0748, 3: 0.0609, 4: 0.0037, 5: 0.0
         044, 6: 0.0464, 7: 0.0939, 8: 0.0052, 9: 0.0042, 10: 0.0097, 11: 0.004, 12: 0.0
         078, 13: 0.0239, 14: 0.0033, 15: 0.0029, 16: 0.0047, 17: 0.003, 18: 0.0026, 19:
         0.0055, 20: 0.1514, 21: 0.0178, 22: 0.1206, 23: 0.1174, 24: 0.0101, 25: 0.0156,
         26: 0.0256, 27: 0.0891, 28: 0.0123, 29: 0.0087})

In [ ]: rfModel.toDebugString

In [ ]: predictions = rfModel.transform(testData)
        predictions.select('label', 'labelIndexed', 'prediction').show()
```

Random Forest

- Here are the quality of our model predictions and testing accuracy.

```
In [38]: rfModel.toDebugString
```

```
Else (feature 3 > 666.0)\n    If (feature 26 <= 0.2264)\n     If (feature
15 <= 0.01443)\n      If (feature 19 <= 0.001344)\n       Predict: 0.0\n
Else (feature 19 > 0.001344)\n      If (feature 16 <= 0.01843)\n
Predict: 1.0\n       Else (feature 16 > 0.01843)\n       Predict: 0.0\n
Else (feature 15 > 0.01443)\n     If (feature 16 <= 0.03354)\n      Pre
dict: 0.0\n      Else (feature 16 > 0.03354)\n       Predict: 1.0\n    Else
(feature 26 > 0.2264)\n     If (feature 20 <= 16.57)\n      If (feature 17
<= 0.0156)\n       Predict: 0.0\n      Else (feature 17 > 0.0156)\n
Predict: 1.0\n     Else (feature 20 > 16.57)\n      If (feature 10 <= 0.318
6)\n       If (feature 7 <= 0.05074)\n       Predict: 0.0\n       Else (fe
ature 7 > 0.05074)\n       Predict: 1.0\n      Else (feature 10 > 0.3186)\n
       Predict: 1.0\n  Tree 99 (weight 1.0):\n    If (feature 2 <= 100.0)\n
If (feature 27 <= 0.1613)\n     If (feature 0 <= 13.16)\n      If (featur
e 20 <= 14.42)\n       Predict: 0.0\n      Else (feature 20 > 14.42)\n
If (feature 7 <= 0.02799)\n       Predict: 1.0\n      Else (feature 7 >
0.02799)\n       Predict: 0.0\n     Else (feature 0 > 13.16)\n      If (f
eature 26 <= 0.3438)\n       If (feature 1 <= 28.21)\n       Predict: 0.0\n
       Else (feature 1 > 28.21)\n       Predict: 1.0\n      Else (feature
26 > 0.3438)\n       If (feature 16 <= 0.05051)\n       Predict: 1.0\n
```

```
In [ ]: predictions = rfModel.transform(testData)
        predictions.select('label', 'labelIndexed', 'prediction').show()
```

Random Forest

- We can see in this example, testing accuracy of random forest was the best. But with the other dataset, the situation may be quite different.
- And as a general rule, the bigger your dataset is, the more features it has, then the quality of complex algorithms like gradient boosted decision trees or random forest will be better.

```
|    B|       0.0|       0.0|
|    B|       0.0|       0.0|
|    B|       0.0|       0.0|
|    B|       0.0|       0.0|
|    B|       0.0|       0.0|
|    B|       0.0|       0.0|
|    B|       0.0|       0.0|
|    B|       0.0|       0.0|
|    B|       0.0|       0.0|
+-----+----------+----------+
only showing top 20 rows
```

```
In [40]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti
         accuracy = evaluator.evaluate(predictions)

         print("Test Error = %g" % (1.0 - accuracy))

         Test Error = 0.0338983

In [ ]:
```

Spark ML, Cross Validation

Cross Validation

- Cross-validation helps to assess the quality of a machine learning model and to find the best model among a family of models.
- First of all, we start SparkContext.

```
In [1]: from pyspark import SparkContext
        from pyspark.sql import SparkSession

In [2]: sc = SparkContext(appName = "module3_week4")

In [3]: ! echo $PYSPARK_SUBMIT_ARGS

        --deploy-mode client --master local[2] --executor-memory 512m --driver-memory 5
        12m --executor-cores 1 --num-executors 2 --conf spark.driver.maxResultSize=256m
        pyspark-shell

In [4]: spark = SparkSession.builder().getOrCreate() # required for dataframes
```

Cross Validation

We use the same dataset which we use for evaluating different decision trees.

In [ ]:
```
!wget https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wis
```

In [ ]:
```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import StringIndexer
```

In [ ]:
```
# Load a text file and convert each line to a Row.

data = []

with open("wdbc.data") as infile:
    for line in infile:
        tokens = line.rstrip("\n").split(",")
        y = tokens[1]
        features = Vectors.dense([float(x) for x in tokens[2:]])

        data.append((y, features))
```

In [ ]:
```
inputDF = spark.createDataFrame(data, ["label", "features"])
```

In [ ]:
```
stringIndexer = StringIndexer(inputCol = "label", outputCol = "labelIndexed")
si_model = stringIndexer.fit(inputDF)
```

Cross Validation

- Finally we have the dataset which is called inputDF2.

```
data = []

with open("wdbc.data") as infile:
    for line in infile:
        tokens = line.rstrip("\n").split(",")
        y = tokens[1]
        features = Vectors.dense([float(x) for x in tokens[2:]])

        data.append((y, features))
```

In [ ]:
```
inputDF = spark.createDataFrame(data, ["label", "features"])
```

In [ ]:
```
stringIndexer = StringIndexer(inputCol = "label", outputCol = "labelIndexed")
si_model = stringIndexer.fit(inputDF)

inputDF2 = si_model.transform(inputDF)
```

In [ ]:
```
inputDF2.show()
```

Cross Validation

- Here is the content for inputDF2.

```
In [10]: inputDF2.show()
```

```
+-----+--------------------+------------+
|label|            features|labelIndexed|
+-----+--------------------+------------+
|    M|[17.99,10.38,122....|         1.0|
|    M|[20.57,17.77,132....|         1.0|
|    M|[19.69,21.25,130....|         1.0|
|    M|[11.42,20.38,77.5...|         1.0|
|    M|[20.29,14.34,135....|         1.0|
|    M|[12.45,15.7,82.57...|         1.0|
|    M|[18.25,19.98,119....|         1.0|
|    M|[13.71,20.83,90.2...|         1.0|
|    M|[13.0,21.82,87.5,...|         1.0|
|    M|[12.46,24.04,83.9...|         1.0|
|    M|[16.02,23.24,102....|         1.0|
|    M|[15.78,17.89,103....|         1.0|
|    M|[19.17,24.8,132.4...|         1.0|
|    M|[15.85,23.95,103....|         1.0|
|    M|[13.73,22.61,93.6...|         1.0|
|    M|[14.54,27.54,96.7...|         1.0|
|    M|[14.68,20.13,94.7...|         1.0|
|    M|[16.13,20.68,108....|         1.0|
|    M|[19.81,22.15,130....|         1.0|
|    B|[13.54,14.36,87.4...|         0.0|
+-----+--------------------+------------+
```

Cross Validation

- What steps are required for doing cross-validation with this dataset? For example, we want to select the best parameters of a single decision tree. We create an object decision tree then we should create a pipeline.

Cross-Validation

```
In [ ]: from pyspark.ml.classification import DecisionTreeClassifier

In [ ]: decisionTree = DecisionTreeClassifier(labelCol = "labelIndexed")

In [ ]:

In [ ]: from pyspark.ml import Pipeline

In [ ]: pipeline = Pipeline(stages = [decisionTree])

In [ ]:

In [ ]: from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator

In [ ]: paramGrid = ParamGridBuilder()\
            .addGrid(decisionTree.maxDepth, [1, 2, 4, 5, 6, 7, 8])\
            .build()
```

Cross Validation

- Pipeline, in general, may contain many stages including feature pre-processing, string indexing, and machine learning, and so on.
- But in this case, pipeline contains only one step, this training of decision tree.

Cross-Validation

```
In [ ]: from pyspark.ml.classification import DecisionTreeClassifier

In [ ]: decisionTree = DecisionTreeClassifier(labelCol = "labelIndexed")

In [ ]:

In [ ]: from pyspark.ml import Pipeline

In [ ]: pipeline = Pipeline(stages = [decisionTree])

In [ ]:

In [ ]: from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator

In [ ]: paramGrid = ParamGridBuilder()\
            .addGrid(decisionTree.maxDepth, [1, 2, 4, 5, 6, 7, 8])\
            .build()
```

Cross Validation

- Then we import their cross-validator and ParamGridBuilder class and you are creating a ParamGridBuilder class.
- For example, we want to select the best maximum depths of a decision tree in the range from 1-8.
- We have now the ParamGridBuilder class, we create an evaluator so we want to select the model which has the best accuracy among others.

```
In [15]: from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
         from pyspark.ml.evaluation import MulticlassClassificationEvaluator

In [16]: paramGrid = ParamGridBuilder()\
             .addGrid(decisionTree.maxDepth, [1, 2, 4, 5, 6, 7, 8])\
             .build()

In [ ]:  #paramGrid = ParamGridBuilder()\
         #    .addGrid(decisionTree.maxDepth, [1, 2, 4, 5, 6, 7, 8])\
         #    .addGrid(decisionTree.minInstancesPerNode, [1, 2, 4, 5, 6, 7, 8])\
         #    .build()

In [ ]:  evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti

         crossval = CrossValidator(estimator = pipeline,
                                   estimatorParamMaps = paramGrid,
                                   evaluator = evaluator,
                                   numFolds = 10)

In [ ]:  cvModel = crossval.fit(inputDF2)
```

Cross Validation

- We create an evaluator so we want to select the model which has the best accuracy among others.
- We create a cross-validator class and pass a pipeline into this class, a ParamGrid and evaluator.

```
In [ ]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti

        crossval = CrossValidator(estimator = pipeline,
                                  estimatorParamMaps = paramGrid,
                                  evaluator = evaluator,
                                  numFolds = 10)
```

```
In [ ]: cvModel = crossval.fit(inputDF2)
```

```
In [ ]: cvModel.avgMetrics
```

```
In [ ]: print cvModel.bestModel.stages[0]
```

```
In [ ]: cvModel.transform(....)
```

Cross Validation

- And finally, we select the number of folds and the number of folds should not be less than 5 or 10.

```
In [ ]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti

        crossval = CrossValidator(estimator = pipeline,
                                  estimatorParamMaps = paramGrid,
                                  evaluator = evaluator,
                                  numFolds = 10)
```

```
In [ ]: cvModel = crossval.fit(inputDF2)
```

```
In [ ]: cvModel.avgMetrics
```

```
In [ ]: print cvModel.bestModel.stages[0]
```

```
In [ ]: cvModel.transform(....)
```

Cross Validation

- We create cvModel and it takes some time because Spark needs to make training and evaluating the quality 10 times.

```
crossval = CrossValidator(estimator = pipeline,
                          estimatorParamMaps = paramGrid,
                          evaluator = evaluator,
                          numFolds = 10)
```

In [18]:
```
cvModel = crossval.fit(inputDF2)
```

In [19]:
```
cvModel.avgMetrics
```

Out[19]:
```
[0.8924563921120648,
 0.9203744192767783,
 0.9418223975919139,
 0.9419915318207598,
 0.9457320946210345,
 0.938026791703486,
 0.9361037147804091]
```

In [ ]:
```
print cvModel.bestModel.stages[0]
```

In [ ]:
```
cvModel.transform(....)
```

Cross Validation

- You can see the average accuracy, amount of folds, for each failure of decision tree depths.

```
In [ ]:

In [15]: from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
         from pyspark.ml.evaluation import MulticlassClassificationEvaluator

In [16]: paramGrid = ParamGridBuilder()\
             .addGrid(decisionTree.maxDepth, [1, 2, 4, 5, 6, 7, 8])\
             .build()

In [ ]: #paramGrid = ParamGridBuilder()\
        #    .addGrid(decisionTree.maxDepth, [1, 2, 4, 5, 6, 7, 8])\
        #    .addGrid(decisionTree.minInstancesPerNode, [1, 2, 4, 5, 6, 7, 8])\
        #    .build()

In [17]: evaluator = MulticlassClassificationEvaluator(labelCol = "labelIndexed", predicti

         crossval = CrossValidator(estimator = pipeline,
                                   estimatorParamMaps = paramGrid,
                                   evaluator = evaluator,
                                   numFolds = 10)
```

Cross Validation

- The first stage of our pipeline was a decision tree and you can get the best model and the best model has depth 6 and it has 47 nodes.

```
                                estimatorParamMaps = paramGrid,
                                evaluator = evaluator,
                                numFolds = 10)
```

In [18]:
```
cvModel = crossval.fit(inputDF2)
```

In [19]:
```
cvModel.avgMetrics
```

Out[19]:
```
[0.8924563921120648,
 0.9203744192767783,
 0.9418223975919139,
 0.9419915318207598,
 0.9457320946210345,
 0.938026791703486,
 0.9361037147804091]
```

In [20]:
```
print cvModel.bestModel.stages[0]
```

```
DecisionTreeClassificationModel (uid=DecisionTreeClassifier_4bceb658f839baff82f
f) of depth 6 with 47 nodes
```

In [ ]:
```
cvModel.transform(....)
```

Cross Validation

- Then we can view this model to make predictions at any other dataset. In ParamGridBuilder, we can use several parameters. For example, maximum depths and some other parameters of a decision tree, for example, minInstancesPerNode and select some other grid here, but in this simple example, we did not do it for the simplicity. And if we evaluate only one parameter, the training is much faster.

```
In [13]: from pyspark.ml import Pipeline

In [14]: pipeline = Pipeline(stages = [decisionTree])

In [ ]:

In [15]: from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
         from pyspark.ml.evaluation import MulticlassClassificationEvaluator

In [16]: paramGrid = ParamGridBuilder()\
             .addGrid(decisionTree.maxDepth, [1, 2, 4, 5, 6, 7, 8])\
             .build()

In [ ]: #paramGrid = ParamGridBuilder()\
        #     .addGrid(decisionTree.maxDepth, [1, 2, 4, 5, 6, 7, 8])\
        #     .addGrid(decisionTree.minInstancesPerNode, [1, 2, 4, 5, 6, 7, 8])\
        #     .build()
```

Conclusion

- In this lecture, we have discussed the concepts of Random Forest, Gradient Boosted Decision Trees and a Case Study with Spark ML Programming, Decision Trees and Ensembles.