Big data: architectures and data analytics

Spark Mllib - Classification

Classification algorithms

- All the available classification algorithms are based on two phases
  - **Model generation** based on a set of **training data**
  - **Prediction** of the **class label** of new **unlabeled data**
- All the classification algorithms available in Spark work **only** on **numerical attributes**
  - Categorical values must be mapped to numerical values (integer, and then double) before applying the MLlib classification algorithms

Classification algorithms

- All the Spark classification algorithms are built on top of an input DataFrame containing (at least) two columns
  - label
    - The class label, i.e., the attribute to be predicted by the classification model
      - It is an integer value (casted to a double)
  - features
    - A vector of doubles containing the values of the predictive attributes of the input records/data points
      - The data type of this column is **pyspark.ml.linalg.Vectors**

Categorical class labels

- The file containing the **unlabeled data** has the same format of the training data file
  - However, the **label column** is **empty** because the class label is **unknown**
- We want to predict the class label value of each unlabeled data by applying the classification model that has been inferred on the training data

Algorithms

- Decision trees
- Random forests
- Neural Networks (Multilayer perceptron)
- Naïve Bayes
- Linear Support Vector Machines
- …

Algorithms

- Decision trees
- Random forests
- Neural Networks (Multilayer perceptron)
- Naïve Bayes
- Linear Support Vector Machines
- …

These algos are shown in the slides

Decision trees

Credits:

Tan, Steinbach, Kumar, Introduction to Data Mining, McGraw Hill 2006

Decision tree classifier

- Decision trees are models that uses a tree-like model of decisions and their possible outcomes/classes
- Only contains conditional control statements
- Commonly used in operations research and machine learning
- A decision tree can be represented as a flowchart-like structure in which:
  - each internal node represents a "test" on an attribute
  - each branch represents the outcome of the test
  - each leaf node represents a class label

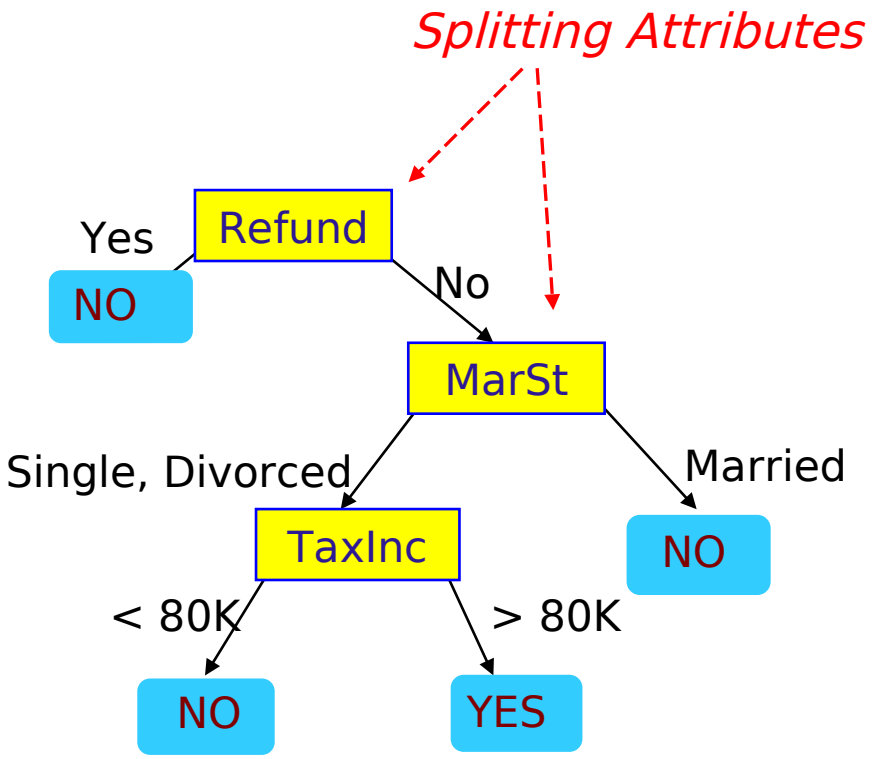
Decision tree classifier

- The model is inferred by analyzing the training data, i.e., the example records/data points for which the value of the class label is known
- Apply the model → predict the value of the class label of new unlabeled records

Example of decision tree

Tid	Refund	Marital Status	Taxable Income	Cheat
	categorical	categorical	continuous	class
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Training Data

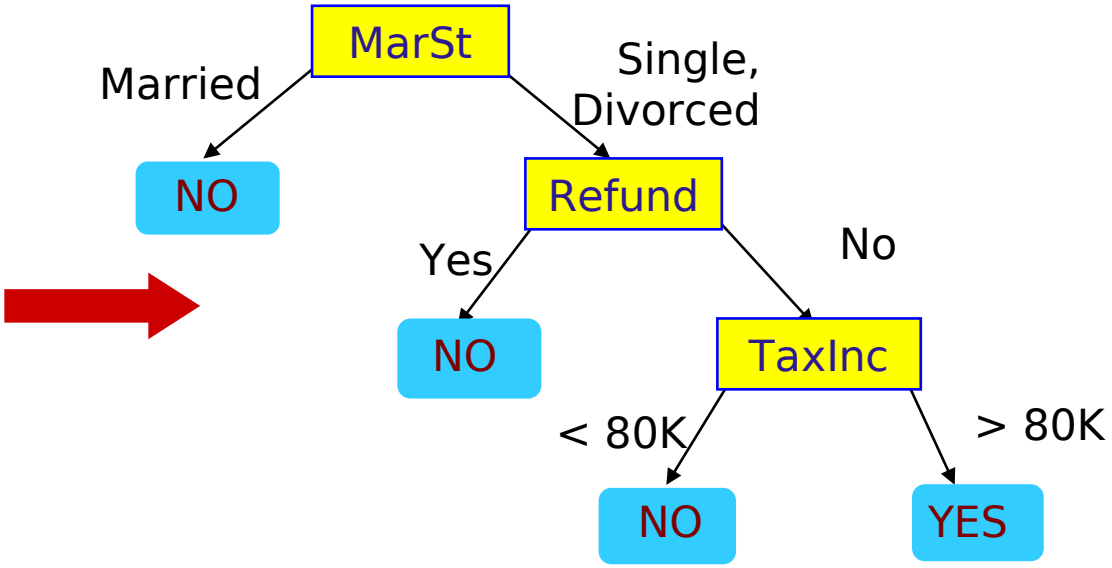


Model: Decision Tree

Another example of decision tree

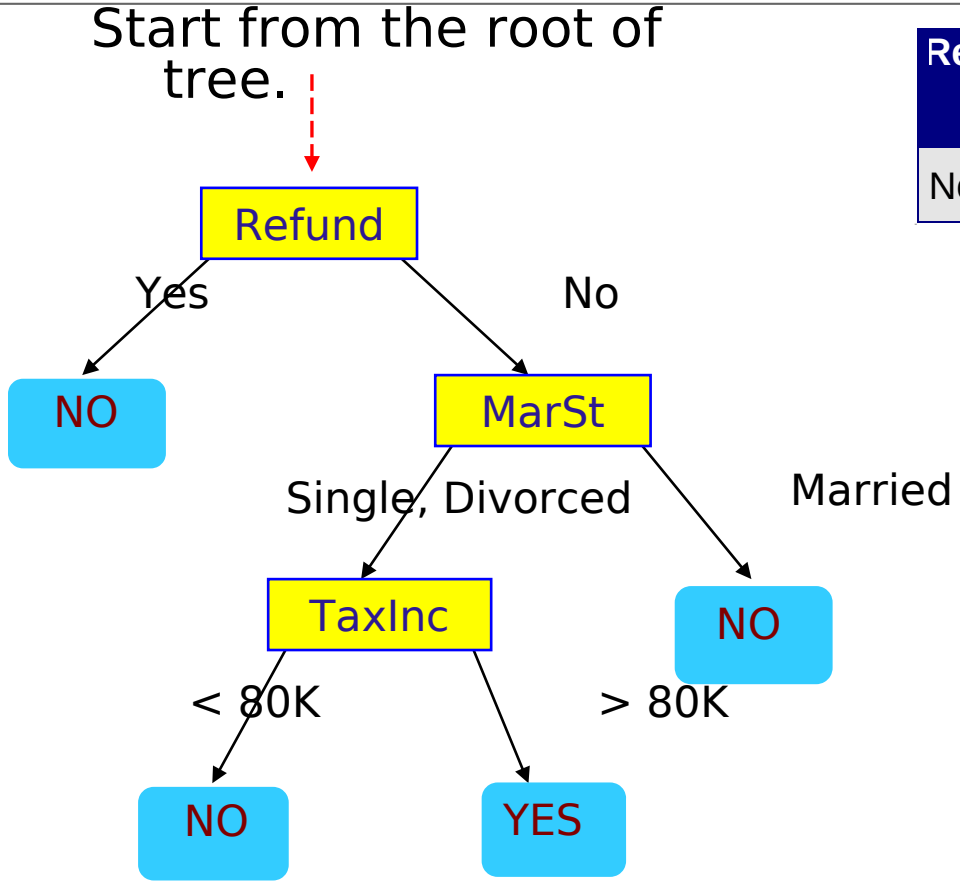
Tid	Refund	Marital Status	Taxable Income	Cheat
	categorical	categorical	continuous	class
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Training Data



There could be more than one tree that fits the same data!

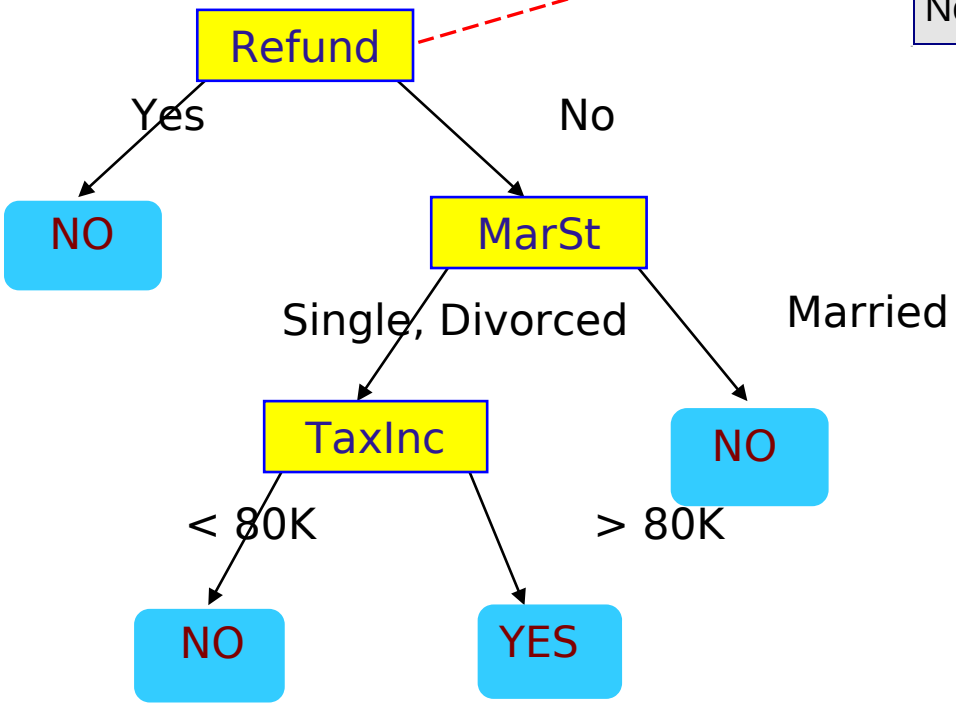
Apply Model to Test Data



Refund	Marital Status	Taxable Income	Cheat
No	Married	80K	?

Test Data

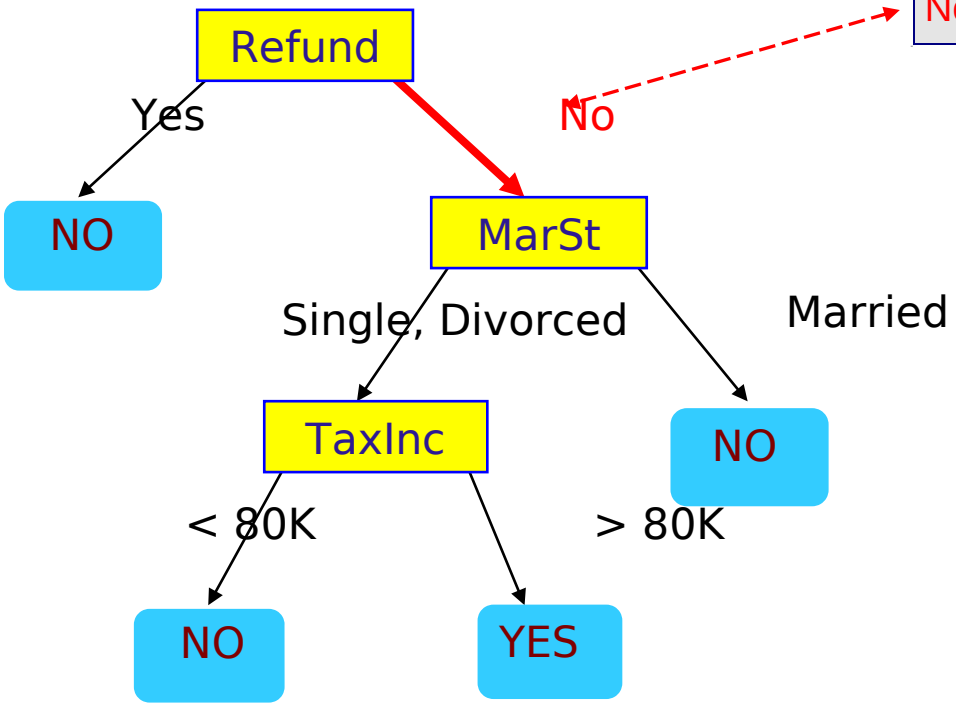
Apply Model to Test Data



Refund	Marital Status	Taxable Income	Cheat
No	Married	80K	?

Test Data

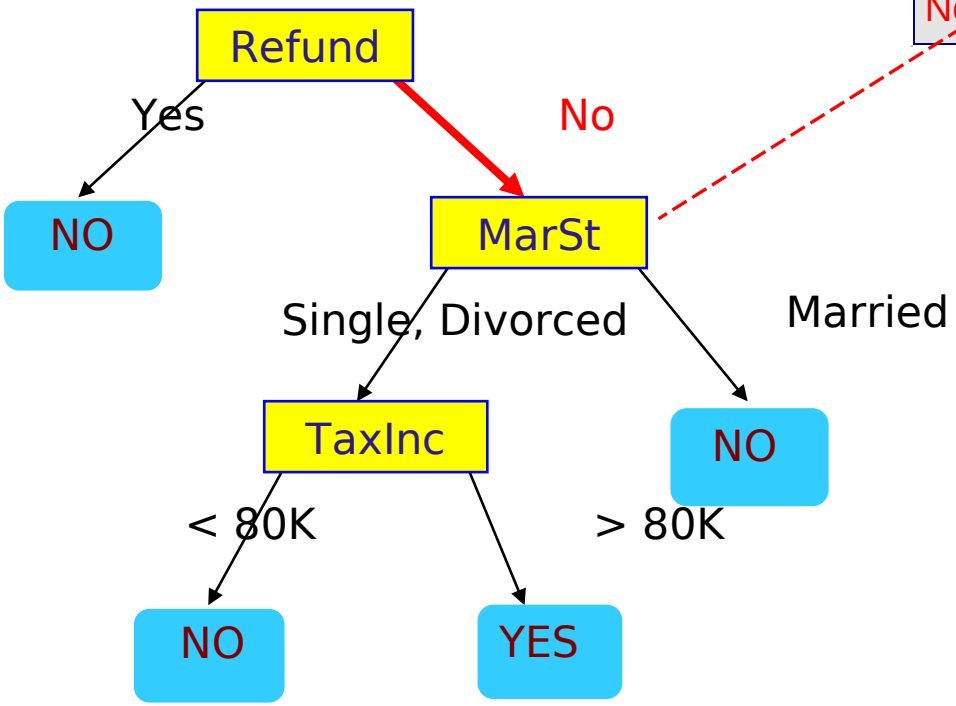
Apply Model to Test Data



Refund	Marital Status	Taxable Income	Cheat
No	Married	80K	?

Test Data

Apply Model to Test Data



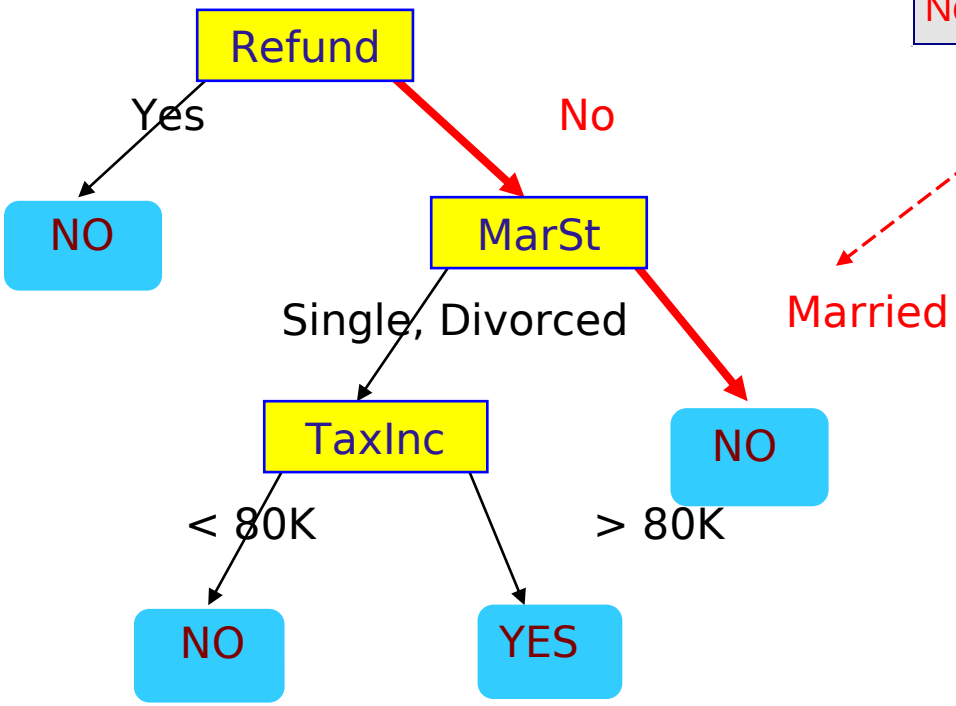
Refund	Marital Status	Taxable Income	Cheat
No	Married	80K	?

Test Data

Apply Model to Test Data

Refund	Marital Status	Taxable Income	Cheat
No	Married	80K	?

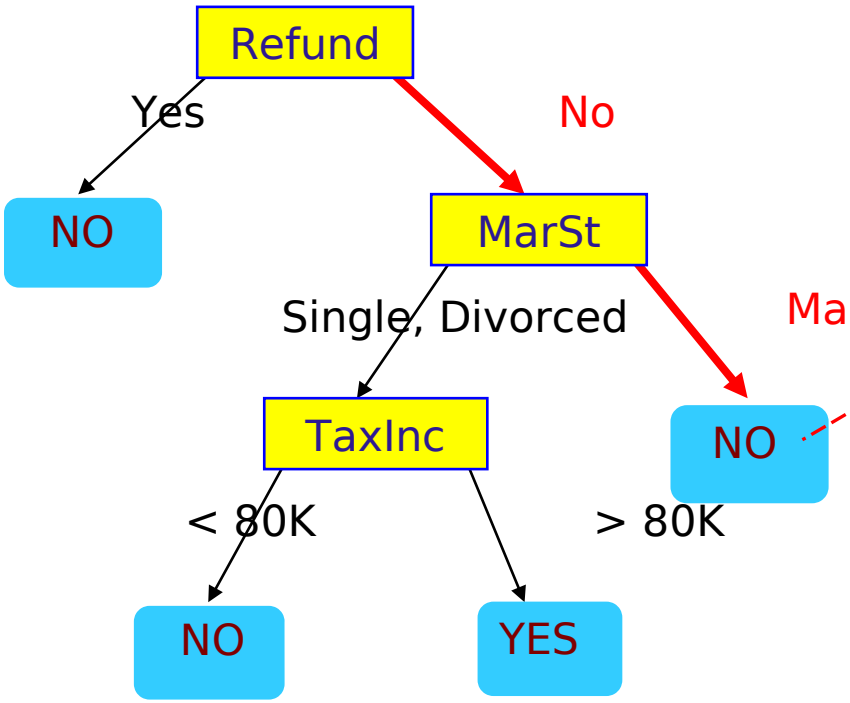
Test Data



Apply Model to Test Data

Refund	Marital Status	Taxable Income	Cheat
No	Married	80K	?

Test Data



Assign Cheat to "No"

Decision tree induction

- Many algorithms to build a decision tree
  - Hunt’s Algorithm
  - CART
  - ID3, C4.5, C5.0
  - ...

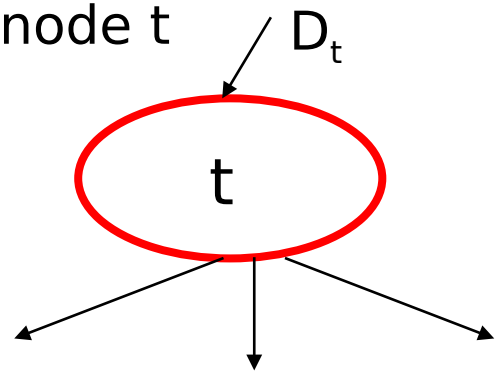
Decision tree induction

Basic steps

- If D_t contains records that belong to more than one class
  - select the “best” attribute A on which to split D_t and label node t as A
  - split D_t into smaller subsets and recursively apply the procedure to each subset
- If D_t contains records that belong to the same class y_t
  - then t is a leaf node labeled as y_t
- If D_t is an empty set
  - then t is a leaf node labeled as the default (majority) class, y_d

Tid	Refund	Marital Status	Taxable Income	Cheat
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

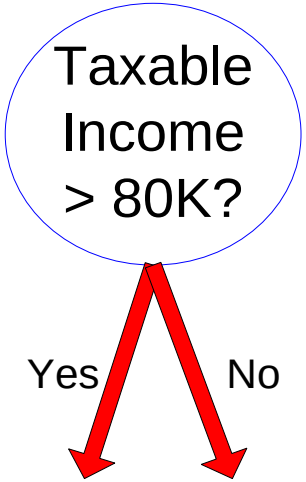
D_t, set of training records that reach a node t



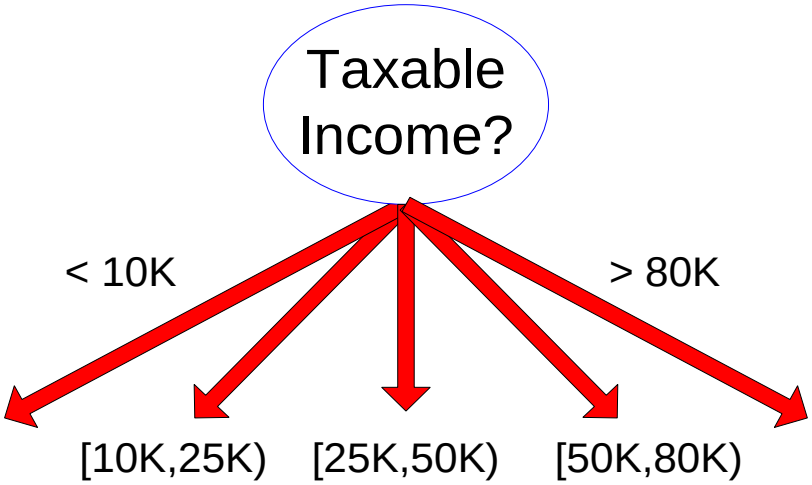
Decision tree induction

- Adopts a greedy strategy
  - “Best” attribute for the split is selected locally at each step
    - not a global optimum
- Issues
  - Structure of test condition
    - Binary split versus multiway split
  - Selection of the best attribute for the split
  - Stopping condition for the algorithm

Splitting on continuous attributes



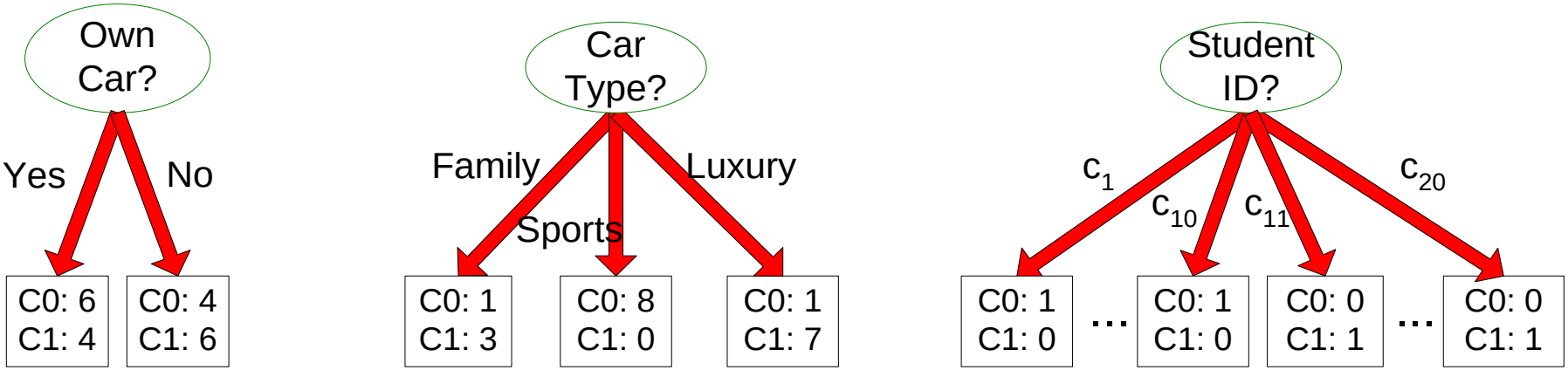
(i) Binary split



(ii) Multi-way split

Selection of the best attribute

Before splitting: 10 records of class 0,
10 records of class 1



Which attribute (test condition) is the best?

Selection of the best attribute

- Attributes with *homogeneous* class distribution are preferred
- Define measure of node impurity

C0: 5
C1: 5

Non-homogeneous, high degree of impurity

C0: 9
C1: 1

Homogeneous, low degree of impurity

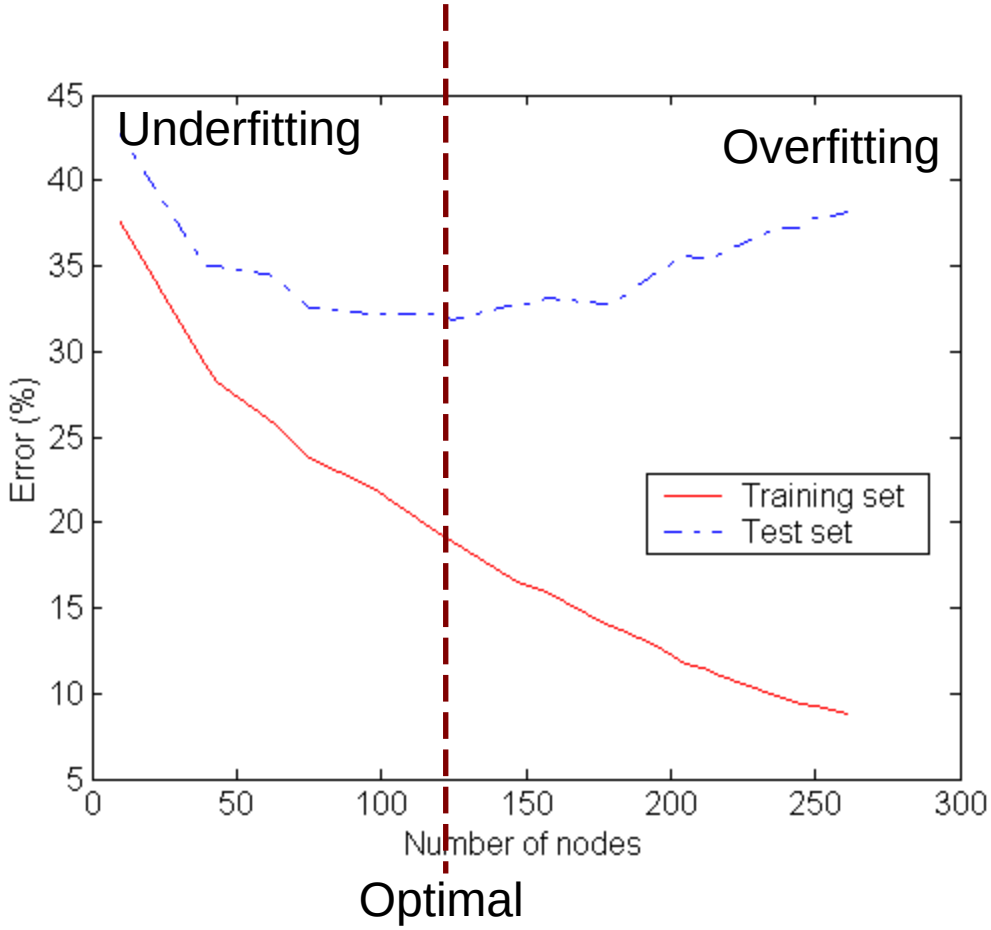
Measures of node impurity

- Many different measures available
  - Gini index
  - Entropy
  - ...
- Different algorithms rely on different measures

Stopping Criteria for Tree Induction

- Stop expanding a node when all the records belong to the same class
- Stop expanding a node when all the records have similar attribute values
- Early termination
  - Pre-pruning
  - Post-pruning

Underfitting and Overfitting



Underfitting: when model is too simple, both training and validation errors are large

How to address overfitting

Pre-Pruning (Early Stopping Rule)

- Stop the algorithm before it becomes a fully-grown tree
- Typical stopping conditions for a node
  - Stop if all instances belong to the same class
  - Stop if all the attribute values are the same
- More restrictive conditions
  - Stop if number of instances is less than some user-specified threshold
  - Stop if class distribution of instances are independent of the available features (e.g., using χ^2 test)
  - Stop if expanding the current node does not improve impurity measures (e.g., Gini or information gain)

How to address overfitting

Post-pruning

- Grow decision tree to its entirety
- Trim the nodes of the decision tree in a bottom-up fashion
- If generalization error improves after trimming, replace sub-tree by a leaf node.
- Class label of leaf node is determined from majority class of instances in the sub-tree

Decision Tree Based Classification

- Advantages
  - Inexpensive to construct
  - Extremely fast at classifying unknown records
  - Easy to interpret for small-sized trees
  - Accuracy is comparable to other classification techniques for many simple data sets

Decision trees in MLlib

- In MLlib how to:
  - Create a classification model based on the **decision tree algorithm**
    - The model is inferred by analyzing the training data, i.e., the example records/data points for which the value of the class label is known
  - Apply the model to new unlabeled data
    - The inferred model is applied to predict the value of the class label of new unlabeled records/data points

Decision trees in MLlib

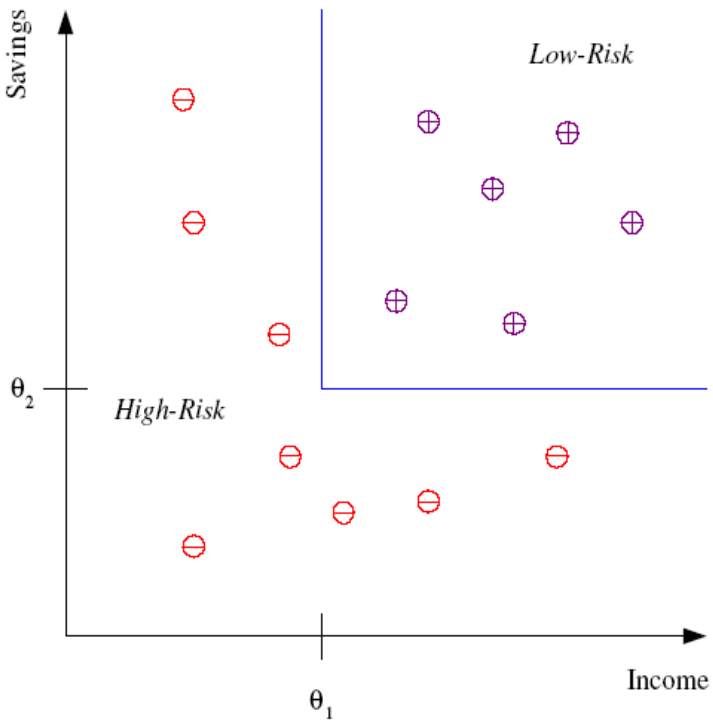
- Use the **DecisionTreeClassifier** estimator from **pyspark.ml.classification** on a DataFrame
- Explicitly specify input columns **featureCol** (vector) and **labelCol** (double)
- Output columns:
  - **predictionCol** with thePredicted label
  - **RawPredictionCol** (vector) with the counts of training instance labels at the tree node which makes the prediction
  - **probabilityCol** probability equal to rawPrediction normalized to a multinomial distribution

Decision trees in MLlib

- (Some) parameters:
  - **maxDepth**: Maximum depth of a tree. Deeper trees are more expressive (potentially allowing higher accuracy), but also more costly to train and more likely to overfit
  - **minInstancesPerNode**: for a node to be split further, each of its children must receive at least this number of training instances
  - **maxBins**: Number of bins used when discretizing continuous features. Increasing maxBins allows to make fine-grained split decisions, but increases computation and communication
  - **maxMemoryInMB**: Amount of memory to be used for collecting statistics. Increasing maxMemoryInMB can lead to faster training (if memory available) by allowing fewer passes over the data. However, amount of communication on each iteration can be proportional to maxMemoryInMB
  - **impurity:** Impurity measure (“Gini”,”Entropy”) used to choose between candidate splits

Decision trees: example

- Credit scoring
- Differentiating between low-risk and high-risk customers from their *income* and *savings*



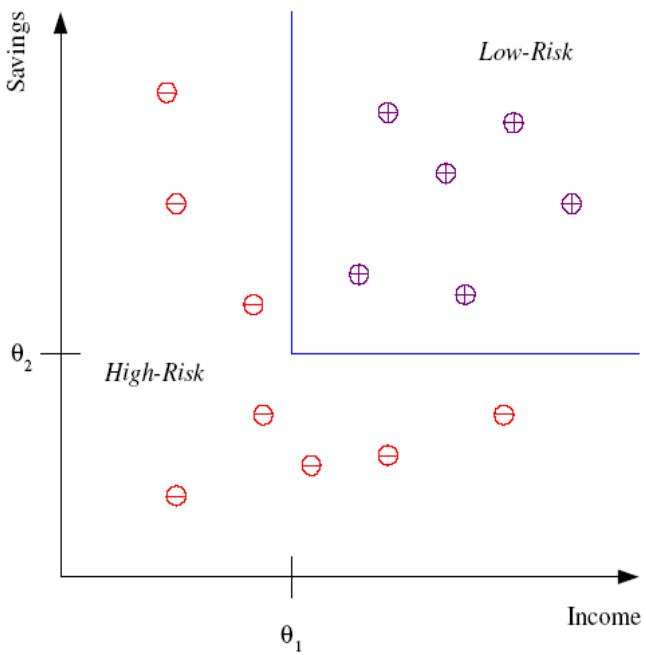
Decision trees: example

- Preprocess the data
  - label
  - features

Decision trees: example

Input DataFrame

Savings	Income	Risk
15000	1100	Low
0	5000	High
20000	800	High
6000	1300	Low
50000	2500	Low
2000	1100	Low
700	1500	High
75000	0	High
4000	500	High
7000	3000	Low
3000	900	High
6000	1200	Low



Decision trees: example

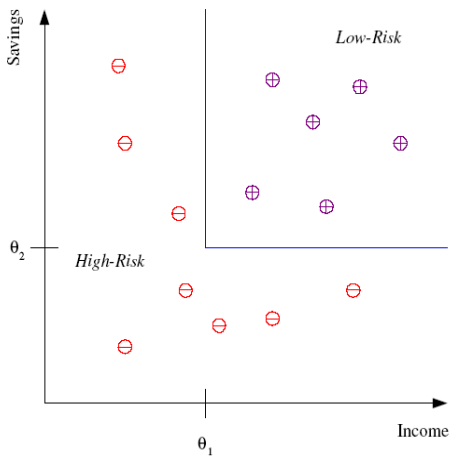
Preprocess the input dataframe:

```
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler

data=spark.read.csv('credit_score.txt',header=True,inferSchema=True)


indexer = StringIndexer(inputCol="Risk", outputCol="RiskIndex",
    handleInvalid="keep")
indexerModel = indexer.fit(data)
indexedDF=indexerModel.transform(data)


va=VectorAssembler(inputCols=["Savings","Income"],
    outputCol="features")
processedDF=va.transform(indexedDF)
```

Decision trees: example

Input DataFrame

Savings	Income	Risk
15000	1100	Low
0	5000	High
20000	800	High
6000	1300	Low
50000	2500	Low
2000	1100	Low
700	1500	High
75000	0	High
4000	500	High
7000	3000	Low
3000	900	High
6000	1200	Low



Preprocessed DataFrame

Savings	Income	Risk	RiskIndex	features
15000	1000	Low	1.0	[15000.0,1000.0]
0	5000	High	0.0	[0.0,5000.0]
20000	800	High	0.0	[20000.0,800.0]
6000	1300	Low	1.0	[6000.0,1300.0]
50000	2500	Low	1.0	[50000.0,2500.0]
2000	1100	Low	1.0	[2000.0,1100.0]
700	1500	High	0.0	[700.0,1500.0]
75000	0	High	0.0	[75000.0,0.0]
4000	500	High	0.0	[4000.0,500.0]
7000	3000	Low	1.0	[7000.0,3000.0]
3000	900	High	0.0	[3000.0,900.0]
6000	1200	Low	1.0	[6000.0,1200.0]

Decision trees: example

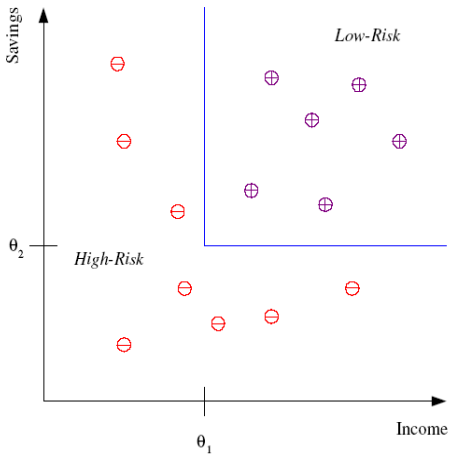
Train the decision tree:

```
from pyspark.ml.classification import DecisionTreeClassifier


# Train a DecisionTree model
dt = DecisionTreeClassifier(labelCol="RiskIndex",
    featuresCol="features")


dtModel=dt.fit(processedDF)
finalDF=dtModel.transform(processedDF)
```

Decision trees: example

Preprocessed DataFrame

Savings	Income	Risk	RiskIndex	features
15000	1000	Low	1.0	[15000.0,1000.0]
0	5000	High	0.0	[0.0,5000.0]
20000	800	High	0.0	[20000.0,800.0]
6000	1300	Low	1.0	[6000.0,1300.0]
50000	2500	Low	1.0	[50000.0,2500.0]
2000	1100	Low	1.0	[2000.0,1100.0]
700	1500	High	0.0	[700.0,1500.0]
75000	0	High	0.0	[75000.0,0.0]
4000	500	High	0.0	[4000.0,500.0]
7000	3000	Low	1.0	[7000.0,3000.0]
3000	900	High	0.0	[3000.0,900.0]
6000	1200	Low	1.0	[6000.0,1200.0]



Output DataFrame

Savings	Income	Risk	RiskIndex	features	rawPrediction	probability	prediction
15000	1000	Low	1.0	[15000.0,1000.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
0	5000	High	0.0	[0.0,5000.0]	[2.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
20000	800	High	0.0	[20000.0,800.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
6000	1300	Low	1.0	[6000.0,1300.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
50000	2500	Low	1.0	[50000.0,2500.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
2000	1100	Low	1.0	[2000.0,1100.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
700	1500	High	0.0	[700.0,1500.0]	[2.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
75000	0	High	0.0	[75000.0,0.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
4000	500	High	0.0	[4000.0,500.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
7000	3000	Low	1.0	[7000.0,3000.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
3000	900	High	0.0	[3000.0,900.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
6000	1200	Low	1.0	[6000.0,1200.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0

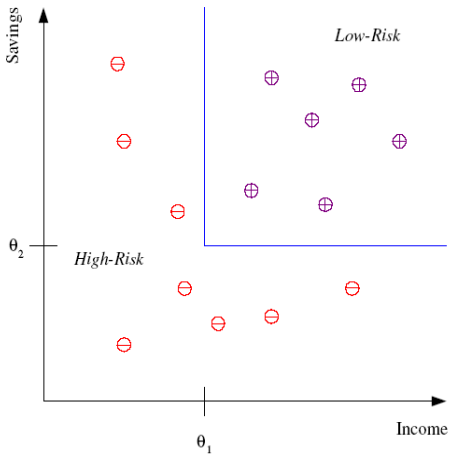
Decision trees: example

- Another file containing the **test data** has the same format of the training data file
  - The class label is not used for the prediction
  - The class label might be **unknown**
- We want to predict the class label value of each unlabeled data by applying the classification model that has been inferred on the training data

Decision trees: example

Test DataFrame

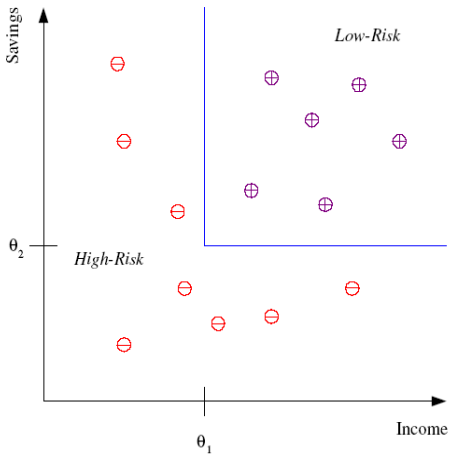
Savings	Income	Risk
100000	10000	Low
100	100	High
3100	900	High
2000	1500	High
3500	1200	Low



Decision trees: example

Test DataFrame

```
+-------+------+----+
|Savings|Income|Risk|
+-------+------+----+
| 100000| 10000| Low|
|    100|   100|High|
|   3100|   900|High|
|   2000|  1500|High|
|   3500|  1200| Low|
+-------+------+----+
```



A value can be specified also for the label column even for unlabeled data. In any case it is not used for the prediction

Decision trees: example

Test the decision tree:

```
testData=spark.read.csv('credit_score_test.txt',header=True,inferSch
    ema=True)

processedTestDF=va.transform(testData)

finalTestDF=dtModel.transform(processedTestDF)
```

Decision trees: example

Test the decision tree:

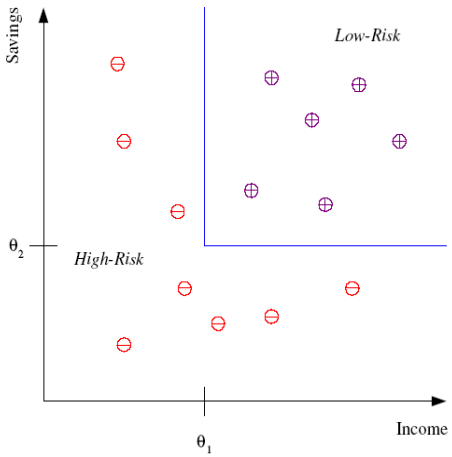
```
testData=spark.read.csv('credit_score_test.txt',header=True,inferSch
    ema=True)

processedTestDF=va.transform(testData)

finalTestDF=dtModel.transform(processedTestDF)
```

The model is applied to new data/records and the class label is predicted for each new data/record.

Decision trees: example

Test DataFrame

```
+-------+------+----+
|Savings|Income|Risk|
+-------+------+----+
| 100000| 10000| Low|
|    100|   100|High|
|   3100|   900|High|
|   2000|  1500|High|
|   3500|  1200| Low|
+-------+------+----+
```



Output Test DataFrame

```
+-------+------+----+------------------+-------------+-------------+----------+
|Savings|Income|Risk|          features|rawPrediction|  probability|prediction|
+-------+------+----+------------------+-------------+-------------+----------+
| 100000| 10000| Low|[100000.0,10000.0]|[0.0,6.0,0.0]|[0.0,1.0,0.0]|       1.0|
|    100|   100|High|     [100.0,100.0]|[4.0,0.0,0.0]|[1.0,0.0,0.0]|       0.0|
|   3100|   900|High|    [3100.0,900.0]|[4.0,0.0,0.0]|[1.0,0.0,0.0]|       0.0|
|   2000|  1500|High|   [2000.0,1500.0]|[0.0,6.0,0.0]|[0.0,1.0,0.0]|       1.0|
|   3500|  1200| Low|   [3500.0,1200.0]|[0.0,6.0,0.0]|[0.0,1.0,0.0]|       1.0|
+-------+------+----+------------------+-------------+-------------+----------+
```

Random forests

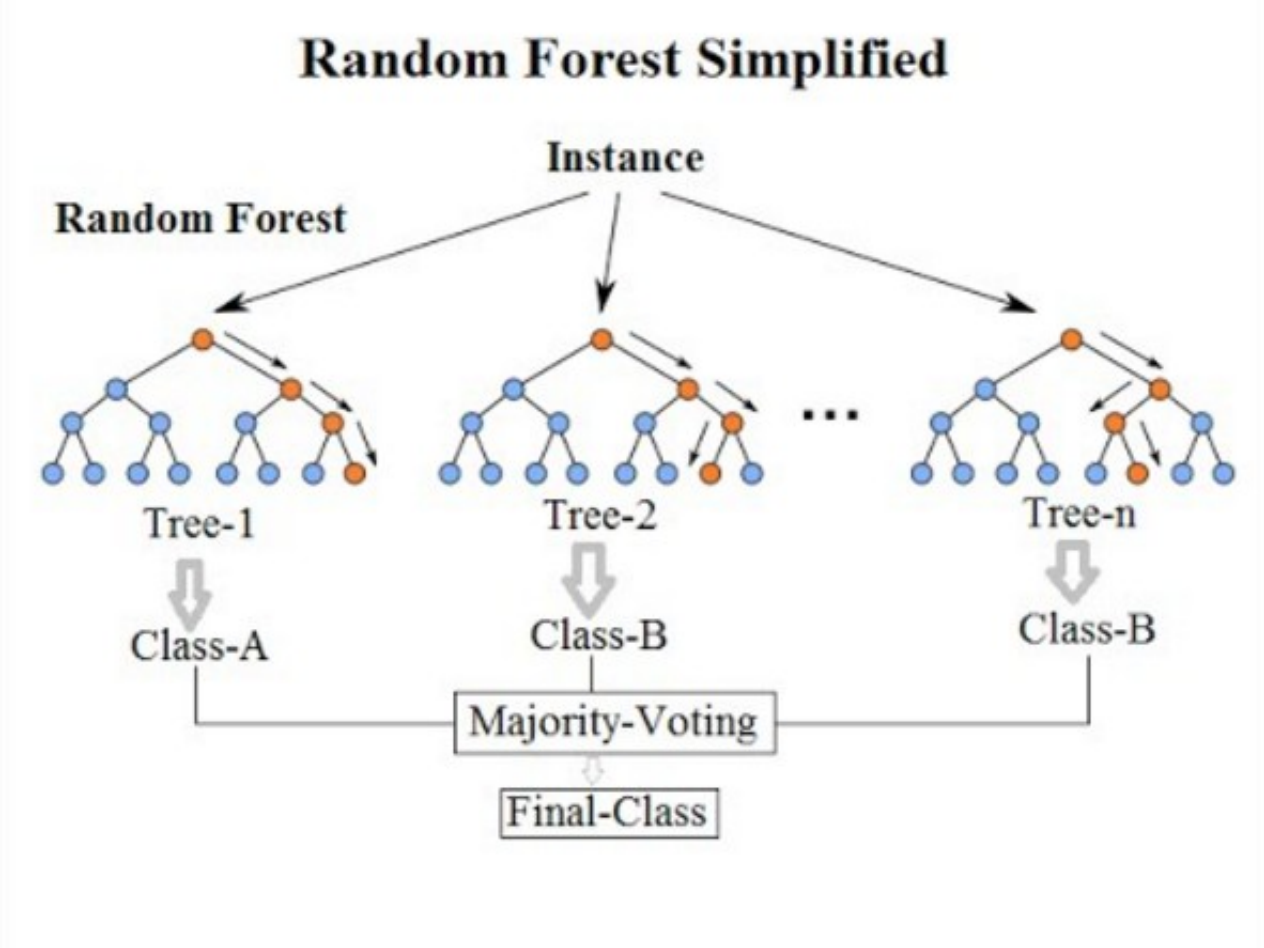
Credits:

Tan,Steinbach, Kumar, Introduction to Data Mining, McGraw Hill 2006

Random forest classifier

- Random forest is an ensemble classifier that consists of many decision trees and outputs the class according to the results of the trees
- E.g., the mode of the output classes
- For each tree of the K trees of the forest, choose a subset of m features (M is the total number of features) and a subset of n training data (N is the total number of data)

Random forest classifier



Random forest classifier

- How to select number of trees K?
  - Build trees until the error no longer decreases
- How to select number of features m?
  - Recommend defaults, half of them, proportional to data,...

Random forest classifier

- Advantages
  - It is one of the most accurate learning algorithms available
  - It runs efficiently on large databases
  - It can handle thousands of input variables without variable deletion
  - It gives estimates of what variables are important in the classification
- Disadvantages
  - Random forests have been observed to overfit for some datasets with noisy classification/regression tasks
  - For categorical variables with different number of levels, random forests are biased in favor of those attributes with more levels

Random forest in MLlib

- Use the **RandomForestClassifier** estimator from **pyspark.ml.classification** on a DataFrame
- Explicitly specify input columns **featureCol** (vector) and **labelCol** (double)
- Output columns:
  - **predictionCol** with the Predicted label
  - **RawPredictionCol** (vector) with the counts of training instance labels at the nodes of the trees which makes the prediction
  - **probabilityCol** probability equal to rawPrediction normalized to a multinomial distribution

Random forest in MLlib

- (Some) parameters:
  - Like in decision trees: **maxDepth, minInstancesPerNode, maxBins, impurity**
  - **numTrees:** Number of trees in the forest. Increasing the number of trees will decrease the variance in predictions, improving the model's test-time accuracy. Training time increases roughly linearly in the number of trees.
  - **subsamplingRate:** This parameter specifies the size of the dataset used for training each tree in the forest, as a fraction of the size of the original dataset. The default (1.0) is recommended, but decreasing this fraction can speed up training.
  - **featureSubsetStrategy:** Number of features to use as candidates for splitting at each tree node. The number is specified as a fraction or function of the total number of features. Decreasing this number will speed up training, but can sometimes impact performance if too low.

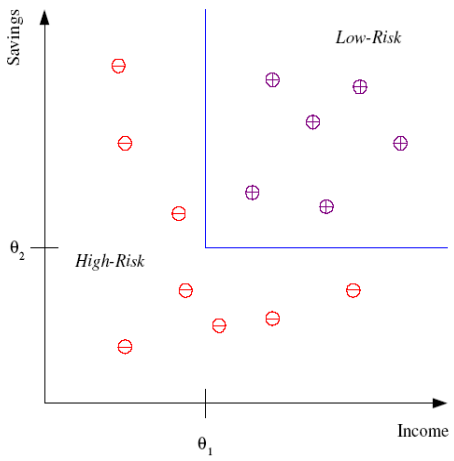
Random forest: example

- Same example as for decision tree
- Same code for preprocessing the data
  - label
  - features

Random forest: example

Preprocessed DataFrame

Savings	Income	Risk	RiskIndex	features
15000	1000	Low	1.0	[15000.0,1000.0]
0	5000	High	0.0	[0.0,5000.0]
20000	800	High	0.0	[20000.0,800.0]
6000	1300	Low	1.0	[6000.0,1300.0]
50000	2500	Low	1.0	[50000.0,2500.0]
2000	1100	Low	1.0	[2000.0,1100.0]
700	1500	High	0.0	[700.0,1500.0]
75000	0	High	0.0	[75000.0,0.0]
4000	500	High	0.0	[4000.0,500.0]
7000	3000	Low	1.0	[7000.0,3000.0]
3000	900	High	0.0	[3000.0,900.0]
6000	1200	Low	1.0	[6000.0,1200.0]



Random forest: example

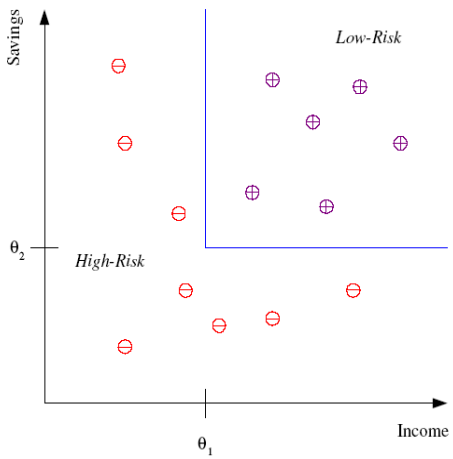
Train the random forest:

```
from pyspark.ml.classification import RandomForestClassifier

rf = RandomForestClassifier(labelCol="RiskIndex",
    featuresCol="features",numTrees=20)

rfModel=rf.fit(processedDF)
finalDF=rfModel.transform(processedDF)
```

Random forest: example

Preprocessed DataFrame

Savings	Income	Risk	RiskIndex	features
15000	1000	Low	1.0	[15000.0,1000.0]
0	5000	High	0.0	[0.0,5000.0]
20000	800	High	0.0	[20000.0,800.0]
6000	1300	Low	1.0	[6000.0,1300.0]
50000	2500	Low	1.0	[50000.0,2500.0]
2000	1100	Low	1.0	[2000.0,1100.0]
700	1500	High	0.0	[700.0,1500.0]
75000	0	High	0.0	[75000.0,0.0]
4000	500	High	0.0	[4000.0,500.0]
7000	3000	Low	1.0	[7000.0,3000.0]
3000	900	High	0.0	[3000.0,900.0]
6000	1200	Low	1.0	[6000.0,1200.0]



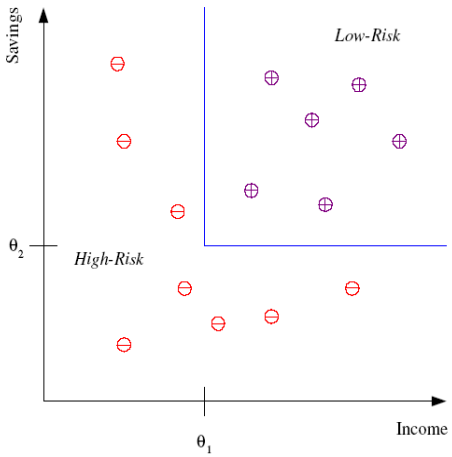
Output DataFrame

Savings	Income	Risk	RiskIndex	features	rawPrediction	probability	prediction
15000	1000	Low	1.0	[15000.0,1000.0]	[3.0,17.0,0.0]	[0.15,0.85,0.0]	1.0
0	5000	High	0.0	[0.0,5000.0]	[19.0,1.0,0.0]	[0.95,0.05,0.0]	0.0
20000	800	High	0.0	[20000.0,800.0]	[15.0,5.0,0.0]	[0.75,0.25,0.0]	0.0
6000	1300	Low	1.0	[6000.0,1300.0]	[0.0,20.0,0.0]	[0.0,1.0,0.0]	1.0
50000	2500	Low	1.0	[50000.0,2500.0]	[3.0,17.0,0.0]	[0.15,0.85,0.0]	1.0
2000	1100	Low	1.0	[2000.0,1100.0]	[6.0,14.0,0.0]	[0.3,0.7,0.0]	1.0
700	1500	High	0.0	[700.0,1500.0]	[12.0,8.0,0.0]	[0.6,0.4,0.0]	0.0
75000	0	High	0.0	[75000.0,0.0]	[15.0,5.0,0.0]	[0.75,0.25,0.0]	0.0
4000	500	High	0.0	[4000.0,500.0]	[18.0,2.0,0.0]	[0.9,0.1,0.0]	0.0
7000	3000	Low	1.0	[7000.0,3000.0]	[0.0,20.0,0.0]	[0.0,1.0,0.0]	1.0
3000	900	High	0.0	[3000.0,900.0]	[14.0,6.0,0.0]	[0.7,0.3,0.0]	0.0
6000	1200	Low	1.0	[6000.0,1200.0]	[0.0,20.0,0.0]	[0.0,1.0,0.0]	1.0

Random forest: example

Test DataFrame

```
+-------+------+----+
|Savings|Income|Risk|
+-------+------+----+
| 100000| 10000| Low|
|    100|   100|High|
|   3100|   900|High|
|   2000|  1500|High|
|   3500|  1200| Low|
+-------+------+----+
```



Random forest: example

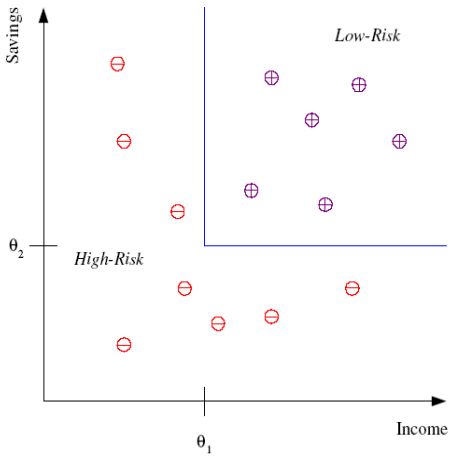
Test the random forest:

```
testData=spark.read.csv('credit_score_test.txt',header=True,inferSch
    ema=True)

processedTestDF=va.transform(testData)

finalTestDF=rfModel.transform(processedTestDF)
```

Random forest: example

Test DataFrame

```
+-------+------+----+
|Savings|Income|Risk|
+-------+------+----+
| 100000| 10000| Low|
|    100|   100|High|
|   3100|   900|High|
|   2000|  1500|High|
|   3500|  1200| Low|
+-------+------+----+
```



Output Test DataFrame

```
+-------+------+----+------------------+--------------+---------------+----------+
|Savings|Income|Risk|          features| rawPrediction|    probability|prediction|
+-------+------+----+------------------+--------------+---------------+----------+
| 100000| 10000| Low|[100000.0,10000.0]|[3.0,17.0,0.0]|[0.15,0.85,0.0]|       1.0|
|    100|   100|High|     [100.0,100.0]|[20.0,0.0,0.0]|  [1.0,0.0,0.0]|       0.0|
|   3100|   900|High|    [3100.0,900.0]|[14.0,6.0,0.0]|  [0.7,0.3,0.0]|       0.0|
|   2000|  1500|High|   [2000.0,1500.0]|[6.0,14.0,0.0]|  [0.3,0.7,0.0]|       1.0|
|   3500|  1200| Low|   [3500.0,1200.0]|[7.0,13.0,0.0]|[0.35,0.65,0.0]|       1.0|
+-------+------+----+------------------+--------------+---------------+----------+
```

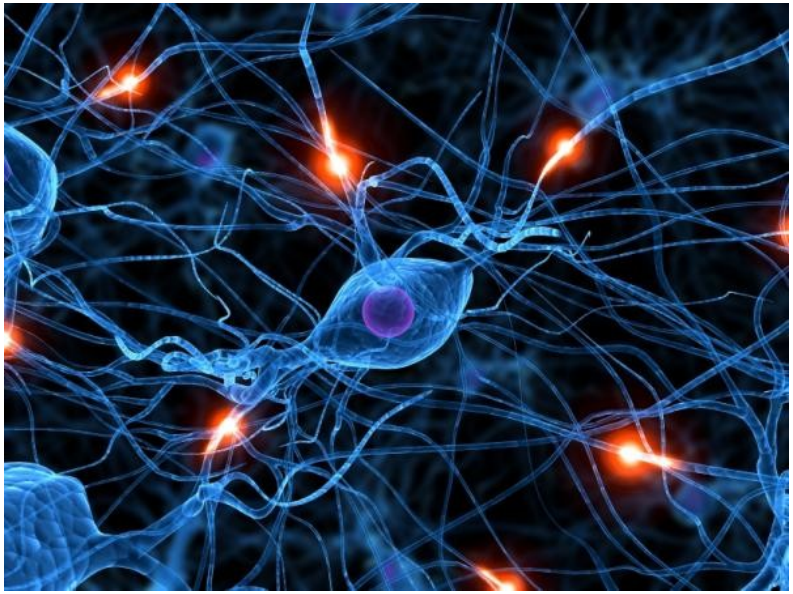
Neural networks

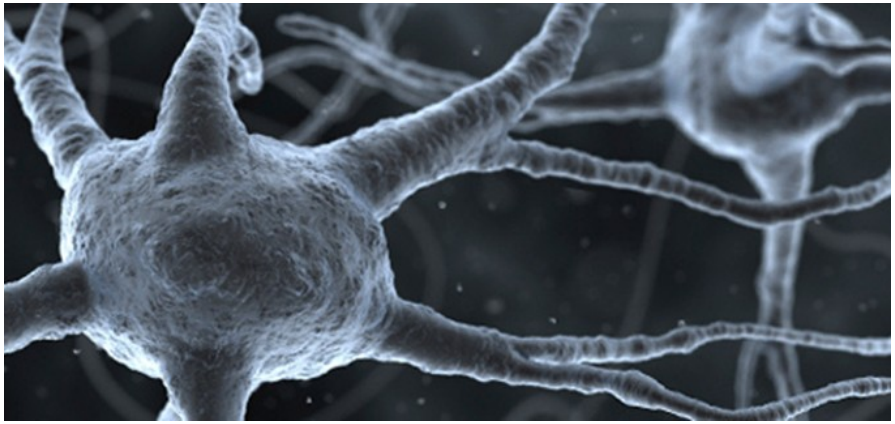
Credits:

Han, Kamber,”Data mining; Concepts and Techniques”, Morgan Kaufmann 2006

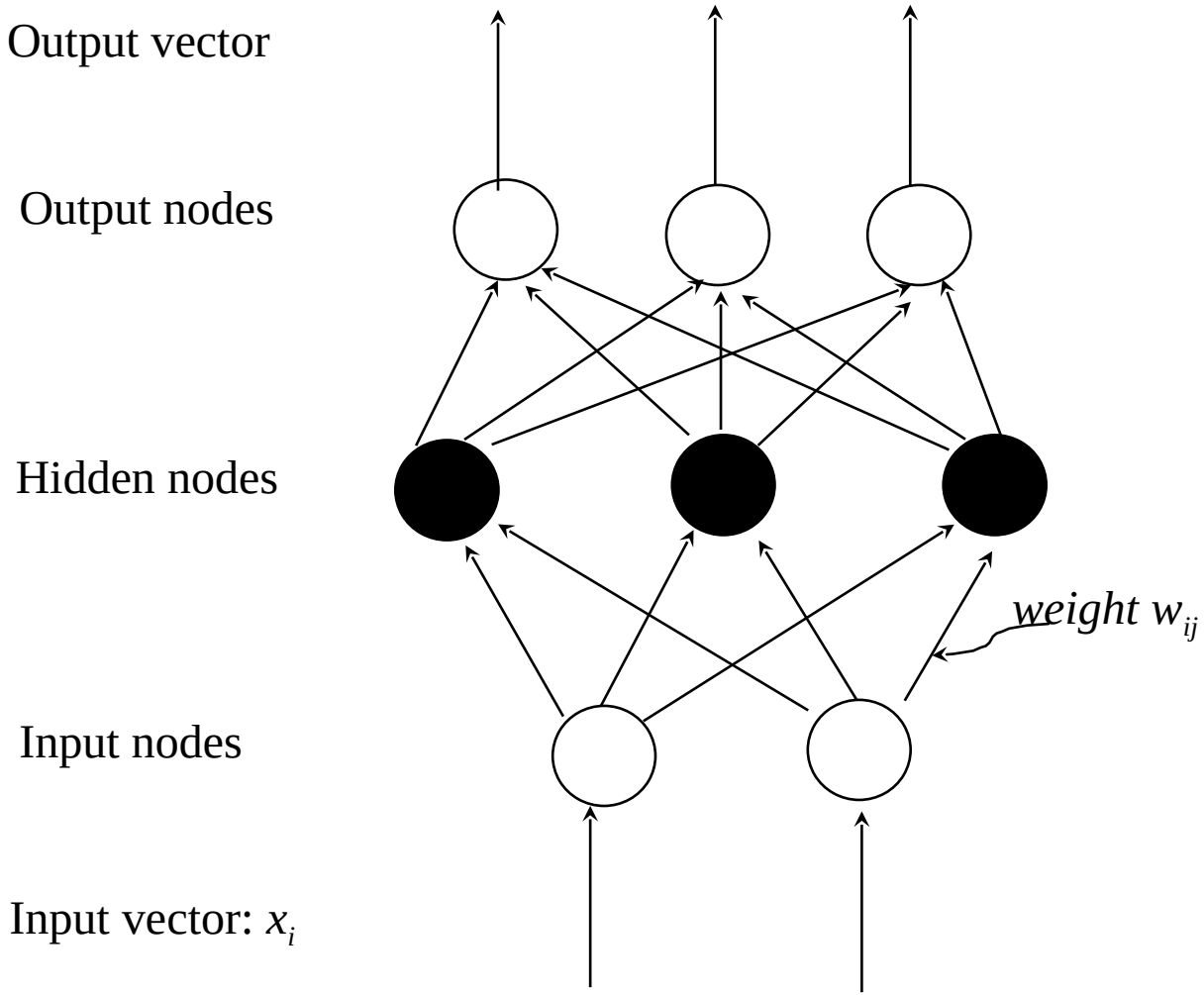
Neural networks

- Inspired to the structure of the human brain
  - Neurons as elaboration units
  - Synapses as connection network

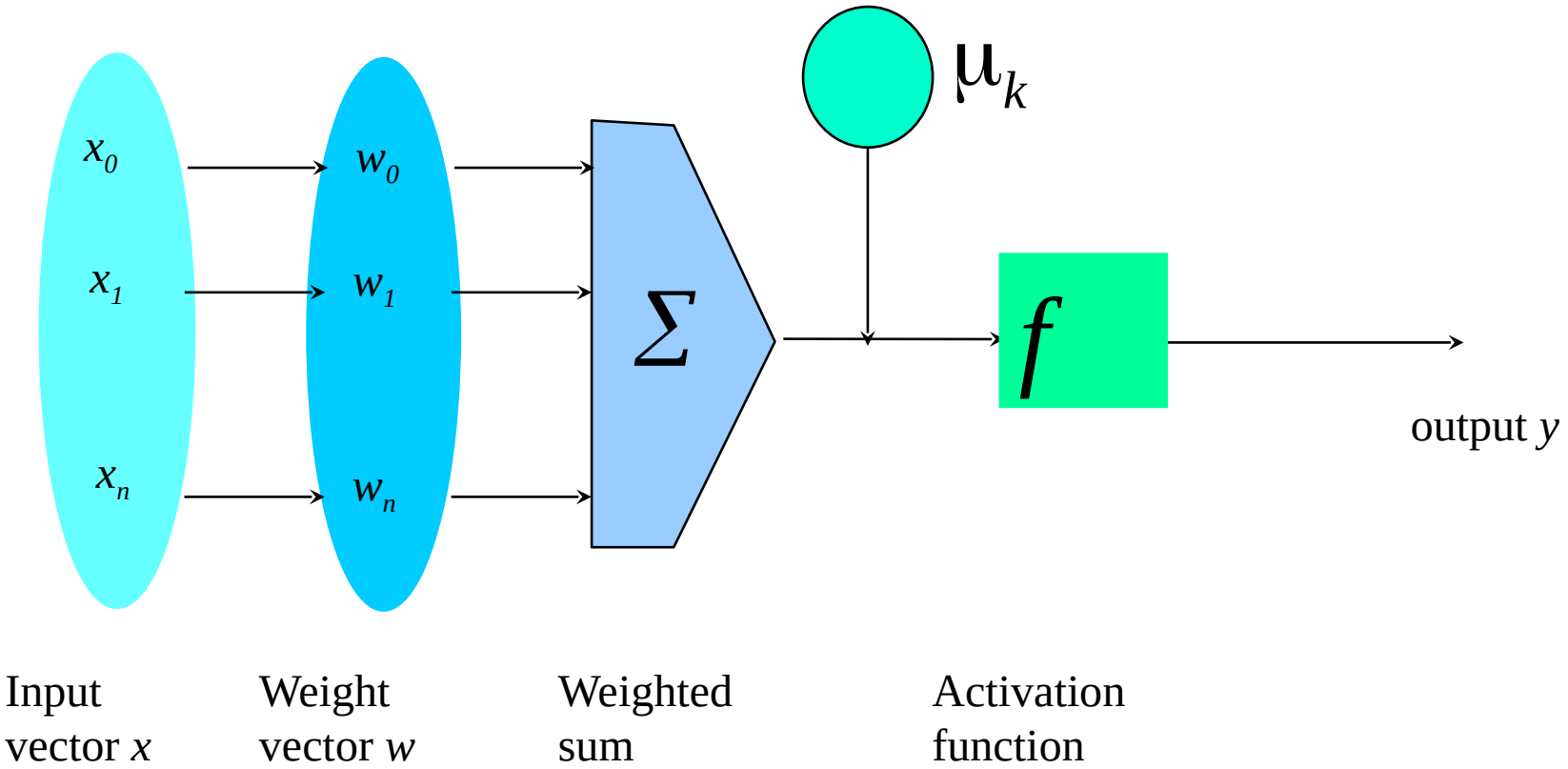




Neural networks



Neural networks



Neural networks

- For each node, definition of
  - set of weights
  - offset value

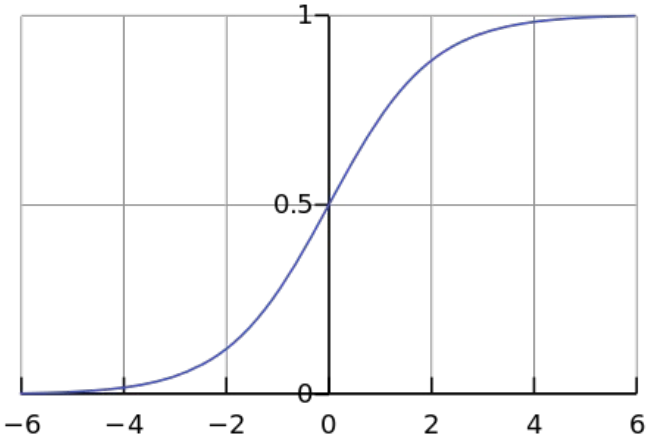
  providing the highest accuracy on the training data
- Iterative approach on training data instances

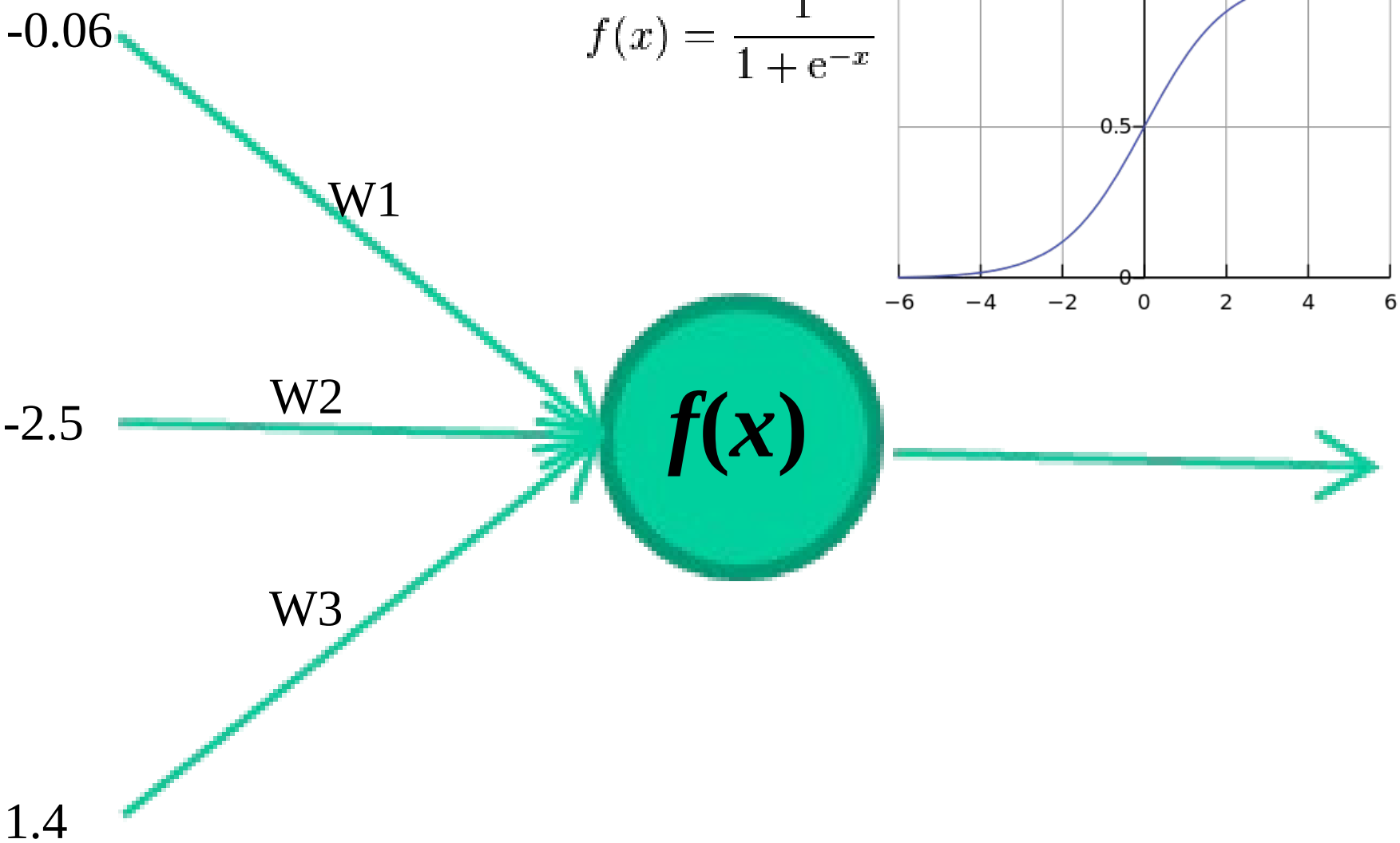
Neural networks

- Base algorithm
  - Initially assign random values to weights and offsets
  - Process instances in the training set one at a time
    - For each neuron, compute the result when applying weights, offset and activation function for the instance
    - Forward propagation until the output is computed
    - Compare the computed output with the expected output, and evaluate error
    - Backpropagation of the error, by updating weights and offset for each neuron
  - The process ends when
    - % of accuracy above a given threshold
    - % of parameter variation (error) below a given threshold
    - The maximum number of epochs is reached

Neural networks

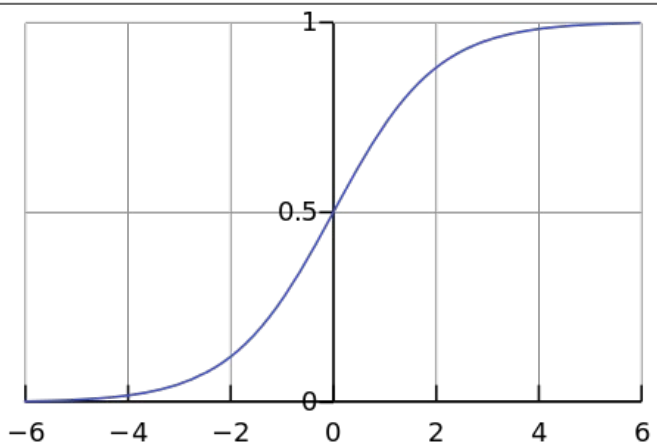
$$f(x) = \frac{1}{1 + e^{-x}}$$

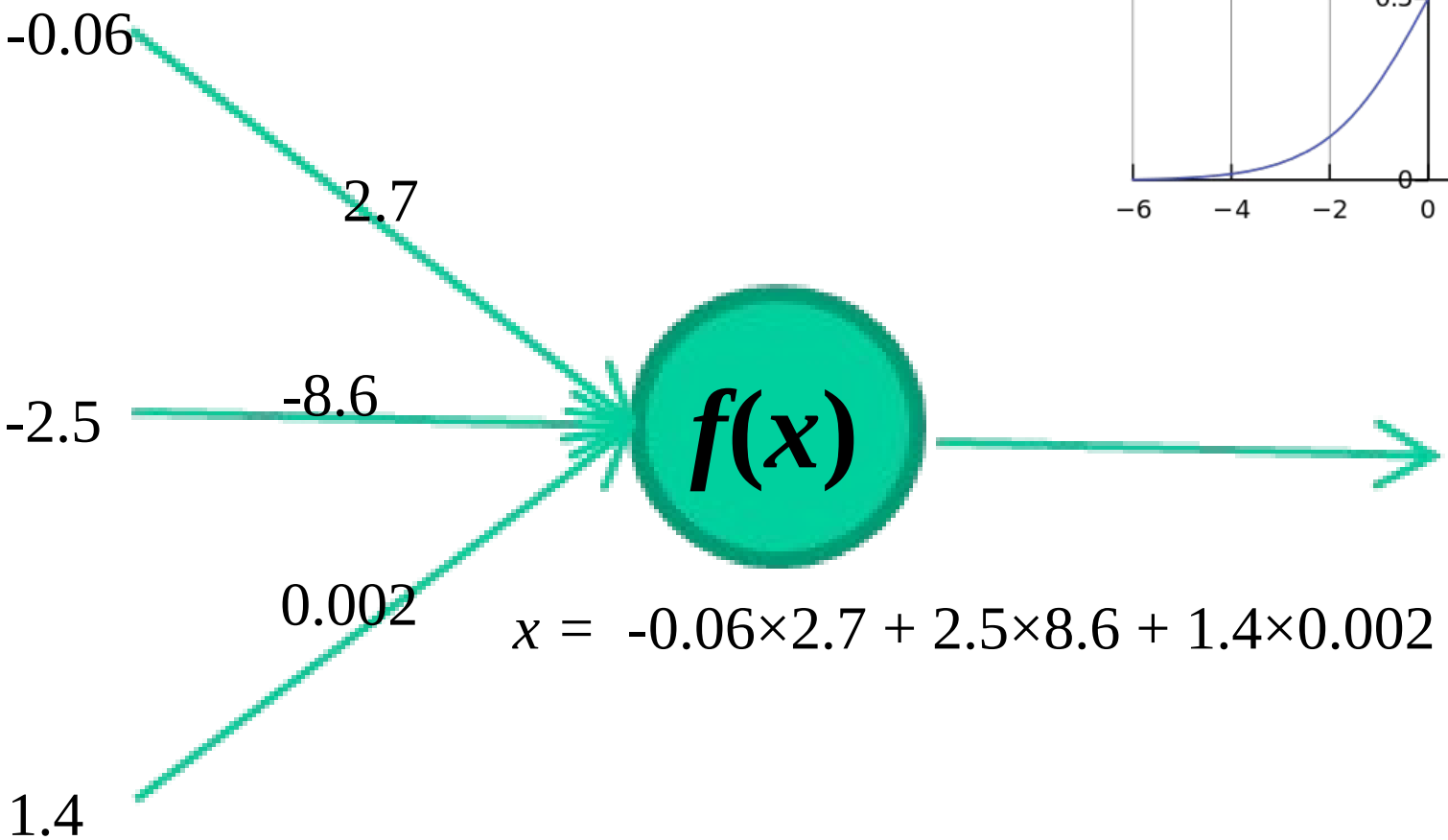




Neural networks

$$f(x) = \frac{1}{1+e^{-x}}$$





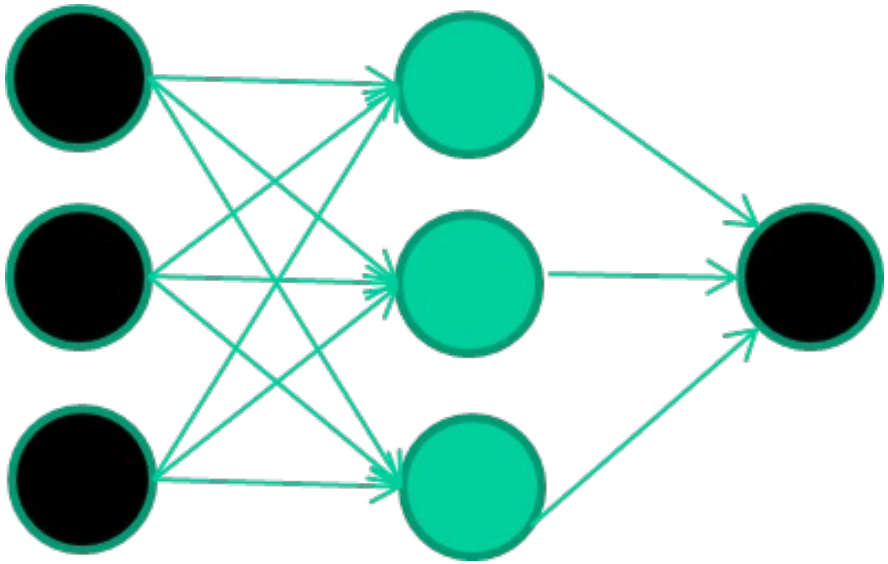
x = -0.06×2.7 + 2.5×8.6 + 1.4×0.002 = 21.34

Neural networks

A dataset

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

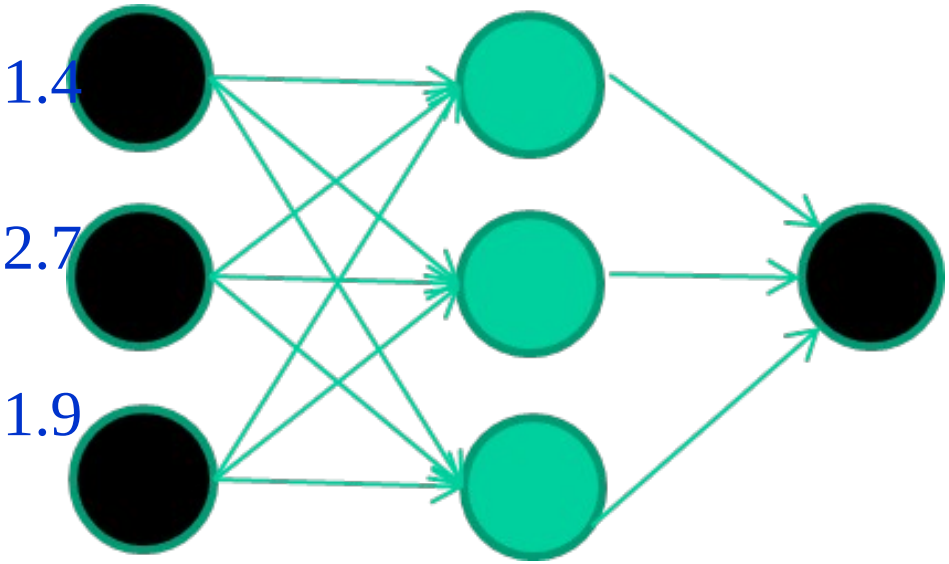


Initialise with random weights

Neural networks

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

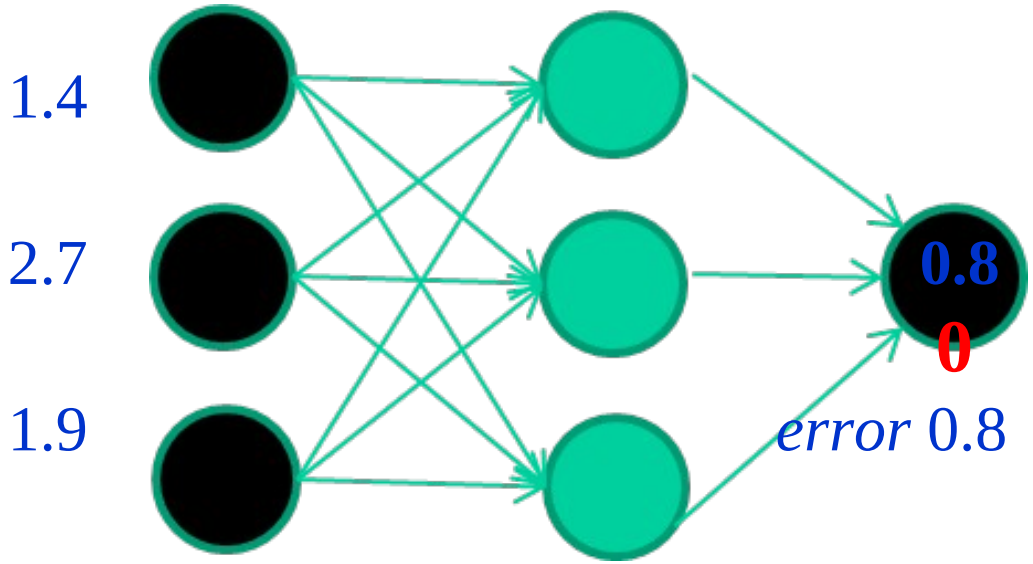


Present a training pattern

Neural networks

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			



Feed it through to get output

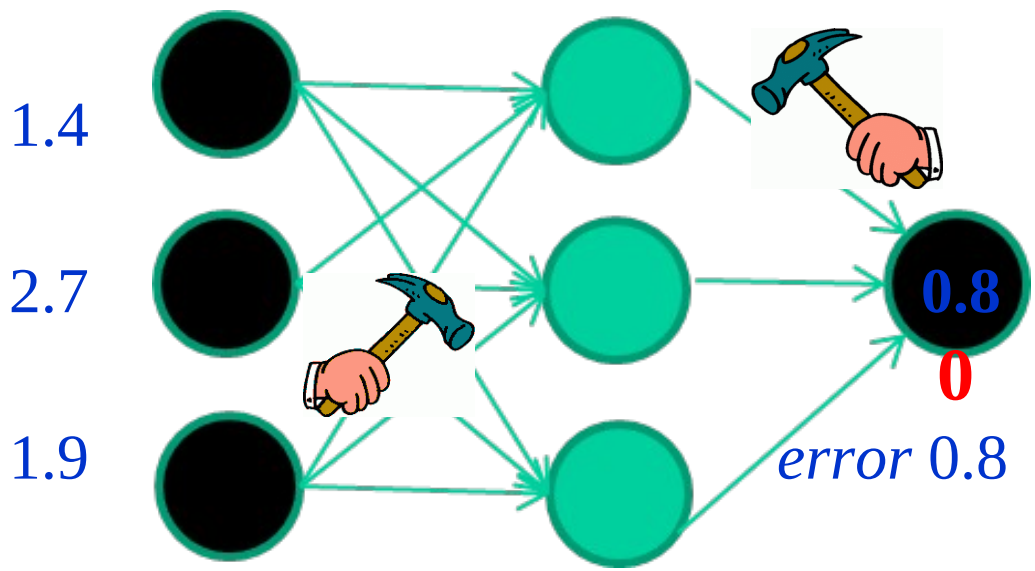
Compare with target output

Neural networks

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

Adjust weights based on error: Backpropagation of error

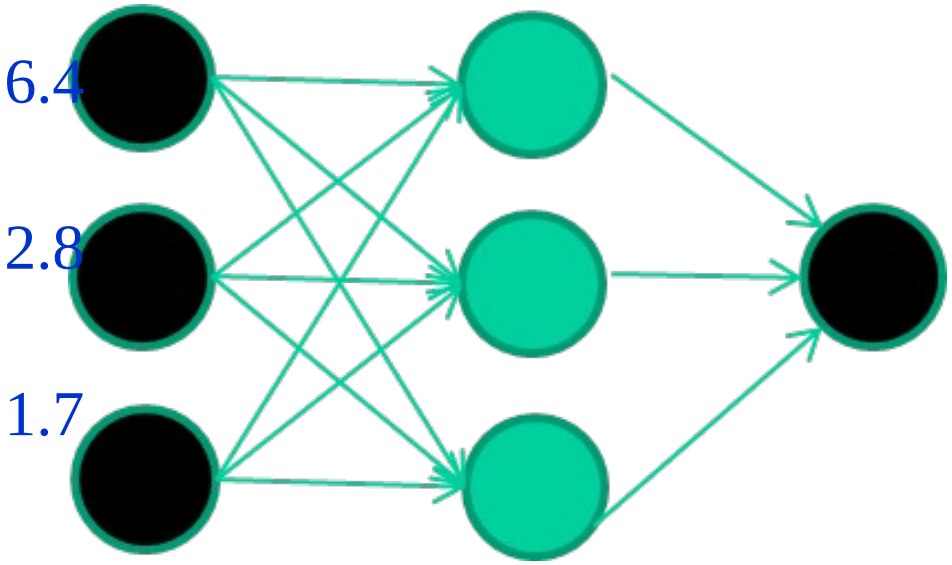


Neural networks

Present a training pattern

Training data

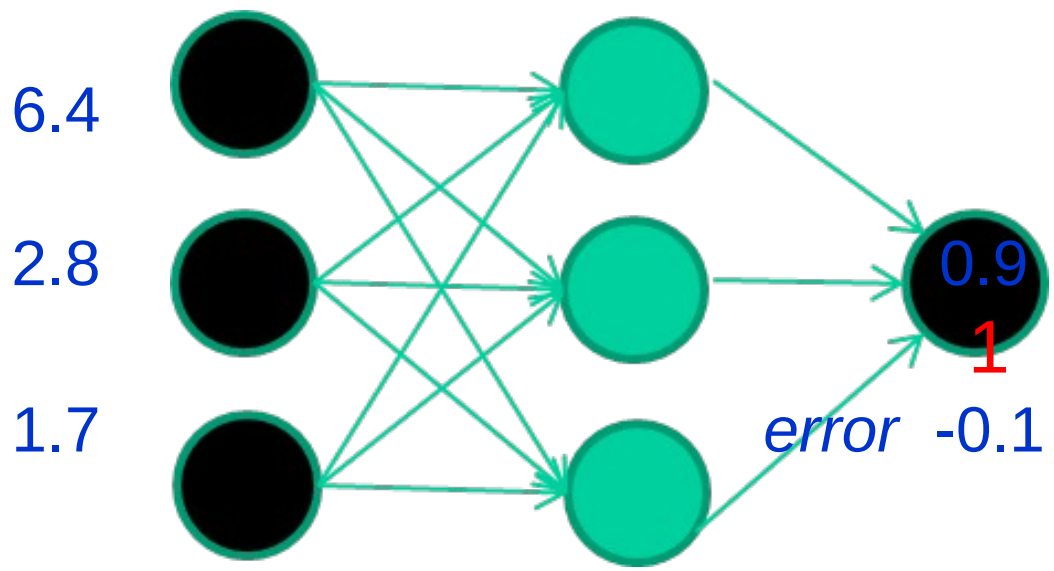
Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			



Neural networks

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			



Feed it through to get output

Compare with target output

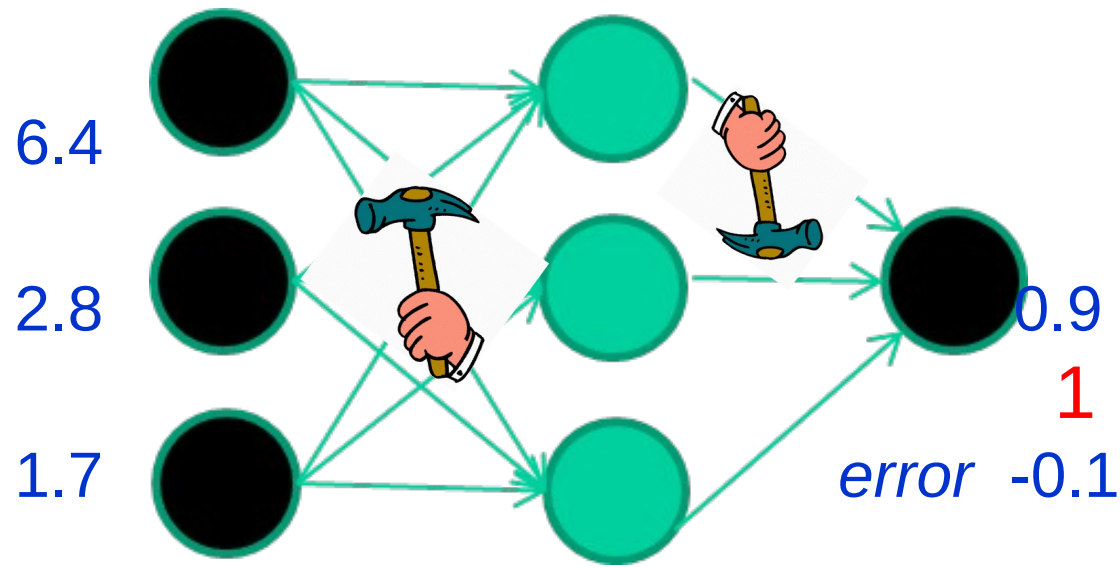
Neural networks

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

And so on ….



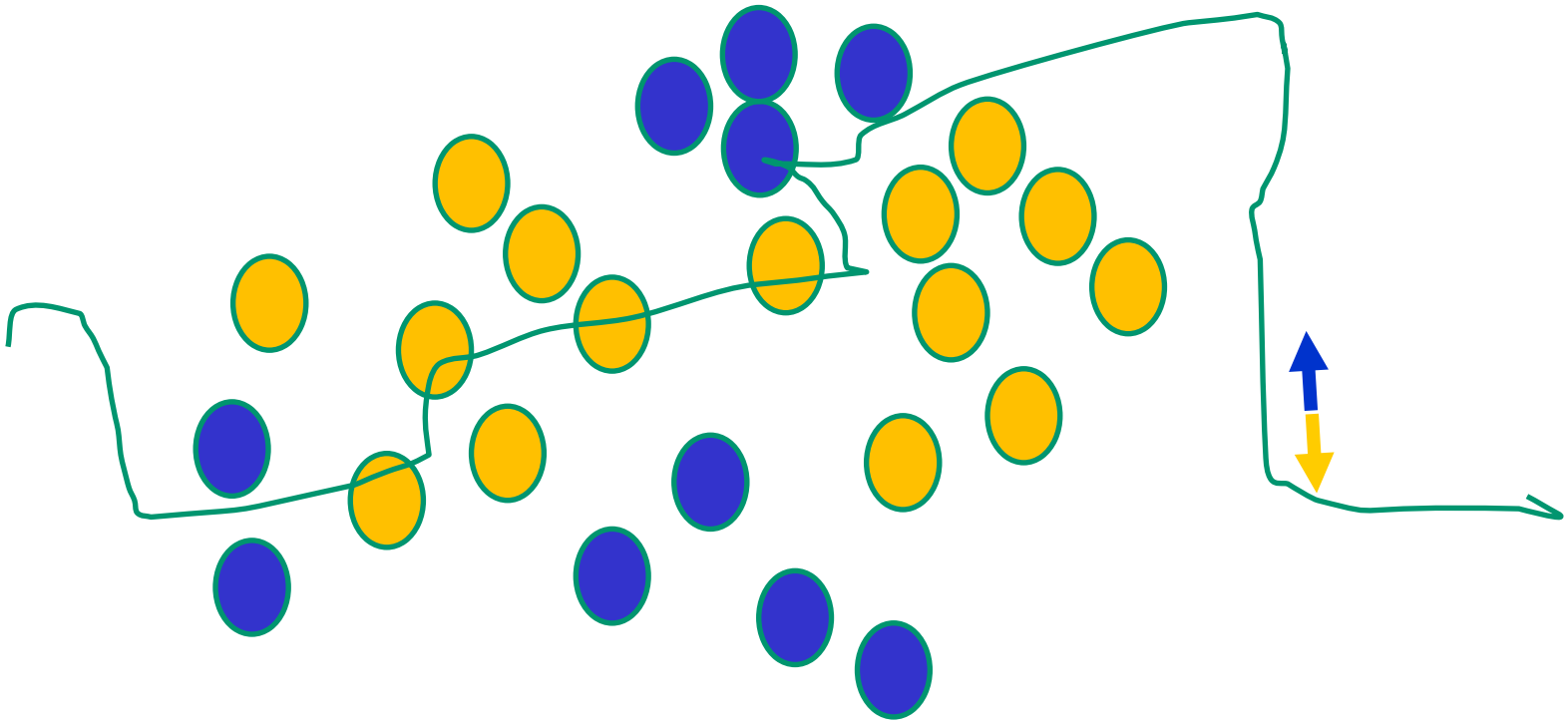
Repeat this thousands, maybe millions of times – each time taking a random training instance, and making slight weight adjustments

Algorithms for weight adjustment are designed to make changes that will reduce the error

Neural networks

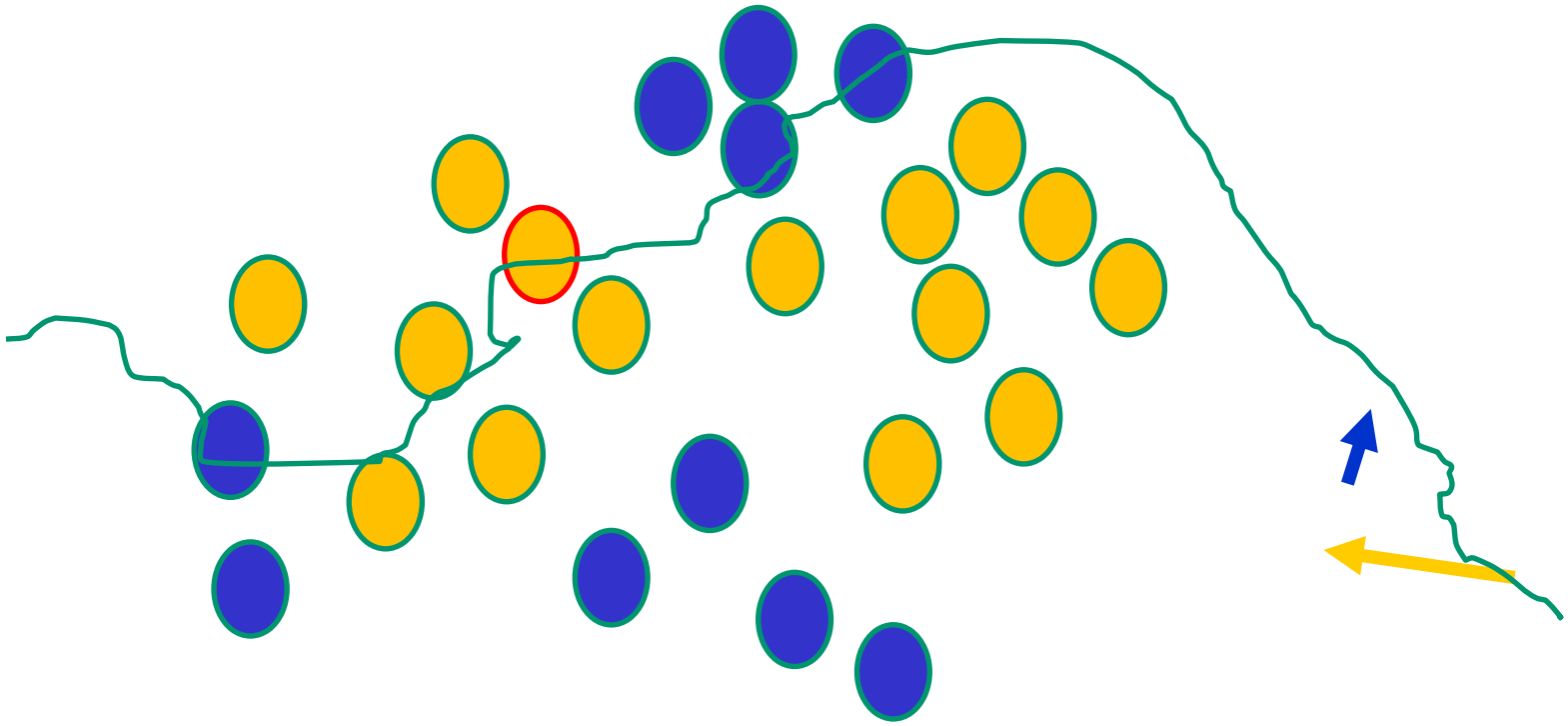
Initial random weights

Decision boundary



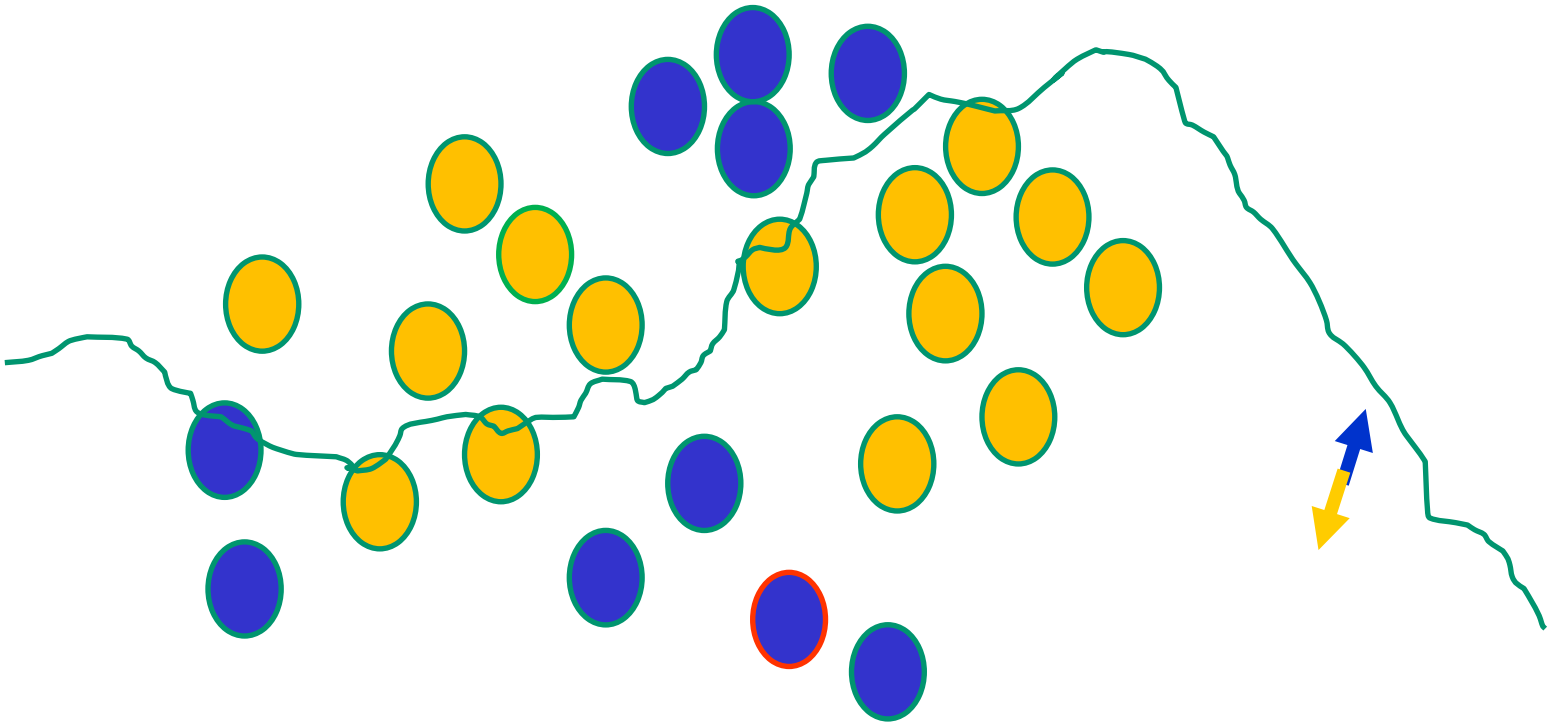
Neural networks

Present a training instance / adjust the weights



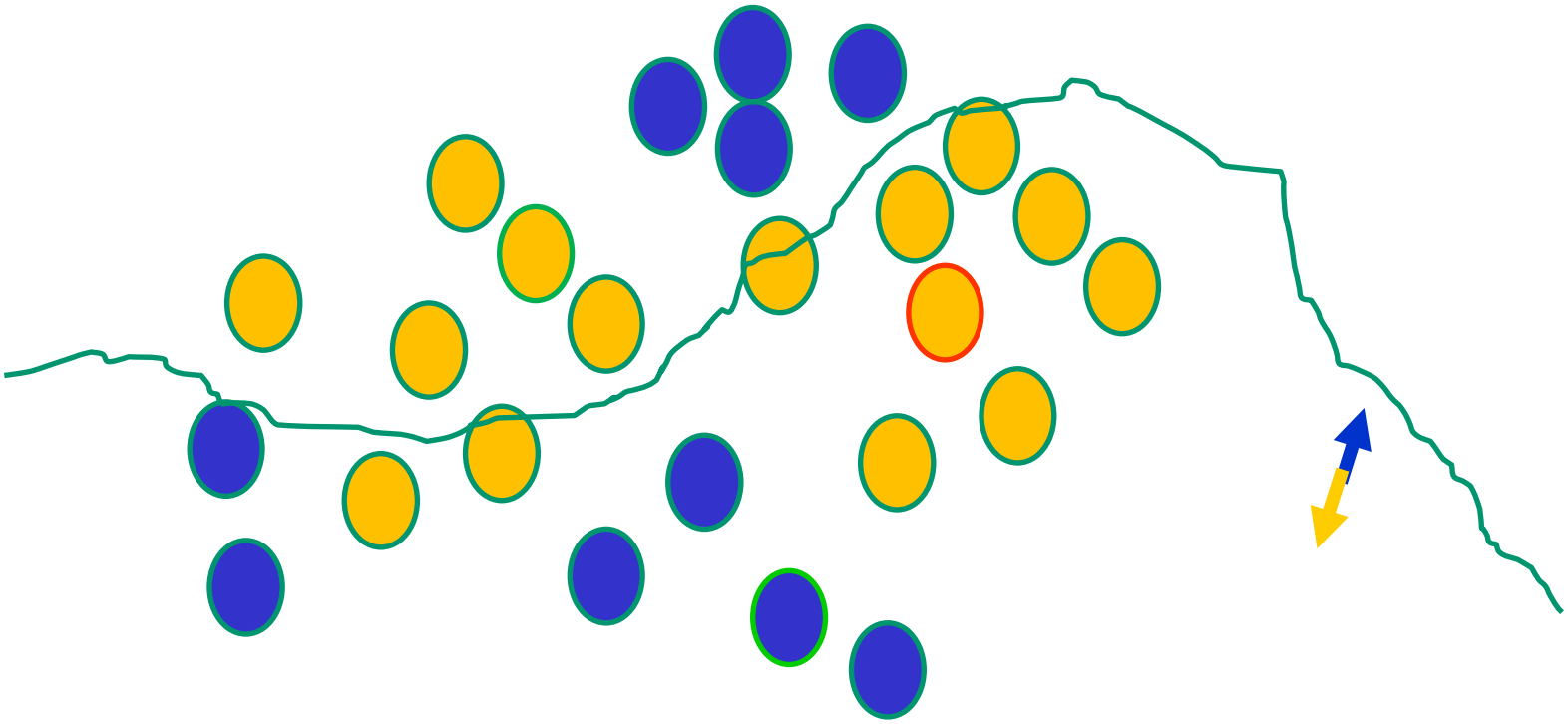
Neural networks

Present a training instance / adjust the weights



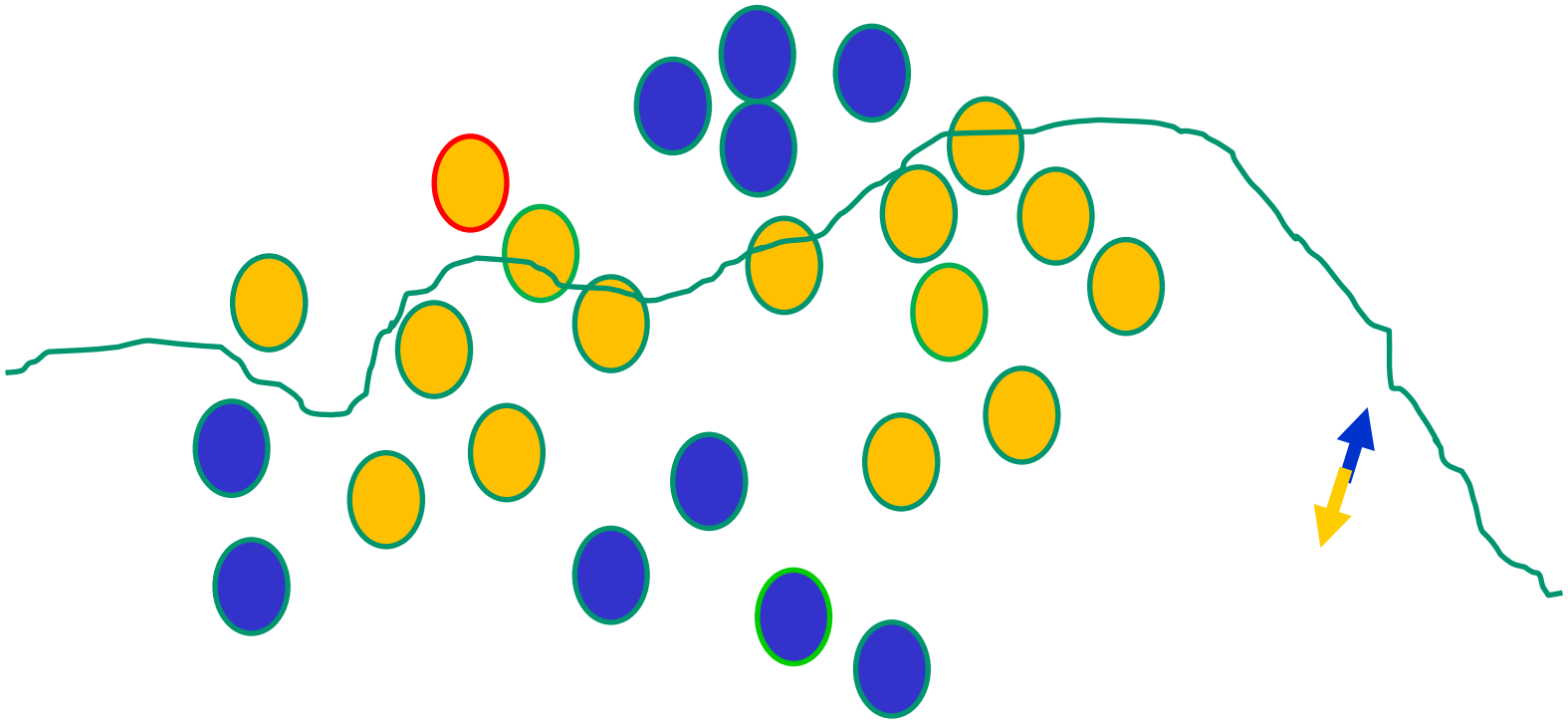
Neural networks

Present a training instance / adjust the weights



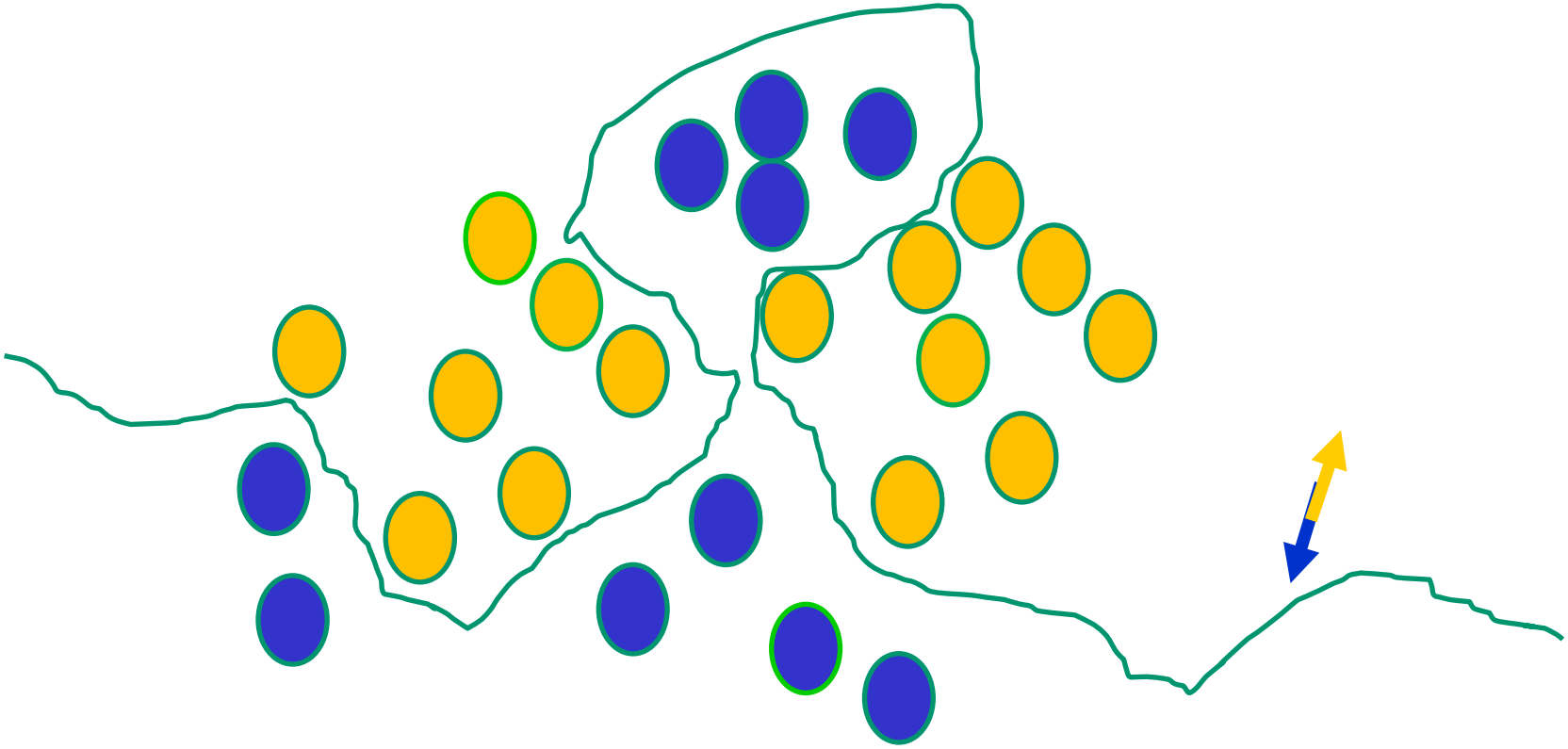
Neural networks

Present a training instance / adjust the weights



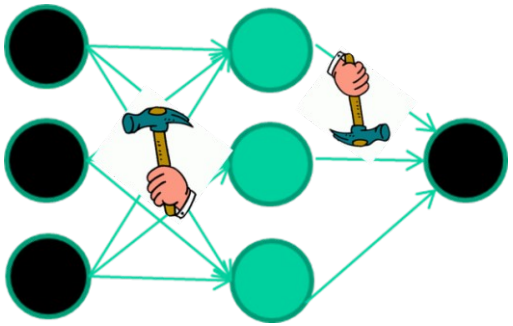
Neural networks

Eventually ….



Neural networks

- Weight-learning algorithms for NNs are “dumb”
- They work by making thousands and thousands of tiny adjustments, each making the network do better at the most recent pattern, but perhaps a little worse on many others
- Eventually this tends to be good enough to learn effective classifiers for many real applications



Neural networks

- Strong points
  - High accuracy
  - Robust to noise and outliers
  - Supports both discrete and continuous output
  - Efficient during classification
- Weak points
  - Long training time
    - weakly scalable in training data size
    - complex configuration
  - Not interpretable model
    - application domain knowledge cannot be exploited in the model

Neural networks in MLlib

- Use the **MultilayerPerceptronClassifier** estimator from **pyspark.ml.classification** on a DataFrame
- Explicitly specify input columns **featureCol** (vector) and **labelCol** (double)
- Output columns:
  - **predictionCol** with the Predicted label
  - **RawPredictionCol** (vector) with the counts of training instance labels at output which makes the prediction
  - **probabilityCol** probability equal to rawPrediction normalized to a multinomial distribution

Neural networks in MLlib

- Multilayer perceptron classifier is a classifier based on the feedforward artificial neural network that employs backpropagation for learning the model.
- It consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network.
- Nodes in the input layer represent the input data.
- The number of nodes N in the output layer corresponds to the number of classes.

Neural networks in MLlib

- (Some) parameters:
  - **maxIter:** Set the maximum number of iterations. Default is 100
  - **layers**: list of integers indicating layer sizes including input size and output size.
  - **blockSize:** Block size for stacking input data in matrices to speed up the computation. Data is stacked within partitions. If block size is more than remaining data in a partition then it is adjusted to the size of this data. Recommended size is between 10 and 1000. Default: 128
  - **seed:** Set the seed for weights initialization.

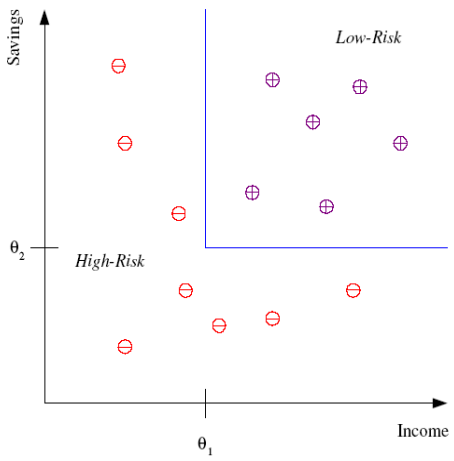
Neural networks: example

- Same example as for decision tree
- Same code for preprocessing the data
  - label
  - features

Neural networks: example

Preprocessed DataFrame

Savings	Income	Risk	RiskIndex	features
15000	1000	Low	1.0	[15000.0,1000.0]
0	5000	High	0.0	[0.0,5000.0]
20000	800	High	0.0	[20000.0,800.0]
6000	1300	Low	1.0	[6000.0,1300.0]
50000	2500	Low	1.0	[50000.0,2500.0]
2000	1100	Low	1.0	[2000.0,1100.0]
700	1500	High	0.0	[700.0,1500.0]
75000	0	High	0.0	[75000.0,0.0]
4000	500	High	0.0	[4000.0,500.0]
7000	3000	Low	1.0	[7000.0,3000.0]
3000	900	High	0.0	[3000.0,900.0]
6000	1200	Low	1.0	[6000.0,1200.0]



Neural networks: example

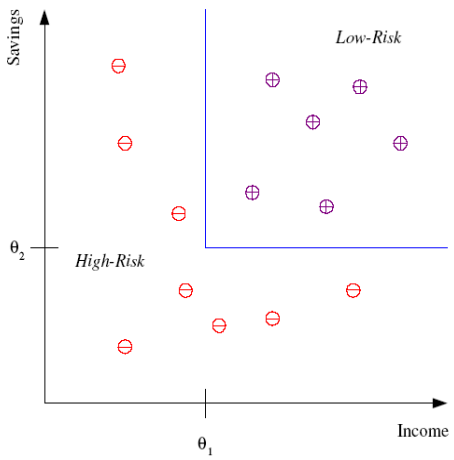
Train the neural network:

```
from pyspark.ml.classification import MultilayerPerceptronClassifier

# specify layers for the neural network:
# input layer of size 2 (features), one intermediate of size 4
# and output of size 3 (classes -> Low, High, Other)
layers = [2, 4, 3]

# create the trainer and set its parameters
nn = MultilayerPerceptronClassifier(labelCol="RiskIndex",
    featuresCol="features",maxIter=200, layers=layers, blockSize=128,
    seed=1234)

# train the model
nnModel = nn.fit(processedDF)

resultDF = nnModel.transform(processedDF)
```

Neural networks: example

Preprocessed DataFrame

Savings	Income	Risk	RiskIndex	features
15000	1000	Low	1.0	[15000.0,1000.0]
0	5000	High	0.0	[0.0,5000.0]
20000	800	High	0.0	[20000.0,800.0]
6000	1300	Low	1.0	[6000.0,1300.0]
50000	2500	Low	1.0	[50000.0,2500.0]
2000	1100	Low	1.0	[2000.0,1100.0]
700	1500	High	0.0	[700.0,1500.0]
75000	0	High	0.0	[75000.0,0.0]
4000	500	High	0.0	[4000.0,500.0]
7000	3000	Low	1.0	[7000.0,3000.0]
3000	900	High	0.0	[3000.0,900.0]
6000	1200	Low	1.0	[6000.0,1200.0]



Output DataFrame

Savings	Income	Risk	RiskIndex	features	rawPrediction	probability	prediction
15000	1000	Low	1.0	[15000.0,1000.0]	[6.91508177454728...]	[0.44444434912192...]	1.0
0	5000	High	0.0	[0.0,5000.0]	[9.49675872014066...]	[0.99999753045838...]	0.0
20000	800	High	0.0	[20000.0,800.0]	[6.91508177454728...]	[0.44444434912192...]	1.0
6000	1300	Low	1.0	[6000.0,1300.0]	[6.91508177454728...]	[0.44444434912192...]	1.0
50000	2500	Low	1.0	[50000.0,2500.0]	[6.91508177454728...]	[0.44444434912192...]	1.0
2000	1100	Low	1.0	[2000.0,1100.0]	[1.26783512041497...]	[3.17838479395997...]	1.0
700	1500	High	0.0	[700.0,1500.0]	[13.3948303637165...]	[0.99999845564030...]	0.0
75000	0	High	0.0	[75000.0,0.0]	[6.91508177454728...]	[0.44444434912192...]	1.0
4000	500	High	0.0	[4000.0,500.0]	[6.91508177454728...]	[0.44444434912192...]	1.0
7000	3000	Low	1.0	[7000.0,3000.0]	[6.91508177454728...]	[0.44444434912192...]	1.0
3000	900	High	0.0	[3000.0,900.0]	[6.91508177454728...]	[0.44444434912192...]	1.0
6000	1200	Low	1.0	[6000.0,1200.0]	[6.91508177454728...]	[0.44444434912192...]	1.0

Neural networks: example

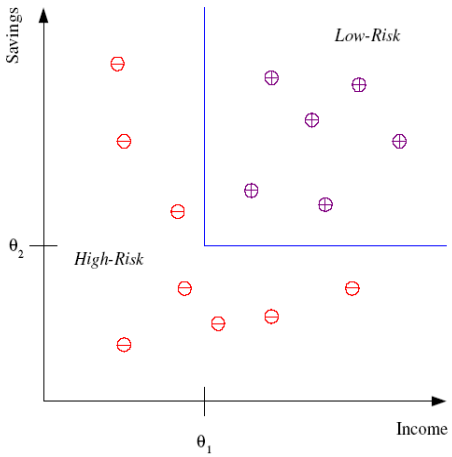
Test the neural network:

```
testData=spark.read.csv('credit_score_test.txt',header=True,inferSch
    ema=True)

processedTestDF=va.transform(testData)

finalTestDF=nnModel.transform(processedTestDF)
```

Neural networks: example

Test DataFrame

```
+-------+------+----+
|Savings|Income|Risk|
+-------+------+----+
| 100000| 10000| Low|
|    100|   100|High|
|   3100|   900|High|
|   2000|  1500|High|
|   3500|  1200| Low|
+-------+------+----+
```



Output Test DataFrame

```
+-------+------+----+-----------------+--------------------+--------------------+----------+
|Savings|Income|Risk|         features|       rawPrediction|         probability|prediction|
+-------+------+----+-----------------+--------------------+--------------------+----------+
| 100000| 10000| Low|[100000.0,10000.0]|[6.91508177454728...|[0.44444434912192...|       1.0|
|    100|   100|High|     [100.0,100.0]|[1.26783512606007...|[3.17838483248764...|       1.0|
|   3100|   900|High|    [3100.0,900.0]|[6.91508177454728...|[0.44444434912192...|       1.0|
|   2000|  1500|High|   [2000.0,1500.0]|[1.26783512041497...|[3.17838479395997...|       1.0|
|   3500|  1200| Low|   [3500.0,1200.0]|[6.91508177454728...|[0.44444434912192...|       1.0|
+-------+------+----+-----------------+--------------------+--------------------+----------+
```

Classification: Performance evaluation

Performance evaluation

- In order to test the goodness of algorithms there are some **evaluators**
- The Evaluator can be a **BinaryClassificationEvaluator** for binary data, or, more generally, a **MulticlassClassificationEvaluator** for multiclass problems.
- Provided metrics are:
  - Accuracy
  - Precision
  - Recall
  - F-measure

Performance evaluation

- Use the **MulticlassClassificationEvaluator** estimator from **pyspark.ml.evaluator** on a DataFrame
- The instantiated estimator has the method **evaluate()** that is applied to a dataframe
- It compares the prediction with the true label
- Output: the double value of the performance

Performance evaluation

- Parameters of **MulticlassClassificationEvaluator**:
  - **metricName**='accuracy', ‘f1’, ‘weightedPrecision’, or ’weightedRecall’
  - **labelCol**:input column with the true label/class
  - **predictionCol**: input column with the predicted class/label

Performance evaluation: example

- Classification model built for decision tree example
- Here we want to see the performance for the training and for the test set

Performance evaluation: example

Output Trained DataFrame

Savings	Income	Risk	RiskIndex	features	rawPrediction	probability	prediction
15000	1000	Low	1.0	[15000.0,1000.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
0	5000	High	0.0	[0.0,5000.0]	[2.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
20000	800	High	0.0	[20000.0,800.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
6000	1300	Low	1.0	[6000.0,1300.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
50000	2500	Low	1.0	[50000.0,2500.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
2000	1100	Low	1.0	[2000.0,1100.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
700	1500	High	0.0	[700.0,1500.0]	[2.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
75000	0	High	0.0	[75000.0,0.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
4000	500	High	0.0	[4000.0,500.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
7000	3000	Low	1.0	[7000.0,3000.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
3000	900	High	0.0	[3000.0,900.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
6000	1200	Low	1.0	[6000.0,1200.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0

Output Test DataFrame

Savings	Income	Risk	RiskIndex	features	rawPrediction	probability	prediction
100000	10000	Low	1.0	[100000.0,10000.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
100	100	High	0.0	[100.0,100.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
3100	900	High	0.0	[3100.0,900.0]	[4.0,0.0,0.0]	[1.0,0.0,0.0]	0.0
2000	1500	High	0.0	[2000.0,1500.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0
3500	1200	Low	1.0	[3500.0,1200.0]	[0.0,6.0,0.0]	[0.0,1.0,0.0]	1.0

Performance evaluation: example

Performance on training data:

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

myEvaluator1 =
    MulticlassClassificationEvaluator(labelCol="RiskIndex",predictionCol="prediction",
    metricName='accuracy')
myEvaluator2 =
    MulticlassClassificationEvaluator(labelCol="RiskIndex",predictionCol="prediction",
    metricName='weightedPrecision')

print("Accuracy on training is ", myEvaluator1.evaluate(finalDF))
print("Weighted precision on training is ", myEvaluator2.evaluate(finalDF))
```

Output

```
Accuracy on training is  1.0
Weighted precision on training is  1.0
```

Performance evaluation: example

Performance on test data:

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

myEvaluator1 =
    MulticlassClassificationEvaluator(labelCol="RiskIndex",predictionCol="prediction",
    metricName='accuracy')
myEvaluator2 =
    MulticlassClassificationEvaluator(labelCol="RiskIndex",predictionCol="prediction",
    metricName='weightedPrecision')

print("Accuracy on test is ", myEvaluator1.evaluate(finalTestDF))
print("Weighted precision on test is ", myEvaluator2.evaluate(finalTestDF))
```

Output

```
Accuracy on test is  0.8
Weighted precision on test is  0.8666666666666667
```

Performance evaluation: RDD APIs

- RDD APIs for performance evaluation
- Tool that allows to evaluate multiple classification metrics at once
- Now (November 2020) you still need to create an RDD from a DataFrame to use this tool
- Likely soon there will be an API for directly operating over DataFrames

Performance evaluation: RDD APIs

- Map DataFrame to an RDD with tuple of label and prediction
- Apply **MulticlassMetric** (or **MultilabelMetric** or **BinaryMetric**) of **pyspark.mllib.evaluation**
- Access the desired computed metric
  - accuracy(label)
  - recall(label)
  - precision(label)
  - f1measure(label)
  - ...

Performance evaluation with RDD APIs: example

Output Test DataFrame

```
+-------+------+----+---------+------------------+-------------+-------------+----------+
|Savings|Income|Risk|RiskIndex|          features|rawPrediction|  probability|prediction|
+-------+------+----+---------+------------------+-------------+-------------+----------+
| 100000| 10000| Low|      1.0|[100000.0,10000.0]|[0.0,6.0,0.0]|[0.0,1.0,0.0]|       1.0|
|    100|   100|High|      0.0|     [100.0,100.0]|[4.0,0.0,0.0]|[1.0,0.0,0.0]|       0.0|
|   3100|   900|High|      0.0|    [3100.0,900.0]|[4.0,0.0,0.0]|[1.0,0.0,0.0]|       0.0|
|   2000|  1500|High|      0.0|   [2000.0,1500.0]|[0.0,6.0,0.0]|[0.0,1.0,0.0]|       1.0|
|   3500|  1200| Low|      1.0|   [3500.0,1200.0]|[0.0,6.0,0.0]|[0.0,1.0,0.0]|       1.0|
+-------+------+----+---------+------------------+-------------+-------------+----------+
```

Performance evaluation with RDD APIs: example

Performance on test data:

```
from pyspark.mllib.evaluation import MulticlassMetrics

outRDD=finalTestDF.select("prediction","RiskIndex").rdd.map(lambda x: (float(x[0]),float(x[1])))
metrics=MulticlassMetrics(outRDD)

# Overall statistics
precision = metrics.precision()
recall = metrics.recall()
f1Score = metrics.fMeasure()
print("Summary Stats")
print("Precision = %s" % precision)
print("Recall = %s" % recall)
print("F1 Score = %s" % f1Score)

# Statistics by class
labels = outRDD.map(lambda lp: lp[1]).distinct().collect()
for label in sorted(labels):
    print("Class %s precision = %s" % (label, metrics.precision(label)))
    print("Class %s recall = %s" % (label, metrics.recall(label)))
    print("Class %s F1 Measure = %s" % (label, metrics.fMeasure(label, beta=1.0)))

# Weighted stats
print("Weighted recall = %s" % metrics.weightedRecall)
print("Weighted precision = %s" % metrics.weightedPrecision)
print("Weighted F(1) Score = %s" % metrics.weightedFMeasure())
print("Weighted F(0.5) Score = %s" % metrics.weightedFMeasure(beta=0.5))
print("Weighted false positive rate = %s" % metrics.weightedFalsePositiveRate)
```

Performance evaluation with RDD APIs: example

Output Test DataFrame

```
+-------+------+----+---------+------------------+-------------+-------------+----------+
|Savings|Income|Risk|RiskIndex|          features|rawPrediction|  probability|prediction|
+-------+------+----+---------+------------------+-------------+-------------+----------+
| 100000| 10000| Low|      1.0|[100000.0,10000.0]||[0.0,6.0,0.0]||[0.0,1.0,0.0]|       1.0|
|    100|   100|High|      0.0|     [100.0,100.0]||[4.0,0.0,0.0]||[1.0,0.0,0.0]|       0.0|
|   3100|   900|High|      0.0|    [3100.0,900.0]||[4.0,0.0,0.0]||[1.0,0.0,0.0]|       0.0|
|   2000|  1500|High|      0.0|   [2000.0,1500.0]||[0.0,6.0,0.0]||[0.0,1.0,0.0]|       1.0|
|   3500|  1200| Low|      1.0|   [3500.0,1200.0]||[0.0,6.0,0.0]||[0.0,1.0,0.0]|       1.0|
+-------+------+----+---------+------------------+-------------+-------------+----------+
```

Computed metrics

```
Summary Stats
Precision = 0.8
Recall = 0.8
F1 Score = 0.8
Class 0.0 precision = 1.0
Class 0.0 recall = 0.6666666666666666
Class 0.0 F1 Measure = 0.8
Class 1.0 precision = 0.6666666666666666
Class 1.0 recall = 1.0
Class 1.0 F1 Measure = 0.8
Weighted recall = 0.8
Weighted precision = 0.8666666666666667
Weighted F(1) Score = 0.8
Weighted F(0.5) Score = 0.8311688311688311
Weighted false positive rate = 0.13333333333333333
```

Classification: Parameter Tuning

Classification: Parameter Tuning

- The setting of the parameters of an algorithm is always a difficult task
- A “brute force” approach can be used to find the setting optimizing a quality index
  - The training data (“TrainValidation”) is split again into two subsets
    - The first set is used to build a model (training data)
    - The second one is used to evaluate the quality of the model (**validation data**)
  - The setting that maximizes a quality index (e.g., the prediction accuracy) is used to build the final model on the whole training dataset

Classification: Parameter Tuning

- Spark supports a **brute-force grid-based approach** to evaluate a set of possible parameter settings on a pipeline
- Input:
  - An MLlib pipeline or estimator
  - A set of values to be evaluated for each input parameter of the pipeline
    - All the possible combinations of the specified parameter values are considered and the related models are automatically generated and evaluated by Spark
  - A quality evaluation metric to evaluate the result of the input pipeline
- Output
  - The model associated with the best parameter setting, in term of quality evaluation metric

Classification: Parameter Tuning

- Tuning may be done for individual **Estimators**, or for entire **Pipelines** which include multiple algorithms, featurization, and other steps.
- Users can tune an entire Pipeline at once, rather than tuning each element in the Pipeline separately.

ParamGridBuilder

- To help construct the parameter grid, users can use the **ParamGridBuilder** utility of **pyspark.ml.tuning**
- By default, sets of parameters from the parameter grid are evaluated in serial
- Parameter evaluation can be done in parallel by setting **parallelism** with a value of 2 or more before running model selection with CrossValidator or TrainValidationSplit.
- The value of parallelism should be chosen carefully to maximize parallelism without exceeding cluster resources, and larger values may not always lead to improved performance.

TrainValidationSplit and CrossValidator

- In pyspark.ml the parameter tuning can be done easily. There are already two tools implemented:
  - **TrainValidationSplit**
  - **CrossValidator**
- These model selection tools work as follows:
  - They split the input data into separate training and validation datasets.
  - For each (training, validation) pair, they iterate through the set of ParamMaps:
    - For each ParamMap, they fit the Estimator using those parameters, get the fitted Model, and evaluate the Model performance using the Evaluator.
  - They select the Model produced by the best-performing set of parameters.

Train-Validation split

- **TrainValidationSplit** of **pyspark.ml.tuning** evaluates each combination of parameters once
- It creates a single (training, validation) dataset pair using the **trainRatio** parameter.
- **TrainValidationSplit** finally fits the Estimator using the best **ParamMap** and the entire dataset.

Cross validation

- One single split of the training set usually is biased
- **CrossValidator** of **pyspark.ml.tuning** evaluates each combination of parameters multiple times
- The cross-validation approach:
  - It creates **k splits** and **k models** (by defining the parameter **numFold)**
  - The **parameter setting** that achieves, on the average, the **best result on the k models** is selected as **final setting** of the algorithm parameters

Cross validation

- Best model is saved into **cvModel.bestModel**
- Perfomance metrics are saved into **cvModel.avgMetrics**
- Best parameter setting can be obtained thanks to **cvModel.getEstimatorParamMaps()[numpy.argmax(cvModel.avgMetrics)]**

Classification: Parameter Tuning - Example

- The following example shows how a grid-based approach can be used to tune the decision tree of the previous example
  - We used the CrossValidator component
- The following parameters of the decision tree algorithm are considered
  - Maximum depth
    - 1, 2, 10
  - Impurity
    - “Gini”, ”Entropy”
  - 6 parameter configurations are evaluated (3 x 2)

Classification: Parameter Tuning - Example

```
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.tuning import  ParamGridBuilder
from pyspark.ml.tuning import  CrossValidator
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

data=spark.read.csv('credit_score.txt',header=True,inferSchema=True)

indexer = StringIndexer(inputCol="Risk", outputCol="RiskIndex",
    handleInvalid="keep")
indexerModel = indexer.fit(data)
indexedDF=indexerModel.transform(data)

va=VectorAssembler(inputCols=["Savings","Income"], outputCol="features")
assembledDF=va.transform(indexedDF)

# Train a DecisionTree model.
dt = DecisionTreeClassifier(labelCol="RiskIndex", featuresCol="features")
```

Classification: Parameter Tuning - Example

```
paramGrid = ParamGridBuilder()\
    .addGrid(dt.maxDepth, [1,2,10]) \
    .addGrid(dt.impurity, ["Gini","Entropy"])\
    .build()

myEvaluator =
    MulticlassClassificationEvaluator(labelCol="RiskIndex",predictionC
    ol="prediction",metricName=”accuracy”)

cv=CrossValidator(estimator=dt,evaluator=myEvaluator,estimatorPa
    ramMaps=paramGrid, numFolds=3)

cvModel=cv.fit(assembledDF)
finalDF=cvModel.transform(assembledDF)
```

Classification: Parameter Tuning - Example

```
paramGrid = ParamGridBuilder()\
    .addGrid(dt.maxDepth, [1,2,10]) \
    .addGrid(dt.impurity, ["Gini","Entropy"])\
    .build()
```

```
myEvaluator =
```

There is one call to the addGrid method for each parameter that we want to set.
Each call to the addGrid method is characterized by
- The parameter we want to consider
- The list of values to test/to consider

```
cvModel=cv.fit(assembledDF)
finalDF=cvModel.transform(assembledDF)
```

Classification: Parameter Tuning - Example

Here, we set
- The estimator/pipeline to be evaluated
- The evaluator (i.e., the object that is used to compute the quality measure that is used to evaluate the quality of the model)
- The set of parameter values to be considered
- The number of folds to consider (i.e., the number or repetitions)

```
MulticlassClassificationEvaluator(labelCol="RiskIndex",predictionCol="prediction", metricName="accuracy")
```

```
cv=CrossValidator(estimator=dt,evaluator=myEvaluator,estimatorParamMaps=paramGrid, numFolds=3)
```

```
cvModel=cv.fit(assembledDF)
finalDF=cvModel.transform(assembledDF)
```

Classification: Parameter Tuning - Example

```
paramGrid = ParamGridBuilder()\
    .addGrid(dt.maxDepth, [1,2,10]) \
    .addGrid(dt.impurity, ["Gini","Entropy"])\
    .build()

myEvaluator =
    MulticlassClassificationEvaluator(labelCol="RiskIndex",predictionC
    ol="prediction")

cv=CrossValidator(estimator=dt,evaluator=myEvaluator,estimatorPa
    ramMaps=paramGrid, numFolds=3)

cvModel=cv.fit(assembledDF)
finalDF=cvModel.transform(assembledDF)
```

The returned model is the one associated with the best parameter setting, based on the result of the cross-validation test

Classification: Parameter Tuning - Example

```
import numpy

cvModel.getEstimatorParamMaps()
    [numpy.argmax(cvModel.avgMetrics)]
```

Classification: Parameter Tuning - Example

```
import numpy

cvModel.getEstimatorParamMaps()
    [numpy.argmax(cvModel.avgMetrics)]
```

The best parameters can be analyzed

Classification: Parameter Tuning - Example

```
import numpy

cvModel.getEstimatorParamMaps()
    [numpy.argmax(cvModel.avgMetrics)]
```

Best Parameters:

```
{Param(parent='undefined', name='maxDepth', doc='Maximum depth of the tree. (>= 0)
E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2 leaf nodes.'): 1,
 Param(parent='undefined', name='impurity', doc='Criterion used for information gain
calculation (case-insensitive). Supported options: entropy, gini'): 'Gini'}
```