Vectors (I)

- A vector has integer-typed and 0-based **indices** and double-typed **values**.
- Two types of local vectors: dense and sparse.
  - A dense vector is backed by a double array representing its entry values
  - A sparse vector is backed by two parallel arrays: indices and values.
- For example, a vector **(1.0, 0.0, 3.0)** can be represented
  - in *dense format* as `[1.0, 0.0, 3.0]`
  - in *sparse format* as `(3, [0, 2], [1.0, 3.0])`

$$\text{dense}: \underbrace{1.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{3.}$$

$$\text{sparse}: \begin{cases} \text{size}: 7 \\ \text{indices}: \underbrace{0}\ \underbrace{6} \\ \text{values}: \underbrace{1.}\ \underbrace{3.} \end{cases}$$

Vectors (II)

```
import numpy as np
from pyspark.mllib.linalg import Vectors

# Use a NumPy array as a dense vector
dv1 = np.array([1.0, 0.0, 3.0])

# Use a Python list as a dense vector
dv2 = [1.0, 0.0, 3.0]

# Create a DenseVector
dv3 = Vectors.dense([1.0, 0.0, 3.0])

# Create a SparseVector
sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])
```

Labeled Points (I)

- A **labeled point** is a vector, either dense or sparse, associated with a label/response.
- Labeled points are used in **supervised learning** algorithms.
- We use a **double** to store a **label**, so we can use labeled points in both regression and classification.
- For **binary classification**, a label should be either **0** (negative) or **1** (positive).
- For **multiclass classification**, labels should be class indices starting from zero: **0, 1, 2, ...**

Labeled Points (II)

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense
feature vector
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse
feature vector
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

Labeled Points (III)

```
from pyspark.mllib.regression import LabeledPoint

data = sc.textFile('sample_data.txt')

# Sample data contains lines of floats, separated by space
numbers = data.map(lambda line: [float(x) for x in line.split()])
print(numbers.take(2))

# We assume the first float if the label
# Remaining floats are features
labeled = numbers.map(lambda v: LabeledPoint(v[0], v[1:]))
print(labeled.take(2))
```

LIBSVM Format

- MLlib supports reading **training examples** stored in the LIBSVM format
- LIBSVM is a commonly used format that represents each document/ record as a **sparse vector**
- Each **text line** represents a **labeled sparse feature vector** using the following format:

```
label index1:value1 index2:value2 ...
```

where

  - **label** is an integer associated with the class label
  - the **indexes** are **one-based** (i.e., integer indexes starting from 1) representing the features
  - the **values** are the (double) values of the features

- **After loading**, the feature **indexes** are converted to **zero-based** (i.e., integer indexes starting from 0)

MLUtils

- This class contains **helper methods** to load, save and pre-process data used in MLlib.
- MLlib supports reading **training examples** stored in the LIBSVM format

```
from pyspark.mllib.util import MLUtils

data = MLUtils.loadLibSVMFile(sc, "sample_libsvm_data.txt")
labels = data.map(lambda x: x.label)
features = data.map(lambda x: x.features)

print(labels.take(10))
print(features.take(1))
```

Basic Statistics (I)

- Methods in the `mllib.stat.Statistics` class offer several widely used statistic functions that work directly on RDDs
- `Statistics.colStats(rdd)`
  - Computes a **statistical summary** of an **RDD of vectors**, which stores the min, max, mean, and variance for each column in the set of vectors. This can be used to obtain a wide variety of statistics in **one pass**.
- `Statistics.corr(rdd, method)`
  - Computes the **correlation matrix** between columns in an **RDD of vectors**, using either the **Pearson** or **Spearman** correlation.
- `Statistics.corr(rdd1, rdd2, method)`
  - Computes the **correlation** between **two RDDs of floating-point values**, using either the **Pearson** or **Spearman** correlation.
- Do not forget **basic statistics** offered by RDDs.

Basic Statistics (II)

```
import numpy as np
from pyspark.mllib.stat import Statistics

mat = sc.parallelize(
    [np.array([1.0, 10.0, 100.0]),
     np.array([2.0, 20.0, 200.0]),
     np.array([3.0, 30.0, 300.0])]
)  # an RDD of Vectors

# Compute column summary statistics.
summary = Statistics.colStats(mat)
# A dense vector containing the mean value for each column
print(summary.mean())
# column-wise variance
print(summary.variance())
# number of nonzeros in each column
print(summary.numNonzeros())
```

Basic Statistics (III)

```
from pyspark.mllib.stat import Statistics

# a series
seriesX = sc.parallelize([1.0,   2.0,  3.0,  3.0,   5.0])

# seriesY must have the same number of cardinality as seriesX
seriesY = sc.parallelize([11.0, 22.0, 33.0, 33.0, 555.0])

# Compute the correlation using Pearson's method.
# Enter "spearman" for Spearman's method.
# If a method is not specified, Pearson's method will be used by default.
print("Correlation is: " + str(Statistics.corr(seriesX, seriesY, method="pearson")))

data = sc.parallelize(
    [np.array([1.0, 10.0, 100.0]),
     np.array([2.0, 20.0, 200.0]),
     np.array([5.0, 33.0, 366.0])]
)  # an RDD of Vectors

# Compute the correlation matrix using Pearson's method.
# Use "spearman" for Spearman's method.
# If a method is not specified, Pearson's method will be used by default.
print(Statistics.corr(data, method="pearson"))
```

TF-IDF

- TF-IDF: **term frequency - inverse document frequency**
- Measures **how often** a word occurs in each document, **weighted** according to **how many** documents that word occurs in
- Words that occur in a **few documents** are given **more weight than** words that occur in **many documents**.
- In MLlib, TF and IDF are implemented in `HashingTF` and `IDF`.

HashingTF

```
from pyspark.mllib.feature import HashingTF, IDF

documents = sc.textFile('tragedies.txt').map(lambda line: line.split(' '))

hashingTF = HashingTF()
tf = hashingTF.transform(documents)

print(tf.take(100))

# ...
# SparseVector(1048576, {0: 1.0}),
# SparseVector(1048576, {144435: 1.0, 151357: 1.0, 296609: 1.0,
#                        653832: 1.0, 667177: 1.0, 749264: 1.0}),
# SparseVector(1048576, {328616: 1.0, 642021: 1.0}),
# SparseVector(1048576, {642021: 1.0, 859700: 1.0}),
# ...
```

Standard Scaler

- **Very common** pre-processing step
- Standardizes features by **scaling to unit variance** and/or removi**ng the mean**
- `StandardScaler` has the following parameters in the constructor:
  - `withMean` (`False` by default). Centers the data with mean before scaling.
    - It will build a dense output, so take care when applying to sparse input.
  - `withStd` (`True` by default). Scales the data to unit standard deviation.
- The `fit()` method takes an input of RDD[Vector], learns the summary statistics, and then return a model which can transform the input dataset into unit standard deviation and/or zero mean features depending how we configure the `StandardScaler`.

Standard Scaler

```
from pyspark.mllib.feature import StandardScaler
from pyspark.mllib.linalg import Vectors

vectors = [Vectors.dense([-2.0, 5.0, 1.0]),
           Vectors.dense([ 2.0, 0.0, 1.0])]

dataset = sc.parallelize(vectors)
scaler = StandardScaler(withMean=True, withStd=True)
model = scaler.fit(dataset)
result = model.transform(dataset)

print(result.collect())
# Result: {[-0.7071, 0.7071, 0.0], [0.7071, -0.7071, 0.0]}
```

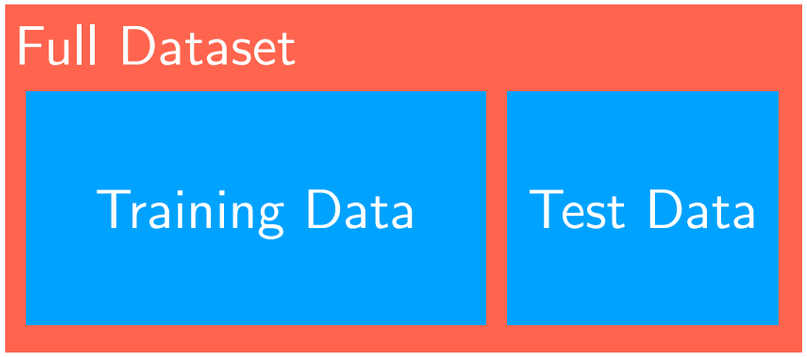
Normalization

- Common operation for text classification or clustering
- Scales individual samples to have unit L_p norm

```
from pyspark.mllib.feature import Normalizer
from pyspark.mllib.util import MLUtils

data = MLUtils.loadLibSVMFile(sc, "sample_libsvm_data.txt")
labels = data.map(lambda x: x.label)
features = data.map(lambda x: x.features)

normalizer1 = Normalizer()
normalizer2 = Normalizer(p=float("inf"))

# Each sample in data1 will be normalized using $L^2$ norm.
data1 = labels.zip(normalizer1.transform(features))

# Each sample in data2 will be normalized using $L^\infty$ norm.
data2 = labels.zip(normalizer2.transform(features))
```

Train & Test Sets

- **Split** data into a **training set** and a **test set**
- Use **training set** when **training a machine learning model**
  - Compute **training error** on the training set.
  - Try to **reduce** this training error
- Use **test set** to **measure the accuracy of the model**
  - **Test error** is the error when you run the **trained model** on **test data** (new data)



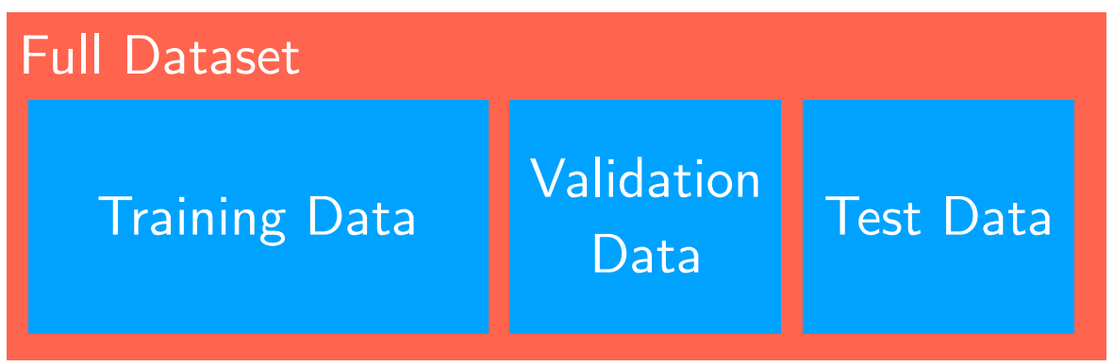

```
from pyspark.mllib.util import MLUtils

data = MLUtils.loadLibSVMFile(sc, "sample_libsvm_data.txt")
training, test = data.randomSplit([0.8, 0.2], seed=11)
```

Hyperparameters

- **Hyperparameters** are **settings** that we can use **to control the behavior** of a learning algorithm
- The values of hyperparameters **are not adapted** by the learning algorithm itself
- We **do not learn** the hyperparameters
  - It is not appropriate to learn that hyperparameter on the **training set**
  - If learned on the training set, such hyperparameters would always result in **overfitting**

Validation Sets

- To find **hyperparameters**, we need a **validation set** of examples that the **training algorithm does not observe**
- We construct the **validation set** from the **training data** (not the test data)
- We split the **training data** into **two disjoint subsets**:
  - One is used to **learn the parameters**
  - The other one (the validation set) is used to **estimate the test error during or after training**, allowing for the hyperparameters to be updated accordingly.



Clustering

- Clustering is an **unsupervised learning** problem
- We aim to **group subsets of entities** with one another based on some notion of **similarity**
- Clustering is often used for **exploratory analysis** and/or as a component of a **hierarchical** supervised learning **pipeline**
  - **Distinct** classifiers or regression models are trained **for each cluster**

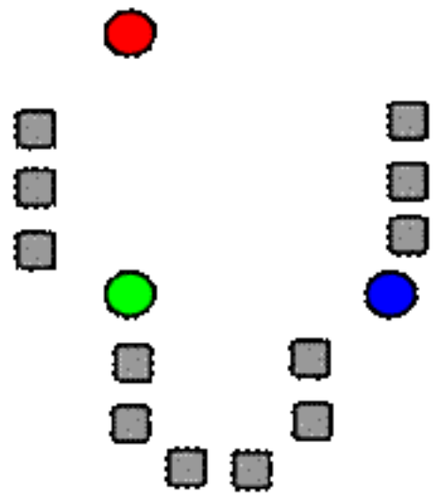
K-Means

- K-Means clustering **partitions** n data points into K clusters
  - each data point belongs to the cluster with a **nearest mean**
- Given a set of data points (x_1, x_2, ..., x_n), K-Means clustering aims to partition the n data points into K ($\leq n$) sets $S = \{S_1, S_2, ..., S_k\}$ so as to minimize the **within-cluster sum of squares** (WCSS):

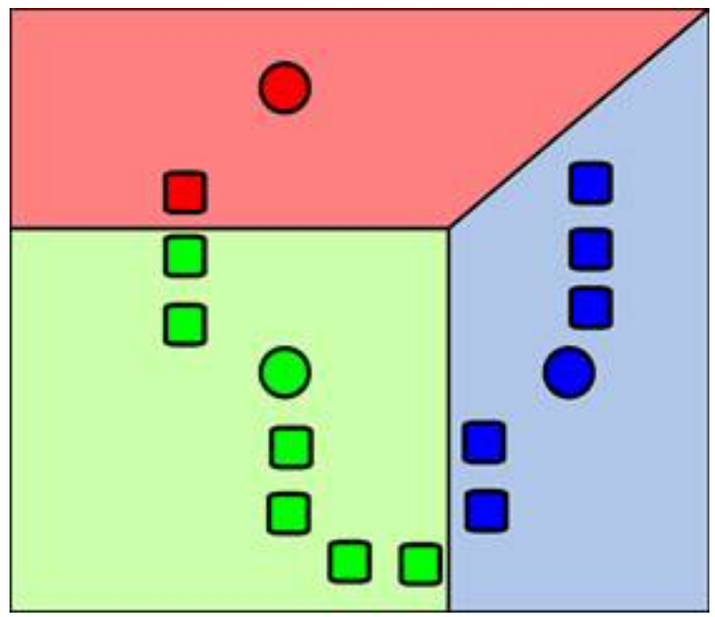
$$\operatorname{argmin} \sum_{i=1}^{K} \sum_{x \in S_i} \|x - \mu_i\|^2$$

where μ_i is the mean of points in S_i

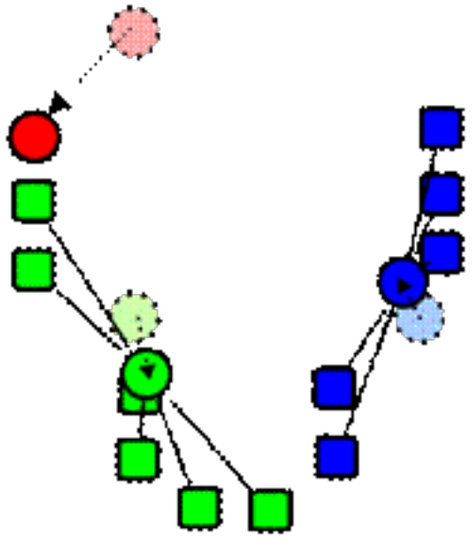
Example



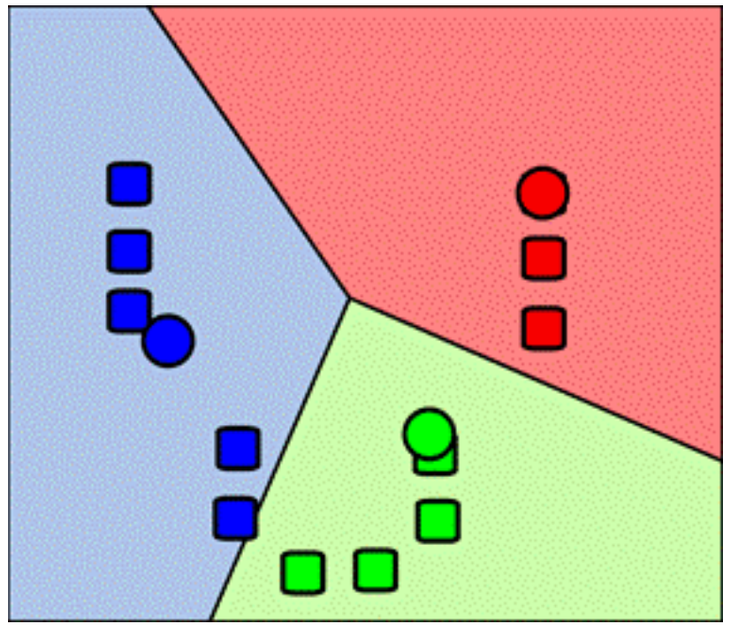
Initially, K=3 "means" are randomly generated



K clusters are created



The centroid of each cluster becomes the new mean



Continue until convergence is reached

K-Means in Spark

- **MLlib** comes bundled with K-Means implementation (KMeans) which can be imported from `pyspark.mllib.clustering` package
- Arguments to `KMeans.train()`:
  - `k` is the number of desired clusters
  - `maxIterations` is the maximum number of iterations to run.
  - `initializationMode` can be either '`random`' or '`k-meansII`'

K-Means in Spark

```
from pyspark.mllib.clustering import KMeans
from numpy import array

# 12 records with height, weight data
data = array([185,72, 170,56, 168,60, 179,68, 182,72,
              188,77, 180,71, 180,70, 183,84, 180,88,
              180,67, 177,76]).reshape(12,2)

# Generate Kmeans
model = KMeans.train(sc.parallelize(data), 2,
                     runs=50, initializationMode="random")
```

K-Means in Spark

```
# Print out the cluster of each data point

print (model.predict(array([185, 71])))
print (model.predict(array([170, 56])))
print (model.predict(array([168, 60])))
print (model.predict(array([179, 68])))
print (model.predict(array([182, 72])))
print (model.predict(array([188, 77])))
print (model.predict(array([180, 71])))
print (model.predict(array([180, 70])))
print (model.predict(array([183, 84])))
print (model.predict(array([180, 88])))
print (model.predict(array([180, 67])))
print (model.predict(array([177, 76])))

# Try the same with 3 clusters
```

K-Means in Spark

- Training and Storing the Model
  - This will create a directory, `savedModelDir` with two subdirectories `data` and `metadata` where the model is stored.

```
from pyspark.mllib.clustering import KMeans
from numpy import array

# 12 records with height, weight data
data = array([185,72, 170,56, 168,60, 179,68, 182,72,
              188,77, 180,71, 180,70, 183,84, 180,88,
              180,67, 177,76]).reshape(12,2)

# Generate Kmeans
model = KMeans.train(sc.parallelize(data),
                     runs=50, initializationMode="random")

model.save(sc, "savedModelDir")
```

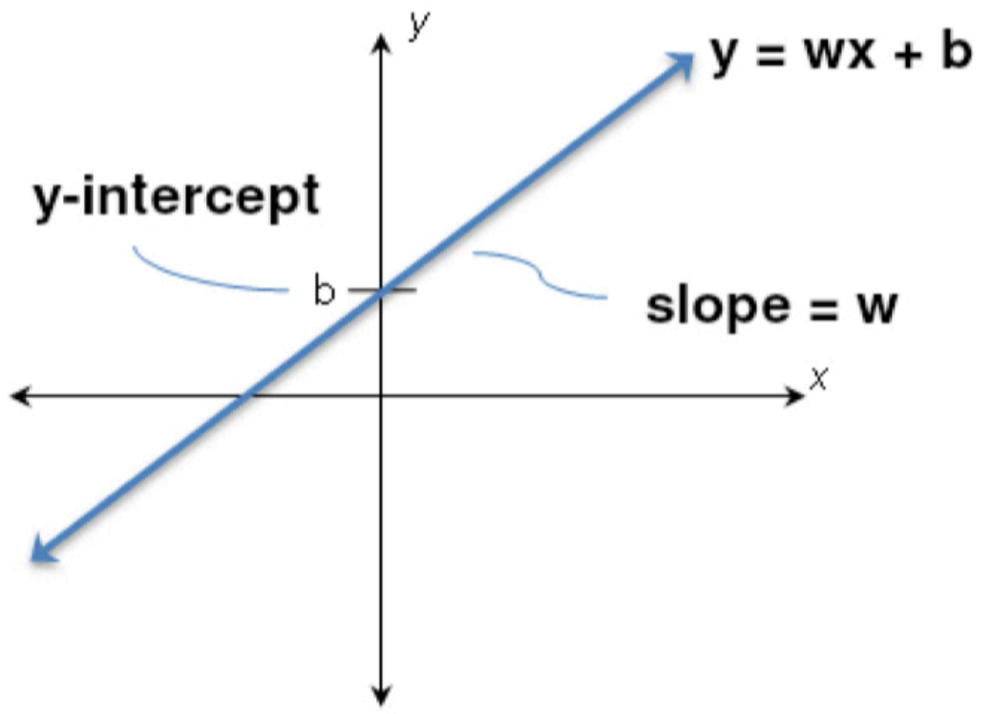
K-Means in Spark

```
model = KMeansModel.load(sc, "savedModelDir")


print (model.predict(array([185, 71])))
print (model.predict(array([170, 56])))
print (model.predict(array([168, 60])))
print (model.predict(array([179, 68])))
print (model.predict(array([182, 72])))
print (model.predict(array([188, 77])))
print (model.predict(array([180, 71])))
print (model.predict(array([180, 70])))
print (model.predict(array([183, 84])))
print (model.predict(array([180, 88])))
print (model.predict(array([180, 67])))
print (model.predict(array([177, 76])))
```

Linear Regression

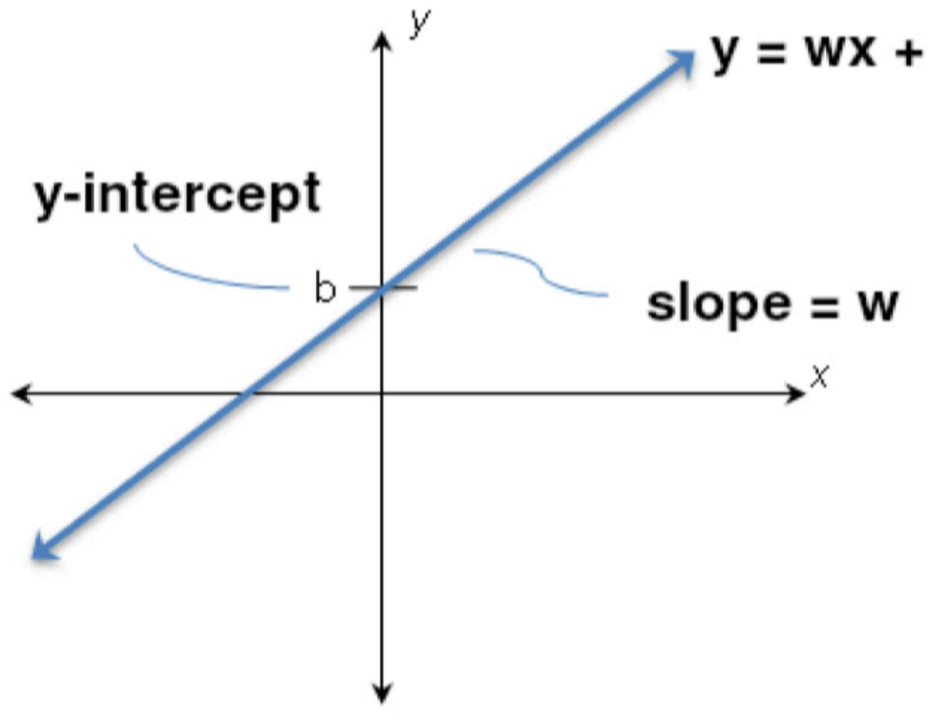
- We want to build a system that takes as input a **vector** $x = \{x_1, x_2, \ldots, x_n\} \in R^n$ and **predicts output** $y \in R^n$
- In linear regression, the output is a **linear function** of the input

$$y = f(x) = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$



Linear Regression

- The **weights** control the behavior of the model
  - if $w_i > 0$: **increasing** the value of feature x_i **increases** the output of the model
  - if $w_i < 0$: **increasing** the value of feature x_i **decreases** the output of the model
  - if $w_i = 0$: the value of feature x_i has **no effect** on the output of the model



- The average training error is the **mean squared error**:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(w^T x_i + b - y_i)$$

Linear Regression in Spark

```
from pyspark.mllib.util import MLUtils
from pyspark.mllib.regression import LinearRegressionWithSGD

data = MLUtils.loadLibSVMFile(sc, "regression_data.txt")
model = LinearRegressionWithSGD.train(data, iterations=10)

valuesAndPreds = data.map(lambda p: (p.label, model.predict(p.features)))

SE = valuesAndPreds.map(lambda vp: (vp[0] - vp[1])**2)
MSE = SE.reduce(lambda x, y: x + y) / valuesAndPreds.count()

print("Mean Squared Error = " + str(MSE))
print("Coefficients = " + str(model.weights))
print("Intercept = " + str(model.intercept))

from pyspark.mllib.regression import LinearRegressionModel
model.save(sc, "lrmodel")
sameModel = LinearRegressionModel.load(sc, "lrmodel")
```

Binary Classification

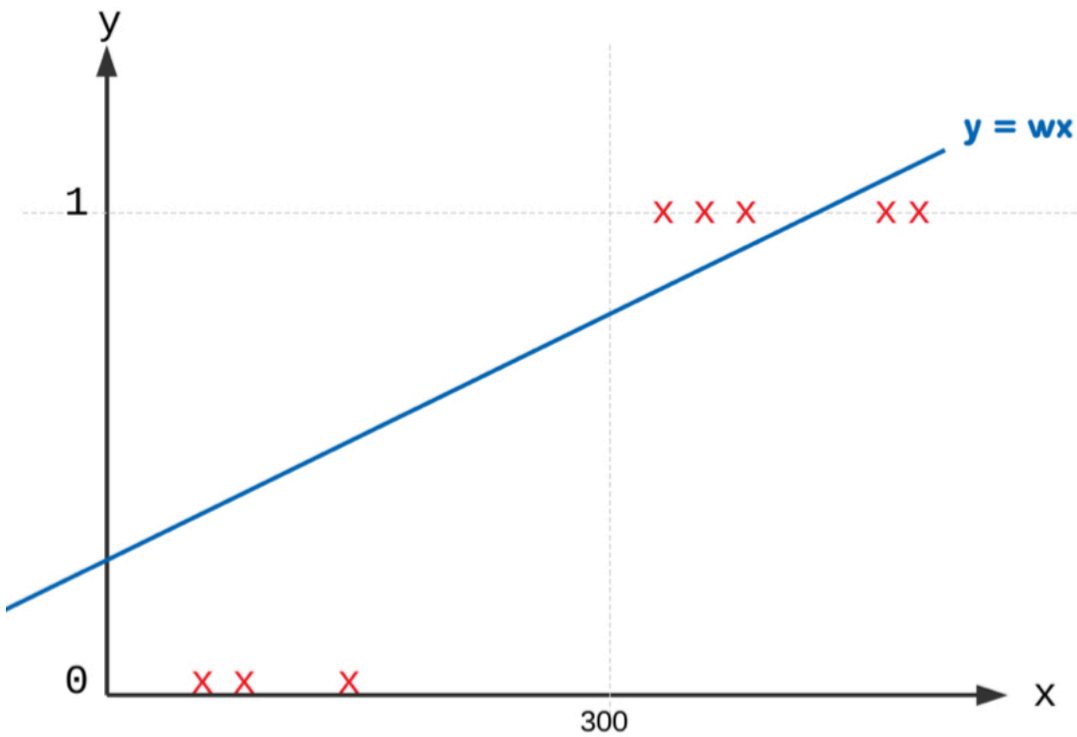
- We want to build a system that takes as input a **vector** $x = \{x_1, x_2, \ldots, x_n\} \in R^n$ and **predicts output** $y \in \{0, 1\}$
  - To specify which of **2 categories** an input x belongs to
- The model computes a **weighted sum** of the input features (plus a bias term)

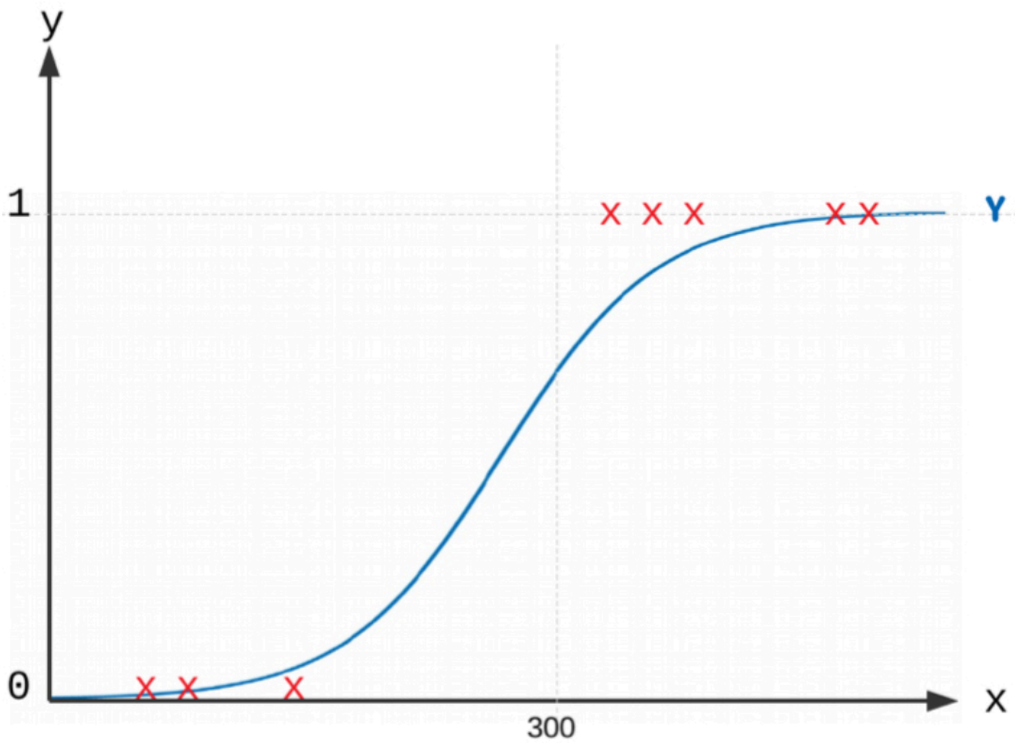
$$z = w_1 x_1 + \ldots + w_n x_n + b$$

- but it outputs the **logistic** of this result

$$y = \frac{1}{1 + e^{-z}}$$

Binary Classification



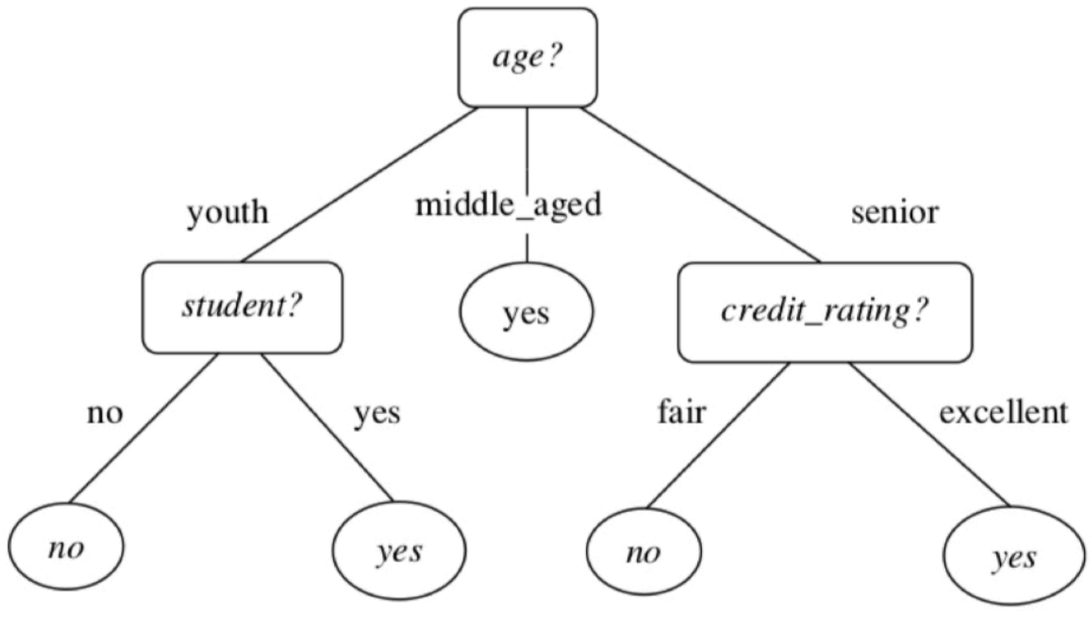


Binary Classification in Spark

```
from pyspark.mllib.classification import LogisticRegressionWithLBFGS, LogisticRegressionModel
from pyspark.mllib.regression import LabeledPoint

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.split(' ')]
    return LabeledPoint(values[0], values[1:])

data = sc.textFile("sample_data.txt")
parsedData = data.map(parsePoint)

# Build the model
model = LogisticRegressionWithLBFGS.train(parsedData)

# Evaluating the model on training data
labelsAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
trainErr = labelsAndPreds.filter(lambda lp: lp[0] != lp[1]).count() / float(parsedData.count())
print("Training Error = " + str(trainErr))

# Save and load model
model.save(sc, "logregmodel")
sameModel = LogisticRegressionModel.load(sc, "logregmodel")
```

Decision Trees

- **Decision trees are popular** methods for the machine learning tasks of **classification** and **regression**
- A decision tree is a **flowchart-like tree structure**
  - The **topmost node**: represents the root
  - Each **branch**: represents an outcome of the test
  - Each **internal node**: denotes a test on an attribute
  - Each **leaf**: holds a class label
- Decision trees are widely used since they are **easy to interpret**



Training Decision Trees

- Decision trees are constructed in a **top-down, recursive, divide-and-conquer** manner
- The tree predicts the **same label** for each **leaf partition**
- Each partition is chosen **greedily** by selecting the **best split** from a set of splits, to **maximize the information gain** at a tree node.
- The **node impurity** is a **measure of the homogeneity of the labels** at the node.
  - Two impurity measures for **classification**: **Gini impurity** and **entropy**
  - One impurity measure for **regression**: **variance**
- The **information gain** is the difference between the **parent node impurity** and the **weighted sum of the two child node impurities**

Decision Trees in Spark

- **Problem specification parameters**
  - Do **not** require **tuning**
  - `algo`: **type** of decision tree, Classification or Regression
  - `numClasses`: number of **classes** (classification only)
  - `categoricalFeatureInfo`: which **features** are **categorical** and **how many** categorical **values** each of those features can take
    - Given as a **map from feature indices to feature arity** (number of categories)
    - Any features not in this map are treated as **continuous**
      - For example, `Map(0 -> 2, 4 -> 10)` specifies that feature `0` is binary (taking values 0 or 1) and that feature `4` has 10 categories (values {0, 1, ..., 9})
      - Note that feature indices are 0-based

Decision Trees in Spark

- **Stopping criteria parameters**
  - `maxDepth`: maximum number of **layers** of a tree
    - **Deeper trees** are **more expressive** (potentially allowing higher accuracy), but they are also **more costly to train** and are **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**
  - `minInfoGain`: for a node to be split further, the split must **improve at least this much** (in terms of information gain)

Decision Trees in Spark

- **Other parameters**
  - `maxBins`: number of bins used when discretizing continuous features
  - `impurity`: impurity measure (discussed above) used to choose between candidate splits

Decision Trees in Spark

```
# CLASSIFICATION

from pyspark.mllib.tree import DecisionTree, DecisionTreeModel
from pyspark.mllib.util import MLUtils

# Load and parse the data file into an RDD of LabeledPoint.
data = MLUtils.loadLibSVMFile(sc, 'sample_libsvm_data.txt')

# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = data.randomSplit([0.7, 0.3])

# Train a DecisionTree model.
model = DecisionTree.trainClassifier(trainingData, numClasses=2, categoricalFeaturesInfo={},
                                     impurity='gini', maxDepth=5, maxBins=32)

# Evaluate model on test instances and compute test error
predictions = model.predict(testData.map(lambda x: x.features))
labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)
testErr = labelsAndPredictions.filter(lambda lp: lp[0] != lp[1]).count() / float(testData.count())
print('Test Error = ' + str(testErr))
print('Learned classification tree model:')
print(model.toDebugString())
```

Ensemble Methods

- An **ensemble** method is a learning algorithm which creates a model composed of **a set of** other **base models**
- MLLib supports two major ensemble algorithms: **Gradient Boosted Trees** and **Random Forests**
- Random forests are **ensembles of decision trees**
- They combine many decision trees in order to **reduce the risk of overfitting**

Tree Ensembles in Spark

```
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils

# Load and parse the data file into an RDD of LabeledPoint.
data = MLUtils.loadLibSVMFile(sc, 'sample_libsvm_data.txt')

(trainingData, testData) = data.randomSplit([0.7, 0.3])

# Train a RandomForest model.
#  Empty categoricalFeaturesInfo indicates all features are continuous.
#  Note: Use larger numTrees in practice.
#  Setting featureSubsetStrategy="auto" lets the algorithm choose.
model = RandomForest.trainClassifier(trainingData, numClasses=2, categoricalFeaturesInfo={},
                                     numTrees=3, featureSubsetStrategy="auto",
                                     impurity='gini', maxDepth=4, maxBins=32)

# Evaluate model on test instances and compute test error
predictions = model.predict(testData.map(lambda x: x.features))
labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)
testErr = labelsAndPredictions.filter(lambda lp: lp[0] != lp[1]).count() / float(testData.count())
print('Test Error = ' + str(testErr))
print('Learned classification forest model:')
print(model.toDebugString())
```