DDAM
DATA MINING WITH SPARK (MLLIB)

Docente: Patrizio Dazzi

WHAT IS MLLIB

MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

ML Algorithms: for clustering, classification, regression and collaborative filtering.

Featurization: feature extraction, transformation, dimensionality reduction, and selection

Persistence: saving and load algorithms, models, and Pipelines

Utilities: linear algebra, statistics, data handling, etc.

https://spark.apache.org/docs/latest/mllib-guide.html

DATA TYPES

MLlib supports local vectors and matrices stored on a single machine, as well as distributed data structures backed by one or more RDDs.

MLlib recognizes the following types as dense vectors:

- NumPy's array
- Python's list, e.g., [1, 2, 3]

and the following as sparse vectors:

- MLlib's SparseVector.
- SciPy's csc_matrix with a single column

DATA TYPES

```
import numpy as np
import scipy.sparse as sps
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 SparseVector.
sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])
# Use a single-column SciPy csc_matrix as a sparse vector.
sv2 = sps.csc_matrix((np.array([1.0, 3.0]), np.array([0, 2]), np.array([0, 2])), shape=(3, 1))
```

LABELED POINTS

A labeled point is a local vector, either dense or sparse, associated with a label/response. In MLlib, 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.

```
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]))
```

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, ...

BASIC STATISTICS

It provides column summary statistics for RDD[Vector] through the function colStats available in Statistics.

```
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)
print(summary.mean())  # a dense vector containing the mean value for each column
print(summary.variance())  # column-wise variance
print(summary.numNonzeros())  # number of nonzeros in each column
```

CORRELATION

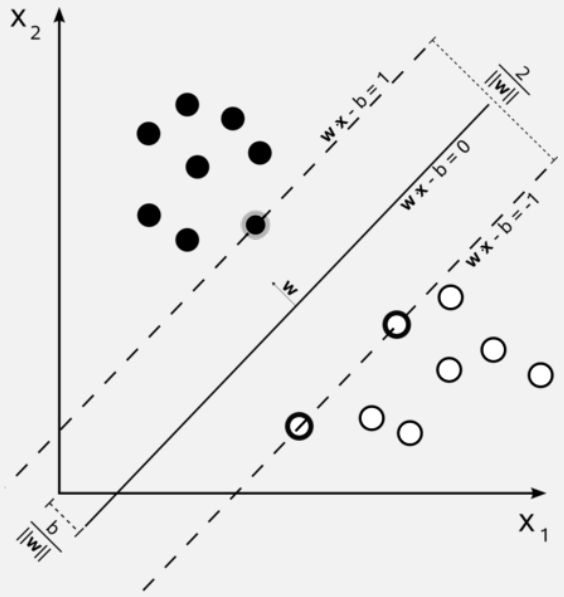
Calculating the correlation between two series of data is a common operation in Statistics.
In spark.mllib we provide the flexibility to calculate pairwise correlations among many series. The supported correlation methods are currently Pearson's and Spearman's correlation.

```
from pyspark.mllib.stat import Statistics

seriesX = sc.parallelize([1.0, 2.0, 3.0, 3.0, 5.0])  # a series
# seriesY must have the same number of partitions and 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

# calculate 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"))
```

LINEAR SUPPORT VECTOR MACHINES (SVMS)

The linear SVM is a standard method for large-scale classification tasks. It is a linear method which tries to separate with a "line" the data in order to reduce the error

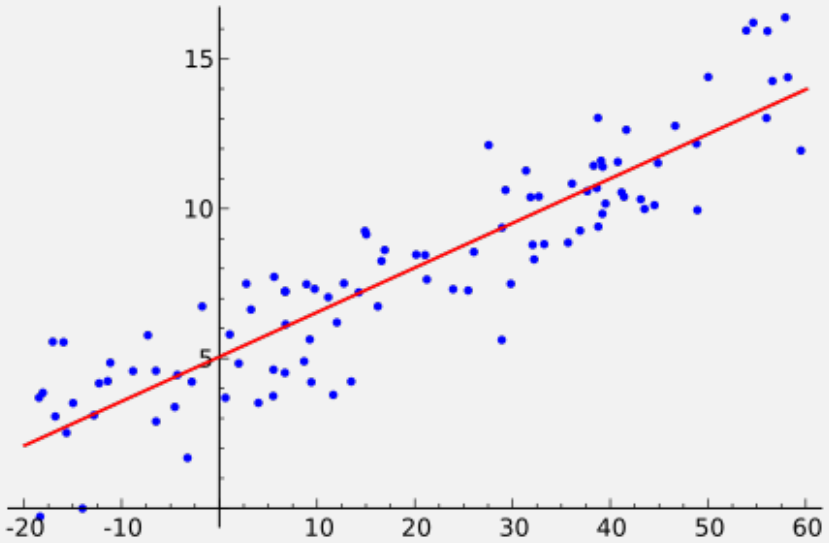
```
from pyspark.mllib.classification import SVMWithSGD, SVMModel
from pyspark.mllib.regression import LabeledPoint

# 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("data/mllib/sample_svm_data.txt")
parsedData = data.map(parsePoint)

# Build the model
model = SVMWithSGD.train(parsedData, iterations=100)

# 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, "target/tmp/pythonSVMWithSGDModel")
sameModel = SVMModel.load(sc, "target/tmp/pythonSVMWithSGDModel")
```



REGRESSION

In statistics, linear regression is a linear approach for modelling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X.

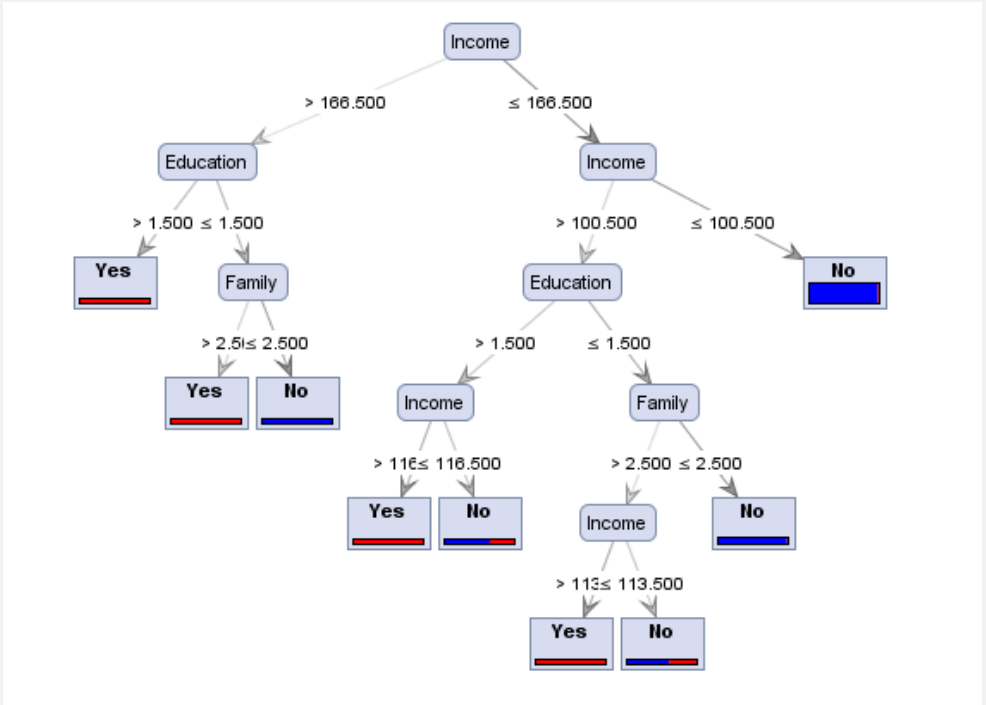
```
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD, LinearRegressionModel

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

data = sc.textFile("data/mllib/ridge-data/lpsa.data")
parsedData = data.map(parsePoint)

# Build the model
model = LinearRegressionWithSGD.train(parsedData, iterations=100, step=0.00000001)

# Evaluate the model on training data
valuesAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
MSE = valuesAndPreds \
    .map(lambda vp: (vp[0] - vp[1])**2) \
    .reduce(lambda x, y: x + y) / valuesAndPreds.count()
print("Mean Squared Error = " + str(MSE))

# Save and load model
model.save(sc, "target/tmp/pythonLinearRegressionWithSGDModel")
sameModel = LinearRegressionModel.load(sc, "target/tmp/pythonLinearRegressionWithSGDModel")
```



DECISION TREE

A decision tree is a structure that includes a **root** node, branches, and leaf nodes. Each internal node denotes a test on an attribute, each branch denotes the outcome of a test, and each leaf node holds a class label.

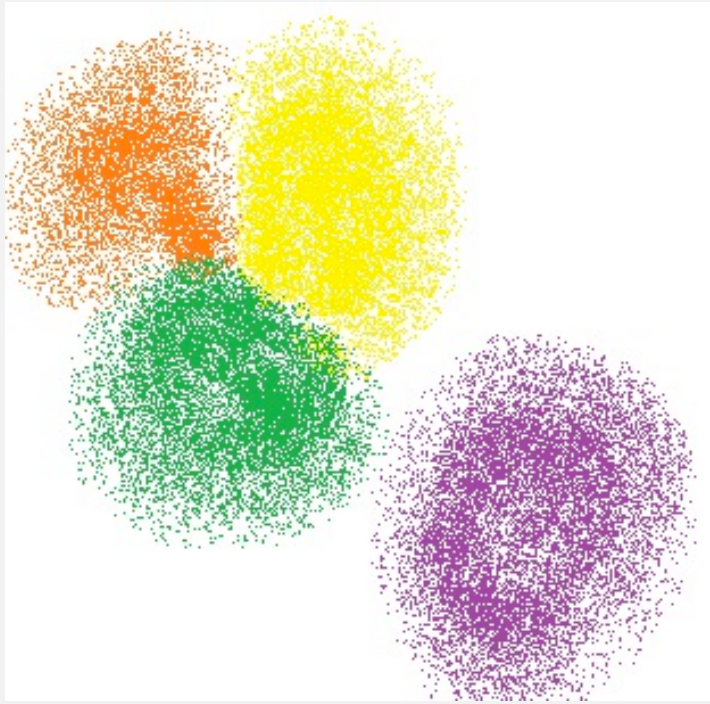
```
from pyspark.mllib.tree import DecisionTree, DecisionTreeModel
from pyspark.mllib.util import MLUtils

# Load and parse the data file into an RDD of LabeledPoint.
data = MLUtils.loadLibSVMFile(sc, 'data/mllib/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.
#  Empty categoricalFeaturesInfo indicates all features are continuous.
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())

# Save and load model
model.save(sc, "target/tmp/myDecisionTreeClassificationModel")
sameModel = DecisionTreeModel.load(sc, "target/tmp/myDecisionTreeClassificationModel")
```



CLUSTERING (K-MEANS)

Clustering is an unsupervised learning problem whereby we aim to group subsets of entities with one another based on some notion of similarity.

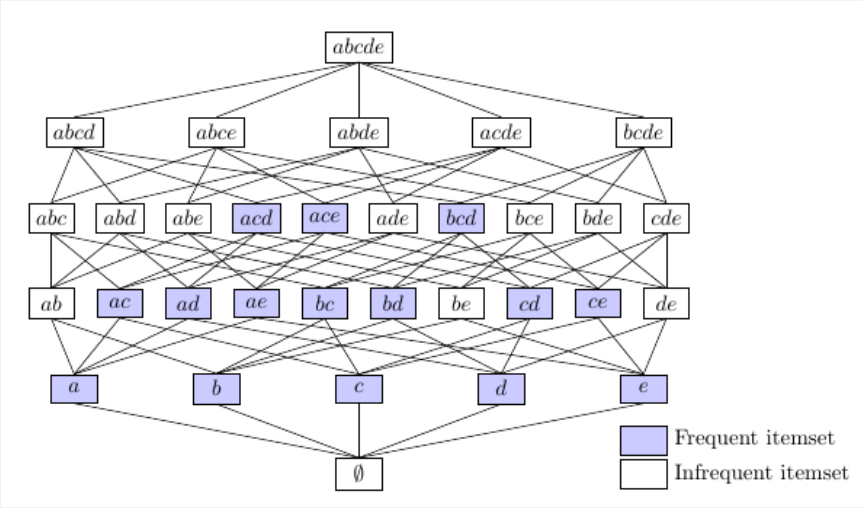
```
from numpy import array
from math import sqrt

from pyspark.mllib.clustering import KMeans, KMeansModel

# Load and parse the data
data = sc.textFile("data/mllib/kmeans_data.txt")
parsedData = data.map(lambda line: array([float(x) for x in line.split(' ')]))

# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations=10, initializationMode="random")

# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(point):
    center = clusters.centers[clusters.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

WSSSE = parsedData.map(lambda point: error(point)).reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WSSSE))

# Save and load model
clusters.save(sc, "target/org/apache/spark/PythonKMeansExample/KMeansModel")
sameModel = KMeansModel.load(sc, "target/org/apache/spark/PythonKMeansExample/KMeansModel")
```



FREQUENT PATTERN MININIG (FP-GROWTH)

Given a dataset of transactions, the first step of FP-growth is to calculate item frequencies and identify frequent items. The second step of FP-growth uses a suffix tree (FP-tree) structure to encode transactions without generating candidate sets explicitly, which are usually expensive to generate. After the second step, the frequent itemsets can be extracted from the FP-tree.

```
from pyspark.mllib.fpm import FPGrowth

data = sc.textFile("data/mllib/sample_fpgrowth.txt")
transactions = data.map(lambda line: line.strip().split(' '))
model = FPGrowth.train(transactions, minSupport=0.2, numPartitions=10)
result = model.freqItemsets().collect()
for fi in result:
    print(fi)
```



WORD2VEC

Word2Vec computes distributed vector representation of words. The main advantage of the distributed representations is that similar words are close in the vector space, which makes generalization to novel patterns easier and model estimation more robust.

```
from pyspark.mllib.feature import Word2Vec

inp = sc.textFile("data/mllib/sample_lda_data.txt").map(lambda row: row.split(" "))

word2vec = Word2Vec()
model = word2vec.fit(inp)

synonyms = model.findSynonyms('1', 5)

for word, cosine_distance in synonyms:
    print("{}: {}".format(word, cosine_distance))
```