Table of Contents

Exploring the internet on hot topic i.e. Machine Learning, I found tutorials created by Microsoft about their SQL Server Data Mining Extensions. Microsoft provides AdventureWorksDW2012 sample database under Microsoft Public License and uses it in their data mining tutorials. Some time ago it inspired me to build and test more general and presumably more scalable solutions (of course, results of computations should be comparable to some level) and evaluate some of MLLib algorithms that are available in Apache Spark.

Dataframe (and Dataset) based Machine Learning API

- Scala example of Decision Tree classification algorithm used for prediction of prospective customer behavior

Scala example of Decision Tree classification algorithm used for prediction of prospective customer behavior

To properly run following example some prerequisites are required:

- AdventureWorks database,
- jdbc driver for Microsoft SQL Server.

After downloading, database should be installed, and driver should be copied e.g. to jars folder to be further distributed to Spark cluster.

User with limited DB rights can be created with this script:

```
CREATE LOGIN [aw-user] WITH PASSWORD = 'aw-pass'
CREATE USER [aw-user] FOR LOGIN [aw-user]
GRANT SELECT ON vTargetMail TO [aw-user]
GRANT SELECT ON ProspectiveBuyer TO [aw-user]
```

Jdbc URL for Express (free version of Microsoft SQL Server) can look like this one:

```
val url = "jdbc:sqlserver://zxspectrum\\SQL2008R2EXPRESS;databas
eName=AdventureWorksDW2008R2;user=aw-user;password=aw-pass"
```

SQL query that provides input about customers (with information which of them bought a bicycle, stored in column BikeBuyer):

```
val query = "(SELECT CustomerKey, Age, BikeBuyer, CommuteDistanc
e, EnglishEducation, Gender, HouseOwnerFlag, MaritalStatus, Numb
erCarsOwned, NumberChildrenAtHome, EnglishOccupation, Region, To
talChildren, YearlyIncome FROM dbo.vTargetMail) as bikebuyers"
```

Connection properties should contain class name of Microsoft jdbc driver. Along with jdbc url, and query this is enough to load data into Dataframe:

```
    val connectionProperties = new Properties()
    connectionProperties.put("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
    val bikeBuyers = spark.read.jdbc(url, queryBikeBuyers, connectionProperties)
```

Show function can be used on this dataframe to sample data, or printSchema to print schema in nice format to the console.

In Spark 2 the main concept of ML workflow is a Pipeline that chains multiple Transformers and Estimators.

Here the pipeline is composed of RFormula, VectorIndexer, DecisionTreeClassifier transformers and estimators.

RFormula to select subset of columns and produce a vector of features and a column of label, VectorIndexer to index categorical features in intermediate dataset and DecisionTreeClassifier to create learning model.

```
val rformula = new RFormula()
  .setFormula("BikeBuyer ~ EnglishEducation + Gender + HouseOwne
rFlag + MaritalStatus + NumberCarsOwned + NumberChildrenAtHome +
 EnglishOccupation + TotalChildren + YearlyIncome ")
  .setFeaturesCol("features")
  .setLabelCol("label")

val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(2)

val dtree = new DecisionTreeClassifier()
  .setLabelCol("label")
  .setFeaturesCol("indexedFeatures")
  .setMaxBins(34)
  .setMaxDepth(20)
  .setImpurity("entropy")

val pipeline = new Pipeline().setStages(Array(rformula, featureI
ndexer, dtree))

val model = pipeline.fit(bikeBuyers)
```

To check how trained model works dataset of potential customers is to be used. SQL query that provides list of potential customers, as well as some characteristics about the customers:

```
val queryProspectiveBuyers = """(SELECT 0 as BikeBuyer, LastName
, FirstName, MaritalStatus, Gender, YearlyIncome, TotalChildren,
 NumberChildrenAtHome,
    EnglishEducation = CASE Education
      WHEN 'High Schoo' THEN 'High School'
      WHEN 'Partial Hi' THEN 'Partial High School'
      WHEN 'Partial Co' THEN 'Partial College'
      WHEN 'Graduate D' THEN 'Graduate Degree'
      WHEN 'Bachelors' THEN 'Bachelors'
    END, Occupation as EnglishOccupation, HouseOwnerFlag, Number
CarsOwned FROM dbo.ProspectiveBuyer) as prospectivebuyers"""
```

Case statement is used to fill gaps in dataset.

```
val prospectiveBuyers = spark.read.jdbc(url, queryProspectiveBuy
ers, connectionProperties)
val prospective = model.transform(prospectiveBuyers)

// Select example rows to display predicted as 1
prospective.select("LastName", "FirstName", "features").filter("
prediction == 1").show(false)
```

RDD based Machine Learning API

As time goes by, Spark community decided to move RDD based machine learning API to maintenance mode in Spark 2.0, and presumably remove it completly in version 3.0 later in the future. Lifetime of the following examples is limited then.

- Using classification to build model for predicting customer behavior
- Naive Bayes exploiting TF-IDF for Spam classification
- Using regression for predicting House prices
- Images classification with Convolutional Neural Networks, Deeplearning4j and Spark

Spark 1.x basic introduction

First thing to do in Apache Spark application is to create SparkConf object that contains information about application, which is farther required by SparkContext object as its constructor parameter.

It is important that the only one SparkContext may be active per JVM, and should be stopped before creating new context or before exiting current application.

Functions that return the configuration for different modes are kept in Application object. First for running application in local mode with as many worker threads as logical cores available:

```
  def configLocalMode(appName: String): SparkConf = {
    val conf = new SparkConf().setAppName(appName)
    conf.setMaster("local[*]")
    conf.set("spark.cassandra.connection.host", cassandraHost)
    conf
  }
}
```

Parameter which sets access to Apache Cassandra database is optional.

For running application in Standalone Cluster mode following function can be used:

```
def configStandaloneClusterMode(appName: String): SparkConf = {
    val conf = new SparkConf().setAppName(appName)
    conf.setMaster(sparkMaster)
    conf.setJars(Array("build/libs/spark-examples-1.0.jar"))
    conf.set("spark.cassandra.connection.host", cassandraHost)
    conf
  }
```

Variable sparkMaster should be pointing at existing Spark Standalone cluster.

To leverage existing Hadoop infrastructure Spark application can be executed on YARN. According to documentation two options are available: yarn-client and yarn-cluster. The most noticeable difference between them is whether you want to receive computational results on your driver application or not. The whole thing is best described at Cloudera blog. It seems that executing on yarn any Spark application can be done either by using spark-submit script or SparkLaucher class (available since version 1.4). SparkLaucher uses java ProcessBuilder class to execute spark-submit script in separate process. It is convenient to read streams connected to created process outputs. Small utility class for doing this task can look like that:

```
class RunnableInputStreamReader(is: InputStream, name: String) e
xtends Runnable {

  val reader = new BufferedReader(new InputStreamReader(is))

  def run() = {
    var line = reader.readLine();
    while (line != null) {
      System.out.println(line);
      line = reader.readLine();
    }
    reader.close();
  }

}
```

To run Spark in yarn-client or yarn-cluster mode, SPARK_HOME and HADOOP_CONF_DIR or YARN_CONF_DIR variables must be set. Environment variables can also be provided in SparkLauncher class' contructor parameter or in case of SPARK_HOME through method setSparkHome. Utility class executing Spark application on existing Hadoop cluster can look like that:

```
object YarnLauncher {

  val mode = "yarn-client"
  val mainClass = "examples.regression.HousePricesPrediction"

  def main(args: Array[String]): Unit = {
    val launcher = new SparkLauncher()
      .setAppResource("build/libs/spark-examples-1.0.jar")
      .setMainClass(mainClass)
      .setMaster(mode)
      .launch();


    val tf = Executors.defaultThreadFactory()
    tf.newThread(new RunnableInputStreamReader(launcher.getInput
Stream(), "input")).start()
    tf.newThread(new RunnableInputStreamReader(launcher.getError
Stream(), "error")).start()

    println("Executing ...")
    val exitCode = launcher.waitFor()
    println(s"Finished, exit code: $exitCode")

  }
}
```

When Spark works in standalone mode, files should be located on network storage available for every Spark's workers or copied to exactly the same location on every node. Using Spark on Yarn requires putting data on HDFS or some other distributed storage like S3. Reading local file go as follows:

```
def localFile(fileName: String): (SparkContext => RDD[String]) =
 sc => {
    sc.textFile("data/" + file)
}
```

Using HDFS requires Apache Hadoop configured and run. The only difference in code is that instead of providing file path, HDFS URL is to be supplied. 192.168.1.34:19000 reflects my local network Hadoop Cluster configuration, so it ought to be replaced with some alternative.

```
def hdfsFile(fileName: String): (SparkContext => RDD[String]) =
sc => {
    sc.textFile("hdfs://192.168.1.34:19000/spark/" + fileName)
}
```

Using classification to build model for predicting customer behavior

Here are notes from examining some classification algorithms available in Spark's MLLib with data excerpted from AdventureWorksDW2012 sample database.

Data is copied into bike-buyers.txt file, which you can find on github.

This file contains following information about customers of fictitious Adventure Works Cycles multinational manufacturing company:

- CustomerKey – internal key assigned to every customer
- Age – customer's age,
- BikeBuyer – flag indicating that customer was bicycle buyer,
- CommuteDistance – working day commute distance,
- EnglishEducation –level of education,
- Gender – customer's gender,
- HouseOwnerFlag - whether customer owns a house or not,
- MaritalStatus - marital status,
- NumberCarsOwned - number of cars owned by customer,
- NumberChildrenAtHome – number of children on education,
- Occupation – customer's occupation,
- Region – region of living,
- TotalChildren - how many children customer has in general,
- YearlyIncome – customer's salary.

Spark's MLLib supervised classification algorithms use data structure called LabeledPoint, which is a local vector, dense in our case, associated with a label/response.

In general, variables (and data) either represent measurements/values on some continuous scale, or represent information about some categorical or discrete characteristics.

For example, incomes of customers represent continuous variable; however, a person's gender, commute distance, occupation, or marital status are categorical or discrete variables: either a person is male or female, single or married, etc.

Some variables could be considered in either way. For example, a customer's age may be considered a continuous variable, or discrete variable with finite categories, which size can be calculated from data provided.

All examples are written in Scala, and most of them has also Java 8 versions.

They can be run in Local or in Standalone Cluster mode. Customer’s data can be loaded from many different sources.

Here, following options are taken into account:

- Local file,
- HDFS,
- Cassandra.

To build project Gradle is used.

Scala example of using Decision Tree algorithm

After loading data into RDD of Strings, conversion into LabeledPoint data structure can be prepared. For binary classification, labels should be negative or positive, represented by 0 or 1. Categorical features ought to be converted to numeric values 0, 1, 2 and so on. In this case, BikeBuyer flag would serve as Label, and all the rest would compose features vector. Customer Key doesn't play any real decision role but helps prevent model overfitting.

Using case class that reflects raw data can make conversion into LabeledPoints a bit easier:

```
case class BikeBuyerModel(customerKey: Int,
                          age: Int,
                          bikeBuyer: Int,
                          commuteDistance: String,
                          englishEducation: String,
                          gender: String,
                          houseOwnerFlag: Int,
                          maritalStatus: String,
                          numberCarsOwned: Int,
                          numberChildrenAtHome: Int,
                          englishOccupation: String,
                          region: String,
                          totalChildren: Int,
                          yearlyIncome: Float)
    extends LabeledPointConverter {

  def label() = bikeBuyer.toDouble
  def features() = BikeBuyerModel.convert(this)
}
```

LabeledPointConverter is trait that could be reused. Case class build with this trait must provide implementation of label and feature.

```
trait LabeledPointConverter {
  def label(): Double
  def features(): Vector
  def toLabeledPoint() = LabeledPoint(label(), features())
}
```

BikeBuyerModel companion object is overridden and together with apply method it provides method for conversion to Vector and marking categorical features.

```
object BikeBuyerModel {

  def apply(row: Array[String]) = new BikeBuyerModel(
    row(0).toInt, row(1).toInt, row(2).toInt, row(3),
    row(4), row(5), row(6).toInt,
    row(7), row(8).toInt, row(9).toInt,
    row(10), row(11), row(12).toInt, row(13).replaceFirst(",", "
.").toFloat)

  def categoricalFeaturesInfo() = {
    Map[Int, Int](2 -> 5, 3 -> 5, 4 -> 2, 6 -> 2, 9 -> 5, 10 -> 3
)
  }

  def convert(model: BikeBuyerModel) = Vectors.dense(
    model.customerKey.toDouble,
    model.age.toDouble,
    model.commuteDistance match {
      case "0-1 Miles"  => 0d
      case "1-2 Miles"  => 1d
      case "2-5 Miles"  => 2d
      case "5-10 Miles" => 3d
      case "10+ Miles"  => 4d
    },
    model.englishEducation match {
      case "High School"         => 0d
      case "Partial High School" => 1d
      case "Partial College"     => 2d
      case "Graduate Degree"     => 3d
      case "Bachelors"           => 4d
```

```
    },
    model.gender match {
      case "M" => 0d
      case "F" => 1d
    },
    model.houseOwnerFlag.toDouble,
    model.maritalStatus match {
      case "S" => 0d
      case "M" => 1d
    },
    model.numberCarsOwned.toDouble,
    model.numberChildrenAtHome.toDouble,
    model.englishOccupation match {
      case "Professional"   => 0d
      case "Clerical"       => 1d
      case "Manual"         => 2d
      case "Management"     => 3d
      case "Skilled Manual" => 4d
    },
    model.region match {
      case "North America" => 0d
      case "Pacific"       => 1d
      case "Europe"        => 2d
    },
    model.totalChildren.toDouble,
    model.yearlyIncome)
}
```

Now, acquiring data in format required by Spark is quite easy:

```
val data = bbFile.map { row => BikeBuyerModel(row.split("\\t")).
toLabeledPoint }
```

After that, data can be split into train and test parts, to conform cross-validation method, when model is trained with part of dataset and its performance is evaluated with another part:

```
val Array(train, test) = data.randomSplit(Array(.9, .1))
```

It seems to be a good moment to cache data for further reuse:

```
train.cache()
test.cache()
```

Now Spark's classification decision tree algorithm can be trained:

```
val numClasses = 2
val impurity = "entropy"
val maxDepth = 20
val maxBins = 24

val dtree = DecisionTree.trainClassifier(train, numClasses, Bike
BuyerModel.categoricalFeaturesInfo(), impurity, maxDepth, maxBin
s)
```

Trained model can be used for prediction of whether potential customer is going to buy a bicycle or not:

```
    test.take(5).foreach {
      x => println(s"Predicted: ${dtree.predict(x.features)}, ac
tual value: ${x.label}")
    }
```

Typical response of prediction for 5 top records from test dataset can look like this:

```
Predicted: 1.0, Label: 1.0
Predicted: 1.0, Label: 1.0
Predicted: 0.0, Label: 0.0
Predicted: 0.0, Label: 1.0
Predicted: 1.0, Label: 1.0
```

To answer questions what is the real performance of this model, what is its ability to provide correct responses, some metrics has to be evaluated. To further check performance of created model, test data can be used to collect predictions and expected values:

```
    val predictionsAndLabels = test.map {
      point => (dtree.predict(point.features), point.label)
    }
```

In general, predictions of each data point from bike buyers dataset can be assigned to one of four categories:

- True Positive, when buyer is predicted as buyer,
- True Negative, when not buyer is predicted as not buyer,
- False Positive, when not buyer is predicted as buyer,
- False Negative, when buyer predicted as not buyer.

Calculations can look like that:

```
val (tp, tn, fp, fn) = predictionsAndLabels.aggregate((0, 0, 0, 0
))(
    seqOp = (t, pal) => {
        val (tp, tn, fp, fn) = t
        (if (pal._1 == pal._2 && pal._2 == 1.0) tp + 1 else tp,
         if (pal._1 == pal._2 && pal._2 == 0.0) tn + 1 else tn,
         if (pal._1 == 1.0 && pal._2 == 0.0) fp + 1 else fp,
         if (pal._1 == 0.0 && pal._2 == 1.0) fn + 1 else fn)
    },
    combOp = (t1, t2) => (t1._1 + t2._1, t1._2 + t2._2, t1._3 +
t2._3, t1._4 + t2._4))
```

Based on above, some other measures can be entered, e.g. in form of utility class:

```
class Stats(val tp: Int, val tn: Int, val fp: Int, val fn: Int)
{
  val TPR = tp / (tp + fn).toDouble
  val recall = TPR
  val sensitivity = TPR
  val TNR = tn / (tn + fp).toDouble
  val specificity = TNR
  val PPV = tp / (tp + fp).toDouble
  val precision = PPV
  val NPV = tn / (tn + fn).toDouble
  val FPR = 1.0 - specificity
  val FNR = 1.0 - recall
  val FDR = 1.0 - precision
  val ACC = (tp + tn) / (tp + fp + fn + tn).toDouble
  val accuracy = ACC
  val F1 = 2 * PPV * TPR / (PPV + TPR).toDouble
  val MCC = (tp * tn - fp * fn).toDouble / math.sqrt((tp + fp).t
oDouble * (tp + fn).toDouble * (fp + tn).toDouble * (tn + fn).to
Double)
}
```

True positive rate TPR, called also recall or sensitivity is defined as the number of samples correctly predicted as belonging to the positive class (true positives) divided by the total number of elements that actually belong to the positive class (i.e. the sum of true positives and false negatives, which are items which were not predicted as belonging to the positive class but should have been)

True negative rate TNR, called also specificity is measure of samples correctly predicted as belonging to negative class divided by the total number of elements actually belonging to the negative class.

Positive predictive value, called also precision is the proportion of samples correctly predicted as belonging to the positive class (true positives) divided by the total number of elements predicted as belonging to the positive class (i.e. the sum of true positives and false positives, which are items incorrectly predicted as belonging to the class).

Negative predictive value is the proportion of samples correctly predicted as negative divided by the total number of true negative results.

False positive rate, called also fall-out is closely related to specificity and is equal to 1 - specificity.

False negative rate is closely related to sensitivity and is equal to 1 - recall.

False discovery rate is closely related to precision and is equal to 1 - precision.

Accuracy is the fraction of samples that the classifier correctly predicted (both positive and negative) to the total number of samples in test data set. Accuracy makes no distinction between classes; correct answers for both positive and negative cases are treated equally. When cost of misclassification is different or if there are a lot more test data of one class than the other, then accuracy would give a very distorted picture, because the class with more examples will dominate the statistic. Bike buyers data set is balanced, it contains 9132 positive and 9352 negative examples.

In contrast to accuracy, Matthews correlation coefficient MCC is generally regarded as a balanced measure of the quality of binary classifications if the classes are of very different sizes.

Mutual combination of precision and recall is called F-Measure or F-Score 2PR/(P+R) and is another metric useful for rating classification accuracy and comparing different classification models or algorithms. F-Score equal to 1 represents model with perfect precision and sensitivity whilst F-Score equal to 0 is the opposite.

```
val stats = Stats(confusionMatrix(predictionsAndLabels))
println(stats.toString)
```

Example output can look like that:

TP: 816.0, TN: 807.0, FP: 123.0, FN: 114.0
TPR (recall/sensitivity): 0.8774193548387097
TNR (specificity): 0.867741935483871
PPV (precision): 0.8690095846645367
NPV: 0.8762214983713354

FPR (fall-out): 0.13225806451612898
FNR: 0.1225806451612903
FDR: 0.1309904153354633
ACC (accuracy): 0.8725806451612903
F1 (F-Measure): 0.8731942215088282
MCC (Matthews correlation coefficient): 0.745196185862156
Some of described metrics and some additional are available in Spark's binary metrics evaluator:

```
    val metrics = new BinaryClassificationMetrics(predictionsAnd
Labels)
```

The ROC curve (receiver operating characteristic) shows the sensitivity of the classifier by plotting the rate of true positives to the rate of false positives.

In other words, the perfect classifier that makes no mistakes would hit a true positive rate of 1, without incurring any false positives.

Calculating area under ROC curve allows to express this relation as single number, which with caution can be used for model comparison.

High value of AUC can be treated as representation of good classification model, which reflects a lot of space under curve, when it goes to point of perfect classification.

Low value of AUC is the opposite.

In Spark ROC curve is available in form of RDD containing (false positive rate, true positive rate) with (0.0, 0.0) prepended and (1.0, 1.0) appended to it, and area under ROC curve is available as single value.

```
val roc = metrics.roc
val auROC = metrics.areaUnderROC
```

Precision-Recall curve (Spark returns it in reverse order) is available along with area under PR curve.

```
    val PR = metrics.pr
    val auPR = metrics.areaUnderPR
```

Values of precision, recall and f-measure can be also read from BinaryClassificationMetrics object. In this case, Spark calculates them for both classes separately:

```
val precision = metrics.precisionByThreshold
precision.foreach { case (t, p) => println(s"Threshold: $t, Prec
ision: $p") }

val recall = metrics.recallByThreshold
recall.foreach { case (t, r) => println(s"Threshold: $t, Recall:
$r") }

val f1Score = metrics.fMeasureByThreshold
f1Score.foreach { case (t, f) => println(s"Threshold: $t, F-scor
e: $f, Beta = 1") }
```

Example output can look like that:

Threshold: 1.0, Precision: 0.8690095846645367
Threshold: 0.0, Precision: 0.5
Threshold: 1.0, Recall: 0.8774193548387097
Threshold: 0.0, Recall: 1.0
Threshold: 1.0, F-score: 0.8731942215088282, Beta = 1
Threshold: 0.0, F-score: 0.6666666666666666, Beta = 1
Threshold: 1.0, F-score: 0.8731942215088282, Beta = 0.5
Threshold: 0.0, F-score: 0.6666666666666666, Beta = 0.5
Area under PR (precision-recall curve) = 0.9038596310419458
Area under ROC (Receiver Operating Characteristic) = 0.8725806451612903

Sample statistic calculations

Suppose after random split, test data set contains 1774 samples, 875 positive and 899 negative. If trained classifier predicts 855 samples as true with 751 actually true, and predicted 919 as false with 795 being false, then Confusion matrix will be:

	Predicted as negative	Predicted as positive
Labelled as negative	795.0 (TN)	104.0 (FP)
Labelled as positive	124.0 (FN)	751.0 (TP)

Overall accuracy is (751 + 795)/(875 + 899) = 1546/1774 = 0.871,

and misclassification for class 0 and 1 is 104/899 = 0,115 and 124/875 = 0,145 accordingly.

For positive class Precision equals to 751/855 = 0,8783 and Recall equals to 751/875 = 0,8582.

MCC is equal to 0,7430.

It can be expected, that for different numbers of examples per class, the average per-class accuracy will be different from the overall accuracy.

Bike buyers data set is well balanced, it contains 9132 positive and 9352 negative examples, so the accuracy separately calculated for each class is 795/899 = 0,884 and 751/875 = 0,858, and average value 0,871 is exactly the same as overall accuracy.

If the classes were not balanced, then the accuracy would give a very distorted picture, because the class with more examples will dominate the statistic and both the average and the individual per-class accuracy should be checked.

Now suppose data set contains 1575 positive and 199 negative samples. If classifier predicted 1451 as true with 1555 actual true, and 95 as false with 119 actual false, then Confusion matrix will be:

	Predicted as negative	Predicted as positive
Labeled as negative	95.0 (TN)	104.0 (FP)
Labeled as positive	24.0 (FN)	1451.0 (TP)

Overall accuracy is (1451 + 95)/(1575 + 199) = 1546/1774 = 0.8714, and misclassification for class 0 and 1 is 104/199 = 0,5226 and 24/1575 = 0,0152 accordingly.

The accuracy calculated for each class is 95/199 = 0,4773 and 1451/1575 = 0,9212 and average of them is 0,6956.

For positive class Precision equals to 1451/1555 = 0,9331 and Recall equals to 1451/1475 = 0,9837, but for negative class Precision equals to 95/119 = 0,7983 and Recall equals to 95/199 = 0,4773.

In this case MCC is equal to 0,5808.

Java 8 example of classification using Decision Tree

There are some differences in using Spark from Java. The most important is that instead of SparkContext its java friendly version called JavaSparkContext must be used. Methods of this class returns java wrappers of RDD objects (JavaRDD) and works with Java collections. To create JavaSparkContext it is convenient to use try-with-resources statement:

```
  try (JavaSparkContext sc = new JavaSparkContext(configLocalMod
e())) {…}
```

JavaSparkContext class implements Closeable interface and calls stop method:

```
  override def close(): Unit = stop()
```

Functions to load data from local storage, HDFS and Cassandra go as follow.

Again, to run on cluster, local file should be located on network storage available for every Spark's workers or copied to exactly the same location on every node.

```
public static JavaRDD<String> localFile(JavaSparkContext sc) {
    return sc.textFile("data/bike-buyers");
}
```

For Hadoop, HDFS URL must be provided:

```
public static JavaRDD<String> hdfsFile(JavaSparkContext sc) {
    return sc.textFile("hdfs://192.168.1.15:9000/spark/bike-buye
rs");
}
```

Access to Cassandra is a bit more complicated. First map function converts key value pair into Spark's Tuple2 containing position of column and value converted to String. Then Stream is sorted by column position. Second map function takes String value from Tuple2, which finally is reduced to single line.

```
import static com.datastax.spark.connector.japi.CassandraJavaUti
l.javaFunctions;

public static JavaRDD<String> cassandraFile(JavaSparkContext sc)
{
    return javaFunctions(sc).cassandraTable("spark", "bike_buyer
s").map(
        row -> {
            return row.toMap().entrySet().stream().
            map(e ->
                new Tuple2<>(row.indexOf(e.getKey()), e.getValue
().toString())
            ).sorted((t1, t2) -> t1._1().compareTo(t2._1())).
            map(t -> t._2()).reduce((a, b) -> a + "\t" + b).get(
);
    });
  }
```

Data conversion into LabeledPoint data structure can be done with little help from class that reflects raw data structure, and makes conversion into LabeledPoints. Here Java is mixed with Scala, BikeBuyerModelJava implements Scala trait. Browsing Spark's source code shows many places with similar approach.

```
public class BikeBuyerModelJava implements LabeledPointConverter
{

    private final Integer customerKey;
    private final Integer age;
    private final Integer bikeBuyer;
    private final String commuteDistance;
    private final String englishEducation;
    private final String gender;
    private final Integer houseOwnerFlag;
```

```
    private final String maritalStatus;
    private final Integer numberCarsOwned;
    private final Integer numberChildrenAtHome;
    private final String englishOccupation;
    private final String region;
    private final Integer totalChildren;
    private final Float yearlyIncome;

    public BikeBuyerModelJava(Integer customerKey, Integer age,
Integer bikeBuyer, String commuteDistance,
            String englishEducation, String gender, Integer hous
eOwnerFlag, String maritalStatus,
            Integer numberCarsOwned, Integer numberChildrenAtHom
e, String englishOccupation, String region,
            Integer totalChildren, Float yearlyIncome) {
        super();
        this.customerKey = customerKey;
        this.age = age;
        this.bikeBuyer = bikeBuyer;
        this.commuteDistance = commuteDistance;
        this.englishEducation = englishEducation;
        this.gender = gender;
        this.houseOwnerFlag = houseOwnerFlag;
        this.maritalStatus = maritalStatus;
        this.numberCarsOwned = numberCarsOwned;
        this.numberChildrenAtHome = numberChildrenAtHome;
        this.englishOccupation = englishOccupation;
        this.region = region;
        this.totalChildren = totalChildren;
        this.yearlyIncome = yearlyIncome;
    }

    public BikeBuyerModelJava(String... row) {
        this(Integer.valueOf(row[0]), Integer.valueOf(row[1]), I
nteger.valueOf(row[2]), row[3], row[4], row[5], Integer
                .valueOf(row[6]), row[7], Integer.valueOf(row[8]
), Integer.valueOf(row[9]), row[10], row[11], Integer
                .valueOf(row[12]), Float.valueOf(row[13].replace
First(",", ".")));
    }
```

```
    @Override
    public LabeledPoint toLabeledPoint() {
        return new LabeledPoint(label(), features());
    }

    @Override
    public double label() {
        return bikeBuyer.doubleValue();
    }

    @Override
    public Vector features() {
        double[] features = new double[getClass().getDeclaredFie
lds().length - 1];
        features[0] = customerKey.doubleValue();
        features[1] = age.doubleValue();
        switch (commuteDistance) {
        case "0-1 Miles":
            features[2] = 0d;
            break;
        case "1-2 Miles":
            features[2] = 1d;
            break;
        case "2-5 Miles":
            features[2] = 2d;
            break;
        case "5-10 Miles":
            features[2] = 3d;
            break;
        case "10+ Miles":
            features[2] = 4d;
            break;
        default:
        }
        switch (englishEducation) {
        case "High School":
            features[3] = 0d;
            break;
        case "Partial High School":
```

```
            features[3] = 1d;
            break;
        case "Partial College":
            features[3] = 2d;
            break;
        case "Graduate Degree":
            features[3] = 3d;
            break;
        case "Bachelors":
            features[3] = 4d;
            break;
        default:
        }
        switch (gender) {
        case "M":
            features[4] = 0d;
            break;
        case "F":
            features[4] = 1d;
            break;
        default:
        }
        features[5] = houseOwnerFlag.doubleValue();
        switch (maritalStatus) {
        case "S":
            features[6] = 0d;
            break;
        case "M":
            features[6] = 1d;
            break;
        default:
        }
        features[7] = numberCarsOwned.doubleValue();
        features[8] = numberChildrenAtHome.doubleValue();
        switch (englishOccupation) {
        case "Professional":
            features[9] = 0d;
            break;
        case "Clerical":
            features[9] = 1d;
```

```
            break;
        case "Manual":
            features[9] = 2d;
            break;
        case "Management":
            features[9] = 3d;
            break;
        case "Skilled Manual":
            features[9] = 4d;
            break;
        default:
        }
        switch (region) {
        case "North America":
            features[10] = 0d;
            break;
        case "Pacific":
            features[10] = 1d;
            break;
        case "Europe":
            features[10] = 2d;
            break;
        default:
        }
        features[11] = totalChildren.doubleValue();
        features[12] = yearlyIncome;
        return Vectors.dense(features);
    }

    public static Map<Integer, Integer> categoricalFeaturesInfo()
{
        return new HashMap<Integer, Integer>() {
            private static final long serialVersionUID = 1L;
            {
                put(2, 5);
                put(3, 5);
                put(4, 2);
                put(6, 2);
                put(9, 5);
                put(10, 3);
```

```
            }
        };
    }
    //and getters
}
```

Now, data in format required by Spark can be acquired:

```
JavaRDD<LabeledPoint> data = bbFile.map(r ->
new BikeBuyerModelJava(r.split("\\t")).toLabeledPoint()
);
```

Splitting data set for training and testing also looks a bit different in Java:

```
  JavaRDD<LabeledPoint>[] split = data.randomSplit( new double[]
 { .9, .1 } );
  JavaRDD<LabeledPoint> train = split[0].cache();
  JavaRDD<LabeledPoint> test = split[1].cache();
```

Now classification model can be built:

```
  Integer numClasses = 2;
  String impurity = "entropy";
  Integer maxDepth = 20;
  Integer maxBins = 34;

  final DecisionTreeModel dtree = DecisionTree.trainClassifier(t
rain, numClasses, BikeBuyerModelJava.categoricalFeaturesInfo(),
impurity, maxDepth, maxBins);
```

After displaying first 5 predictions:

```
test.take(5).forEach(x -> {
  System.out.println(String.format("Predicted: %.1f, Label: %.1f"
, dtree.predict(x.features()), x.label()));
});
```

With sample output:

```
Predicted: 1,0, Label: 1,0
Predicted: 1,0, Label: 1,0
Predicted: 1,0, Label: 1,0
Predicted: 1,0, Label: 1,0
Predicted: 0,0, Label: 0,0
```

Metrics can be evaluated:

```
JavaPairRDD<Object, Object> predictionsAndLabels = test.mapToPai
r(
    p -> new Tuple2<Object, Object>(dtree.predict(p.features()),
 p.label())
);

Stats stats = Stats.apply(confusionMatrix(predictionsAndLabels.r
dd()));
System.out.println(stats.toString());

BinaryClassificationMetrics metrics = new BinaryClassificationMe
trics(predictionsAndLabels.rdd());
printMetrics(metrics);
```

Tuning Decision Tree algorithm

Spark decision tree classification algorithm takes some parameters that can significantly influence quality of created model. There is no simple theory that tells what values should be used. Answer can be found by iterations, and selecting model with best characteristics.

```
    val numClasses = 2
    val tuning =
      for (
        impurity <- Array("entropy", "gini");
        maxDepth <- Range(5, 25, 5);
        maxBins <- Range(10, 50, 2)
      ) yield {
        val model = DecisionTree.trainClassifier(train, numClass
es, BikeBuyerModel.categoricalFeaturesInfo(), impurity, maxDepth
, maxBins)
        val predictionsAndLabels = test.map {
          point => (model.predict(point.features), point.label)
        }
        val stats = Stats(confusionMatrix(predictionsAndLabels))
        val metrics = new BinaryClassificationMetrics(prediction
sAndLabels)
        val auPR = metrics.areaUnderPR()
        val auROC = metrics.areaUnderROC()
        ((impurity, maxDepth, maxBins), stats.MCC, stats.ACC, au
PR, auROC)
      }
    tuning.sortBy(_._2).reverse.foreach{
      x => println(x._1 + " " + x._2 + " " + x._3+ " " + x._3)
    }
```

Example output can look like that:

```
(gini,20,32) 0.7591468068014129 0.8795698924731182 0.9094022702677811, 0.8795698924731183
(gini,20,36) 0.7561452549554005 0.8779569892473118 0.9069982737027384 0.8779569892473119
(entropy,20,20) 0.7527160219196528 0.8763440860215054 0.9078155249224453 0.8763440860215054
(entropy,20,28) 0.7494662650097887 0.874731182795699 0.9058491511945511 0.874731182795699
(gini,20,28) 0.7473122599794635 0.8736559139784946 0.9053086334016228 0.8736559139784947
(gini,20,48) 0.7463211280086185 0.8731182795698925 0.9058083298117106 0.8731182795698925
```

Results is sorted by MCC, but it can be noticed that it is also almost perfectly sorted by accuracy and values of areas under ROC and PR curves. Now better decision tree parameters can be used to create better classification model.

Classification by using Ensembles of Classifiers

A classification ensemble model is the combination of two or more classification models. Random forest is an ensemble tree technique that builds a model with defined number of decision trees that is better classifier than average tree in the forest. So the whole performs better than any of its parts.

In Spark Random Forest model is created on decision tree algorithm with default voting strategy, where each tree votes on classification, and higher number of votes wins.

According to Spark documentation Random forests are one of the most successful machine learning models for classification and regression, with reduced risk of over-fitting. The algorithm injects randomness into the training process so that each decision tree is a bit different.

Scala program:

```
val sc = new SparkContext(configLocalMode)
val bbFile = localFile(sc)

val data = bbFile.map { row =>
  BikeBuyerModel(row.split("\\t")).toLabeledPoint
}

val Array(train, test) = data.randomSplit(Array(.9, .1), 102059L
)
train.cache()
test.cache()

val numClasses = 2
val numTrees = 10
val featureSubsetStrategy = "auto"
val impurity = "entropy"
val maxDepth = 20
val maxBins = 34
```

```
val model = RandomForest.trainClassifier(train, numClasses, Bike
BuyerModel.categoricalFeaturesInfo, numTrees, featureSubsetStrat
egy, impurity, maxDepth, maxBins)

test.take(5).foreach {
  x => println(s"Predicted: ${model.predict(x.features)}, Label:
${x.label}")
}

val predictionsAndLabels = test.map {
  point => (model.predict(point.features), point.label)
}

val stats = Stats(confusionMatrix(predictionsAndLabels))
println(stats.toString)

val metrics = new BinaryClassificationMetrics(predictionsAndLabe
ls)
printMetrics(metrics)

sc.stop()
```

Random Forest Java 8 example

Java 8 version on binary classification by Random Forest:

```
try (JavaSparkContext sc = new JavaSparkContext(configLocalMode(
))) {
    JavaRDD<String> bbFile = localFile(sc);

    JavaRDD<LabeledPoint> data = bbFile.map(r -> new BikeBuyerMo
delJava(r.split("\\t")).toLabeledPoint());
    JavaRDD<LabeledPoint>[] split = data.randomSplit(new double[
] { .9, .1 });
    JavaRDD<LabeledPoint> train = split[0].cache();
    JavaRDD<LabeledPoint> test = split[1].cache();

    Integer numClasses = 2;
    Integer numTrees = 10;
    String featureSubsetStrategy = "auto";
    String impurity = "entropy";
    Integer maxDepth = 20;
    Integer maxBins = 34;
    Integer seed = 12345;

    final RandomForestModel model = RandomForest.trainClassifier
(train, numClasses, BikeBuyerModelJava.categoricalFeaturesInfo()
, numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins,
seed);

    test.take(5).forEach(x -> {
        System.out.println(String.format("Predicted: %.1f, Label
: %.1f", model.predict(x.features()), x.label()));
    });

    JavaPairRDD<Object, Object> predictionsAndLabels = test.mapT
oPair(
        p -> new Tuple2<Object, Object>(model.predict(p.features
()), p.label())
    );
```

```
    Stats stats = Stats.apply(confusionMatrix(predictionsAndLabe
ls.rdd()));
    System.out.println(stats.toString());

    BinaryClassificationMetrics metrics = new BinaryClassificati
onMetrics(predictionsAndLabels.rdd());
    printMetrics(metrics);
}
```

Tuning Random Forest algorithm

The whole tuning program which tests Random Forest algorithm with different number of trees:

```
val sc = new SparkContext(configLocalMode)
val bbFile = localFile(sc)

val data = bbFile.map { row =>
  BikeBuyerModel(row.split("\\t")).toLabeledPoint
}

val Array(train, test) = data.randomSplit(Array(.9, .1), 102059L
)
train.cache()
test.cache()

val numClasses = 2
val featureSubsetStrategy = "auto"
val impurity = "entropy"
val maxDepth = 20
val maxBins = 34

val tuning = for (numTrees <- Range(2, 20)) yield {
  val model = RandomForest.trainClassifier(train, numClasses, Bi
keBuyerModel.categoricalFeaturesInfo, numTrees, featureSubsetStr
ategy, impurity, maxDepth, maxBins)
  val predictionsAndLabels = test.map {
    point => (model.predict(point.features), point.label)
  }
  val stats = Stats(confusionMatrix(predictionsAndLabels))
  val metrics = new BinaryClassificationMetrics(predictionsAndLa
bels)
  (numTrees, stats.MCC, stats.ACC, metrics.areaUnderPR, metrics.
areaUnderROC)
}
tuning.sortBy(_._2).reverse.foreach{
  x => println(x._1 + " " + x._2 + " " + x._3+ " " + x._4+ " " +
 x._5)
}

sc.stop()
```

Sample output sorted by Matthews correlation coefficient:

```
15 0.7710216770066503 0.885483870967742 0.9149414393859618 0.885
4838709677418<br>
12 0.7673416812762877 0.8833333333333333 0.9100277971969772 0.88
33333333333333<br>
13 0.7655913978494624 0.8827956989247312 0.9120967741935484 0.88
27956989247312<br>
14 0.7594366282306201 0.8795698924731182 0.9080105277365367 0.87
95698924731183<br>
9 0.759352282307894 0.8795698924731182 0.9113187437828619 0.8795
698924731182<br>
...
```

Naive Bayes exploiting TF-IDF for Spam classification

Following example is run on data set described as the SMS Spam Collection v.1. This is a set of SMS tagged messages that have been collected for SMS Spam research. It contains one set of SMS messages in English of 5,574 messages, tagged according being ham (legitimate) or spam. More details can be found on web page.

As classifier NaiveBayes is used. Every SMS message regardless of its label is transformed into features vector. The term frequency (TF) represents the number of occurrences of particular term within message. The inverse document frequency (IDF) represents frequency of term for the whole messages set. Term Frequency–Inverse Document Frequency that stands for TF-IDF is a product of these two statistics.

Scala version can look as follow:

```
    val hash = new HashingTF(numFeatures = 100000)
    val raw = sc.textFile("data/sms-labeled.txt").distinct().map
 {
      _.split("\\t+")
    }.map {
      a => (a(0), a(1).split("\\s+").map(_.toLowerCase()))
    }.map {
      t => (t._1, t._2, hash.transform(t._2))
    }.cache

    val idf = new IDF().fit(raw.map(_._3))
    val data = raw.map {
      t => LabeledPoint(if (t._1 == "spam") 1 else 0, idf.transf
orm(t._3))
    }

    val Array(train, test) = data.randomSplit(Array(.8, .2), 102
059L)
    val model = NaiveBayes.train(train)
    evaluateModel(model, test)
```

RDD named raw could be kept in simpler form but, some additional computations are provided.

First of all, the helper method for model evaluation:

```
  def evaluateModel(model: NaiveBayesModel, test: RDD[LabeledPoi
nt]) = {
    val predict = model.predict(test.map(_.features))

    test.take(5).foreach {
      x => println(s"Predicted: ${model.predict(x.features)}, La
bel: ${x.label}")
    }

    val predictionsAndLabels = test.map {
      point => (model.predict(point.features), point.label)
    }

    val stats = Stats(confusionMatrix(predictionsAndLabels))
    println(stats.toString)

    val metrics = new BinaryClassificationMetrics(predictionsAnd
Labels)
    printMetrics(metrics)
  }
```

Typical response of prediction for 5 top records from test dataset can look like this:

```
Predicted: 1.0, Label: 1.0
Predicted: 1.0, Label: 1.0
Predicted: 0.0, Label: 0.0
Predicted: 0.0, Label: 0.0
Predicted: 0.0, Label: 0.0
```

Output after model evaluation can look like that:

```
TP: 123.0, TN: 879.0, FP: 24.0, FN: 12.0
TPR (recall/sensitivity): 0.9111111111111111
TNR (specificity): 0.973421926910299
PPV (precision): 0.8367346938775511
NPV: 0.9865319865319865
FPR (fall-out): 0.02657807308970095
FNR: 0.0888888888888889
FDR: 0.16326530612244894
ACC (accuracy): 0.9653179190751445
F1 (F-Measure): 0.8723404255319148
MCC (Matthews correlation coefficient): 0.8533502082524206
Threshold: 1.0, Precision: 0.8367346938775511
Threshold: 0.0, Precision: 0.13005780346820808
Threshold: 0.0, Recall: 1.0
Threshold: 1.0, Recall: 0.9111111111111111
Threshold: 0.0, F-score: 0.23017902813299232, Beta = 1
Threshold: 1.0, F-score: 0.8723404255319148, Beta = 1
Threshold: 1.0, F-score: 0.8723404255319148, Beta = 0.5
Threshold: 0.0, F-score: 0.23017902813299232, Beta = 0.5
Area under PR (precision-recall curve) = 0.8797032493151403
Area under ROC (Receiver Operating Characteristic) = 0.942266519
0107051
```

Supplementary method that does the same computations as those provided by Spark creators:

```
  def tfidf(data: RDD[(String, Array[String])])(implicit sc: Spa
rkContext) = {
    val docs = data.count.toDouble
    //TF - terms frequencies
    val tfs = data.map {
      t => (t._1, t._2.foldLeft(Map.empty[String, Int])((m, s) =
> m + (s -> (1 + m.getOrElse(s, 0)))))
    }

    //TF-IDF
    val idfs = data.flatMap(_._2).map((_, 1)).reduceByKey(_ + _)
.map {
      case (term, count) => (term, math.log(docs / (1 + count)))
    }.collectAsMap

    //idfs.lookup("").lift(0).getOrElse(0d)
    tfs.map {
      case (m, tf) =>
        (m, tf.map {
          case (term, freq) => (term, freq * idfs.getOrElse(term
, 0d))
        })
    }
  }
```

which can be used e.g. to display terms with highest frequencies that are used in spam messages:

```
    val termsInSpamMsgs = tfidf(raw.filter(_._1 == "spam").map(t
 => (t._1, t._2))).sortBy(_._2.values, ascending = false)
    termsInSpamMsgs.take(10).foreach(println)
```

Output (limited to just 3 records) can look as follows:

```
(spam,Map(poly -> 11.749883315444682, sleepingwith, -> 5.7884299
48716486, 4 -> 2.1508437889901, pobox365o4w45wq -> 5.78842994871
6486, tones -> 3.3035232989284853, all -> 3.223480591254949, gr8
 -> 4.402135587596595, to -> 0.09301572373080139, direct -> 4.08
368185647806, eg -> 3.9966704794884307, finest, -> 5.78842994871
6486, 8007 -> 3.48584485572244, crazyin, -> 5.788429948716486, 2
u -> 5.788429948716486, rply -> 4.689817660048376, with -> 1.938
2823470064272, titles: -> 5.09528276815654, title -> 5.095282768
15654, ymca -> 5.788429948716486, :getzed.co.uk -> 5.78842994871
6486, 300p -> 5.788429948716486, mobs -> 5.788429948716486, brea
the1 -> 5.788429948716486))
(spam,Map(it's -> 10.19056553631308, your -> 0.9926394031197446,
 learn -> 5.788429948716486, txts! -> 5.788429948716486, but ->
4.535666980221118, incredible -> 5.788429948716486, mind. -> 5.7
88429948716486, blow -> 5.788429948716486, it -> 3.3460829133472
814, 18p/txt -> 5.788429948716486, reply -> 1.9382823470064272,
that -> 3.3035232989284853, to -> 0.09301572373080139, now -> 2.
0995504946025494, you -> 1.0566271117950283, believe -> 5.788429
948716486, g -> 5.788429948716486, truly -> 5.382964840608321, t
hings -> 5.788429948716486, will -> 2.768005062572123, from -> 1
.7024536361649016, true. -> 5.788429948716486, o2fwd -> 5.788429
948716486, won't -> 5.382964840608321, amazing -> 5.382964840608
321, only -> 2.456225438541282))
(spam,Map(am -> 7.833255543629789, borin -> 5.788429948716486, x
x -> 4.8721392168423305, u -> 3.8137323015460964, & -> 1.9817674
58946166, luv -> 4.689817660048376, 09099725823 -> 5.78842994871
6486, calls£1/minmoremobsemspobox45po139wa -> 5.382964840608321,
 now -> 4.199100989205099, chat -> 2.985069567809951, here -> 4.
8721392168423305, cum -> 5.09528276815654, over -> 4.40213558759
6595, hope -> 4.8721392168423305, claire -> 10.765929681216642,
2nite? -> 5.382964840608321, alone -> 5.09528276815654, 2 -> 1.4
576966084301548, havin -> 5.788429948716486, c -> 3.708988407036
65, time -> 3.70898840703665, wanna -> 4.08368185647806))
...
```

Java 8 version can look like that:

```
        try (JavaSparkContext sc = new JavaSparkContext(configLo
calMode("NaiveBayes exploiting TFIDF for spam classification in
Java 8"))) {
            HashingTF hash = new HashingTF(100000);
            JavaRDD<String> file = localFile("sms-labeled.txt",
sc);
            JavaRDD<Tuple3<String, List<String>, Vector>> raw =
file.distinct().map(
                s -> s.split("\\t+")
            ).map(
                a -> new Tuple2<>(a[0], Arrays.stream(a[1].split(
"\\s+")).map(w -> w.toLowerCase()).collect(Collectors.toList()))
            ).map(
                t -> new Tuple3<>(t._1, t._2, hash.transform(t._
2))
            ).cache();

            IDFModel idf = new IDF().fit(raw.map(t -> t._3()).rd
d());
            JavaRDD<LabeledPoint> data = raw.map(t -> {
                int label = 0;
                if(t._1().equals("spam")){
                    label = 1;
                }
                return new LabeledPoint(label, idf.transform(t._
3()));
            });

            JavaRDD<LabeledPoint>[] split = data.randomSplit(new
double[] { .8, .2 });
            JavaRDD<LabeledPoint> train = split[0].cache();
            JavaRDD<LabeledPoint> test = split[1].cache();

            NaiveBayesModel model = NaiveBayes.train(train.rdd()
);
            evaluateModel(model, test);
        }
```

And method for model evaluation:

```
        test.take(5).forEach(x -> {
            System.out.println(String.format("Predicted: %.1f, L
abel: %.1f", model.predict(x.features()), x.label()));
        });

        JavaPairRDD<Object, Object> predictionsAndLabels = test.
mapToPair(
            p -> new Tuple2<Object, Object>(model.predict(p.feat
ures()), p.label())
        );

        Stats stats = Stats.apply(confusionMatrix(predictionsAnd
Labels.rdd()));
        System.out.println(stats.toString());

        BinaryClassificationMetrics metrics = new BinaryClassifi
cationMetrics(predictionsAndLabels.rdd());
        printMetrics(metrics);
```

Using regression for predicting House prices

In this example different regression algorithms are used to predict house prices. Data set can be found in data folder on github in file house-data.csv.

Evaluation regression algorithm, at least few things ought to be taken into account:

- Regression model
- Model performance

Application name is set to:

```
val regressionApp = "Decision Tree Algorithm as classifier of Bi
ke Buyers"
```

Spark's linear regression like classification algorithms work with LabeledPoint data structure. Using case class that reflects raw data can make conversion into LabeledPoints a bit easier:

```
case class HouseModel(id: Long,
                      date: java.sql.Date,
                      price: Double,
                      bedrooms: Int,
                      bathrooms: Double,
                      sqft_living: Int,
                      sqft_lot: Int,
                      floors: Double,
                      waterfront: Int,
                      view: Int,
                      condition: Int,
                      grade: Int,
                      sqft_above: Int,
                      sqft_basement: Int,
                      yr_built: Int,
                      yr_renovated: Int,
                      zipcode: String,
                      lat: Double,
                      long: Double,
                      sqft_living15: Int,
                      sqft_lot15: Int)
    extends LabeledPointConverter {

  def label() = price.toDouble
  def features() = HouseModel.convert(this)
}
```

HouseModel companion object is overridden and together with apply method it provides method for conversion to Vector and marking categorical features

```
object HouseModel {

  def df = new java.text.SimpleDateFormat("yyyyMMdd'T'hhmmss")

  def apply(row: Array[String]) = new HouseModel(
    row(0).toLong, new java.sql.Date(df.parse(row(1)).getTime),
    row(2).toInt, row(3).toInt,
    row(4).toDouble, row(5).toInt, row(6).toInt,
    row(7).toDouble, row(8).toInt, row(9).toInt,
    row(10).toInt, row(11).toInt, row(12).toInt,
    row(13).toInt, row(14).toInt, row(15).toInt, row(16),
    row(17).toDouble, row(18).toDouble, row(19).toInt, row(20).t
oInt)

  def convert(model: HouseModel) = Vectors.dense(
    model.id.toDouble,
    model.bedrooms.toDouble,
    model.bathrooms,
    model.sqft_living.toDouble,
    model.sqft_lot.toDouble,
    model.floors,
    model.waterfront.toDouble,
    model.view.toDouble,
    model.condition.toDouble,
    model.grade.toDouble,
    model.sqft_above.toDouble,
    model.sqft_basement.toDouble,
    model.yr_built.toDouble,
    model.yr_renovated.toDouble,
    model.lat,
    model.long,
    model.sqft_living15.toDouble,
    model.sqft_lot15.toDouble)
}
```

Data is loaded following way:

```
val houses = hdFile.map(_.split(",")).
  filter { t => catching(classOf[NumberFormatException]).opt(t(0
).toLong).isDefined }.
  map { HouseModel(_).toLabeledPoint() }
```

And splitted into training and test set:

```
val Array(train, test) = houses.randomSplit(Array(.9, .1), 10204
L)
```

In regression it is recommended that the input variables have a mean of 0. It's easy to achieve by using the StandardScaler from Spark MLLib.

```
val scaler = new StandardScaler(withMean = true, withStd = true)
.fit(train.map(dp => dp.features))
val scaledTrain = train.map(dp => new LabeledPoint(dp.label, sca
ler.transform(dp.features))).cache()
val scaledTest = test.map(dp => new LabeledPoint(dp.label, scale
r.transform(dp.features))).cache()
```

Summary statistics can be calculated:

```
val stats2 = Statistics.colStats(scaledTrain.map { x => x.featur
es })
println(s"Max : ${stats2.max}, Min : ${stats2.min}, and Mean : $
{stats2.mean} and Variance : ${stats2.variance}")
```

Linear regression in Scala

Function which creates linear regression model can look like that:

```
  def createLinearRegressionModel(rdd: RDD[LabeledPoint], numIte
rations: Int = 100, stepSize: Double = 0.01) = {
    LinearRegressionWithSGD.train(rdd, numIterations, stepSize)
  }
```

Decision tree in Scala

Function which creates decision tree model for regression can look like that:

```
  def createDecisionTreeRegressionModel(rdd: RDD[LabeledPoint],
maxDepth:Int = 10, maxBins:Int = 20) = {
    val impurity = "variance"
    DecisionTree.trainRegressor(rdd, Map[Int, Int](), impurity,
maxDepth, maxBins)
  }
```

Model evaluation

Regardless of method of creating regression model, linear:

```
val model = createLinearRegressionModel(scaledTrain)
```

or based on decision tree:

```
val model = createDecisionTreeRegressionModel(scaledTrain)
```

evaluation requires of using test data to calculate predictions and store them with actual prices:

```
val predictionsAndValues = scaledTest.map {
  point => (model.predict(point.features), point.label)
}
```

Mean house price:

```
scaledTest.map { x => x.label }.mean()
```

Max prediction error:

```
predictionsAndValues.map { case (p, v) => math.abs(v - p) }.max
```

Root mean squared error:

```
math.sqrt(predictionsAndValues.map { case (p, v) => math.pow((v
- p), 2) }.mean())
```

Sample output can look like that:

```
Mean house price: 534198.5644402637
Max prediction error: 1710000.0
Root Mean Squared Error: 149589.65702292052
```

Java 8 example of regression algorithms used for House prices predictions

Instead of using Scala case class regular Java bean can be used:

```
public class HouseModelJava implements LabeledPointConverter {

    private final Long id;
    private final Date date;
    private final Double price;
    private final Integer bedrooms;
    private final Double bathrooms;
    private final Integer sqft_living;
    private final Integer sqft_lot;
    private final Double floors;
    private final Integer waterfront;
    private final Integer view;
    private final Integer condition;
    private final Integer grade;
    private final Integer sqft_above;
    private final Integer sqft_basement;
    private final Integer yr_built;
    private final Integer yr_renovated;
    private final String zipcode;
    private final Double latitude;
    private final Double longitude;
    private final Integer sqft_living15;
    private final Integer sqft_lot15;

    public HouseModelJava(Long id, Date date, Double price, Inte
ger bedrooms, Double bathrooms, Integer sqft_living,
            Integer sqft_lot, Double floors, Integer waterfront,
 Integer view, Integer condition, Integer grade,
            Integer sqft_above, Integer sqft_basement, Integer y
r_built, Integer yr_renovated, String zipcode,
            Double latitude, Double longitude, Integer sqft_livi
ng15, Integer sqft_lot15) {
```

```
        super();
        this.id = id;
        this.date = date;
        this.price = price;
        this.bedrooms = bedrooms;
        this.bathrooms = bathrooms;
        this.sqft_living = sqft_living;
        this.sqft_lot = sqft_lot;
        this.floors = floors;
        this.waterfront = waterfront;
        this.view = view;
        this.condition = condition;
        this.grade = grade;
        this.sqft_above = sqft_above;
        this.sqft_basement = sqft_basement;
        this.yr_built = yr_built;
        this.yr_renovated = yr_renovated;
        this.zipcode = zipcode;
        this.latitude = latitude;
        this.longitude = longitude;
        this.sqft_living15 = sqft_living15;
        this.sqft_lot15 = sqft_lot15;
    }

    public HouseModelJava(String... row) {
        this(Long.parseLong(row[0]), new Date(parseDate(row[1]))
, Double.parseDouble(row[2]), Integer.parseInt(row[3]),
                Double.parseDouble(row[4]), Integer.parseInt(row[
5]), Integer.parseInt(row[6]), Double
                        .parseDouble(row[7]), Integer.parseInt(r
ow[8]), Integer.parseInt(row[9]), Integer
                        .parseInt(row[10]), Integer.parseInt(row[
11]), Integer.parseInt(row[12]), Integer
                        .parseInt(row[13]), Integer.parseInt(row[
14]), Integer.parseInt(row[15]), row[16], Double
                        .parseDouble(row[17]), Double.parseDoubl
e(row[18]), Integer.parseInt(row[19]), Integer
                        .parseInt(row[20]));
    }
```

```
    @Override
    public LabeledPoint toLabeledPoint() {
        return new LabeledPoint(label(), features());
    }

    @Override
    public double label() {
        return price;
    }

    @Override
    public Vector features() {
        double[] features = { id, bedrooms, bathrooms, sqft_livi
ng, sqft_lot, floors, waterfront, view, condition,
                grade, sqft_above, sqft_basement, yr_built, yr_r
enovated, latitude, longitude, sqft_living15,
                sqft_lot15 };
        return Vectors.dense(features);
    }

    static Long parseDate(String value) {
        try {
            return new java.text.SimpleDateFormat("yyyyMMdd'T'hh
mmss").parse(value).getTime();
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
    /* getters */
}
```

Function which creates linear regression model in Java 8 can look like that:

```
    static LinearRegressionModel createLinearRegressionModel(Jav
aRDD<LabeledPoint> rdd, Integer numIterations,
            Double stepSize) {
        return LinearRegressionWithSGD.train(rdd.rdd(),
                numIterations == null ? 100 : numIterations,
                stepSize == null ? 0.01 : stepSize);
    }
```

Function which creates decision tree model in Java 8 for regression can look like that:

```
    static DecisionTreeModel createDecisionTreeRegressionModel(J
avaRDD<LabeledPoint> rdd, Integer maxDepth,
            Integer maxBins) {
        String impurity = "variance";
        return DecisionTree.trainRegressor(rdd,
                Collections.emptyMap(),
                impurity, maxDepth == null ? 10 : maxDepth,
                maxBins == null ? 20 : maxBins);
    }
```

And the rest of the code:

```
        try (JavaSparkContext sc = new JavaSparkContext(configLo
calMode(regressionApp()))) {
            JavaRDD<String> hdFile = localFile("house-data.csv",
 sc);

            JavaRDD<LabeledPoint> houses = hdFile.map(s -> s.spl
it(",")).
                    filter(t -> !"date".equals(t[1]) ).
                    map(a -> new HouseModelJava(a).toLabeledPoin
t());

            StandardScalerModel scaler = new StandardScaler(true
, true).fit(houses.map(dp -> dp.features()).rdd());

            JavaRDD<LabeledPoint>[] split = houses.map(
```

```
                    dp -> new LabeledPoint(dp.label(), scaler.tr
ansform(dp.features())))
                    .randomSplit(new double[] { .9, .1 }, 10204L
);
            JavaRDD<LabeledPoint> train = split[0].cache();
            JavaRDD<LabeledPoint> test = split[1].cache();

            DecisionTreeModel model = createDecisionTreeRegressi
onModel(train, null, null);

            test.take(5)
                    .stream()
                    .forEach(
                            x -> System.out.println(String.forma
t("Predicted: %.1f, Label: %.1f",
                                    model.predict(x.features()),
 x.label()))
                    );

            JavaPairRDD<Object, Object> predictionsAndValues = t
est.mapToPair(
                    p -> new Tuple2<Object, Object>(model.predic
t(p.features()), p.label())
                    );

            System.out.println("Mean house price: " + test.mapTo
Double(x -> x.label()).mean());
            System.out.println("Max prediction error: "
                    + predictionsAndValues.mapToDouble(
                            t2 -> Math.abs(Double.class.cast(t2.
_2) - Double.class.cast(t2._1))).max());

            RegressionMetrics metrics = new RegressionMetrics(pr
edictionsAndValues.rdd());

            System.out.println(String.format("Mean Squared Error
: %.2f", metrics.meanSquaredError()));
            System.out.println(String.format("Root Mean Squared
Error: %.2f", metrics.rootMeanSquaredError()));
            System.out.println(String.format("Coefficient of Det
```

```
ermination R-squared: %.2f", metrics.r2()));
            System.out.println(String.format("Mean Absoloute Err
or: %.2f", metrics.meanAbsoluteError()));
            System.out.println(String.format("Explained variance
: %.2f", metrics.explainedVariance()));
        }
    }
```

Sample output can look like that:

Predicted: 481028,4, Label: 510000,0

Predicted: 732639,1, Label: 937000,0

Predicted: 437099,1, Label: 438000,0

Predicted: 632961,1, Label: 580500,0

Predicted: 328065,5, Label: 322500,0

Mean house price: 534198.5644402637

Max prediction error: 1710000.0

Mean Squared Error: 22243597500,87

Root Mean Squared Error: 149142,88

Coefficient of Determination R-squared: 0,81

Mean Absoloute Error: 87380,94

Explained variance: 110086143895,00

Images classification with Convolutional Neural Networks, Deeplearning4j and Spark

Deeplearning4j is a library which provides implementation for some deep learning algorithms. Among the others it supports Convolutional Neural Network, which is kind of artificial neural network inspired by nature (how animal visual cortex works). The idea behind CNN is quite similar to "regular" neural network. There are layers of neurons which have inputs, weights and biases that are used to compute dot-product, and its linear or non-linear transformation. The most significant difference is that images are assumed as network inputs, neurons between layer are not fully connected to each other and some of them share their weights. More detailed information about such neural networks can be found on this web site created by Andrej Karpathy from Stanford University. Here, all code is written in Java.

Dataset and its augmentation

As primary dataset for all experiments in this example, small subset of well-known CIFAR-10 dataset is used. It contains about 6k RGB images in size 32x32 pixels, divided into 4 distinct categories: bird, car, cat, dog, and can be found on github. Images are stored in text file with category name in first column, and image data in second column. Each channel (r,g,b) is separated, so first 32x32 bytes is for Red, next for Green, and last for Blue. Data set can be loaded and transformed into dl4j internal data formats with following code:

```
JavaRDD<String> raw = sc.textFile("data/images-data-rgb.csv");
String first = raw.first();

JavaPairRDD<String, String> labelData = raw
    .filter(f -> f.equals(first) == false)
    .mapToPair(r -> {
        String[] tab = r.split(";");
        return new Tuple2<>(tab[0], tab[1]);
    });

JavaRDD<Tuple2<INDArray, double[]>> labelsWithData = labelData
    .map(t -> {
        INDArray label = FeatureUtil.toOutcomeVector(labels.get(
t._1).intValue(), labels.size());
        double[] arr = Arrays.stream(t._2.split(" "))
            .map(normalize1)
            .mapToDouble(Double::doubleValue).toArray();
        return new Tuple2<>(label, arr);
    });
```

One step farther has to be taken, as dl4j expects their own "DataSet" data type. Here with splitting into training and test data:

```
JavaRDD<Tuple2<INDArray, double[]>>[] splited = labelsWithData.r
andomSplit(new double[] { .8, .2 }, seed);

JavaRDD<DataSet> testDataset = splited[1]
    .map(t -> {
        INDArray features = Nd4j.create(t._2, new int[] { 1, t._
2.length });
        return new DataSet(features, t._1);
    }).cache();
log.info("Number of test images {}", testDataset.count());
```

Because used data set isn't huge, some augmentation can help achieving better accuracy. The simplest method is to add to it its images, but flipped horizontally. For trained network it should be irrelevant whether e.g. cat is looking to the left or right. Here, all training images are flipped, but of course some subset of initial training data set could be used. Input data values are between 0 and 255, and after simple normalization between -1.0 and 1.0. Procedure is as follows:

```
JavaRDD<DataSet> plain = splited[0]
    .map(t -> {
        INDArray features = Nd4j.create(t._2, new int[] { 1, t._
2.length });
        return new DataSet(features, t._1);
    });

JavaRDD<DataSet> flipped = splited[0]
    .map(t -> {
        double[] arr = t._2;
        int idx = 0;
        double[] farr = new double[arr.length];
        for (int i = 0; i < arr.length; i += trainer.width) {
            double[] temp = Arrays.copyOfRange(arr, i, i + train
er.width);
            ArrayUtils.reverse(temp);
            for (int j = 0; j < trainer.height; ++j) {
                farr[idx++] = temp[j];
            }
        }
        INDArray features = Nd4j.create(farr, new int[] { 1, far
r.length });
        return new DataSet(features, t._1);
    });

JavaRDD<DataSet> trainDataset = plain.union(flipped).cache();
log.info("Number of train images {}", trainDataset.count());
```

For curiosity or for test purposes example image from data set can be turned into BufferedImage by using this code:

```
    static int IMAGE_DEPTH = 3;
    static int IMAGE_WIDTH = 32;
    static int IMAGE_HIGHT = 32;
    static int IMAGE_SIZE = 32 * 32;

    public static BufferedImage getImageFromArray(int[] pixels)
{
        BufferedImage image = new BufferedImage(IMAGE_WIDTH, IMA
GE_HIGHT, BufferedImage.TYPE_INT_RGB);
        for (int i = 0; i < pixels.length / IMAGE_DEPTH; ++i) {
            int rgb = new Color(pixels[i], pixels[i + IMAGE_SIZE
], pixels[i + IMAGE_SIZE + IMAGE_SIZE]).getRGB();
            image.setRGB(i % IMAGE_WIDTH, i / IMAGE_HIGHT, rgb);
        }
        return image;
    }
```

and after that easily saved e.g. in PNG format:

```
  int[] pixels = sampleImage(new File("data/images-data-rgb.csv"
));
  BufferedImage bi = getImageFromArray(pixels);
  ImageIO.write(bi, "PNG", new File("test.png"));
```

with function for selecting sample image from provided dataset:

```
    public static int[] sampleImage(File file) {
        try (BufferedReader br = new BufferedReader(new FileRead
er(file));) {
            return br.lines().findAny().map(l -> Arrays.stream(l
.split(";")[1].split(" ")).mapToInt(Integer::parseInt).toArray()
).get();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
```

Network architecture selection

For Convolutional Neural Networks architectures following parameters are crucial:

- number of layers, which defines network depth
- kind of layer, as one of the following:
  - convolutional, for which number of filters that will be learned by network, filter size, padding and stride must be specified,
  - subsampling, with pooling operation (MAX, AVG) and filter size,
  - local response normalization,
  - dense, which is classic fully-connected layer, usually with some dropout regularization,
  - output, with softmax function, which produces a distribution over class labels
- network input parameters, image width, height and number of channels (3 for rgb images)

Here, several different architectures are tested:

- [CONV(3x3, 64) -> NORM -> SUB(2x2, MAX)] *2 -> [CONV(3x3, 64)]* 2 -> SUB(2x2, MAX) -> FC(384)
- [CONV(3x3, 64) -> SUB(2x2, MAX)] * 3 -> NORM -> FC(384)
- [CONV(3x3, 32) -> SUB(2x2, MAX)] * 2 -> CONV(3x3, 64) -> SUB(2x2, MAX) -> NORM -> FC(384)
- CONV(5x5, 25) -> NORM -> SUB(2x2, MAX) -> CONV(3x3, 50) -> SUB(2x2, MAX) -> NORM -> FC(400)

where CONV stands for convolutional layer, SUB subsampling layer, NORM - local response normalization layer and FC fully connected dense layer. Numbers in brackets refer to the most important parameters on each layer.

Definitions are stored in helper class, and look as follow:

```
    public static NetworkModel net1 = (learningRate, width, heig
ht, channels, numLabels) -> {
```

```
        int iterations = 1;

        int layer = 0;
        MultiLayerConfiguration.Builder builder = new NeuralNetC
onfiguration.Builder()
                .seed(seed)
                .iterations(iterations)
                .regularization(true).l1(0.0001).l2(0.0001)//ela
stic net regularization
                .learningRate(learningRate)
                .optimizationAlgo(OptimizationAlgorithm.STOCHAST
IC_GRADIENT_DESCENT)
                .updater(Updater.NESTEROVS).momentum(0.9)
                .gradientNormalization(GradientNormalization.Ren
ormalizeL2PerLayer)
                .useDropConnect(true)
                .leakyreluAlpha(0.02)
                .list()
                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .nIn(channels)
                        .padding(1, 1)
                        .nOut(64)
                        .weightInit(WeightInit.RELU)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new LocalResponseNormalization.B
uilder().build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())

                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(1, 1)
                        .nOut(64)
                        .weightInit(WeightInit.RELU)
                        .activation("leakyrelu")
```

```
                        .build())
                .layer(layer++, new LocalResponseNormalization.B
uilder().build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())

                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(0, 0)
                        .nOut(64)
                        .weightInit(WeightInit.RELU)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(0, 0)
                        .nOut(64)
                        .weightInit(WeightInit.RELU)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())
                .layer(layer++, new DenseLayer.Builder().activat
ion("relu")
                        .name("dense")
                        .weightInit(WeightInit.NORMALIZED)
                        .nOut(384)
                        .dropOut(0.5)
                        .build())
                .layer(layer++, new OutputLayer.Builder(LossFunc
tions.LossFunction.NEGATIVELOGLIKELIHOOD)
                        .nOut(numLabels)
                        .weightInit(WeightInit.XAVIER)
                        .activation("softmax")
                        .build())
                .backprop(true)
```

```
                .pretrain(false)
                .cnnInputSize(width, height, channels);
        return builder.build();
    };

    public static NetworkModel net2 = (learningRate, width, heig
ht, channels, numLabels) -> {

        int iterations = 1;

        int layer = 0;

        MultiLayerConfiguration.Builder builder = new NeuralNetC
onfiguration.Builder()
                .seed(seed)
                .iterations(iterations)
                .regularization(true).l1(0.0001).l2(0.0001)
                .learningRate(learningRate)
                .optimizationAlgo(OptimizationAlgorithm.STOCHAST
IC_GRADIENT_DESCENT)
                .updater(Updater.NESTEROVS).momentum(.9)
                .gradientNormalization(GradientNormalization.Ren
ormalizeL2PerLayer)
                .useDropConnect(true)
                .leakyreluAlpha(0.02)
                .list()
                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(1, 1)
                        .nOut(64)
                        .weightInit(WeightInit.VI)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())

                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
```

```
                        .padding(1, 1)
                        .nOut(64)
                        .weightInit(WeightInit.VI)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())

                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(0, 0)
                        .nOut(64)
                        .weightInit(WeightInit.VI)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())

                .layer(layer++, new LocalResponseNormalization.B
uilder().build())

                .layer(layer++, new DenseLayer.Builder().activat
ion("relu")
                        .name("dense")
                        .weightInit(WeightInit.VI)
                        .nOut(384)
                        .dropOut(0.5)
                        .build())
                .layer(layer++, new OutputLayer.Builder(LossFunc
tions.LossFunction.NEGATIVELOGLIKELIHOOD)
                        .nOut(numLabels)
                        .weightInit(WeightInit.VI)
                        .activation("softmax")
                        .build())
                .backprop(true)
                .pretrain(false)
```

```
                .cnnInputSize(width, height, channels);
        return builder.build();
    };

    public static NetworkModel net3 = (learningRate, width, heig
ht, channels, numLabels) -> {

        int iterations = 1;

        int layer = 0;

        MultiLayerConfiguration.Builder builder = new NeuralNetC
onfiguration.Builder()
                .seed(seed)
                .iterations(iterations)
                .regularization(true).l1(0.0001).l2(0.0001)
                .learningRate(learningRate)
                .optimizationAlgo(OptimizationAlgorithm.STOCHAST
IC_GRADIENT_DESCENT)
                .updater(Updater.NESTEROVS).momentum(0.9)
                .gradientNormalization(GradientNormalization.Ren
ormalizeL2PerLayer)
                .useDropConnect(true)
                .leakyreluAlpha(0.02)
                .list()
                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(1, 1)
                        .nOut(32)
                        .weightInit(WeightInit.VI)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())

                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(1, 1)
```

```
                        .nOut(32)
                        .weightInit(WeightInit.VI)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())

                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(0, 0)
                        .nOut(64)
                        .weightInit(WeightInit.VI)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())

                .layer(layer++, new LocalResponseNormalization.B
uilder().build())

                .layer(layer++, new DenseLayer.Builder().activat
ion("relu")
                        .name("dense")
                        .weightInit(WeightInit.VI)
                        .nOut(384)
                        .dropOut(.5)
                        .build())
                .layer(layer++, new OutputLayer.Builder(LossFunc
tions.LossFunction.NEGATIVELOGLIKELIHOOD)
                        .nOut(numLabels)
                        .weightInit(WeightInit.VI)
                        .activation("softmax")
                        .build())
                .backprop(true)
                .pretrain(false)
                .cnnInputSize(width, height, channels);
```

```
        return builder.build();
    };

    public static NetworkModel net4 = (learningRate, width, heig
ht, channels, numLabels) -> {

        int iterations = 1;

        int layer = 0;

        MultiLayerConfiguration.Builder builder = new NeuralNetC
onfiguration.Builder()
                .seed(seed)
                .iterations(iterations)
                .regularization(true).l2(0.0005)
                .learningRate(learningRate)
                .optimizationAlgo(OptimizationAlgorithm.STOCHAST
IC_GRADIENT_DESCENT)
                .updater(Updater.NESTEROVS).momentum(.9)
                .gradientNormalization(GradientNormalization.Ren
ormalizeL2PerLayer)
                .useDropConnect(true)
                .leakyreluAlpha(0.02)
                .minimize(false)
                .list()
                .layer(layer++, new ConvolutionLayer.Builder(5, 5
)
                        .nIn(channels)
                        .padding(2, 2)
                        .nOut(25)
                        .weightInit(WeightInit.RELU)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new LocalResponseNormalization.B
uilder().build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())
```

```
                .layer(layer++, new ConvolutionLayer.Builder(3, 3
)
                        .padding(1, 1)
                        .nOut(50)
                        .weightInit(WeightInit.RELU)
                        .activation("leakyrelu")
                        .build())
                .layer(layer++, new SubsamplingLayer.Builder(Sub
samplingLayer.PoolingType.MAX)
                        .kernelSize(2, 2)
                        .build())
                .layer(layer++, new LocalResponseNormalization.B
uilder().build())

                .layer(layer++, new DenseLayer.Builder().activat
ion("relu")
                        .name("dense")
                        .weightInit(WeightInit.NORMALIZED)
                        .nOut(400)
                        .dropOut(0.5)
                        .build())
                .layer(layer++, new OutputLayer.Builder(LossFunc
tions.LossFunction.NEGATIVELOGLIKELIHOOD)
                        .nOut(numLabels)
                        .weightInit(WeightInit.XAVIER)
                        .activation("softmax")
                        .build())
                .backprop(true)
                .pretrain(false)
                .cnnInputSize(width, height, channels);
        return builder.build();
    };
```

NetworkModel is Java 8 functional interface:

```
    @FunctionalInterface
    public interface NetworkModel extends Serializable {
        MultiLayerConfiguration apply (double learningRate, int
width, int height, int channels, int numLabels);
    }
```

Learning process

After selecting CNN architecture, the process of learning consists of proper initialization and tunning network hyperparameters. This task usually requires some experiments, and doing this on Spark is no exception. Following parameters ought to be set:

- number of epochs is the number of full training cycles with using the whole training data set,
- updater, for all examples set to NESTEROVS with momentum set to 0.9,
- l1 and/or l2 regularization,
- learning rate, after several tests adjusted to quite high value 0.75 for start,
- learning rate decay factors (number of steps with no increased accuracy after which learning rate will be decayed by a specified value)
- number of iterations, for all examples set to 1,
- batch size, twelve times number of cores available,
- alpha parameter for leaky RELU neurons activation function, set to 0.02 (default is 0.01),
- dropout value (only for fully connected layer) set to 0.5.

Some additional useful information about effective learning of neural networks and their evaluation can be found here and also here.

The whole learning process is coded in following method:

```
public void train(JavaRDD<DataSet> train, JavaRDD<DataSet> test)
{

    int batchSize = 12 * cores;
    int lrCount = 0;
    double bestAccuracy = Double.MIN_VALUE;

    double learningRate = initialLearningRate;

    int trainCount = Long.valueOf(train.count()).intValue();
    log.info("Number of training images {}", trainCount);
    log.info("Number of test images {}", test.count());
```

```
    MultiLayerNetwork net = new MultiLayerNetwork(model.apply(le
arningRate, width, height, channels, numLabels));
    net.init();

    Map<Integer, Double> acc = new HashMap<>();
    for (int i = 0; i < epochs; i++) {

        SparkDl4jMultiLayer sparkNetwork = networkToSparkNetwork
.apply(net);
        final MultiLayerNetwork nn = sparkNetwork.fitDataSet(tra
in, batchSize, trainCount, cores);
        log.info("Epoch {} completed", i);

        JavaPairRDD<Object, Object> predictionsAndLabels = test.
mapToPair(
                ds -> new Tuple2<>(label(nn.output(ds.getFeature
Matrix(), false)), label(ds.getLabels()))
                );
        MulticlassMetrics metrics = new MulticlassMetrics(predic
tionsAndLabels.rdd());
        double accuracy = 1.0 * predictionsAndLabels.filter(x ->
 x._1.equals(x._2)).count() / test.count();
        log.info("Epoch {} accuracy {} ", i, accuracy);
        acc.put(i, accuracy);
        predictionsAndLabels.take(10).forEach(t -> log.info("pre
dicted {}, label {}", t._1, t._2));
        log.info("confusionMatrix {}", metrics.confusionMatrix()
);

        INDArray params = nn.params();
        if (accuracy > bestAccuracy) {
            bestAccuracy = accuracy;
            try {
                ModelSerializer.writeModel(nn, new File(workingD
ir, Double.toString(accuracy)), false);
            } catch (IOException e) {
                log.error("Error writing trained model", e);
            }
            lrCount = 0;
```

```
        } else {

            if (++lrCount % stepDecayTreshold == 0) {
                learningRate *= learningRateDecayFactor;
            }
            if (lrCount >= resetLearningRateThreshold) {
                lrCount = 0;
                learningRate = initialLearningRate;
            }
            if (learningRate < minimumLearningRate) {
                lrCount = 0;
                learningRate = initialLearningRate;
            }
            if (bestAccuracy - accuracy > downgradeAccuracyThres
hold) {
                params = ModelLoader.load(workingDir, bestAccura
cy);
            }
        }
        net = new MultiLayerNetwork(model.apply(learningRate, wi
dth, height, channels, numLabels));
        net.init();
        net.setParameters(params);
        log.info("Learning rate {} for epoch {}", learningRate,
i + 1);
    }
    log.info("Training completed");

}
```

Decay of learning rate is in this example done manually, and every model with accuracy better then its predecessors is stored in working folder. Number of epochs as well as every other parameters can be provided for learning process:

```
NetworkTrainer trainer = new NetworkTrainer.Builder()
    .model(ModelLibrary.net1)
    .networkToSparkNetwork(net -> new SparkDl4jMultiLayer(sc, ne
t))
    .numLabels(labels.size())
    .cores(NUM_CORES).build();
...
trainer.train(trainDataset, testDataset);
```

Training networks with selected architectures leads to models with 75% - 77% accuracy after several epochs. Usually after epoch 50 there is no improvement, regardless of learning rate that is being used.

Ensambles of best models

To move accuracy few percents up, ensambles of models can be used. Two similar options are presented:

- ensamble of three models with averaged outputs,
- ensamble of nine models with voting strategy. For the latter, 80% accuracy is achieved.

After re-creating test dataset, models selected to cooperate can be loaded from disk:

```
            String dir = EnsambleAvg.class.getClassLoader().getR
esource("models").getFile();
            MultiLayerNetwork n1 = ModelSerializer.restoreMultiL
ayerNetwork(new File(dir, "0.7596314907872697"));
            MultiLayerNetwork n2 = ModelSerializer.restoreMultiL
ayerNetwork(new File(dir, "0.7763819095477387"));
            MultiLayerNetwork n3 = ModelSerializer.restoreMultiL
ayerNetwork(new File(dir, "0.7646566164154104"));
```

or loaded all at once:

```
List<MultiLayerNetwork> nets = Arrays
    .stream(new File(EnsambleVote.class.getClassLoader().getReso
urce("models").getFile()).listFiles())
    .map(f -> {
        MultiLayerNetwork n = null;
        try {
            n = ModelSerializer.restoreMultiLayerNetwork(f);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return n;
    }).collect(Collectors.toList());
```

By using function that averages their outputs, ensamble can be established:

```
static double[] predict(int numLabels, DataSet ds, MultiLayerNet
work... nets) {

    List<double[]> outputs = Arrays.stream(nets)
        .map(net -> net.output(ds.getFeatureMatrix(), false).dat
a().asDouble())
        .collect(Collectors.toList());

    double[] result = new double[numLabels];
    Arrays.fill(result, 0d);

    outputs.forEach(d -> {
        for (int i = 0; i < numLabels; ++i) {
            result[i] += d[i];
        }
    });

    for (int i = 0; i < numLabels; ++i) {
        result[i] /= nets.length;
    }

    return result;
}
```

Misclassified images can be shown together with their predicted labels:

```
test.filter(
        ds -> label(predict(numLabels, ds, nets.toArray(new Mult
iLayerNetwork[0]))) != label(ds
            .getLabels()))
    .foreach(
            ds -> log.info("predicted {}, label {}",
                    asString(predict(numLabels, ds, nets.toArray(
new MultiLayerNetwork[0]))),
                    label(ds.getLabels()))
    );
```

Evaluation looks almost the same as for single model:

```
JavaPairRDD<Object, Object> predictionsAndLabels = test
    .mapToPair(
    ds -> new Tuple2<>(label(predict(numLabels, ds, n1, n2, n3))
, label(ds.getLabels()))
    );

MulticlassMetrics metrics = new MulticlassMetrics(predictionsAnd
Labels.rdd());
double accuracy = 1.0 * predictionsAndLabels.filter(x -> x._1.eq
uals(x._2)).count() / test.count();
log.info("accuracy {} ", accuracy);
predictionsAndLabels.take(10).forEach(t -> log.info("predicted {
}, label {}", t._1, t._2));
log.info("confusionMatrix {}", metrics.confusionMatrix());
```

Function for voting:

```
static double[] predict(int numLabels, DataSet ds, MultiLayerNet
work... nets) {

    List<double[]> outputs = Arrays.stream(nets)
        .map(net -> net.output(ds.getFeatureMatrix(), false).dat
a().asDouble())
        .collect(Collectors.toList());

    double[] result = new double[numLabels];
    Arrays.fill(result, Double.MIN_VALUE);

    outputs.forEach(d -> {
        double max = Arrays.stream(d).max().getAsDouble();
        for (int i = 0; i < numLabels; ++i) {
            if (d[i] < max) {
                continue;// use only max value
            }
            if (result[i] == Double.MIN_VALUE) {
                result[i] = 1d;
            } else {
                result[i] += 1d;
            }
        }
    });
    return result;
}
```

CIFAR-10 Dataset

Presented techniques can be utilized to classify images from the full CIFAR 10 dataset. The most significant difference is how to load images (although images representation is similar). Helper method for downloading and unpacking cifar 10 dataset is provided:

```
public static void downloadAndExtract() {

    if (new File("data/cifar-10-batches-bin", TEST_DATA_FILE).ex
ists() == false) {
        try {
            if (new File("data", ARCHIVE_BINARY_FILE).exists() =
= false) {
                URL website = new URL("http://www.cs.toronto.edu
/~kriz/" + ARCHIVE_BINARY_FILE);
                FileOutputStream fos = new FileOutputStream("dat
a/" + ARCHIVE_BINARY_FILE);
                fos.getChannel().transferFrom(Channels.newChanne
l(website.openStream()), 0, Long.MAX_VALUE);
                fos.close();
            }
            TarArchiveInputStream tar =
                    new TarArchiveInputStream(
                            new GZIPInputStream(new FileInputStr
eam("data/" + ARCHIVE_BINARY_FILE)));
            TarArchiveEntry entry = null;
            while ((entry = tar.getNextTarEntry()) != null) {
                if (entry.isDirectory()) {
                    new File("data", entry.getName()).mkdirs();
                } else {
                    byte data[] = new byte[2048];
                    int count;
                    BufferedOutputStream bos = new BufferedOutpu
tStream(
                            new FileOutputStream(new File("data/"
, entry.getName())), 2048);
```

```
                    while ((count = tar.read(data, 0, 2048)) != -
1) {
                        bos.write(data, 0, count);
                    }
                    bos.close();
                }
            }
            tar.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

which after checking the existence of test data file in local data folder starts to download tar.gz archive or just unpack it if already downloaded.

Unpacked files are transformed into arrays of double values, with label representing one of 10 categories at index 0 and RGB channels following them. Data augmentation can be done exactly the same way, by horizontally flipping training images:

```
JavaPairRDD<String, PortableDataStream> files = sc.binaryFiles("
data/cifar-10-batches-bin");

JavaRDD<double[]> imagesTrain = files
    .filter(f -> ArrayUtils.contains(CifarReader.TRAIN_DATA_FILE
S, extractFileName.apply(f._1)))
    .flatMap(f -> CifarReader.rawDouble(f._2.open()));

JavaRDD<double[]> imagesTest = files
    .filter(f -> CifarReader.TEST_DATA_FILE.equals(extractFileNa
me.apply(f._1)))
    .flatMap(f -> CifarReader.rawDouble(f._2.open()));

JavaRDD<DataSet> testDataset = imagesTest
    .map(i -> {
        INDArray label = FeatureUtil.toOutcomeVector(Double.valu
```

```
eOf(i[0]).intValue(), numLabels);
        double[] arr = Arrays.stream(ArrayUtils.remove(i, 0)).bo
xed().map(normalize2)
            .mapToDouble(Double::doubleValue).toArray();
        INDArray features = Nd4j.create(arr, new int[] { 1, arr.
length });
        return new DataSet(features, label);
    }).cache();
log.info("Number of test images {}", testDataset.count());


JavaPairRDD<INDArray, double[]> labelsWithDataTrain = imagesTrai
n.mapToPair(
    i -> {
        INDArray label = FeatureUtil.toOutcomeVector(Double.valu
eOf(i[0]).intValue(), numLabels);
        double[] arr = Arrays.stream(ArrayUtils.remove(i, 0)).bo
xed().map(normalize2).mapToDouble(Double::doubleValue).toArray()
;
        return new Tuple2<>(label, arr);
    });

JavaRDD<DataSet> flipped = labelsWithDataTrain
    .map(t -> {
        double[] arr = t._2;
        int idx = 0;
        double[] farr = new double[arr.length];
        for (int i = 0; i < arr.length; i += trainer.getWidth())
 {
            double[] temp = Arrays.copyOfRange(arr, i, i + train
er.getWidth());
            ArrayUtils.reverse(temp);
            for (int j = 0; j < trainer.getHeight(); ++j) {
                farr[idx++] = temp[j];
            }
        }
        INDArray features = Nd4j.create(farr, new int[] { 1, far
r.length });
        return new DataSet(features, t._1);
    });
```

```
JavaRDD<DataSet> trainDataset = labelsWithDataTrain
    .map(t -> {
        INDArray features = Nd4j.create(t._2, new int[] { 1, t._
2.length });
        return new DataSet(features, t._1);
    }).union(flipped).cache();
log.info("Number of train images {}", trainDataset.count());
```

Training procedure can be also the same (as for primary dataset), but in this case every single epoch is going to take much more time.

Using Apache Cassandra with Apache Spark

Cassandra is available by default on port 9042. For example on host:

```
  val cassandraHost = "192.168.1.34"
```

To use it from Spark, location of Cassandra must be added to spark configuration as shown in introduction.

Function to load data from Cassandra go as follow:

```
def cassandraFile: (SparkContext => RDD[String]) = sc => {
    import com.datastax.spark.connector._
    sc.cassandraTable("spark", "bike_buyers").map { row => row.c
olumnValues.mkString("\t") }
}
```

CassandraRows are mapped into Strings, only to keep the same form, as after reading from text file. More reasonably solution could transform rows directly into something more useful. To load data into Cassandra simple ETL program written in Scala can look like this:

```
object LoadBikeBuyers {

  def main(args: Array[String]): Unit = {

    org.apache.log4j.BasicConfigurator.configure()
    val host = args(0)
    val cc = com.datastax.spark.connector.cql.CassandraConnector(
Set(InetAddress.getByName(host)))

    val keyspaceCql = Source.fromInputStream(getClass.getResourc
eAsStream("/create_spark_keyspace.cql")).mkString
    val tableCql = Source.fromInputStream(getClass.getResourceAs
```

```
Stream("/create_bike_buyers_table.cql")).mkString

    val bbFile = Source.fromFile("data/bike-buyers.txt", "utf8")
.getLines()

    cc.withSessionDo(s => {
      s.execute(keyspaceCql)
      s.execute(tableCql)

      val columns = s.getCluster.getMetadata.getKeyspace("spark"
).getTable("bike_buyers").getColumns

      bbFile.map { x => x.split("\\t") }.foreach(row => {

        val insert = new QueryBuilder(s.getCluster).insertInto("
spark", "bike_buyers")
        row.zipWithIndex.foreach(vi => {
          val column = columns(vi._2)
          insert.value(column.getName,
            if (column.getType == DataType.text()) vi._1
            else if (column.getType == DataType.cfloat()) vi._1.
replaceFirst(",", ".").toFloat
            else vi._1.toInt)
        })
        s.execute(insert)
      })
    })
  }
}
```

Both scripts (create_spark_keyspace.cql, create_bike_buyers_table.cql) are on github.

After executing of LoadBikeBuyers.scala, keyspace “spark” and table “bike_buyers” are created, and content of bike-buyers file is loaded into it.

Running Apache Spark standalone cluster on Docker

For those who are familiar with Docker technology, it can be one of the simplest way of running Spark standalone cluster.

Here is the Dockerfile which can be used to build image (docker build .) with Spark 2.1.0 and Oracle's server JDK 1.8.121 on Ubuntu 14.04 LTS operating system:

```
FROM ubuntu:14.04

RUN apt-get update && apt-get -y install curl

# JAVA
ARG JAVA_ARCHIVE=http://download.oracle.com/otn-pub/java/jdk/8u1
21-b13/e9e7ea248e2c4826b92b3f075a80e441/server-jre-8u121-linux-x
64.tar.gz
ENV JAVA_HOME /usr/local/jdk1.8.0_121

ENV PATH $PATH:$JAVA_HOME/bin
RUN curl -sL --retry 3 --insecure \
  --header "Cookie: oraclelicense=accept-securebackup-cookie;" $
JAVA_ARCHIVE \
  | tar -xz -C /usr/local/ && ln -s $JAVA_HOME /usr/local/java

# SPARK
ARG SPARK_ARCHIVE=http://d3kbcqa49mib13.cloudfront.net/spark-2.1
.0-bin-hadoop2.7.tgz
RUN curl -s $SPARK_ARCHIVE | tar -xz -C /usr/local/

ENV SPARK_HOME /usr/local/spark-2.1.0-bin-hadoop2.7
ENV PATH $PATH:$SPARK_HOME/bin

COPY ha.conf $SPARK_HOME/conf

EXPOSE 4040 6066 7077 8080

WORKDIR $SPARK_HOME
```

Having this, docker compose to run multiple containers at the same time can be used. One master and one worker makes cluster ready, but compose file can be extended, and other workers can be added.
You can find this file on gettyimages or after some modifications here:

```
spark-master:
    image: spark-2
    command: bin/spark-class org.apache.spark.deploy.master.Mast
```

```
er -h spark-master
    hostname: spark-master
    environment:
      MASTER: spark://spark-master:7077
      SPARK_CONF_DIR: /conf
      SPARK_PUBLIC_DNS: 127.0.0.1
    expose:
      - 7001
      - 7002
      - 7003
      - 7004
      - 7005
      - 7006
      - 7077
      - 6066
    ports:
      - 4040:4040
      - 6066:6066
      - 7077:7077
      - 8080:8080
    volumes:
      - ./conf/spark-master:/conf
      - ./data:/tmp/data

spark-worker-1:
    image: spark-2
    command: bin/spark-class org.apache.spark.deploy.worker.Work
er spark://spark-master:7077
    hostname: spark-worker-1
    environment:
      SPARK_CONF_DIR: /conf
      SPARK_PUBLIC_DNS: 127.0.0.1
      SPARK_WORKER_CORES: 2
      SPARK_WORKER_MEMORY: 2g
      SPARK_WORKER_PORT: 8881
      SPARK_WORKER_WEBUI_PORT: 8081
    links:
      - spark-master
    expose:
      - 7012
```

```
      - 7013
      - 7014
      - 7015
      - 7016
      - 8881
    ports:
      - 8081:8081
    volumes:
      - ./conf/spark-worker-1:/conf
      - ./data:/tmp/data
```

SPARK_PUBLIC_DNS variable is set to localhost, but this is going to work only on Linux. Mac and Windows users should replace it e.g. with an IP address of virtual machine which is hosting docker.
Conf folder contains subfolders with spark-defaults.conf files, and its content is mounted to containers /conf directory.

To start Apache Spark Standalone cluster:

```
docker-compose up
```

command should be executed from folder in which docker-compose.yml file is located.