Tutorial:
Introduction to Spark MLlib

Faraz Faghri
Last update: September 19, 2015

1 Overview

Welcome

Welcome to the Spark MLlib tutorial. This tutorial covers the critical skills needed to develop a MLlib machine learning application. It starts with an already deployed MLlib environment where students will execute a series of hands-on labs.

Objectives

Upon completing this tutorial, students will be able to:

- Execute built-in MLlib machine learning algorithms
- Run one clustering and one classification algorithm (K-means and Naïve Bayesian)
- Run MLlib applications and examine the output

Structure

This guide is designed as a set of step-by-step instructions for hands-on exercises. It teaches you how to use the MLlib module in Spark in a series of examples. You should complete the exercises in the order described in the following steps:

1. Prepare an input dataset for clustering
2. Prepare HDFS and upload the dataset on it
3. Use a **K-means** application
4. Run the application and examine the output
5. Prepare an input dataset for classification
6. Prepare HDFS and upload the dataset on it
7. Use a **Naïve Bayesian** application
8. Run the application and examine the output

2 Requirements

This tutorial is designed to work on the **Hortonworks Sandbox 2.3** virtual machine. You need to have a working HortonWorks Sandbox machine either locally or on the Amazon Web Services.

All assignments are also designed based on **JDK 7** (included in the virtual machine).

Please refer to **Tutorial: Run HortonWorks Sandbox 2.3 Locally** or **Tutorial: Run HortonWorks Sandbox 2.3 on AWS** for more information.

3 Setup Virtual Machine

Step 1: Start the virtual machine; then connect to it through the SSH.

Step 2: After successfully logging in, you should see a prompt similar to the following:

```
[root@sandbox ~]#
```

Step 3: In order to build and package MLlib applications with JAVA, we need **maven**. In order to install maven, run the following commands:

```
# wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo
# yum install apache-maven
```

Step 4: Check the installation using the following command:

```
# mvn -version
```

Step 5: Create some working directories in HDFS where you will store the input data sets and results:

```
# hadoop fs -mkdir -p /mllib-tutorial/input
```

Step 6: Create another directory for results:

```
# hadoop fs -mkdir -p /mllib-tutorial/output
```

4 Setup working environment

MLlib is part of Spark, which is already installed and configured on your Hortonworks Sandbox Virtual Machine.

For this tutorial, clone the “cloudapp-mllib-tutorial” GitHub repository, which includes the code and data we are going to use:

Step 1: Clone the following GitHub repository and change your working directory:

```
# git clone https://github.com/ffaghri1/cloudapp-mllib-tutorial.git
# cd cloudapp-mllib-tutorial/
```

5 K-means clustering

In this section, we will use car data extracted from the 1974 *Motor Trend* US magazine. Dataset comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Step 1: Examine the input data set:

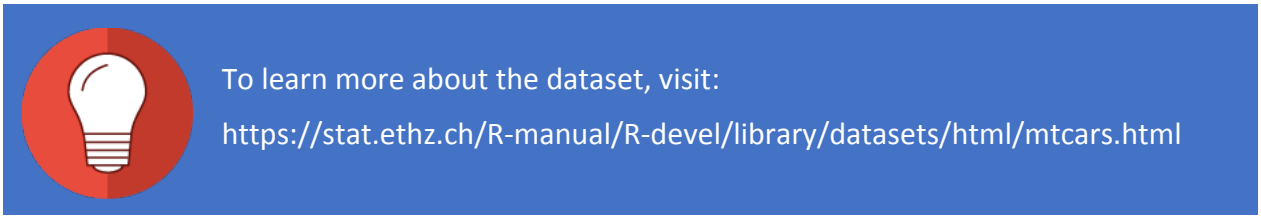
```
# nano data/mtcars.csv
```

As you can see, car names are in the first column. The rest of the columns correspond to the following information for each car:

```
cars,mpg,cyl,disp,hp,drat,wt,qsec,vs,am,gear,carb
```

Step 2: Copy the car dataset to HDFS:

```
# hadoop fs -put data/mtcars.csv /mllib-tutorial/input
```



Our goal is to use these properties to cluster cars into multiple categories using the K-means algorithm. K-means is one of the most used algorithms in Machine Learning. K-means belong to a class of machine learning algorithms called clustering algorithms. Clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters) [Wikipedia].

Now let's build a Kmeans implementation that uses Spark MLlib.

Step 1: Make sure you are in the "cloudapp-mllib-tutorial" directory. You can go to that directory by using the following command:

```
# cd ~/cloudapp-mllib-tutorial
```

Step 2: Let's find out how the Kmeans algorithm is developed using MLlib (also available in **Appendix A**):

```
# nano src/KMeansExample.java
```

Step 3: Build the code:

```
# mvn clean package
```

Step 4: Submit the compiled application to Spark to cluster the car dataset into three groups:

```
# spark-submit --class KMeansExample target/mllib.mp-1.0-SNAPSHOT.jar 3
/mllib-tutorial/input/mtcars.csv
```

Step 5 (optional): After submission, Spark will execute the application and print out the result among logging information. In case you want to see only the results, you can forward log the output to /dev/null:

```
# spark-submit --class KMeansExample target/mllib.mp-1.0-SNAPSHOT.jar 3
/mllib-tutorial/input/mtcars.csv 2> /dev/null
```

Note: You are printing the output on the screen. This is not the right practice with big data; we are only doing it for educational purposes in this tutorial. When you are dealing with large datasets, your clusters will be large, and the best practice is to right the output to HDFS and read it as needed.

6 Naive Bayes classifier

In machine learning, classification is the problem of identifying to which set of categories (sub-populations) a new observation belongs. You do this using as a basis a training set of data containing observations (or instances) whose category membership is known.

An example would be assigning a given email into "spam" or "non-spam" classes or assigning a diagnosis to a given patient as described by observed characteristics of the patient (gender, blood pressure, presence or absence of certain symptoms, etc.) [Wikipedia].

In this part of the tutorial, we will use a Naïve Bayesian classifier, which is a simple and effective method and one of the first methods to try for a classification problem. We are going to use the Naïve Bayesian classifier to predict the onset of diabetes.

To learn more about the dataset, visit UCI Machine Learning Repository:

https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes

First, prepare the data set, and then apply the classifier.

Step 1: Make sure you are in the “cloudapp-mllib-tutorial” directory. You can go to that directory by using the following command:

```
# cd ~/cloudapp-mllib-tutorial/
```

Step 2: Examine the input data set "Pima Indians Diabetes Data Set":

```
# nano data/pima-indians-diabetes.data
```

This dataset is comprised of 768 observations of medical details for Pima Indians patients. The records describe instantaneous measurements taken from the patient such as their age, the number of times pregnant, and their blood workup. All patients are women aged 21 or older. All attributes are numeric, and their units vary from attribute to attribute.

Each record has a class value that indicates whether the patient suffered an onset of diabetes within five years of when the measurements were taken (1) or not (0); you can see this indicator in the last column.

The goal of a classifier is to build a model that takes a patient's set of measurements and gives a result that could tell you whether that patient is diabetic.

This is a standard dataset that has been studied a lot in machine learning literature. A good prediction accuracy is 70%-76%.

Step 3: For supervised classifiers, we need a "training data" set to train our machine learning algorithm and a "test dataset" to evaluate the model. We are going to use 80% of the data for training and 20% for testing.

Create the training data set by extracting 615 samples:

```
# head -615 data/pima-indians-diabetes.data > data/training.data
```

Step 4: Extract 153 samples for testing:

```
# tail -153 data/pima-indians-diabetes.data > data/test.data
```

Step 5: Place data on HDFS:

```
# hadoop fs -put data/training.data /mllib-tutorial/input
# hadoop fs -put data/test.data /mllib-tutorial/input
```

Now that the dataset is ready, we are going to build a Naïve Bayesian classifier. First, you should compile a Naïve Bayesian implementation that uses Spark MLlib.

Step 1: Make sure you are in the "cloudapp-mllib-tutorial" directory. You can go to that directory by using the following command:

```
# cd ~/cloudapp-mllib-tutorial
```

Step 2: Find out how the Naïve Bayesian algorithm is developed using MLlib (also available in **Appendix B**):

```
# nano src/NaiveBayesExample.java
```

Step 3: Build the code:

```
# mvn clean package
```

Step 4: Submit the compiled application to Spark to classify the Pima Indians Diabetes dataset and build a model capable of identifying diabetic patients:

```
# spark-submit --class NaiveBayesExample target/mllib.mp-1.0-
SNAPSHOT.jar /mllib-tutorial/input/training.data /mllib-
tutorial/input/test.data
```

Step 5 (optional): After submission, Spark will execute the application and print out the result among logging information. In case you want to only see the result, you can forward the log output to /dev/null:

```
# spark-submit --class NaiveBayesExample target/mllib.mp-1.0-
SNAPSHOT.jar /mllib-tutorial/input/training.data /mllib-
tutorial/input/test.data 2> /dev/null
```

You should see a number about 0.61, which means the classifier has built a model which is capable of labeling sample patients as diabetics or non diabetic with accuracy of 61%. As we pointed out, a good prediction accuracy for this dataset is about 70%-76%. To build better classifiers there are usually three options: 1. Try other algorithms, 2. Add more data with more sample patients (the true meaning of Big Data), and 3. Extend the feature set, meaning you gather more information and measurements for each patient.

Note: We are printing the output on the screen. This is not the right practice with big data; we are only doing it for educational purposes in this tutorial. When you are dealing with large datasets, your clusters will be large and the best practice it to right the output to HDFS and read it as you need.

Appendix A: “KMeansExample.java” Source Code

```
import java.util.regex.Pattern;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;

import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.api.java.function.VoidFunction;
import org.apache.spark.mllib.clustering.KMeans;
import org.apache.spark.mllib.clustering.KMeansModel;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import scala.Tuple2;


public final class KMeansExample {

  private static class ParsePoint implements Function<String, Vector> {
    private static final Pattern SPACE = Pattern.compile(",");

    public Vector call(String line) {
      String[] tok = SPACE.split(line);
      double[] point = new double[tok.length-1];
      for (int i = 1; i < tok.length; ++i) {
        point[i-1] = Double.parseDouble(tok[i]);
      }
      return Vectors.dense(point);
    }
  }

  private static class ParseTitle implements Function<String, String> {
    private static final Pattern SPACE = Pattern.compile(",");

    public String call(String line) {
      String[] tok = SPACE.split(line);
      return tok[0];
    }
  }

  private static class PrintCluster implements VoidFunction<Tuple2<Integer, Iterable<String>>> {
    private KMeansModel model;
    public PrintCluster(KMeansModel model) {
```

```
        this.model = model;
    }

    public void call(Tuple2<Integer, Iterable<String>> Cars) throws Exception {
        String ret = "[";
        for(String car: Cars._2()){
            ret += car + ", ";
        }
        System.out.println(ret + "]");
    }
}


private static class ClusterCars implements PairFunction<Tuple2<String, Vector>, Integer, String> {
    private KMeansModel model;
    public ClusterCars(KMeansModel model) {
        this.model = model;
    }

    public Tuple2<Integer, String> call(Tuple2<String, Vector> args) {
        String title = args._1();
        Vector point = args._2();
        int cluster = model.predict(point);
        return new Tuple2<Integer, String>(cluster, title);
    }
}

public static void main(String[] args) {
    if (args.length < 2) {
        System.err.println(
                "Usage: GaussianMixtureMP <number_of_clusters> <input_file>");
        System.exit(1);
    }
    String inputFile = args[1];
    int k = Integer.parseInt(args[0]);;
    int iterations = 100;
    int runs = 1;

    SparkConf sparkConf = new SparkConf().setAppName("KMeans Example");
    JavaSparkContext sc = new JavaSparkContext(sparkConf);
    JavaRDD<String> lines = sc.textFile(inputFile);

    JavaRDD<Vector> points = lines.map(new ParsePoint());
    JavaRDD<String> titles = lines.map(new ParseTitle());

    KMeansModel model = KMeans.train(points.rdd(), k, iterations, runs, KMeans.RANDOM(), 0);

    JavaPairRDD<Integer, Iterable<String>> clusters =titles.zip(points).mapToPair(new
ClusterCars(model)).groupByKey();
    clusters.foreach(new PrintCluster(model));
```

```
        sc.stop();
    }
}
```

Appendix B: “NaiveBayesExample.java” Source Code

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.mllib.classification.NaiveBayes;
import org.apache.spark.mllib.classification.NaiveBayesModel;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.mllib.regression.LabeledPoint;
import scala.Tuple2;

import java.util.regex.Pattern;

public final class NaiveBayesExample {
      private static class DataToPoint implements Function<String, LabeledPoint> {
         private static final Pattern SPACE = Pattern.compile(",");

         public LabeledPoint call(String line) throws Exception {
            String[] tok = SPACE.split(line);
            double label = Double.parseDouble(tok[tok.length-1]);
            double[] point = new double[tok.length-1];
            for (int i = 0; i < tok.length - 1; ++i) {
               point[i] = Double.parseDouble(tok[i]);
            }
            return new LabeledPoint(label, Vectors.dense(point));
         }
      }

   public static void main(String[] args) {
      if (args.length < 2) {
         System.err.println(
               "Usage: NaiveBayesExample <training_data> <test_data>");
         System.exit(1);
      }
      String training_data_path = args[0];
      String test_data_path = args[0];

      SparkConf sparkConf = new SparkConf().setAppName("NaiveBayesExample");
      JavaSparkContext sc = new JavaSparkContext(sparkConf);

      JavaRDD<LabeledPoint> train = sc.textFile(training_data_path).map(new DataToPoint());
      JavaRDD<LabeledPoint> test = sc.textFile(training_data_path).map(new DataToPoint());
```

```
        final NaiveBayesModel model = NaiveBayes.train(train.rdd(), 1.0);

        JavaPairRDD<Double, Double> predictionAndLabel =
                test.mapToPair(new PairFunction<LabeledPoint, Double, Double>() {
                    public Tuple2<Double, Double> call(LabeledPoint p) {
                        return new Tuple2<Double, Double>(model.predict(p.features()), p.label());
                    }
                });

        double accuracy = predictionAndLabel.filter(new Function<Tuple2<Double, Double>, Boolean>() {
            public Boolean call(Tuple2<Double, Double> pl) {
                return pl._1().equals(pl._2());
            }
        }).count() / (double) test.count();

        System.out.println(accuracy);

        sc.stop();
    }

}
```