**Activity Sheet**

Contents

[Streaming: 1](#_Toc460152391)

[Background: 1](#_Toc460152392)

[Code: 1](#_Toc460152393)

[Clustering: 1](#_Toc460152394)

[Problem Statement: 1](#_Toc460152395)

[SparkMLLib: 2](#_Toc460152396)

[Classification: 3](#_Toc460152397)

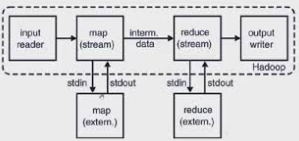
[Problem Statement: 3](#_Toc460152398)

[Decision Trees on SparkMLLib: 3](#_Toc460152399)

[Logistic Regression Spark MLlib: 5](#_Toc460152400)

# Streaming:

### Background:

 Hadoop Streaming is a generic API which allows writing Mappers and Reduces in any language. But the basic concept remains the same. Mappers and Reducers receive their input and output on stdin and stdout as (key, value) pairs. Apache Hadoop uses streams as per UNIX standard between your application and Hadoop system. Streaming is the best fit for text processing. The data view is line oriented and processed as a key/value pair separated by 'tab' character. The program reads each line and processes it as per the requirement.

### Python Streaming:

1. Create directory and load data

hdfs dfs -mkdir Streaming

hdfs dfs -put transactions.csv Streaming

1. Submit job to hadoop:

hadoop jar /usr/lib/hadoop-0.20-mapreduce/contrib/streaming /hadoop-streaming.jar \

-file pathOfFile/mapper.py -mapper mapper.py \

-file pathOfFile/reducer.py -reducer reducer.py \

-input Streaming /transactions.csv -output Streaming/output

Note: pathOfFile should be the actual path of local file where your python code is residing.

-file 🡪 mapper and reducer python files path

-input 🡪 path of input file on HDFS

-output 🡪 path where output is residing

3. check the output

hdfs dfs -cat Streaming/output/part-00000

# Clustering:

### Problem Statement:



Bimbo Bakeries USA focuses on providing the highest quality baked goods at a great value to customers and consumers. With many of America’s most beloved bread and sweet baked goods brands, we are the nation’s leading baking company.

Daily they serve over 10 million stores. Now the challenge is over 1000’s of products, across million stores, accurately serving their needs is a major concern.

To solve this problem, let’s identify similar stores based on transactional, demographic and other features. But can this be done on a single machine, with 1000’s of iterations??

As an alternative, let us try scalable machine learning algorithms on Spark-MLLib.

### SparkMLLib:

Perform the following steps in sequence:

1. Create a HDFS directory with name **“clustering”** under /user/cloudera
2. Now push **“featuresforClustering1\_data”** file in to clustering.
3. To understand the features provided, check **“featuresforClustering1\_names”**
4. Login as root user
5. Fire **spark-shell**
6. Import required packages:

import org.apache.spark.mllib.clustering.KMeans

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.util.MLUtils

1. Load and parse the data

val data = sc.textFile("/user/cloudera/clustering/featuresforClustering1\_data.csv")

val parsedData = data.map(s => Vectors.dense(s.split(',').map(\_.toDouble))).cache()

1. Cluster the data into six classes using KMeans

val numClusters = 6

val numIterations = 10

val clusters = KMeans.train(parsedData, numClusters, numIterations)

1. Evaluate clustering by computing Within Set Sum of Squared Errors

val WSSSE = clusters.computeCost(parsedData)

println("Within Set Sum of Squared Errors = " + WSSSE)

1. Cluster Centers

val clusterCenters1 = clusters.clusterCenters

1. Total Clusters

clusters.k

# Classification:

### Problem Statement:

The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be ('yes') or not ('no') subscribed. The classification goal is to predict if the client will subscribe (yes/no) a term deposit (variable y).

For attribute information, check **“Data set information”, “Bank\_Classification\_names”** files provided.

### Decision Trees on SparkMLLib:

1. Create a HDFS directory **“classification”**
2. Import packages:

import org.apache.spark.mllib.tree.DecisionTree

import org.apache.spark.mllib.tree.model.DecisionTreeModel

import org.apache.spark.mllib.util.MLUtils

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.evaluation.MulticlassMetrics

import org.apache.spark.mllib.linalg.Vectors

1. Load and parse data:

val data = sc.textFile("/user/cloudera/classification/Bank\_Classification\_data.csv")

val parsedData = data.map { line =>

val parts = line.split(';')

LabeledPoint(parts(1).toDouble, Vectors.dense(parts(0).split(',').map(\_.toDouble)))

}.cache()

1. Split parsedData into training (80%), validation (10%) and testing(10%)

val splits = parsedData.randomSplit(Array(0.8, 0.1,0.1), seed = 11L)

val training = splits(0).cache()

val validation = splits(1).cache()

val test = splits(1)

1. Data Check:

training.take(1)

validation.take(1)

1. Train a DecisionTree model:

val numClasses = 2

val categoricalFeaturesInfo = Map[Int, Int]()

val impurity = "gini"

val maxDepth = 5

val maxBins = 32

val model = DecisionTree.trainClassifier(training, numClasses, categoricalFeaturesInfo,

impurity, maxDepth, maxBins)

1. Evaluate model on validation instances and compute test error

val labelAndPreds = validation.map { point =>

val prediction = model.predict(point.features)

(point.label, prediction)

}

1. Metrics:

val metrics = new MulticlassMetrics(labelAndPreds)

1. Confusion Matrix:

val confMat = metrics.confusionMatrix

1. Precision:

val precision = metrics.precision

1. Recall

val recall = metrics.recall

1. FMeasure

val fmeasure = metrics.fMeasure

1. Model

model.toString

model.toDebugString

1. Top Node

model.topNode

### Logistic Regression Spark MLlib:

1. Import packages:

import org.apache.spark.mllib.util.MLUtils

import org.apache.spark.mllib.linalg.{Vector, Vectors}

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.evaluation.MulticlassMetrics

import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS, LogisticRegressionModel}

1. Load and parse data:

val data = sc.textFile("/user/cloudera/classification/Bank\_Classification\_data.csv")

val parsedData = data.map { line =>

val parts = line.split(';')

LabeledPoint(parts(1).toDouble, Vectors.dense(parts(0).split(',').map(\_.toDouble)))

}.cache()

1. Split parsedData into training (80%), validation (10%) and testing(10%):

val splits = parsedData.randomSplit(Array(0.8, 0.1,0.1), seed = 11L)

val training = splits(0).cache()

val validation = splits(1).cache()

val test = splits(1)

1. Data check:

training.take(1)

1. Run training algorithm to build the model:

val model = new LogisticRegressionWithLBFGS().setNumClasses(2).run(training)

1. Compute raw scores on the validation set:

val predictionAndLabels = validation.map { case LabeledPoint(label, features) =>

val prediction = model.predict(features)

(prediction, label)

}

1. Metrics:

val metrics = new MulticlassMetrics(predictionAndLabels)

1. Confusion Matrix:

val confMat = metrics.confusionMatrix

1. Precision:

val precision = metrics.precision

1. Recall:

val recall = metrics.recall

1. FMeasure

val fmeasure = metrics.fMeasure