Introduction to statistical computing in Scala: an application of the k-Nearest Neighbors classifier

Roxana Tesileanu

INCDS 2017

Table of Contents

[Abstract 1](#__RefHeading___Toc655_455876533)

[Getting started 1](#__RefHeading___Toc214_241067100)

[Basic vector operations 2](#__RefHeading___Toc129_1943657966)

[Reading files for classification tasks 3](#__RefHeading___Toc809_1943657966)

[K-Nearest Neighbors (kNN) classification 4](#__RefHeading___Toc1178_1943657966)

[Bibliography 6](#__RefHeading___Toc130_537809076)

[Appendix A: Source file - src/main/scala/mai/scalaML/BasicVectorOP.scala 7](#__RefHeading___Toc1616_1943657966)

[Appendix B: Source file – src/main/scala/mai/scalaML/ReadFile.scala 8](#__RefHeading___Toc1618_1943657966)

[Appendix C: Source file – src/main/scala/mai/scalaML/kNN.scala 9](#__RefHeading___Toc1620_1943657966)

[Appendix D: Source file – src/main/scala/mai/scalaML/AutoNorm.scala 11](#__RefHeading___Toc450_1348730725)

[Appendix E: Source file – src/main/scala/mai/scalaML/TestFrameWorkClassif.scala 12](#__RefHeading___Toc517_478062575)

# Abstract

1. Statistical computing evolves at a high speed, mainly because statisticians have recognized the advantage of being able to design their algorithms according to their needs. The present paper documents the implementation in Scala of the k-Nearest Neighbors (kNN) classifier based on Euclidean distances. It represents the first part of a statistical package called scalaML used for classification, which will include in the future also topics like decision trees and logistic regression.

# Getting started

This work gathers together my learning steps towards using Scala as an environment for statistical computing. My background in statistics and linear algebra helped a lot, so it does assume the interested people passed their statistics courses. The big advantage of Scala is it allows you to express your thoughts in a natural manner, so having a clear idea of what you want to implement really helps a lot.

In order to begin with statistical computing in Scala, you need to install the appropriate tools on your computer. Because different operating systems require different installing procedures, I will let you check the websites of the tools used and extract the information needed to install them on your computer. So, visit the website of the Scala language and of sbt and follow the instructions found there. If you encounter big problems you can grasp to the introduction into the Scala language of Jason Swartz1 and come back later. This is a normal iterative approach used by anyone who wants to get into a new field. The main aim of this work is to serve as documentation for the scalaML package developed by me at INCDS. The GitHub repository of the project can be found at the following link:

[https://github.com/RoxanaTesileanu/multivariate\_analyses/tree/master/DeepLearning/src/main/scala/com/mai/scalaML](https://github.com/RoxanaTesileanu/multivariate_analyses/tree/master/DeepLearning/src/main/scala/com/incds/scalaML).

As mentioned previously, in order to use the scalaML package, you need to install Scala and sbt. Supposed you passed through these initial steps successfully, you can create a new Scala project using sbt. Again, if you encounter big problems, you can check the book of Joshua Suereth and Matthew Farwell2 for an introduction in sbt.

# Basic vector operations

Because most statistical algorithms manipulate datasets which are collections of vectors, manipulating vectors is an essential task. I’ve chosen to implement some of the most widely used vector operations in Scala, because I try to stick to the main Scala types as much as possible while developing ML algorithms. This makes the further development task easier. The vector operations are found in the BasicVectorOP.scala file of the scalaML package and include: vector addition, vector subtraction, elementwise multiplication and dot product for any two vectors of type Array[Double], and matrix multiplication for any two matrices of type Array[Array[Double]], where each Array[Double] represents an observation (i.e. a row of the dataset) and all elements with the same index of all the inside arrays (i.e. of any Array[Double]) represent a column (i.e. a variable of the dataset).

## Vector addition

Suppose you have two vectors a and b of type Array[Double]. You can perform an elementwise addition with the function arrayAdd(a,b). Suppose you’ve started a Scala session with sbt. You can test the function in REPL using the following commands:

scala> import com.mai.scalaML.BasicVectorOP.\_

import com.mai.scalaML.BasicVectorOP.\_

scala> val a = Array(1.0, 2.0, 1.0)

a: Array[Double] = Array(1.0, 2.0, 1.0)

scala> val b = Array(2.0, 0.0, 1.0)

b: Array[Double] = Array(2.0, 0.0, 1.0)

scala> arrayAdd(a,b)

res0: Array[Double] = Array(3.0, 2.0, 2.0)

## Vector subtraction

Using the same two vectors from above you can test the arraySubt() function:

scala> arraySubt(a,b)

res1: Array[Double] = Array(-1.0, 2.0, 0.0)

## Elementwise multiplication of two vectors

Using the same two vectors from above you can test the arrayMultipl() function:

scala> arrayMultipl(a,b)

res2: Array[Double] = Array(2.0, 0.0, 1.0)

## Dot product of two vectors

Using the same two vectors from above you can test the dot() function:

scala> dot(a,b)

res3: Double = 3.0

## Matrix multiplication

Suppose you have two matrices a and b of type Array[Array[Double]]. You can test the matrixMultipl() in REPL:

scala> val a = Array(Array(0.1, 0.2), Array(0.1, 0.2))

a: Array[Array[Double]] = Array(Array(0.1, 0.2), Array(0.1, 0.2))

scala> val b = Array(Array(0.3, 0.4), Array(0.5, 0.6))

b: Array[Array[Double]] = Array(Array(0.3, 0.4), Array(0.5, 0.6))

scala> matrixMultipl(a,b)

res4: Array[Array[Double]] = Array(Array(0.13, 0.16), Array(0.13, 0.16))

# Reading files for classification tasks

You can read txt and csv files for classification tasks using the readFileClassif() function found in the ReadFile object. To use it import the ReadFile object in REPL. The first row of the file to be read is assumed to contain the header and the last column is assumed to contain the labels. The function returns a tuple with the information needed for classification (i.e. a dataMatrix, the dataLabels and the used classes).

scala> import com.mai.scalaML.ReadFile.\_

import com.mai.scalaML.ReadFile.\_

scala> val path = "path to file”

path: String = path to file

scala> val myData = readFileClassif(path, "\t")

myData: (Vector[Array[Double]], Vector[Int], Range) = (Vector(Array(14488.0, 7.153469, 1.673904), Array(26052.0, 1.441871, 0.805124), …), Vector(2, 1, 1, 1, 1, 3, 3, 1, 3, 1,……), Range(1,2,3))

The readFileClassif() function, takes two parameters of type String: the path to the file and the delimitation of the file (i.e. “,” for comma delimited files, or “\t” for tab delimited files). The variable val mydata is the return of the function and represents a tuple of three pieces of information which can be accessed using the tuple indexes :

scala> myData.\_1

res6: Vector[Array[Double]] = Vector(Array(14488.0, 7.153469, 1.673904), Array(26052.0, 1.441871, 0.805124),……

scala> myData.\_2

res7: Vector[Int] = Vector(2, 1, 1, 1, 1, 3, 3, 1, 3, 1, 1, 2, 1, 1, 1, 1, 1, 2, 3, 2, 1, 2, 3, 2, 3, 2, 3, 2, 1, 3, 1, 3, 1, 2, 1, 1,……

scala> myData.\_3

res8: Range = Range(1, 2, 3)

The above pieces of information represent the input for the classification algorithms: a Vector[Array[Double]] which represents the data matrix, a Vector[Int] which represents the labels and a Range which stores the used classes.

# K-Nearest Neighbors (kNN) classification

The object kNN found in the kNN.scala file contains the necessary items for performing a kNN classification. Suppose you take over the information from the above use of the readFileClassif() function. Then your information is packed in a tuple called myData. In order to use the classifykNN() function which represents the kNN algorithm, you reconstruct your dataset using the information from the tuple:

scala> import com.mai.scalaML.kNN.\_

import com.mai.scalaML.kNN.\_

scala> val dataSet = new CreateData(myData.\_1, myData.\_2, myData.\_3)

dataSet: com.mai.scalaML.kNN.CreateData = CreateData(Vector([D@4ce4b393, [D@577cdf65, [D@22c83e6c,……..

Afterwards, you can apply the classifykNN() function, which takes three parameters: one new observation of type Vector[Array[Double]] which needs to be classified, the dataset itself packed in the val dataSet, and the number of nearest neighbors used by the classifier. The classifier returns the class to which the new observation was attributed to.

scala> classifykNN(Vector(Array(14488.0, 7.153469, 1.673904)), dataSet, 3)

res10: Int = 2

## Full example of the use of the kNN classifier:

scala> val dataMatrix = Vector(Array(1.0, 1.1), Array(1.0, 1.0), Array(0.0, 0.0), Array(0.0, 0.1))

dataMatrix: scala.collection.immutable.Vector[Array[Double]] = Vector(Array(1.0, 1.1), Array(1.0, 1.0), Array(0.0, 0.0), Array(0.0, 0.1))

scala> val dataLabels = Vector(1, 1, 2, 2)

dataLabels: scala.collection.immutable.Vector[Int] = Vector(1, 1, 2, 2)

scala> val classes = Range (dataLabels.min, dataLabels.max+1)

classes: scala.collection.immutable.Range = Range(1, 2)

scala> val dataSet = new CreateData(dataMatrix, dataLabels, classes)

dataSet: com.mai.scalaML.kNN.CreateData = CreateData(Vector([D@4fbc0544, [D@631e316c, [D@14cb1821, [D@75b438e2),Vector(1, 1, 2, 2),Range(1, 2))

scala> val result = classifykNN(Vector(Array(0.1, 0.1)), dataSet, 3)

result: Int = 2

scala> val result2 = classifykNN(Vector(Array(1.0, 0.9)), dataSet, 3)

result2: Int = 1

If you decide the variables should be normalized, because they are measured in very different units or scales, you can normalize the dataMatrix before using the kNN classifier, to scale each variable from 0 to 1:

scala> import com.mai.scalaML.AutoNorm.\_

import com.mai.scalaML.AutoNorm.\_

Read your file:

scala> import com.mai.scalaML.ReadFile.\_

import com.mai.scalaML.ReadFile.\_

scala> val path = "file path”

path: String = file path

scala> val myData = readFileClassif(path, "\t") // “\t” if tab-delimited

myData: (Vector[Array[Double]], Vector[Int], Range) = (Vector(Array(14488.0, 7.153469, 1.673904), Array(26052.0, 1.441871, 0.805124), Array(75136.0, 13.147394, 0.428964), Array(38344.0, 1.669788, 0.134296),……

scala> val dataMatrix = myData.\_1

dataMatrix: Vector[Array[Double]] = Vector(Array(14488.0, 7.153469, 1.673904), Array(26052.0, 1.441871, 0.805124), Array(75136.0, 13.147394, 0.428964), Array(38344.0, 1.669788, 0.134296),……

Now, use the autoNorm() function to get the normalized version of the dataMatrix:

scala> val normalizedDataMatrix = autoNorm(dataMatrix)

normalizedDataMatrix: Vector[Array[Double]] = Vector(Array(0.15873259342850568, 0.3419546659888891, 0.9872441587123406), Array(0.2854294260076912, 0.06892523280719681, 0.47449628503016766), Array(0.8232007274878661, 0.6284800736390028, 0.2524892865215854), ……

Use the kNN classifier in the same way as above.

## Computing the error rate of the kNN classifier

You can compute the error rate of the kNN classifier with the testFrameWorkClassif() function, which takes three parameters:

- the dataset of type [CreateData],

- a partition which is a Double from 0 to 1 to indicate the proportion of the test data to the total data (e.g. a partition = 0.3 means 30% of the whole dataset is used in testing), and

- an Int which indicates the number of k nearest neighbors used by the kNN classifier.

# Bibliography

1. Swartz, J. *Learning Scala*. (O’Reilly, 2015).

2. Suereth, J. & Farwell, M. *SBT in action: the simple Scala Build Tool*. (Manning Publications Co., 2016).

3. Odersky, M., Spoon, L. & Venners, B. *Programming in Scala*. (Artima, 2010).

4. Bugnion, P. *Scala for data science*. (Packt Publishing, 2016).

5. Harrington, P. *Machine learning in action*. (Manning Publications Co., 2012).

6. Karau, H., Konwinski, A., Wendell, P. & Zaharia, M. *Learning Spark*. (O’Reilly, 2015).

7. Nicolas, P. *Scala for machine learning*. (Packt Publishing, 2015).

8. Trask, A. *Grokking deep learning*. (Manning Publications Co., 2017).

9. Wilkinson, D. *Statistical computing with Scala: a functional approach to data science*. (2017).

10. Dawkins, P. *Paul’s notes on linear algebra*. (Lamar University, 2005).

11. Quinn, G. & Keough, M. *Experimental design and data analysis for biologists*. (Cambridge University Press, 2002).

12. Venables, W. N. & Ripley, B. D. *Modern applied statistics with S*. (Springer, 2002).

13. Lial, M., Greenwell, R. & Ritchey, N. *Calculus with applications*. (Pearson, 2012).

14. Pfeffer, A. *Practical probabilistic programming*. (Manning Publications Co.).

15. *Scala Standard Library 2.12.0*. (EPFL, 2003-2016). <http://www.scala-lang.org/api/2.12.0/scala/index.html>

# Appendix A: Source file - src/main/scala/mai/scalaML/BasicVectorOP.scala

package com.mai.scalaML

/\*

The BasicVectorOP object offers basic operations for two arrays: dot product, elementwise addition, subtraction, and multiplication. I will extend it to matrices multiplication.

\*/

object BasicVectorOP {

def dot (a : Array[Double], b: Array[Double]) : Double = {

assert(a.length == b.length, "The dot product cannot be computed!")

val pairs = a zip b

val pairProd = for (p <- pairs) yield {p.\_1 \* p.\_2}

val dot = pairProd.sum

dot

}

def arrayOP(a: Array[Double], b: Array[Double], f: (Double, Double) => Double) = {

val pairs = a zip b

val OP = for (p <- pairs) yield {f(p.\_1, p.\_2)}

OP

}

def arrayAdd (a: Array[Double], b:Array[Double]) : Array[Double] = {

assert(a.length == b.length, "Arrays cannot be added!")

val elementwiseAddition = arrayOP(a,b, \_ + \_)

elementwiseAddition

}

def arraySubt (a: Array[Double], b: Array[Double]) : Array[Double] = {

assert(a.length == b.length, "Arrays cannot be subtracted!")

val elementwiseSubtraction = arrayOP(a,b, \_ - \_)

elementwiseSubtraction

}

def arrayMultipl (a: Array[Double], b:Array[Double]) : Array[Double] = {

assert(a.length == b.length, "Elementwise multiplication is not possible!")

val elementwiseMultipl = arrayOP(a,b, \_ \* \_)

elementwiseMultipl

}

def matrixMultipl (a:Array[Array[Double]], b: Array[Array[Double]]) : Array[Array[Double]] = {

val tb = b.transpose

assert( tb(0).length == a(0).length, "Matrices cannot be multiplied!")

val c = for (i <- a) yield (tb.map { j => dot(j, i) })

c

}

}

/\* References:

Dawkins Paul 2005 - “Paul’s notes on linear algebra*”*, Lamar University, http://tutorial.math.lamar.edu

Swartz Jason 2015 - "Learning Scala", Manning Publications Co., Shelter Island

Trask Andrew 2017 - "Grokking deep learning", Manning Publications Co., Shelter Island

Odersky Martin, Spoon Lex, Venners Bill 2010 - "Programming in Scala", Second Edition, Artima, Walnut Creek

\*/

# Appendix B: Source file – src/main/scala/mai/scalaML/ReadFile.scala

/\* The function readFileClassif() reads txt and csv files. The first row is assumed to contain the header and the last column is assumed to contain the labels. It returns a tuple with the information needed for the kNN classifier.

\*/

package com.mai.scalaML

object ReadFile {

import scala.math.\_

def readFileClassif (filename: String, delim: String) : (Vector[Array[Double]], Vector[Int], Range) = {

val src = scala.io.Source.fromFile(filename)

val data = src.getLines.map(\_.split(delim)).toArray.drop(1)

src.close()

val len = data(0).length

val data2 = data.map( for (i <- \_) yield(i))

val data3= data2.map( for (i <- \_) yield (i.toDouble))

val dataMatrix= data3.map{ case x => x.take(len-1)}.toVector

val dataLabels = data3.map{ case x => x.last.toInt}.toVector

val classes = Range(dataLabels.min, dataLabels.max+1)

(dataMatrix, dataLabels, classes)

}

}

/\* References:

Peter Harrington - "Machine Learning in Action", 2012

Jason Swartz - "Learning Scala", 2015

Martin Odersky, Lex Spoon, Bill Venners - "Programming in Scala", Second Edition, 2010

\*/

# Appendix C: Source file – src/main/scala/mai/scalaML/kNN.scala

package com.mai.scalaML

object kNN{

import scala.math.\_

import scala.util.Sorting

import com.mai.scalaML.BasicVectorOP.\_

val dataMatrix = Vector(Array(1.0, 1.1), Array(1.0, 1.0), Array(0.0, 0.0), Array(0.0, 0.1))

val dataLabels = Vector(1, 1, 2, 2)

val classes = Range (dataLabels.min, dataLabels.max+1)

case class CreateData (dataMatrix: Vector[Array[Double]], dataLabels: Vector[Int], classes: Range)

val dataSet = new CreateData(dataMatrix, dataLabels, classes)

def classifykNN (P:Vector[Array[Double]], dataSet: CreateData, k: Int) : Int={

def distance (P1: Vector[Array[Double]], P2: Vector[Array[Double]]) : Double = {

val featuresP1 = for (i <- P1(0)) yield i

val featuresP2 = for (i <- P2(0)) yield i

val pointDiff = arraySubt(featuresP1, featuresP2)

val pointDiffPow = for (i <- pointDiff) yield pow(i,2)

val d = sqrt(pointDiffPow.sum)

d

}

def sortingDist (dataSet: CreateData) : Vector[(Double, Int)] = {

val distances = dataSet.dataMatrix map(x => distance(P, Vector(x)))

val labeledDist = distances zip (dataSet.dataLabels)

val sortedDist = labeledDist.sortBy(\_.\_1)

sortedDist

}

val sortedDist = sortingDist(dataSet)

def countingClasses (sortedDist: Vector[(Double, Int)]) : Int = {

val kN = sortedDist.slice(0, k)

val countClasses =for (i <- dataSet.classes) yield(kN.count(\_.\_2 == i))

val classesAndCounts = classes zip countClasses

val result = classesAndCounts.sortBy(\_.\_2).last.\_1

result

}

return(countingClasses(sortedDist))

}

//trying out the kNN algorithm:

val result = classifykNN(Vector(Array(0.1, 0.1)), dataSet, 3)

}

/\* References:

Machine Learning in Action - Peter Harrington, 2012

Programming in Scala - Martin Odersky, Lex Spoon, Bill Venners, 2010

Learning Scala - Jason Swartz, 2015

\*/

# Appendix D: Source file – src/main/scala/mai/scalaML/AutoNorm.scala

package com.mai.scalaML

object AutoNorm {

def autoNorm (dataMatrix : Vector[Array[Double]]) : Vector[Array[Double]] = {

val len = dataMatrix(0).length

val variables = for (i <- 0 to (len -1)) yield { for (a <- dataMatrix) yield a(i) }

val normalizedVariables = for (v <- variables) yield { v.map{ a => (a - v.min)/(v.max - v.min)}}

val minMaxRange = for (v <- variables) yield { (v.min, v.max, v.max - v.min) }

val len2 = normalizedVariables(0).length

val normalizedDataMatrix = for (i <- 0 to (len2-1)) yield { for (nv <- normalizedVariables) yield (nv(i))}

val normalizedDataMatrix2 = normalizedDataMatrix.map{ v => v.toArray}.toVector

normalizedDataMatrix2

}

}

/\* References:

Harrington Peter 2012 - "Machine learning in action", Manning Publications Co., Shelter Island

Odersky Martin, Spoon Lex, Venners Bill 2010 - "Programming in Scala", Second Edition, Artima, Walnut Creek

\*/

# Appendix E: Source file – src/main/scala/mai/scalaML/TestFrameWorkClassif.scala

/\* In order to test the kNN classifier you can use the testFrameWorkClassif() function. The error rate is given by the total number of errors (misclassified observations) divied by the total number of tested observations. The testFrameWorkClassif() function takes three parameters: the dataset of type [CreateData], a partition from 0 to 1, which indicates the proportion of test to total data (e.g. partition= 0.3 means 30% of the whole dataset) and an Int which indicates the number of k nearest neighbors used by the kNN classifier.

\*/

package com.mai.scalaML

object TestFrameWorkClassif {

import scala.util.Random.\_

import com.mai.scalaML.kNN.\_

def testFrameWorkClassif ( dataSet: CreateData, partition : Double, k :Int) : Double = {

val dataMatrix = dataSet.dataMatrix

val dataLabels = dataSet.dataLabels

val dataPlusLabels = dataMatrix.zip(dataLabels)

val shuffledDataPlusLabels = shuffle(dataPlusLabels)

val lenDataSet = shuffledDataPlusLabels.length

val testAndClassSets = shuffledDataPlusLabels.splitAt((lenDataSet.toDouble \* partition).toInt)

val testDataSet = testAndClassSets.\_1

val classifDataSet = testAndClassSets.\_2

val testDataMatrix = testDataSet.map{ x => x.\_1 }

val testDataLabels = testDataSet.map{ x => x.\_2 }

val classesTest = dataSet.classes

val usedTestDataSet = new CreateData(testDataMatrix, testDataLabels, classesTest)

val classifDataMatrix = classifDataSet.map{ x => x.\_1 }

val classifDataLabels = classifDataSet.map{ x => x.\_2 }

val usedClassifDataSet = new CreateData(classifDataMatrix, classifDataLabels, classesTest)

val resultsTestData = testDataMatrix.map{ x => classifykNN(Vector(x), usedClassifDataSet, k) }

val errorRateTuple = resultsTestData.zip(testDataLabels)

val errorRateMatch = errorRateTuple.map{ x => if (x.\_1 == x.\_2) 1 ; else 0 }

val goodResults = errorRateMatch.count( \_ == 1)

val wrongResults = errorRateMatch.count( \_ == 0)

val errorRate = wrongResults.toDouble / (testDataSet.length).toDouble

errorRate

}

}

/\* References:

Harrington Peter 2012 - "Machine learning in action", Manning Publications Co., Shelter Island

Odersky Martin, Spoon Lex, Venners Bill 2010 - "Programming in Scala", Second Edition, Artima, Walnut Creek

Bugnion Pascal 2016 - "Scala for Data Science", Packt Publishing, Birmingham

Scala Standard Library 2.12.0 -scala.util.Random - http://www.scala-lang.org/api/2.12.1/scala/util/Random.html

\*/