Big Data Technologies

Spark Project

Date: 26/October/2016

GALDO SEARA Luis

GONZÁLEZ HUESCA Juan Manuel

ISSARANE Hausmane

Abstract

In this project, Apache Spark is used to develop a natural language processing.
Apache Spark is a fast and general engine for large-scale data processing, as we can see on their website (http://spark.apache.org/).

Natural Language processing is a multi-disciplinary field related to computer science and artificial intelligence. It is used in many application such as machine translation, information extraction, summarization or question answering.

The given Ling-Spam email dataset contains two directories, “spam” and “ham”, containing the spam and legitimate emails, respectively. Natural Language processing is applied to develop an algorithm that will obtain the keywords from the given emails to classify them.

Code

Download of data set

A data set called "*ling-spam.zip*" containing Ham and Spam emails was downloaded from the Jalon website, then it was uploaded to the local file system using the Ambari website (http://127.0.0.1:8080/). Once there the zip file was moved to HDFS using the command line and unzipped in the final destination (hdfs dfs -ls /tmp/ling-spam).

How to execute

The project was compiled and executed in the scala version "2.10.4". In a first moment, it was compiled in the scala version "2.11.8" but one of the lines, *down.collect.toArray.toMap* , could not be executed in this version so, following the teacher's recommendation, the previous version was used.

The directory of the project is SpamFilter, and it follows the typical layout of a sbt directory. To be able to compile and execute the project the indications of https://spark.apache.org/docs/1.4.1/quick-start.html were followed.

This is the execution line of the project:
/usr/bin/spark-submit --class "spamFilter" --master local[4]
target/scala-2.10/spamfilter_2.10-0.1-SNAPSHOT.jar

probaWordDir

//The first step is to gather the content of the files (emails) of the indicated directory. That is kept in the variable *files*. This variable is an RDD with structure [(String,String)] so in this case [(file name, content of the file)]

val files = sc.wholeTextFiles(filesDir)

//The number of the files in the directory is saved in the variable *nbFiles*. This variable is type "long".

val nbFiles = sc.wholeTextFiles(filesDir).count

//Then get all the unique words per file (email) and store in the variable *uwpf*. This variable is an RDD with structure [String]

val uwpf = files.flatMap(f=>f._2.split("\\s+").distinct)

//After, the count of the number of files where each word occurs is stored in the variable *wordDirOccurency* with RDD structure [(String, Int)], in this case [(word),(number of files where it occurs)]. Then an array called *filt* is created containing all the characters that need to be removed. This array is converted into a 2D-array where the first position is a character and the second one is zero, which is then converted into an RDD using *parallelize*.

var wordDirOccurency = uwpf.map(f=>(f,1)).reduceByKey(_+_)

val filt = Array(".",":",",","," ","/","\\","-","'","(",")","@")

val ma = filt.map(f=>(f,0))
val nuevo = sc.parallelize(ma)

//Once the RDD with the characters to be removed is available, *subtractByKey* is used and the final *wordDirOccurency* is obtained.

val wordDirOccurency = wordDirOcurrency.subtractByKey(nuevo)

//Finally, the probability of occurrences of a word is stored in the probaWord RDD with structure [(String,Double)], in this case [(word,probability)].

val probaWord = wordDirOccurency.mapValues(a=>a/nbFiles.toDouble)

//The variables *probaWord* and *nbFiles* are returned.

return (probaWord, nbFiles)

computeMutualInformationFactor

//At this point, the probability that a word occurs (or not) in a given class is obtained. To get it, fullOuterJoin is applied to *probaW* and *probaWC*. After this, an RDD with the structure [(String,(Option[Double],Option[Double]))] is created. Then, using getOrElse, if the first optional value is *Some* then is kept, otherwise, the value assigned is *probaDefault* (whose value is already defined). The new RDD has a structure [(String, Double)].

val probaWordMerge = probaWC.fullOuterJoin(probaW)
val probaWordMerged =
probaWordMerge.mapValues(f=>f._1.getOrElse(probaDefault))

//The result of the operation *P(occurs)*P(class)*, which is the denominator of the logarithm part of the mutual information factor formula, is stored in the variable *down*. This variable has an RDD format [(String, Double)]. The variable *down* is converted into a map so the value can be accessed through the key, resulting in a new variable called *down2*.

val down = probaW.mapValues(f=>f*probaC)
val down2 = down.collect.toArray.toMap

//The result of the division from the logarithm part of the equation is stored in the variable *inside* with RDD structure [(String, Double)].

val inside = probaWordMerged.map(f=>(f._1,(f._2/down2(f._1))))

//After having the result of the division, following the formula, the logarithm base 2 is applied to *inside*, resulting in an RDD (*outside*) with the same structure as the one before. This variable is converted into a map for the same reasons as *down2*, having a new variable called *outside2*.

val outside =
inside.mapValues(a=>(java.lang.Math.log(a))/(java.lang.Math.log(2)))
val outside2 = outside.collect.toArray.toMap

//To finish the calculation of the mutual information factor formula, a multiplication is required between *probaWordMerged* and the result of the logarithm operation, *outside2*.

val fin = probaWordMerged.map(f=>(f._1,f._2*outside2(f._1)))

//An RDD with structure [(String, Double)] is returned.

return fin

main function

//A park context is created following the example from the website provided (https://spark.apache.org/docs/1.4.1/quick-start.html).

val conf = new SparkConf().setAppName("Spam Filter")

val sc = new SparkContext(conf)

//The function *probaWordDir* is called with the folders Spam and Ham to obtain *probaWord* and *nbFiles* of each of them. The variable *totalFiles* is the sum of the total number of files between Ham and Spam.

val (probaWordHam, nbFilesHam) =
probaWordDir(sc,"/tmp/ling-spam/ham/*.txt")
val (probaWordSpam, nbFilesSpam) =
probaWordDir(sc,"/tmp/ling-spam/spam/*.txt")
val totalFiles = nbFilesHam + nbFilesSpam

//The value of the variable *probaDefualt* is calculated using the information provided in the project description.

val probaDefault = 0.2/totalFiles.toDouble

//The probabilities of words not occurring in a given class is calculated by the complementary value of the probability that it occurs.

val probaWordHamFalse = probaWordHam.mapValues(f=>1.0-f)
val probaWordSpamFalse = probaWordSpam.mapValues(f=>1.0-f)

//Two new variables, *wordHam* and *wordSpam*, are created, containing the number of times each words occurs. This contains the same information as *wordDirOcurrency*. Once this is obtained, both variables are merged using *union* and using *reduceByKey*. those words that appear in both classes will become one only key, and as a value, they will have the total number of appearances.

val wordHam = probaWordHam.mapValues(f=>f*nbFilesHam.toDouble)
val wordSpam = probaWordSpam.mapValues(f=>f*nbFilesSpam.toDouble)
val words = wordHam.union(wordSpam)
val words2 = words.reduceByKey(_+_)

//To calculate the probability of a word occurring in any given class, *words2* is divided by the total number of files, *totalFiles*. The probability of a word not occurring in any given class is the complementary of *probaW*, and its value is kept in *probaWFalse*.

val probaW = words2.mapValues(a=>a/totalFiles.toDouble)
val probaWFalse = probaW.mapValues(a=>1.0-a)

//The probability *P(class)* is calculated at this point, taking into account the number of files for each class, ham and spam.

val probaCHam = nbFilesHam/totalFiles.toDouble
val probaCSpam = nbFilesSpam/totalFiles.toDouble

//The main for RDDs with structure [(String, Double)] are created at this point, one for each combination of occurrence and the class (True and Ham, False and Ham, True and Spam, False and Spam).

val (trueHam) =
computeMutualInformationFactor(probaWordHam,probaW,probaCHam,probaDefault)

val (falseHam) = computeMutualInformationFactor(probaWordHamFalse,probaWFalse,probaCHam,probaDefault)

val (trueSpam) = computeMutualInformationFactor(probaWordSpam,probaW,probaCSpam,probaDefault)

val (falseSpam) = computeMutualInformationFactor(probaWordSpamFalse,probaWFalse,probaCSpam,probaDefault)

//Once these four values have been obtained, the sum to obtain the Mutual Information (MI) of each word can be computed. To do this, *union* is used to put together all the results and *reduceByKey* to add the values of those whose key is the same.

val fin = trueHam.union(falseHam).union(trueSpam).union(falseSpam)

val finfin = fin.reduceByKey(_+_)

//The next step is to organize them in a way that if we print the 10 first, those are the ones with the highests values.

val fin2 = finfin.takeOrdered(10)(Ordering[Double].reverse.on(x=>x._2))

fin2.take(10).foreach(println)

//Now, *fin2* is converted in to a RDD so it can be put in a file.

val fin3 = sc.parallelize(fin2)

//The results are stored in topWords.txt. Coalesce is used so all the info it is created in one file, and not several.

fin3.coalesce(1,true).saveAsTextFile("/tmp/topWords.txt")

Results

Content of topWords.txt
(Subject:,NaN)
(language,3.969966048863867)
(!,3.920182208865646)
(university,3.6748385285772716)
(free,3.584507179516706)
(remove,3.5498873467221648)
(money,3.3425600782307194)
(our,3.278757108578538)
(english,3.272218788750045)
(click,3.2721287884230525)

Analysis

The purpose of the project was to extract the most informative words in the data set (whether they are key in spam or ham), action which is called the Spam Filter. The mutual information factors gave us the weight that each word has in the whole data set, and starting from there we can dismiss all the uninformative words as they are very common and could be found in any ham or spam email. So, after executing the code we obtained the mutual information factors for all of the words in the data set and ordering by bigger to smaller value we can filter the top most informative words, which can be used to create an email classifier to filter the spam emails and send them to a different folder than the inbox.
As it can be seen in the results above, some key words are language, university, free, money… that could be easily classified into one group or another.
There is one result, *Subject:*, whose result in NaN. This is because the probability of it happening is 1, so the complementary is 0. When this probability is applied to the formula, the final result of it is NaN, since there can not be a division where the denominator is 0.
This word should not be taken into account to decide if an email belongs to ham or spam.

Usage

The list of top words obtain in the spam filter can be used to classify the emails as legitimate or spam. First, it's important to determine to which class each of the top words belong to, this can be done by checking for each word the probability that they have in every class independently, in this case the probability for words in ham can be checked in "*probaWordHam*" and the probability for words in spam can be checked in "*probaWordSpam*", and the class can be determined where the word has the highest probability. Once the class of every top word is known we can start to classify the emails following the next steps:

1. When an e-mail is received, it will be cleaned and lemmatized.
2. All the words from the email will be taken and compared with the top words from the Spam Filter.

3. Every time there is a match between a word in the email and a word in the top words from the Spam filter we will add in the summatory the mutual information factor to the right group, whether ham or spam.
4. At the end we will have the total of the summatory from the mutual information factors depending how many words and their factors for every group, ham or spam, and depending of the total value we can define if the email is spam or ham.
5. For obvious reasons it's very important not to make the mistake to classify a ham email in the spam folder, so in this case when an email is classified as spam we will introduce a decision factor (K, to be determined after further testing) to determine an acceptable ratio to classify the email as spam, in the case the email is classified as ham from the beginning it will be kept there. To determine if an email is spam, the summatory of the mutual information factors of spam should be equal or bigger than K times the summatory of the mutual information factors of ham.

This would be one of the solutions to compute an email and classify it in the right folder, but there are plenty of options to classify the emails based on the spam filter top most informative words, and also using different algorithms; the best one would depend on each particular needs.

Conclusion

The project was very interesting and challenging; we were able to implement the knowledge we acquired during the lectures and the practical sessions but we also had to do some research to understand better the goal of the project and to be able to complete it. Along the way, we consolidated the programming skills in scala and now we are able to do more complex project and exploit all the possibilities of the language in the data science field.
Seeing this project as one that could be applied in real life called our attention and made as perform our bests in order to finish it in the best possible way.
We are looking forward for the second part of this subject to reinforce our knowledge and discover more of this big data world.
This project represents the beginning of our journey as student of the master in data science, as we want to focus and specialize on this field, and find in the future new ways to exploit all the available data and information in order to make the life better.