TF-IDF PROJECT

April 1, 2018

AUDIGIER Benoit - BOISSONNET Priscille - NIELLY Cyprien

1 Introduction on TF-IDF

1.1 Description

TF-IDF = Term Frequency-Inverse Document Frequency

This statistical mesure is used to evaluate the importance of a word in document, in comparison to all the words that are in a collection of documents. The TF-IDF scores are usually combined in a matrix, with columns containing the words and rows containing the documents. The value at each position should correspond to the importance of the column's term to the row's document.

This formula is often used in Natural Language Processing projects.

1.2 The Formula

They are various formulas that can be used to compute the TF_IDF.

We chose to focus on two most commonly used ones. The first one is the most frequently used, whereas the second one is less used but more meaningfull (see "Uses").

- First formula: $TF - IDF = WordCount * \log(1 + \frac{N}{DocWord})$
- Second formula: $TF - IDF = WordCount/Ndoc * \log(1 + \frac{N}{DocWord})$

Where: * *WordCount* is the number of time the word appears in the document * *Ndoc* is the total number of words in the document * *N* is the number of documents in the collection * *DocWord* is the number of documents containing the word

We add +1 in the log, in order to avoid avoir a null result when a word is contained in all documents.

2 Presentation of the test set of documents

We will generate several sets of testing files using Lorem ipsum generator, each one a set of 5 files with the number of words doubling from one set to the next one.

We begin at 50 words, and end up at 1600 by document.

3 MapReduce resolution

We first try to slove this problem with MapReduce, running on Hadoop.

3.1 First formula

For this formula, we have implemented two mappers and two reducers. We describe here the global logics behind these algorithms. More details are provided in the comments of each code.

- 1st mapper : produce as outputs : (word, namefile)
- 1st reducer : produce two kinds of outputs :
  - (word. , number of files that contain this word)
  - (word.namefile, word count in this namefile)

Thanks to the shuffle and sort, we will read in a first time the number of files that contain this word at the beginning. Then, we'll read the word count for a given word in a given document. We will be able thus, be able to compute the TF IDF.

Nevertheless, we have to be sure that all the keys, which begin by the same word, will be read by the same reducer task.

For that, we precise it for the execution with the following syntax to launch the code :

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar **-D map.output.key.field.separator=. -D mapred.text.key.partitioner.options=-k1,2** -input /user/hadoop/wc/output_temp -output /user/hadoop/wc/output -file /home/hadoop/mapper_id.py -mapper /home/hadoop/mapper_id.py -file /home/hadoop/new_reducer2.py -reducer /home/hadoop/new_reducer2.py

So, now, we implement the second mapper-reducer pair.

- 2nd mapper : mapper identity
- 2nd reducer : produce one kind of output : (word|namefile, tf-idf)

1st mapper

```
In [7]: #!/usr/bin/env python
        import sys
        import os
        import fileinput
        import re

        for line in sys.stdin:
            #get file names
            filepath = os.environ['map_input_file']
            namefile = os.path.split(filepath)[-1]
            namefile = namefile.split(".")[0]

            #in case of local path:
            #l = input_file.split("/")
            #namefile = l[len(l)- 1]

            #remove leading and trailing whitespace
            line = line.strip()
            #split the line into a table
```

```
        words  = line.split()

        for word in words :
            #cleaning steps:
            #1 - lower case
            word = word.lower()
            #2 - remove special characters
            word = re.sub('\W+', ' ', word)
            #3 - remove numbers
            word = re.sub(r'[0-9]+', ' ', word)
            #4 - remove leading and trailing whitespace after the cleaning
            word = word.split()

            for w in word :
                #print the couples (key = word, value = namefile)
                print('%s\t%s' % (w, namefile))
```

1st reducer

```
In [10]: #!/usr/bin/env python
         from operator import itemgetter
         import sys

         #initialize variables
         current_word = None
         current_doc = None
         #current_count is the word count of a word, in a given document
         current_count = 0
         #total_word counts is, for a given word, the number
         #of documents that contain this word
         total_word = 0

         for line in sys.stdin:
             #remove leading and trailing whitespace
             line = line.strip()
             #parse the input we got from the mapper
             word, doc = line.split('\t', 1)
             if (current_word == word):
                 if(current_doc == doc):
                     #we have the same word in the same document, we update the word
                     #count
                     current_count += 1
                 else:
                     #the same word but in another document,
                     #we print the word count of the current document
                     print('%s\t%s' % (current_word+"."+current_doc, current_count))
                     #initialize a new word count for this document
                     current_count = 1
```

```
                current_doc = doc
                #add +1 in total_word as the word is seen in a new document
                total_word +=1

        else :
            #the word has changed
            if current_word:
                #it is not the first line
                #print the word count for the current document
                print('%s\t%s' % (current_word+"."+current_doc, current_count))
                #print total_word for the current_word
                print('%s\t%s' % (current_word+".", total_word))
            #update the variables for this new word
            current_word = word
            current_doc = doc
            current_count = 1
            total_word = 1

        if(current_word == word):
            if(current_doc == doc):
                print('%s\t%s' % (current_word+"."+current_doc, current_count))
                print('%s\t%s' % (current_word+".", total_word))
            else:
                print('%s\t%s' % (current_word+".", total_word))
```

2nd mapper

```
In [11]: #!/usr/bin/env python
         import sys

         for line in sys.stdin:
             # split the line into key value
             key, value  = line.split()
             # print the exact same (key, value) couple
             print('%s\t%s' % (key, value))
```

2nd reducer

```
In [12]: #!/usr/bin/env python
         from operator import itemgetter
         from math import log
         import sys

         #we have two types of inputs for reducer2
             #(word.name_of_the_document, word count in this document)
             #(word., number of documents that contain this word)
             #N the number of documents
         N = 5
```

```
for line in sys.stdin:
    #remove leading and trailing whitespace
    line = line.strip()
    #parse the input we got from the mapper
    key, value = line.split('\t', 1)
    word, namefile = key.split('.',1)
    #convert value into an integer
    value = int(value)

    if (namefile == ""):
        total_doc_per_word = value

    else:
        #compute the TF-IDF
        tfidf = value*log(1+N/total_doc_per_word)
        #print the result
        print('%s\t%s' % (word+"|"+namefile, tfidf))
```

We obtain thus, at the end the following result :

3.2 Second formula

For this formula, we have identically two mappers and two reducers. We describe here the global logics behind these algorithms. More details are provided in the comments of each code.

- 1st mapper : produce two kinds of outputs : (1 | word, namefile) or (2 | namefile, word)

1 and 2 allows us to detect in which case we are, whether the word is the key and the namefile is the value, or the opposite. The idea is to have the words as keys in order to compute the word frequencies for example. On the same time, we also have the namefile as keys in order to count the number of words per document.

- 1st reducer : produce three kinds of outputs : - (1 | namefile, total word count in this namefile) - (word | , number of files that contain this word) - (word | namefile, word count in this namefile)

Thanks to the shuffle and sort, we will read in a first time the 1 | namefile keys : so, we collect the information about the documents at the beginning. Then, for a given word, we'll read the number of files that contain this word at the beginning. Finally, we'll read the word count for a given word in a given document. As before we will have read the total word count for this document and the number of files that contain this word, we have all the necessary informations to compute the TF - IDF

So, now, we implement the second mapper-reducer pair.

- 2nd mapper : mapper identity
- 2nd reducer : produce one kind of output : (word | namefile, tf-idf)

1st mapper

In [3]:
```
#!/usr/bin/env python
import sys
import os
import fileinput
import re

for line in sys.stdin:
    #get file names
    filepath = os.environ['map_input_file']
    namefile = os.path.split(filepath)[-1]
    namefile = namefile.split(".")[0]

    #remove leading and trailing whitespace
    line = line.strip()
    #split the line into a table
    words  = line.split()

    for word in words :
        #cleaning steps:
        #1 - lower case
        word = word.lower()
        #2 - remove special characters
        word = re.sub('\W+', ' ', word)
        #3 - remove numbers
        word = re.sub(r'[0-9]+', ' ', word)
        #4 - remove leading and trailing whitespace after the cleaning
        word = word.split()

        for w in word :
            #case 1: key is the word, value is the namefile
            print('%s\t%s' % ("1"+"|"+w, namefile))
            #case 2: key is the namefile, value is the word
            print('%s\t%s' % ("2"+"|"+namefile, w))
```

1st reducer

In [13]:
```
#!/usr/bin/env python

from operator import itemgetter
import sys


#After the mapper and shuffle and sort, we have all the case 1 couples
#then all the case 2 couples.

#Initialisation of variables
current_word = None
current_doc = None
```

```
#current_count is the word_count of a word, in a given document in
#case 1. It is the number of words in a given document in case 2.
current_count = 0
#total_word counts, for a given word, the number of documents where
#it is present
total_word = 0
# change_case : boolean who tells when we move from case 1 to case 2
change_case = False


for line in sys.stdin:
    line = line.strip()

    # parse the input we got from mapper.py
    key, value = line.split('\t', 1)
    cas, true_key = key.split('|',1)

    case = int(case)

    #First, we are in case 1, where key = word and value = filename
    if(case==1):
        word=true_key
        doc = value

        if (current_word==word):
            if(current_doc!=doc):
                #here, we have the same word but in another document
                #We print the word_count of the current_doc
                print('%s\t%s' % (current_word+"|"+current_doc, current_count))
                #then, we initialise a new word_count for this document
                current_count=1
                current_doc=doc
                # We add +1 in total_word as the word is seen in a new docuement
                total_word +=1

            else:
                #here, we have the same word in the same document, we update our
                #word count
                current_count += 1

        else :
            if current_word:
                #here, we change the word, and current_word isn't None (means
                #that we are not in the first line)
                # Print the word_count for the current_document
                print('%s\t%s' % (current_word+"|"+current_doc, current_count))
                # Print the total_word for the current_word
                print('%s\t%s' % (current_word+"|", total_word))
```

```
                    #We update our variables for this new word
                    current_word = word
                    current_doc=doc
                    current_count=1
                    total_word=1


            # Case 2, where key = filename and value = word
            if(case == 2):

                if (change_case==False):
                    # we enter here if it's the first time we deal with case 2
                    # we print the last word of case 1
                    if (current_word==word):
                        print('%s\t%s' % (current_word+"|"+current_doc, current_count))
                        print('%s\t%s' % (current_word+"|", total_word))
                    #Then, we re-initialise our variables
                    current_word=None
                    current_doc = None
                    current_count = 0
                    #Boolean change_case equals now, True
                    change_case=True

                #Case 2 : we read the word and the doc
                word = value
                doc = true_key

                if (current_doc==doc):
                    #here, we have a new word, in the same document. We add +1 to
                    #current_count that counts the number of words in the current_doc
                    current_count +=1

                else:
                    if(current_doc):
                        #we change of document, we print the current_count
                        print('%s\t%s' % ("1"+"|"+current_doc, current_count))
                    current_doc = doc
                    current_count=1

        if(change_case):
            if (current_doc ==doc) :
            #We print the last line
                print('%s\t%s' % ("1"+"|"+current_doc, current_count))
```

2nd mapper

```
In [4]: #!/usr/bin/env python
        import sys
```

```
for line in sys.stdin:
    # split the line into key value
        key, value  = line.split()
        # print the exact same (key, value) couple
        print('%s\t%s' % (key, value))
```

2nd reduce

```
In [5]: #!/usr/bin/env python
        from operator import itemgetter
        from math import log
        import sys

        #we have three types of inputs for reducer2
                #(1|namefile, total word count in this namefile)
                #(word| , number of files that contain this word)
                #(word|namefile, word count in this namefile)

        #Thanks to the shuffle and sort, informations about the namefile are firstly
        # as the key begins with "1|"
        #Then, for a given word, we read the number of namefiles with this word,
        #before the various word counts

        #creation of a dictionary that summarizes all the informations about the
        #namefiles
        dict_namefile = {}
        #N the number of documents
        N = 5

        for line in sys.stdin:
                #remove leading and trailing whitespace
            line = line.strip()
            #parse the input we got from the mapper
            key, value = line.split('\t', 1)
            #avant is the first part of the key, apres is the second part of the key
            avant, apres = key.split('|',1)

            #convert value into an integer
            value = int(value)

            if (avant == "1"):
                namefile = apres
                nbre_doc = value
                dict_namefile[namefile] = nbre_doc

            else :
                #second case, dealing with words
```

```
        if (apres == ""):
            #here, new word
            #keep the value of the number of documents containing this word
            total_doc_per_word = value
        else :
            #here, same word but new document
            #read the word and the namefile
            word = avant
            namefile = apres
            #from the dictionary, retrieve the informations about the
            #number of words in this document
            nbre_doc = dict_namefile[namefile]

            #computation of the TF-IDF
            tfidf = value/nbre_doc*log(1+N/total_doc_per_word)
            #print the result
            print('%s\t%s' % (word+"|"+namefile, tfidf))
```

4 Spark resolution

In spark, the problem of the formula is much easier to deal with. The only difference is the line to compute the number of document where each word appears, and the calculation of the tf idf formula itself.

The idea is quite simple: we map all the words with the document they are in, ("document", "word"), and the only tricky part is when computing the number of document the word is present in, then we reverse the key and the value, apply a distinct and count by key : `u.distinct().map(lambda (title,word):(word,title)).countByKey()`

The formula can be changed here in the tf_idf_calc function (commented on the code below).

```
In [ ]: from pyspark import SparkConf, SparkContext
        conf = SparkConf().setAppName("tfidf calculator")
        sc = SparkContext(conf=conf)

        import re
        import numpy as np


        # Tokenization of the sentences using regular expression
        def tokenize(s):
            return re.split("\\W+", s.lower())

        # Computation of the tf_idf score using the term frequency of each
        # ('document', 'word') and the document frequency of each 'word'
        def tf_idf_calc(tf_all, df_all, wordsInDoc):
            result = []
            for key, value in tf_all.items():
```

```
        doc = key[0]
        term = key[1]
        df = df_all[term]
        if (df>0):
            # We use the formula with the division by the number of documents
            # where the word is present
            # the other one : float(value)*np.log(1 + N/df)
            tf_idf_score = float(value)/wordsInDoc[doc]*np.log(1 + N/df)
        result.append({"doc":doc, "word":term, "score":tf_idf_score})
    return result

# First, we retrieve the data by keeping the only the filename as a key for
# each line, and count the number of documents at the same time
all_text = sc.emptyRDD()

# We have here 5 documents:
N = 5

# We assume the current files to be stored in the input folder.
for i in range(5):
    filename = "hdfs:///user/hadoop/wc/input/" + i + ".txt"
    # We use only the name of the document without extension. We make sure to
    # do the split only once per document
    shortFilename = filename.split("/")[-1].split(".")[0]
    all_text += sc.textFile(filename).keyBy(lambda x: shortFilename)


# Then we tokenize using the function defined before
u = all_text.map(lambda x : (x[0],tokenize(x[1])))


# And we generate a couple for every "document", "word"
u = u.reduceByKey(lambda x,y : x+y).flatMapValues(lambda x: x)

# We will have to use u several times after that:
u.persist()


# Calculation of TF
tf_all = u.countByValue()
# Calculation of the appearance of each word in how many documents (this is
# not necessary if we use only the first formula).
wordsInDoc = u.countByKey()

# Calculation of DF (we only need to invert to "word", "document" and count by
# key
df_all = u.distinct().map(lambda (title,word):(word,title)).countByKey()
```

```
# Calculation of TF-IDF score
tf_idf = tf_idf_calc(tf_all, df_all, wordsInDoc)

# Let's print the first elements of the matrix.
print("The five first elements of our matrix:\n\n\n\n")
print(tf_idf[0:5])
print("\n\n\n\n")
```

5 Time complexity comparison

Here are the plot of the Map Reduce and Spark jobs (for a size i, the number of words in the documents is $50 * 2^i$

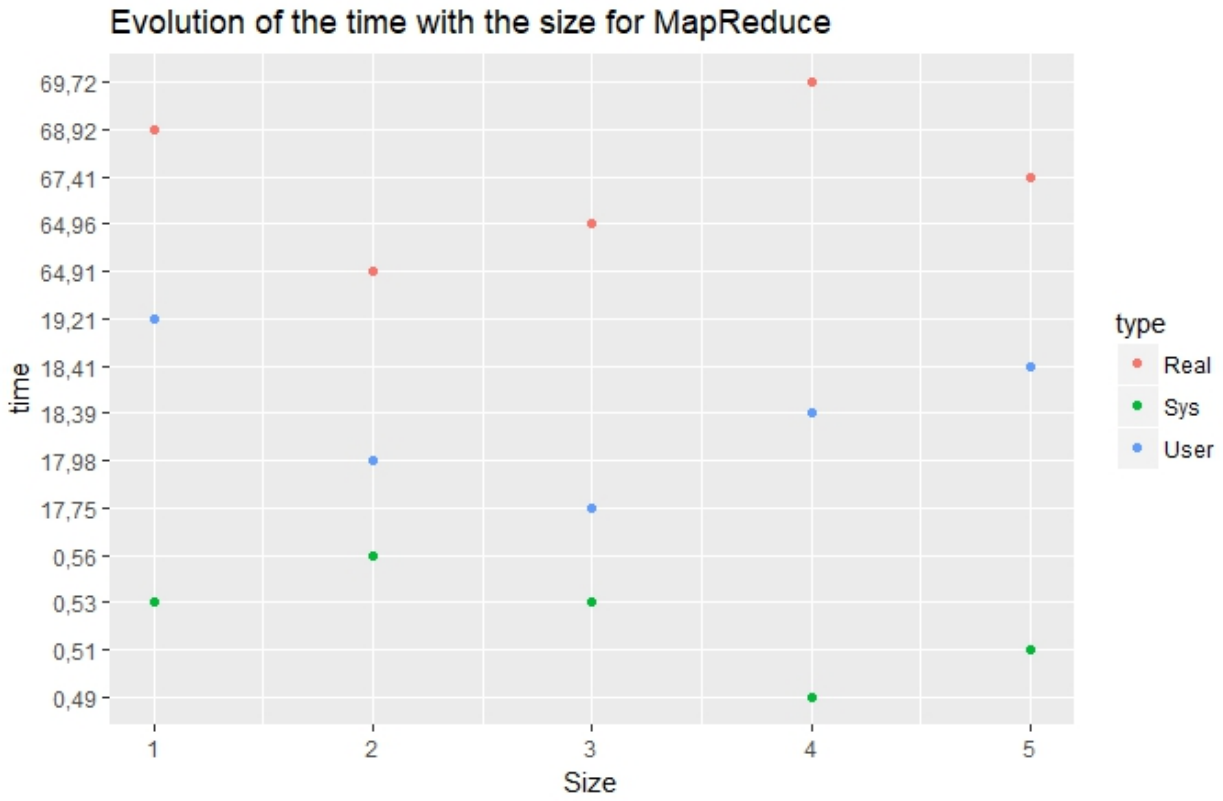

Figure 1: Evolution of the execution of time of the MapReduce job in function of the size of the input.

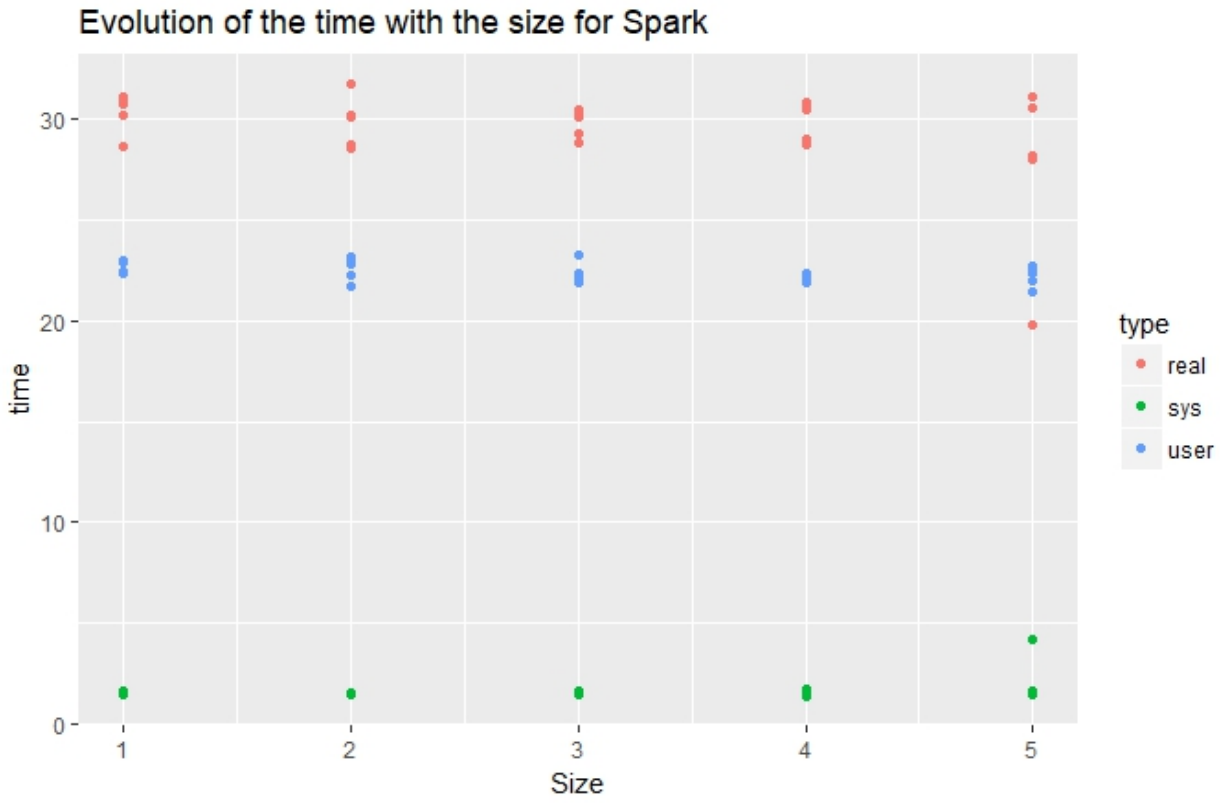

Figure 2: Evolution of the execution of time of the Spark job in function of the size of the input.

We can see that it is difficult to make a statement about the time evolution in function of the size; the number of node used is adapted to the size of the files given as input.

We actually tried with way bigger files (around 1000 times the size of the last one, so 1 600 000 words, but the results came out to be even better (they were not launched at the same moment of the day)...