Distributed data analytics (2016-2017)
Information Systems and Machine Learning Lab
University of Hildesheim

Exercise Sheet 6

Mario Rodríguez Ruiz

May 26, 2017

Contents

List of Figures

List of Tables

1 Explain your system

	Home System
Machine	Asus Notebook ROG G60Jx
Operating System	Windows 10 Pro 64-bit
CPU	Intel Core i7 720QM @1.60GHz
Number of cores	4
Number of threads	8
RAM	16GB @665MHz (9-9-9-24)
Programming language version Python	v3.6.1:69c0db5 64 bit
Programming language version Java	v1.8
Hadoop	v2.6.0

Table 1.1: My system

2 Data cleaning and text tokenization

2.1 Cleaning: remove all punctuations and numbers

First to remove all punctuations and numbers, I've done this task in the Mapper.
After reading each line the ***sub*** function is called, imported from ***re*** in which it specifies which characters are the only ones that can be included in each phrase.

```
import re

# input comes from STDIN
for line in sys.stdin:
  # remove anything other than an alphabet letter
  line = re.sub('[^A-Za-z]+', ' ', line)
  [...]
```

To indicate which characters are allowed, the expression $'[`A - Za - z]+'$ is used. This indicates that everything that is not a letter of the alphabet is excluded.

```
line = re.sub('[^A-Za-z]+', ' ', line)
```

The following is to indicate by which character to replace each letter not allowed, in this case will be done with a space (' ') and finally the complete phrase to be examined (**line**).

2.2 Stopping: removing meaningless words

The most important thing at this point is knowing how to differentiate when it comes to one input and when another. That is, when it is processing the entry containing the Common English Words and when it is processing the file containing the data it want to count.

```
# split the line into words
words = line.split()
```

```
# storage line for english words
en_words = line

# if line size is high than 550 it is because
# the text file is the common english words
if(len(line)>550):
  print ('{}\t{}'.format(en_words, 0))
# it is the text file to evaluate
else:
  # each word found in the data file
  for word in words:
    if len(word)>1:
      # write the results to STDOUT
      print ('{}\t{}'.format(word, 1))
```

To differentiate this, what I have done has been to distinguish them through their size. As you know the minimum size (as columns) that has the Common Words file, it could be characterize the IF-switch for it.

As it could be can see in the code fragment above, if the size of Line is greater than 550 (columns) is because it is the file of Common Words. With this it can be already differentiated between inputs from the Mapper and then work comfortably on the Reducer.

2.2.1 mapper.py

```
import sys
import re

# input comes from STDIN
for line in sys.stdin:
  # remove anything other than an alphabet letter
  line = re.sub('[^A-Za-z]+', ' ', line)
  # converts the text to lowercase
  line = line.lower()
  # split the line into words
  words = line.split()
  # storage line for english words
  en_words = line

  # if line size is high than 550 it is because
  # the text file is the common english words
  if(len(line)>550):
    print ('{}\t{}'.format(en_words, 0))
  # it is the text file to evaluate
  else:
    # each word found in the data file
    for word in words:
```

```
      if len(word)>1:
        # write the results to STDOUT
        print ('{}\t{}'.format(word, 1))
```

In the part of Reducer, there are two details to take into account: The first one is to know when the input from the Mapper is Common English Words or when it comes to the text file to be examined.
The second one is the way each word is counted.

```
# input comes from STDIN
for line in sys.stdin:
  line = line.strip()
  word, count = line.split('\t', 1)
  # Converts input to numeric value
  count = int(count)
  # If count is zero it means that the input is the Common English
  if count == 0:
    [...]
  # if count is one means that the entry is now the word bank
  elif count == 1:
    [...]
```

In order to solve the first question, a simple IF-switch has been used. Once the input from the Mapper in which both the text and a counter (or differentiator) have been stored, these results are analyzed.

If the second input is equal to zero, this variable works as a differentiator. That is, the entry is about the common words english.
On the contrary, if it is a number one it means that the file is the data file and that it will work as a counter.

```
# If count is zero it means that the input is the Common English
elif count == 1:
  [...]
  elif word not in common_english_words:
    if current_word:
      # write result to STDOUT
      print ('{}\t{}'.format(current_word, current_count))
    current_count = count
    current_word = word
```

Once the Common Words have been stored in the Reducer, the check can be performed.

As seen in the previous code fragment, for each word it is checked if it is in the array of Common Words English. Only in the case of not being in these would store a new word in the Reducer.

2.2.2 reduce.py

```
import sys

current_word = None
current_count = 0
word = None
common_english_words = []

# input comes from STDIN
for line in sys.stdin:
  line = line.strip()
  word, count = line.split('\t', 1)
  # Converts input to numeric value
  count = int(count)
  # If count is zero it means that the input is the Common English
  if count == 0:
    common_english_words = word.split(' ')
  # if count is one means that the entry is now the word bank
  elif count == 1:
    # it works because Hadoop sorts map output
    # by key (word) before it's passed to the reducer
    if current_word == word:
      current_count += count
    # If the word isn't a Common English
    elif word not in common_english_words:
      if current_word:
        # write result to STDOUT
        print ('{}\t{}'.format(current_word, current_count))
      current_count = count
      current_word = word

# write the last word if needed
if current_word == word:
  print ('{}\t{}'.format(current_word, current_count))
```

3 Calculate TFIDF scores of words/tokens

3.1 Mapper step-by-step

The first part of the **mapper** is the same as in the previous exercise. Where it starts to differentiate is in the part where the name of the input file is obtained.

This name is necessary to be able to classify the punctuation of the words according to the documents which contains them and, in addition, have counted the documents that have been processed so that can then apply the mathematical formulas correctly.

```
# Find the name of the input file
try:
  input_file = os.environ['mapreduce_map_input_file']
except KeyError:
  input_file = os.environ['map_input_file']
```

The rest of the code is similar to the one presented in the previous exercise with the exception of a single detail: now the **input processed file name** is also passed in the output in order to work with it in the reducer.

```
# write the results to STDOUT
print ('{}\t{}\t{}'.format(word, 1, input_file))
```

It should be remembered that **one or a zero** was also write in the output (in addition to each word) depending on the file: a zero when it was the CEW file and a one when it was a normal file to process.

3.1.1 mapper.py

```
import sys
import re
import os

# input comes from STDIN
for line in sys.stdin:
  # remove anything other than an alphabet letter
  line = re.sub('[^A-Za-z]+', ' ', line)
  # converts the text to lowercase
  line = line.lower()
  # split the line into words
  words = line.split()
  # storage line for english words
  en_words = line

  # Find the name of the input file
  try:
    input_file = os.environ['mapreduce_map_input_file']
  except KeyError:
    input_file = os.environ['map_input_file']

  # if line size is high than 550 it is because
  # the text file is the common english words
  if(len(line)>550):
```

```
    # write the results to STDOUT
    print ('{}\t{}\t{}'.format(en_words, 0, input_file))
  else:
    # For each word of the line
    for word in words:
      if len(word)>1:
        # write the results to STDOUT
        print ('{}\t{}\t{}'.format(word, 1, input_file))
```

3.2 Reducer step-by-step

To perform this task, it has been convenient to use some shared variables to be able to collect all the necessary data.

```
# To save the CEW list
common_english_words = []
# To save the contents of each of the documents
docs = {}
# To save the input file names and use them as indexes
inputs = []
# number of inputs documents (without accounting for the CEW)
n_doc = 0
```

The fundamental first is "**common_english_words**", in which will be stored the words that doesn't want to count; the second "**docs**" to store the contents of each of the documents; the third "**inputs**" to store each of the names of the input files to use as indexes of the lists; and finally "**n_doc**" that counts the number of documents analyzed.

When adding a new input in the reducer (input_) be has had to specify in its corresponding line. Changes to the previous version (WordCount) when processing the entry are as follows: The first thing to check is if the file is the CEW (through the variable "count" explained above).

```
wordd, count, input_ = line.split('\t')
# Converts input to numeric value
count = int(count)

# If count is zero it means that the input is the Common English
if count == 0:
  common_english_words = wordd.split(' ')
# If count is one and the name of the input file isn't yet known
elif count == 1 and input_ not in inputs and wordd not in common_english_words:
  # Save the new name in the list of names
  inputs.append(input_)
```

```
  # Initialize new list with index of the new name
  docs[input_] = wordd + ' '
# If count is one and the word isn't a CEW
elif count == 1 and wordd not in common_english_words:
  # Add new word to the end of the list identified through the index
  docs[input_] += wordd + ' '
```

If it is not the file, it is checked by its own name if it has already been analyzed. If it is not already known, a new list is initialized with its name as index and the first word that is processed.
Always keep in mind before adding a new word other than a CEW.

Once all the inputs have been processed, all that is left is to perform the calculations.

```
bloblist = []
# Stores the inputs documents (no CEW) in TextBlob format
for i in inputs:
  i = tb(docs[i])
  bloblist.append(i)
```

To do this, the lists that contain the words of each document will be transformed into **TextBlod format** to perform operations in a more manageable way through this library.

Finally, the function tfidf is called, in which operations are performed for each word based on all the documents that have been processed in the input (except the CEW).

```
# For each word of each document it calculates the TFIDF score
for i, blob in enumerate(bloblist):
  print("\nTop words in document \"{}\"".format(inputs[i]))
  scores = {word: tfidf(word, blob, bloblist) for word in blob.words}
  # sort from highest to lowest list by tfidf scores
  scores = sorted(scores.items(), key=lambda x: x[1], reverse=True)
  print("\t--------------")
  print("\tWord\tTF-IDF")
  print("\t--------------")
  for word, score in scores[:3]:
  print("\t{}\t{}".format(word, round(score, 3)))
```

Term frequency which is the number of times a word appears in a document:

```
def tf(word, blob):
  return blob.words.count(word) / len(blob.words)
```

Number of documents containing a specific word

```
def n_containing(word, bloblist):
  return sum(1 for blob in bloblist if word in blob.words)
```

Inverse document frequency which measures how common a word is among all processed documents:

```
def idf(word, bloblist):
  return math.log(len(bloblist) / (1 + n_containing(word, bloblist)))
```

Computes the TF-IDF score:

```
def tfidf(word, blob, bloblist):
  return tf(word, blob) * idf(word, bloblist)
```

3.2.1 reduce.py

```
import sys
import math
from textblob import TextBlob as tb

# To save the CEW list
common_english_words = []
# To save the contents of each of the documents
docs = {}
# To save the input file names and use them as indexes
inputs = []
# number of inputs documents (without accounting for the CEW)
n_doc = 0

# term frequency which is the number of times a word appears in a document blob
def tf(word, blob):
return blob.words.count(word) / len(blob.words)

# number of documents containing word
def n_containing(word, bloblist):
return sum(1 for blob in bloblist if word in blob.words)

# inverse document frequency which measures how common a word is among all documents in bloblist
def idf(word, bloblist):
return math.log(len(bloblist) / (1 + n_containing(word, bloblist)))

# computes the TF-IDF score
```

```
def tfidf(word, blob, bloblist):
return tf(word, blob) * idf(word, bloblist)

# input comes from STDIN
for line in sys.stdin:
  line = line.strip()
  wordd, count, input_ = line.split('\t')
  # Converts input to numeric value
  count = int(count)

  # If count is zero it means that the input is the Common English
  if count == 0:
    common_english_words = wordd.split(' ')
  # If count is one and the name of the input file isn't yet known
    elif count == 1 and input_ not in inputs and wordd not in common_english_words:
    # Save the new name in the list of names
    inputs.append(input_)
    # Initialize new list with index of the new name
    docs[input_] = wordd + ' '
  # If count is one and the word isn't a CEW
  elif count == 1 and wordd not in common_english_words:
    # Add new word to the end of the list identified through the index
    docs[input_] += wordd + ' '

bloblist = []
# Stores the inputs documents (no CEW) in TextBlob format
for i in inputs:
i = tb(docs[i])
bloblist.append(i)

# For each word of each document it calculates the TFIDF score
for i, blob in enumerate(bloblist):
  print("\nTop words in document \"{}\"".format(inputs[i]))
  scores = {word: tfidf(word, blob, bloblist) for word in blob.words}
  # sort from highest to lowest list by tfidf scores
  scores = sorted(scores.items(), key=lambda x: x[1], reverse=True)
  print("\t--------------")
  print("\tWord\tTF-IDF")
  print("\t--------------")
  for word, score in scores[:3]:
  print("\t{}\t{}".format(word, round(score, 3)))
```

3.3 Execution test

Image 3.1 shows the execution command that has been carried out through Eclipse.

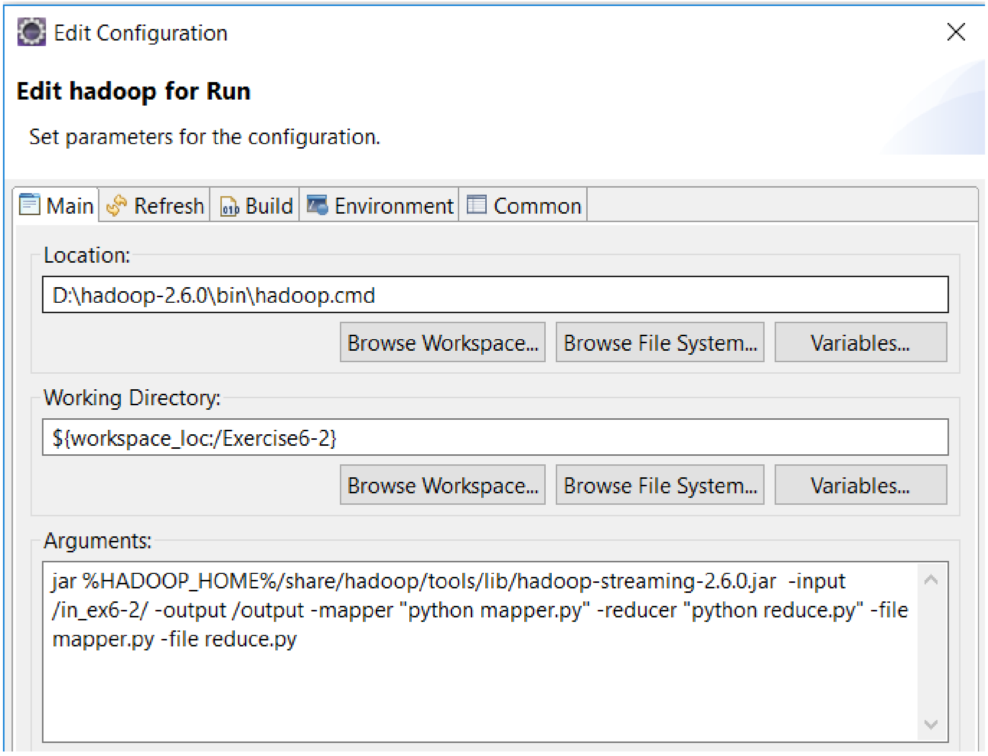

Figure 3.1: Eclipse run configurations

Image 3.2 shows the input files to be processed.

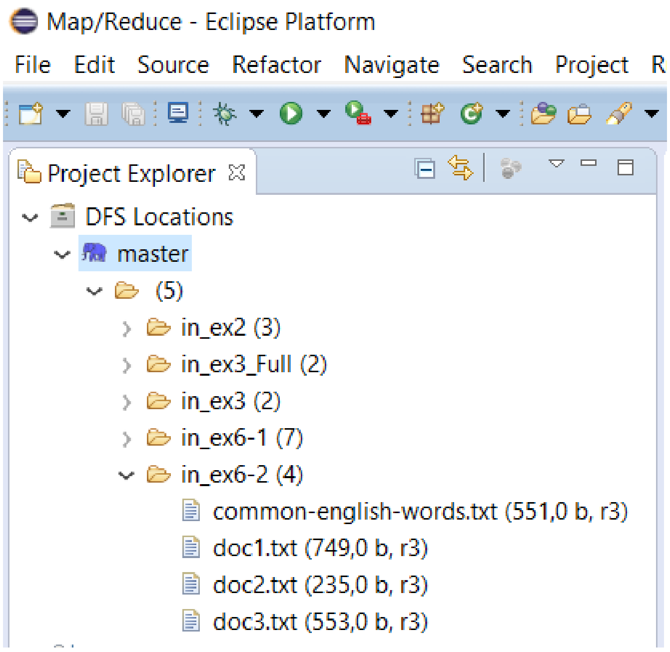

Figure 3.2: Hadoop input files

Image 3.3 shows the output files with the results of the TFIDF calculation.

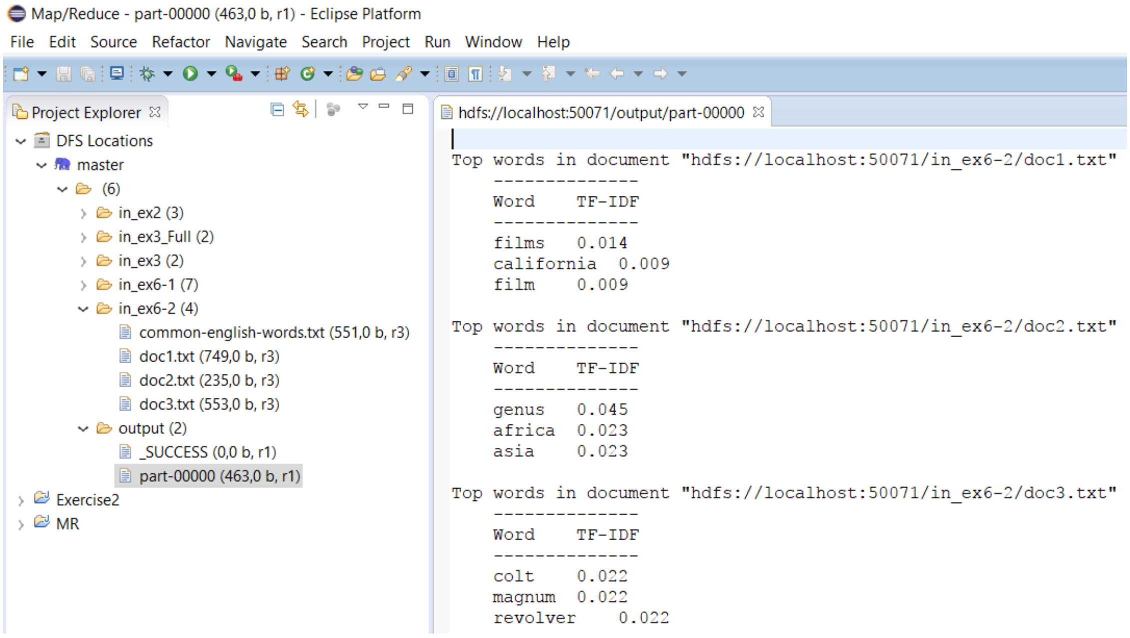

Figure 3.3: Results of the TFIDF calculation

Image 3.4 shows the results of the TFIDF calculation of tutorial http://stevenloria.com/finding-important-words-in-a-document-using-tf-idf/

```
Top words in document 1
    Word: films, TF-IDF: 0.00997
    Word: film, TF-IDF: 0.00665
    Word: California, TF-IDF: 0.00665
Top words in document 2
    Word: genus, TF-IDF: 0.02192
    Word: among, TF-IDF: 0.01096
    Word: Currently, TF-IDF: 0.01096
Top words in document 3
    Word: Magnum, TF-IDF: 0.01382
    Word: revolver, TF-IDF: 0.01382
    Word: Colt, TF-IDF: 0.01382
```

Figure 3.4: Results of the TFIDF calculation tutorial

The results are not exactly the same because this tutorial takes into account all the characters, being also the words of the **common english word**.