Word Counting

With this lab we practice basic operations on lists, string and dictionaries, as well as fundamental programming concepts such as loops, if statements and functions. The lab is designed to be simple but despite its simplicity it also demonstrates a technique which is in the foundations of many algorithms which allow machines to understand human language.

Note that machines do not understand language the way we do. Instead they collect statistics which allow them to make inferences which are right most of the time, even when the machine lacks real understanding. The most basic statistics is plain word counting, i.e. to collect which words appear in a document and how often. There are four parts in this lab:

- Tokenization - how to split a running text into a sequence of words (tokens);
- Counting - aggregate statistics for the frequencies of the different words;
- Printing - print the final results;
- Completed Program - assembling a complete program by using the pieces above.

Before you go further with the lab, download the file lab1.zip

Download lab1.zip

which contains the files that you will need.

Tokenization

In most languages, it is easy to separate words by just looking for the spaces in between. There are exceptions of course but as longs as we stick with English (or Swedish), this is mostly true. Try this two lines in the Python shell:

```
>>> text = 'Simple is better than complex'
>>> text.split()
['Simple', 'is', 'better', 'than', 'complex']
```

It looks as if Python already knows how to separate into words. The method `split` simply splits a string into a list of words if they are separated by one or more spaces. Unfortunately this solution wouldn't go very far:

```
>>> text = 'In the face of ambiguity, refuse the temptation to
guess.'
>>> text.split()
['In', 'the', 'face', 'of', 'ambiguity,', 'refuse', 'the',
'temptation', 'to', 'guess.']
```

Do you see the problem? Punctuation symbols such as comma and dot are usually glued to the last token without space, despite that we don't consider them part of the word. The method `split` does not know anything about punctuation, and that is the problem. We can do something better, but lets first clarify what we count as a word:

- a word cannot contain white spaces. White spaces are used to detect word boundaries but are otherwise ignored;
- a word is any sequence of one or more letters, i.e. symbols from the alphabet;
- a word is any sequence of digits, e.g. the symbols '0' … '9';
- any other symbol not mentioned above is counted as a single word containing only the symbol alone.

Your first task is to define a function called `tokenize` which should take a complete document as a list of text lines and produce a list of tokens. For example:

```
>>> document = ['"They had 16 rolls of duct tape, 2 bags of
clothes pins,',
...             '130 hampsters from the cancer labs down the hall,
and',
...             'at least 500 pounds of grape jello and unknown
amounts of chopped liver"',
...             'said the source on a recent Geraldo interview.']
>>> tokenize(document)
['"', 'they', 'had', '16', 'rolls', 'of', 'duct', 'tape', ',',
'2', 'bags', 'of', 'clothes', 'pins', ',',
 '130', 'hampsters', 'from', 'the', 'cancer', 'labs', 'down',
'the', 'hall', ',', 'and', 'at', 'least',
 '500', 'pounds', 'of', 'grape', 'jello', 'and', 'unknown',
'amounts', 'of', 'chopped', 'liver', '"',
 'said', 'the', 'source', 'on', 'a', 'recent', 'geraldo',
'interview', '.']
```

The function should be placed in a module called `wordfreq`, i.e. in a file called `wordfreq.py`. To start with, you can
put a dummy tokenize function which does nothing, e.g.:

```
def tokenize(lines):
    pass
```

Here `pass` is a keyword which tells Python that there is nothing to be done. Once you do that, you can also use our test program called test.py. You should place it in the same folder where you saved `wordfreq.py`, and run it as follows:

```
$ python3 test.py
Condition failed:
   tokenize([]) == []
tokenize returned/printed:
None
Condition failed:
   tokenize(['']) == []
tokenize returned/printed:
None
Condition failed:
   tokenize(['   ']) == []
tokenize returned/printed:
None
Condition failed:
   tokenize(['This is a simple sentence']) == ['this', 'is', 'a',
'simple', 'sentence']
tokenize returned/printed:
None
Condition failed:
   tokenize(['I told you!']) == ['i', 'told', 'you', '!']
tokenize returned/printed:
None
Condition failed:
   tokenize(['The 10 little chicks']) == ['the', '10', 'little',
'chicks']
tokenize returned/printed:
None
Condition failed:
   tokenize(['15th anniversary']) == ['15', 'th', 'anniversary']
tokenize returned/printed:
None
Condition failed:
   tokenize(['He is in the room, she said.']) == ['he', 'is',
'in', 'the', 'room', ',', 'she', 'said', '.']
tokenize returned/printed:
None
countWords is not implemented yet!
printTopMost is not implemented yet!
0 out of 8 passed.
```

The above is the output that I got when I run that test with the dummy implementation. For every test that failed you see what went wrong and at the end you also see statistics about how many tests passed.

Now to go further you should remove the keyword `pass` and start writing your code there.

Processing Lines

The input to our function is a list of text lines. The first step to do is to go through all lines, split each of them into words, accumulate the recognized words and finally return the complete list. This can look as follows:

```
words = []
for line in lines:
  pass
return words
```

Here the actual word splitting should be in place of the `pass` keyword, and each new word should be added to the list 'words'. Note that when we write code, unlike with prose, we often don't write everything sequentially. Instead we often write the general outline of the program, as I did above, and later we fill in the gaps with pieces of code which are hopefully simpler than the whole program. We recommend you that you do the same. The keyword `pass` can be useful for that.

So far we found how to process each line separately, but to be able to split words, we need to look at individual characters. That is just as easy. Try to insert the following code in place of the `pass` keyword:

```
start = 0
while start < len(line):
  print(line[start])
  start = start+1
```

After that import the `wordfreq` module in the Python shell:

```
$ python3
>>> import wordfreq
>>> wordfreq.tokenize(['apple','pie'])
a
p
p
l
e
p
i
e
[]
```

As you can see the new version of tokenize just prints all the characters in the input list. What the new code above does is to just use a `while` loop which goes through all characters. For that purpose we use the `start` variable which keeps track of the current position. At the end of the loop it is incremented to go to the next character, and the loop keeps going until the end of the current line is reached. In the middle of the loop, the `print` function prints the current character.

Note that the Python shell also prints `[]` at the end. This is the value of variable `words`, which we returned from the function. We still don't add new words to it, so Python just prints an empty list.

Recognizing Words

The next thing that you need to do is to recognize what kind of word you have, according to the specification that we gave in the beginning of the assignment. For that it would be useful to know about a few methods that Python provides:

What is white space? The type string in Python has a method called `isspace` which returns True only if the string contains only white spaces:

```
>>> ' '.isspace()
True
>>> 'p'.isspace()
False
```

-

What is a letter? Similarly you can check whether something is a letter or not:

```
>>> 'z'.isalpha()
True
>>> 'F'.isalpha()
True
>>> 'ä'.isalpha()
True
>>> '-'.isalpha()
False
```

Note that this is not limited to the Latin alphabet:

```
>>> 'Ω'.isalpha()
True
>>> 'я'.isalpha()
True
>>> 'ॐ'.isalpha()
True
```

-

Finally digits are also easy:

```
>>> '0'.isdigit()
```

```
True
>>> '9'.isdigit()
True
>>> '٢'.isdigit()
True
>>> 'A'.isdigit()
False
```

-

As we said white spaces in the input should be ignored, but the current draft of our `tokenize` function doesn’t do that yet. Fix that yourself by inserting one more `while` loop, just before the `print` function. The loop should increment the `start` variable as long as the current character is a space. If that is successful, the following should happen:

```
>>> wordfreq.tokenize(['    sweet  apple  tart'])
s
w
e
e
t
a
p
p
l
e
t
a
r
t
[]
```

The function should print only characters that are not spaces.

The next step is to recognize what kind of word you have. As we said in the beginning, there are three kinds of words:

- those starting with a letter;
- those starting with a digit;
- and those which contain only one character which is neither letter nor digit.

Replace the current `print` function with a sequence of `if` statements which check what kind of character you have. Inside each if statement print the current character and its type. The output should look like:

```
>>> wordfreq.tokenize(['1 pie.'])
1 is a digit
```

```
p is a letter
i is a letter
e is a letter
. is a symbol
[]
```

Note that if a character is a letter, it cannot be digit or symbol and vice versa. Use `elif` to avoid unnecessary checks in the program.

We are almost done now. We know what kind of word we have encountered, we just have to extract the word itself. When the word starts with a letter or a digit, we need to take all consecutive characters of the same kind as part of the word.
Replace the print statements which print letters and digits with while loops which iterate as long as there are more letters/digits respectively.

Note that in order to extract the current word, you need to know its starting and ending position. Let the variable `start`, that we already have, store the starting position. For the new loops you can use a new variable called `end`. Before the new while loops, set that variable to be equal to `start` and then in the loops themselves, increment `end`. At the end `start` will point to the beginning and `end` to the end of the word.

To extract the word itself, use Python’s slicing syntax, e.g. `line[start:end]` would give you the word. After that just append it to the list `words`.

At that point you have successfully extracted the word and saved it in the right place. Variable `start` points to the beginning and `end` points to the end. In order to allow the next word to be recognized as well. Set `start` to be equal to `end`. In this way the program will search for the next word after the end of the current one.

Finally you should take care of the case when the current character is neither a letter nor a digit. This is much simpler since then the character counts as a word by itself. You should be able to figure out, how to handle that on your own.

If everything went fine you should get an output like this:

```
>>> wordfreq.tokenize(['10 sweet apple tarts.'])
['10','sweet','apple','tarts','.']
```

There is one final thing to do before the function is complete. When we count words, we usually want to ignore capitalization, e.g. ‘Sweet’ and ‘sweet’ are really the same words. Capital letters can be converted to small letters by using the method `lower`, e.g. `'Sweet'.lower() == 'sweet'`. Use the method to lower case all words before they are added to the list. After that test it:

```
>>> wordfreq.tokenize(['10 Sweet Apple Tarts.'])
```

```
['10','sweet','apple','tarts','.']
```

When you have gone so far, use our testing script to make sure that there no corner cases that you missed out:

```
$ python3 test.py
countWords is not implemented yet!
printTopMost is not implemented yet!
8 out of 8 passed.
```

Don’t worry about the messages about `countWords` and `printTopMost`. They will disappear after you solve the next two parts of the lab.

Counting

The second part of the lab is to implement a function which takes a list of words and counts how often each word appears. In addition, the function takes a list of stop words. These are words which are not interesting and should be ignored while counting. For example, you should get that working:

```
>>> wordfreq.countWords(['it','is','a','book'], ['a','is','it'])
{'book': 1}
```

Here the result is a dictionary where the keys are words and the values are counts. In the example ‘a’, ‘is’ and ‘it’ are stop words and are whence ignored. ‘book’ is the only other word and it appears exactly once.

The implementation is simple. Start with the skeleton:

```
def countWords(words, stopWords):
  pass
```

and replace the keyword `pass` with your code which should do roughly the following:

- Initialize a variable with an empty dictionary.
- Start a loop that goes through all words in the list. If the current word appears in the list `stopWords` then ignore it.
- The first time when you encounter a new word, add it with count 1 to the dictionary. The next time just increment the count with one.
- return the dictionary from the function after you have counted all words.

Note that you can find whether you have already seen the word by using the operator `in` in Python. If the variable `frequencies` refers to the dictionary, then the expression `word not in frequencies` will evaluate to `True` only if this is the first time when you have encountered the word.

When you are done, run the test program, to make sure that everything works well.

Printing

The last missing piece is to be able to print the collected statistics. We already have a way to construct a dictionary where the keys are the words and the values are the counts. We just have to iterate through the entries and print the data. Precisely for that the dictionary type has a method called `items`. You can, for instance, do that:

```
for word,freq in frequencies.items():
  print(word,freq)
```

Here, `items` returns a sequence of key-value pairs which in our case will be the word and its count.

Your task in this part is just slightly more involved. You have to complete the function:

```
def printTopMost(frequencies,n):
  pass
```

which takes a dictionary of frequencies and prints the top most `n` words. For example:

```
>>> printTopMost({'text': 9, 'word': 30, 'fiction': 6, 'count': 
11, 'counting': 7, 'novel': 6},3):
word                    30
count                   11
text                     9
```

In the output, each word and its count must be printed on a separated line. The numbers should be aligned to the right, and the words to the left.

In order to get the top most words we need to sort the sequence of words and counts. Python offers the function `sorted` for that purpose. For example:

```
>>> sorted([5,2,1,8])
[1, 2, 5, 8]
```

If you want to sort the data in another order then you can add the optional argument `key`. For example, to sort the list in decreasing order, you can also do that:

```
>>> sorted([5,2,1,8], key=lambda x: -x)
[8, 5, 2, 1]
```

Here the function that we supply as value for `key` is applied to every element in the sequence, and then the sequence is sorted by that value instead of by the elements themselves. In particular negating the numbers in the example has the effect of sorting the sequence in decreasing order.

Similarly if you want to sort tuples of data by using only one of values in the tuple then you can do something like this:

```
>>> sorted([("alpha",1),("gamma",3),("beta",2)], key=lambda x:
x[1])
[("alpha",1),("beta",2),("gamma",3)]
```

To complete the task apply the function `sorted` to the result from the method `items` by using an appropriate `key` function. Since you only need to print the top `n` words, stop the printing as soon as you have printed enough.

In order to get a nicely aligned table of words and counts, use the methods `ljust` and `rjust` for strings. These methods add spaces to the end/beginning of a string until the string reaches a certain length. For example:

```
>>> "a".ljust(5)
'a    '
>>> "a".rjust(5)
'    a'
```

To get exactly the same alignment as in the example for printTopMost, make the column for the words 20 characters wide and the one for the numbers 5 characters wide.

Run the test program again when you are done with this part of the lab.

Completed Program

It is time to piece the different parts into a complete working program.
Start a new module called `topmost` (file topmost.py) and import
the module `wordfreq` from it. We want a program which can be executed
from the command line, like this:

```
$ python3 topmost.py eng_stopwords.txt examples/article1.txt 20
word                   30
words                  21
```

```
count                     11
text                       9
000                        9
counting                   7
fiction                    6
novel                      6
rules                      5
length                     5
used                       4
usually                    4
details                    4
software                   4
sources                    4
processing                 4
segmentation               4
rule                       4
novels                     4
number                     3
```

Here after the name of the program we can write the path to two files.
The first contains a list of stop words and the second contains the
file to be analyzed. The last parameter is the number of words to
be printed. In order to access those parameters
(also called command line arguments) from your program, you need
to import the module `sys`:

```
import sys
```

After that `sys.argv` will give you the list of arguments, where `sys.argv[1]` is the first argument.

You need a way to read a file. Here is how this is done in Python:

```
inp_file = open(sys.argv[1])
....
inp_file.close()
```

What you get from the function `open` is a file object which you can use
in a loop to iterate through the text lines, e.g.:

```
for line in inp_file:
    # do something with the current line
```

Since in `tokenize` we just iterate through the argument lines,
we can just as well pass the file directly to the function.

Note: If you work on Windows, then in order to read the file, you might need to use:

```
inp_file = open(sys.argv[1], encoding="utf-8")
....
inp_file.close()
```

The reason is that unlike all other modern operating systems, Windows is still not fully Unicode compliant.

When you read from the file with stop words, note that each line that you read from a file ends with the special character `'\n'` which indicates that this is the end of the line. Those must be removed by using the method `strip()` for strings.

Finally, every time when you work with files, you should not forget to close the file, when you don't use it any more. This releases resources which other programs on the same computer can use later.

Complete the program by writing a single `main` function which takes the command line arguments as input and uses the functions `tokenize`, `countWords` and `printTopMost` to achieve the goal. Note that to be able to run the program as shown above, after you define the `main` function, you should also call it, i.e. add the line:

```
main()
```

Test the program with the text files included in the examples folder. All the files are actually articles from Wikipedia. Can you guess what each article is about by looking at the output from you program and without reading the text? Since you have written the program yourself, you know that it doesn't do any magic and it doesn't have any intelligence on its own. Yet, it has managed to find the topic of the article. This is an example where very simple techniques can create the illusion of artificial intelligence. You probably realize also that the program has many limitations, more advanced techniques can go a long way forward, but they will be also make the lab way too difficult.

Using the web (Required)

It is tempting to try the program on other texts, but downloading files one by one is tedious and boring. Fortunately it is easy to make the program fetch data directly from the web.
For that you need to use the `urllib` library. First import the module:

```
import urllib.request
```

after that fetching a file and splitting it into lines is as easy as:

```
response = urllib.request.urlopen(sys.argv[2])
lines = response.read().decode("utf8").splitlines()
```

We will not go into the details of the `urllib` library here. If you want to know more, read the documentation

Links to an external site.

.
For now, just use the lines above to get the data which you should then feed to your existing functions. The new program should work with files as before, but it should also
recognize URLs. If the second argument for the program starts with either `http://` or `https://`, then you know that you should fetch the data from the web.
Here are some example free texts:

- http://www.gutenberg.org/files/15/15-0.txt
- Links to an external site.
-
- http://www.gutenberg.org/cache/epub/103/pg103.txt
- Links to an external site.
-
- http://www.gutenberg.org/cache/epub/164/pg164.txt
- Links to an external site.
-
- http://www.gutenberg.org/cache/epub/17192/pg17192.txt
- Links to an external site.
-
- http://www.gutenberg.org/cache/epub/10900/pg10900.txt
- Links to an external site.
-
- http://www.gutenberg.org/files/1998/1998-0.txt
- Links to an external site.
-

You can find more by searching on the home page of Project Gutenberg.
When you find a book, click on the version labeled as “Plain Text UTF-8”. After that copy the URL from the address bar of the browser.

Sometimes people have difficulties accessing the Gutenberg website. If this happens then you can use the backup that I created:

- http://www.grammaticalframework.org/~krasimir/15-0.txt
- Links to an external site.
-
- http://www.grammaticalframework.org/~krasimir/pg103.txt
- Links to an external site.
-
- http://www.grammaticalframework.org/~krasimir/pg164.txt
- Links to an external site.
-
- http://www.grammaticalframework.org/~krasimir/pg17192.txt
- Links to an external site.
-
- http://www.grammaticalframework.org/~krasimir/pg10900.txt
- Links to an external site.
-
- http://www.grammaticalframework.org/~krasimir/1998-0.txt
- Links to an external site.
-

Note that you can just as well use your program on arbitrary web pages (Wikipedia for instance).
However, a web page contains not just text but also formatting instructions and programming code.
If you don't separated those nicely the outcome from your program will be quite messy.
All this can be done but goes beyound the scope of a simple lab.

Upload the files topmost.py and wordfreq.py to Canvas!

Cosine Similarity (Only for DAT505, i.e. Global Systems)

Word counting is the basis of many algorithms in Natural Language Processing (NLP). This last part of the lab is only one example.

Often we want to compare how similar two texts are. For example we may have a collection of texts which we want to split into groups with the similar genre, author, content, etc. Alternatively, we may want to search in the collection by using keywords, and rank the matching documents by how similar they are with the query. In both cases it is useful to have a numerical measure of similarity for two texts.

Using the word counts is a good starting point. Obviously if two documents are using the same words with approximately the same frequency then they are more similar. Of course this is nothing else but just an approximation. For instance we totally ignore the word order, as well as the meaning of the words. There are ways to capture these details as well but we will not go in this direction.

How do we turn the word counts for two documents into a single numeric similarity score? The inspiration comes from geometry. Recall from basic math that a vector in geometry is a directed line which in a coordinate system is described with two numbers:

�→=(�1,�2)

. The length of a vector is computed by using the Pythagorean theorem:

|�→|2=�12+�22

. Two vectors can be multiplied to get their scalar product:

�→⋅�→=�1�1+�2�2

which can be proven to be equal to

|�→||�→|cos�

, where

�

is the angle between the vectors. In particular this also means that

|�→|2=�→⋅�→

. The cosine of the angle between the two vectors:

cos�=�→⋅�→|�→||�→|

measures to what extend they point in the same direction. The smaller the angle (cosine closer to 1), the more similar the directions are. Alternatively when they are orthogonal we get a cosine of 0. For general vectors we can also get negative cosines, but since we will build our vectors from the word counts, all components of the vector will be positive and therefore the cosine is always between zero and one.

The above formulae are for vectors in the plane but are easily generalizable for arbitrary number of directions. When we work with text then we represent each text with a vector in a highly-dimensional space with one dimension for every possible word in the language. The coordinate in each direction is the word count. For example the text "the red apple is red" is represented with the vector

(�apple=1,�green=0,�is=1,�pear=0,�the=1,�red=2,�queen=0,…)

Then cosine of the angle between any two vectors (texts) is still computed by the given formula, except that the scalar product is now defined as:

�→·�→=∑�����

where the index

�

ranges over all possible directions, i.e. all possible words. But how do we iterate over all possible words in a language? We don't really have to. In each of the documents

�→

and

�→

only a finite set of words is used. For every word that doesn't appear in both documents, the product

����

is simply zero. This means that it is enough to iterate only over the words that appear in both documents which is easy.

With the help of the `wordfreq` module and the above formulae write a new program `cosine.py` which given a list of files computes the similarity of the first file with all of the rest. The program should be possible to run like this:

```
$ python3 cosine.py eng_stopwords.txt article2.txt article3.txt
article4.txt
0.39427322052532304
0.2536088121257201
```

Here we compare three Wikipedia articles: article2.txt (about [the electron](#)

[Links to an external site.](#)

), article3.txt (about [the proton](#)

[Links to an external site.](#)

), and article4.txt (about [the Higgs boson](#)

[Links to an external site.](#)

). Our program predicts that the first two documents are more similar, they are after all about the particles that are the building blocks of the atom. There is still quite a similarity with the Higgs boson article. It is still about particle physics! If on the other hand, we compare article2.txt with article1.txt (about word counting

Links to an external site.

), then the similarity is close to zero:

```
$ python3 cosine.py eng_stopwords.txt article2.txt article1.txt
0.05304630224555635
```

Upload the files cosine.py to Canvas as well!