UPC UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona
FIB

- CAIM Grupo 13 de Laboratorio
- Empezamos a las 12:05

MapReduce

In this session:

- You will learn how to use the `map-reduce` python library MRJob
- You will implement a document clustering algorithm and use it to explore a set of documents

MRJob

There are different implementations of the `map-reduce` framework, we have chosen the MRJob[1] library because of its flexibility and the possibility of using different infrastructures to run the processes like HADOOP or SPARK or even cloud services like Amazon *Elastic MapReduce* or Google Cloud Dataproc.

You have the documentation about how to program `map-reduce` jobs with `mrjob` in here

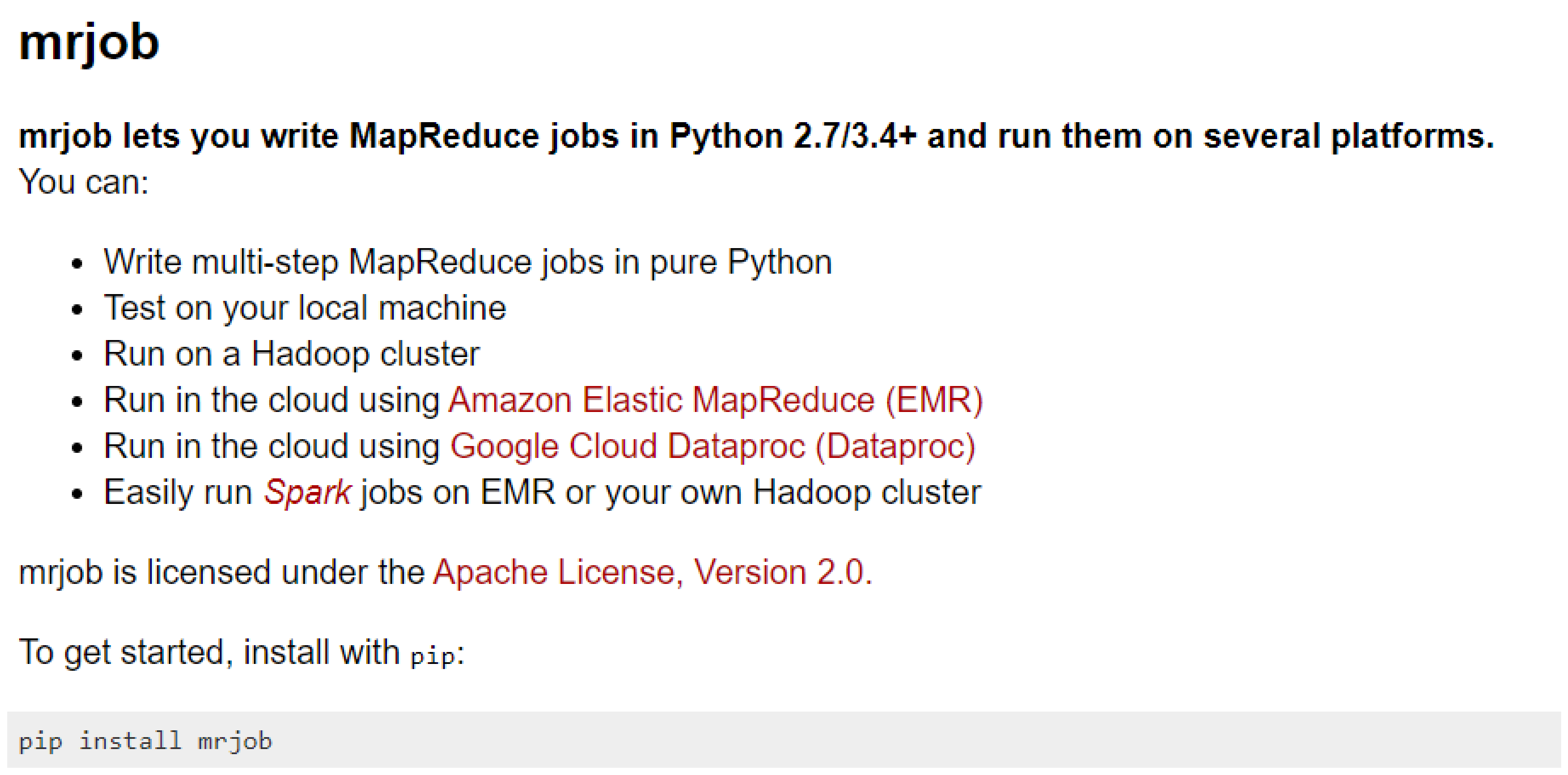
The library is based on two python classes, `mrjob.job.MRJob` that defines a `map-reduce` job and `mrjob.step.MRStep` that allows to define several `map-reduce` steps inside a `MRJob` object.

To define a basic job it is only necessary to create a `MRJob` object and to redefine the `mapper` and `reducer` methods. In the documentation you will see that there are more methods that can be defined to do things before and after the map and reduce steps and also an additional step called `combiner` that allows to combine results from the map step before being sent to the reduce step.

The methods `mapper`, `combiner` and `reducer` all receive a key and a values parameters. The mapper receives in the value parameter an element from the input each time it is called, but the `combiner` and `reducer` receive in the values parameter a python generator that we have to iterate to obtain its values and only can be consumed once.

The rest of the methods do not receive parameters and have to use internal structures filled by the `map-reduce` steps during their execution.

MRJob



mrjob

mrjob lets you write MapReduce jobs in Python 2.7/3.4+ and run them on several platforms. You can:

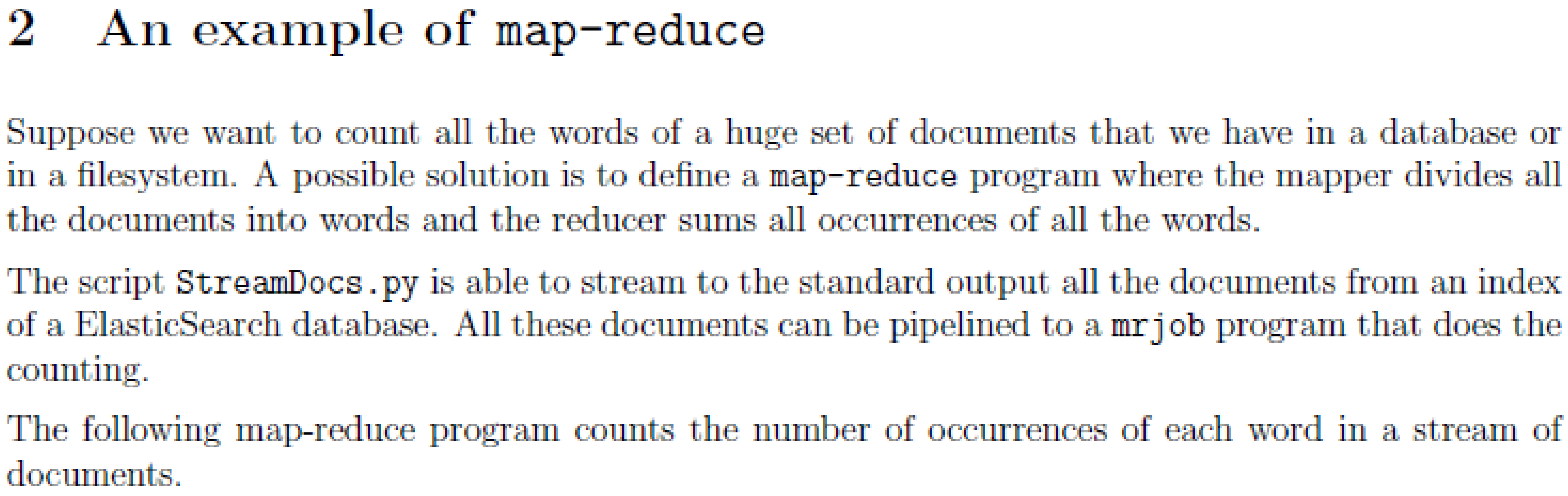
- Write multi-step MapReduce jobs in pure Python
- Test on your local machine
- Run on a Hadoop cluster
- Run in the cloud using Amazon Elastic MapReduce (EMR)
- Run in the cloud using Google Cloud Dataproc (Dataproc)
- Easily run *Spark* jobs on EMR or your own Hadoop cluster

mrjob is licensed under the Apache License, Version 2.0.

To get started, install with `pip`:

```
pip install mrjob
```

MRJob



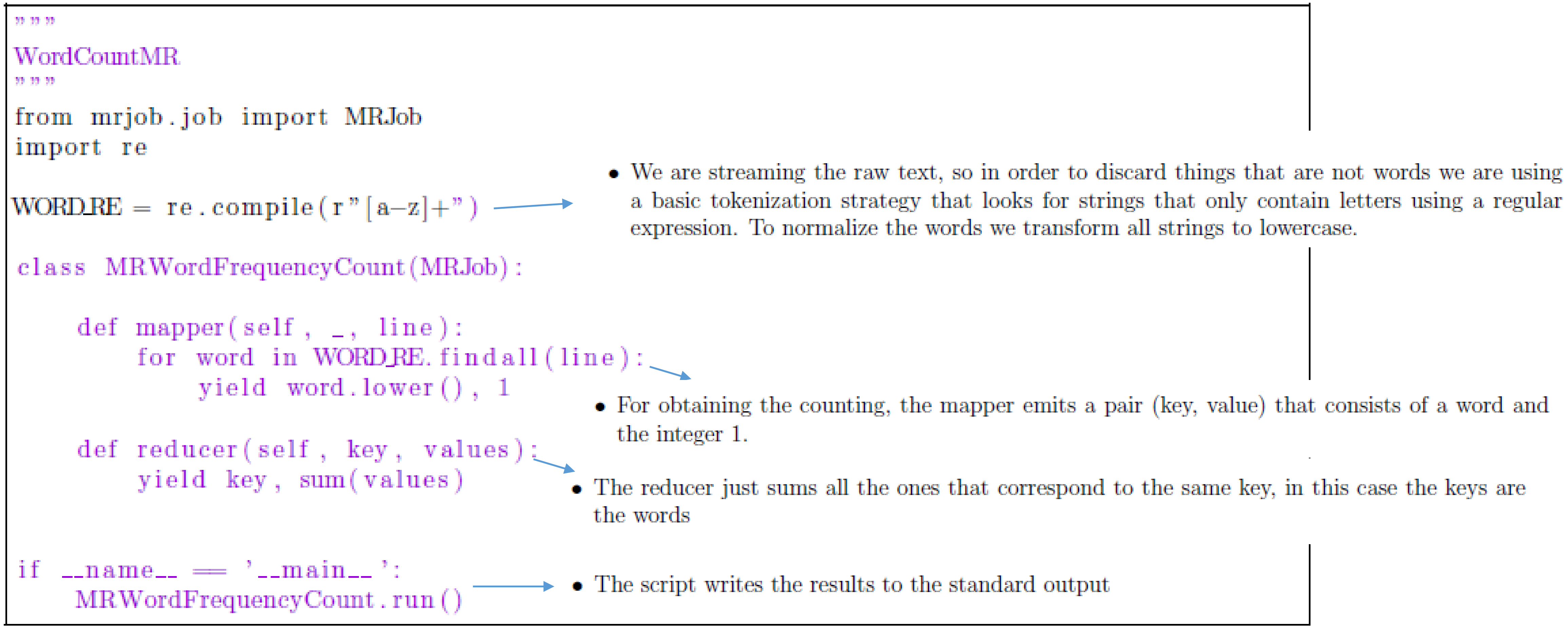
2 An example of `map-reduce`

Suppose we want to count all the words of a huge set of documents that we have in a database or in a filesystem. A possible solution is to define a `map-reduce` program where the mapper divides all the documents into words and the reducer sums all occurrences of all the words.

The script `StreamDocs.py` is able to stream to the standard output all the documents from an index of a ElasticSearch database. All these documents can be pipelined to a `mrjob` program that does the counting.

The following map-reduce program counts the number of occurrences of each word in a stream of documents.

MRJob

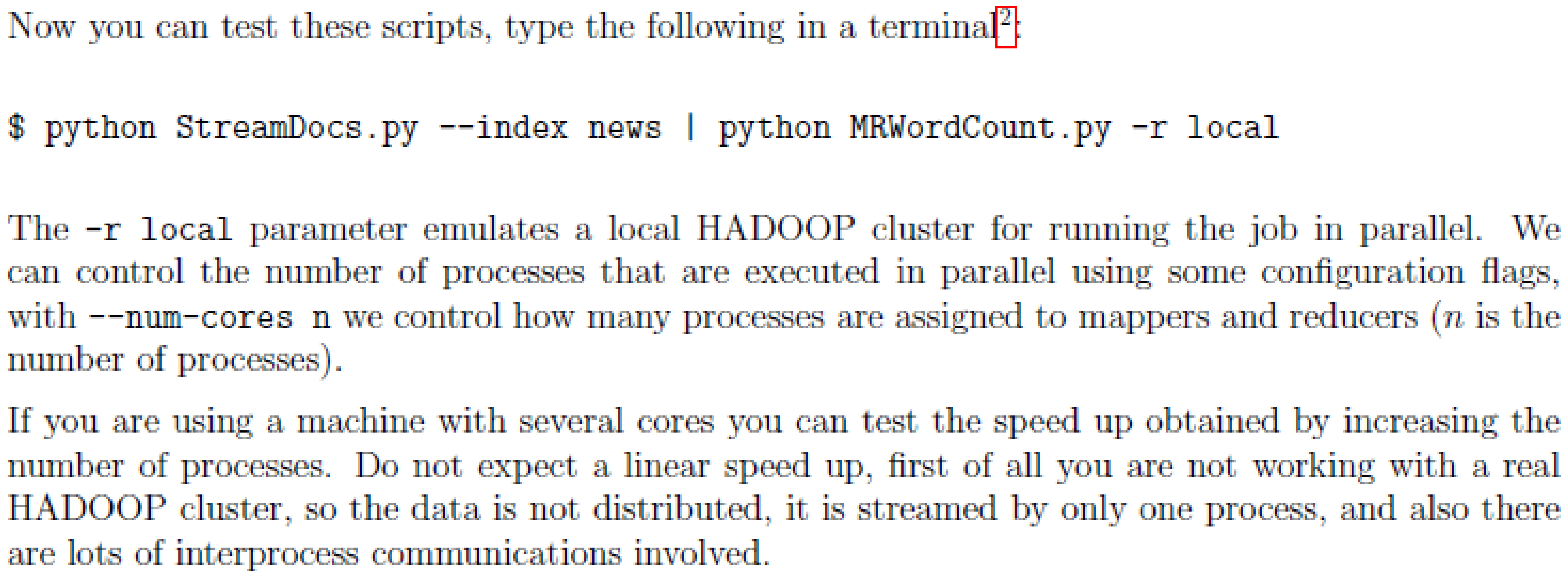


```
"""
WordCountMR
"""
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[a-z]+")

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def reducer(self, key, values):
        yield key, sum(values)


if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

- We are streaming the raw text, so in order to discard things that are not words we are using a basic tokenization strategy that looks for strings that only contain letters using a regular expression. To normalize the words we transform all strings to lowercase.
- For obtaining the counting, the mapper emits a pair (key, value) that consists of a word and the integer 1.
- The reducer just sums all the ones that correspond to the same key, in this case the keys are the words
- The script writes the results to the standard output

Listing 1: MRWordCount.py

MRJob



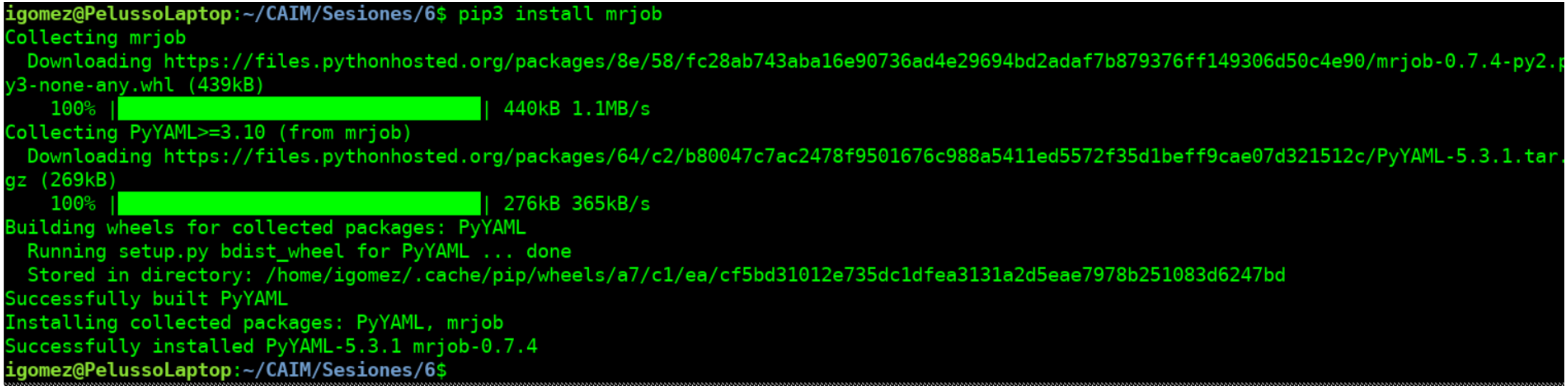
Now you can test these scripts, type the following in a terminal[2]:

```
$ python StreamDocs.py --index news | python MRWordCount.py -r local
```

The `-r local` parameter emulates a local HADOOP cluster for running the job in parallel. We can control the number of processes that are executed in parallel using some configuration flags, with `--num-cores n` we control how many processes are assigned to mappers and reducers (n is the number of processes).

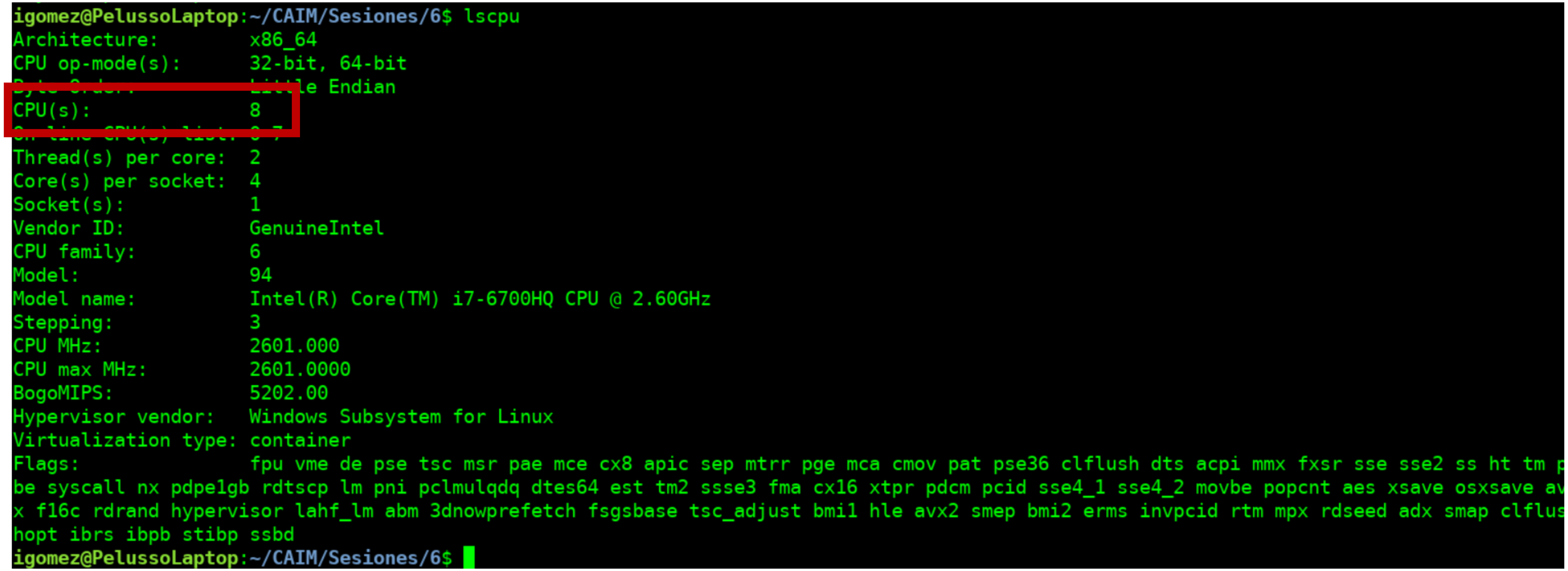
If you are using a machine with several cores you can test the speed up obtained by increasing the number of processes. Do not expect a linear speed up, first of all you are not working with a real HADOOP cluster, so the data is not distributed, it is streamed by only one process, and also there are lots of interprocess communications involved.

MRJob



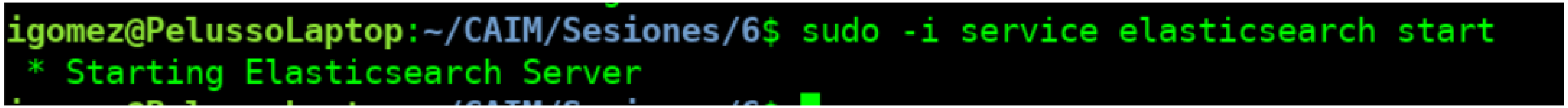
```
igomez@PelussoLaptop:~/CAIM/Sesiones/6$ pip3 install mrjob
Collecting mrjob
  Downloading https://files.pythonhosted.org/packages/8e/58/fc28ab743aba16e90736ad4e29694bd2adaf7b879376ff149306d50c4e90/mrjob-0.7.4-py2.p
y3-none-any.whl (439kB)
    100% |████████████████████████████████| 440kB 1.1MB/s
Collecting PyYAML>=3.10 (from mrjob)
  Downloading https://files.pythonhosted.org/packages/64/c2/b80047c7ac2478f9501676c988a5411ed5572f35d1beff9cae07d321512c/PyYAML-5.3.1.tar.
gz (269kB)
    100% |████████████████████████████████| 276kB 365kB/s
Building wheels for collected packages: PyYAML
  Running setup.py bdist_wheel for PyYAML ... done
  Stored in directory: /home/igomez/.cache/pip/wheels/a7/c1/ea/cf5bd31012e735dc1dfea3131a2d5eae7978b251083d6247bd
Successfully built PyYAML
Installing collected packages: PyYAML, mrjob
Successfully installed PyYAML-5.3.1 mrjob-0.7.4
igomez@PelussoLaptop:~/CAIM/Sesiones/6$
```

MRJob

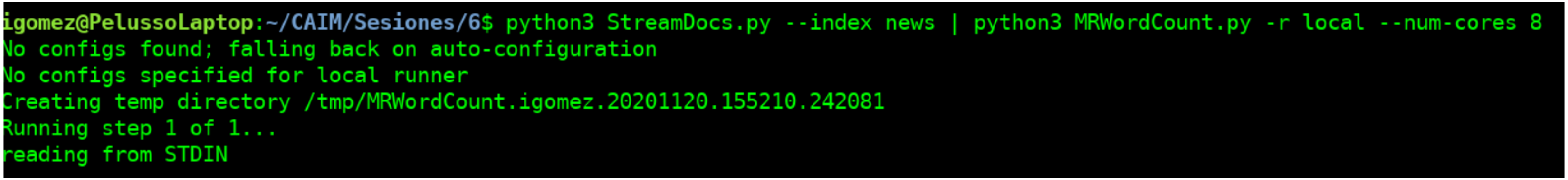


```
igomez@PelussoLaptop:~/CAIM/Sesiones/6$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               94
Model name:          Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Stepping:            3
CPU MHz:             2601.000
CPU max MHz:         2601.0000
BogoMIPS:            5202.00
Hypervisor vendor:   Windows Subsystem for Linux
Virtualization type: container
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm p
be syscall nx pdpe1gb rdtscp lm pni pclmulqdq dtes64 est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 movbe popcnt aes xsave osxsave av
x f16c rdrand hypervisor lahf_lm abm 3dnowprefetch fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdseed adx smap clflus
hopt ibrs ibpb stibp ssbd
igomez@PelussoLaptop:~/CAIM/Sesiones/6$
```

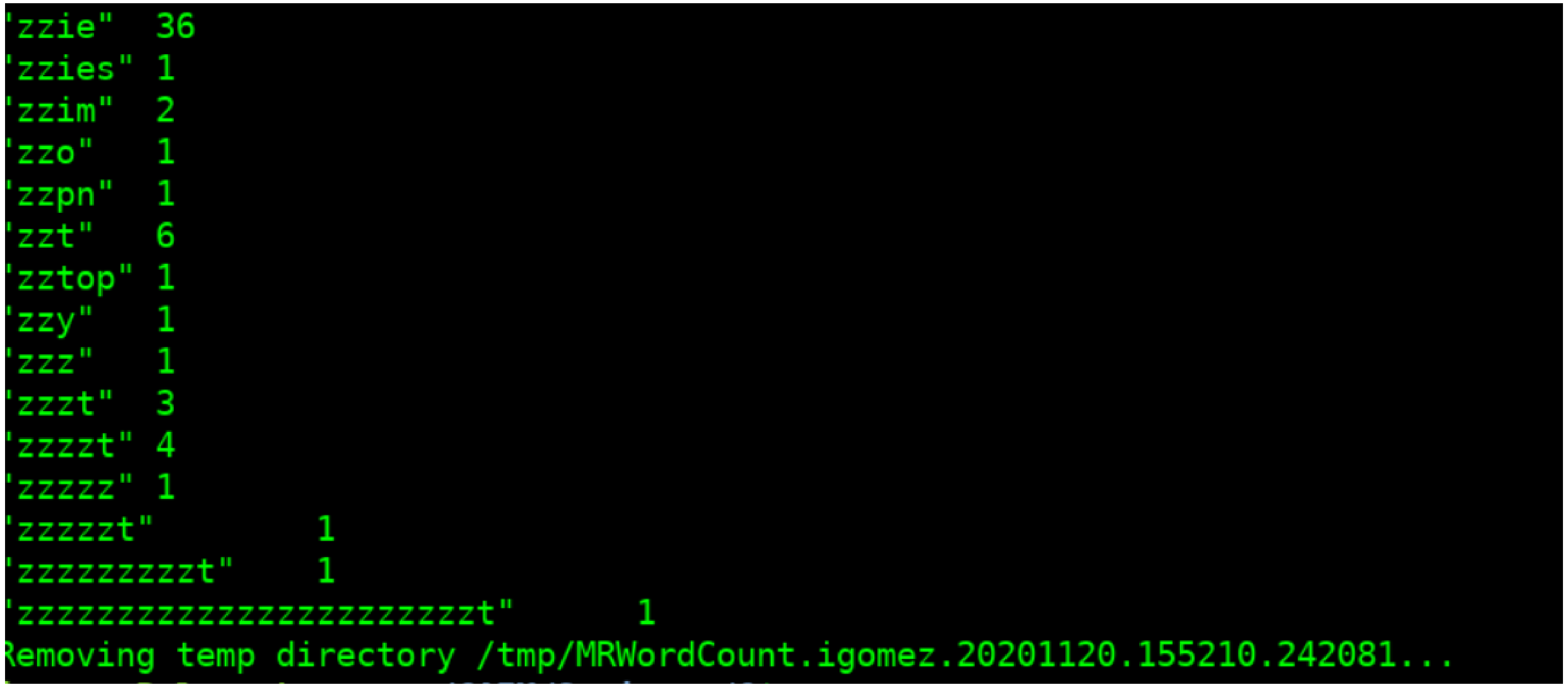
MRJob



```
igomez@PelussoLaptop:~/CAIM/Sesiones/6$ sudo -i service elasticsearch start
 * Starting Elasticsearch Server
```

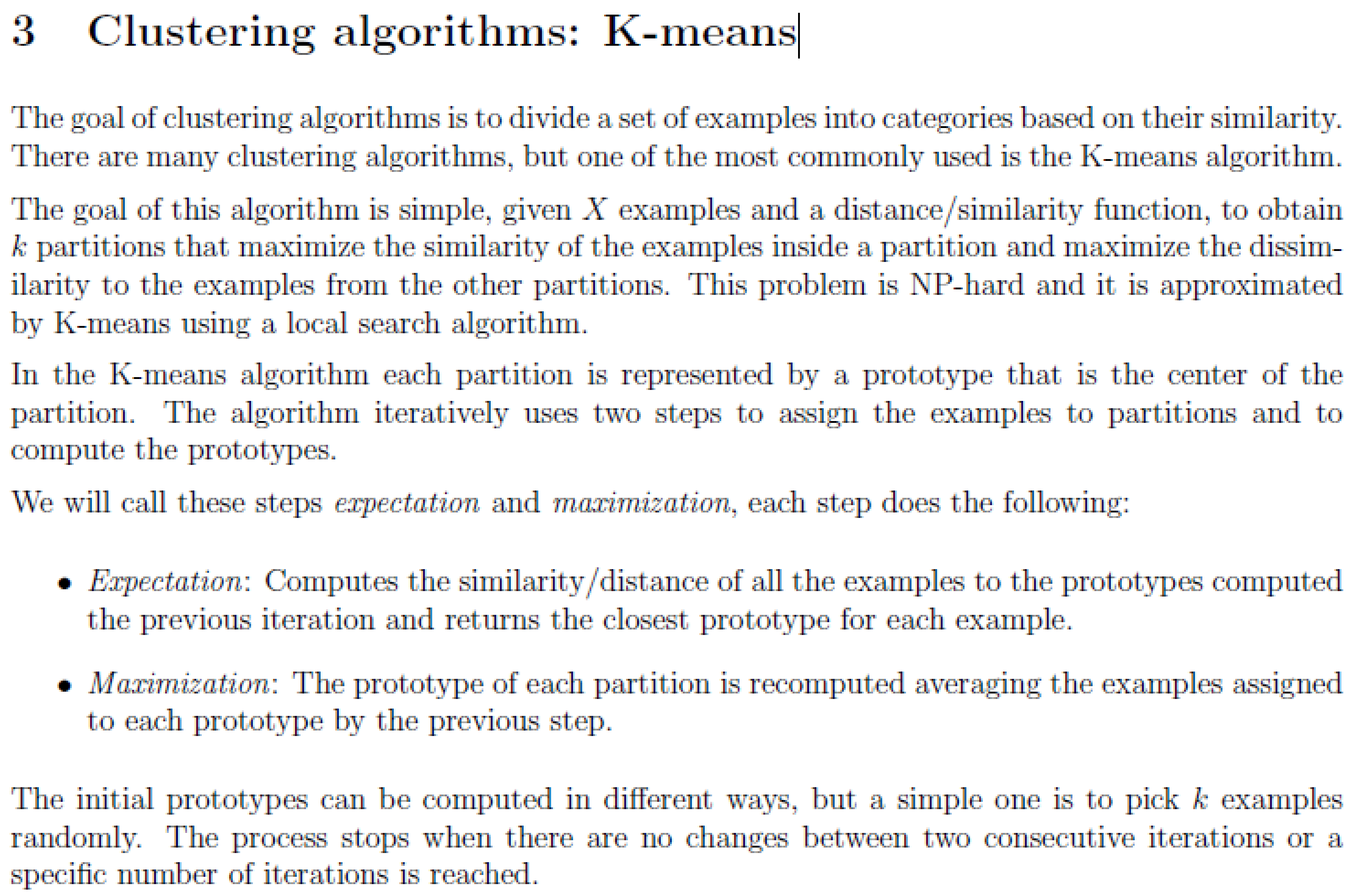


```
igomez@PelussoLaptop:~/CAIM/Sesiones/6$ python3 StreamDocs.py --index news | python3 MRWordCount.py -r local --num-cores 8
No configs found; falling back on auto-configuration
No configs specified for local runner
Creating temp directory /tmp/MRWordCount.igomez.20201120.155210.242081
Running step 1 of 1...
reading from STDIN
```



```
"zzie"  36
"zzies" 1
"zzim"  2
"zzo"   1
"zzpn"  1
"zzt"   6
"zztop" 1
"zzy"   1
"zzz"   1
"zzzt"  3
"zzzzt" 4
"zzzzz" 1
"zzzzzt"        1
"zzzzzzzzzt"    1
"zzzzzzzzzzzzzzzzzzzzzzzt"      1
Removing temp directory /tmp/MRWordCount.igomez.20201120.155210.242081...
```

K-Means



3 Clustering algorithms: K-means

The goal of clustering algorithms is to divide a set of examples into categories based on their similarity. There are many clustering algorithms, but one of the most commonly used is the K-means algorithm.

The goal of this algorithm is simple, given X examples and a distance/similarity function, to obtain k partitions that maximize the similarity of the examples inside a partition and maximize the dissimilarity to the examples from the other partitions. This problem is NP-hard and it is approximated by K-means using a local search algorithm.

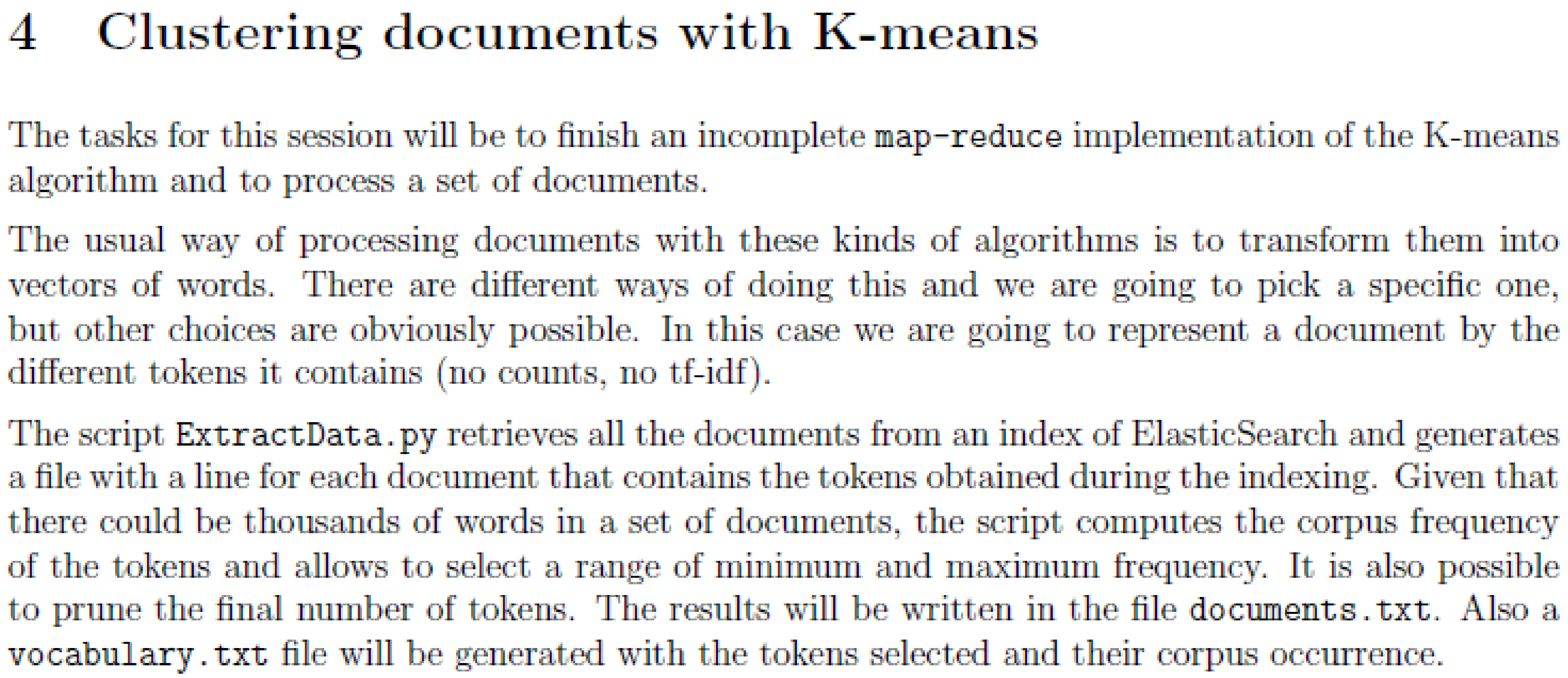
In the K-means algorithm each partition is represented by a prototype that is the center of the partition. The algorithm iteratively uses two steps to assign the examples to partitions and to compute the prototypes.

We will call these steps *expectation* and *maximization*, each step does the following:

- *Expectation*: Computes the similarity/distance of all the examples to the prototypes computed the previous iteration and returns the closest prototype for each example.
- *Maximization*: The prototype of each partition is recomputed averaging the examples assigned to each prototype by the previous step.

The initial prototypes can be computed in different ways, but a simple one is to pick k examples randomly. The process stops when there are no changes between two consecutive iterations or a specific number of iterations is reached.

K-Means on elasticSearch



4 Clustering documents with K-means

The tasks for this session will be to finish an incomplete `map-reduce` implementation of the K-means algorithm and to process a set of documents.

The usual way of processing documents with these kinds of algorithms is to transform them into vectors of words. There are different ways of doing this and we are going to pick a specific one, but other choices are obviously possible. In this case we are going to represent a document by the different tokens it contains (no counts, no tf-idf).

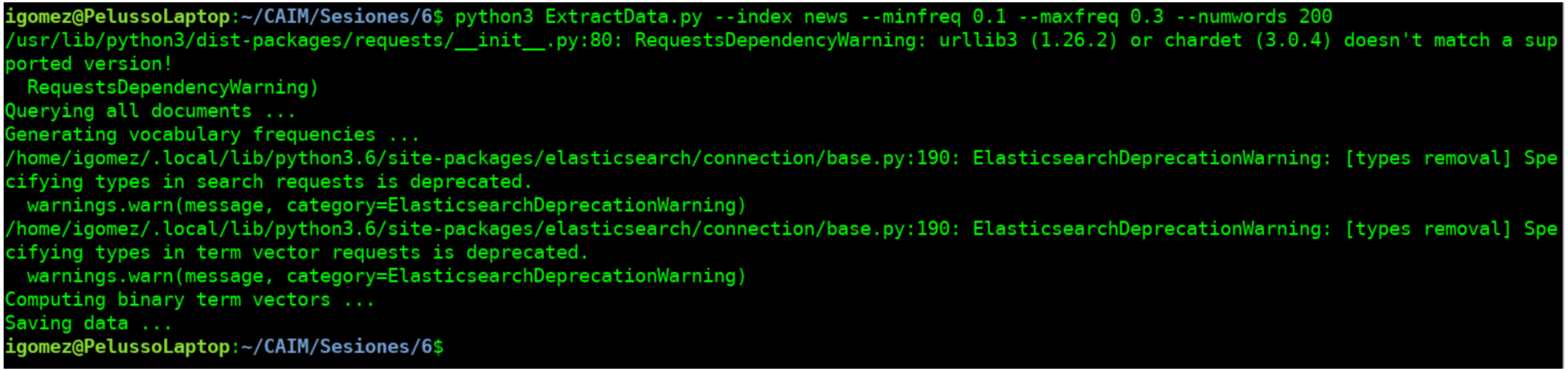
The script `ExtractData.py` retrieves all the documents from an index of ElasticSearch and generates a file with a line for each document that contains the tokens obtained during the indexing. Given that there could be thousands of words in a set of documents, the script computes the corpus frequency of the tokens and allows to select a range of minimum and maximum frequency. It is also possible to prune the final number of tokens. The results will be written in the file `documents.txt`. Also a `vocabulary.txt` file will be generated with the tokens selected and their corpus occurrence.

K-Means on elasticSearch

For example:

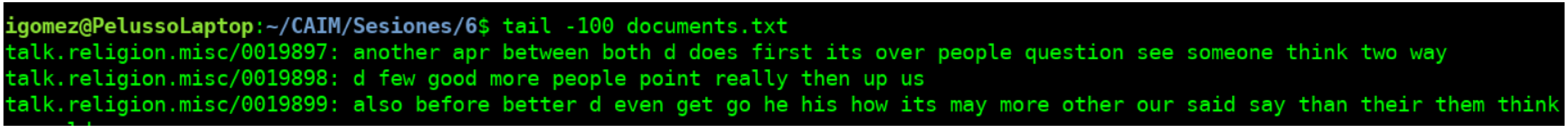
```
$ python ExtractData.py --index news --minfreq 0.1 --maxfreq 0.3 --numwords 200
```

will generate a dataset that will have words that appear at least in the 10% of the documents but no more than in the 30%. The list will be cut to the 200 more frequent words.

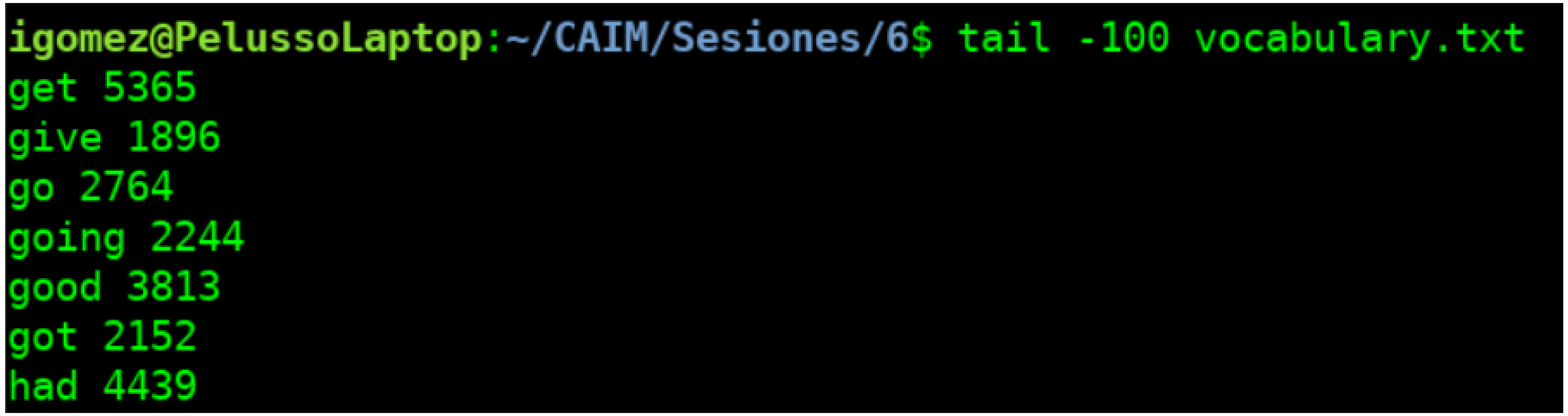


```
igomez@PelussoLaptop:~/CAIM/Sesiones/6$ python3 ExtractData.py --index news --minfreq 0.1 --maxfreq 0.3 --numwords 200
/usr/lib/python3/dist-packages/requests/__init__.py:80: RequestsDependencyWarning: urllib3 (1.26.2) or chardet (3.0.4) doesn't match a sup
ported version!
  RequestsDependencyWarning)
Querying all documents ...
Generating vocabulary frequencies ...
/home/igomez/.local/lib/python3.6/site-packages/elasticsearch/connection/base.py:190: ElasticsearchDeprecationWarning: [types removal] Spe
cifying types in search requests is deprecated.
  warnings.warn(message, category=ElasticsearchDeprecationWarning)
/home/igomez/.local/lib/python3.6/site-packages/elasticsearch/connection/base.py:190: ElasticsearchDeprecationWarning: [types removal] Spe
cifying types in term vector requests is deprecated.
  warnings.warn(message, category=ElasticsearchDeprecationWarning)
Computing binary term vectors ...
Saving data ...
igomez@PelussoLaptop:~/CAIM/Sesiones/6$
```

K-Means on elasticSearch

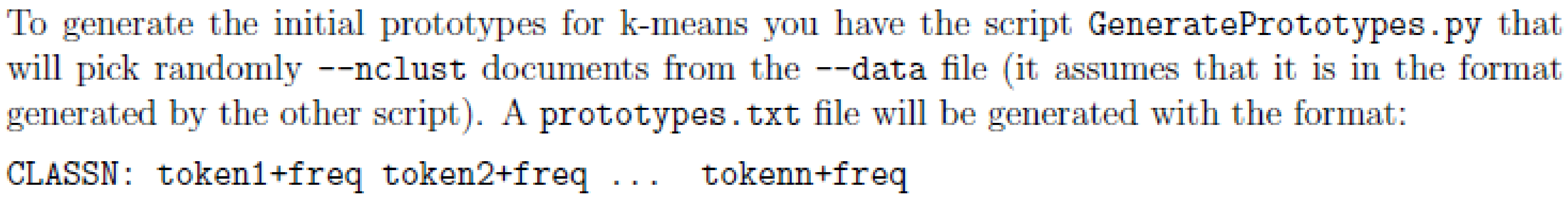


```
igomez@PelussoLaptop:~/CAIM/Sesiones/6$ tail -100 documents.txt
talk.religion.misc/0019897: another apr between both d does first its over people question see someone think two way
talk.religion.misc/0019898: d few good more people point really then up us
talk.religion.misc/0019899: also before better d even get go he his how its may more other our said say than their them think
```



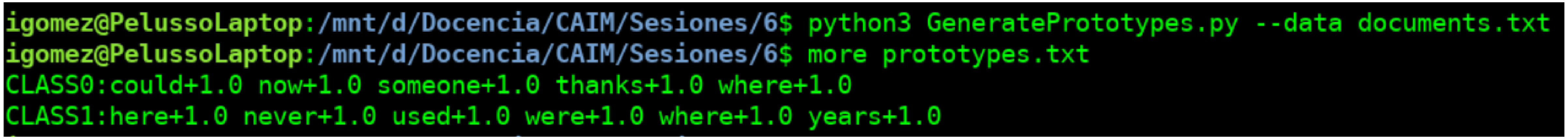
```
igomez@PelussoLaptop:~/CAIM/Sesiones/6$ tail -100 vocabulary.txt
get 5365
give 1896
go 2764
going 2244
good 3813
got 2152
had 4439
```

K-Means on elasticSearch



To generate the initial prototypes for k-means you have the script `GeneratePrototypes.py` that will pick randomly `--nclust` documents from the `--data` file (it assumes that it is in the format generated by the other script). A `prototypes.txt` file will be generated with the format:

```
CLASSN: token1+freq token2+freq ...  tokenn+freq
```



```
igomez@PelussoLaptop:/mnt/d/Docencia/CAIM/Sesiones/6$ python3 GeneratePrototypes.py --data documents.txt
igomez@PelussoLaptop:/mnt/d/Docencia/CAIM/Sesiones/6$ more prototypes.txt
CLASS0:could+1.0 now+1.0 someone+1.0 thanks+1.0 where+1.0
CLASS1:here+1.0 never+1.0 used+1.0 were+1.0 where+1.0 years+1.0
```

K-Means on elasticSearch

Because we are representing a document by its tokens, a prototype (the center of the cluster) is just the tokens of all the documents and their normalized frequency in the cluster (count word/num examples in the cluster).

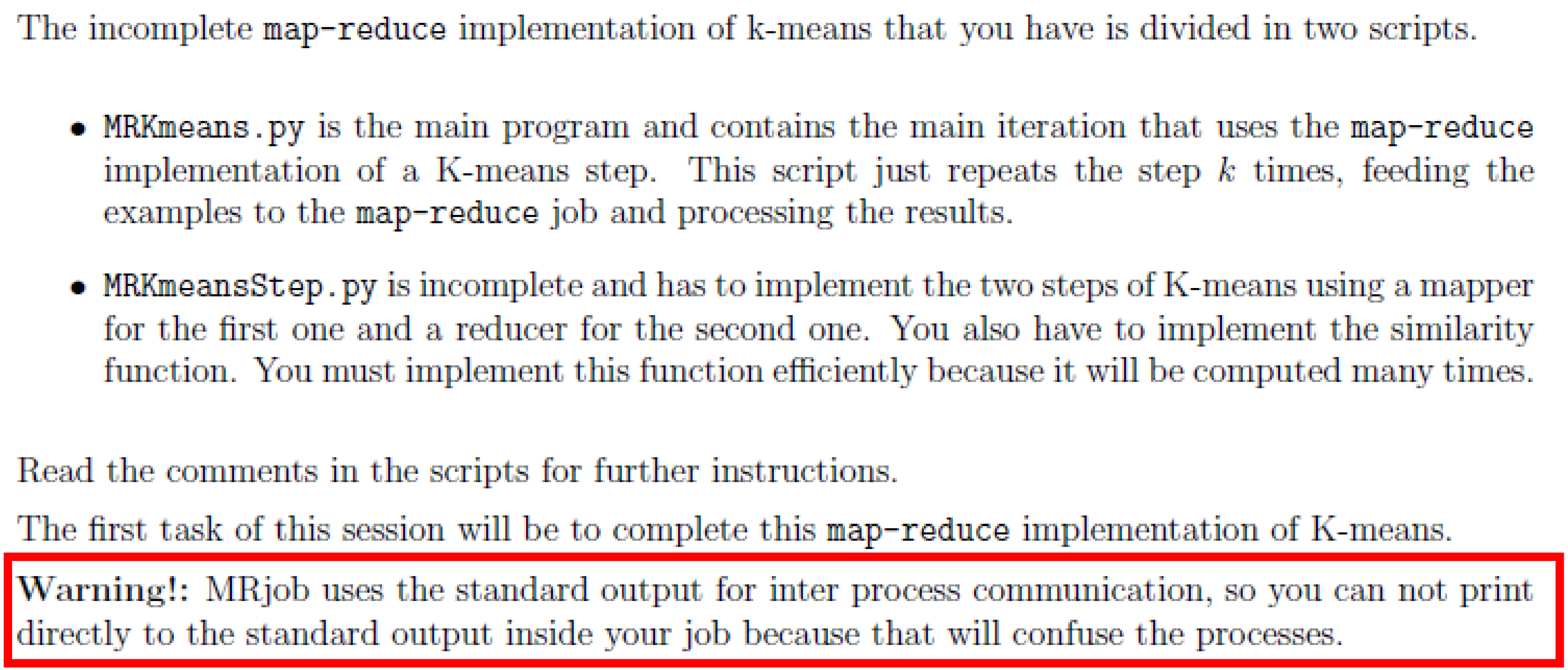
To perform the clustering we need a similarity/distance function, we can use different ones. In this case we are going to use the **Jaccard index**, but for instance, the cosine similarity could also be used. This index is defined as:

$$Jaccard(doc_1, doc_2) = \frac{doc_1 \cdot doc_2}{||doc_1||_2^2 + ||doc_2||_2^2 - doc_1 \cdot doc_2}$$

Notice the square norm of the vectors (no squared roots involved). **Notice also that the values for the tokens in the prototypes, differently from the documents, are real numbers, not 0 or 1, and that we are comparing prototypes with documents.**[3]

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

TODO



The incomplete `map-reduce` implementation of k-means that you have is divided in two scripts.

- `MRKmeans.py` is the main program and contains the main iteration that uses the `map-reduce` implementation of a K-means step. This script just repeats the step k times, feeding the examples to the `map-reduce` job and processing the results.

- `MRKmeansStep.py` is incomplete and has to implement the two steps of K-means using a mapper for the first one and a reducer for the second one. You also have to implement the similarity function. You must implement this function efficiently because it will be computed many times.

Read the comments in the scripts for further instructions.

The first task of this session will be to complete this `map-reduce` implementation of K-means.

Warning!: MRjob uses the standard output for inter process communication, so you can not print directly to the standard output inside your job because that will confuse the processes.

Mirad el script MRWordCount.py

MRJob

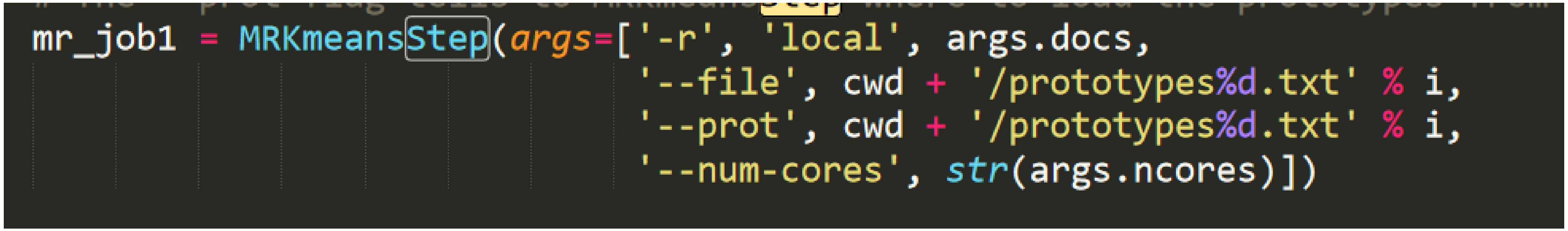
```python
"""
WordCountMR
"""
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[a-z]+")

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def reducer(self, key, values):
        yield key, sum(values)


if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

- We are streaming the raw text, so in order to discard things that are not words we are using a basic tokenization strategy that looks for strings that only contain letters using a regular expression. To normalize the words we transform all strings to lowercase.
- For obtaining the counting, the mapper emits a pair (key, value) that consists of a word and the integer 1.
- The reducer just sums all the ones that correspond to the same key, in this case the keys are the words
- The script writes the results to the standard output

Listing 1: MRWordCount.py

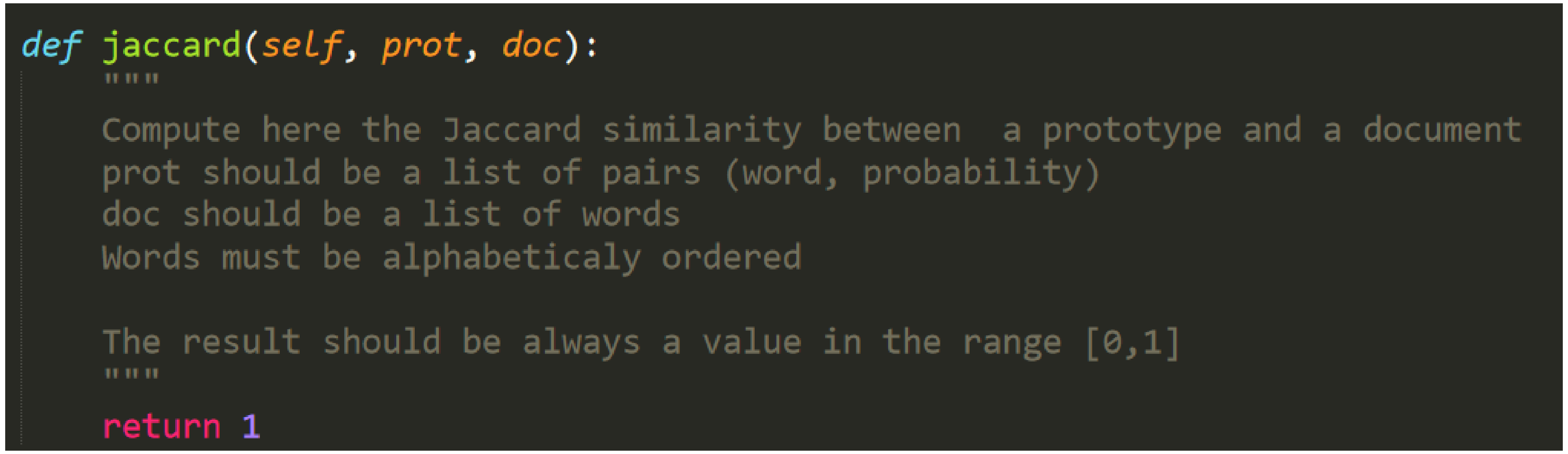
TODO

- `MRKmeans.py` is the main program and contains the main iteration that uses the `map-reduce` implementation of a K-means step. This script just repeats the step k times, feeding the examples to the `map-reduce` job and processing the results.



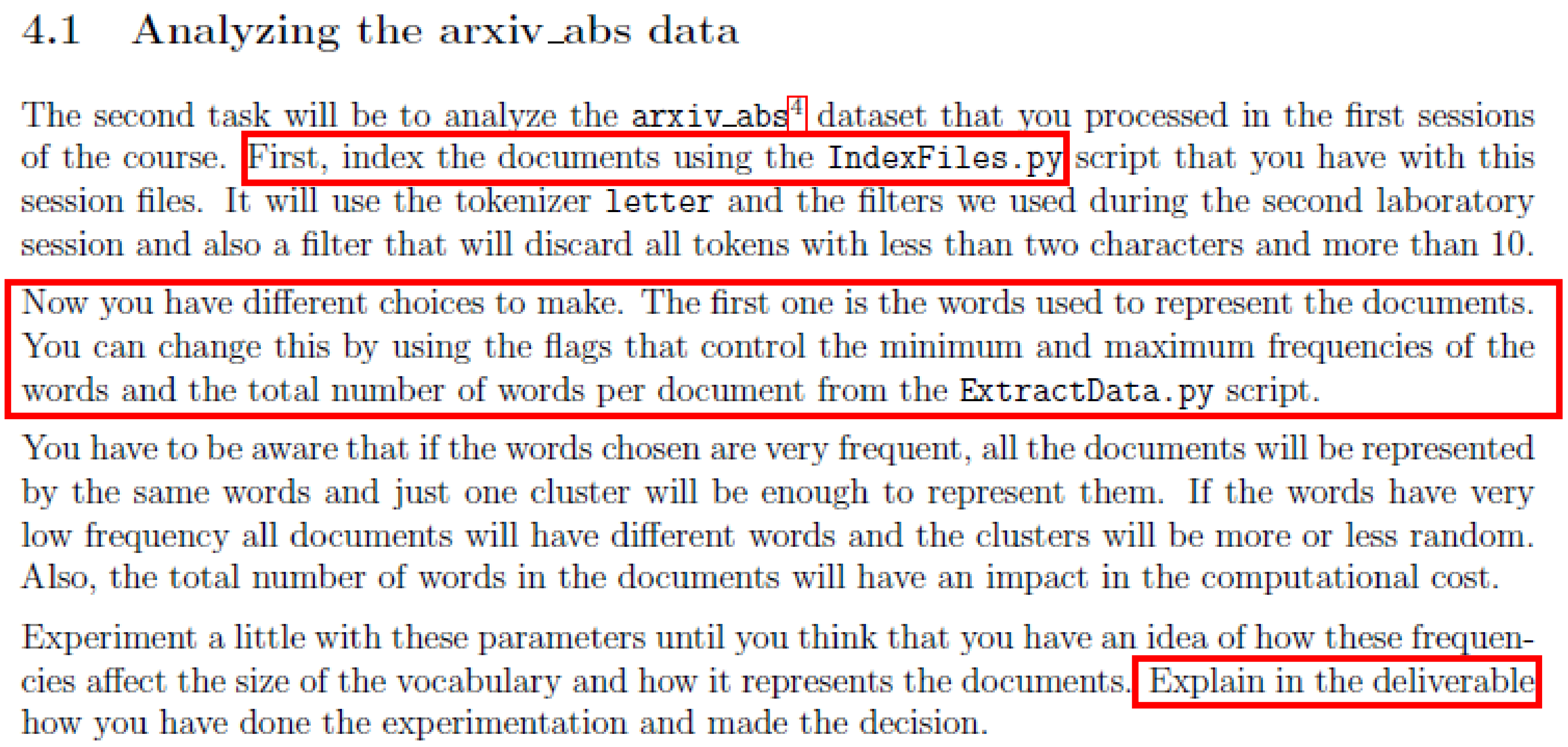
```python
mr_job1 = MRKmeansStep(args=['-r', 'local', args.docs,
                              '--file', cwd + '/prototypes%d.txt' % i,
                              '--prot', cwd + '/prototypes%d.txt' % i,
                              '--num-cores', str(args.ncores)])
```

- `MRKmeansStep.py` is incomplete and has to implement the two steps of K-means using a mapper for the first one and a reducer for the second one. You also have to implement the similarity function. You must implement this function efficiently because it will be computed many times.



```python
def jaccard(self, prot, doc):
    """
    Compute here the Jaccard similarity between  a prototype and a document
    prot should be a list of pairs (word, probability)
    doc should be a list of words
    Words must be alphabeticaly ordered

    The result should be always a value in the range [0,1]
    """
    return 1
```

TODO



4.1 Analyzing the arxiv_abs data

The second task will be to analyze the `arxiv_abs`[4] dataset that you processed in the first sessions of the course. First, index the documents using the `IndexFiles.py` script that you have with this session files. It will use the tokenizer `letter` and the filters we used during the second laboratory session and also a filter that will discard all tokens with less than two characters and more than 10.

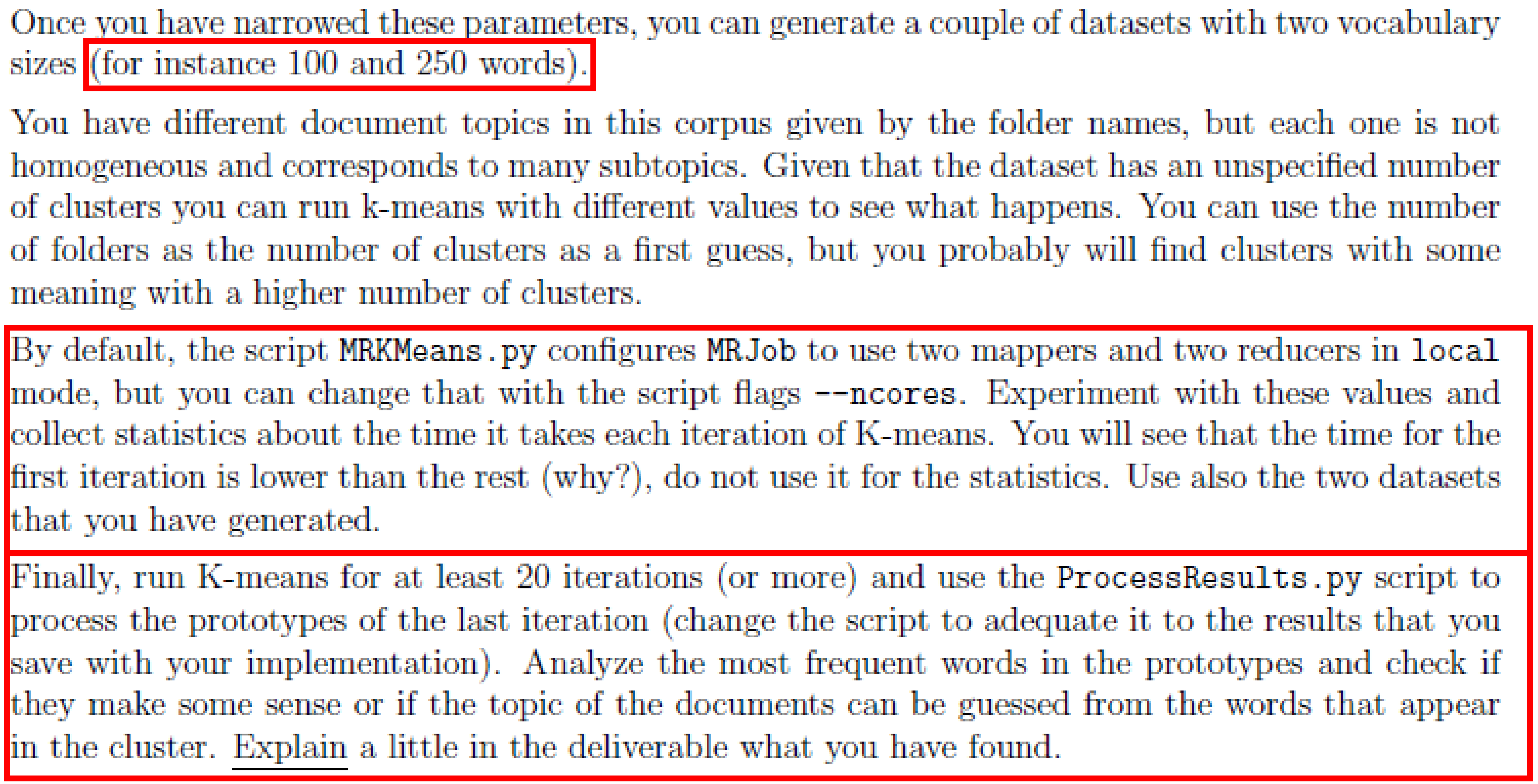
Now you have different choices to make. The first one is the words used to represent the documents. You can change this by using the flags that control the minimum and maximum frequencies of the words and the total number of words per document from the `ExtractData.py` script.

You have to be aware that if the words chosen are very frequent, all the documents will be represented by the same words and just one cluster will be enough to represent them. If the words have very low frequency all documents will have different words and the clusters will be more or less random. Also, the total number of words in the documents will have an impact in the computational cost.

Experiment a little with these parameters until you think that you have an idea of how these frequencies affect the size of the vocabulary and how it represents the documents. Explain in the deliverable how you have done the experimentation and made the decision.

Usad el indexFiles.py de **ESTA** sesión

TODO



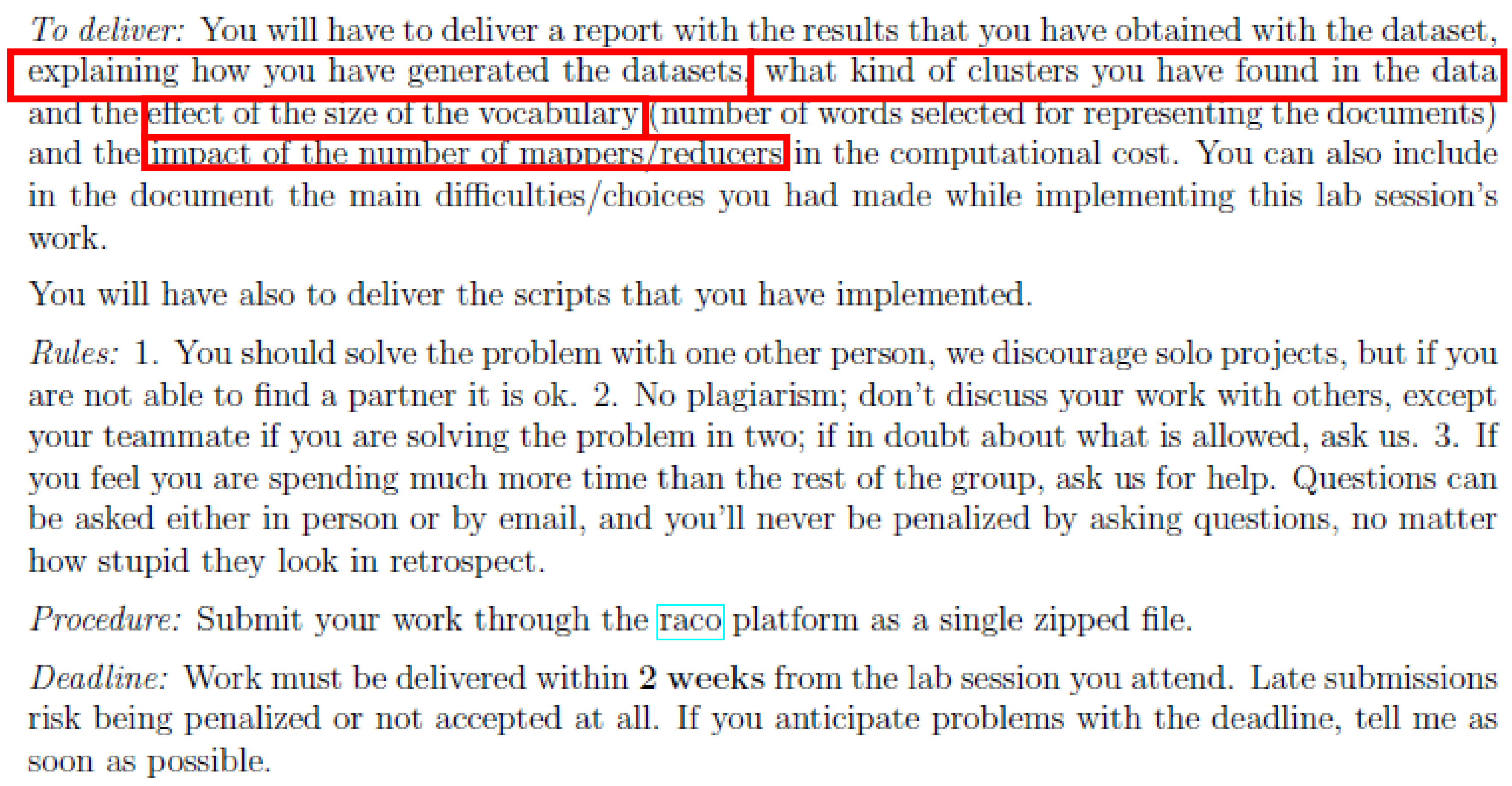
Once you have narrowed these parameters, you can generate a couple of datasets with two vocabulary sizes (for instance 100 and 250 words).

You have different document topics in this corpus given by the folder names, but each one is not homogeneous and corresponds to many subtopics. Given that the dataset has an unspecified number of clusters you can run k-means with different values to see what happens. You can use the number of folders as the number of clusters as a first guess, but you probably will find clusters with some meaning with a higher number of clusters.

By default, the script `MRKMeans.py` configures `MRJob` to use two mappers and two reducers in `local` mode, but you can change that with the script flags `--ncores`. Experiment with these values and collect statistics about the time it takes each iteration of K-means. You will see that the time for the first iteration is lower than the rest (why?), do not use it for the statistics. Use also the two datasets that you have generated.

Finally, run K-means for at least 20 iterations (or more) and use the `ProcessResults.py` script to process the prototypes of the last iteration (change the script to adequate it to the results that you save with your implementation). Analyze the most frequent words in the prototypes and check if they make some sense or if the topic of the documents can be guessed from the words that appear in the cluster. Explain a little in the deliverable what you have found.

Entrega



To deliver: You will have to deliver a report with the results that you have obtained with the dataset, explaining how you have generated the datasets, what kind of clusters you have found in the data and the effect of the size of the vocabulary (number of words selected for representing the documents) and the impact of the number of mappers/reducers in the computational cost. You can also include in the document the main difficulties/choices you had made while implementing this lab session's work.

You will have also to deliver the scripts that you have implemented.

Rules: 1. You should solve the problem with one other person, we discourage solo projects, but if you are not able to find a partner it is ok. 2. No plagiarism; don't discuss your work with others, except your teammate if you are solving the problem in two; if in doubt about what is allowed, ask us. 3. If you feel you are spending much more time than the rest of the group, ask us for help. Questions can be asked either in person or by email, and you'll never be penalized by asking questions, no matter how stupid they look in retrospect.

Procedure: Submit your work through the raco platform as a single zipped file.

Deadline: Work must be delivered within **2 weeks** from the lab session you attend. Late submissions risk being penalized or not accepted at all. If you anticipate problems with the deadline, tell me as soon as possible.

30-Noviembre-2021

Pistas

- Explicad (con experimentos) como escogéis el rango de frecuencias de términos para generar los data sets y como afectan a los clusters.
  - Min, max
- Usad la distancia de Jaccard tal como se explica en la sesión de laboratorio. Cuidado, no es binaria.
- Explorad tiempos de ejecución con diferentes números de cores
  - Por que la primera ejecución del k-means tarda menos?
- Explicad los clusters que obténeis en base a sus palabras más frecuentes, y que impacto tiene el número de clusters seleccionados.
  - Explorad con el número de clústers adecuado.