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

1 MRJob: A `map-reduce` python library

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.

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.

[1] This library has been developed by *Yelp* (a restaurant review website) as part of its software infrastructure.

```
"""
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()
```

Listing 1: MRWordCount.py

There are a few things worth noting.

- 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

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 reduceres (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.

[2]You will need ElasticSeach running and a set of documents indexed in the index news.

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.

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.

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.

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
```

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 its normalized frequency in the cluster (count word/num examples in the cluster).

To perform the clustering we need a similarity/distance function, we can also 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.

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.

4.1 Analyzing the arxiv_abs data

The second task will be to analyze the `arxiv_abs`[3] 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.

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

[3]If you are doing this task with a computer that does not allow more than two threads you should use the 20 newsgroups dataset instead.

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 in 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 by the words that appear in the cluster. Explain a little in the deliverable what you have found.

5 Deliverables

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.