Analyze letter frequency through Apache Hadoop

Report on Cloud Computing's project

Giovanni Bergami
g.bergami@studenti.unipi.it

Marco Bologna
m.bologna2@studenti.unipi.it

Gabriele Frassi
g.frassi2@studenti.unipi.it

Department of Information Engineering, University of Pisa

A.Y. 2023-2024

Contents

1 Introduction

The goal of this project is to implement a *LetterFrequency counter* using a map-reduce procedure with *Apache Hadoop* (installed on three different virtual machines), using it on some datasets of text. The following tasks have been done:

- retrieving datasets about specific languages;
- using the results of the *LetterFrequency counter* to make comparisons among languages.

A comparison among different map-reduce approaches has been done (different combiner modalities and number of reducers) and also tests on the input dimension were made, spanning from few KB to our biggest dataset of 2.37GB.

Other informations

- We used two map-reduce jobs:
  - *[Job 1]* Counting the total number of letters within a dataset.
  - *[Job 2]* Calculate the frequency of each letter within a dataset.
- For simplicity, we evaluated the frequency of letters in the Latin alphabet only, excluding accents and diacritical marks. For instance, letters like $\c{c}$ were treated as c and $\tilde{n}$ as n. This approach allows us to focus on the standard letters without distinguishing between variations caused by special characters.

2 Implementation

2.1 Execution of the code

The project can be executed using the .jar file, with the following command from the "target" folder

```
> hadoop jar file.Jar it.unipi.hadoop.Start datasetName output
  nReducers InMapperCombining
```

where

- `nReducers` specifies the number of reducers of the second job. It's important to notice that the number of reducers for the first job is always 1, since there is only one key to be reduced.
- `InMapperCombining` can be 0 (combiner, default value) or 1 (In-Mapper combiner).

2.2 Output retrieval

The output can be seen with the following command:

```
> hadoop fs -cat output/part-r*
```

2.3 Developed classes

There are four different classes, since there are two combiner modalities (*in-mapper combiner* and *combiner*) and two jobs (*count* and *frequency*). The following classes are the ones used within the Hadoop framework:

- *[Job 1, Combiner] LetterCount*
- *[Job 2, Combiner] LetterFrequency*
- *[Job 1, In-Mapper combiner] inMapperLetterCount*
- *[Job 2,In-Mapper combiner] inMapperLetterFrequency*

The execution of the second job depends on the results of the first one (the number of occurrences of letters within the analyzed dataset). A function in the start method has been implemented in order to address this aspect: it reads the results from the output, once the first job is performed, and add it to the configuration; then, during the setup of the reducer of the second job, the output is read from the configuration.

2.4 Description of the program with pseudocode

2.4.1 *Combiner LetterCount*

In this part we count the number of occurrences of letters within a dataset. The mappers convert the text in lowercase and for each letter they emit a key-value pair in the format $\langle \text{total\_letters}, 1 \rangle$. The reducer sums each received key-value pair and obtains the total number of letters. The combiner is equal to the reducer.

Pseudocode You can check the pseudocode on *Figure.1*

2.4.2 *Combiner LetterFrequency*

In this part we evaluate the frequency of each letter within a dataset.

1. The mapper takes the text and emits a key-value pair $\langle \text{letter}, 1 \rangle$. The letter is obtained from the text, and it is considered in lowercase and without accents. The combiner takes in this outputs and aggregates them.
2. From the context, during the setup, the reducer gets the total number of letters. It also aggregates the results of the combiners and divide every value within the pair $\langle \text{letter}, \text{value} \rangle$ for the total number of letters, thus obtaining the frequency. At last, it emits $\langle \text{letter}, \text{frequency} \rangle$ pairs.

Pseudocode You can check the pseudocode on *Figure.2*

2.4.3 *In-Mapper combiner LetterCount*

In this part the *LetterCount* is performed with an *In-Mapper combiner*. This means that the reducer remains the same, while the mapper uses a counter variable to count the letters of its inputSplit. In the cleanup, it emits the $\langle \text{total\_letters}, \text{count} \rangle$ pair.

Pseudocode You can check the pseudocode on *Figure.3*

2.4.4 *In-Mapper combiner LetterFrequency*

In this part the *LetterFrequency* is calculated with an *In-mapper combiner*. This means that the reducer stays the same, while the mapper uses an hashmap to store the count of the letters of its inputSplit. As before, each letter is converted in lowercase and accents are removed, then it is added to the map (if not present) or the corresponding counter variable is incremented by one. In the cleanup, a key-value pair is emitted for each entry of the hashmap.

Why Hashmap? We have chosen the hashmap because it is faster than creating a list of objects containing key-value pairs, since in order to access them it would be necessary to loop over them each time a new letter is found (too many times).

Pseudocode You can check the pseudocode on *Figure.4*

Algorithm 1 LetterCount

```
1:  Class LetterCount
2:    Class LetterCountMapper extends Mapper
3:      Variables:
4:        one ← 1
5:        tot_letters ← "total_letters"
6:
7:      Method map(key, value, context):
8:        text ← convert value to lowercase string
9:        for each character c in text:
10:         if c is a letter:
11:           write (tot_letters, one) to context
12:
13:   Class LetterCountReducer extends Reducer
14:     Variables:
15:       result ← 0
16:
17:     Method reduce(key, values, context):
18:       sum ← 0
19:       for each val in values:
20:         sum ← sum + val
21:       Set result to sum
22:       write (key, result) to context
```

Figure 1: Combiner LetterCount pseudocode

Algorithm 2 LetterFrequency

```
1:  Class LetterFrequency
2:
3:    Class LetterFrequencyMapper extends Mapper
4:      Variables:
5:        one ← 1
6:        letter ← empty text
7:
8:      Method map(key, value, context):
9:        text ← convert value to lowercase string
10:       for each character c in text:
11:         if c is a letter:
12:           letter ← strip accents from c
13:           write (letter, one) to context
14:
15:   Class LetterFrequencyCombiner extends Reducer
16:     Variables:
17:       result ← 0
18:
19:     Method reduce(key, values, context):
20:       tot ← 0
21:       for each val in values:
22:         tot ← tot + val
23:       Set result to tot
24:       write (key, result) to context
25:
26:   Class LetterFrequencyReducer extends Reducer
27:     Variables:
28:       totalLetters ← 1.0
29:
30:     Method setup(context):
31:       conf ← get configuration from context
32:       totalLetters ← get double value "totalLetters" from conf, default is 1.0
33:
34:     Method reduce(key, values, context):
35:       tot ← 0.0
36:       for each val in values:
37:         tot ← tot + val
38:       result ← tot / totalLetters
39:       write (key, result) to context
```

Figure 2: Combiner LetterFrequency pseudocode

Algorithm 3 InMapperLetterCount

```
1:  Class InMapperLetterCount
2:    Class LetterCountMapper extends Mapper
3:      Variables:
4:        count ← 0
5:        tot_letters ← "total_letters"
6:
7:      Method setup(context):
8:        Initialize count to 0
9:
10:     Method map(key, value, context):
11:       text ← convert value to lowercase string
12:       for each character c in text:
13:         if c is a letter:
14:           Increment count by 1
15:
16:     Method cleanup(context):
17:       Write (tot_letters, count) to context
18:
19:   Class LetterCountReducer extends Reducer
20:     Variables:
21:       result ← 0
22:
23:     Method reduce(key, values, context):
24:       sum ← 0
25:       for each val in values:
26:         sum ← sum + val
27:       Set result to sum
28:       write (key, result) to context
```

Figure 3: In-Mapper combiner LetterCount pseudocode

Algorithm 4 InMapperLetterFrequency

```
1:  Class InMapperLetterFrequency
2:    Class LetterFrequencyMapper extends Mapper
3:      Variables:
4:        one ← 1
5:        lettersCounter ← empty map
6:
7:      Method setup(context):
8:        Initialize lettersCounter as an empty map
9:
10:     Method map(key, value, context):
11:       data ← convert value to lowercase string
12:       for each character c in data:
13:         if c is a letter:
14:           letter ← strip accents from c
15:           if lettersCounter contains letter:
16:             Increment count of letter in lettersCounter by 1
17:           else:
18:             Add letter to lettersCounter with count 1
19:
20:     Method cleanup(context):
21:       for each entry in lettersCounter:
22:         write (entry.key, entry.value) to context
23:
24:   Class LetterFrequencyReducer extends Reducer
25:     Variables:
26:       totalLetters ← 1.0
27:
28:     Method setup(context):
29:       conf ← get configuration from context
30:       totalLetters ← get double value "totalLetters" from conf, default is 1.0
31:
32:     Method reduce(key, values, context):
33:       tot ← 0.0
34:       for each val in values:
35:         tot ← tot + val
36:       result ← tot / totalLetters
37:       write (key, result) to context
```

Figure 4: n-Mapper combiner LetterFrequency pseudocode

3 Analysis

3.1 Adopted datasets

Several datasets regarding five languages (American English, British English, Italian, Spanish and Portoguese) were used in order to make comparisons. Sources of datasets are available in the following list.

- **English language**.

  Two datasets: one with articles from the USA newspaper *The New York Times* and another one with articles from British newspaper *The Guardian*. These distinct datasets will be useful to allow comparisons between British English and American English on letter frequencies.

  - *The New York Times*
    `https://huggingface.co/datasets/ErikCikalleshi/new_york_times_news_2000_2007`
  - *The Guardian*
    `https://huggingface.co/datasets/Stefan171/TheGuardian-Articles`

- **Italian language**.

  One dataset with Italian articles from the local online newspaper *GoNews* (active in Tuscany, particularly in *Empolese-Valdelsa territory*). This dataset is homemade, created on Python with *BeautifulSoup package*.

  - *Gonews, source page*
    `https://www.gonews.it/homepage/gonews24/`

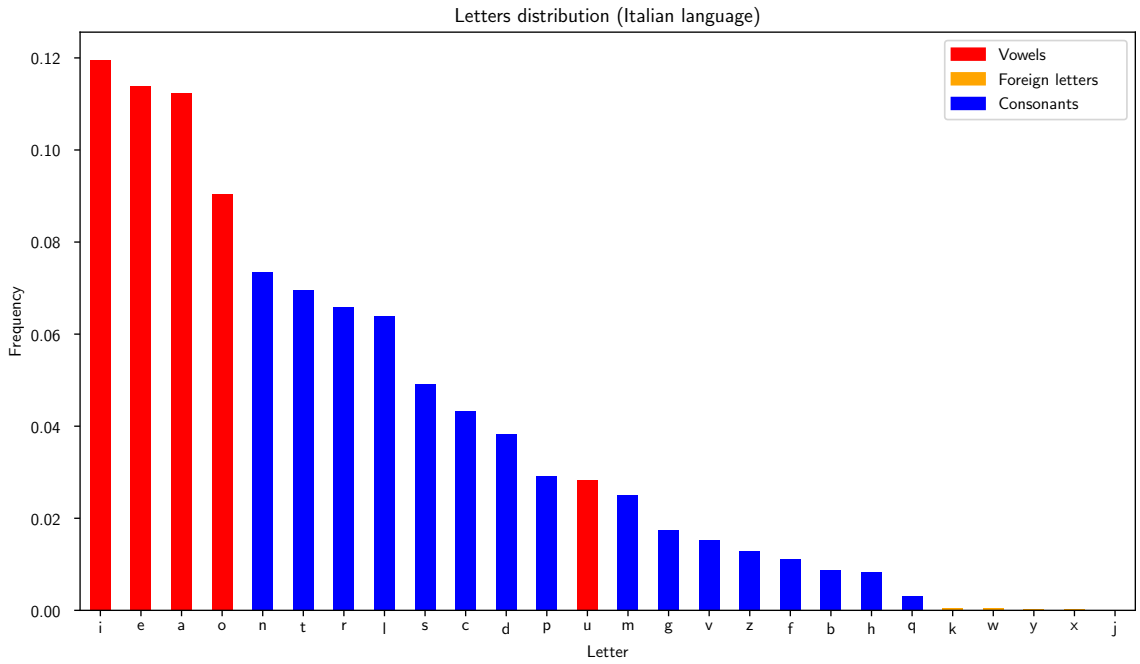
- **Spanish and Portoguese languages**.

  Datasets with Spanish and Portoguese articles, without references to particular newspapers.

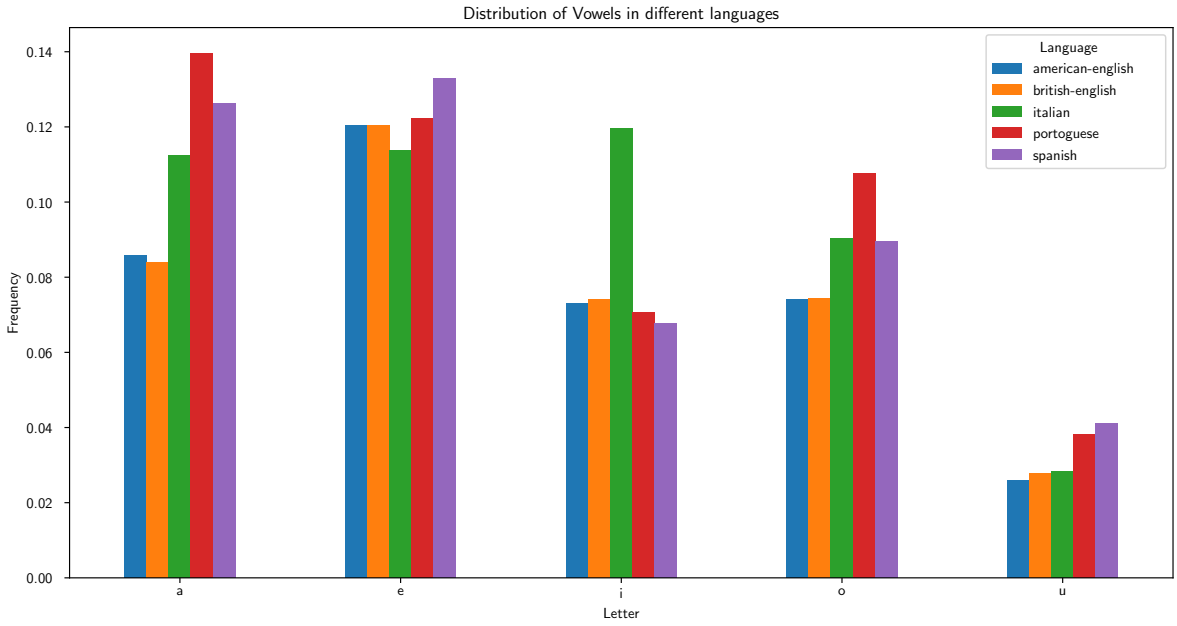
  - *Spanish articles*
    `https://huggingface.co/datasets/ELiRF/dacsa`
  - *Portoguese articles*
    `https://huggingface.co/datasets/iara-project/news-articles-ptbr-dataset`

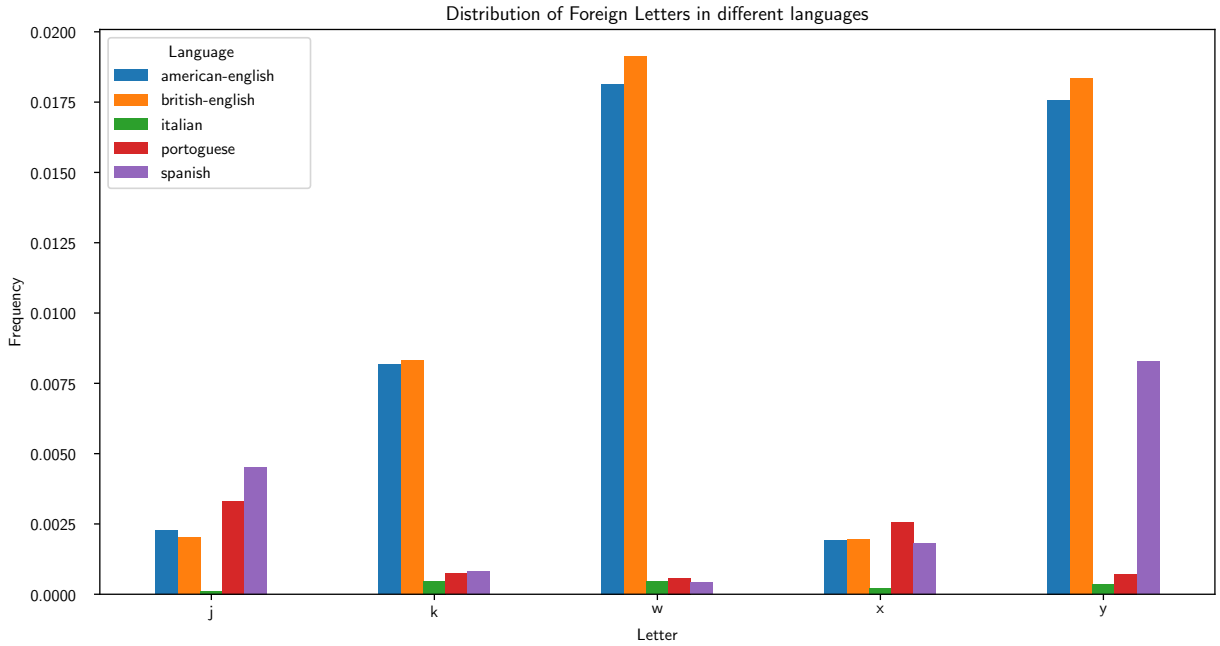
3.2 Insights on letter frequency

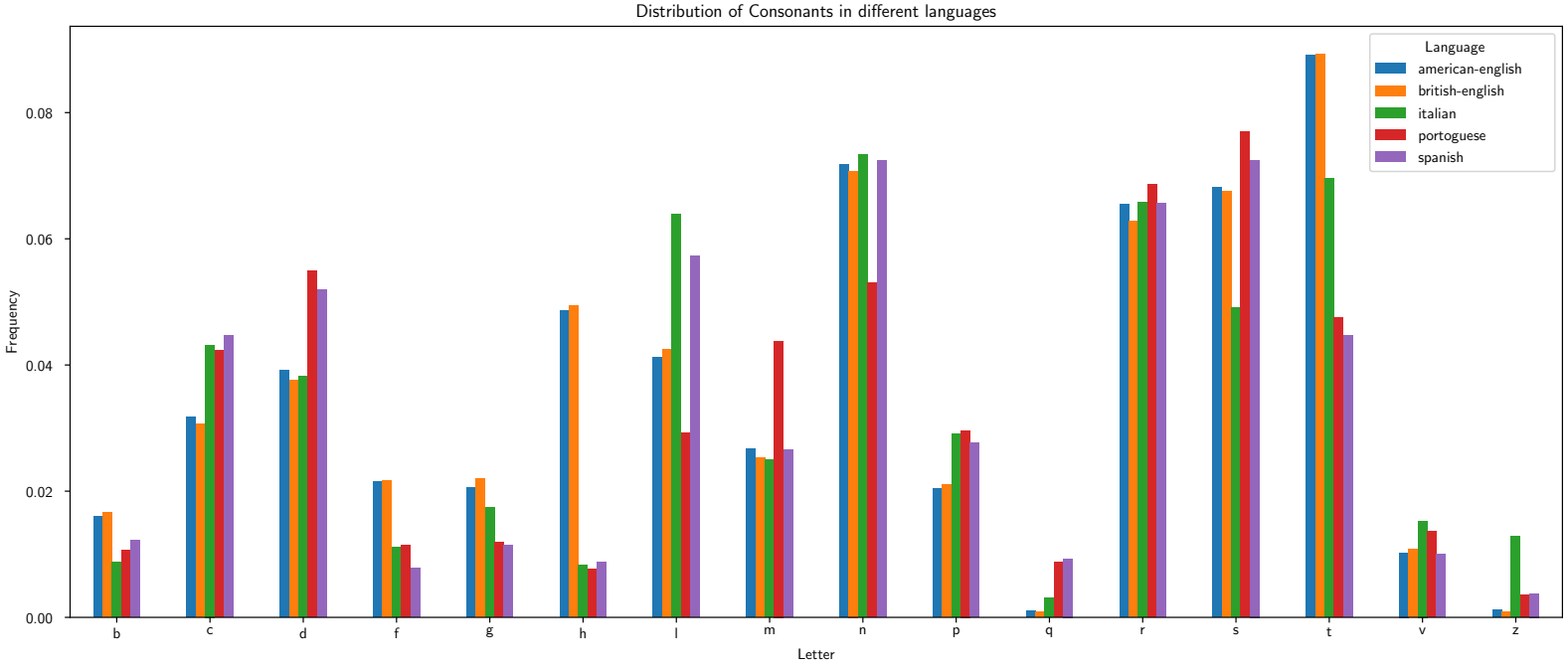
The output of the map-reduce procedure is the frequency of the 26 characters of the alphabet based on the input text, an example for the Italian language is shown in the graph below:



The results revealed that in general, vowels tend to occur more frequently than consonants across languages, reflecting their essential role in forming syllables and words.
The following graphs shows differencies in letter frequencies between different languages:



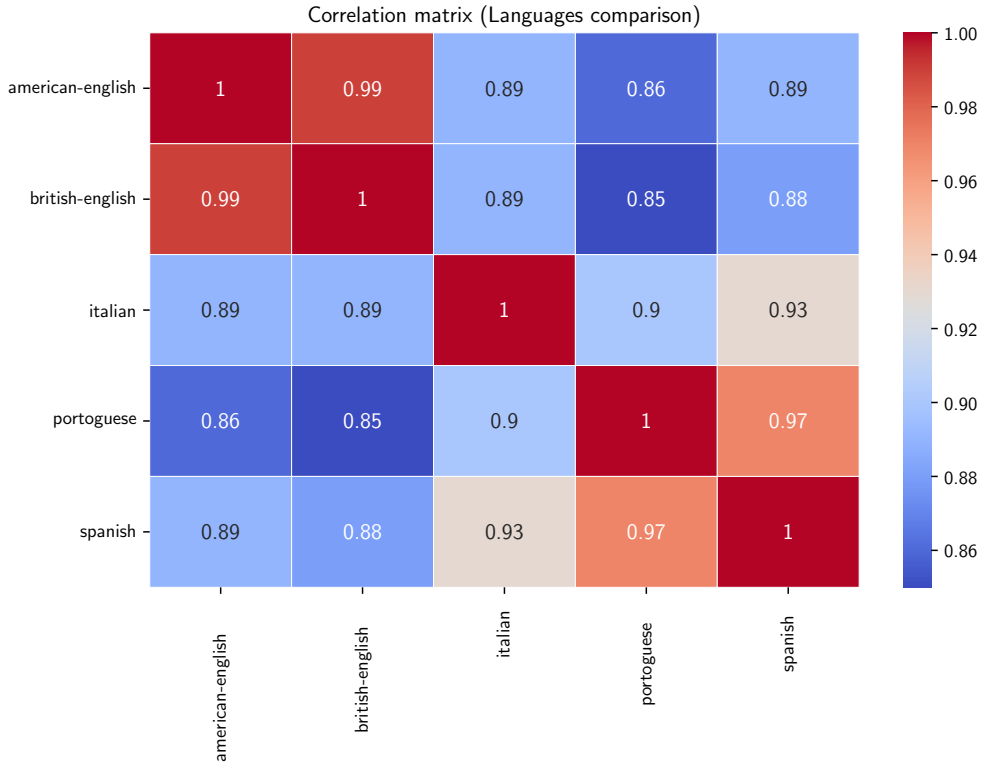




Correlation matrix Additionally, examining the correlation matrix on these languages reveals interesting insights.

- The correlation matrix highlights a strong correlation between British English and American English, which is expected given their linguistic similarities. Therefore, while there may be subtle differences in letter usage or frequency due to regional preferences or historical influences, the overall patterns remain strongly aligned.

- Additionally, there is a notable correlation between Spanish and Portuguese, reflecting their linguistic affinity and Italian shows a closer similarity to Spanish rather than to Portuguese, based on our results.

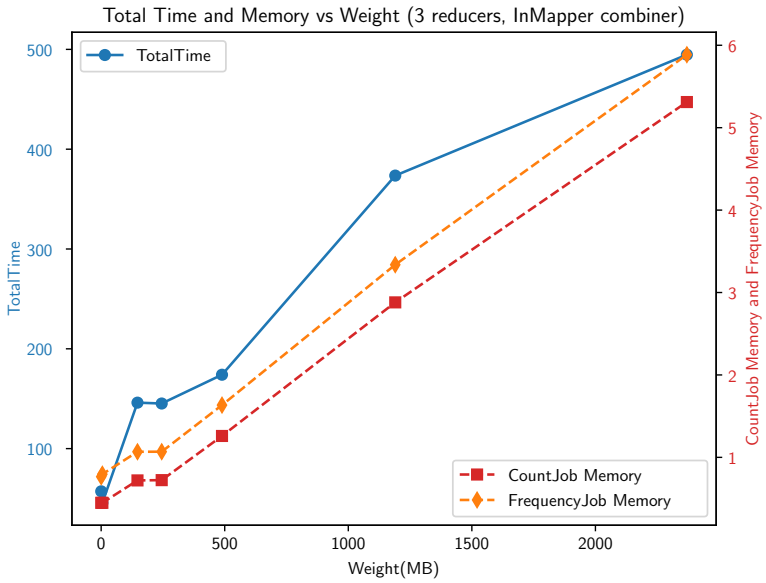


3.3 Statistics on *MapReduce* execution using splits of *NYT dataset*

The maximum reached number of inputSplits is 19.

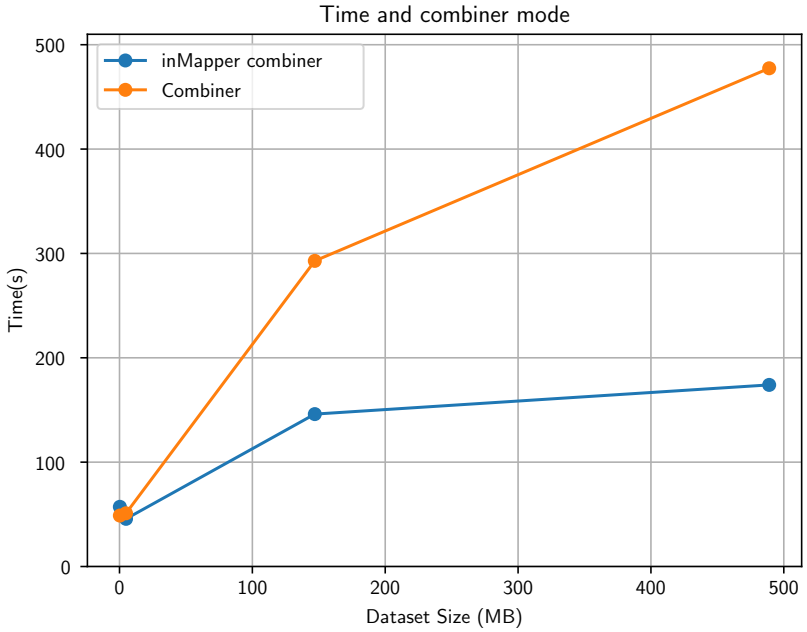
3.3.1 Performance and dataset size

As shown in the plot, we can see that as the size of the dataset increases, the time of completion increases less than linearly, while the memory usage increases linearly. The first job's occupied memory is always less than the second job's one, and the difference remains constant, so we can conclude that it doesn't depend on the size of the dataset.



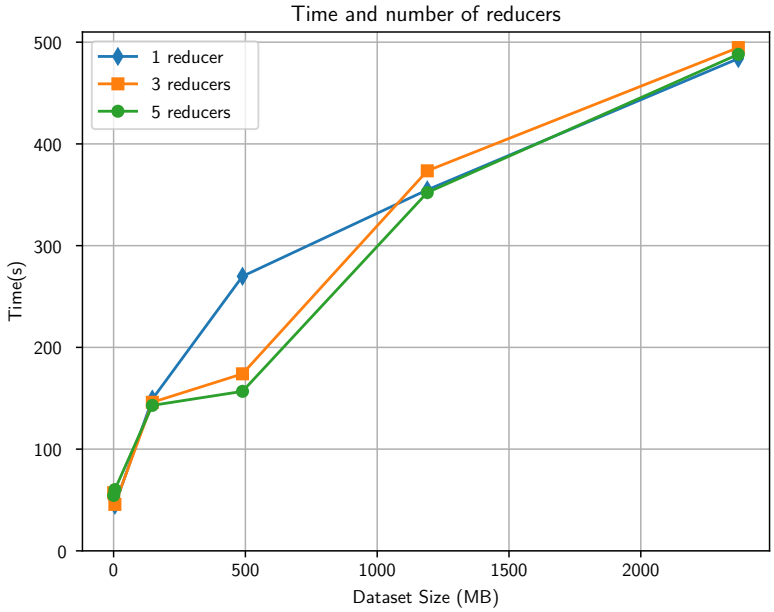
3.3.2 Performance and combiner mode

In this graph we can see that usage of the *In-mapper combiner* significantly speeds up the performance of the system, but the improvement on very small datasets is negligible.



3.3.3 Performance comparison: number of reducers

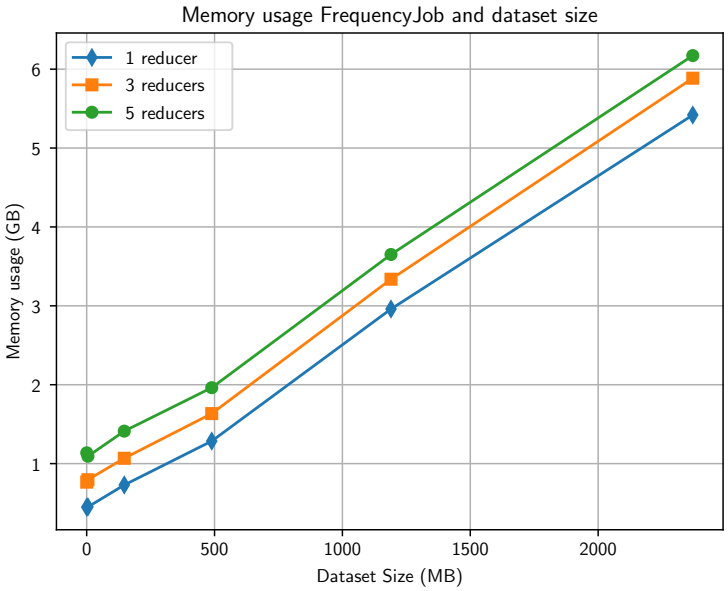
Time In this graph we can see the different performances of time within respect to the dataset size, considering different possible numbers of reducers.



In general, it seems that the number of the reducers doesn't affect the performance too much, at least for dataset of this size and considering that we have three machines to run Hadoop.

Maybe, while using with bigger datasets or more machines, the spent time should increase if less reducers are used. Look at the particular behaviour of the 500 MB dataset. As said in the beginning, all this tests were performed using different splits of the same dataset.

Memory usage The memory usage increases using more reducers, but the increase don't depend on the dataset size.



3.4 Performance comparison: distributed vs not-distributed approaches

A comparison of performances has been done between Hadoop execution and the execution of a program with the same task, but without a distributed approach.

A fourty-six-lines code has been written using Python. The result showed us that Hadoop adoption is convenient only when we are working on datasets of significant dimensions: time execution is smaller on Python with datasets with a $<$ 500 MB size, due to overhead of Hadoop executions.