Letter Frequency Analysis through Hadoop MapReduce

Matteo Comini Niccolò Settimelli Lorenzo Vittori

A.A. 2023/2024

Contents

1 Introduction

In the realm of natural language processing, understanding the linguistic relationships between languages is fundamental. One intriguing aspect is to explore how the frequency distribution of letters varies across different languages and how closely related they are to their ancestral language, Latin. Latin, as the mother language of several modern European languages, provides a rich basis for comparative linguistic analysis.

The objective of this project is to employ the MapReduce framework in Hadoop to analyze texts in various languages, specifically English, French, Spanish, Italian, and Romanian, in comparison to Latin. The analysis focuses on two main functions: *LetterCount* to tally occurrences of each letter in the texts, and *LetterFrequency* to compute the proportional frequency of each letter relative to the total number of letters in the text.

By leveraging MapReduce, we can efficiently process large volumes of text data distributed across multiple nodes. This approach allows us to calculate letter frequencies in parallel, making it scalable for extensive linguistic datasets.

1.1 Research Questions

This study aims to address the following research questions:

1. **Similarity to Latin:** Which of the analyzed languages (English, French, Spanish, Italian, Romanian) exhibits the closest letter frequency distribution to Latin?

2. **Influence of Language Evolution:** How have the letter frequencies evolved from Latin to its descendant languages over time?

Understanding these nuances not only sheds light on historical linguistic evolution but also has practical implications for fields such as cryptography, language translation, and stylometry.

In the subsequent sections, we will delve into the methodology, implementation details using Hadoop's MapReduce paradigm, results, and discussions to draw insights into the linguistic relationships and evolutionary trajectories of these languages.

2 Algorithm Design

2.1 InMapper approach

This pseudocode describes the LetterCount function using the InMapper method. The mapper processes the input text to count the total number of letters, and the reducer aggregates these counts.

Algorithm 1 Letter Count

```
    class LetterCounterMapper
 1: procedure setup
 2:     reducerValue ← 0
 3: end procedure
 4: procedure map(docid a, doc d)
 5:     line ← toString(d)
 6:     for c in line do
 7:         carattere ← processCharacter(c)
 8:         if carattere ≠ 0 then
 9:             reducerValue ← reducerValue + 1
10:         end if
11:     end for
12: end procedure
13: procedure cleanup
14:     emit("total_letters", reducerValue)
15: end procedure
```

Algorithm 2 Letter Count Reduction

```
    class LetterCounterReducer
1: procedure reduce(term t, counts [c1, c2, ...])
2:     sum ← 0
3:     for count in counts [c1, c2, ...] do
4:         sum ← sum + count
5:     end for
6:     emit(t, sum)
7: end procedure
```

This pseudocode outlines the LetterFrequency function using the InMapper method. The mapper processes the input text to count occurrences of each letter, and the reducer calculates the frequency of each letter based on the total letter count.

Pseudocode of the Letter Frequency Algorithm

Algorithm 3 Letter Frequency Mapper

```
    class FrequencyLetterMapper
 1: procedure setup
 2:     mapletter ← new HashMap()
 3: end procedure
 4: procedure map(docid a, doc d)
 5:     line ← toString(d)
 6:     for c in line do
 7:         carattere ← processCharacter(c)
 8:         if carattere ≠ 0 then
 9:             if mapletter.containsKey(carattere) then
10:                 count ← mapletter.get(carattere)
11:                 count ← count + 1
12:                 mapletter.put(carattere, count)
13:             else
14:                 mapletter.put(carattere, 1)
15:             end if
16:         end if
17:     end for
18: end procedure
19: procedure cleanup
20:     for entry in mapletter.entrySet() do
21:         emit(entry.getKey(), entry.getValue())
22:     end for
23: end procedure
```

Algorithm 4 Letter Frequency Reducer

```
    class FrequencyLetterReducer
 1: procedure setup
 2:     totalLetter ← context.getConfiguration().getLong("totalLetters",
    1)
 3: end procedure
 4: procedure reduce(letter l, counts [c1, c2, ...])
 5:     sum ← 0
 6:     for count in counts do
 7:         sum ← sum + count
 8:     end for
 9:     frequency ← sum / totalLetter
10:     emit(l, frequency)
11: end procedure
```

2.2 Combiner Approach

This pseudocode describes the LetterCount function using the Combiner method. The mapper processes the input text to count the total number of letters, the combiner aggregates intermediate counts, and the reducer aggregates these counts.

Algorithm 5 Letter Count Mapper

```
    class LetterCounterMapper
 1: procedure map(docid a, doc d)
 2:     reducerKey ← "total_letters"
 3:     reducerValue ← 1
 4:     line ← toString(d)
 5:     for c in line do
 6:         carattere ← processCharacter(c)
 7:         if carattere ≠ 0 then
 8:             emit(reducerKey, reducerValue)
 9:         end if
10:     end for
11: end procedure
```

Algorithm 6 Letter Count Reducer

```
    class LetterCounterReducer
 1: procedure reduce(term t, counts [c1, c2, ...])
 2:     sum ← 0
 3:     for count in counts [c1, c2, ...] do
 4:         sum ← sum + count
 5:     end for
 6:     emit(t, sum)
 7: end procedure
```

This pseudocode outlines the LetterFrequency function with a Combiner. The mapper processes the input text to count occurrences of each letter. The combiner aggregates intermediate counts, and the reducer calculates the frequency of each letter based on the total letter count

Algorithm 7 Letter Frequency Combiner

```
    class FrequencyLetterCombiner
 1: procedure reduce(term t, counts [c1, c2, ...])
 2:     sum ← 0
 3:     for count in counts [c1, c2, ...] do
 4:         sum ← sum + count
 5:     end for
 6:     emit(t, sum)
 7: end procedure
```

Algorithm 8 Letter Frequency Reducer

```
    class FrequencyLetterReducer
 1: procedure setup
 2:     totalLetter ← context.getConfiguration().getLong("totalLetters",
    2)
 3: end procedure
 4: procedure reduce(term t, counts [c1, c2, ...])
 5:     sum ← 0
 6:     for count in counts [c1, c2, ...] do
 7:         sum ← sum + count
 8:     end for
 9:     frequency ← sum / totalLetter
10:     emit(t, frequency)
11: end procedure
```

3 Results

3.1 Analysis of Aeneid Translations in Romance Languages

The outputs of various translations of the *Aeneid* in the five national languages derived from Latin were analyzed alongside the output of the original text using Jupyter Notebook. The goal was to find correlations between them, specifically to determine which language most closely resembles the original Latin based on letter frequency in the text.

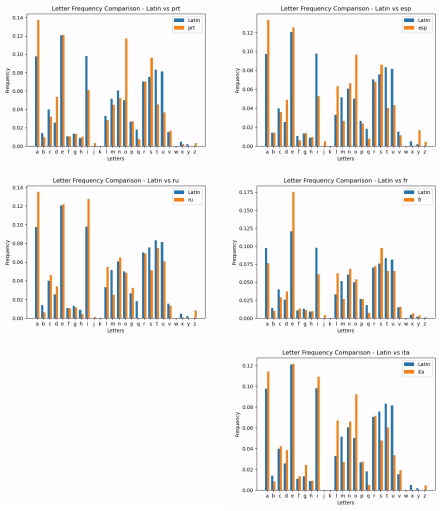

Figure 1: letter frequencies

From the graph, we observe that the frequency trends among the various languages do not differ significantly. Notably, there are large peaks for vowels, which are fundamental in Romance languages, and a sparse presence of letters derived from the Anglo-Saxon alphabet such as *j, k, w, x, y.*

To better understand the similarity between these languages, we examine the correlation matrix below, focusing on the row corresponding to Latin. As expected, there is a very strong correlation coefficient between all the languages and Latin, the mother tongue. Specifically, we observe a maximum correlation coefficient of 0.94 for Romanian and a minimum of 0.84 for the languages of the Iberian Peninsula. On the left, we also present the correlation matrix for vowels. Interestingly, the use of vowels does not have a high correlation coefficient among all languages. In some cases, the correlation coefficients decrease significantly in relation to Latin, such as for Italian, Spanish, and Portuguese.

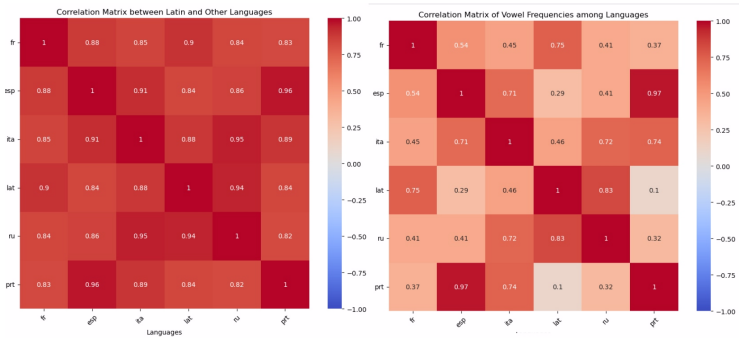

Figure 2: correlation matrices

This initial analysis suggests that Romanian maintains a very strong correlation with the original Latin. However, we can conduct further analyses:

```
Language: fr
Most frequent vowel: e - Frequency: 0.1753405864550382 - Vowel closeness coefficient: 1.45
Most frequent consonant: s - Frequency: 0.0970025511352281 - Consonant closeness coefficient: 1.28

Language: esp
Most frequent vowel: a - Frequency: 0.1332197607754088 - Vowel closeness coefficient: 1.37
Most frequent consonant: s - Frequency: 0.0860333368028192 - Consonant closeness coefficient: 1.14

Language: ita
Most frequent vowel: e - Frequency: 0.1213377988803217 - Vowel closeness coefficient: 1.01
Most frequent consonant: r - Frequency: 0.071572185237911 - Consonant closeness coefficient: 1.02

Language: lat
Most frequent vowel: e - Frequency: 0.1205849544279689 - Vowel closeness coefficient: 1.00
Most frequent consonant: t - Frequency: 0.0831587539110325 - Consonant closeness coefficient: 1.00

Language: ru
Most frequent vowel: a - Frequency: 0.1349427002102122 - Vowel closeness coefficient: 1.38
Most frequent consonant: t - Frequency: 0.0749983047399471 - Consonant closeness coefficient: 0.90

Language: prt
Most frequent vowel: a - Frequency: 0.1373519354792168 - Vowel closeness coefficient: 1.41
Most frequent consonant: s - Frequency: 0.0962401350434995 - Consonant closeness coefficient: 1.27
```

Figure 3: language statistics

For each Romance language, we collected the most frequently used vowel and consonant, and calculated the fraction of their frequency relative to the equivalent frequency in the Latin text. We observe that the most common letter is always a vowel, and it is noteworthy that the frequency of these letters is closely aligned with their use in Latin.

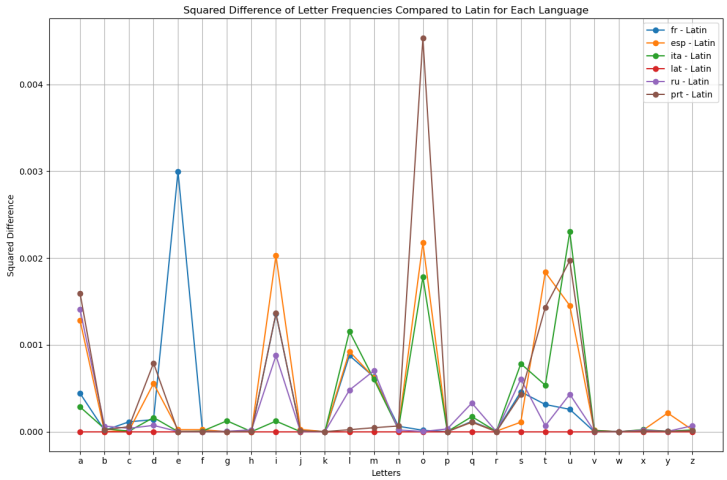

Figure 4: quadratic difference distributions

As a final analysis, we present the distribution of the quadratic difference of all letters between the various languages and Latin, to highlight where this distribution has minimal values. This analysis clearly shows that most languages have undergone significant influences from other linguistic components, resulting in considerable differences in the frequency of certain letters. The use of vowels has markedly increased in all Mediterranean languages, along with general changes in frequency across the alphabet. Romanian, however, consistently exhibits a much lower quadratic difference compared to other Romance languages. Specifically, it has an average quadratic difference of 0.00023, significantly lower than that of the other languages, which hover around 0.004.

This brief analysis demonstrates that all five translations of the *Aeneid* exhibit a letter frequency distribution very similar to that of their Latin root. Nevertheless, due to various linguistic influences after the fall of the Roman Empire, many of these languages have incorporated numerous other influences. Among these, Romanian stands out as the language that has retained the closest affinity to Latin, despite the fact that Romania is geographically distant from Rome. This is primarily because Dacia, the last region conquered by Rome, did not experience the same barbarian invasions as the European regions bordering the Mediterranean. Consequently, other linguistic influences arrived later and less effectively, creating a case of linguistic isolation.

3.2 Performance evaluation

Analyzing the performance of InMapper and Combiner methods for processing files of 3KB and 3MB reveals distinct characteristics and practical implications:

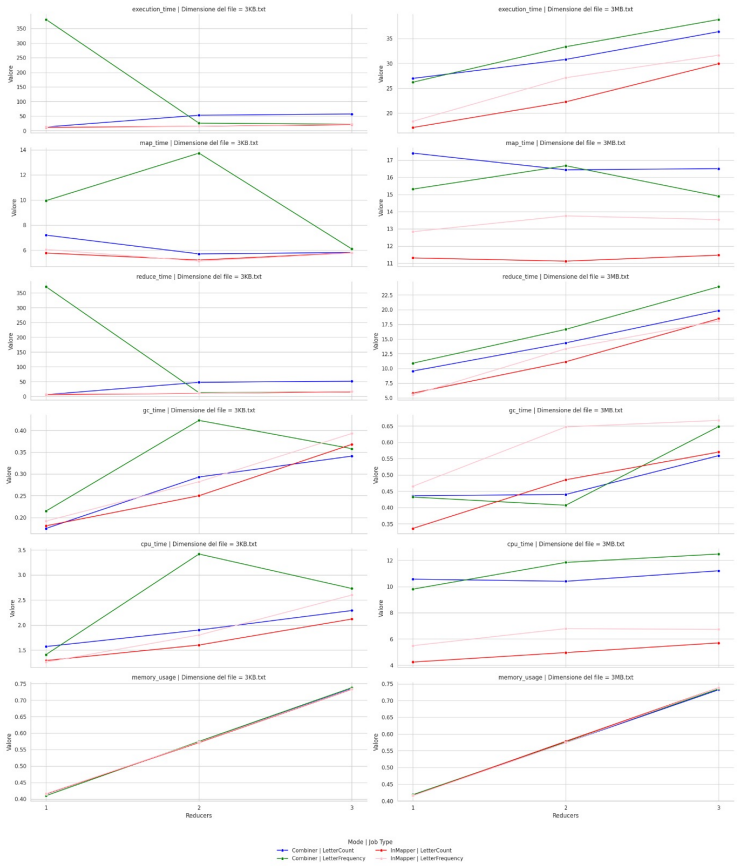

Figure 5: Comparing performance of two methods with different number of reducers

- **InMapper Method:**
  - Maintains low and stable execution and mapping times across both small (3KB) and large (3MB) files.
  - Processes data directly, bypassing intermediate steps, which reduces overhead.
  - Particularly effective for applications demanding rapid data processing and minimal delays.
  - Shows predictable overhead in garbage collection and CPU usage, increasing with more reducers.
- **Combiner Method:**
  - Exhibits more variability in performance, especially for small files.

  - Higher initial execution times for small files, improves with more reducers in LetterFrequency mode.
  - LetterCount mode remains stable but shows a slight increase in execution time with more reducers.
  - For larger files, shows steady rise in execution and reduction times as file size and number of reducers grow.
  - Higher CPU load and garbage collection times, particularly in LetterFrequency mode.

- **Memory Usage:**
  - Both methods show a linear increase in memory usage with more reducers.
  - InMapper has predictable overhead; Combiner introduces additional complexity.

- **Design and Application:**
  - InMapper's direct data processing minimizes intermediate steps and overhead, suitable for rapid processing of both small and large files.
  - Combiner's additional processing layer can reduce data volume before the reduce phase but adds complexity and overhead.
  - Optimal performance requires careful consideration of specific application needs and file characteristics.

These findings highlight that InMapper is generally more efficient for rapid data processing with minimal delays, while Combiner can be beneficial for reducing data volume but requires careful configuration to manage its complexity and overhead effectively.

At a later time we decide to compare this result with that of a larger file.

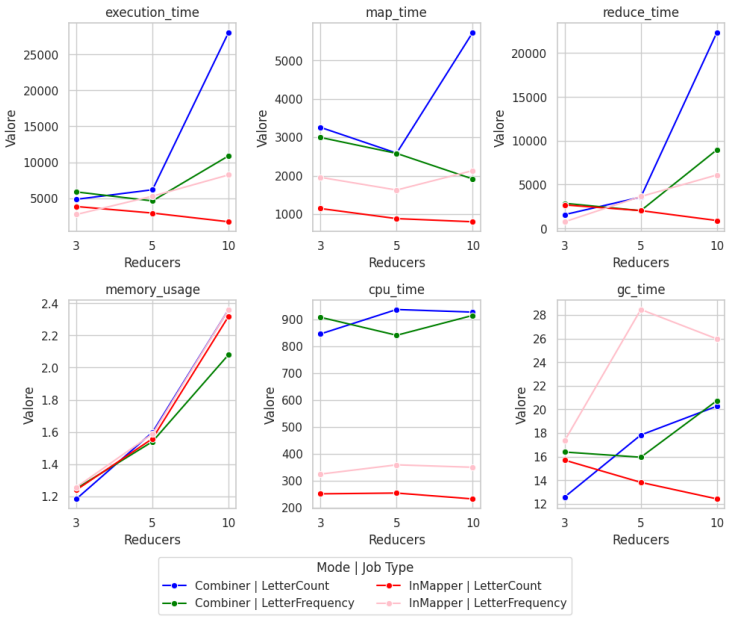

Figure 6: Comparing performance of two methods with different number of reducers for a big text file

Analyzing the performance of InMapper and Combiner methods for a 300MB file, considering variations in the number of reducers (3, 5, and 10), reveals important insights into their efficiencies and behaviors under heavy data loads. This discussion integrates key observations on execution time, mapping time, reduction time, garbage collection time, CPU time, and memory usage, offering a cohesive view of how each method handles large datasets.

- **Execution Time:**
  - **InMapper:**
    - *LetterCount:* Shows a significant reduction in execution time with an increase in the number of reducers, indicating effective workload distribution without additional complexity.
    - *LetterFrequency:* Maintains stable execution times, suggesting effective workload distribution without significant advantages from additional reducers.
  - **Combiner:**
    - *LetterCount:* Experiences a notable increase in execution time as more reducers are added, rising from 5000 ms to over 25,000 ms, reflecting inefficiency in handling the combination phase with more reducers.

    - *LetterFrequency:* Manages better but still shows an increase in times, indicating that the combination phase adds complexity and overhead with more reducers.

- **Mapping and Reduction Times:**
  - **InMapper:**
    - Consistently low mapping times (1000 ms), reflecting its ability to map data without significant complexity.
    - Effectively managed reduction times, with LetterCount benefiting from a slight decrease as reducers increase.
  - **Combiner:**
    - Increasing mapping times with more reducers, indicating that the combination phase introduces complexity that negatively impacts performance.
    - Marked increase in reduction times, especially for LetterCount, showing that the overhead in data combination grows with more reducers.

- **Garbage Collection and CPU Time:**
  - **InMapper:**
    - Maintains low and stable garbage collection times, suggesting effective resource control even with more reducers.
    - Low CPU times, demonstrating efficient use of CPU resources without significant overheads.
  - **Combiner:**
    - Shows pronounced increases in garbage collection and CPU times. LetterFrequency shows a significant peak with 5 reducers, then stabilizes, while LetterCount remains consistently high.
    - Indicates that the combination phase introduces additional load on resource management, becoming less efficient with more reducers.

- **Memory Usage:**
  - Both methods show a linear increase in memory usage with more reducers.
  - Despite differences in execution and resource management, memory usage increases similarly for InMapper and Combiner.
  - InMapper demonstrates overall more efficient management.

In summary, InMapper proves to be the more efficient method for handling large datasets, maintaining low overhead and optimizing workload distribution

with an increasing number of reducers. Combiner, while useful for data combination, presents significant overhead that becomes more problematic with more reducers, requiring careful configuration to optimize performance. The choice between InMapper and Combiner should be guided by the specific needs of the application and the size of the files to be processed, with InMapper offering a clear advantage in terms of overall efficiency and workload management at scale.