Exact, Approximate and Space-Saving Counting Approaches for Identifying Letter Frequency in Text

Tiago Carvalho 104142

***Resumo* - Este estudo explora a eficácia de diferentes algoritmos para analisar a frequência de letras em arquivos de texto: um contador exato, um contador aproximado e um contador de economia de espaço. O objetivo principal é avaliar a precisão e a eficiência computacional desses algoritmos em várias obras literárias em diferentes idiomas. Neste relatório, será detalhada a implementação dos algoritmos. Uma análise detalhada dos resultados será realizada, destacando os compromissos entre precisão e eficiência, e fornecendo insights para examinar os resultados produzidos. Os resultados também serão comparados entre todas as abordagens descritas.**

***Abstract* – This study explores the efficacy of different algorithms for analyzing the frequency of letters in text files: an exact counter, an approximate counter, and a space-saving counter. The primary objective is to assess the accuracy and computational efficiency of these algorithms across multiple literary works in different languages. In this report, it will be detailed the implementation of the algorithms. A detailed analysis of the findings, highlighting the trade-offs between accuracy and efficiency, and providing insights will be conducted in order to examine the results produced. The results will also be compared between all the described approaches.**

I. Introduction

The field of text analysis often requires efficient and accurate methods for quantifying elements within large volumes of text. One fundamental aspect of this analysis is the frequency of characters, which can provide valuable insights in various applications, from linguistic research to cryptographic studies. By delving into the realm of letter frequency analysis in literary works, the project aims to explore and compare different methodologies in terms of their accuracy and computational efficiency.

This investigation encompasses three different methods that can be used as possible approaches to effectively obtain the frequency of letter in text files.

The exact counter method serves as the benchmark for accuracy, providing precise counts of each letter's occurrences in the text. The approximate counter method, utilizing a probabilistic approach, estimates letter frequencies with a fixed probability (1/8 in this particular case). This approach is often favoured in scenarios demanding computational speed over exact precision. The space-saving counter algorithm is designed for efficiency in handling large-scale data streams. It is examined its performance with different parameter settings to assess its accuracy and suitability for real-time data processing.

All the mentioned approaches in study, as well as the code blocks to produce the plots in this report, were developed in Python3.11.

II. Study Environment

The original content that was used to obtain all the results regarding the algorithms was selected carefully in order to ensure a fair and thorough comparison. The study involves five textbooks from Project Gutenberg, regarding the theme "New Testament of the Bible". This type of content was selected because another important point in study is the variety of present languages, and the Bible is possibly the most translated book in the world.

Each text is written in a different language: English, Greek, Swedish, Spanish, and Portuguese. However, the obtained files were not a translation of the same original file, so the actual content inside each file varies between them.

A. Pre-processing the content.

After obtaining the content files in the different languages, a certain pre-processing was made in the text, so that the structure of the files and the extra written text content weren't influencing aspects.

This pre-processing includes removing headers and footers, eliminating "*stopwords"* and punctuation, normalizing the text, and converting all letters to uppercase.

The "*stopwords"* considered to be discarded were obtained via Python's "NLTK", "Natural Language Toolkit" library. The library was chosen due to its variety of supported languages and how easy it is access them. After removing the headers and the unwanted words and punctuation, all the digits were

removed from the text and remaining letters were converted to uppercase. Last but not least, considering the variety of languages in study, a text normalization using Python's "unicodedata" library was made.

B. Storing the content.

At the end, the text contents were saved to all the respective files inside the "processed" folder and were ready to be used by the algorithms.

III. Exact Counter

The first module in this study is the exact counter. As the name suggests, upon inserting the pre-processed content files, it will return the exact letter counting, i.e. the correct answer.

The exact counter module is designed to accurately count and record the frequency of each letter in a given text file. This module is particularly useful in situations where precise character frequency data is required, such as in linguistic analysis or data encryption.

Utilizing Counter from Python's "collections", the function counts the frequency of each alphabetic character in the text. The letter counts are then saved to a new file. This file is named after the original file and stored in a directory specifically for these exact counts.

The module can also be run manually with command-line arguments to specify a single file or process all files within the "processed" directory.

The following pseudocode represents most of the algorithm's logic and functionalities:

```
FUNCTION count_letters_and_save(file_path)
    OPEN content_file at file_path
    READ text from content_file
    INITIALIZE letter_counts using Counter
    COUNT each letter occurrence in text

    SORT letter_counts in descending order
    WRITE sorted letter_counts to file

    RETURN sorted letter_counts
```

IV. Approximate Counter

The Approximate Counter is an algorithm designed to estimate the frequency of each letter in a text file. Unlike the exact counter, it employs a probabilistic approach, offering a trade-off between computational efficiency and precision.

This approach involves a little bit more logic than the exact counter since it's no longer working with an exact science. In order to avoid counting all the letters, this algorithm uses approximation, i.e. a random number generator to decide whether to count the letter or not, based on a fixed probability. In this implementation, the probability is set to 1/8 by default.

After the initial probabilistic counting, the algorithm multiplies the counts by the reciprocal of the probability, in this case 8 for a probability of 1/8, to estimate the total counts of each letter. The estimated counts are sorted and saved to a file within a designated directory. Each letter's estimated count is written to this file and later can be compared with the expected exact value calculated previously by the exact counter.

This approach also uses the Counter from Python's "collections", possibly increasing each time a new letter is iteracted.

Here's how the algorithm is represented in pseudocode:

```
FUNCTION approx_count(file_path, prob=1/8)
    OPEN content_file at file_path
    READ text from content_file
    INITIALIZE counts using Counter

    FOR each letter in text:
        DETERMINE random probability
        INCREMENT letter count counts


    MULTIPLY each count by the reciprocal of
the probability

    SORT counts in descending order
    WRITE sorted counts to file

    RETURN sorted counts
```

V. Space-Saving Counter

The Space-Saving Counter is an algorithm developed to efficiently identify the most frequent items (in this case, letters) in a data stream. It is particularly valuable in scenarios where data is voluminous or arrives in a continuous stream, and there is a need to maintain a running tally of the most common elements.

This algorithm is particularly useful when a restrict quantity of memory is available for counting large volumes of letter. In this case, a limited number of letters is being counted at that time and new letters won't mean, the counter needs to allocate more memory for the counting.

As each letter is read from the file, the algorithm updates the counts in the following manner:

- If the letter is already being tracked in the counter, its count is incremented.
- If the limit number of letters being counter hasn't been reached, the new letter is added to the counter.
- If the counter is already tracking the limit number of letters, the least frequent letter is replaced with the new letter, and its count is incremented.

In this case, no more that the predefined number of letters will be counted, i.e. if the user determines that the result should quantity a total of ten letter and the text has eleven or more, even though the algorithm will iterate all the letters, not all of them will be included in the final result.

After finishing the counting, once again, the estimated counts of the most frequent letters are sorted and saved to a file. The counts are sorted in descending order of frequency.

For this algorithm, the pseudocode would be presented like this:

```
FUNCTION spacesaving_count(file_path, k=10)
    OPEN content_file at file_path
    READ text from content_file

    FOR each letter in text:
        IF letter is in counters
           INCREMENT counter for letter
        ELSE IF counters limit isn't reached
           ADD letter to counters
        ELSE
           FIND letter with minimum count
           REPLACE it with the new letter,
incrementing the count

    SORT counters in descending order
    WRITE sorted counters to file

    RETURN sorted counters
```

VI. Experiment Result

As part of this report a detailed experiment was conducted in order to verify the efficiency and accuracy of the mentioned algorithms.

In this experiment, the language was considered a major factor because, even though the algorithms are able to process most language contents, in this particular case, some content file had a larger volume of content which could influence the observations and the discussion of the results. To try to work around this situation, the average absolute and relative errors, i.e. the difference between the exact and estimated values for each letter, were also registered.

Additionally, it was registered the allocated memory during the testing.

In the following plots and tables, it's possible to visualize the results obtain from the described experiment. The plots were produced using the stored results in saved files and the use of Python's "pandas" and "matplotlib" libraries.

A. English.

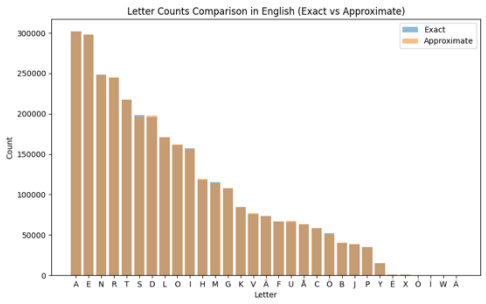

Fig. 1 – English Exact vs Approximate Comparison.

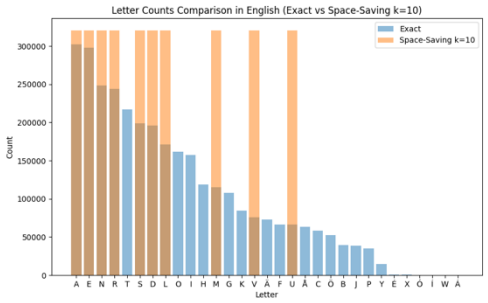

Fig. 2 – English Exact vs Space-Saving Comparison.

	EXACT		APPROX		SS
E	92923	E	93760	E	92923
T	68817	T	67760	O	70863
A	58231	A	58672	A	70826
O	57281	H	57672	M	70825
H	56869	O	57328	T	70824
N	50602	N	50920	I	70823
I	48154	I	49120	H	70823
S	46413	S	46152	W	70823
R	38194	R	38464	U	70822
D	33947	D	33360	L	70822

Table 1 – Top 10 most frequent English letters for the three algorithmic approaches

B. Spanish.

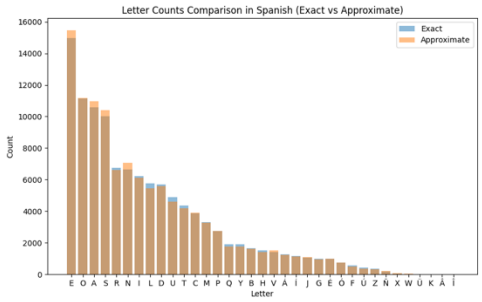

Fig. 3 – Spanish Exact vs Approximate Comparison.

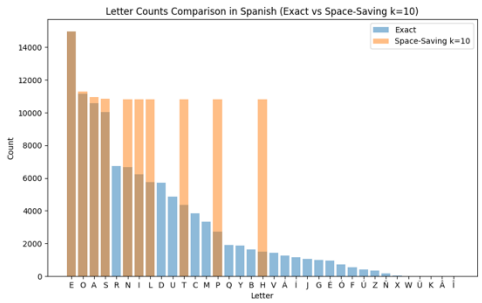

Fig. 4 – Spanish Exact vs Space-Saving Comparison.

	EXACT		APPROX		SS
E	14968	E	15464	E	14969
O	11144	O	11184	O	11301
A	10581	A	10976	A	10957
S	10023	S	10416	S	10863
R	6751	N	7064	T	10829
N	6656	R	6600	L	10827
I	6216	I	6112	I	10826
L	5759	D	5576	H	10826
D	5707	L	5456	N	10826
U	4872	U	4592	P	10826

Table 2 – Top 10 most frequent Spanish letters for the three algorithmic approaches

C. Greek.

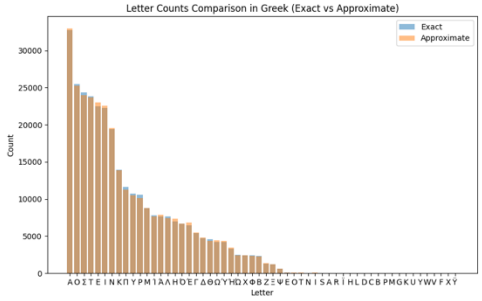

Fig. 5 – Greek Exact vs Approximate Comparison.

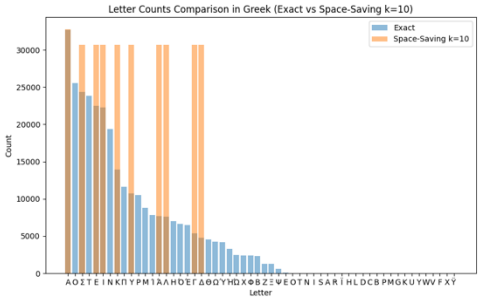

Fig. 6 – Greek Exact vs Space-Saving Comparison.

	EXACT		APPROX		SS
Α	32730	Α	32976	Α	32768
Ο	25513	Ο	25240	Ι	30660
Σ	24362	Σ	23960	Ε	30658
Τ	23805	Τ	23696	Γ	30657
Ε	22468	Ε	23040	Σ	30656
Ι	22257	Ι	22536	Υ	30656
Ν	19381	Ν	19536	Λ	30656
Κ	13946	Κ	13880	Κ	30656
Π	11612	Π	11264	Ά	30656
Υ	10691	Υ	10504	Δ	30655

Table 3 – Top 10 most frequent Greek letters for the three algorithmic approaches

D. Portuguese.

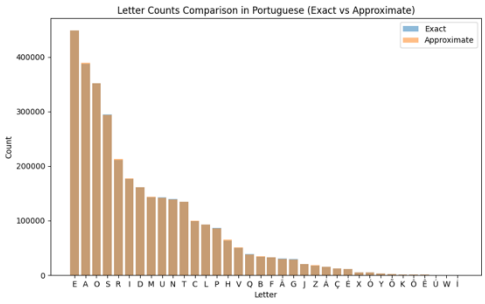

Fig. 7 – Portuguese Exact vs Approximate Comparison.

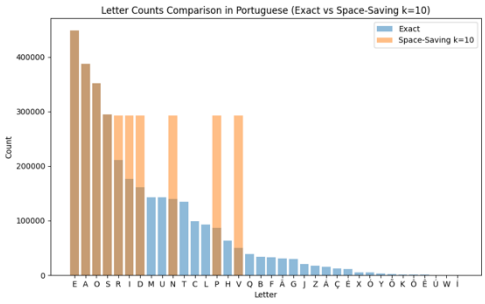

Fig. 8 – Portuguese Exact vs Space-Saving Comparison.

EXACT		APPROX		SS	
E	448758	E	448672	E	448758
A	387951	A	389896	A	387954
O	352248	O	351816	O	352268
S	294256	S	293952	S	295075
R	211466	R	212784	R	293146
I	176116	I	177680	D	293145
D	161362	D	161512	V	293145
M	143166	M	143840	N	293144
U	142739	U	141488	I	293144
N	139464	N	139056	P	293143

Table 4 – Top 10 most frequent Portuguese letters for the three algorithmic approaches

E. Swedish.

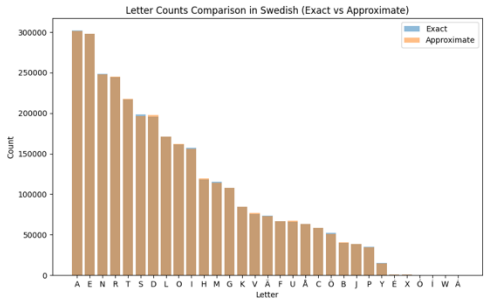

Fig. 9 – Swedish Exact vs Approximate Comparison.

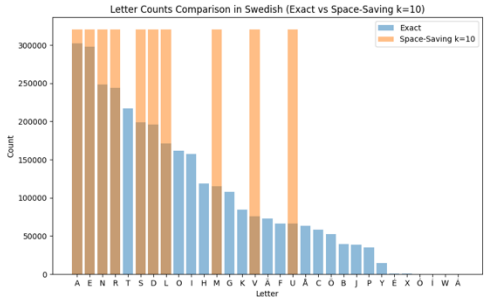

Fig. 10 – Swedish Exact vs Space-Saving Comparison.

EXACT		APPROX		SS	
A	302151	A	301488	E	320426
E	298021	E	297744	A	320423
N	248409	N	248136	R	320423
R	244216	R	244848	L	320421
T	216714	T	217648	M	320420
S	198562	D	197680	N	320420
D	195777	S	196528	S	320420
L	170920	L	171264	D	320420
O	161418	O	161784	U	320419
I	157079	I	155896	V	320419

Table 5 – Top 10 most frequent Swedish letters for the three algorithmic approaches

F. Absolute and Relative Errors.

As mentioned above the average absolute and relative error have been registered. To obtain more accurate results, the approximate approach was used fifty times, so that it was possible to obtain some sort of average value for the error.

Since, the absolute error doesn't provide any conclusions to this study, because the content files do not contain the same volume of text, the following plots are only related to the relative error.

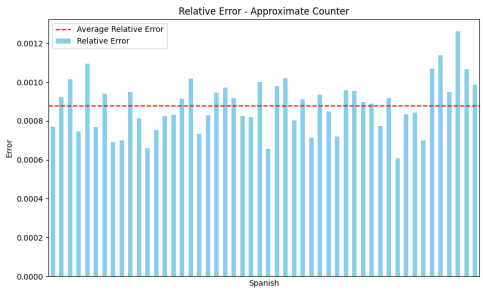

Fig. 11 – Spanish Approximate Counter Relative Error.

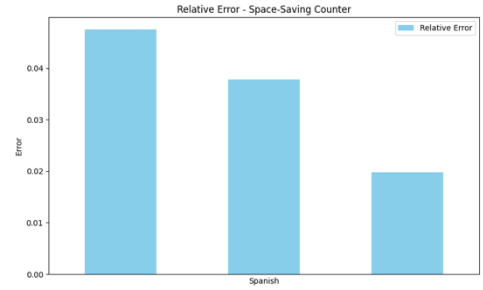

Fig. 12 – Spanish Space-Saving Counter Relative Error.

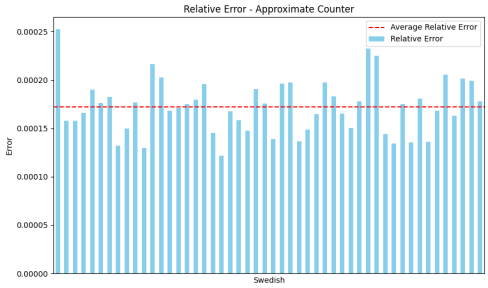

Fig. 13 – Swedish Approximate Counter Relative Error.

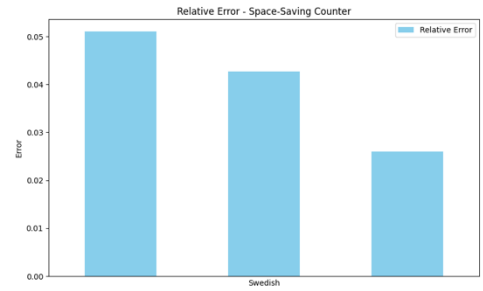

Fig. 14 – Swedish Space-Saving Counter Relative Error.

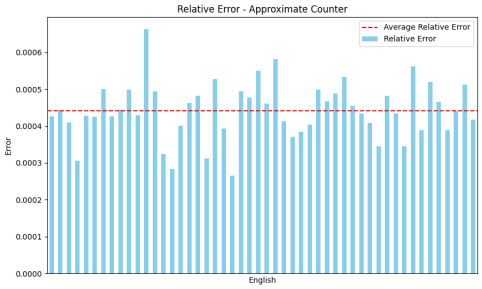

Fig. 15 – English Approximate Counter Relative Error.

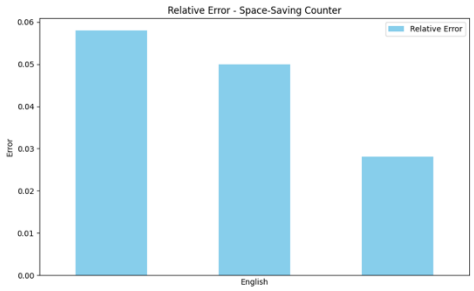

Fig. 16 – English Space-Saving Counter Relative Error.

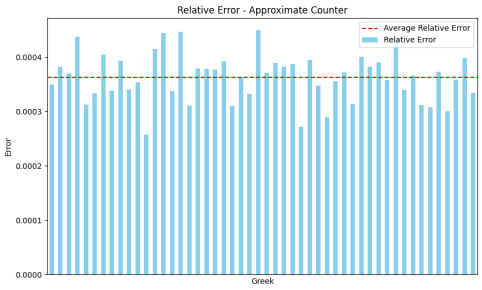

Fig. 17 – Greek Approximate Counter Relative Error.

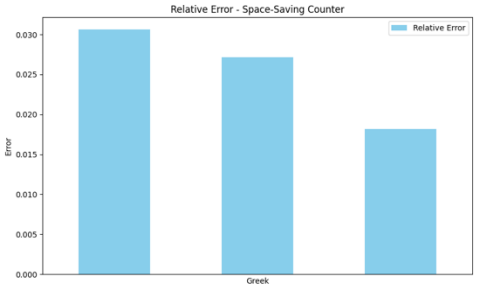

Fig. 18 – Greek Space-Saving Counter Relative Error.

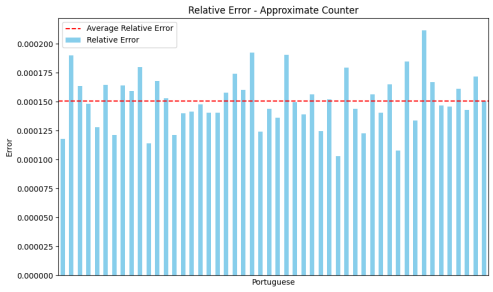

Fig. 19 – Portuguese Approximate Counter Relative Error.

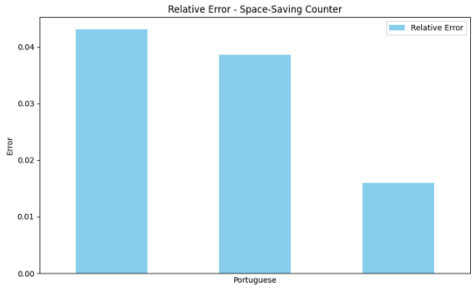

Fig. 20 – Portuguese Space-Saving Counter Relative Error.

F. Execution Time and Allocated Memory.

During the experiment, the time and memory was measured and, as result, those values were saved in a new *json* file.

The registered values were the following:

	VALUE
TIME	99.902896 seconds
MEMORY PEAK	13.92887 MB

Table 6 – Performance Metrics Registered.

VII. Discussion

A. Efficiency and Accuracy.

The Exact Counter algorithm offers the highest level of accuracy as it precisely counts the frequency of each letter in a text file, ensuring no data is missed or inaccurately represented. However, this method can be less efficient in terms of computational resources, particularly with very large datasets or in real-time processing scenarios, as it requires processing every single character in the text.

The Approximate Counter, using a fixed probability approach, trades off a degree of accuracy for improved computational efficiency.

In the results it's visible that, for the volume of content text inputted, the accuracy of the algorithm was not bad at all. The relative error values registered were very low for each language. This shows that this algorithm provides some reliability for studies similar to this one.

The Space-Saving Counter, by maintaining a fixed-size list of the most frequent items, offers a balance between accuracy and efficiency, particularly for identifying the top ten frequent elements.

In the results, it is clear that the accuracy of this approach is not the as low as the approximate counter, at least for this volume of data. However, upon closer look to the first letter registered by the

algorithm, its visible that the error is not as high as for the letter with lower counting's.

This proves that, in order to obtain greater accuracy for the letter with the greatest frequencies, it's necessary to increase the number of counters. This is visible in the relative error plots for this approach. Increasing the value of *k*, decreases the value of the registered error. However, increasing this value of *k* only makes it so that this approach becomes more like an exact counter.

B. Languages in study.

One important aspect visible in the results is that, even though this study incorporates five different languages, the results are not language-specific, i.e. the language doesn't influence the final result.

However, in the space-saving approach, the language can affect the results to some extent. Languages that have more letters or characters require the space-saving algorithm to use more counters in order to obtain a determined accuracy. Greek is a great example. As its visible, not all of the most frequent letters were registered by the algorithm. This happens because of the diversity of Greek's alphabet, having more characters than the English's alphabet, for example.

VIII. Conclusion

In conclusion, this study demonstrates the practicality and effectiveness of three letter frequency counting algorithms in processing literary works across different languages.

The Exact Counter offers unparalleled accuracy, making it ideal for scenarios where precise character counts are crucial.

The Approximate Counter, with its probabilistic approach, presents a viable alternative for situations requiring quicker processing, achieving a reasonable balance between accuracy and efficiency.

The Space-Saving Counter emerges as an effective tool for identifying the most frequent letters, particularly in data-intensive environments, although its accuracy depends on the number of counters.

Across different languages, these algorithms exhibit consistent performance, highlighting their adaptability and utility in diverse linguistic contexts. The insights gained from this study contribute to the broader understanding of text analysis methodologies and their application in various fields of research.

IX. References

[1] https://chat.openai.com/

[2] https://inventwithpython.com/hacking/chapter20.html

[3] https://medium.com/@Nougat-Waffle/caesar-cipher-and-frequency-analysis-with-python-635b04e0186f

[4] Cai, Q., & Brysbaert, M. (2010). SUBTLEX-CH: Chinese Word and Character Frequencies Based on Film Subtitles. PLoS ONE, 5(6), e10729