Software Design Project 2: A Zipfian Analysis of Poetry

By: **Samuel Cabrera Valencia**

In [1]:
```
from functions import scrape_poet
from functions import plot_zipf_poet
from functions import plot_zipf_poet_all
from functions import plot_zipf_top500
from functions import plot_zipf_top500_all
from functions import all_english_zipf
from functions import all_english_zipf_all
%load_ext autoreload
%autoreload
```

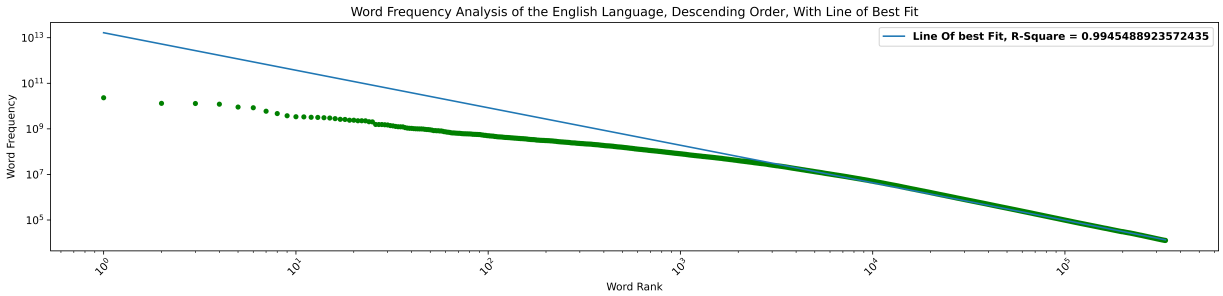
Overfiew: What is Zipf? Why Poetry?

In the mid 20th Century Linguist George Kingsley Zipf (along with a lot of other mathematicians, statisticians, and linguists) noticed that the frequency of words used in almost all languages follow a logarithmic pattern such that the second most used word occurs approximately half as often as the most used word, the third used a third as often as the first, the fourth a fourth as often and so on such that approximately:

$$f(r) \propto \frac{1}{r^{\alpha}}$$

This function calculates the frequency of a word appearance, where **r** is the rank of the word, and α is a constant to better fit the law(approx. = 1)[1]. As an example, here is a plot of the word frequency of all words that appear on the Google Web Trillion Word Corpus[2] (which serves as a good analog for the entire English language as a whole:

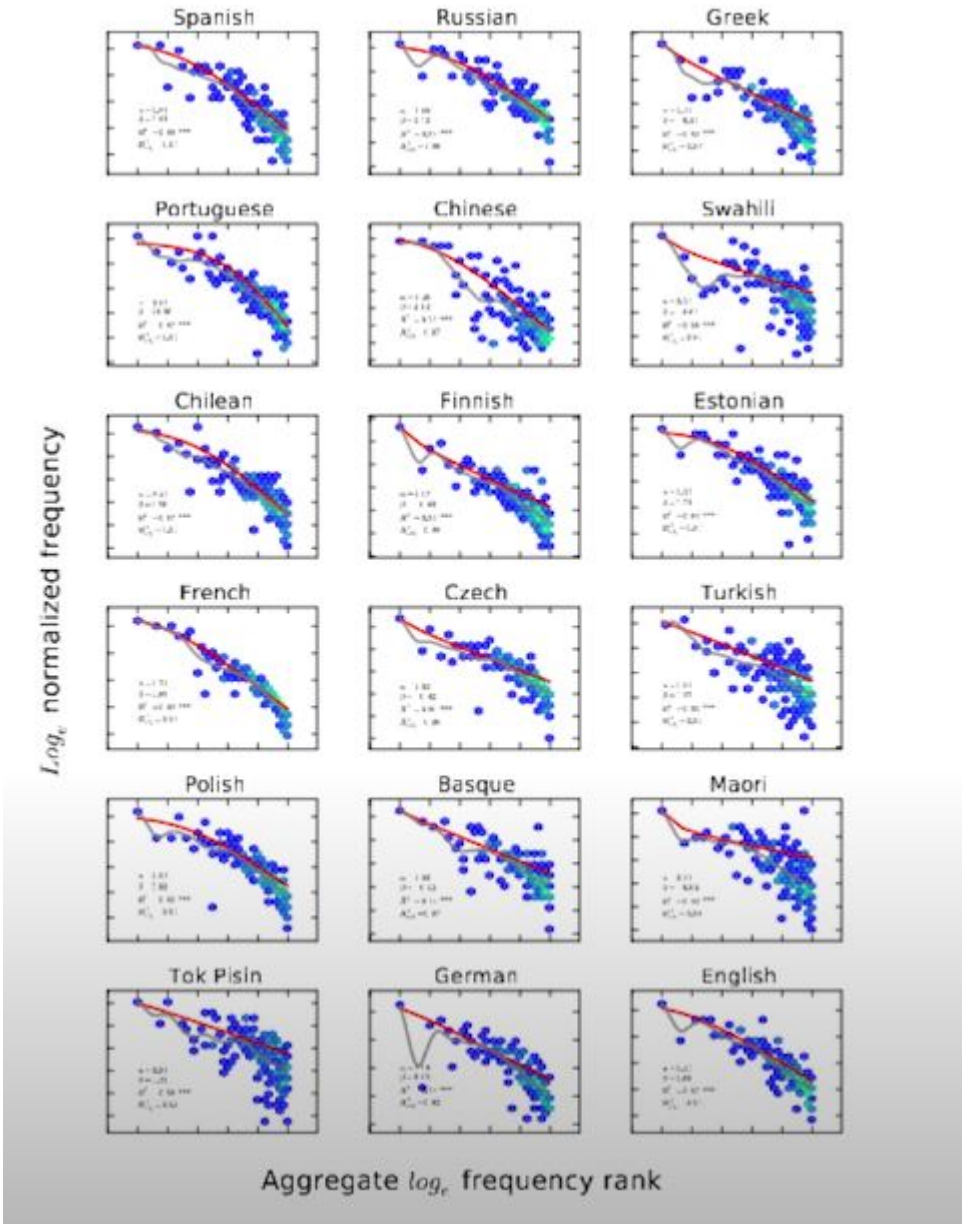
In [2]:
```
all_english_zipf_all()
```



These sort of loglog graphs tend to curve fit to a line best when trying to capture a power law in this domain, because of this the R^2 value is used to describe how well this forumula desribes the behavior of the formula. Having an R^2 so close to **1** means that a powerlaw describes the

relationship between frequency and rank in the english language very well (Despite what it looks like on the graph, there are magnitues more words on the less used side, meaning that this straight line fits the data much better than might be instantly evident).

What makes Zipf's law truly remarkable is that it not only fits for the english language, but for words in books, and even for pretty much all sets of words in all other languages:



A really amazing refference for how ubiquitous zipf's law, and other power laws are is this Vsauce Video (https://www.youtube.com/watch?v=fCn8zs912OE&t=194s/)[3] (which also ultimately served as the inspriation for this porject) is a very good refference.

So what about poetry?

We saw that this law not only describes languages, but also books, and other general collections of words. The question that this report attempts to answer is: **Is poetry zipfy (does it behave following a power-law)?** and **Are there writers who are more or less zipfian than eachother?**

There are many reasons that one can come up with for reasons that poetry might not follow a zipfian pattern. First, poetry is a lot more inentional and a lot of times, more particular with the language it uses than speech and even most prose writing. Second, since poetry has such a diversity of structures, different rules, and overall different word placement than regular language, this might make it work less as a power law.

Wether or nor poetry follows zipf, can possibly used by experts to draw observations or conclusions on general patterns in rule in poetry or be studied as to why it breaks the pattern.

Background: Libraries; Scraping, Mathematical Tools, and Plots

To facilitate working with the big datasets we were going to have to obtain, multiple libraries work as the backbone of the scraping and analysis code:

Scraping:

`requests` [4] the usual python requests library for interacting with websites

`beautiful soup` [5] for easier interfacing with html to scrape links and text off the website

Tools:

`numpy` [6] provides a lot of the mathematical expressions and some of the curve fit tools

`Scipy` [7] helps with more difficult curve fits

`Language Tool Python` [8] Helps with correcting some of the miss-formatted text after being scraped

`Collections, Counter` [9] makes counting word frequency on a list a lot easier

`String` [10] Helps with cleaning up the scraped text with more exact measures

`math` [11] another library with useful mathematical functions

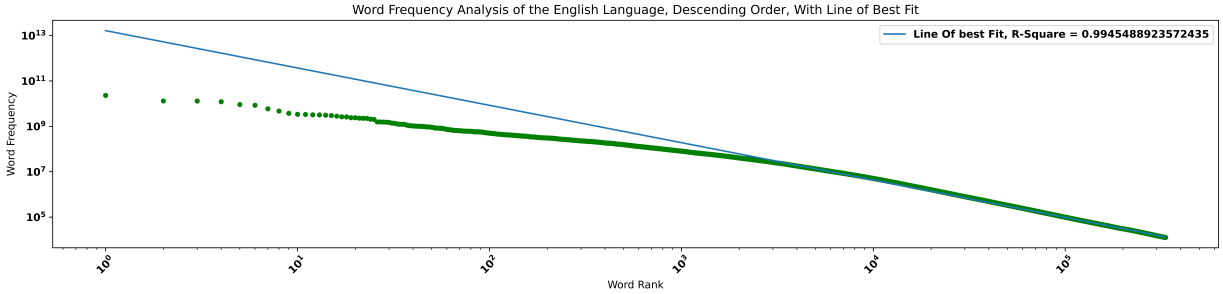
Plots:

`MatplotLib` [12] most powerful and versatile library for plotting in python

Background: Curve Fits and Types of Analysis

As mentioned before, a common way to study how a set of data behaves, or what pattern it follows, is to fit it to a mathematical function. We will look at the collected data in two different perspectives, the whole dataset of word frequency over word rank on a loglog plot:

In [3]:
```
all_english_zipf_all()
```



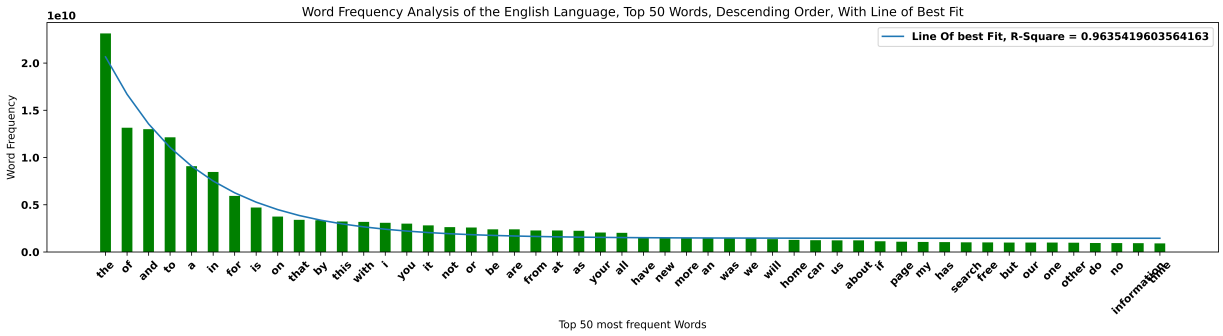
Which serves to describe the whole dataset as zipfian, using a curve fit of the form:

$$\log\big(f(r)\big) = e^m * r^k$$

Similar to before, **r** is word rank *F*, **m** and **k** are both arbitrary curve fit variables. is frequency, and the big idea of using a curve fit like this is that we get to fit the equivalent of a straight line on to the dataset. With this we can qualitatively observe how closely the data fits the line, and quantitatively we can use the tool of R^2, which quantifies how "different" the fit line is to the actual data, telling us how closely it describes it.

The second way we are going to look at the data is a closeup of the top 50 words in a language, for the sake of thoroughness, curiosity, and to observe the different behavior of the set of top words over their frequency:

In [4]:
```
all_english_zipf()
```



Since we are looking at these plots on a semi-log (or for many of the datasets, just regular scale) the curve fit we use must be a bit different, and to mitigate the slight deviation that occurs with the top 50 words from the full dataset, we use a fit of the form:

$$f(r) = ae^{br} + c$$

where like before **r** and *F* represent rank and frequency and a, b, and c simply being arbitrary curve fit values. We will also derive and include the R^2 of these curves, which tells us how well the fit represents the data, note that this value will be different for the full dataset and the top 50 words. In all honesty both of these inclusions add little insight into the behavior of the full system, but are still useful to get some more detail.

Methodology: Collecting a ton of Poems and Crunching Numbers

Part 1: Poem Scraping

The first thing to do for answering the question of poetry working in a Zipfian manner was to collect the poems themselves. To this end, the easiest way of amassing with any sort of speed is through creating a script that goes to a database for poems and copies every poem on to a text file to then be analyzed.

The first piece of this puzzle was which database to use. Though there are several great websites for poetry, I wanted to choose a database that had a plainer layout which tends to make the programming overhead lower. I found poemhunter.com (poemhunter.com) which seemed both plain and had useful categorize likes poems organized by poets and a top 500 poems list.

Once this was chosen, I decided to run with the idea of analyzing the top 500 poems to get a general sense of how poetry word frequency behaves and then a further script where I could input the name of a poet and scrape all of their poems off of the website. To do this, `requests` and `beautiful soup` would be used to extract the data and then the `language tool python` would be used to correct any spacing errors that could occur with the scraping. All of these poems would be stored into an list of poems, which would then be written into a text file. The scraping scripts are `scrape_top500()` which scrapes, formats, and downloads a .txt file of the top 500 poems and `scrape_poet({poet-name})` which does the same operation as the former function but instead takes in a string of the poet name in the form of `name-lastname-...(if the author as multiple names )` and gets the .txt file of all of that poet's poems. It is important to mention that the way the code was written was with a tight "one hot function" philosophy. Instead of breaking up the scraping and storing steps into different sub-functions, I wrote it so each function is not dependent on any external functions (though the plotting functions are dependent on having the right dataset in your folder.

Part 2: Cleaning Data and Counting Words

The first thing we need to do with the saved data is to clean it up a bit before we can process it, since we want to look at it word-by-word, we load the scraped .txt file into a single string, then use the break it up into a word-by-word list using `.split()` . With this, we then remove any punctuation from the data and set it all to lowercase to make sure everything is being counted properly. Once this is done `Counter` from the `Collections` library is used, which simple counts the occurrence of each object in a list and creates a variable with the results.

Part 3: Pandas Data-Framing

Once the data was processed, it gets put into a `Pandas` Data frame which is useful for organizing the list in decreasing order and depending on the type of plot, dropping all values after the first 50. This sets up the raw data itself to be plotted, now on to:

Part 4: Curve Fitting and R^2

Depending on the type of plot, top 50 words or whole dataset, as described earlier we use different curve fit for each one, and both are implemented using a different library. For the top 50 functions, we used `Scipy` 's `optimize.curve_fit` which is a very powerful tool creating non-linear curve fits and can be fed in the curve fit function mentioned for the top 50 words fit. For the whole dataset, we instead used `Numpy` 's `polyfit` function which serves to produce the coefficients needed for the curvefit formula for this sort of graph. The R^2 for both functions are calculated with the formulas:

$$\overline{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

$$SS_{tot} = \sum_i (y_i - \overline{y})^2$$

$$SS_{res} = \sum_i (y_i - f_i)^2$$

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

Part 5: Plotting

All the plotting was done through `MatPlotLib` . To better visualize the data being presented, the top 50 words plots use a bar-graph. The whole dataset plots use a scatterplot to better understand the overall behavior of the spread.

There are a total of 7 plotting functions that take care of parts 2,3,4, and 5 of the methodology, these are :

- `plot_zipf_top500()` : Plots the top 50 most used words in the top 500 poems of poemhunter.com (poemhunter.com) with a curve-fit and R^2

- `plot_zipf_top500_all()` : Plots a loglog plot of word frequecy over word rank in the top 500 poems of [poemhunter.com] with a curvefit and R^2 (poemhunter.com)

- `plot_zipf_poet({poet-name})` : Takes in a poet name written with hypens between the names (i.e. `edgar-allen-poe`) and plots the top 50 must used words in a poet's catalog of poems with a curvefit and R^2

- `plot_zipf_poet_all({poet-name})` : Takes in a poet name written with hypens between the names (i.e. `edgar-allen-poe`) and plots loglog plot of word frequecy over word rank in a poet's catalog of poems with a curvefit and R^2

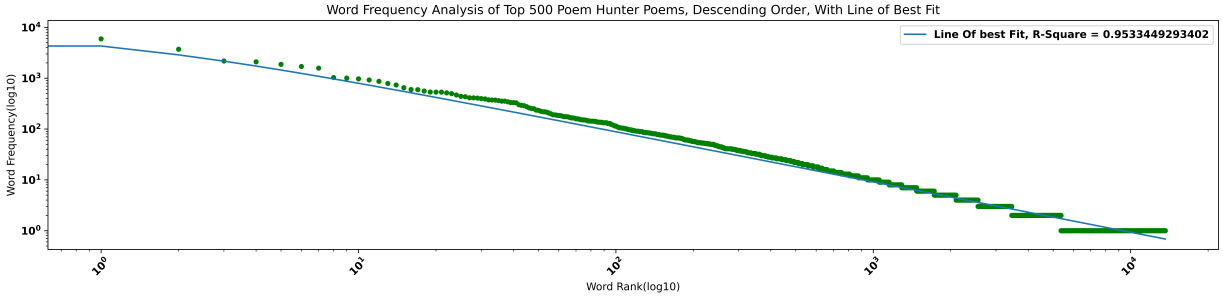
- `all_english_zipf()` :Plots the top 50 most used words in the Google Web Trillion Word Corpus as obtained from [2] with a curve-fit and R^2

- `all_english_zipf_all()` : Plots a loglog plot of word frequecy over word rank in the Google Web Trillion Word Corpus as obtained from [2] with a curve-fit and R^2

Results:

Is poetry Zipfian?

In [5]:
```
plot_zipf_top500_all()
```

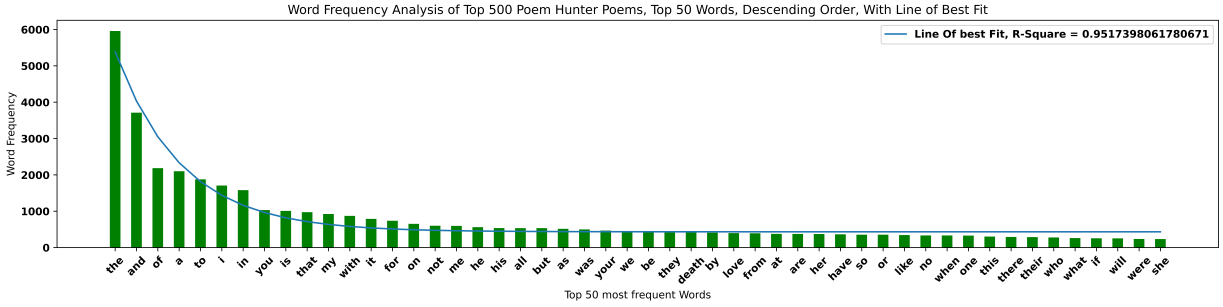


It very much is.

What this plot tells us, is that in an arbitrary selection of poems, the word frequency spread will presumably follow zifp's law. It's $R^2 = .95$ tells us that using a power law to describe the frequency of words maps very well to it's word rank.

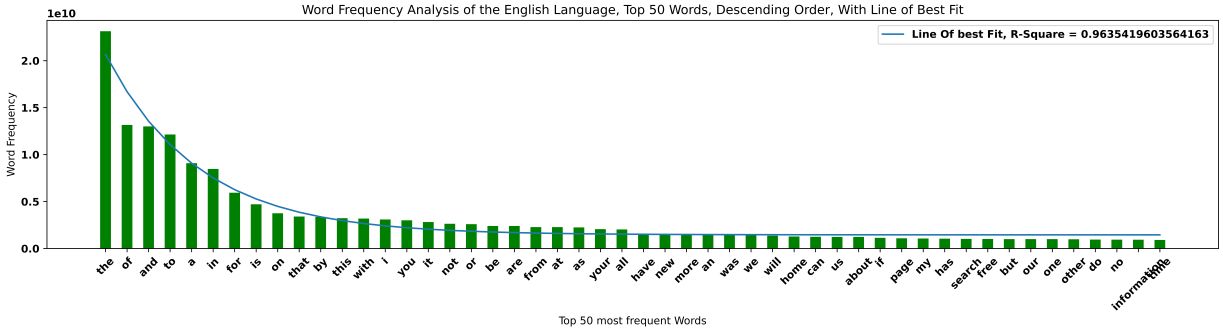
In a closer inspection we can look at the top 50 words:

In [22]:
```
catch = plot_zipf_top500()
```



The top 50 words alone map fairly well to an exponential decay function. It's important to notice that especially the top 10 most used words looks very similar to the entire english language's top 10 words:

In [7]:
```
all_english_zipf()
```



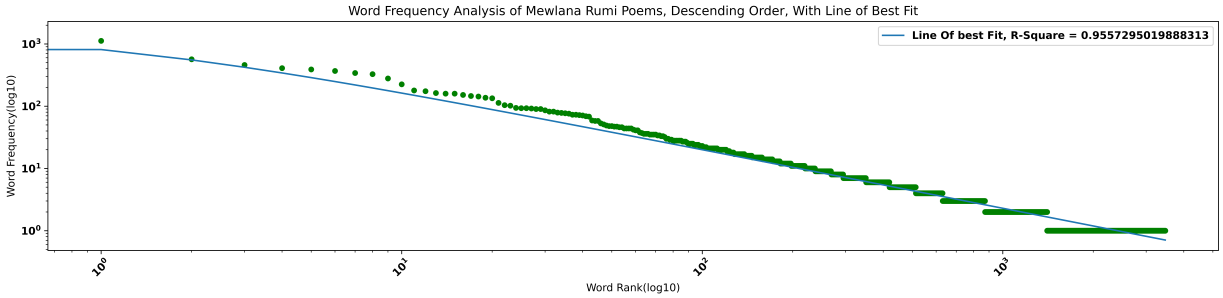
This tells us that even if poetry in general deviates in structure more than regular use, it does not deviate as much in the word choice for its "fundamental structure" of articles such as "a" and "the", in addition to conjunctions such as "and", prepositions such as "of" and "in", and verbs such as "is". It is valuable to notice though the deviations in language from regular English, showing off in the top 50 more personal words such as "my" and "me". Overall, though, poetry might use a different vocabulary, but it shares the overall Zipfian behavior of language.

Poet-By-Poet Analysis

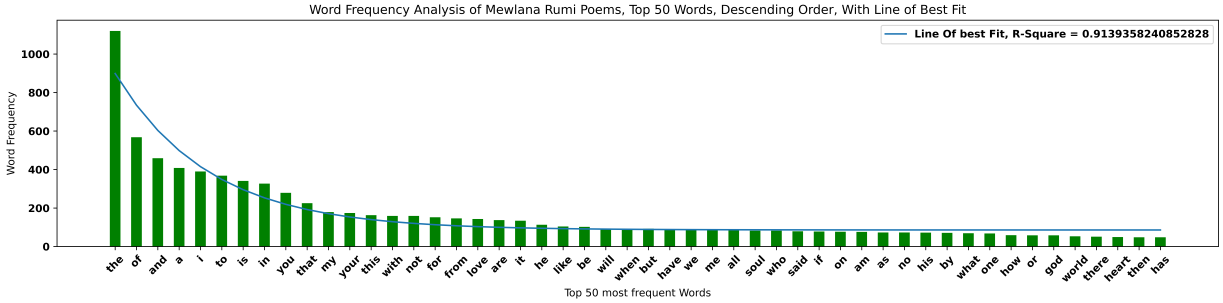
Now that we have determined that poetry in general follows Zipf's law, I felt it could be valueble to look at how well we can fit it to various poets. If a poet breaks Zipf's law, it could tell us that the poet maybe writes in a different way or has a more diverse and/or intentional word choice in their poems. Since there is a lot of possible poets to study I chose a combination of personal favorites and well-known/popular poets: (if there is a poet you don't see on the list, feel free to run the appropriate function and plotting functions to add it!)

Mewlana Jalaluddin Rumi:

In [8]:
```
plot_zipf_poet_all("mewlana-jalaluddin-rumi")
```

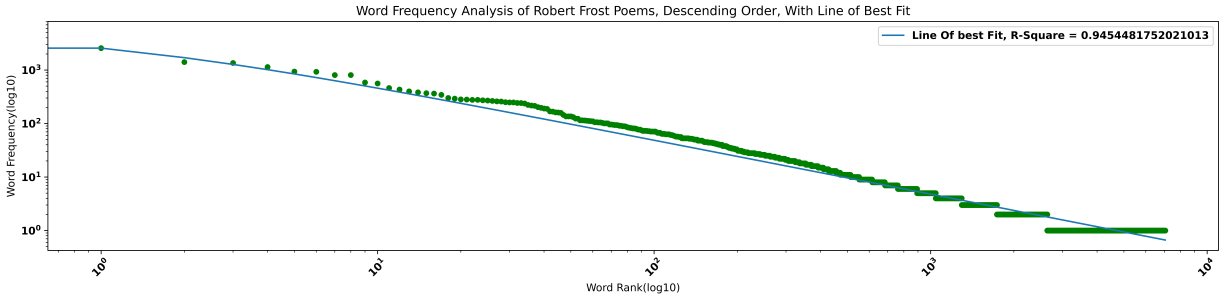


In [9]:
```
catch = plot_zipf_poet("mewlana-jalaluddin-rumi")
```

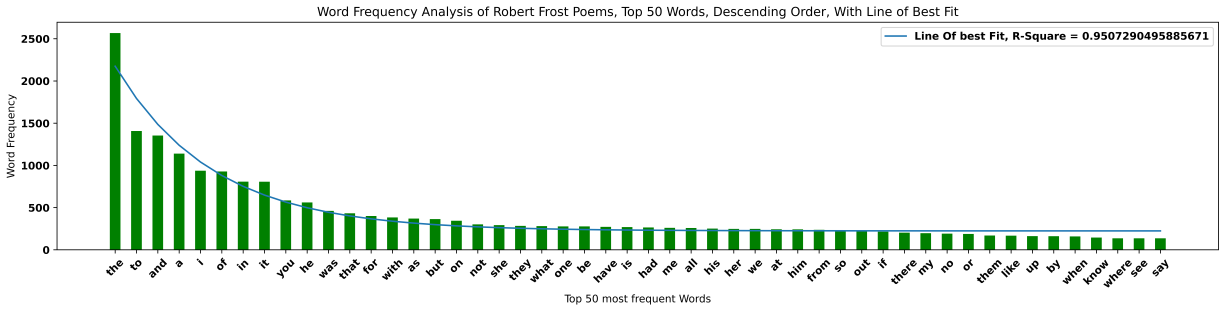


Robert Frost:

In [10]:
```
plot_zipf_poet_all("robert-frost")
```

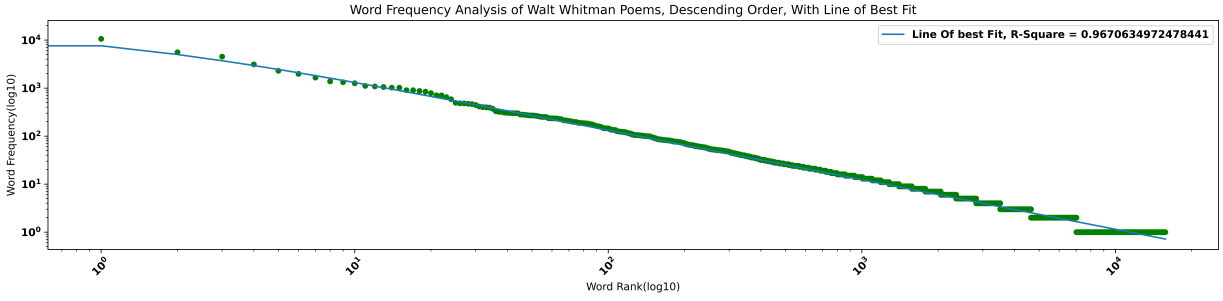


In [11]:
```
catch = plot_zipf_poet("robert-frost")
```

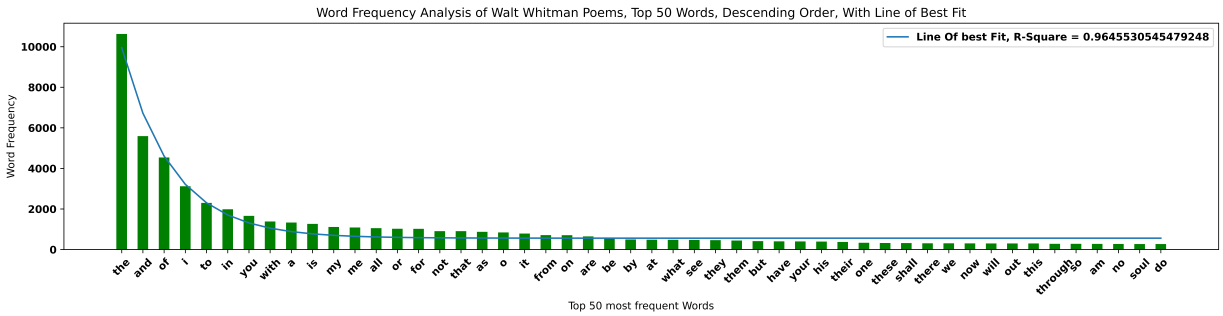


Walt Whitman

In [12]:
```
plot_zipf_poet_all("walt-whitman")
```

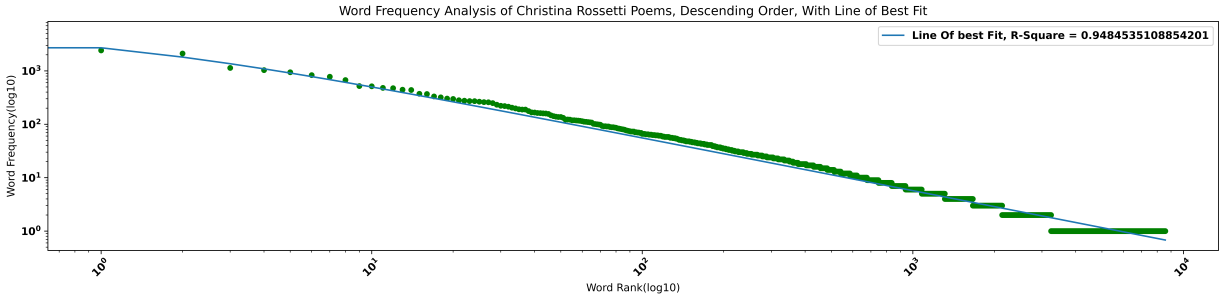


In [13]:
```
catch = plot_zipf_poet("walt-whitman")
```

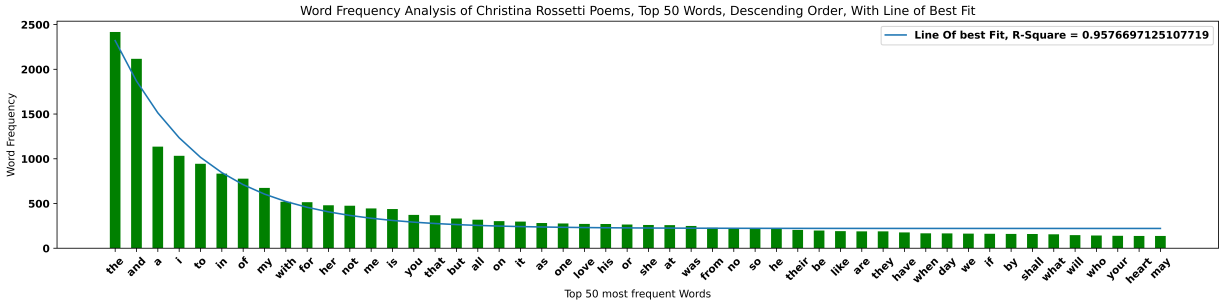


Christina Georgina Rossetti

In [14]:
```
plot_zipf_poet_all("christina-georgina-rossetti")
```

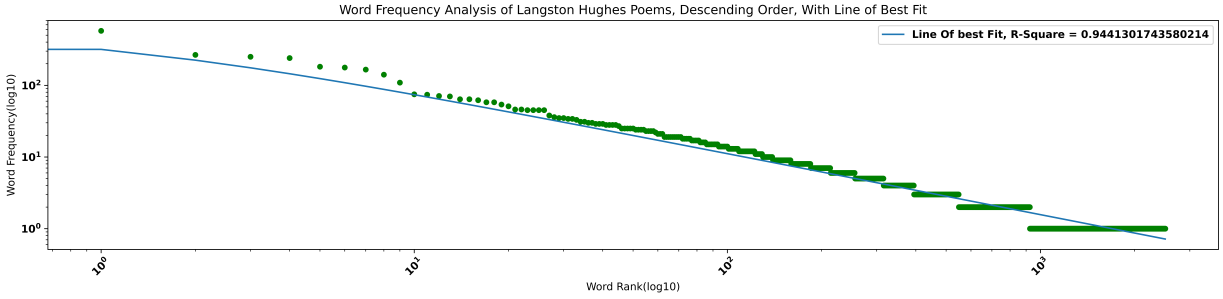


In [15]:
```
catch = plot_zipf_poet("christina-georgina-rossetti")
```

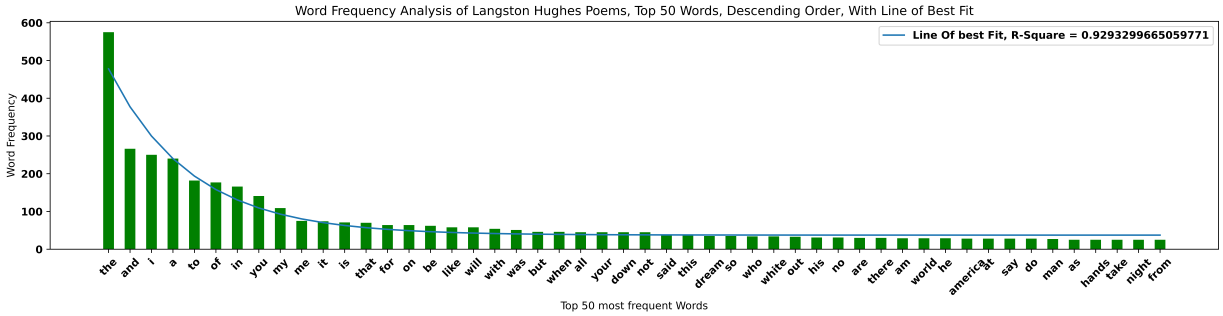


Langston Hughes

In [16]:
```
plot_zipf_poet_all("langston-hughes")
```

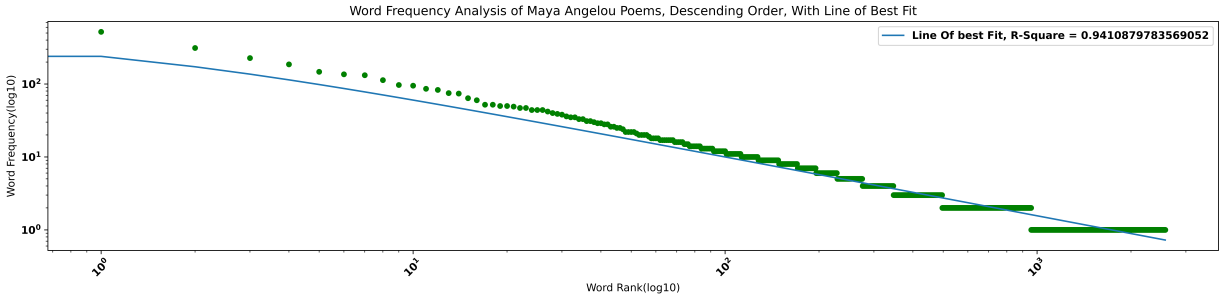


In [17]:
```
catch = plot_zipf_poet("langston-hughes")
```

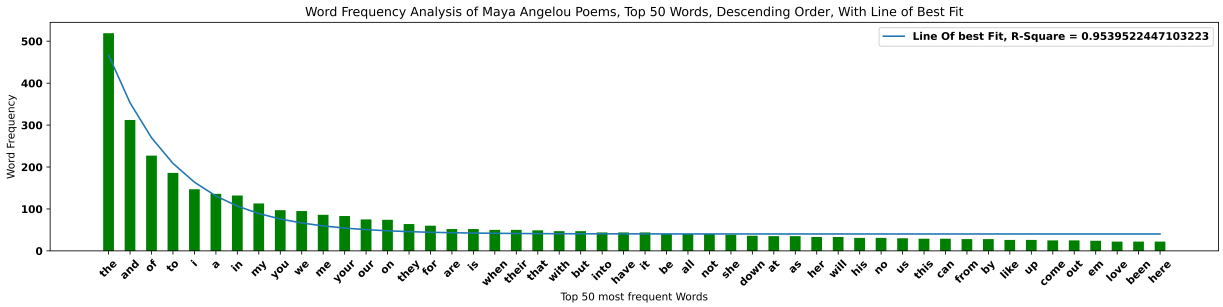


Maya Angelou

In [18]:
```
plot_zipf_poet_all("maya-angelou")
```

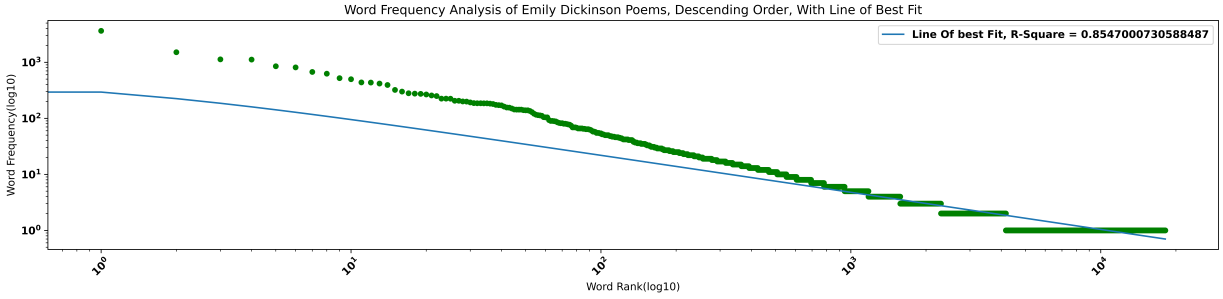


In [19]:
```
catch = plot_zipf_poet("maya-angelou")
```

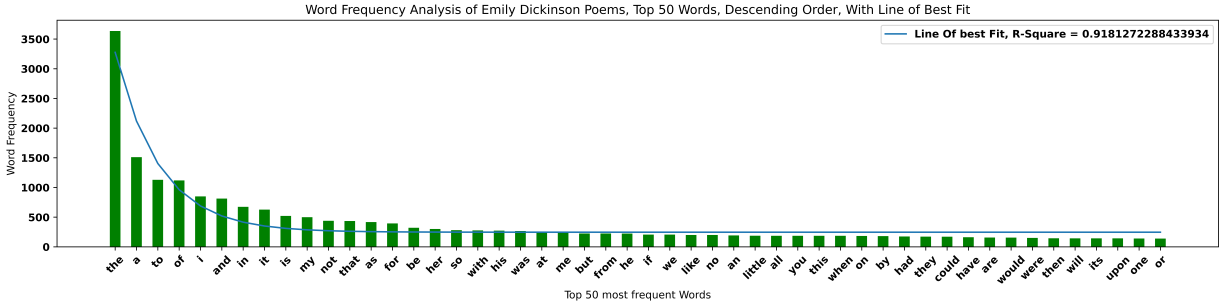


Emily Dickinson

In [20]:
```
plot_zipf_poet_all("emily-dickinson")
```



In [21]:
```
catch = plot_zipf_poet("emily-dickinson")
```



In this long list of poets, most of them look very similar in terms of their plots and for the most part, quantitatively speaking they are. Despite this there is a reason I listed Emily Dickinson last and that's because in this dataset she is by far the most interesting. This is because, and this is what I meant by "quantitatively speaking", the R^2 for all but Dickinson, are $> .9$ which means that a power law very well describes their distribution of words. For Emily Dickinson though, we have an $R^2 = .85$, which by no means says that a power law poorly describes word frequency in her work. Despite this, the deviation is big enough that we can say that it seems like Emily Dickinson uses words in a less regular way than her fellow poets. Overall, we see very little evidence for poetry being radically different than regular prose or language for it being any less beholding to Zipfs law.

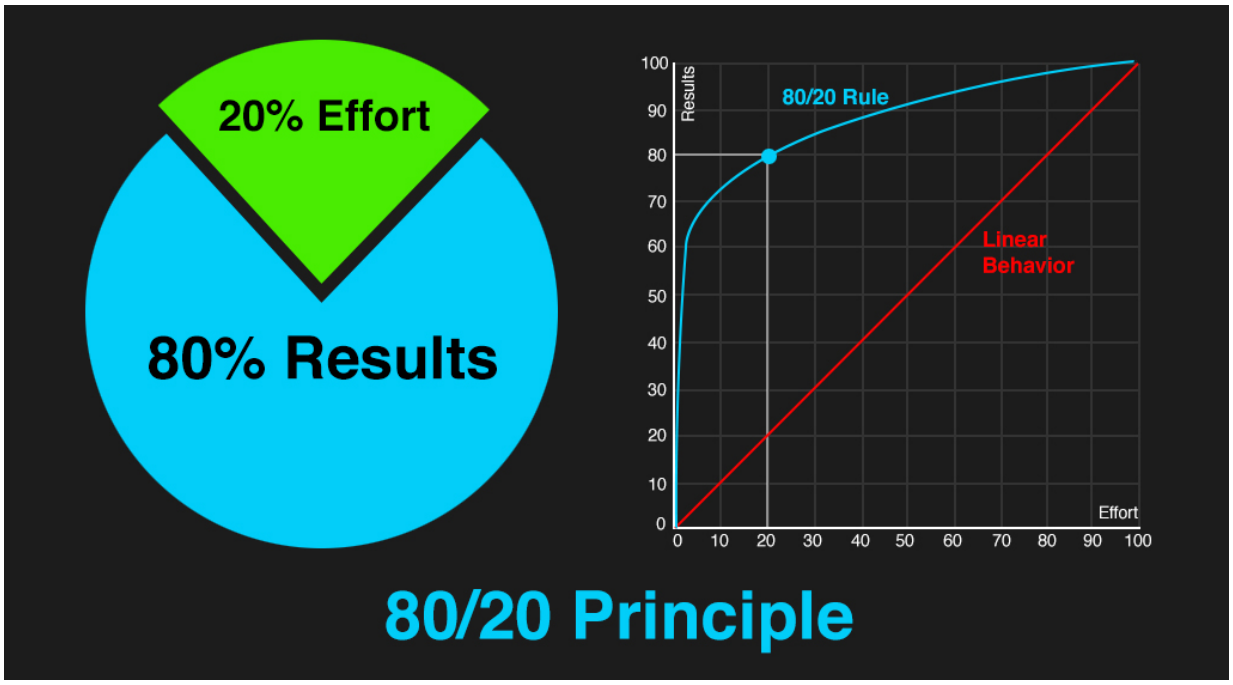
Conclusions: Why Does this happen? What Does it Mean? What Did I Learn?

Why is Poetry Zipfian:

There is not empirical explanation. One of the main appeals and hurdles of working with Zipfian word distributions is that the reason why language behaves in a Zipfian manner and that is behavior is ubiquitous with everything from speech, to prose, to names, to evidently poetry, and to a lot of other groups of words, is not really known. That is not to say there are not explanations, there is an abundance of these, I am going to present the one I find most convincing:

The Pareto Principle and Law of Least Effort

This explanation, which mainly references the previously referenced Vsauce Video (https://www.youtube.com/watch?v=fCn8zs912OE&t=194s)[3] and this paper 13 (https://sci-hub.se/https://www.tandfonline.com/doi/full/10.1080/00107510500052444?scroll=top&needAccess=false) by MEJ Newman. Both sources point to the "80/20 Principle" which is sometimes also called the "Pareto Principle" as at least a partial culprit of this common behavior. This Principle basically operates upon the concept that 20% of the causes cause 80% of the results, in the context of word frequency analysis, 20% of the words get 80% of the use.



[14]

This argument would use the Principle of Least effort, that things operate constantly choosing paths of least resistance, would mean that we use the smallest possible vocabulary to describe the most amounts of ideas. It would then make sense that in the process of writing poetry, even if word choice is intentional, a writer will inevitably use more common and well-known words than more obscure ones. Even if they use more obscure words than normal, the rate at which they use those words pales in comparison to their use of common words. For cases such as Emily Dickinson, we can make the claim that she has a more robust language than a lot of her peers, but even with that, she inevitably still fits a power law at least decently.

What Did I Learn?

In terms of whether or not poetry was Zipfian, I was completely expecting it to be. This is because of the insane ubiquity of Zipf's law, even if poetry is structurally and vocabulary different than most other language use, it does not deviate as much as one would think. It was still valuable to go through this process of analyzing poetry in general and artists in specific and I definitely see how this sort of analysis could be useful to a linguist attempting to study how poets write different from each other.

I also learned a ton about python libraries I'd never really touched, like `Beautiful Soup` (which meant learning some html stuff too), `SciPy` , and getting more practice in installing and using libraries one finds online like `Language Tool Python` . In addition, I know feel a lot more comfortable with `requests` and the whole concept of file reading/writing. It was also fun to think through the best curve-fit functions and think about philosophical questions on determinism and language.

Challenges

The hardest thing about this project was familiarizing myself with `Beautiful Soup` , getting `Language Tool Python` to work, formatting the text correctly for counting and getting the curve-fits to be properly representative of the dataset and it's behavior. Also figuring out the R^2 formula

for loglog plots took a fair amount of time.

Overall, most of the hardest parts of this project resulted in google/stack overflow rabbit-holes and were a bit tedious to solve in the short-term but somewhat straightforward once figured out. All these challenges resulted in a greater number of things I ended up learning.

Next Steps:

If I were to continue working in this project, I would like to mass-sweep and find the least Zipf-y poet for the sake of exploring their literature and trying to come up with a reason as to why they don't follow Zipf as closely. Also looking at poetry in different languages could be interesting if they are perhaps more different than their base language (I am thinking about something like Japanese with Haikus and Tankas). Cleaning up the functions and making the code more functional and less brute force would also be a priority.

References:

[1]https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4176592/ (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4176592/)

[2]https://www.kaggle.com/rtatman/english-word-frequency (https://www.kaggle.com/rtatman/english-word-frequency)

[3]https://www.youtube.com/watch?v=fCn8zs912OE&t=194s (https://www.youtube.com/watch?v=fCn8zs912OE&t=194s)

[4]https://requests.readthedocs.io/en/master/ (https://requests.readthedocs.io/en/master/)

[5]https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (https://www.crummy.com/software/BeautifulSoup/bs4/doc/)

[6]https://numpy.org/ (https://numpy.org/)

[7]https://www.scipy.org/ (https://www.scipy.org/)

[8]https://pypi.org/project/language-tool-python/ (https://pypi.org/project/language-tool-python/)

[9]https://docs.python.org/2/library/collections.html (https://docs.python.org/2/library/collections.html)

[10]https://docs.python.org/2/library/string.html (https://docs.python.org/2/library/string.html)

[11]https://docs.python.org/3/library/math.html (https://docs.python.org/3/library/math.html)

[12]https://matplotlib.org/ (https://matplotlib.org/)

[13]https://sci-hub.se/https://www.tandfonline.com/doi/full/10.1080/00107510500052444?scroll=top&needAccess=false (https://sci-hub.se/https://www.tandfonline.com/doi/full/10.1080/00107510500052444?scroll=top&needAccess=false)

[14] https://www.interaction-design.org/literature/article/the-pareto-principle-and-your-user-experience-work (https://www.interaction-design.org/literature/article/the-pareto-principle-and-your-user-experience-work)