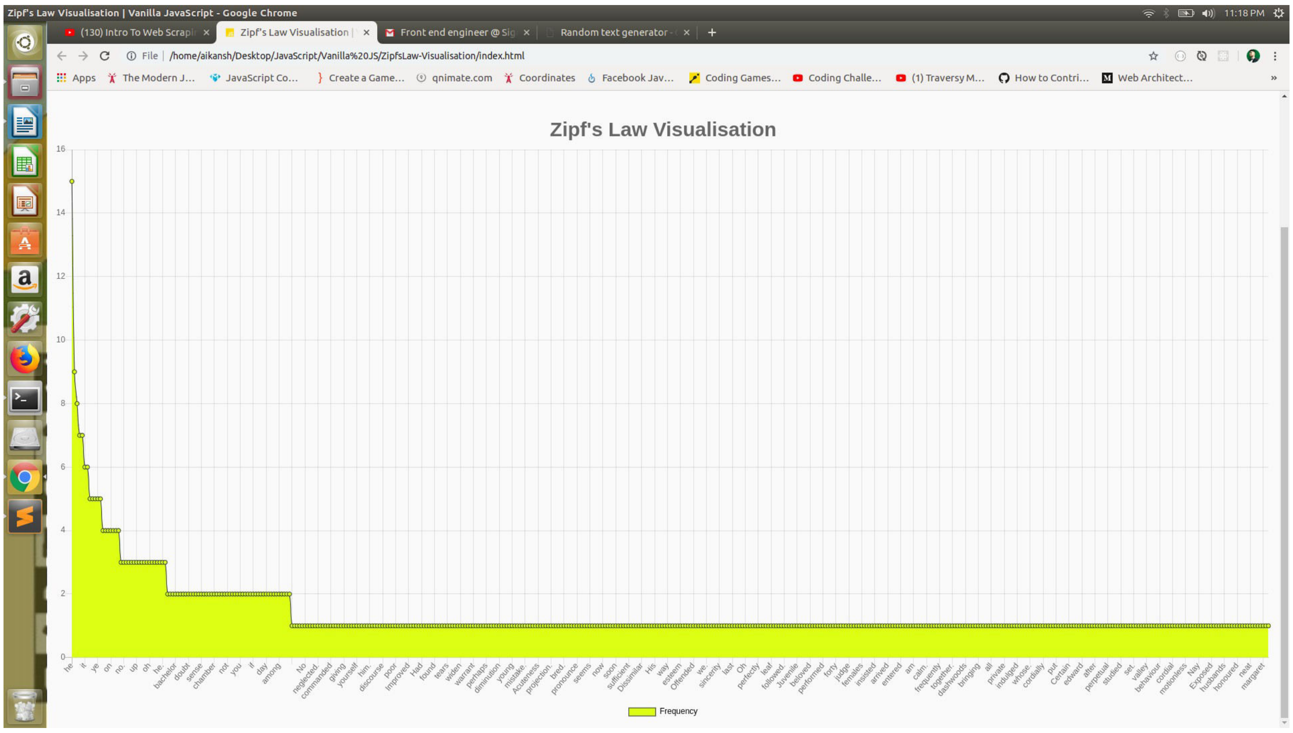
Task: Demonstrate Zipf’s law through visualization & match with the theoretical distribution

Tech stack used:

- Vanilla Javascript (why make things complicated? :D)
- CSS (some very basic styling & layout)
- Chart.js (charting library for frontend visualisation)

What the end Result looks like?



Approach taken:

1. Take user input

- Used <textarea/> tag for allowing user to enter any amount of text by either typing or pasting it from somewhere. This is the text I used to run zipf's law on.
- Used <button/> tag which fires an init function to get the visualization on the frontend.
- The method called on clicking 'Visualize' is called *init()* as it start the process, only if the input field is non-empty. Then, *parseData()* is called to curate the data.

2. Curate this data to make it word by word

- *parseData()* method stores the user input in a const container called string.
- This string is converted to an array(*arrOfWords*) having all the words in the text as it's elements. I used regex for tabs & newline characters: The special character \s will match any white space character. And to account for multiple whitespace characters, I can just use *s*+ since the + character matches the preceding character (in this case, white space) if it occurs 1 or more times.

3. Counting the frequency of each word

- Counting Words and Creating an Object(*wordCounts*): Next, I need to do two things: 1) count the instances of each word, and 2) convert this array into a list of properties within an object.
- *Object. create(null)* creates an object with no prototype and no inherited properties or methods. This helps in handling reserved keywords like toString and prototype as well.
- For filling this object, I loop over *arrOfWords* and add {key:value} pairs according to words and their frequency, respectively.
- Then I pass this filled up object to another function called curateData() which sorts it descendingly in order of frequency of words.
- sortWordCounts() method converts the object into an array of arrays(2 elements each: {word, frequency}). This new array called sortable is sorted by using array.prototype.sort method which takes a function as param and sorts as per returned value being -ve/+ve.

4. Making the graph using Chart.js object

- makeGraph() method takes two arguments, wordArray having the list of all the words in text(x-axis) and wordFrequencyArray having the respective frequencies of these words(y-axis).
- Chart.js object has data.lables property for x-axis and data.datasets[0].data for y-axis.
- Other properties of zipfLawChart are used for some basic styling of the graph. Type of the graph plotted is ‘line’ which helps in clarity of visualization for the plot.

Possible FAQs

Q. Why vanilla JS? Why not React.js or some framework?

For a small project like this, or one where performance is absolutely critical, I think the prospect of creating a complete app in VanillaJS doesn't seem quite so daunting and it's better to not make things complicated. Well-structured Vanilla JavaScript will always be faster than React. Moreover, the source code(written in React.js) for this task was available on multiple github accounts. I wanted to write the code from scratch and find my own way of doing things!
Although, I find React as a very impressive and fun option to work with where complex Uis are involved and different pages need to be rendered on same root (SPAs).

Q. Why Chart.js?

Chart.js is a community maintained open-source library that helps you easily visualize data using JavaScript. It supports 8 different chart types (including bars, lines, & pies), and they’re all responsive. In other words, you set up your chart once, and Chart.js will do the heavy-lifting for you and make sure that it’s always legible (for example by removing some uncritical details if the chart gets smaller). Although less flexible and capable than D3, it’s easier to wrap your head around and to get started with, yet powerful enough to cover more than just your basic needs.

In a gist, this is what you need to do to draw a chart with Chart.js:

1. Define where on your page to draw the graph.
2. Define what type of graph you want to draw.
3. Supply Chart.js with data, labels, and other options.

…and you’ll get a beautiful, responsive, graph!