This report is divided into 7 parts to evaluate each function on its own.

1. **Creating a Wordcloud with TFIDF/TF**

Words within the “docs” are first tokenized using word\_tokenize and then checked if it contains any stopwords. Stopword list of nltk and some extra hand-written stopwords are removed from the tokenized texts and they are appended to new list “words\_modified”.

Afterwards, they are added to “comment\_words” string one by one, then word cloud is created with “wordcloud” library. For the dataset T\_sample5000.pkl, wordcloud below is created within 4.513 minutes.



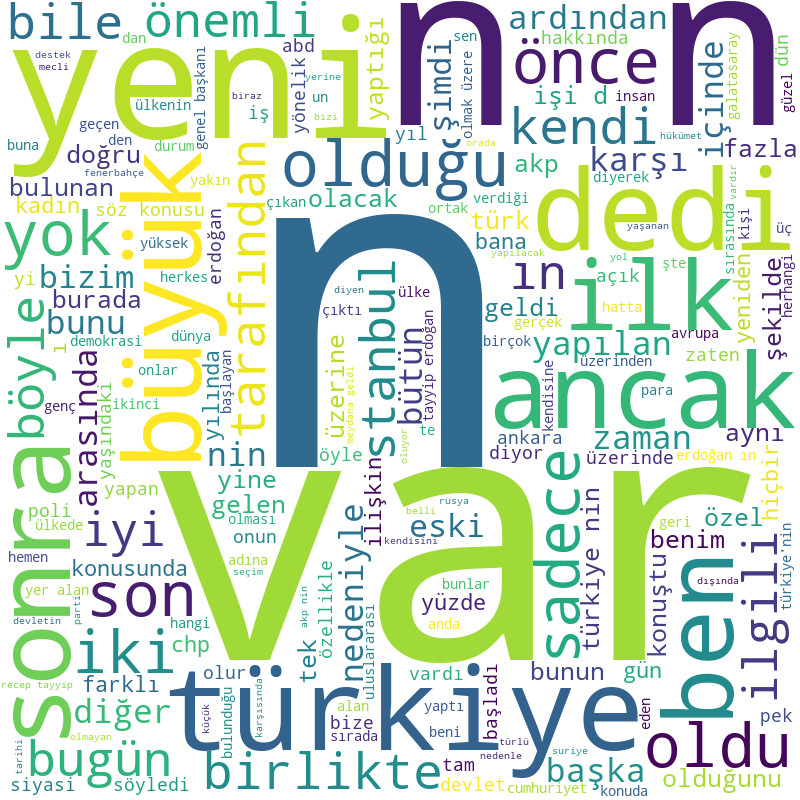
For the dataset T\_sample20000.pkl, wordcloud below is created within 66.513 minutes.



Bigger datasets could not be used because they consume too much time.

On the previous runs, wordcloud below was produced. Since this was the first wordcloud, I do not know how much time it took to be generated but it was around 3-4 minutes. If we look closely, we can observe that some of the stopwords are present within the wordcloud (These stopwords were absent in the nltk stopwords list and added later on by hardcoding). Some of these stopwords are: var, önce, dedi, oldu, yok bile …

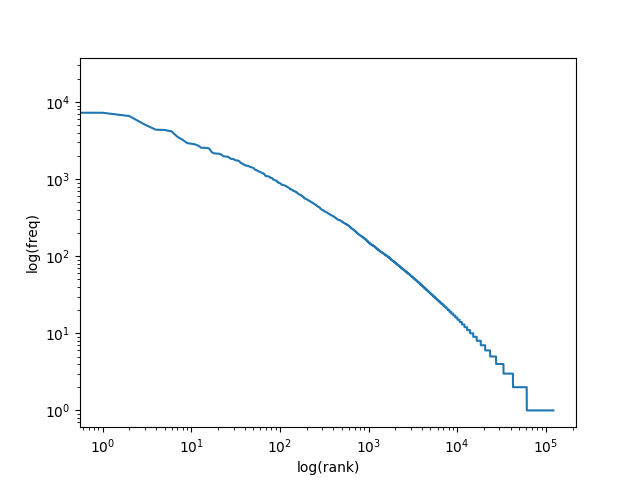
They can be seen in the stopwords array in the code.

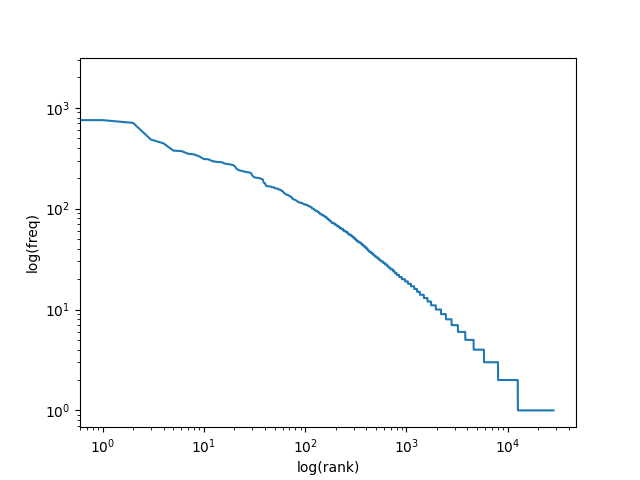


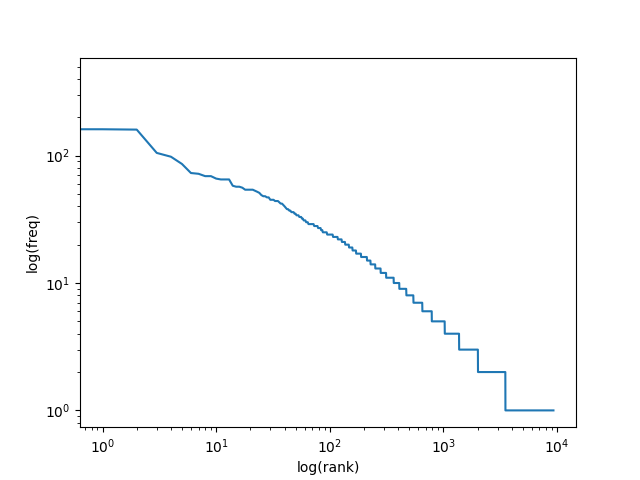
1. **Creating Ziph’s Plot**

Words within the “docs” are first tokenized using word\_tokenize and then they are added to the string “comment\_words” . Words within “comment\_words” are picked with a help of a regular expression and put into “words” arry. Frequency table is created according to their frequencies within the “words”. Values of the “frequency” table is sorted, and that variable is used to create Ziph’s plot. X and y axis are labeled with log since table is created with “plt.loglog” function.

For the dataset T\_sample5000.pkl, Ziph’s Plot below is created within 3.88 minutes.



Another run was made with the same sample but using only [0:500] instances. It has a very similar shape as can be seen below. It took 0.046 minutes to draw the graph below.

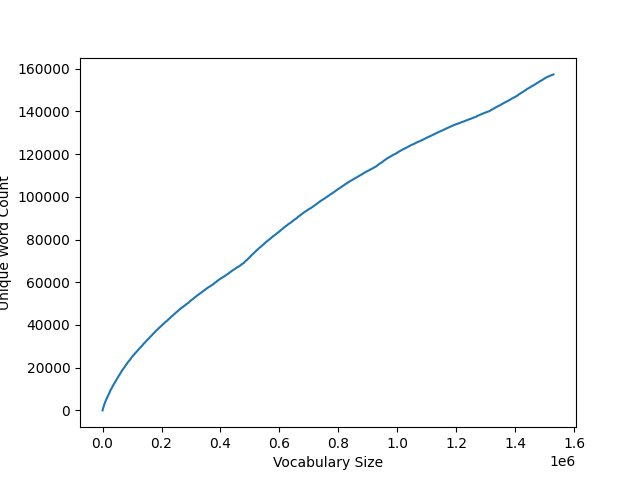
Last sample run is made with the same dataset but using only [0:100] instances. Graph below is created within 0.026 minutes. It is very similar to the graphs above but only numbers are smaller since only first 100 instances of the dataset are used.

For the dataset T\_sample20000.pkl, Ziph’s Plot took more than an hour, so it is cancelled while running and bigger datasets are not evaluated since their time cost is too high.

1. **Creating Heap’s Plot**

For each item in the “docs”, word\_tokenize is used and for each word in the list, unique words are counted. Vocabulary size is incremented after each iteration and if a unique word is seen, unique word count is also incremented. At each iteration, “unique\_word\_count” and “vocab\_size” variables are pushed into separate lists named “y\_val” and “x\_val”, respectively. At the end of the loop, Heap’s plot is generated with the variables inside of the “x\_val” and “y\_val” lists.

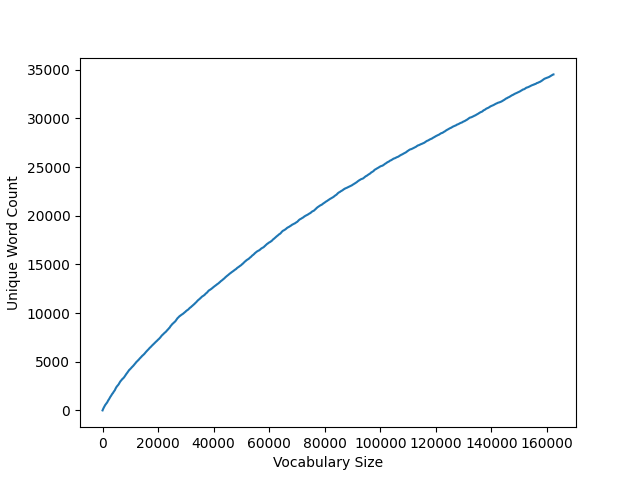
For the dataset T\_sample5000.pkl, Heaps’ Plot below is created within 39.721 minutes.



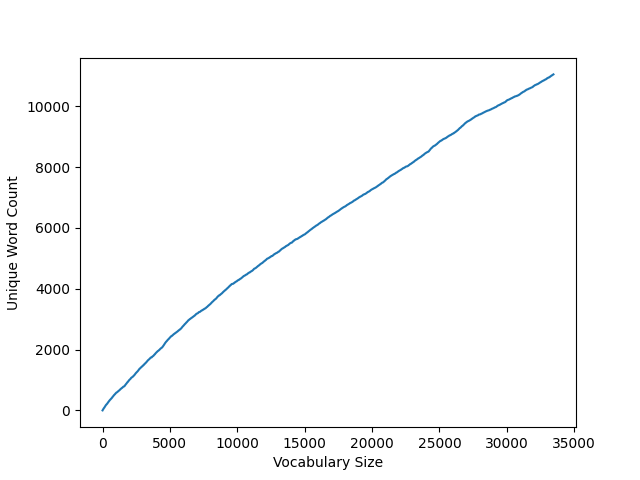
We can observe a decreasing slope as we go to the right of the curve.

Since the time left is very little, I was able to run docs[0:500] and docs[0:100] of the T\_sample5000.pkl to see the correct format of the Heaps’ Plot.

Below T\_sample5000.pkl with only first 500 instances. It took 0.937 minutes to draw it.



As you can see above, vocabulary size increases faster than the unique word count, creating a decreasing slope while we move to the right.

Another sample run is made with the same dataset but using only the first 100 instances and graph below is created within 0.054 minutes. It has a similar shape with the graphs above.

1. **Creating Language Model**

First, every string in “docs” is concatenated into one string variable “s” and lowered. Afterwards, they are tokenized using “word\_tokenize”. Then “padded\_everygram\_pipeline” is called with tokenized texts for preprocessing purposes.

Then according to the string parameter “model\_type” MLE or KneserNeyInterpolated model is created and the model is returned.

For the dataset T\_sample5000.pkl, MLE language model is created within 1.37 minutes.

For the dataset T\_sample5000.pkl, KneserNeyInterpolated model is created within 1.311 minutes.

1. **Generating sentence with the trained model**

Five different loops are set for creating 5 different sentences. At first, given “text” parameters is used to generate words with the model. At next iterations of the loop, previously generated token is given as “text\_seed” and words are generated and pushed into the list until “</s>” is seen for each loop. Lists are detokenized using “TreebankWordDetokenizer” and perplexities are calculated with the everygrams of lists of words of each sentence. For a reason that I cannot resolve, perplexities of the sentences are calculated as infinity, even though I give them ngrams of the sentences. I tried ngrams, everygrams but result did not change. Since all of them are infinity and there is no minimum among them, first sentence is returned via function.

KneserNeyModel was not stopping while generating a sentence, because of that reason I put a word limit of 200 for a sentence. If number of words within the sentence exceeds 200, it terminates automatically and generated sentence is composed of 200 words.

1. **Creating Word Vectors**

First, “docs” is tokenized with word\_tokenize. After that, according to string parameter “model\_type”, Word2Vec is called with given dimension\_size, window parameters. Workers are set to number of cpu’s minus 1 for safety reasons and minimum count is set to 5 (Frequencies lower than 5 are ignored within the model.).

Model is created without the vocabulary at first. Vocabular is added at next line with “build\_vocab” function. Model is trained and afterwards model.init\_sims(replace = True) is called so that it does not consume too much memory. And model is returned.

For the dataset T\_sample5000.pkl, CBOW word vector is created within 0.73 minutes.

For the dataset T\_sample5000.pkl, Skipgram word vector is created within 1.993 minutes.

1. **Using word relationships**

Empty “difference\_vector”, “sum\_vector” and “xminusy\_vector” is created with size equal to the vector size of the given model. For each pair in the example tuple list, difference vector of elements inside of each tuple is calculated. For instance, for tuple (fransa, paris), WE.wv[‘fransa’] – WE.wv[‘paris’] is calculated. If any element within the vocabulary is not found, KeyError exception is throwed, if none of the pairs are found within the vocabulary “Sorry, this operation cannot be performed!1” is printed.

After loop is completed, average vector is calculated by dividing xminusy\_vector (sum of differences) into count.

Then, if x value is missing, average vector added to the y value to obtain x value. If y value is missing, average value is subtracted from the x value to obtain y value. If count is not incremented after the operation, meaning and error occurred within the try block, "Sorry, this operation cannot be performed!2" is printed. If KeyError is throwed then “Sorry, this operation cannot be performed! KeyError at example\_tuple\_Test, at item ‘item’ ” is printed.