Distributions

September 12, 2023

1 Distributions of Words & Sentences

This assignment is comprised of three tasks for ITCS 4111 and an additional task for ITCS 5111:

1. The first task is to compute the frequency vs. rank distribution of the words in Moby Dick. For this, you will need to tokenize the document and create a vocabulary mapping word types to their document frequency.
2. The second task is to segment the document into sentences and compute the sentence length distribution. Here you will experiment with spaCy's default sentence segmenter as well as the simple rule-based Sentencizer.
3. The third task is the same as the first except that we use subword tokenization.
4. Use spacy's NE recognizer to find all named entities in the first 2,500 paragraphs. Count how many times they appear in the document and consolidate them based on their most frequent type.

1.1 Write Your Name Here: Claire Ardern

2 Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of ll cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a PDF version showing the code and the output of all cells, and save it in the same folder that contains the notebook file.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing we will see when grading!
7. Submit **both** your PDF and notebook on Canvas.
8. Make sure your your Canvas submission contains the correct files by downloading it after posting it on Canvas.

2.1 Word distributions using the SpaCy tokenizer (40 + 10 points)

First, create the spaCy tokenizer.

[54]:
```
from spacy.lang.en import English
nlp = English()
```

```
tokenizer = nlp.tokenizer
```

Create a *vocab* dictionary. This dictionary will map tokens to their counts in the input text file.

[55]:
```
vocab = {}
```

Read the input file line by line.

1. Tokenize each line.
2. For each token in the line that contains only letters, convert it to lower case and increment the corresponding count in the dictionary.
   - If the token does not exist in the dictionary yet, insert it with a count of 1. For example, the first time the token 'water' is encountered, the code should evaluate *vocab['water'] = 1*.

At the end of this code segment, *vocab* should map each word type to the number of times it appeared in the entire document. There should be 16830 word types and 214287 words in Moby Dick.

[56]:
```
with open('../data/melville-moby_dick.txt', 'r') as f:
    for line in f:

        tokens = tokenizer(line)

        for tok in tokens:

            if tok.text.isalpha():
                tok = tok.text.lower()

                if tok in vocab.keys():
                    vocab[tok] += 1
                elif tok not in vocab.keys():
                    vocab[tok] = 1

print('There are', len(vocab), 'word types in Moby Dick.')
print('There are', sum(vocab.values()), 'words in Moby Dick.')
```

```
There are 16830 word types in Moby Dick.
There are 214287 words in Moby Dick.
```

Create a list *ranked* of tuples *(word, freq)* that contains all the words in the vocabulary *vocab* sorted by frequency. For example, if *vocab = {'duck':2, 'goose':5, 'turkey':3}*, then *ranked = [('goose', 5), ('turkey', 3), ('duck', 2)]*.

[57]:
```
ranked = [(k, v) for k, v in vocab.items()]
ranked.sort(key=lambda x: x[1], reverse=True)
```

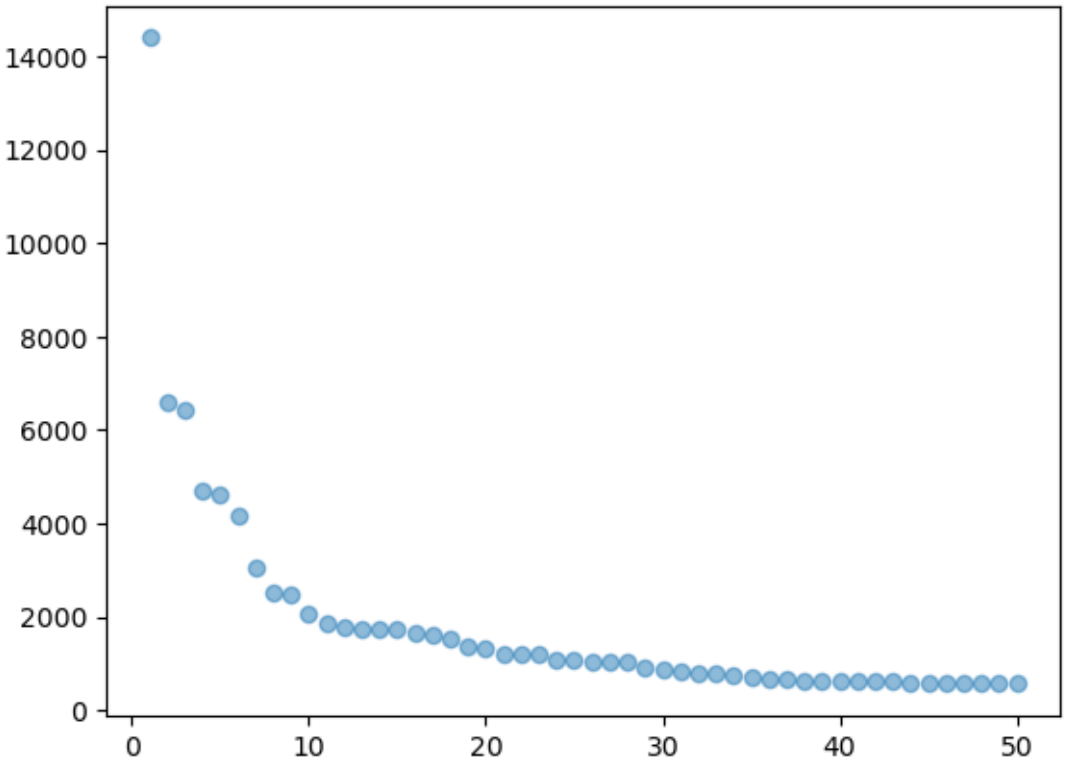
Print the top 10 words in the sorted list.

[58]:
```
print('Size of vocabulary:', len(ranked))
for word, freq in ranked[:10]:
    print(word, freq)
```

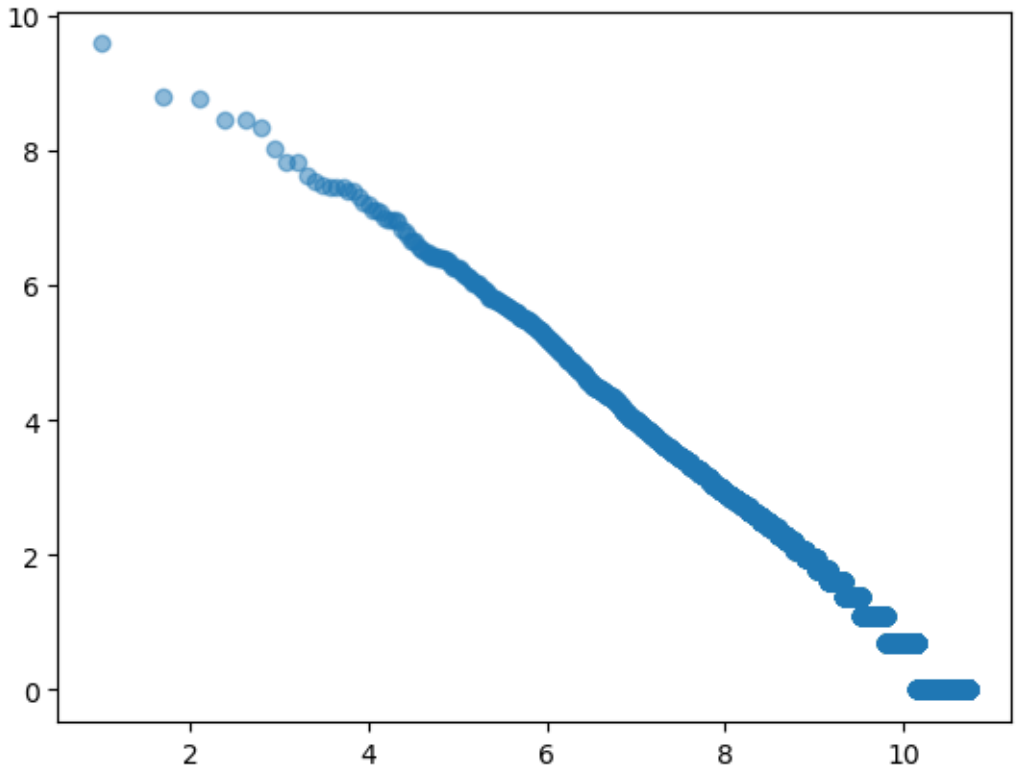
```
Size of vocabulary: 16830
the 14388
of 6606
and 6414
a 4698
to 4618
in 4164
that 3061
his 2527
it 2489
i 2068
```

Plot the frequency vs. rank of the top ranked words in Moby Dick.

[59]:
```
import matplotlib.pyplot as plt
ranks = range(1, 50 + 1)
freqs = [t[1] for t in ranked[:50]]
plt.scatter(ranks, freqs, c='#1f77b4', alpha=0.5)
plt.show()
```



[60]:
```
import math
ranks = [1 + math.log(r) for r in range(1, len(ranked) + 1)]
freqs = [math.log(t[1]) for t in ranked]
plt.scatter(ranks, freqs, c='#1f77b4', alpha=0.5)
plt.show()
```



2.2 Sentence distributions (40 + 10 points)

First, try to create the spaCy nlp object from the entire text of Moby Dick. This will likely not work, it is not a good idea to read all the text.

[61]:
```
import spacy

nlp = spacy.load("en_core_web_sm")
text = open('../data/melville-moby_dick.txt', 'r').read()
doc = nlp(text)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[61], line 5
      3 nlp = spacy.load("en_core_web_sm")
```

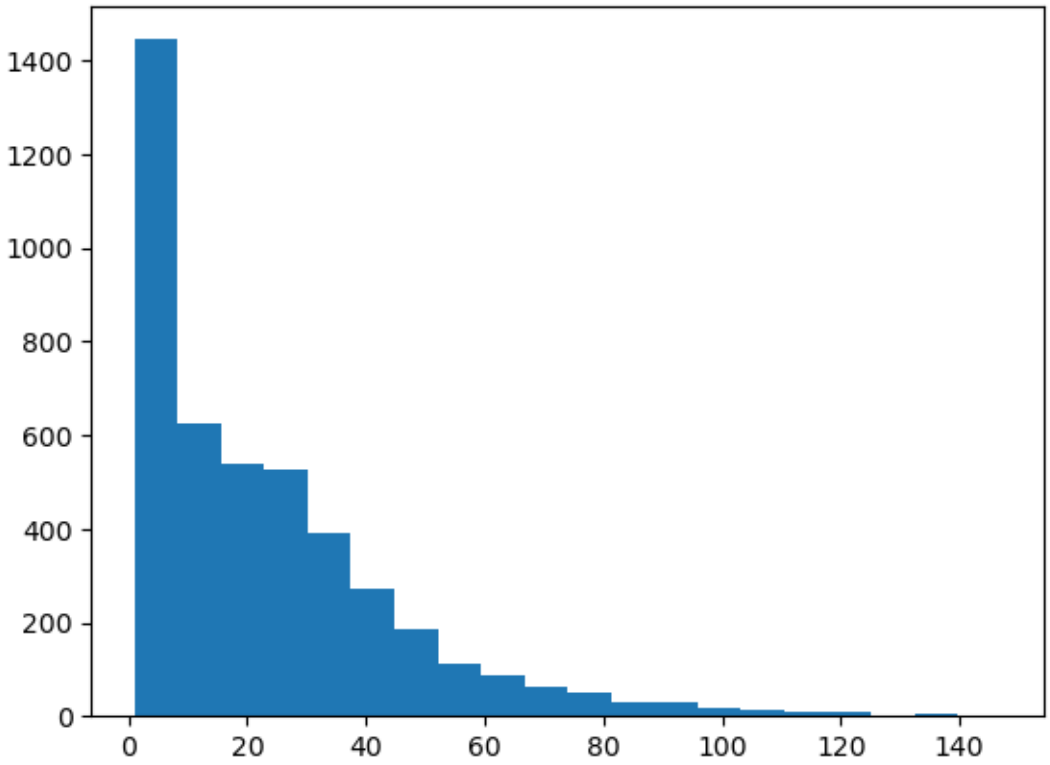
```
      4 text = open('../data/melville-moby_dick.txt', 'r').read()
----> 5 doc = nlp(text)

File ~\AppData\Roaming\Python\Python39\site-packages\spacy\language.py:1030, in
 ↪Language.__call__(self, text, disable, component_cfg)
   1009 def __call__(
   1010     self,
   1011     text: Union[str, Doc],
   (…)
   1014     component_cfg: Optional[Dict[str, Dict[str, Any]]] = None,
   1015 ) -> Doc:
   1016     """Apply the pipeline to some text. The text can span multiple␣
 ↪sentences,
   1017     and can contain arbitrary whitespace. Alignment into the original␣
 ↪string
   1018     is preserved.
   (…)
   1028     DOCS: https://spacy.io/api/language#call
   1029     """
-> 1030     doc = self._ensure_doc(text)
   1031     if component_cfg is None:
   1032         component_cfg = {}

File ~\AppData\Roaming\Python\Python39\site-packages\spacy\language.py:1121, in
 ↪Language._ensure_doc(self, doc_like)
   1119     return doc_like
   1120 if isinstance(doc_like, str):
-> 1121     return self.make_doc(doc_like)
   1122 if isinstance(doc_like, bytes):
   1123     return Doc(self.vocab).from_bytes(doc_like)

File ~\AppData\Roaming\Python\Python39\site-packages\spacy\language.py:1110, in
 ↪Language.make_doc(self, text)
   1104 """Turn a text into a Doc object.
   1105
   1106 text (str): The text to process.
   1107 RETURNS (Doc): The processed doc.
   1108 """
   1109 if len(text) > self.max_length:
-> 1110     raise ValueError(
   1111         Errors.E088.format(length=len(text), max_length=self.max_length
   1112     )
   1113 return self.tokenizer(text)
```

```
ValueError: [E088] Text of length 1220066 exceeds maximum of 1000000. The parser
↪and NER models require roughly 1GB of temporary memory per 100,000 characters
↪in the input. This means long texts may cause memory allocation errors. If
↪you're not using the parser or NER, it's probably safe to increase the `nlp.
↪max_length` limit. The limit is in number of characters, so you can check
↪whether your inputs are too long by checking `len(text)`.
```

Instead, read the document paragraph by paragraph, i.e. in chunks of text separated by empty lines. Before using spaCy to segment a paragraph into sentences, replace each end of line character with a whitespace, to allow a sentence to span multiple lines. After sentence segmentation, for each sentence in the paragraph append its length (in tokens) to *lengths*. Use the default *nlp* class to process each paragraph and split it into sentences. Stop after processing 1000 paragraphs. This will be slow, so be patient.

[62]:
```
import spacy

nlp = spacy.load("en_core_web_sm")

# the number of paragraphs read so far.
count = 0
# stores the length of each sentence processed so far.
lengths = []
# make sure the file is read line by line.
with open('../data/melville-moby_dick.txt', 'r') as f:
    # YOUR CODE GOES HERE

    parag = ""

    for line in f:

        if count <= 1000:


            # Add the line to the current paragraph
            if line.strip():
                parag = parag + line

            # Check if it is the end of a paragraph
            elif parag:
                count += 1
                sentences = nlp(parag).sents

                for sent in sentences:
                    lengths.append(len(sent))

                parag = ""
```

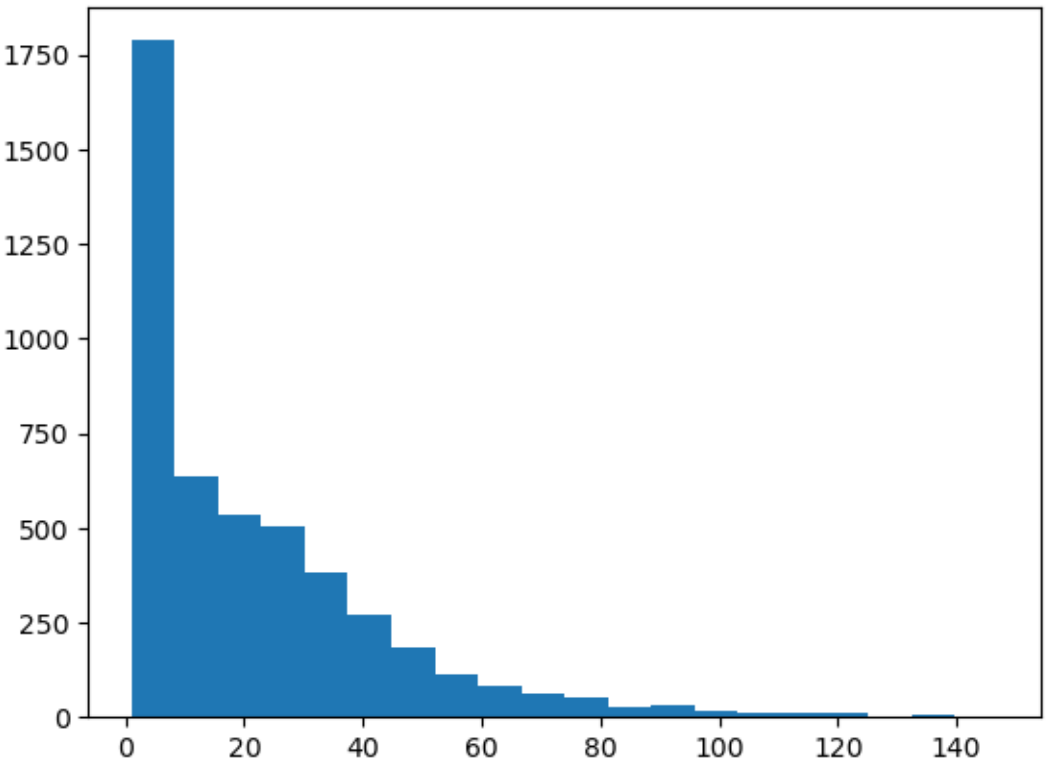
```
len150 = [l for l in lengths if l <= 150]
plt.hist(len150, bins = 20)
plt.show()
```



Next, do the same processing as above, but use the more robust Sentencizer to split paragraphs into sentences. Note the speedup.

[63]:
```
from spacy.lang.en import English

nlp = English()
nlp.add_pipe("sentencizer")

# the number of paragraphs read so far.
count = 0
# stores the length of each sentence processed so far.
lengths = []
with open('../data/melville-moby_dick.txt', 'r') as f:
    # YOUR CODE GOES HERE

    parag = ""

    for line in f:
```

```
        if count <= 1000:

            # Add the line to the current paragraph
            if line.strip():
                parag = parag + line

            # Check if it is the end of a paragraph
            elif parag:
                count += 1
                sentences = nlp(parag).sents

                for sent in sentences:
                    lengths.append(len(sent))

                parag = ""


len150 = [l for l in lengths if l <= 150]
plt.hist(len150, bins = 20)
plt.show()
```



Note the difference between the two histograms. Identify at least 5 examples of sentences in Moby Dick that are segmented differently by the two approaches. Copy them below and explain the differences. Which method seems to be more accurate?

EXAMPLE 1 Method 1: incontinently that foul great swallow of his, **and perisheth in, the Method 2:** incontinently that foul great swallow of his, **and perisheth in the**

EXAMPLE 2 Method 1: southern seas, as I may say, by a hundred to one; **than we have to, the Method 2:** southern seas, as I may say, by a hundred to one; **than we have to the**

EXAMPLE 3 Method 1: "Where away?" demanded the captain. **Method 2: "Where away?",** demanded the captain.

EXAMPLE 4 Method 1: "The whale fell directly over him, and probably **killed him in, a Method 2:**"The whale fell directly over him, and probably **killed him in a**

EXAMPLE 5 Method 1: "The Whale is harpooned to be sure; **but bethink you,, how** you would **Method 2:**"The Whale is harpooned to be sure; **but bethink you, how** you would

In each of the above examples, the two methods of sentence segmentation achieve different results. In the majority of the included example cases, it seems that Method 1 segments the sentence in awkward, inaccurate places. This can be seen with the addition of commas indicating sentences segments. Based on these examples, it seems that Method 2 (the more robust Sentencizer) is more accurate.

2.3 Word distribution using OpenAI's subword tokenization (30 points)

In this part, we will compute the frequency vs. rank based on the the BPE subword tokenization created by the tiktoken module from OpenAI.

Read the input file line by line.

1. Tokenize each line using `tiktoken` encoder and decoder for GPT-3.5.
2. For each token in the line that contains only letters, convert it to lower case and increment the corresponding count in the dictionary.
   - If the token does not exist in the dictionary yet, insert it with a count of 1. For example, the first time the token 'water' is encountered, the code should evaluate *vocab['water'] = 1*.

At the end of this code segment, *vocab* should map each word type to the number of times it appeared in the entire document. There should be 14619 unique types and 248615 total tokens in Moby Dick.

[64]:
```
import tiktoken

# To get the tokeniser corresponding to a specific model in the OpenAI API:
enc = tiktoken.encoding_for_model("gpt-3.5-turbo")

vocab = {}
with open('../data/melville-moby_dick.txt', 'r') as f:
    for line in f:
        # YOUR CODE HERE
```

```
        tokens = [enc.decode_single_token_bytes(token) for token in enc.
↪encode(line)]

        for t in tokens:
            t = t.strip()
            if t.isalpha():

                # IF YOU CONVERT TO LOWER CASE, INCORRECT # UNIQUE TYPES
                #t = t.lower()

                if t in vocab.keys():
                    vocab[t] += 1
                elif t not in vocab.keys():
                    vocab[t] = 1


print('There are', len(vocab), 'unique tokens in Moby Dick.')
print('There are', sum(vocab.values()), 'tokens in Moby Dick.')
```

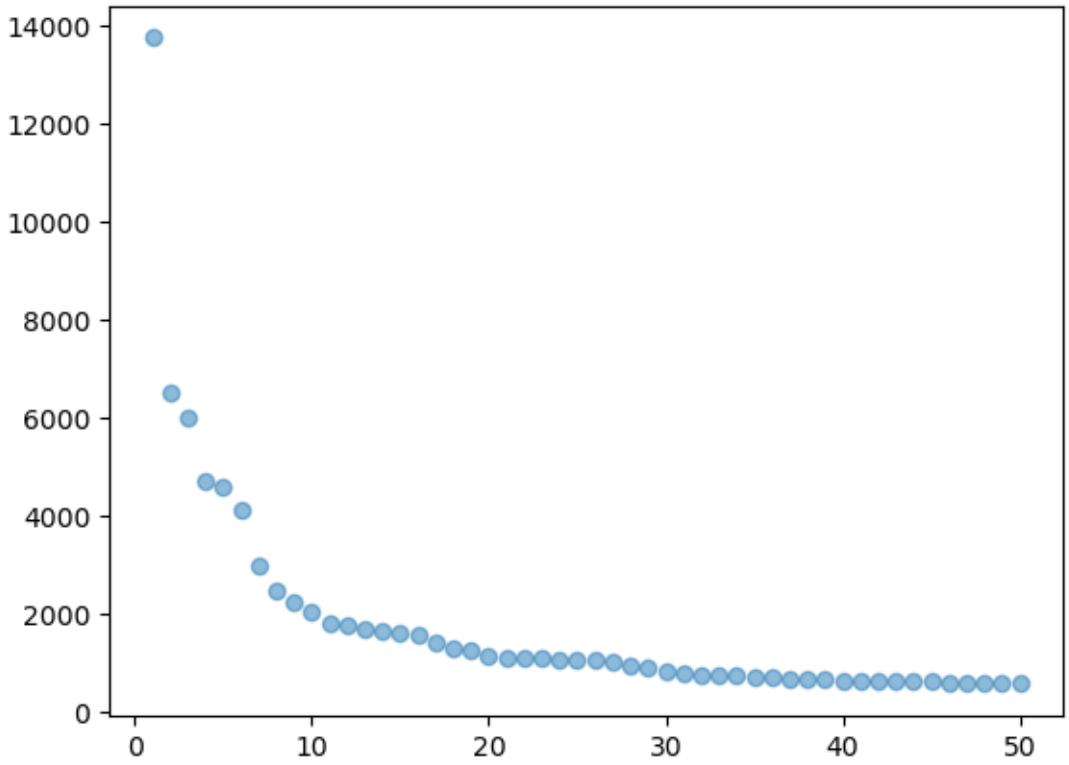
```
There are 14619 unique tokens in Moby Dick.
There are 248615 tokens in Moby Dick.
```

Rank the tokens based on their frequency, then plot frequency vs. rank.

[65]:
```
ranked = [(k, v) for k, v in vocab.items()]
ranked.sort(key=lambda x: x[1], reverse=True)

print('Size of vocabulary:', len(ranked))
for word, freq in ranked[:10]:
    print(word, freq)
```

```
Size of vocabulary: 14619
b'the' 13739
b'of' 6527
b'and' 6024
b'a' 4711
b'to' 4582
b'in' 4137
b'that' 2982
b'his' 2473
b'it' 2260
b'I' 2071
```

[66]:
```
import matplotlib.pyplot as plt
ranks = range(1, 50 + 1)
freqs = [t[1] for t in ranked[:50]]
plt.scatter(ranks, freqs, c='#1f77b4', alpha=0.5)
plt.show()
```



2.4 [5111] Named Entities (10 + 10 + 10 + 10 + 10 points)

Useful documentation is at: - https://spacy.io/usage/linguistic-features#named-entities - https://spacy.io/api/entityrecognizer

[67]:
```
import spacy

nlp = spacy.load("en_core_web_sm")

# These are all the entity types covered by spaCy's NE recognizer.
nlp.pipe_labels['ner']
```

[67]:
```
['CARDINAL',
 'DATE',
 'EVENT',
 'FAC',
 'GPE',
 'LANGUAGE',
 'LAW',
 'LOC',
 'MONEY',
```

```
 'NORP',
 'ORDINAL',
 'ORG',
 'PERCENT',
 'PERSON',
 'PRODUCT',
 'QUANTITY',
 'TIME',
 'WORK_OF_ART']
```

Read the first 2,500 paragraphs in Moby Dick and extract all named entities into a dictionary `ne_counts` that maps each *named entity* to its frequency. By *named entity* we mean a tuple *(name, type)* where *name* is the entity name as a string, and *type* is its entity type. For example, if the name 'Ahab' appears with the NE type 'PERSON' 50 times, then the dictionary should map the key *('Ahab', 'PERSON')* to the value *50*.

[68]:
```
# The number of paragraphs read so far.
count = 0
# Stores the dictionary of named entites and their counts.
ne_counts = {}

# Make sure the file is read line by line.
with open('../data/melville-moby_dick.txt', 'r') as f:
    # YOUR CODE GOES HERE
    parag = ""

    for line in f:
        line = line.replace("\n", ' ')

        if count <= 2500:

            # Add the line to the current paragraph
            if line.strip():
                parag = parag + line

            # Check if it is the end of a paragraph :
            elif parag:

                count += 1
                entities = nlp(parag).ents

                for ent in entities:

                    if (ent.text, ent.label_)  in ne_counts.keys():
                        ne_counts[(ent.text, ent.label_)] += 1
                    elif (ent.text, ent.label_) not in ne_counts.keys():
                        ne_counts[(ent.text, ent.label_)] = 1
```

```
                parag = ""
```

Create a list `ranked_ne` containing all the items in the `ne_counts` dictionary that is sorted in descending order by their frequency.

[69]:
```
ranked_ne = [(k, v) for k, v in ne_counts.items()]
ranked_ne.sort(key=lambda x: x[1], reverse=True)

# This should display 2974 unique named entities, with the top two being
# ('Ahab', 'PERSON') 347 and ('one', 'CARDINAL') 335
print('Unique named entities:', len(ranked_ne))
for ne, count in ranked_ne[:50]:
    print(ne, count)
```

```
Unique named entities: 2673
('one', 'CARDINAL') 335
('Ahab', 'PERSON') 313
('two', 'CARDINAL') 204
('Queequeg', 'ORG') 193
('first', 'ORDINAL') 178
('Starbuck', 'PERSON') 140
('three', 'CARDINAL') 134
('Stubb', 'PERSON') 93
('half', 'CARDINAL') 85
('Pequod', 'GPE') 82
('Peleg', 'PERSON') 68
('Bildad', 'GPE') 59
('Nantucket', 'GPE') 57
('Moby Dick', 'PERSON') 52
('Indian', 'NORP') 47
('Pequod', 'ORG') 47
('Leviathan', 'GPE') 45
('second', 'ORDINAL') 45
('four', 'CARDINAL') 42
('Tashtego', 'ORG') 39
('Flask', 'ORG') 36
('thou', 'GPE') 31
('American', 'NORP') 28
('quarter', 'CARDINAL') 28
('night', 'TIME') 27
('Greenland', 'GPE') 26
('English', 'LANGUAGE') 25
('thou', 'CARDINAL') 24
('Dutch', 'NORP') 23
('Pacific', 'LOC') 23
('Jonah', 'GPE') 23
('the Sperm Whale', 'ORG') 23
```

```
('One', 'CARDINAL') 19
('Gabriel', 'PERSON') 18
('English', 'NORP') 18
('Nantucketer', 'ORG') 18
('French', 'NORP') 18
('Christian', 'NORP') 17
('Stubb', 'ORG') 17
('the White Whale', 'ORG') 17
('Lakeman', 'PERSON') 17
('ten', 'CARDINAL') 16
('New Bedford', 'GPE') 16
('Pip', 'PERSON') 16
('Atlantic', 'LOC') 15
('First', 'ORDINAL') 15
('Yojo', 'PERSON') 15
('Fedallah', 'PERSON') 15
('Cape Horn', 'LOC') 14
('Ishmael', 'GPE') 14
```

2.4.1 Consolidate named entities

Some names appear with more than one type, most often due to errors in named entity recognition. One way to fix such errors is to use the fact that typically a name occurs with just one meaning in a document, as such it has just one type. In this part of the assignment, we will consolidate the extracted names such that the counts for the same name appearing with multiple types are added together, and by associating the name with the type that it appears with most often.

Create a dictionary `ne_types` that maps each name to a dictionary that contains all the types the name appears with, where each type is mapped to the corresponding count. Use information from the dictionary `ne_counts` above.

[70]:
```
ne_types = {}

# YOUR CODE HERE

for k, v in ne_counts.items():

    if k[0] not in ne_types.keys():

        ne_types[k[0]] = {}
        ne_types[k[0]][k[1]] = v

    elif k[0] in ne_types.keys():

        ne_types[k[0]][k[1]] = v
```

```
print(ne_types['Queequeg']) # this should print {'GPE': 109, 'NORP': 98,␣
 ↪'PERSON': 4, 'LANGUAGE': 8}

print(ne_types['Gabriel']) # this should print {'PERSON': 18, 'ORG': 1}
```

```
{'ORG': 193, 'NORP': 12, 'PERSON': 3}
{'PERSON': 18}
```

Create the consolidated dictionary `ne_cons` that maps each name to a tuple that contains its most frequent type and the total count over all types. Use information from the dictionary `ne_types` above.

[71]:
```
ne_cons = {}

# YOUR CODE HERE
for k in ne_types:

    #res = ne_types.get(k, {}).get('is')
    res = ne_types.get(k, {})
    #print(res)
    value = max(res.values())
    ne_cons[k] = (max(res, key=res.get), value)

print(ne_cons['Queequeg']) # this should print ('GPE', 219)

print(ne_cons['Gabriel']) # this should print ('PERSON', 19)
```

```
('ORG', 193)
('PERSON', 18)
```

Create a list `ranked_nec` that contains only the consolidated entries from `ne_cons` whose type is among the types listed in the list `types` below, sorted in descending order based on their total counts.

[72]:
```
types = ['PERSON', 'GPE', 'ORG', 'LOC', 'FAC']

# YOUR CODE HERE
ranked_nec = []

for k,v in ne_cons.items():
    if v[0] in types:
        ranked_nec.append((k,v))

ranked_nec.sort(key=lambda x: x[1], reverse=True)

# This should display 1632 consolidated named entities, with the top two␣
 ↪entries being
# Ahab ('PERSON', 347) and Queequeg ('GPE', 219)
print('Consolidated named entities:', len(ranked_nec))
```

```
for ne, count in ranked_nec[:30]:
    print(ne, count)
```

```
Consolidated named entities: 1452
Ahab ('PERSON', 313)
Starbuck ('PERSON', 140)
Stubb ('PERSON', 93)
Peleg ('PERSON', 68)
Moby Dick ('PERSON', 52)
Gabriel ('PERSON', 18)
Lakeman ('PERSON', 17)
Pip ('PERSON', 16)
Yojo ('PERSON', 15)
Fedallah ('PERSON', 15)
Hussey ('PERSON', 13)
Cook ('PERSON', 12)
Sperm Whale ('PERSON', 11)
Mapple ('PERSON', 9)
lee ('PERSON', 8)
Ramadan ('PERSON', 7)
Don Sebastian ('PERSON', 7)
Jeroboam ('PERSON', 7)
Yarman ('PERSON', 7)
Whales ('PERSON', 6)
Jove ('PERSON', 6)
Bulkington ('PERSON', 6)
Joppa ('PERSON', 6)
Nineveh ('PERSON', 6)
Pagan ('PERSON', 6)
Noah ('PERSON', 6)
Manxman ('PERSON', 6)
Fleece ('PERSON', 6)
Frenchman ('PERSON', 6)
Landlord ('PERSON', 5)
```

[Bonus points 1] (10 points) Select one name from the dictionary `ne_counts` that appears frequently with 2 types and explain why you think spaCy's named entity recognizer associated the name with those 2 types.

[73]:
```
for k, v in ne_counts.items():
    if k[0] == "Stubb":
        print((k,v))
```

```
(('Stubb', 'PERSON'), 93)
(('Stubb', 'ORG'), 17)
```

EXPLANATION

The name "Stubb" from the dictionary ne_counts appears frequently with 2 types.

SpaCy's named entity recognizer likely associated the name with these 2 types because Stubb is not a word commonly used in the English language. This word has no meaning other than, in this case, as a character name in Moby Dick. SpaCy has likely recognized that Stubb is not a regular word but instead must be some sort of title - either for a person or organization. This can be seen in the above output.

[Bonus points 2] (20 points) Find all the syntactic dependency paths connecting the subject Ahab with a direct object, e.g. 'Ahab' —> nsubj —> <verb> —> dobj —> <object>. Rank all the object words based on how frequently they appear connected to 'Ahab' through this syntactic pattern, and for the top 10 objects display the list of verbs that are used with each object.

Useful documentation is at: - https://spacy.io/usage/linguistic-features#dependency-parse

[ ]:
```
# YOUR CODE HERE
```

2.5 Bonus points

Anything extra goes here. For example:

- Write code Li (1992) showing that just random typing of letters including a space will generate "words" with a Zipfian distribution. Generate at least 1 million characters before your compute word frequencies.
  - Show mathematically that random typing results in a Zipf's distribution by computing probabilities for all words that contain just 1 letter, 2 letters, …
- Implement the BPE algorithm, where you break ties by selecting to merge in lexicographic order. Train the BPE algorithm on a large corpus and then use it to do subword tokenization on the Mobby Dick corpus. What are the top 10 most frequenct tokens and how does it compare with what you got from `tiktokenizer`.

Bonus Option 1: Zipfian Distribution

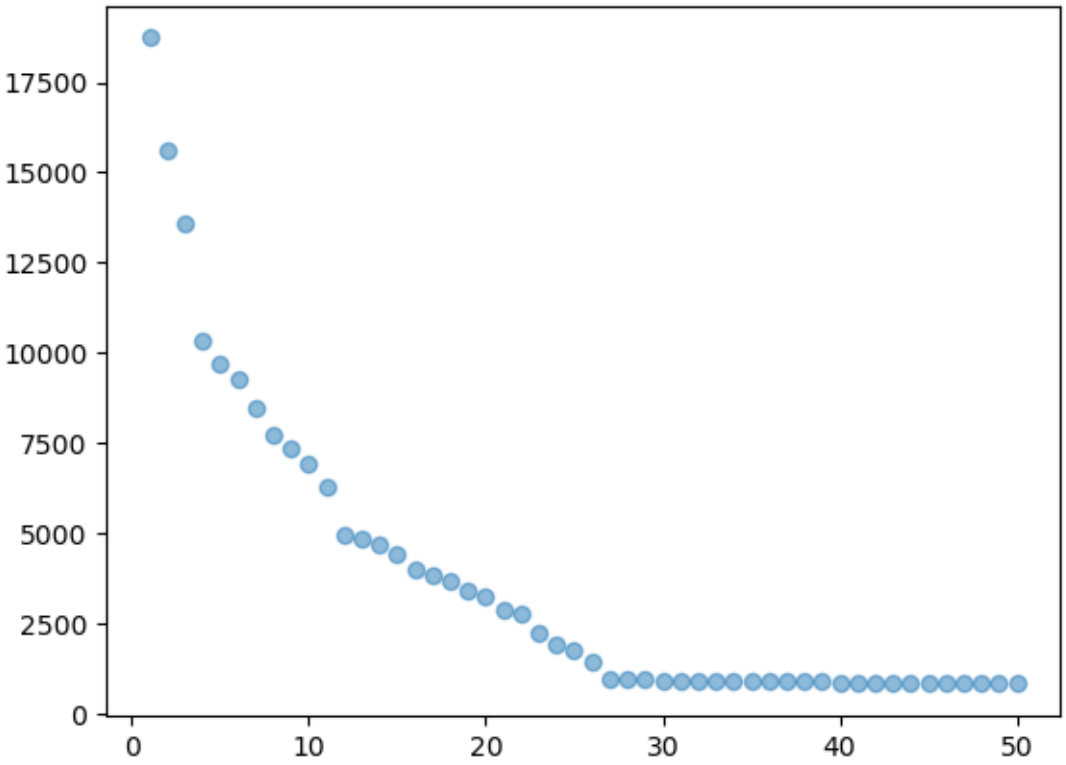
[74]:
```
import random
import string
from string import ascii_lowercase

char_count = 0
text_chunk = ""

# Generating 1 million characters worth of "words"
while char_count < 1000000:

    # Character options are lowercase letters plus whitespace character
    chars = string.ascii_lowercase + " "

    # Generates the "word" by joining 8 random characters from the given options
    # Since it is possible that one of the random characters is a space, not␣
 ↪all words will be 8 characters
```

```
    word = "".join(random.choice(chars) for _ in range(8))

    # In the case that there are no spaces included in the "word" (it is one␣
 ↪word, not more) add a space on the end
    # This will ensure that consecutive "words" wont run together to create a␣
 ↪larger word
    if " " not in word:
        word = word + " "

    char_count += len(word)
    text_chunk += word

#print(text_chunk)
```

[75]:
```
import tiktoken

# To get the tokeniser corresponding to a specific model in the OpenAI API:
enc = tiktoken.encoding_for_model("gpt-3.5-turbo")

bonus_vocab = {}
tokens = [enc.decode_single_token_bytes(token) for token in enc.
 ↪encode(text_chunk)]

for t in tokens:
    t = t.strip()
    if t.isalpha():

        if t in bonus_vocab.keys():
            bonus_vocab[t] += 1
        elif t not in bonus_vocab.keys():
            bonus_vocab[t] = 1


print('There are', len(bonus_vocab), 'unique tokens in the text.')
print('There are', sum(bonus_vocab.values()), 'tokens in the text.')
```

```
There are 5721 unique tokens in the text.
There are 499828 tokens in the text.
```

[76]:
```
bonus_ranked = [(k, v) for k, v in bonus_vocab.items()]
bonus_ranked.sort(key=lambda x: x[1], reverse=True)

import matplotlib.pyplot as plt
bonus_ranks = range(1, 50 + 1)
bonus_freqs = [t[1] for t in bonus_ranked[:50]]
plt.scatter(bonus_ranks, bonus_freqs, c='#1f77b4', alpha=0.5)
plt.show()
```


17500
15000
12500
10000
7500
5000
2500
0
0
10
20
30
40
50

[ ]: