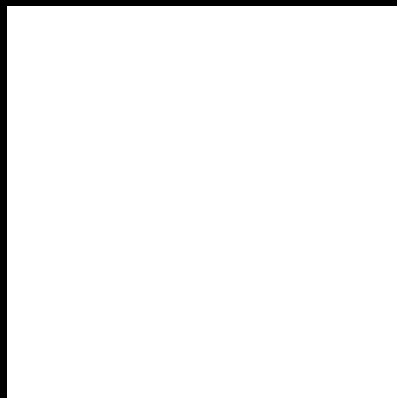


Session_11: Intro to NLP

Credit: Terence Broad



What is natural language processing?



Natural language processing (NLP) is a subfield of computer science and artificial intelligence (AI) that uses machine learning to enable computers to understand and communicate with human language.

NLP enables computers and digital devices to recognize, understand and generate text and speech by combining computational linguistics—the rule-based modeling of human language—together with statistical modeling, machine learning (ML) and deep learning.

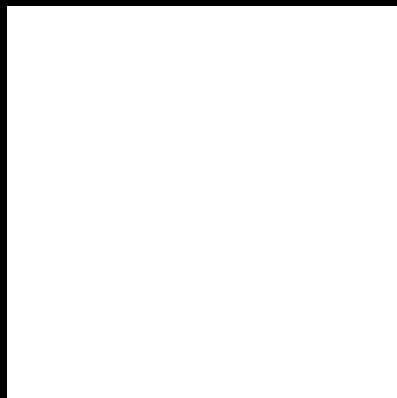
NLP research has enabled the era of generative AI, from the communication skills of large language models (LLMs) (eg. ChatGPT) to the ability of image generation models to understand requests.

NLP is already part of everyday life for many, powering search engines, prompting chatbots for customer service with spoken commands, voice-operated GPS systems and digital assistants on smartphones.

Lets try one now!!! <https://deepai.org/machine-learning-model/text2img>



What is natural language processing?



Natural language is human communication

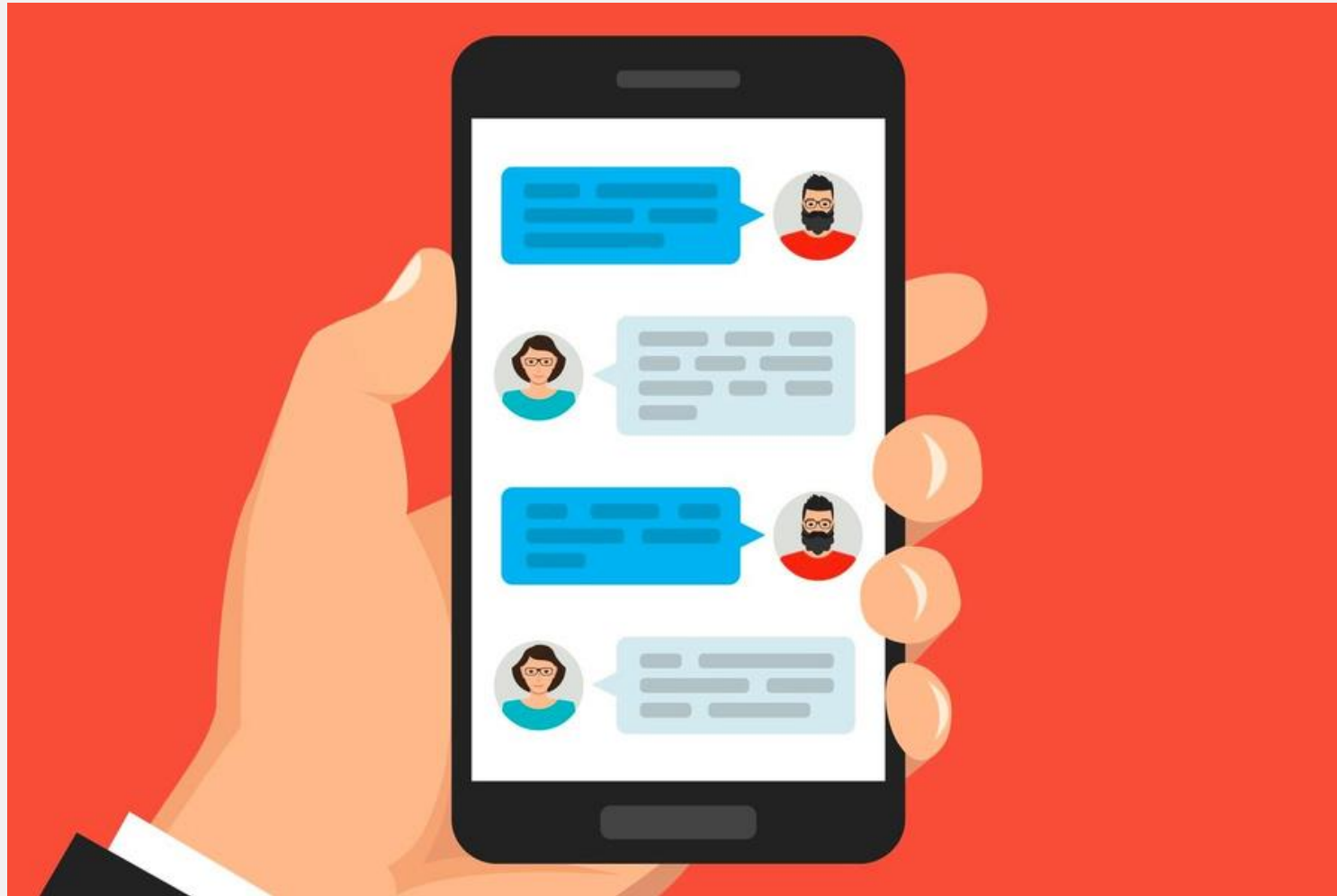


Spoken languages are natural language



Sign languages are natural language

Handwritten text is natural language

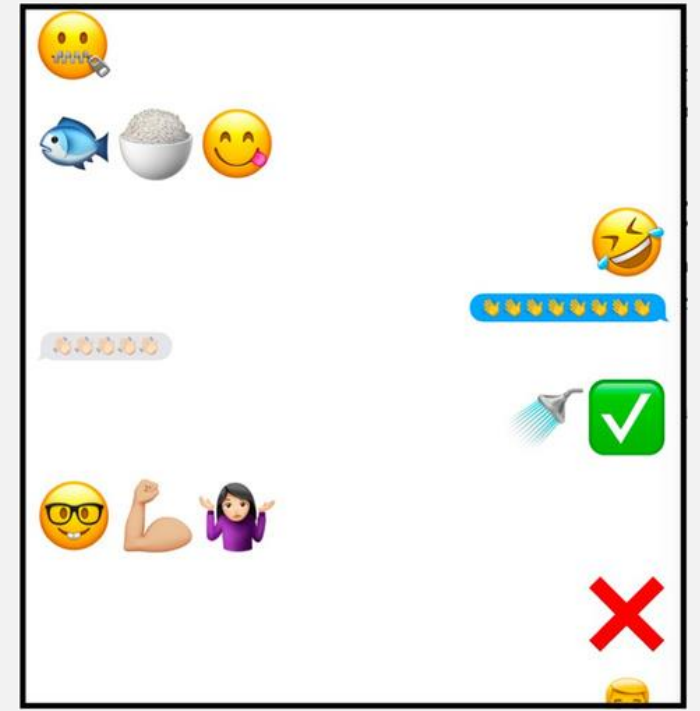
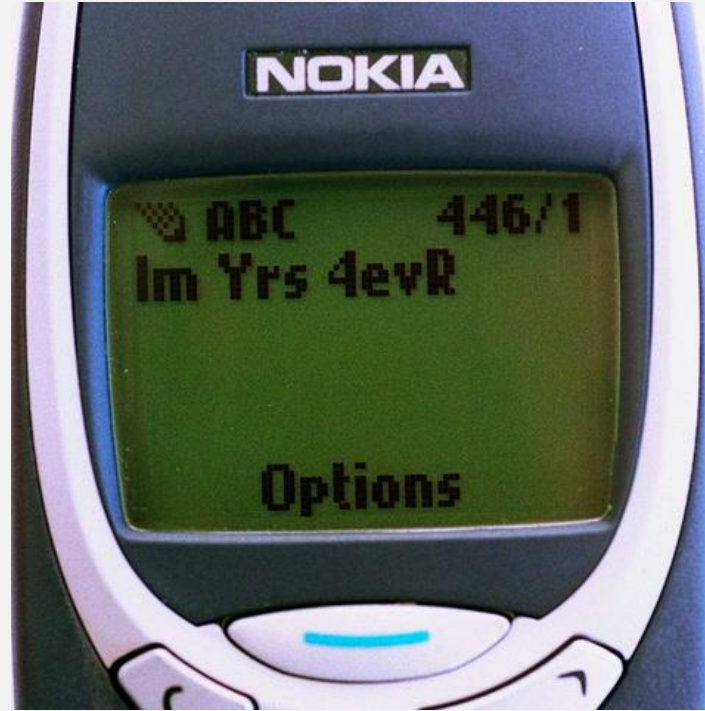
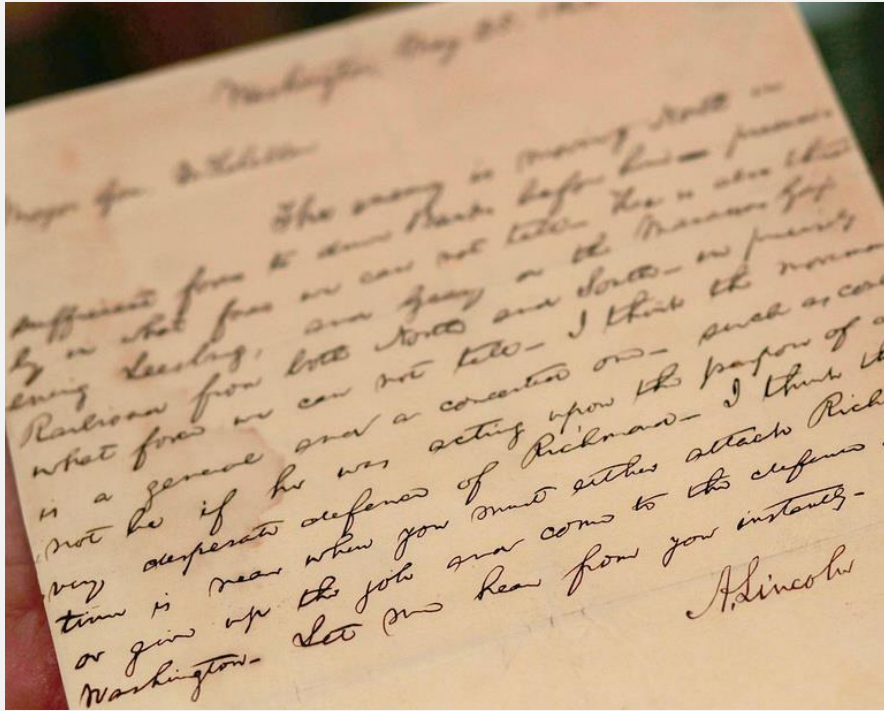


Digital text is natural language

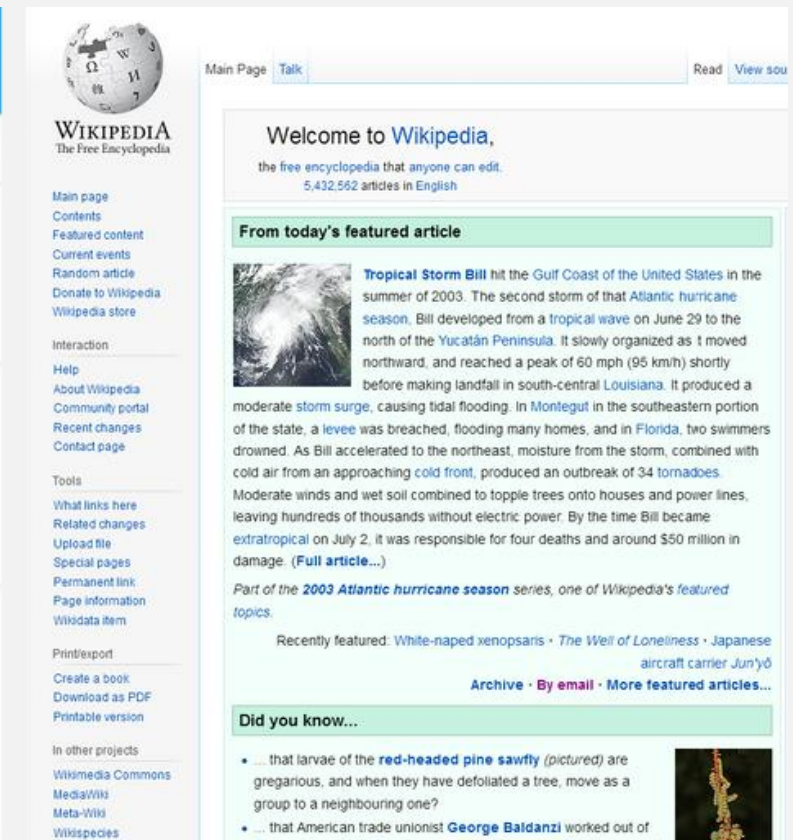
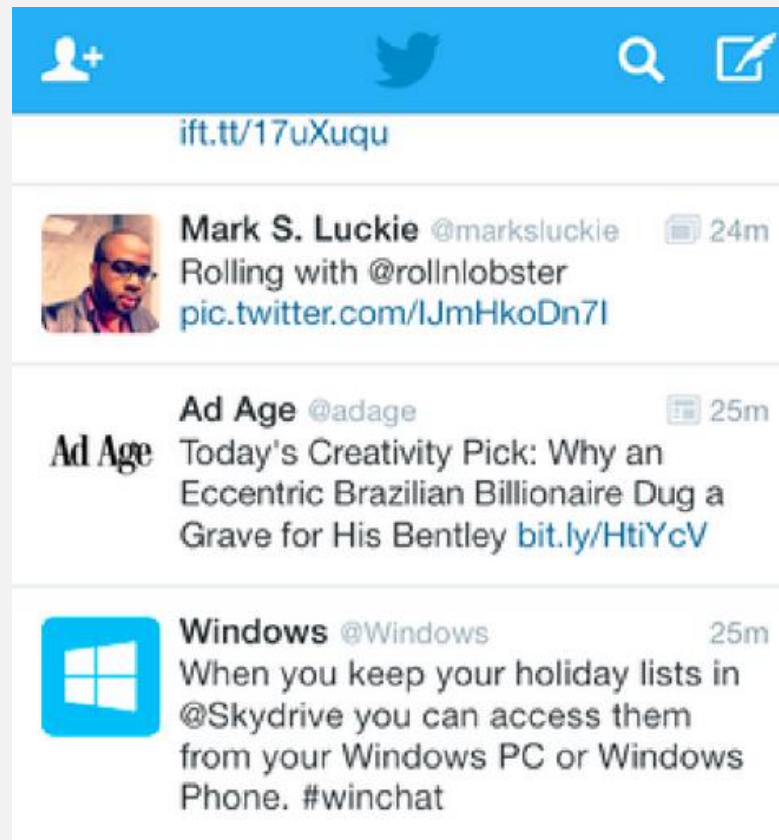
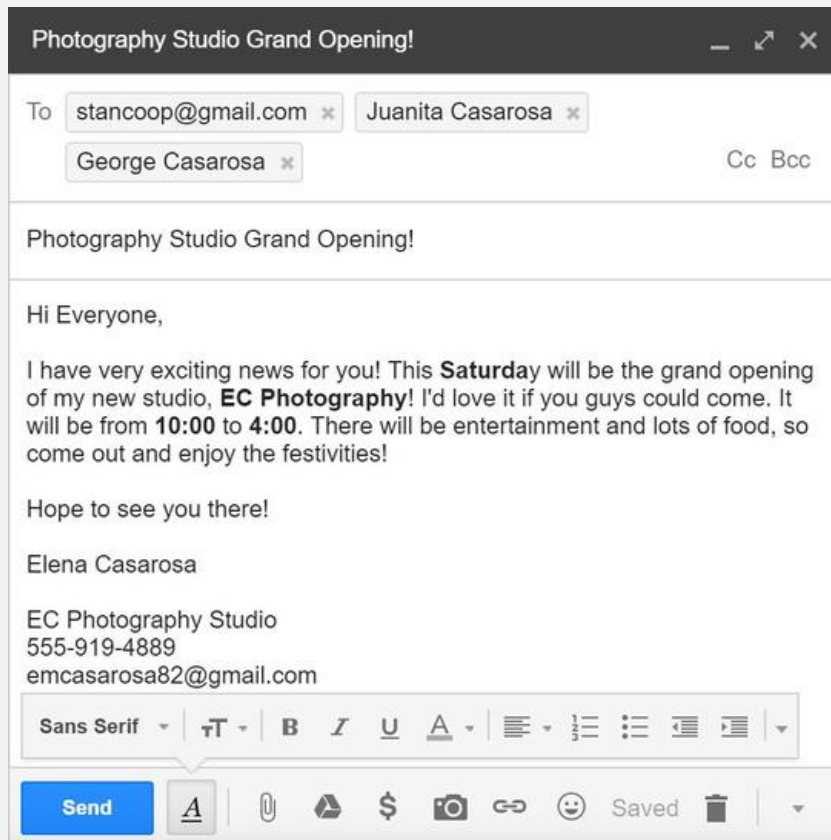


Emojis are natural language

Natural language can take many forms and can be represented in many different ways



The means by which we represent language and communicate using it has a massive impact on how that language is used

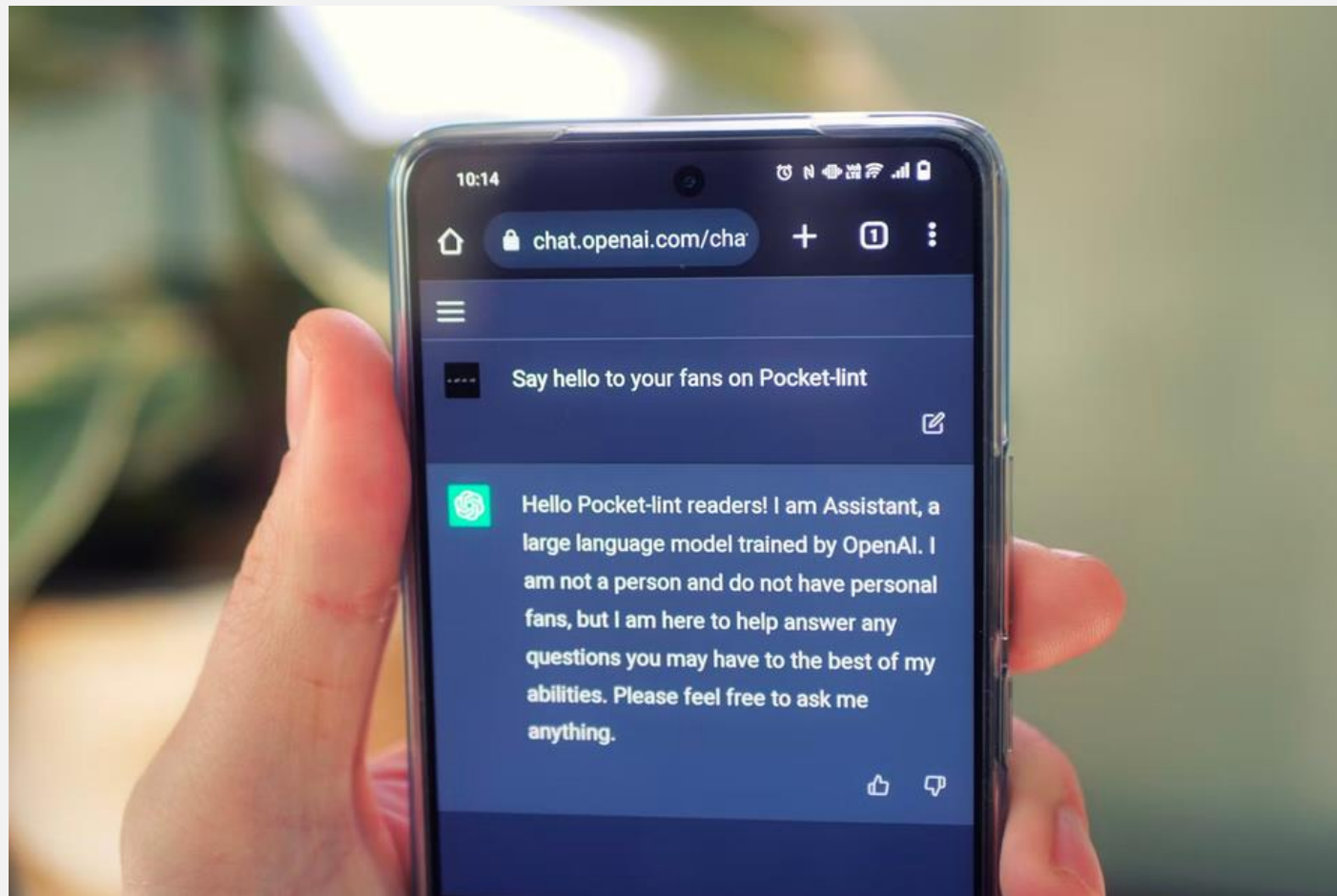


A huge amount of written communication is now mediated by digital computers (and the internet)



Processing and understanding this data from global internet communications has led to many of the most important breakthroughs in AI

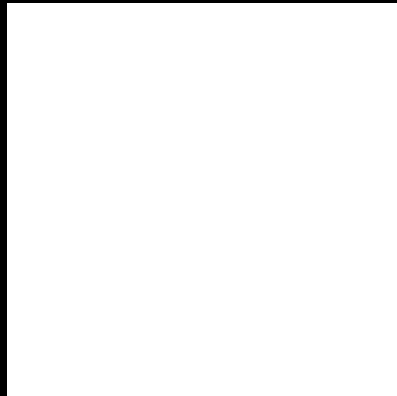




AI language models like ChatGPT are so successful because we have so much data from internet communication to train them



Key things to consider on this unit



```

cook@pop-os:~$ ascii -d
 0 NUL    16 DLE    32      48 0      64 @      80 P      96 `     112 p
 1 SOH    17 DC1    33 !     49 1      65 A      81 Q      97 a     113 q
 2 STX    18 DC2    34 "     50 2      66 B      82 R      98 b     114 r
 3 ETX    19 DC3    35 #     51 3      67 C      83 S      99 c     115 s
 4 EOT    20 DC4    36 $     52 4      68 D      84 T     100 d     116 t
 5 ENQ    21 NAK    37 %     53 5      69 E      85 U     101 e     117 u
 6 ACK    22 SYN    38 &     54 6      70 F      86 V     102 f     118 v
 7 BEL    23 ETB    39 '     55 7      71 G      87 W     103 g     119 w
 8 BS     24 CAN    40 (     56 8      72 H      88 X     104 h     120 x
 9 HT     25 EM     41 )     57 9      73 I      89 Y     105 i     121 y
10 LF     26 SUB    42 *     58 :     74 J      90 Z     106 j     122 z
11 VT     27 ESC    43 +     59 ;     75 K      91 [     107 k     123 {
12 FF     28 FS     44 ,     60 <     76 L      92 \     108 l     124 |
13 CR     29 GS     45 -     61 =     77 M      93 ]     109 m     125 }
14 SO     30 RS     46 .     62 >     78 N      94 ^     110 n     126 ~
15 SI     31 US     47 /     63 ?     79 O      95 _     111 o     127 DEL

```

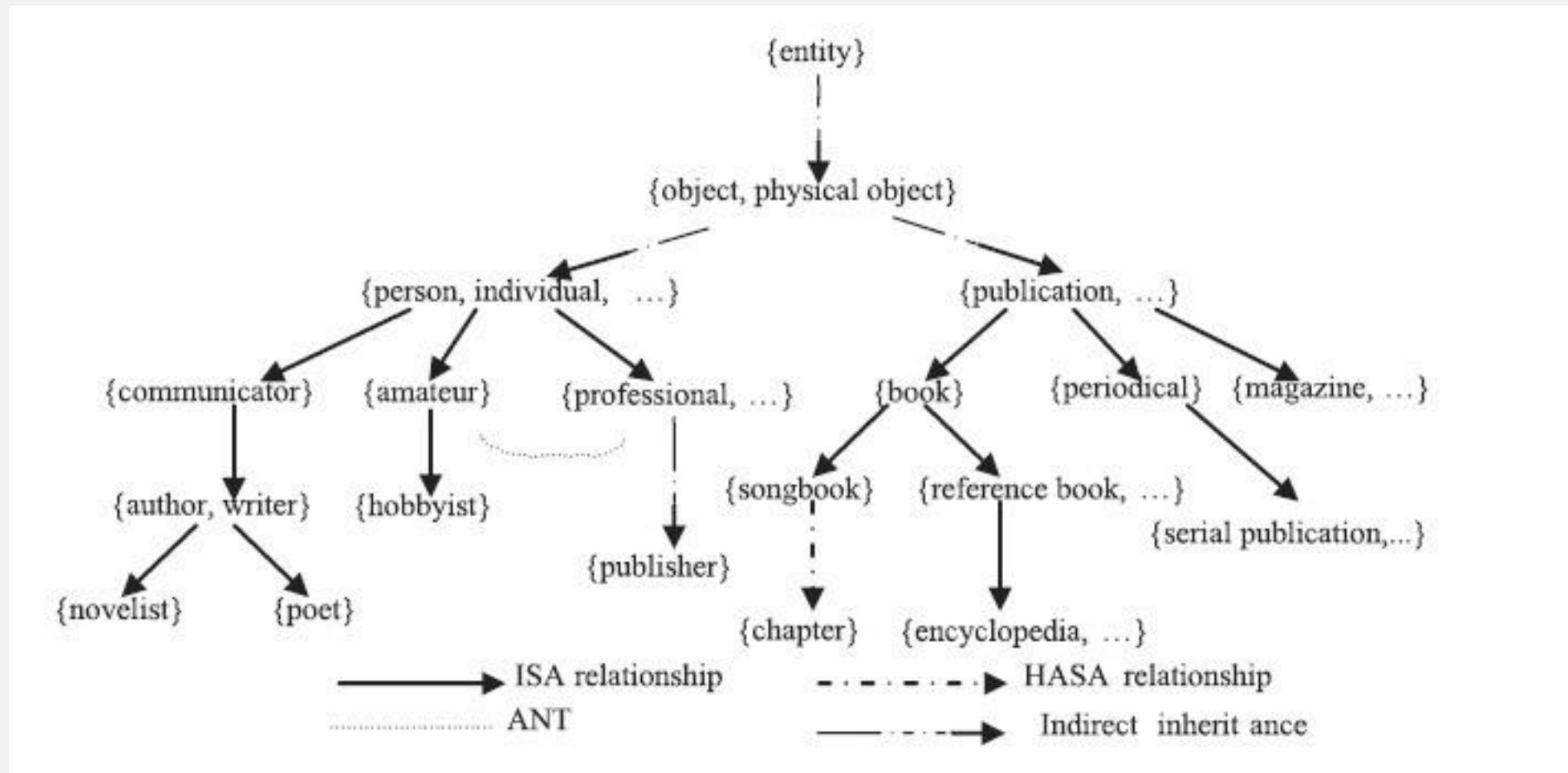
How computers represent text

Lucy said that the car's engine was running all night

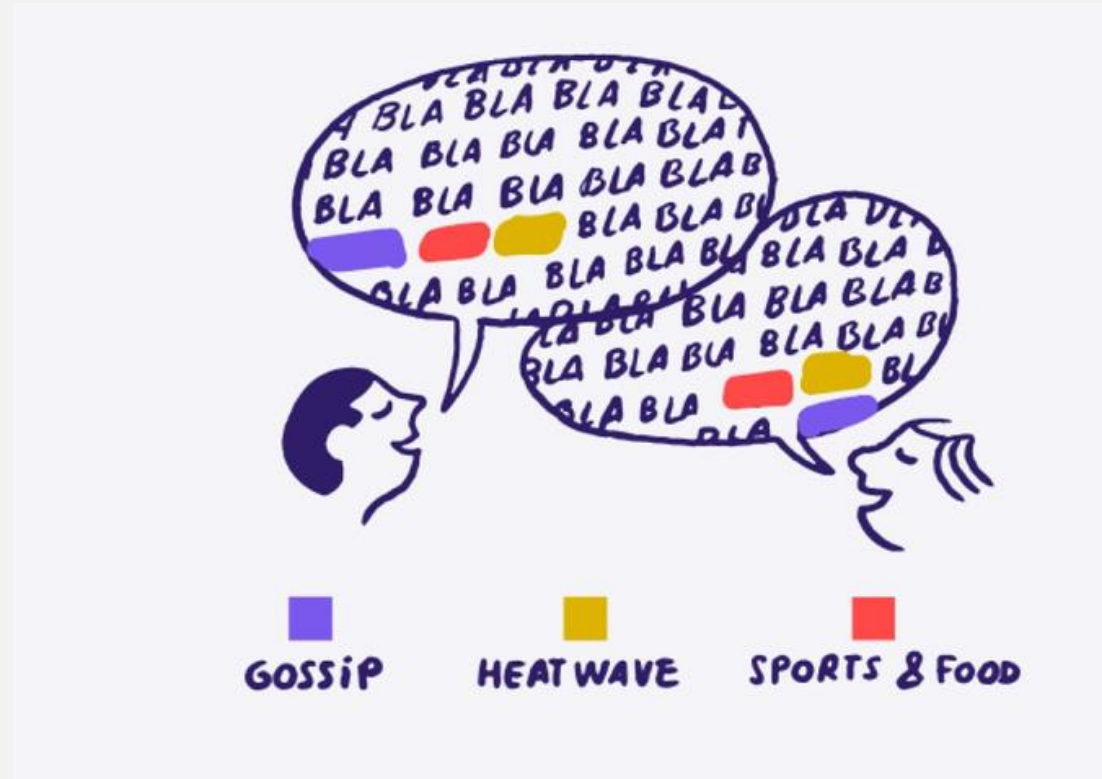


Lucy say that the car engine be run all night

How text can be manipulated and simplified



How the meaning of words can be represented



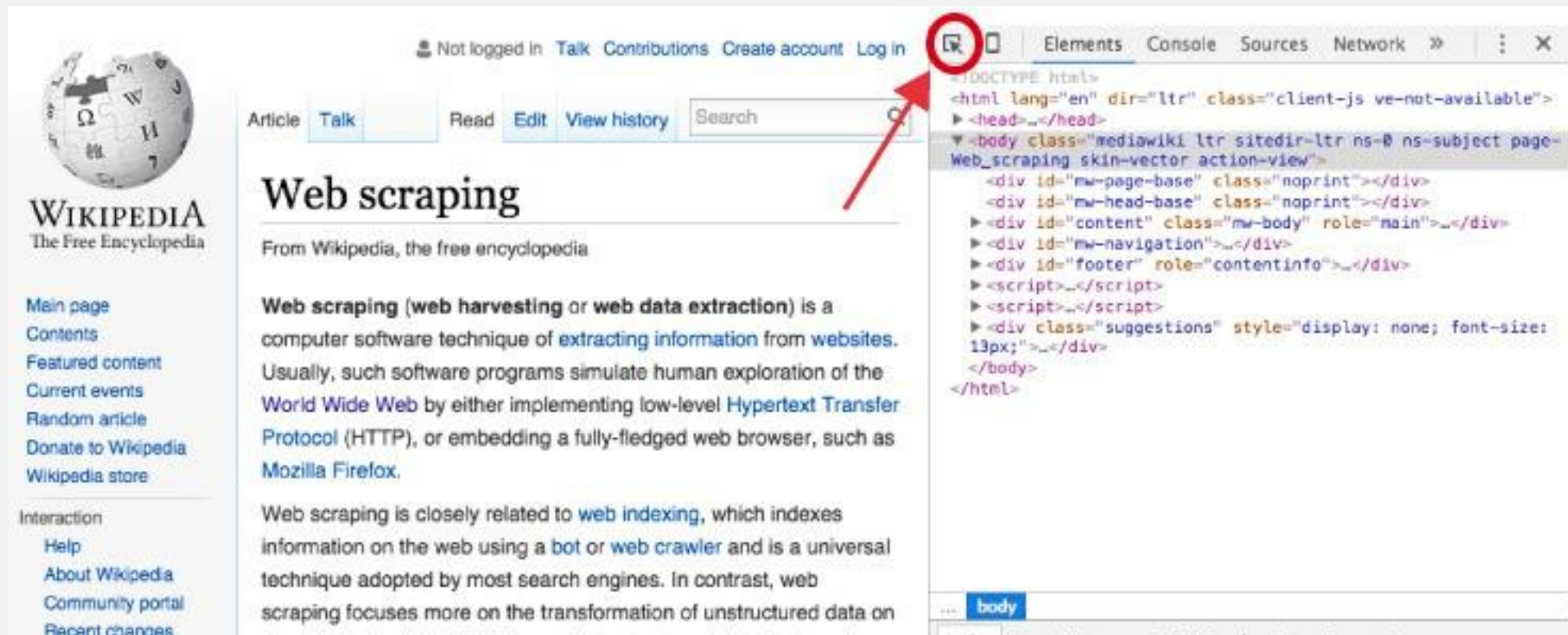
How to figure out what the topics of texts are



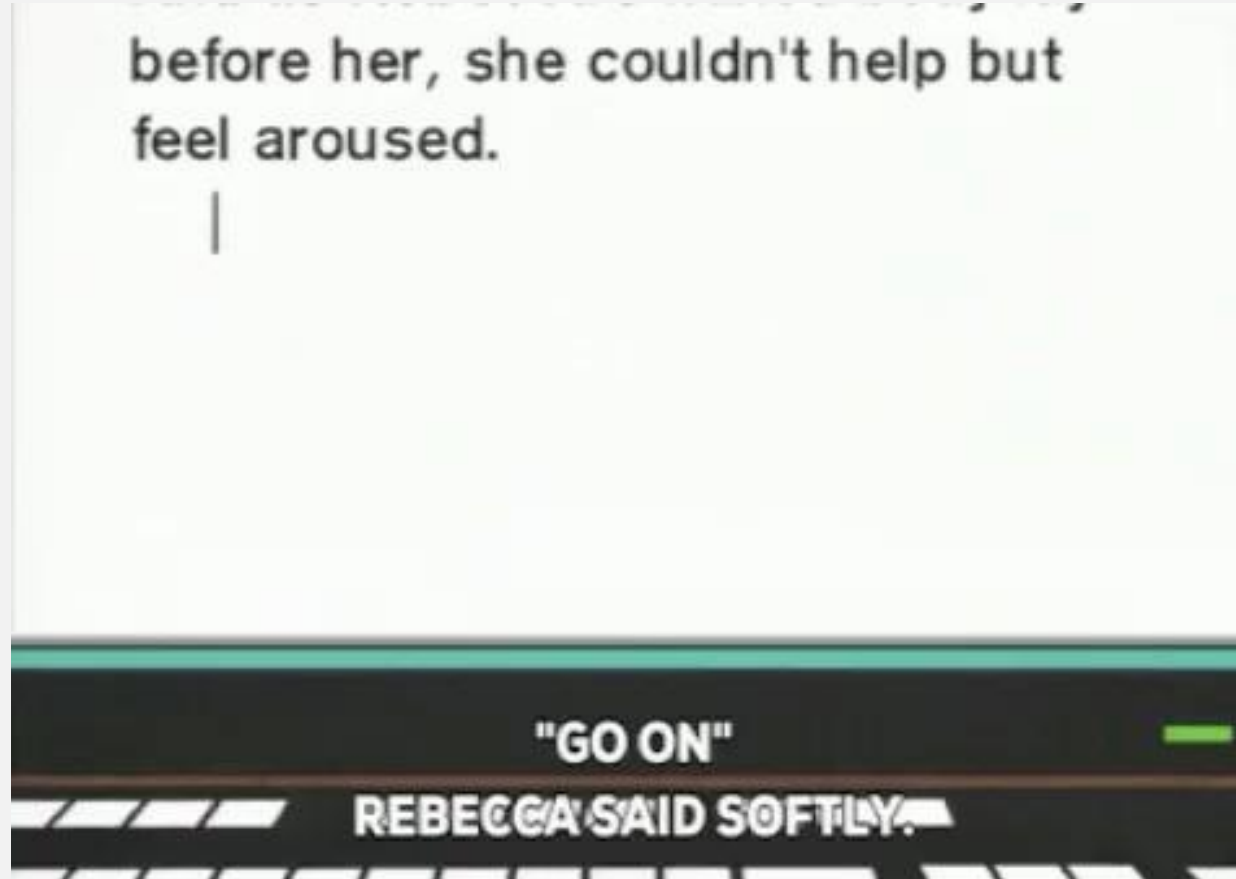
How to classify text into categories



How to figure out the sentiment of a text



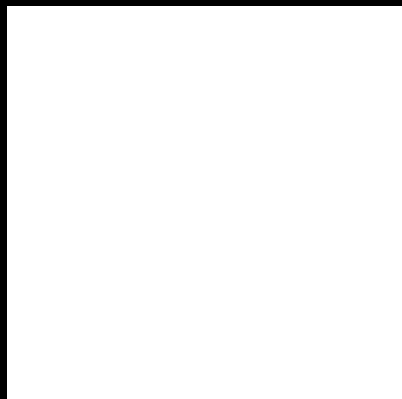
how to scrape text from the internet (you know this!)



How to generate text with AI

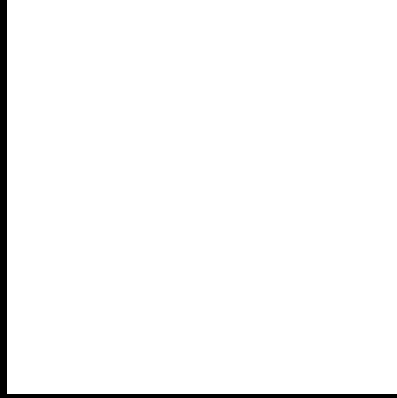


Summary

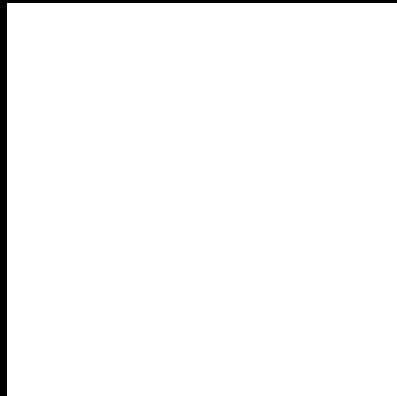


Why are we studying NLP?

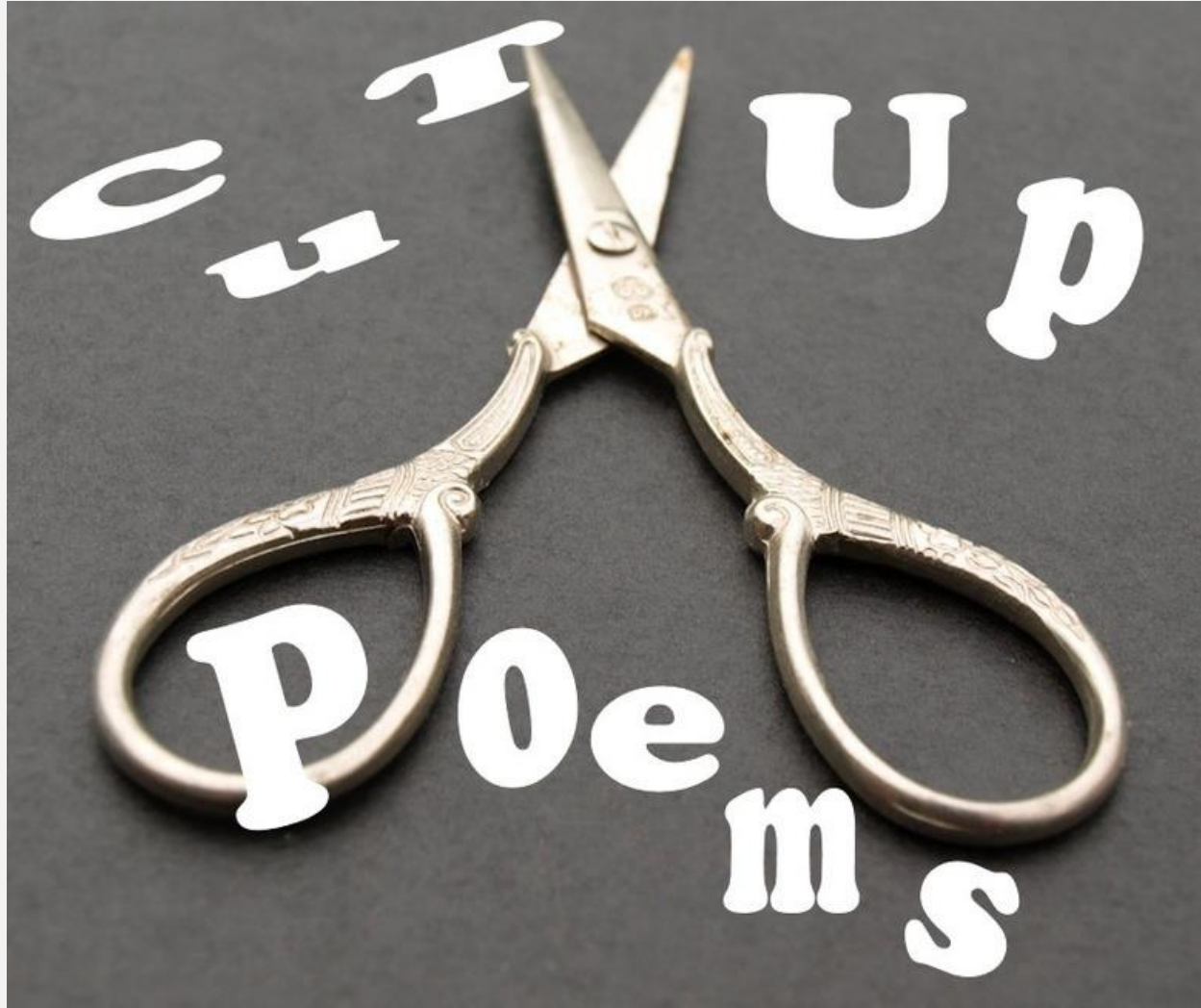
This is a very exciting time to be studying NLP! The most important breakthroughs in AI are currently happening in the NLP space

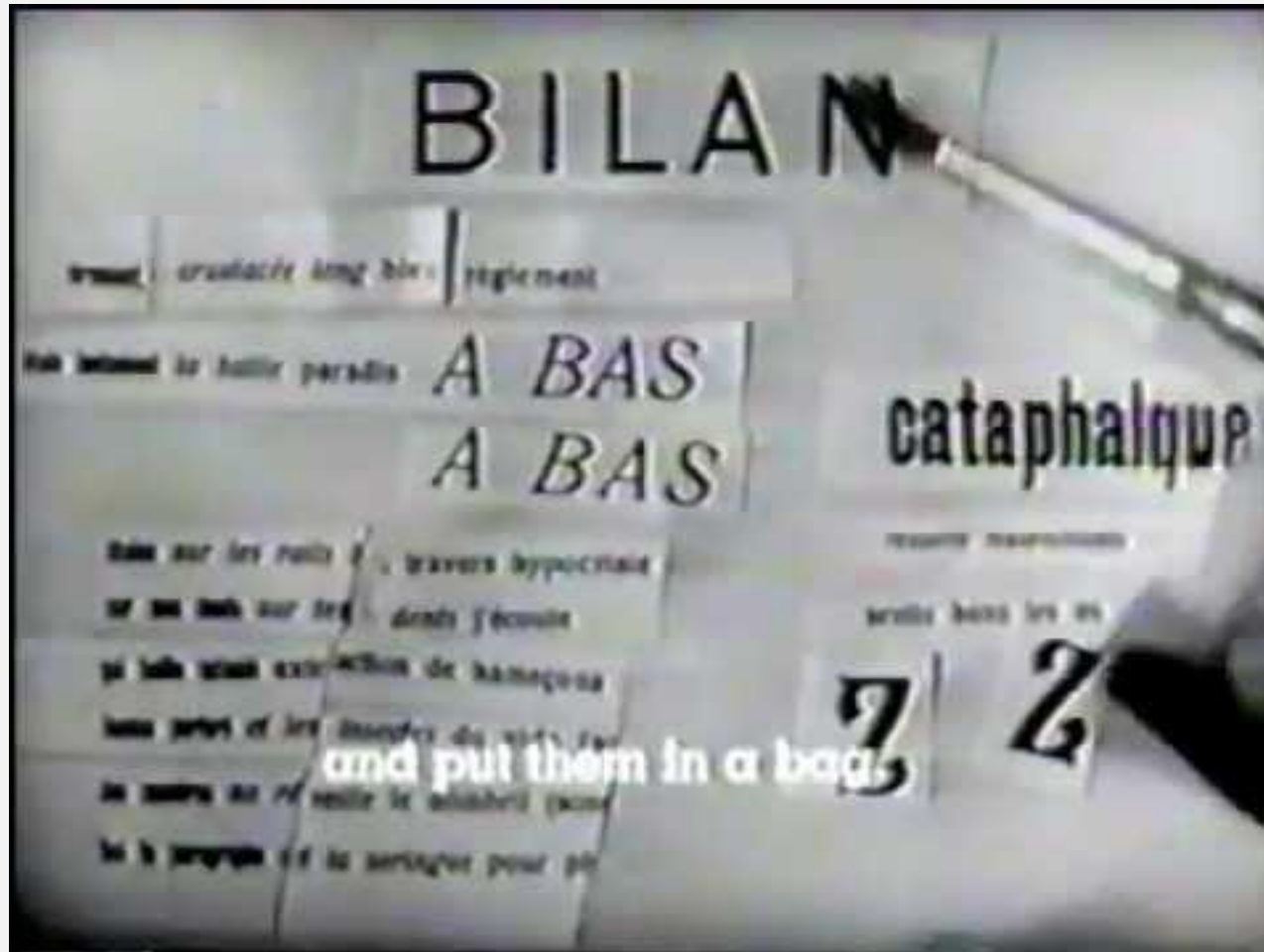


Making cut-up poetry (with paper and scissors)

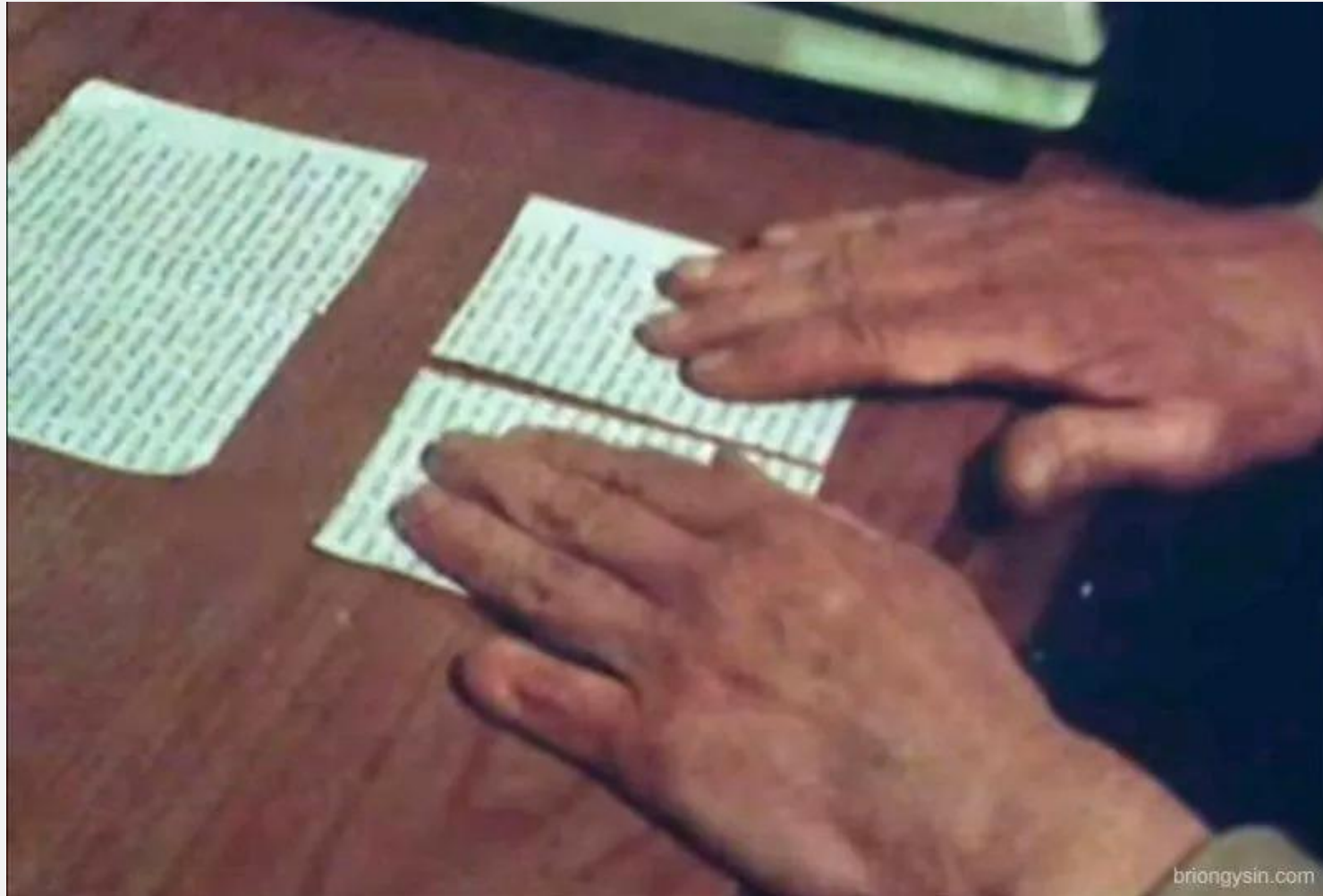


We are going to make

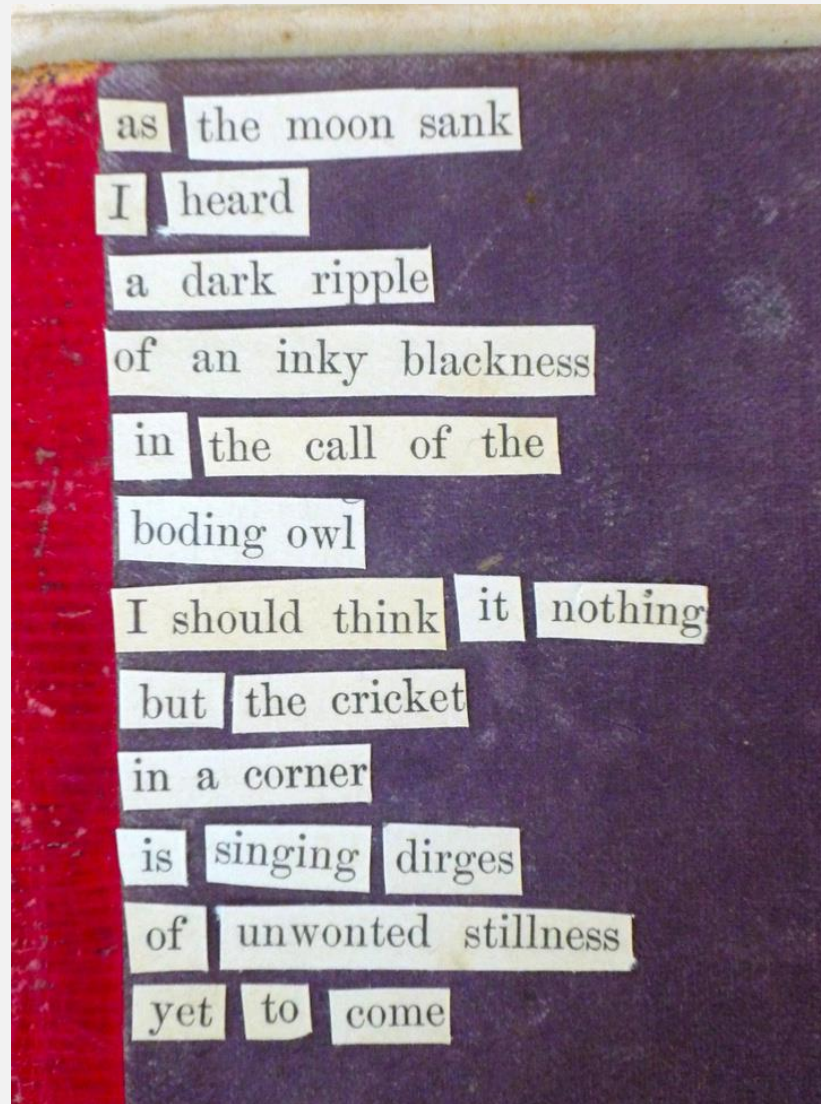




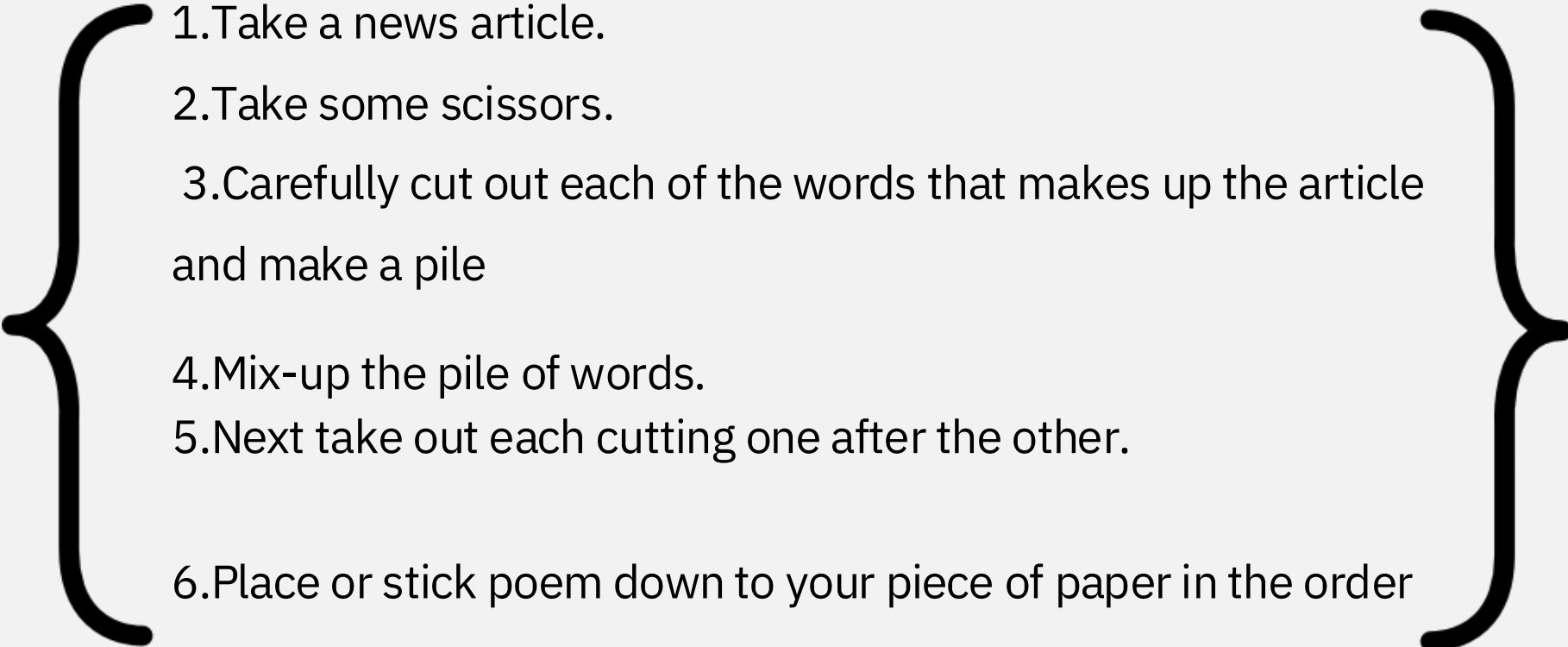
The "cut-up" technique was developed during the Dada art movement in the 1920's



And was popularised by the author William Burroughs in the 1950"s and 60"s



David Bowies cut up lyrics for the song
"Blackout"

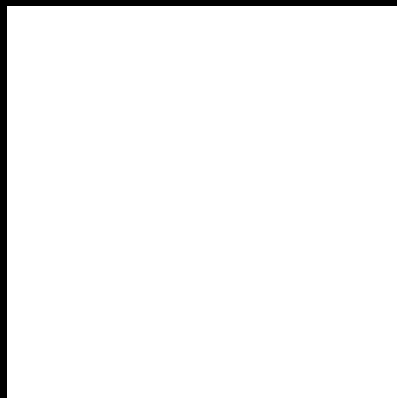
- 
1. Take a news article.
 2. Take some scissors.
 3. Carefully cut out each of the words that makes up the article and make a pile
 4. Mix-up the pile of words.
 5. Next take out each cutting one after the other.
 6. Place or stick poem down to your piece of paper in the order you

drew the words.

This is an
algorithm

What is an algorithm?

- An algorithm is a set of instructions for completing a task or solving a problem
- Algorithms can be instructions for humans or for computers
- When we write in coding languages, we are giving instructions for the computer follow
- Now lets see our algorithm as it is written for a computer



Making cut-up poetry (with code)



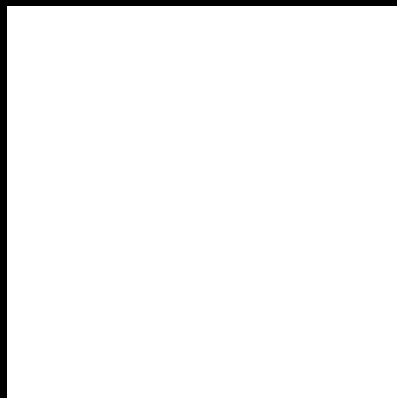
How to make a cut-up poem (with code)

1. Take a text variable (aka a string)
2. Split the text string wherever there is whitespace into a list of words
3. Make an empty text variable called *poem*
4. Randomly pick a new word from the list and append it to the poem
5. Append a whitespace character to the poem after the word
6. Stop after a certain number of words have been added to the poem

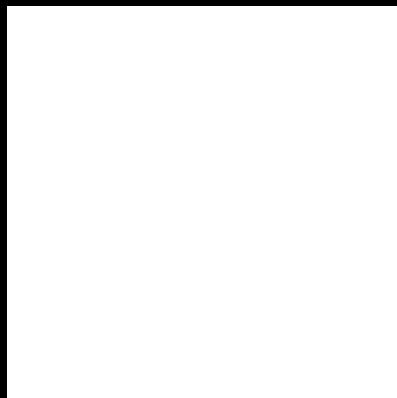
Instructions for code activity

1. Grab the cut-upmachine.ipynb notebook (in the git repo)
2. Find a news article from the web and copy the text in
3. Run the code to make new cut up poems
4. Share your poems with the rest of the class!

This will be task 1 of the labs! But we have a lot more to do for now.....



Representing Documents as numbers



If we want to take a document or set of documents, and use them with machine learning techniques, or other mathematical operation to uncover some new information about them, we can't just use the text and characters.

We need a new representation, and that needs to be numerical. Here we will be taking our first steps to representing collections of documents as numbers.

- Tokenisation
- Bags of Words
- n-Grams

Building a Vocabulary

The first step to getting a new, better representation of a text document is splitting it up into its constituent parts. We call these **tokens** and deciding "*what a token is*" is an important choice.

Basic Tokenisation - `str.split()`

What is the simplest way split a **`**String**`** into **`**tokens**`**?

Introducing the "**`str.split()`**" function. Here we take a long string (multiple words) and split it into separate words (or **`**tokens**`**) based on spaces.

```
sentence = "I'm learning new things every day. Day is nice."  
  
tokens = sentence.split()  
  
print(tokens)
```

What gets returned is a **List**, containing our split string. We store it in the variable `tokens`, and print it out:

```
["I'm", "learning", "new", "things", "every", "day.", "Day", "is", "nice."]
```

How good is our vocabulary?

We have made a vocabulary from, (or **tokenised**) our text document by splitting every time we see a space.

Does this seem sensible? Does this capture everything that we would consider a separate word in the document?

What about "isn't"? Is this one token ("isn't"), or two tokens ("is" and "not")?
Or "taxi cab"?

Is that two tokens, do we care that "taxi" and "cab" are both used? Do we need this as a separate concept from "Uber", "limo", "Hackney Carriage" or "car"?.

If we take it as one, do we miss out on other combinations like "black cab" or "taxi driver"? What about punctuation?

We will try to solve this later....

There are many ways we can improve our vocabulary after we split it. For example we can use numpy to remove duplicates

```
import numpy as np

sentence = "I like to think (it has to be!) of a cybernetic ecology  
where we are free of our labors"

tokenised = sentence.split()

vocab = np.unique(tokenised)

print(vocab)
print(len(vocab))
```

Vocabulary (unique tokens):

```
['(it' 'I' 'a' 'are' 'be!)' 'cybernetic' 'ecology' 'free' 'has' 'labors' 'like' 'of'  
'our' 'think' 'to' 'we' 'where'] 17
```

One-Hot Encoding

Now that we have derived a **vocabulary** (if not exactly perfect yet) for the sentence, we can represent it as a set of numbers.

To create a **one-hot encoding**, we assign **each token in a document** a vector that is the length of the vocabulary, with each slot in this vector representing a token in the vocabulary. These slots can either be **1 or 0**.

For the slot that represents that token in the vocabulary, we set to 1. Every other slot is 0.

This leaves us with a **2-d List (or Dataframe as we've seen before)**, where each row is a list as long as the vocabulary. Each row only ever has a single 1 in it.

[illegible]

```
#Split into tokens based on spaces
tokenised = sentence.split()

#Get the unique tokens
vocab = np.unique(tokenised)

#Make a matrix of zeros using the lengths of the separated sentence and vocab
one_hot = np.zeros((len(tokenised), len(vocab)))

#Go through the separated sentence and set the appropriate item to 1
for i in range(len(tokenised)):
    #Get the word
    word = tokenised[i]
    #find the index of the word in the vocab
    match = np.where(vocab == word)
    #Set it to 1 (hot)
    one_hot[i, match] = 1

print(pd.DataFrame(one_hot, columns = vocab))
```


This **one-hot encoding** doesn't lose any information. We keep a reference to every token, as well as the sequence in which they appear. As we have seen, small differences and nuance in natural language can have big effects in meaning.

But its a lot of numbers for a small amount of information. This being said, it is also super **sparse**, which just means there are lots of 0s, and there are actually lots of techniques for really efficiently storing sparse data.

We've successfully represented our sentence as a matrix of numbers, which we can use with various mathematical techniques moving forwards.

Bag of Words

Using **one-hot encoding**, we have a **vector** the length of the vocabulary for every word. However, this can quickly get out of hand with longer documents and bigger vocabularies.

One improvement we can make to this representation is to simply count up every occurrence of each word in the vocabulary for each document, and then store this count for each word in the vocabulary. This is what we call a bag of words, in which we represent a document by its words and the frequency at which they appear.

This means we only have one **word frequency vector** for each document, rather than a one-hot encoding vector for each word. If we had multiple documents (or sentences), we could make a **word frequency vector** for each one and store them as a **matrix** (2D array).

That was a very complicated way of saying this...

```
#Use a Counter to create a Bag of Words (word-frequency vector)
from collections import Counter
bow = Counter(tokenised)
print(bow)
```

```
Counter({'to': 2,
        'of': 2,
        'I': 1,
        'like': 1,
        'think': 1,
        '(it': 1,
        'has': 1,
        'be!)': 1,
        'a': 1,
        'cybernetic': 1,
        'ecology': 1,
        'where': 1,
        'we': 1,
        'are': 1,
        'free': 1,
        'our': 1,
        'labors': 1})
```

Working with large documents (eg. Books)

What can we find out about each chapter by counting the amount of times a word appears in each? Are any similar to each other? How can we adjust our vocabulary to help us out? First we're going to look at the book as a whole.

We're going to use a number of techniques to try and tweak our vocabulary so that it contains the most information for when we start using this bag of words in tasks like topic modelling or classification (you will learn what this means in future classes).

This means we want to count things that have the same meaning as the same token, but we also don't want to throw away any information that might help our processing.

```
fs = open('C:\\Users\\James Gibbons-MacGre\\Downloads\\hacking.txt', 'r')
book = fs.read()

#tokens is a 1D array
tokens = book.split()
#Get the unique tokens (our vocabulary)
vocab = np.unique(tokens)
print("total words:", len(tokens), "unique words:", len(vocab))
#Create a Bag of Words using a Counter
Counter(tokens).most_common(20)
```

```
[('the', 8603), ('to', 4314), ('a', 3775), ('of', 3622), ('and', 2877), ('was',
2349), ('in', 2260), ('he', 1944), ('had', 1724), ('his', 1533), ('The', 1392),
('on', 1249), ('for', 1158), ('it', 1018), ('that', 974), ('with', 920), ('He',
845), ('at', 811), ('as', 725), ('from', 708), ('would', 641), ('be', 631),
('were', 629), ('they', 620), ('computer', 613)]
```

This is what you'd expect the most common words in any book to be... but it's not very useful...

Stop words

We can see that the commonly occurring words don't tell us much about this specific book. Traditionally, in NLP it has been useful to remove words that occur commonly. They don't tell us very much about each document because they are contained in almost all the documents. These are known as **stop words**. Examples: the, a, she, he, his, her, to, was...

We'll just see how removing stop words from our bag effects what we can see. Although our vocabulary size is basically the same (so we're not saving much in efficiency), the list most common words are much more informative and tells us more about the specific book we're looking at.

```
!pip install scikit-learn
```

```
from sklearn.feature_extraction import _stop_words # stop words are in lower case
tokens_without_stop_words = []
for t in tokens:
    if not t in _stop_words.ENGLISH_STOP_WORDS:
        tokens_without_stop_words.append(t)
stop_vocab = np.unique(tokens_without_stop_words)
print("unique words", len(stop_vocab))
Counter(tokens_without_stop_words).most_common(20)
```

```
[('The', 1401), ('He', 845), ('"', 824), ('computer', 742), ('Par', 498), ('It',
496), ('hackers', 399), ('Electron', 382), ('didn't', 374), ('Phoenix', 360),
('time', 349), ('Anthrax', 337), ('just', 334), ('like', 333), ('I', 321),
('Mendax', 306), ('hacking', 293), ('They', 287), ('worm', 283), ('hacker', 282),
('police', 275), ('phone', 268), ('people', 251), ('In', 237), ('said', 217)]
```

This now tells us much more about the subject of the book. Note the stop words excluded here are the ones set by default in the .sklearn library

Stemming

Stemming attempt to remove suffixes from words that contain the same base. This reduces variation and can help when we reduce the documents into a more distilled form (like a bag of words).

- hacking, hackers, hacked, hacks
- computer, computing, computers

Depending on our task, it might be the case that we only really care about knowing if **any** of these words appear, not whether they each appear individually. For example, I might be doing a search for paragraphs about hacking and it may be that I would miss out on key documents otherwise if I only searched for one of the words.

Stemming can be quite a challenging task however. If we want to combine pluralisations, for example, we can't just remove the "s" from the end of all nouns, what about

- grass (not a plural)
- mice, octopi (plural, no s)
- geniuses (plural, es)

We're going to use the built-in stemmer in the NLTK library. This reduces our vocabulary in the hacking book dramatically!

```
%pip install nltk
```

```
from nltk.stem.porter import PorterStemmer  
stemmer = PorterStemmer()  
word_list = ['feet', 'foot', 'foots', 'footing']  
for word in word_list:  
    print(word, "->", stemmer.stem(word))
```

#Doesn't always work

```
word_list =  
['organise', 'organises', 'organised', 'organisation', "organs", "organ", "organic"]  
for word in word_list:  
    print(word, "->", stemmer.stem(word))
```

This more or less works....

```
feet -> feet  
foot -> foot  
foots -> foot  
footing -> foot
```

This doesn't work, meaning is lost

```
organise -> organis  
organises -> organis  
organised -> organis  
organisation -> organis  
organs -> organ  
organ -> organ  
organic -> organ
```

Lemmatisation

Lemmatisation is a technique similar to stemming, apart from it attempts to find similar meanings, as opposed to just similar roots.

Like with all these *_normalisation_* techniques, reducing your vocabulary will reduce precision but may make your model better at generalising and more efficient.

For example, lemmatisation would be able to separate **logged** and **dog**, which have quite different meanings but would get combined by a stemmer.

Lemmatisation

We can use this to find the closest alternative word with the same meaning to the original. We can specify nouns, verbs or adjectives.

- Nouns are people, places, or things.
- Verbs are action words.
- Adjectives are descriptive words.

```
import nltk
nltk.download("wordnet")
nltk.download('omw-1.4')
from nltk.stem import WordNetLemmatizer
lem = WordNetLemmatizer()

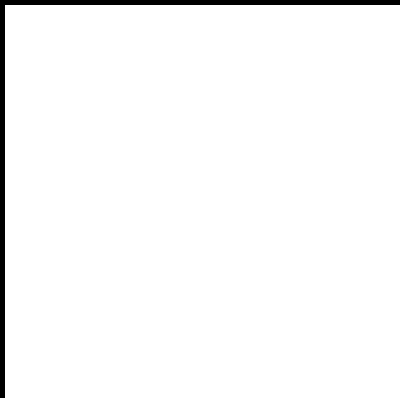
# the `pos` (part-of-speech/grammatical function) tag

print(lem.lemmatize("better")) # pos is by default 'n' --> it will try to find the closest noun
which might not be ideal

print(lem.lemmatize("better", pos="v")) # let's try to find the closest verb --> fails too

print(lem.lemmatize("better", pos="a")) # returns 'good' which is a suitable adjective
```

better
better
good



End

