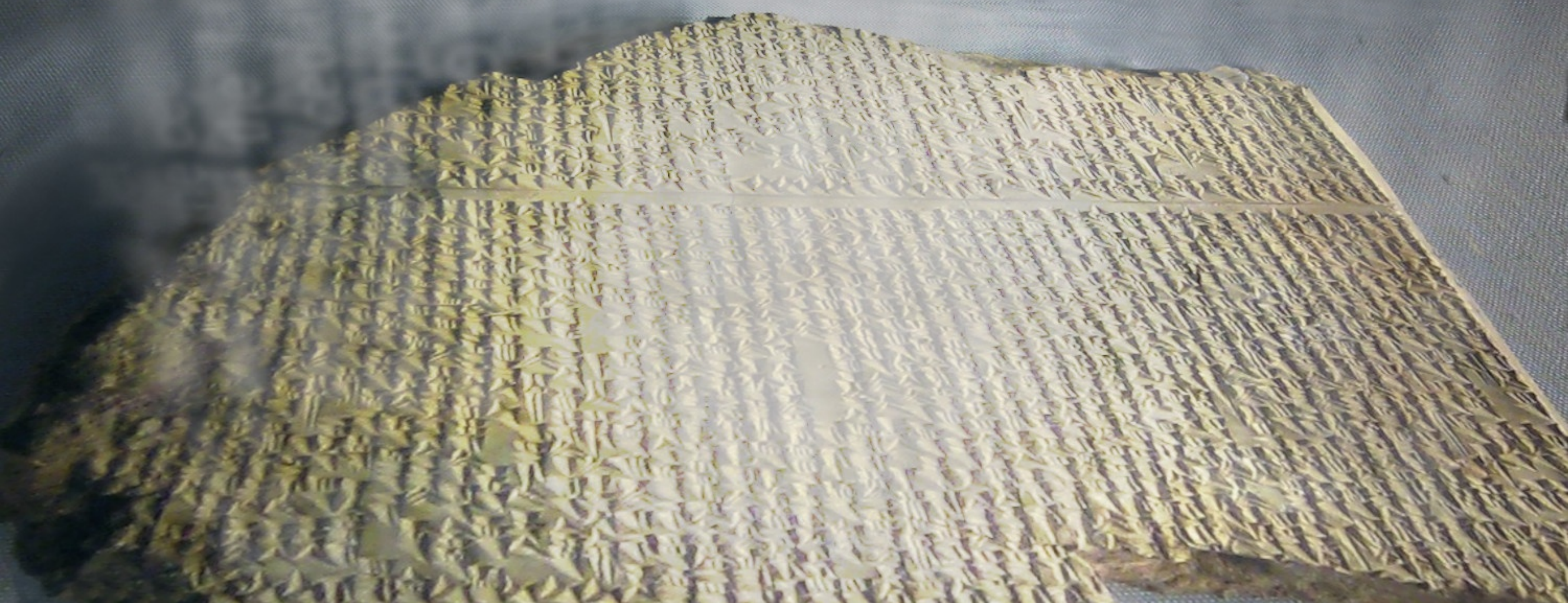
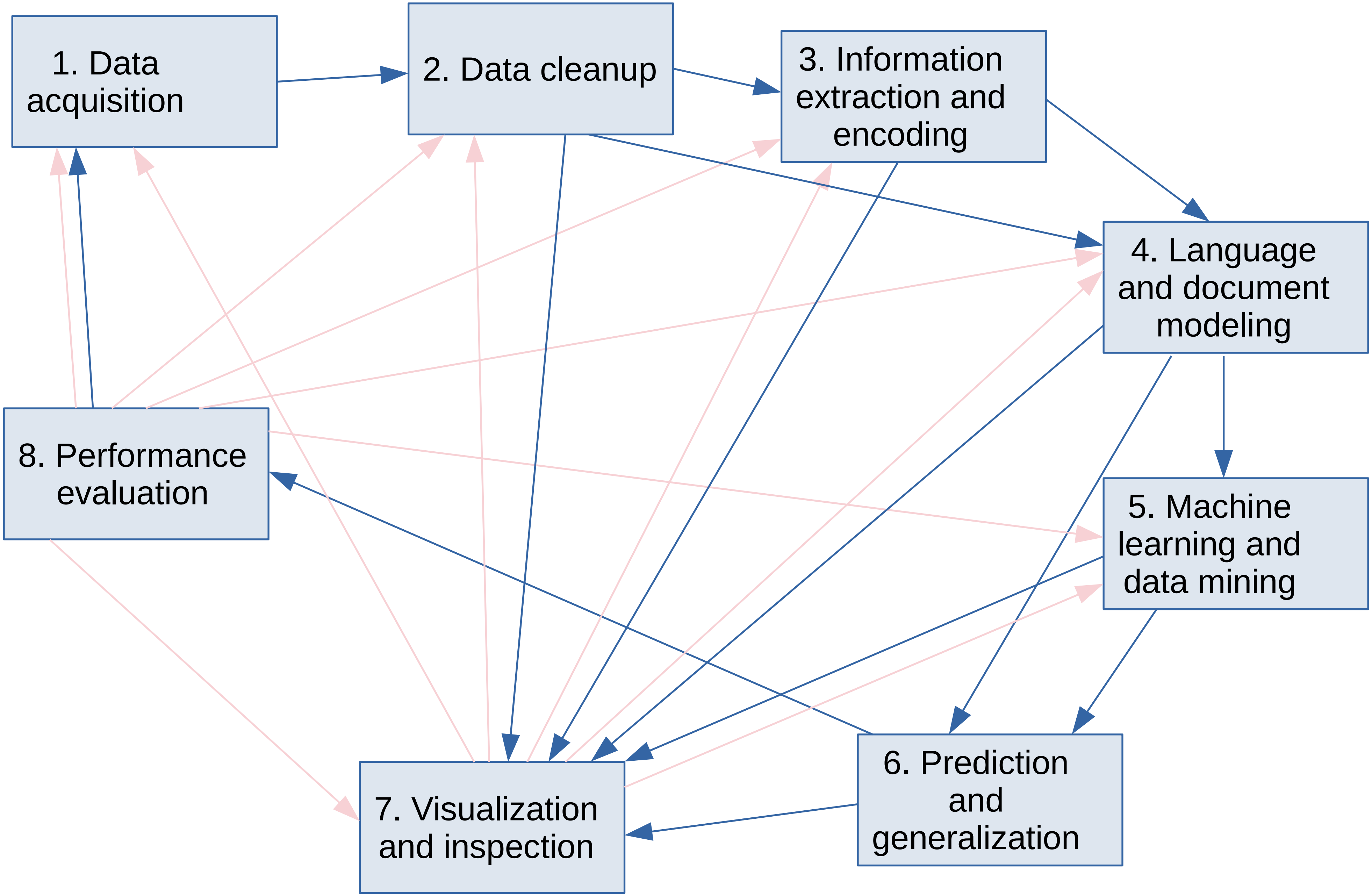
LECTURE 2:
BASIC TEXT PROCESSING



LECTURE 2: BASIC TEXT PROCESSING

Chapter 4: Text Processing Stages

Text processing stages



Text processing stages 1: acquisition

- **Data acquisition:** gathering text data.
- Ready-made collections online:
  - 20 newsgroups, Wikipedia, Project Gutenberg, ...
  - Multiple lists of data sets online. For example https://www.kaggle.com/tags/text-data, https://lionbridge.ai/datasets/the-best-25-datasets-for-natural-language-processing/, https://github.com/niderhoff/nlp-datasets, https://en.wikipedia.org/wiki/List_of_datasets_for_machine-learning_research
  - Formats vary a lot, will need conversion before reading in
- Accessing data via APIs
  - Twitter etc. services have APIs for access but limit rate of access
  - Typical data format: JSON
- Directly crawling data from the web
  - Based on a known source or information retrieval results
  - Website HTML/XML content has many parts that are about presentation rather than content, or about navigation, ads etc. undesirable content: requires cleanup
- Keep in mind when creating self-made data sets:
  - May contain personally identifiable data: gathering affected by research ethics, GDPR and other laws
  - Copyright law: depending on the situation might prevent collection and/or redistribution, or affect licensing obligations of resulting data

Text processing stages 1: acquisition

- **Reading in a downloaded file collection in Python:** os library
- Assume your data are already text files. If they are PDFs, DOCs, ODTs etc. there are graphical and command-line tools available on the web to convert them.
- Must first get a list of files and verify which ones are text files:

```
#%% Getting a list of directory contents
import os
def gettextlist(directory_path):
    directory_textfiles=[]
    directory_nontextfiles=[]
    directory_nonfiles=[]
    # Process each item in the directory
    directory_contents=os.listdir(directory_path)
    for contentitem in directory_contents:
        temp_fullpath=os.path.join(directory_path, contentitem)
        # Non-files (e.g. subdirectories) are stored separately
        if os.path.isfile(temp_fullpath)==0:
            directory_nonfiles.append(contentitem)
        else:
            # Is this a non-text file (not ending in .txt)?
            if temp_fullpath.find('.txt')==-1:
                directory_nontextfiles.append(contentitem)
            else:
                # This is a text file
                directory_textfiles.append(contentitem)
    return(directory_textfiles,directory_nontextfiles,directory_nonfiles)

mydirectory_path='c:/jaakkos_files/work/teaching/tampere_text_analytics_2023'
mydirectory_contentlists=gettextlist(mydirectory_path)
```

Text processing stages 1: acquisition

- Basic file crawler: read all files in a single directory

```
#%% Basic file crawler
def basicfilecrawler(directory_path):
    # Store filenames read and their text content
    num_files_read=0
    crawled_filenames=[]
    crawled_texts=[]
    directory_contentlists=gettextlist(directory_path)
    # In this basic crawled we just process text files
    # and do not handle subdirectories
    directory_textfiles=directory_contentlists[0]
    for contentitem in directory_textfiles:
        print('Reading file:')
        print(contentitem)
        # Open the file and read its contents
        temp_fullpath=os.path.join(directory_path, contentitem)
        temp_file=open(temp_fullpath,'r',encoding='utf-8',errors='ignore')
        temp_text=temp_file.read()
        temp_file.close()
        # Store the read filename and content
        crawled_filenames.append(contentitem)
        crawled_texts.append(temp_text)
        num_files_read=num_files_read+1
    return(crawled_filenames,crawled_texts)

mycrawled_filenames_and_texts=basicfilecrawler('c:/jaakkos_files/work/teaching/
tampere_text_analytics_2023')
mycrawled_filenames=mycrawled_filenames_and_texts[0]
mycrawled_texts=mycrawled_filenames_and_texts[1]
```

Text processing stages 1: acquisition

- **Web page crawling in Python:** scrapy, **beautifulsoup** libraries
- Example:

```
#%% Get the content of a page using the requests library
import requests
mywebpage_url='https://www.sis.uta.fi/~tojape/'
#mywebpage_url='https://www.tuni.fi/en/'
mywebpage_html=requests.get(mywebpage_url)

#%% Parse the HTML content using beautifulsoup
import bs4
mywebpage_parsed=bs4.BeautifulSoup(mywebpage_html.content,'html.parser')

#%% Get the text content of the page
def getpagetext(parsedpage):
    # Remove HTML elements that are scripts
    scriptelements=parsedpage.find_all('script')
    # Concatenate the text content from all table cells
    for scriptelement in scriptelements:
        # Extract this script element from the page.
        # This changes the page given to this function!
        scriptelement.extract()
    pagetext=parsedpage.get_text()
    return(pagetext)

mywebpage_text=getpagetext(mywebpage_parsed)
print(mywebpage_text)
```

Text processing stages 1: acquisition

- If necessary, beautifulsoup allows to search for individual cells. Be careful to avoid duplicating text: contents of nested cells are also listed in their parents!
  Example:

```
# Find HTML elements that are table cells or 'div' cells
tablecells=parsedpage.find_all(['td','div'])
# Concatenate the text content from all table or div
cells
pagetext=''
for tablecell in tablecells:
    pagetext=pagetext+'\n'+tablecell.text.strip()
```

Text processing stages 1: acquisition

- **To crawl further pages, we analyze links on the page we already crawled:**
- Example:

```
#%% Find linked pages in Finnish sites, but not PDF or PS files
def getpageurls(webpage_parsed):
    # Find elements that are hyperlinks
    pagelinkelements=webpage_parsed.find_all('a')
    pageurls=[];
    for pagelink in pagelinkelements:
        pageurl_isok=1
        try:
            pageurl=pagelink['href']
        except:
            pageurl_isok=0
        if pageurl_isok==1:
            # Check that the url does NOT contain these strings
            if (pageurl.find('.pdf')!=-1)|(pageurl.find('.ps')!=-1):
                pageurl_isok=0
            # Check that the url DOES contain these strings
            if (pageurl.find('http')==-1)|(pageurl.find('.fi')==-1):
                pageurl_isok=0
        if pageurl_isok==1:
            pageurls.append(pageurl)
    return(pageurls)
mywebpage_urls=getpageurls(mywebpage_parsed)
print(mywebpage_urls)
```

Text processing stages 1: acquisition

- **Basic crawling procedure: start from a seed page, crawl until there are enough**

```
#%% Basic web crawler
def basicwebcrawler(seedpage_url,maxpages):
    # Store URLs crawled and their text content
    num_pages_crawled=0
    crawled_urls=[]
    crawled_texts=[]
    # Remaining pages to crawl: start from a seed page URL
    pagestocrawl=[seedpage_url]
    # Process remaining pages until a desired number
    # of pages have been found
    while (num_pages_crawled<maxpages)&(len(pagestocrawl)>0):
        # Retrieve the topmost remaining page and parse it
        pagetocrawl_url=pagestocrawl[0]
        print('Getting page:')
        print(pagetocrawl_url)
        pagetocrawl_html=requests.get(pagetocrawl_url)
        pagetocrawl_parsed=bs4.BeautifulSoup(pagetocrawl_html.content,'html.parser')
        # Get the text and URLs of the page
        pagetocrawl_text=getpagetext(pagetocrawl_parsed)
        pagetocrawl_urls=getpageurls(pagetocrawl_parsed)
        # Store the URL and content of the processed page
        num_pages_crawled=num_pages_crawled+1
        crawled_urls.append(pagetocrawl_url)
        crawled_texts.append(pagetocrawl_text)
        # Remove the processed page from remaining pages,
        # but add the new URLs
        pagestocrawl=pagestocrawl[1:len(pagestocrawl)]
        pagestocrawl.extend(pagetocrawl_urls)
    return(crawled_urls,crawled_texts)
mycrawled_urls_and_texts=basiccrawler('https://www.sis.uta.fi/~tojape/',10)
mycrawled_urls=mycrawled_urls_and_texts[0]
mycrawled_texts=mycrawled_urls_and_texts[1]
```

Text preprocessing stages 2: cleanup

- **Data cleanup 1: Removing leading/trailing whitespace.** Usually noninformative, but might indicate whether a paragraph is indented or not.

- In Python:

```
mytext='   Look, here    are    some    words!\n   Great!       '
mytext.strip()
Out: 'Look, here    are    some    words!\n   Great!'
```

- **Removing multiple consecutive whitespace.** Sometimes noninformative but might indicate emphasis or paragraph/section/chapter breaks.

```
' '.join(mytext.split())
Out: 'Look, here are some words! Great!'
```

Text preprocessing stages 2: cleanup

- **Data cleanup 2: Tokenization.** Break apart a string of text into individual sentences and words. Sometimes also into paragraphs or lines. It's not just detecting spaces and periods, for example because periods in abbreviations (J. Smith, e.g., N.Y.C., et al.) do not end sentences.
- In NLTK: the Punkt sentence tokenizer is a pretrained unsupervised model that includes models for abbreviation words, collocations and sentence-starting words.

```
sentenceSplitter=nltk.data.load('tokenizers/punkt/english.pickle')
sentenceSplitter("E.g., J. Smith knows... and I know. But do you?")
Out: ['E.g., J. Smith knows... and I know.', 'But do you?']

nltk.word_tokenize("Hey, what's going on? Who's that?")
Out: ['Hey', ',', 'what', "'s", 'going', 'on', '?', 'Who', "'s", 'that', '?']
```

Text processing stages 3: encoding

- NLTK has its own internal text representation format of tokenized texts, we'll need it for further steps:

```
mytokenizedtext=nltk.word_tokenize("Hey, what's going on? Who's that?")

mynltktext=nltk.Text(mytokenizedtext)
```

- For a list of texts:

```
#%% Tokenize loaded texts and change them to NLTK format
import nltk
mycrawled_nltktexts=[]
for k in range(len(mycrawled_texts)):
    temp_tokenizedtext=nltk.word_tokenize(mycrawled_texts[k])
    temp_nltktext=nltk.Text(temp_tokenizedtext)
    mycrawled_nltktexts.append(temp_nltktext)
```

Text processing stages 3: encoding

- **Information extraction and encoding 1: case removal**
- To avoid counting different capitalizations as several words, it's useful to turn words into lowercase. This loses information: name vs common word, title vs main text, acronym vs common word.

```
'Text Processing Stages'.lower()

Out: 'text processing stages'
```

- For all texts:

```
#%% Make all crawled texts lowercase
mycrawled_lowercasetexts=[]
for k in range(len(mycrawled_nltktexts)):
    temp_lowercasetext=[]
    for l in range(len(mycrawled_nltktexts[k])):
        lowercaseword=mycrawled_nltktexts[k][l].lower()
        temp_lowercasetext.append(lowercaseword)
    temp_lowercasetest=nltk.Text(temp_lowercasetext)
    mycrawled_lowercasetexts.append(temp_lowercasetext)
```

Text processing stages 3: encoding

- **Information extraction and encoding 1: stemming and lemmatization**

- **Stemming** turns each word into the **stem** of the word (stems need not be valid words)

```
stemmer=nltk.stem.porter.PorterStemmer()
stemmer.stem('modelling')
Out: 'model'
stemmer.stem('incredible')
Out: 'incred'
```

Text processing stages 3: encoding

- Stem all crawled documents:

```
#%% Stem the loaded texts
stemmer=nltk.stem.porter.PorterStemmer()
def stemtext(nltktexttostem):
    stemmedtext=[]
    for l in range(len(nltktexttostem)):
        # Stem the word
        wordtostem=nltktexttostem[l]
        stemmedword=stemmer.stem(wordtostem)
        # Store the stemmed word
        stemmedtext.append(stemmedword)
    return(stemmedtext)
mycrawled_stemmedtexts=[]
for k in range(len(mycrawled_lowercasetexts)):
    temp_stemmedtext=stemtext(mycrawled_lowercasetexts[k])
    temp_stemmedtext=nltk.Text(temp_stemmedtext)
    mycrawled_stemmedtexts.append(temp_stemmedtext)
```

Text processing stages 3: encoding

- **Lemmatization** means turning words into basic forms called **lemmas**.
  - Needs knowledge of the part-of-speech: e.g. 'lighter' can be either a noun or an adjective, and 'automated' can be a verb or an adjective, with different lemmas.

```
# Download wordnet resource if you do not have it already
nltk.download('wordnet')
lemmatizer=nltk.stem.WordNetLemmatizer()
lemmatizer.lemmatize('better','a')
Out[309]: 'good'
lemmatizer.lemmatize('lighter','n')
Out[309]: 'lighter'
lemmatizer.lemmatize('lighter','a')
Out[310]: 'light'
lemmatizer.lemmatize('automated','v')
Out[311]: 'automate'
lemmatizer.lemmatize('automated','a')
Out[312]: 'automated'
```

- The WordNet lemmatizer assumes 'noun' by default, but in practice you would need to perform **part-of-speech tagging** (POS-tagging) for the text to categorize the part-of-speech of each word based on the sentences they are in.

Text processing stages 3: encoding

- **Part-of-speech tagging:** it is a nontrivial task based on machine learning models. For the moment we will just show how to use a ready-made tagger.

```
# Download tagger resource if you do not have it already
nltk.download('averaged_perceptron_tagger')
text1=nltk.Text(nltk.word_tokenize('it is lighter than before'))
nltk.pos_tag(text1)
Out:
[('it', 'PRP'),
 ('is', 'VBZ'),
 ('lighter', 'JJR'),
 ('than', 'IN'),
 ('before', 'IN')]
```

Here 'lighter' is tagged as a comparative adjective (JJR)

```
text2=nltk.Text(nltk.word_tokenize('it is lighter than before'))
nltk.pos_tag(text2)
nltk.pos_tag(nltk.Text(nltk.word_tokenize('it is a lighter that I bought')))
Out:
[('it', 'PRP'),
 ('is', 'VBZ'),
 ('a', 'DT'),
 ('lighter', 'NN'),
 ('that', 'WDT'),
 ('I', 'PRP'),
 ('bought', 'VBD')]
```

Here 'lighter' is tagged as a noun (NN). The tagger uses Penn Treebank tags, use this to see descriptions:

```
nltk.download('tagsets')
nltk.help.upenn_tagset()
```

Text processing stages 3: encoding

- NLTK's part of speech tags cannot be used directly with the WordNet lemmatizer, they must be converted. The starting letter of the POS tags is useful: N is for nouns, V for verbs, J for adjectives, R for adverbs, others such as pronouns cannot be lemmatized by WordNet.

```
#%% Convert a POS tag for WordNet
def tagtowordnet(postag):
    wordnettag=-1
    if postag[0]=='N':
        wordnettag='n'
    elif postag[0]=='V':
        wordnettag='v'
    elif postag[0]=='J':
        wordnettag='a'
    elif postag[0]=='R':
        wordnettag='r'
    return(wordnettag)
```

Text processing stages 3: encoding

- Lemmatize all crawled texts:

```
#%% POS tag and lemmatize the loaded texts
# Download tagger and wordnet resources if you do not have them already
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
lemmatizer=nltk.stem.WordNetLemmatizer()

def lemmatizetext(nltktexttolemmatize):
    # Tag the text with POS tags
    taggedtext=nltk.pos_tag(nltktexttolemmatize)
    # Lemmatize each word text
    lemmatizedtext=[]
    for l in range(len(taggedtext)):
        # Lemmatize a word using the WordNet converted POS tag
        wordtolemmatize=taggedtext[l][0]
        wordnettag=tagtowordnet(taggedtext[l][1])
        if wordnettag!=-1:
            lemmatizedword=lemmatizer.lemmatize(wordtolemmatize,wordnettag)
        else:
            lemmatizedword=wordtolemmatize
        # Store the lemmatized word
        lemmatizedtext.append(lemmatizedword)
    return(lemmatizedtext)

mycrawled_lemmatizedtexts=[]
for k in range(len(mycrawled_lowercasetexts)):
    lemmatizedtext=lemmatizetext(mycrawled_lowercasetexts[k])
    lemmatizedtext=nltk.Text(lemmatizedtext)
    mycrawled_lemmatizedtexts.append(lemmatizedtext)
```

Text processing stages 3: encoding

- Many future steps require knowledge of the **vocabulary** (list of unique words/stems/lemmas over the collection):

```
#%% Find the vocabulary, in a distributed fashion
import numpy
myvocabularies=[]
myindices_in_vocabularies=[]
# Find the vocabulary of each document
for k in range(len(mycrawled_lemmatizedtexts)):
    # Get unique words and where they occur
    temptext=mycrawled_lemmatizedtexts[k]
    uniqueresults=numpy.unique(temptext,return_inverse=True)
    uniquewords=uniqueresults[0]
    wordindices=uniqueresults[1]
    # Store the vocabulary and indices of document words in it
    myvocabularies.append(uniquewords)
    myindices_in_vocabularies.append(wordindices)

myvocabularies[0]
Out: array(['!', '$', '&', ..., 'ziyuan', '©', 'âkerlund'],
dtype='<U25')
```

Text processing stages 3: encoding

- Multiple vocabularies can be unified:

```
# Unify the vocabularies.
# First concatenate all vocabularies
tempvocabulary=[]
for k in range(len(mycrawled_lemmatizedtexts)):
    tempvocabulary.extend(myvocabularies[k])
# Find the unique elements among all vocabularies
uniqueresults=numpy.unique(tempvocabulary,return_inverse=True)
unifiedvocabulary=uniqueresults[0]
wordindices=uniqueresults[1]
# Translate previous indices to the unified vocabulary.
# Must keep track where each vocabulary started in
# the concatenated one.
vocabularystart=0
myindices_in_unifiedvocabulary=[]
for k in range(len(mycrawled_lemmatizedtexts)):
    # In order to shift word indices, we must temporarily
    # change their data type to a Numpy array
    tempindices=numpy.array(myindices_in_vocabularies[k])
    tempindices=tempindices+vocabularystart
    tempindices=wordindices[tempindices]
    myindices_in_unifiedvocabulary.append(tempindices)
    vocabularystart=vocabularystart+len(myvocabularies[k])
```

Text processing stages 3: encoding

- A piece or the unified vocabulary (words 1000-1050):

```
unifiedvocabulary[1000:1050]
Out:
array(['college', 'color', 'columbia',
'combination', 'combinatorial', 'combine',
'combining', 'come', 'comet', 'command',
'commitment', 'committee', 'communicate',
'communication', 'community', 'comp', 'company',
'compare', 'competence', 'competitive',
'competitiveness', 'compile', 'complement',
'complementary', 'complete', 'complex', 'component',
'comprehension', 'comprises', 'compromise',
'computation', 'computational', 'computationally',
'compute', 'computer', 'computing', 'conceptual',
'conditionally', 'conference',
'conferences/workshops', 'congrats',
'congratulation', 'connect', 'connection',
'consider', 'consist', 'consists', 'constantly',
'constitute', 'constrained'], dtype='<U48')
```

Text processing stages 3: encoding

- Documents can be represented as a vector of indices to the unified vocabulary. Words 600-650 words in document 1 (TUNI homepage):

```
myindices_in_unifiedvocabulary[1][600:650]
Out:
array([4373, 3984, 4373, 3984, 4373,  821, 3034, 3001, 4077, 2816,
2845,
       4373, 1902, 1562,   19,  621,  619, 1972, 3485, 1547, 3001,
3852,
        821, 3529,  769, 3130, 4373,   19, 3472, 2813, 3984, 4373,
3001,
        698, 3657, 3984, 4373, 3001,  698, 3657,    6, 3980,    7,
821,
       3034, 3001, 4077, 2431,  668, 2816], dtype=int64)
unifiedvocabulary[myindices_in_unifiedvocabulary[1]
[600:650]]
Out:
array(['university', 'tampere', 'university', 'tampere',
'university','be', 'one', 'of', 'the', 'most', 'multidisciplinary',
'university', 'in', 'finland', '.', 'almost', 'all',
'internationally', 'recognise', 'field', 'of', 'study', 'be',
'represent', 'at', 'our', 'university', '.', 'read', 'more',
'tampere', 'university', 'of', 'applied', 'science', 'tampere',
'university', 'of', 'applied', 'science', '(', 'tamk', ')', 'be',
'one', 'of', 'the', 'large', 'and', 'most'], dtype='<U48')
```

LECTURE 2:
BASIC TEXT PROCESSING

Chapter 5:
Basic Word Statistics
and Vocabulary Pruning

Counting prominent words

- Word counts can be inspected by several statistics:
  - total count of a word over all documents,
  - total number of documents where the word occurs,
  - mean count over documents,
  - variance over documents; and many others
- In Python:

```
#%% Count the numbers of occurrences of each unique word
# Let's count also various statistics over the documents
unifiedvocabulary_totaloccurrencecounts=numpy.zeros((len(unifiedvocabulary),1))
unifiedvocabulary_documentcounts=numpy.zeros((len(unifiedvocabulary),1))
unifiedvocabulary_meancounts=numpy.zeros((len(unifiedvocabulary),1))
unifiedvocabulary_countvariances=numpy.zeros((len(unifiedvocabulary),1))
```

Counting prominent words

```
# First pass: count occurrences
for k in range(len(mycrawled_lemmatizedtexts)):
    print(k)
    occurrencecounts=numpy.zeros((len(unifiedvocabulary),1))
    for l in range(len(myindices_in_unifiedvocabulary[k])):
        occurrencecounts[myindices_in_unifiedvocabulary[k][l]]= \
            occurrencecounts[myindices_in_unifiedvocabulary[k][l]]+1
    unifiedvocabulary_totaloccurrencecounts= \
        unifiedvocabulary_totaloccurrencecounts+occurrencecounts
    unifiedvocabulary_documentcounts= \
        unifiedvocabulary_documentcounts+(occurrencecounts>0)

# Mean occurrence counts over documents
unifiedvocabulary_meancounts= \
    unifiedvocabulary_totaloccurrencecounts/len(mycrawled_lemmatizedtexts)

# Second pass to count variances
for k in range(len(mycrawled_lemmatizedtexts)):
    print(k)
    occurrencecounts=numpy.zeros((len(unifiedvocabulary),1))
    for l in range(len(myindices_in_unifiedvocabulary[k])):
        occurrencecounts[myindices_in_unifiedvocabulary[k][l]]= \
            occurrencecounts[myindices_in_unifiedvocabulary[k][l]]+1
    unifiedvocabulary_countvariances=unifiedvocabulary_countvariances+ \
        (occurrencecounts-unifiedvocabulary_meancounts)**2
unifiedvocabulary_countvariances= \
    unifiedvocabulary_countvariances/(len(mycrawled_lemmatizedtexts)-1)
```

Inspecting prominent words

- Resulting words can be inspected: sort words by each statistic, print words with highest value of the statistic

```
#%% Inspect frequent words
# Sort words by largest total (or mean) occurrence count
highest_totaloccurrences_indices=numpy.argsort(\
    -1*unifiedvocabulary_totaloccurrencecounts,axis=0)
print(numpy.squeeze(unifiedvocabulary[\
    highest_totaloccurrences_indices[1:100]]))
print(numpy.squeeze(\
    unifiedvocabulary_totaloccurrencecounts[\
    highest_totaloccurrences_indices[1:100]]))
```

- The Twenty Newsgroups (20NG) data set is a famous old data set of emails sent to different topical USENET news groups (somewhat like subreddits of Reddit today). The data set has messages for twenty different newsgroups, 1000 messages from each.

- Let's process the 20NG data set according to the pipeline, and inspect the most frequent words!

Inspecting prominent words

- 20 newsgroups, words with highest total word count:

```
[':' ',' 'the' '.' '!' 'be' '--' '@' 'to' ')' 'of' '(' 'a' 'and' 'i' '<'
 'in' 'that' "'ax" '?' 'it' 'have' "''" 'for' 'you' 'do' 'from' '``' 'on'
 'this' 'not' '$' '|' 'with' '#' "'s" 'as' 'cantaloupe.srv.cs.cmu.edu' ';'
 "n't" '-' '%' 'or' 'if' ']' 'but' 'they' 'line' '[' 'subject' 'date'
 'newsgroups' 'path' 'message-id' 'organization' '...' "'" '&' 'apr' 'by'
 'at' 'can' 'gmt' 'what' 'an' 'write' 'would' 'my' 'there' 'one' 'all'
 'we' 'will' '1993' 'use' 'about' 're' '`' 'he' 'get' 'reference' 'so'
 'your' 'article' 'say' 'no' 'any' 'who' 'me' 'some' 'know' 'news'
 'sender' 'which' 'howland.reston.ans.net' '1' 'out' 'make' 'like']
[308665. 305640. 256006. 253425. 231060. 195516. 186472. 151243. 128253.
 124342. 122058. 121734. 107680. 101532.  87092.  85864.  85541.  70471.
  61717.  58759.  58028.  56367.  53426.  49541.  47797.  46005.  39501.
  38817.  34748.  34600.  33742.  33257.  32981.  30180.  29506.  29229.
  27682.  26047.  25974.  25580.  25510.  25452.  25370.  24095.  23297.
  23171.  23051.  22754.  22015.  21720.  20874.  20396.  20361.  20082.
  20040.  20037.  19977.  19968.  19628.  19362.  19296.  18388.  17760.
  17652.  17337.  17164.  17014.  16799.  16747.  16129.  16029.  15882.
  15332.  14874.  14852.  14591.  14552.  14204.  14118.  14113.  13796.
  13206.  13072.  13035.  13008.  12881.  12114.  11923.  11742.  11486.
  11426.  11398.  10983.  10940.  10905.  10837.  10654.  10561.  10405.]
```

- Most of the top frequent words are either due to the nature of the messages (email header information) or because they are **stop words** that are uninformative about the content (punctuation; a,the; but,for; etc.)

Inspecting prominent words

- Resulting words can be inspected: sort words by each statistic, print words with highest value of the statistic

```
# Sort words by largest total document count
highest_documentoccurrences_indices=numpy.argsort(\
    -1*unifiedvocabulary_documentcounts,axis=0)
print(numpy.squeeze(unifiedvocabulary[\
    highest_documentoccurrences_indices[1:100]]))
print(numpy.squeeze(\
    unifiedvocabulary_documentcounts[\
    highest_documentoccurrences_indices[1:100]]))
```

Inspecting prominent words

- 20 newsgroups, words with highest total document count:

```
['' 'subject' '>' 'from' '@' 'cantaloupe.srv.cs.cmu.edu' 'path' '<'
 'newsgroups' 'message-id' '!' 'line' ',' ')' '(' '.' 'organization' 'the'
 'be' 'apr' 'of' 'to' 'a' 'gmt' 'in' 'and' 'i' 'have' 'for' 'that' 'it'
 '?' 'do' 're' '1993' 'reference' 'on' 'this' 'you' '--' 'write' 'with'
 'not' 'sender' 'howland.reston.ans.net' "'s" 'if' 'but' 'or' "''" "n't"
 'article' 'as' '``' 'at' 'nntp-posting-host' 'can' 'an'
 'zaphod.mps.ohio-state.edu' 'what' 'there' 'would' 'they' '-'
 'university' 'one' 'my' 'all' '...' 'by' '93' 'news' 'about' 'get' 'so'
 'any' 'me' 'know' 'no' 'use' 'like' 'will' 'crabapple.srv.cs.cmu.edu'
 'some' 'just' 'out' 'noc.near.net' 'news.sei.cmu.edu' 'xref' 'say' '$'
 'das-news.harvard.edu' 'make' 'your' 'who' 'think' 'more' 'which' 'when']
[19997. 19997. 19997. 19997. 19997. 19997. 19997. 19997. 19997. 19997.
 19997. 19947. 19743. 19608. 19596. 19515. 19218. 18697. 18564. 18396.
 17527. 17517. 17407. 17015. 16832. 16694. 16537. 14709. 14617. 14285.
 14251. 13847. 13637. 13539. 13232. 12832. 12060. 11976. 11723. 11538.
 11172. 11160. 11134. 10890. 10890. 10543. 10321. 10301. 10262. 10222.
 10052.  9248.  9160.  9087.  8902.  8586.  8539.  8260.  8237.  8163.
  8075.  8022.  7942.  7924.  7857.  7829.  7792.  7783.  7741.  7703.
  7686.  7599.  7518.  7452.  7295.  7167.  6898.  6755.  6627.  6577.
  6426.  6425.  6235.  6219.  6214.  6206.  6132.  6056.  6052.  5935.
  5846.  5834.  5795.  5788.  5690.  5675.  5602.  5568.  5509.]
```

- Most of the top frequent words are again either due to the nature of the messages (email header information) or because they are **stop words** that are uninformative about the content (punctuation; a,the; but,for; etc.)

Inspecting prominent words

- Resulting words can be inspected: sort words by each statistic, print words with highest value of the statistic

```
# Sort by largest variance of count over documents
highest_countvariances_indices=numpy.argsort(\
    -1*unifiedvocabulary_countvariances,axis=0)
print(numpy.squeeze(unifiedvocabulary[\
    highest_countvariances_indices[1:100]]))
print(numpy.squeeze(\
    unifiedvocabulary_countvariances[\
    highest_countvariances_indices[1:100]]))
```

Inspecting prominent words

- 20 newsgroups, words with highest variance of count over documents:

```
["'ax" '--' '(' '@' ',' '`' '%' ')' '<' '$' '#' '!' '``' 'the' "''" '&'
 '.' '?' ';' ']' ':' '[' "'" 'be' 'x' 'of' 'to' 'and' 'max' 'm' 'a' 'in'
 'i' '0' 'that' '-' '1' 'you' '|' 'it' 'r' 'q' 'g' 'for' 'have' '2' 'do'
 'p' '*' 'they' 'on' '7' 'q,3' '...' 'db' 'this' "'s" 'not' 'we' '=' 'he'
 'with' 'or' 'file' "n't" 'as' 'n' '3' 'b8f' '_' 'by' 'a86' '4' '0d'
 'from' 'image' '+' 'b' '/' 'if' '145' 'o' '*/' 'w' 'jpeg' 'c' 'say' '5'
 'use' 'f' 'can' 'h' 'at' '\\' 'd' 'will' 'but' 'what' '9']
[2.31385026e+04 5.10833271e+03 1.97899732e+03 1.72792804e+03 1.36613857e+03 1.14835161e+03 1.14226540e+03 1.11119381e+03
 1.03339025e+03 8.64996964e+02 8.48719707e+02 8.02762520e+02 7.87145615e+02 7.76200379e+02 7.53716629e+02 7.48663881e+02
 7.10752636e+02 6.67528311e+02 6.65297985e+02 6.38842139e+02 6.28565224e+02 6.13663050e+02 3.83489097e+02 3.76795656e+02
 2.42628173e+02 1.82667657e+02 1.72595120e+02 1.61295525e+02 1.20326961e+02 1.19953447e+02 1.10472990e+02 7.08945278e+01
 6.81021345e+01 6.80772730e+01 6.45329894e+01 5.08903337e+01 3.81987443e+01 3.56819465e+01 3.54732708e+01 3.26273406e+01
 3.22408478e+01 3.22023424e+01 3.20089523e+01 3.08456219e+01 2.99705557e+01 2.72867862e+01 2.34713004e+01 1.89942342e+01
 1.89535414e+01 1.86998192e+01 1.77483381e+01 1.74992230e+01 1.53589340e+01 1.52892539e+01 1.51227133e+01 1.42914093e+01
 1.32011337e+01 1.28224311e+01 1.25278161e+01 1.21016164e+01 1.20880480e+01 1.17876349e+01 1.15605785e+01 1.13689877e+01
 1.08437823e+01 1.07597317e+01 1.07031238e+01 9.80075522e+00 9.56500781e+00 8.87841384e+00 8.77659711e+00 8.42420120e+00
 8.38486846e+00 8.08329236e+00 7.94143057e+00 7.91817919e+00 7.77909082e+00 7.45941030e+00 7.18507298e+00 7.12563482e+00
 6.93567082e+00 6.81984541e+00 6.79389226e+00 6.73115550e+00 6.58960980e+00 6.53544674e+00 6.35998838e+00 6.33768675e+00
 6.19780207e+00 6.18304892e+00 5.80453773e+00 5.78976288e+00 5.67505632e+00 5.67425326e+00 5.67277218e+00 5.63696367e+00
 5.63243702e+00 5.61282294e+00 5.60039341e+00]
```

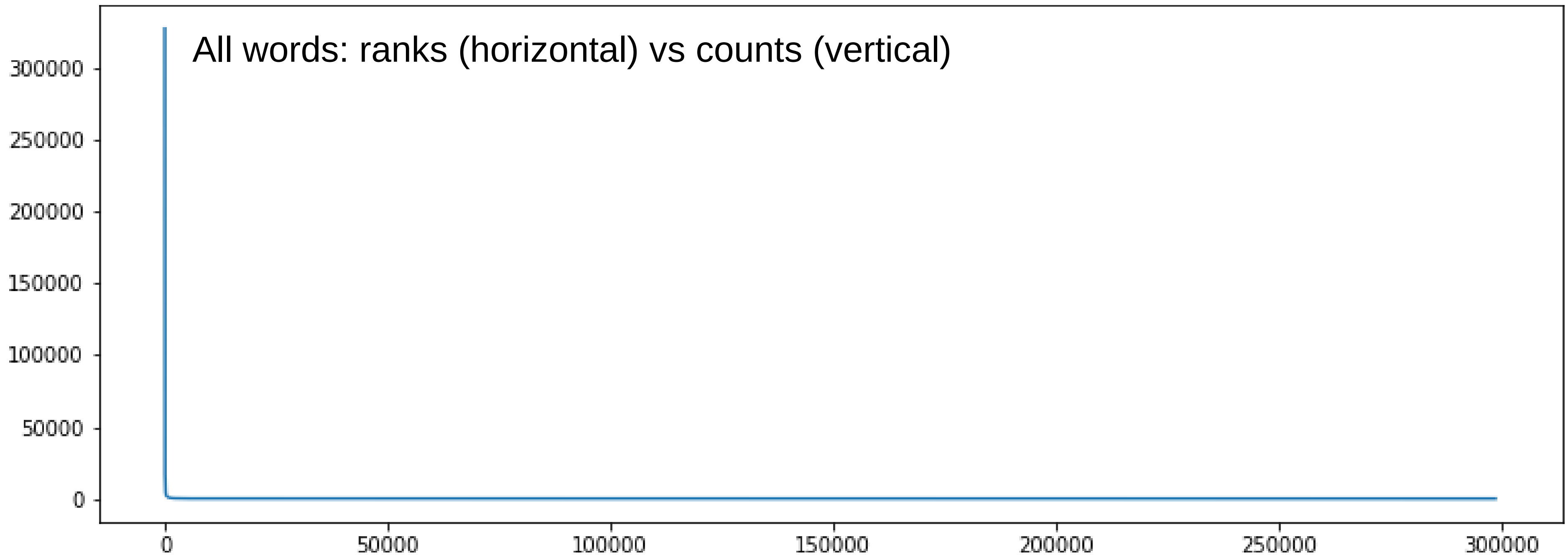
- Most of the top varying words are either stop words or perhaps part of some code fragments used in some newsgroups
- In order to analyze the content of the newsgroups, we should **prune the vocabulary** of uninformative words

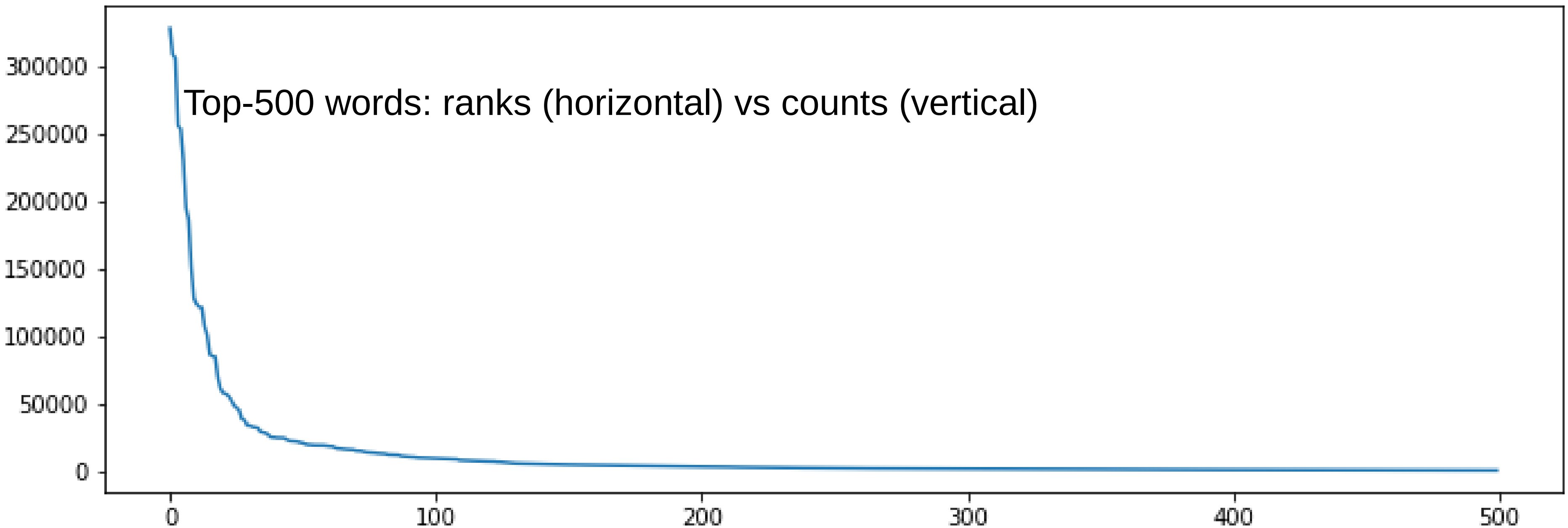
Inspecting prominent words

- Let's plot the most frequent words against their ranks. In Python:

```
#%% Make a frequency plot of the words
# Import the plotting library
import matplotlib.pyplot
# Tell the library we want each plot in its own window
%matplotlib auto
# Create a figure and an axis
myfigure, myaxes = matplotlib.pyplot.subplots();
# Plot the sorted occurrence counts of the words against their ranks
horizontalpositions=range(len(unifiedvocabulary))
verticalpositions=numpy.squeeze(unifiedvocabulary_totaloccurrencecounts[\
    highest_totaloccurrences_indices])
myaxes.plot(horizontalpositions,verticalpositions);
# Plot the top-500 occurrence counts of the words against their ranks
myfigure, myaxes = matplotlib.pyplot.subplots();
horizontalpositions=range(500)
verticalpositions=numpy.squeeze(unifiedvocabulary_totaloccurrencecounts[\
    highest_totaloccurrences_indices[0:500]])
myaxes.plot(horizontalpositions,verticalpositions);
```

Inspecting prominent words





Zipf's law

- Named after American linguist George Kingsley Zipf
- Specific case of a **rank-frequency distribution**
- Given a corpus of natural-language utterances, the **frequency** of a word w is inversely proportional to its **rank** in the frequency table

p(w) is the proportion of word w in the corpus (probability to get w in a random draw from the corpus)

$$p(w) = \frac{1/rank(w)^a}{\sum_{k=1}^{V} 1/k^a}$$

a is an exponent characterizing the distribution

- The few top most common words account for most of all word occurrences
- Occurs also in many other domains than text

Zipf's law

- 20 newsgroups, top 200 word counts compared to Zipf's law: