COMP3420 Lesson 8

Greg Baker

2023-04-24

Contents

1 Reading

Readings

- Jurafsky and Martin Chapter 14.1
- (Optional) The NLTK Book (Chapters 1 and 2) might be helpful: `https://www.nltk.org/book`

2 Free stuff

Common natural language processing libraries

NLTK The easiest to learn, good for teaching. We'll use this a lot. `www.nltk.org`

spaCy What you are more likely to use in a job. `https://spacy.io`

scikit-learn Has some text processing capabilities

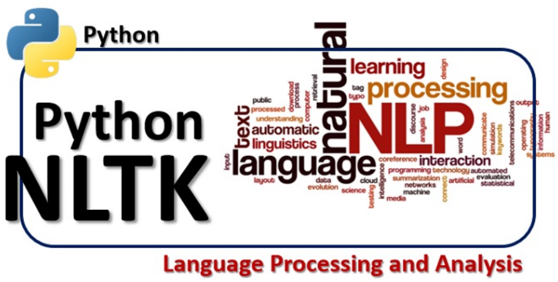
Others worth mentioning: gensim, TextBlob

Installing NLTK

- `http://www.nltk.org/install.html`.
- Pre-installed in Anaconda.
- Or `pip install nltk`
- Or `conda install nltk`

But, you'll also need to use `nltk.download()` to fetch many *corpora* and *models*. Common ones:

- punkt
- wordnet
- gutenberg



NLTK Packaged Tools

Some NLTK tools that are useful for text pre-processing are:

- `word_tokenize(text)`
- `sent_tokenize(text)`

In later lessons we'll use:

- `pos_tag(tokens)`
- `pos_tag_sents(sentences)`
- `PorterStemmer()`

Project Gutenberg

- Oldest digital library (1971)
- 70,000 free books (HTML, EPUB)
- Mostly books where copyright has expired
- NLTK has some famous Project Gutenberg books

Using Gutenberg sample data

All NLTK modules are under the `nltk` namespace.

```
#!/usr/bin/env python
import nltk
nltk.download('gutenberg')
for id in nltk.corpus.gutenberg.fileids():
    print(id)
```

Output:

```
[nltk_data] Downloading package gutenberg to /home/gregb/nltk_data...
[nltk_data]   Unzipping corpora/gutenberg.zip.
austen-emma.txt
austen-persuasion.txt
austen-sense.txt
bible-kjv.txt
blake-poems.txt
bryant-stories.txt
burgess-busterbrown.txt
carroll-alice.txt
chesterton-ball.txt
chesterton-brown.txt
chesterton-thursday.txt
edgeworth-parents.txt
...
```

3 Lexico-statistics

Some simple metrics from Jane Austin's "Emma"

```
#!/usr/bin/env python3
import nltk
import collections
import matplotlib.pyplot

emma =
    nltk.corpus.gutenberg.words('austen-emma.txt')
print(f"The number of words is {len(emma)=}
    ")
print(f"Distinct words = {len(set(emma))}")
print(f"First ten words... {emma[:10]=}")
emma_counter = collections.Counter(emma)
print(f"Top ten most fcommon words:
    {emma_counter.most_common(10)=}")
```

Output

```
The number of words is len(emma)=192427
Distinct words = 7811
First ten words... emma[:10]=['[', 'Emma', 'by', 'Jane',
'Austen', '1816',']', 'VOLUME', 'I', 'CHAPTER']
Top ten most fcommon words: emma_counter.most_common(10)
=[(',', 11454), ('.', 6928), ('to', 5183), ('the', 4844)
, ('and', 4672), ('of', 4279), ('I', 3178), ('a', 3004),
('was', 2385), ('her', 2381)]
```

Stylistic cues

We often have distinctive metrics in our speech and writing (which are often used in anti-plagiary programs) such as:

- Rate at which new words are introduced
- Zipf's law coefficients
- Proportion of text using common words
- Proportion of past-tense verbs to present tense. (This is correlated with introversion or extraversion!)

Fun reading: **The Secret Life of Pronouns: What Our Words Say About Us** by James W Bennebaker (University of Texas)

Zipf's Law

$$f(r) = \frac{C}{r^s}$$

- $f(r)$ is the frequency of the rth most common word

- C is a constant of proportionality
- s is the Zipf exponent, which measures whether you are concise or wordy.

  Shakespeare $s \approx 1$

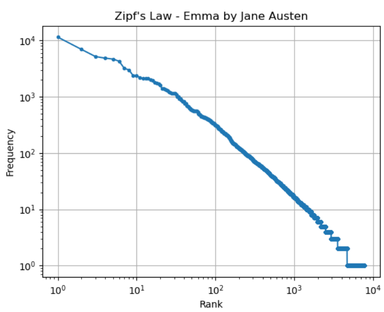
  G.K. Chesterton $s \approx 1.1$

  Jane Austen $1.2 < s < 1.4$

Graphing Zipf's law for Emma

```
ranks = list(range(1, len(word_frequencies)
    + 1))

# Create a log-log plot showing word
    frequencies vs ranking
fig, ax = matplotlib.pyplot.subplots()
ax.loglog(ranks, word_frequencies,
    marker='.')
ax.set_xlabel('Rank')
ax.set_ylabel('Frequency')
ax.set_title("Zipf's Law - Emma by Jane
    Austen")
ax.grid(True)

# Save the figure
fig.savefig("emma_zipf.png")
```

Zipf's law for Emma

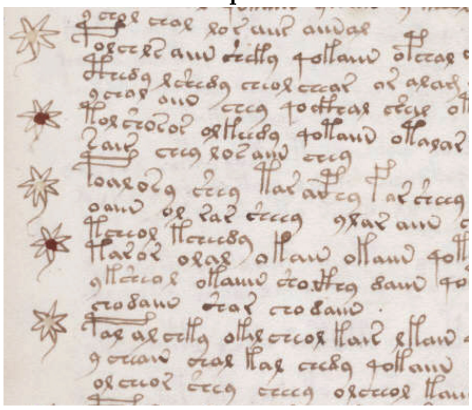


Calculating Zipf's law coefficients

```
import sklearn.linear_model
import pandas
import math
X = pandas.DataFrame({'ranks': ranks,
    'frequencies': word_frequencies})
X['log_rank'] = X['ranks'].map(math.log10)
X['log_frequencies'] =
    X['frequencies'].map(math.log10)
lr = sklearn.linear_model.LinearRegression()
lr.fit(X[['log_rank']], X.log_frequencies)
print(f"log_frequencies = {lr.coef_[0]} *
    log_rank + {lr.intercept_}")
```

Output: `log_frequencies = -1.4046388550730255 * log_rank + 5.3713421`

Practical uses of Zip's Law



Is the Voynich document a real language?

How would we recognise a SETI signal as being language?

Heap's Law / Herdan's Law

Herdan's law is an extension of Zipf's law.

$$V = kN^{\beta}$$

where:

- V is the size of the vocabulary
- N is the size of the corpus
- k and β are constants that depend on the language and the type of text
- Usually $.67 < \beta < .75$ (Jane Austen is verbose, so very low β; Shakespeare is concise, so very high β)

How many hits will you get?

If you search for occurrences of a word in a corpus, on average you will get this many hits:

$$\frac{N}{V} = \frac{N}{kN^{\beta}} = \frac{N^{1-\beta}}{k}$$

Calculating Herdan's Law Parameters on "Emma" (1/3)

```
#!/usr/bin/env python3
import nltk
import math
emma =
    nltk.corpus.gutenberg.words('austen-emma.txt'
vocab_so_far = set()
vocab_sizes = []
word_counts = []
log_word_counts = []
log_vocab_sizes = []
for i,w in enumerate(emma):
    vocab_so_far.update([w])
    vocab_sizes.append(len(vocab_so_far))
    word_counts.append(i+1)
    log_word_counts.append(math.log10(i+1))
    log_vocab_sizes.append(math.log10(len(vocab_so_far)))
```

Calculating Herdan's Law Parameters on "Emma" (2/3)

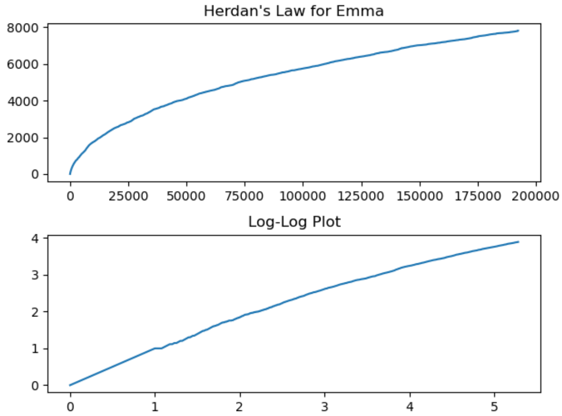
```
import pandas
import numpy
herdans_data =
    pandas.Series(data=vocab_sizes,
    index=word_counts)
log_data =
    pandas.Series(data=log_vocab_sizes,
    index=log_word_counts)
beta, log_k = numpy.polyfit(log_word_counts,
    log_vocab_sizes, 1)
k = 10**log_k

# Print the values of k and beta
print("k =", k)
print("beta =", beta)
```

Output

```
k = 11.80853406135783
beta = 0.5376699535119386
```

Calculating Herdan's Law Parameters on "Emma" (3/3)

```
import matplotlib.pyplot
fig, axes =
    matplotlib.pyplot.subplots(nrows=2)
herdans_data.plot(ax=axes[0],
    title="Herdan's Law for Emma")
log_data.plot(ax=axes[1], title="Log-Log
    Plot")
fig.tight_layout()
fig.savefig('herdans.png')
```

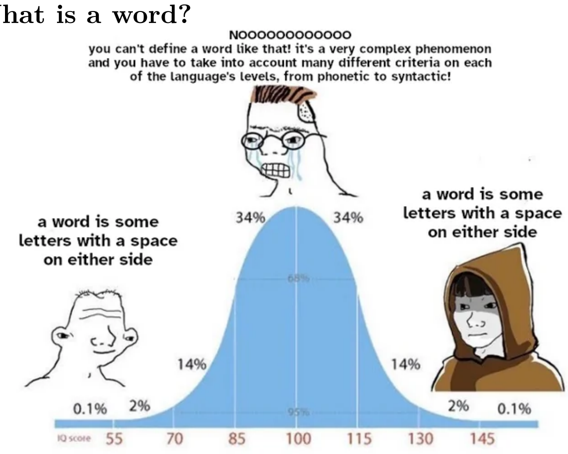


Summary

- NLTK is a Python library
- It has some convenient project Gutenberg books
- Zipf's Law and Herdan's Law are interesting *lexico-statistics* often used in *author identification*

4 Words

What is a word?



Space-based tokenization

A very simple way to tokenize!

- For languages that use space characters between words
- Arabic, Cyrillic, Greek, Latin, etc., based writing systems
- Segment off a token between instances of spaces

Split on regex: `\b`

Issues in Tokenization

Can't just blindly remove punctuation:

- m.p.h., Ph.D., AT&T, cap'n
- prices ($45.55)
- dates (01/02/06)
- URLs (http://www.stanford.edu)
- hashtags (#nlproc)
- email addresses (someone@mq.edu.au)

Clitic: a word that doesn't stand on its own

- "are" in we're, French "je" in j'ai, le in l'honneur

When should multiword expressions (MWE) be words?

- New York, rock'n'roll

Tokenization in NLTK

Bird, Loper and Klein (2009), Natural Language Processing with Python. O'Reilly

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Default Sentence and Word Tokenisation with NLTK

- NLTK can split English text into sentences and words.
  - Sentence segmentation splits text into a list of sentences.
  - Word tokenisation splits text into a list of words (tokens).
- Usually you split into sentences first, and then into words.

Using word_tokenize and sent_tokenize

```python
#!/usr/bin/env python3
import nltk
text = "Who has a Ph.D? I don't, yet."
print(nltk.sent_tokenize(text))
for s in nltk.sent_tokenize(text):
  for i,w in enumerate(nltk.word_tokenize(s)):
    print(f"Word #{i} is {w}")
```

Output:

```
['Who has a Ph.D?', "I don't, yet."]
Word #0 is Who
Word #1 is has
Word #2 is a
Word #3 is Ph.D
Word #4 is ?
Word #0 is I
Word #1 is do
Word #2 is n't
Word #3 is ,
Word #4 is yet
Word #5 is .
```

Tokenization in languages without spaces

Many languages (like Chinese, Japanese, Thai) don't use spaces to separate words!

How do we decide where the token boundaries should be?

How to do word tokenization in Chinese?

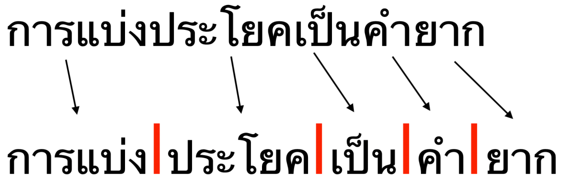
姚明进入总决赛 yáo míng jìn rù zǒng jué sài "Yao Ming reaches the finals"

3 words?
姚明 进入 总决赛
YaoMing reaches finals

5 words?
姚 明 进入 总 决赛
Yao Ming reaches overall finals

7 characters? (don't use words at all):
姚 明 进 入 总 决 赛
Yao Ming enter enter overall decision game

Work tokenization in Thai



Some heuristics, but often "word boundaries are whatever the dictionary says are word boundaries".

Byte-pair encoding

Another option for text tokenization (which is used by OpenAI for GPT) is **BPE**.

Instead of:

- white-space segmentation
- single-character segmentation

Use the data to tell us how to tokenize.

Subword tokenization (because tokens can be parts of words as well as whole words)

Multi-word tokenization (it multiple words regularly go together)

Byte Pair Encoding (BPE) token learner

Let the vocabulary be the set of all individual characters $= A, B, C, D, \ldots, a, b, c, d \ldots$

Repeat:

- Choose the two symbols that are most frequently adjacent in the training corpus (say 'A', 'B')
- Add a new merged symbol 'AB' to the vocabulary
- Replace every adjacent 'A' 'B' in the corpus with 'AB'.

Until k merges have been done, or the vocabulary is the target size

Hugging Face

- One of the top AI / text companies in the world
- Create lots of open source software
- And some nice tutorials, e.g. this one on BPE: `https://youtu.be/HEikzVL-lZU`

Install their tokenizer package with `conda install tokenizers` or `pip install tokenizers`.

Using Huggingface's BPE Tokenizer

```
#!/usr/bin/env python3
import nltk
import tokenizers
tok =
    tokenizers.Tokenizer(tokenizers.models.BPE())
trainer = tokenizers.trainers.BpeTrainer(
    vocab_size=200, # way too low for real
        usage
    special_tokens=["[UNK]", "[CLS]",
        "[SEP]"]
)
tok.train(files=[nltk.corpus.gutenberg.abspath('austen-emma.t
         trainer=trainer)
print(f"{tok.get_vocab_size()=}")
#print(tok.get_vocab())
sentence = "Emma thought little of this."
output = tok.encode(sentence)
print(output.tokens)
tok.save('bpe-example.json')
```

BPE Tokenizer Output

```
tok.get_vocab_size()=200
['E', 'm', 'm', 'a ', 'th', 'ou', 'gh', 't ', 'l',
 'it', 't', 'le ', 'of ', 'th', 'is', '.']
```

Why BPE is awesome

- Can handle any encoding: UTF-8, UTF-16, ASCII, CP1252. Input is bytes.
- Works with any language, and produces results that look like “words” (Zipf’s Law and Herdan’s Law apply)
  - Any human language
  - Computer programming languages
  - Animal languages?

Bigrams

A bigram is a sequence of two words, and is a little faster to compute than BPE. If your search is getting too many hits, you can make your vocabulary richer quickly by using bigrams.

```
>>>
    list(nltk.bigrams([1,2,3,4,5,6]))
[(1, 2), (2, 3), (3, 4), (4, 5),
    (5, 6)]
>>> list(nltk.bigrams(emma))[:3]
[('[', 'Emma'), ('Emma', 'by'),
    ('by', 'Jane')]
```

Ngrams

Why stop at 2? For very large corpora, you might need 3-grams or 4-grams!

- A bigram is an ngram where n is 2.
- A trigram is an ngram where n is 3.

```
>>> list(nltk.ngrams(emma,4))[:5]
[('[', 'Emma', 'by', 'Jane'),
 ('Emma', 'by', 'Jane', 'Austen'),
 ('by', 'Jane', 'Austen', '1816'),
 ('Jane', 'Austen', '1816', ']'),
 ('Austen', '1816', ']', 'VOLUME')]
```

5 IR

Information Retrieval

Information Retrieval (IR)

- IR is about searching for information.
- IR typically means “document retrieval”.
- IR is one of the core components of Web search.

`http://boston.lti.cs.cmu.edu/classes/11-744/treclogo-c.gif`

Stages in an IR System

1: Indexing

- This stage is done off-line, prior to running any searches.
- The goal is to reduce the documents to a description: the indices.
- We want to optimise the representation: for example, ignore the terms that do not contribute.

2: Retrieval

- Use the indices to retrieve the documents (ignore the remaining information in the documents).
- We want retrieval to be fast.

6 Vectorization Part 1

Bag of Words Representation

Bag of words (BoW)

- At indexing time, a compact representation of the document is built.
- The document is seen as a bag of words.
- Information about word position is (often) discarded.
- Only the important words are kept.

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity. Recently, the bag-of-words model has also been used for computer vision. $\Longrightarrow$

{bag, bag-of-words, computer, disregarding, document, grammar, information, IR, keeping, language, model, multiplicity, multiset, natural, order, processing, representation, represented, retrieval, sentence, simplifying, text, vision, word, words}

Stop Words

Stop words

- A simple (but rarely-used) solution to determine important words is to keep a list of non-important words: the stop words.
- All stop words in a document are ignored.
- Stop words are language-specific.
- Typically, stop words are connecting words.

Stop words in NLTK

```
>>> from nltk.corpus import
    stopwords
>>> stop =
    stopwords.words('english')
>>> stop[:5]
['i', 'me', 'my', 'myself', 'we']
```

Term Frequency

- Usually, words that are not frequent are not important.
- Words that are too frequent may occur in most documents and therefore can't be used to discriminate among documents.
- *Usually, important words are in the middle.*

tf.idf

tf.idf

- Term frequency: If a word is very frequent in a document, it is important for the document.

  $tf(t,d) =$ frequency of word t in document d

- Inverse document frequency: If a word appears in many documents, it is not important for any of the documents.

  $$idf(t) = \log \frac{\text{number of documents}}{\text{number of documents that contain } t}$$

- $tf.idf$ combines these two characteristics.

  $$tf.idf(t,d) = tf(t,d) \times idf(t)$$

tf is a function of the term and the document, whereas idf is a function of the term, across all documents. To compute $tf.idf$ we need to have a collection of documents, otherwise idf is irrelevant.

Problems with Bag of Word Representations

BoW representations ignore important information such as:

Word position: "Australia beat New Zealand" is not the same as "New Zealand beat Australia"

Morphology: If you search for "table", a webpage that uses the word "tables" might be relevant.

Words with similar meanings: If you search for "truck", a webpage that uses the word "lorry" might be relevant.

Ambiguity: If you search for "Apple" you might be interested in the company and not in the fruit.

Still, BoW representations are very simple, fast, and often surprisingly good.

Beyond BoW Representations

- A simple way to account for (some) information about word positions is to use n-grams:
  - Bigrams, trigrams, 4-grams (usually there is no need for longer n-grams).
- Thus, instead of representing a text as a bag of words, it can be represented as a bag of n-grams.

From Documents/Sentences/Search Terms to Vectors

- We need to documents and sentences and search terms into vectors.
- The best way of doing this is with distributional semantics (a few weeks' time).
- The second-best way (and the most explainable) is to create a sparse matrix of the occurrence of a word/stem/n-gram/byte-pair-encoded in each document or sentence.
  - Weighting it using *tf.idf* is quite good.
  - Weighting it using other algorithms such as BM25 is marginally better

Example of Bag-of-Words Vector Space Model

Template:
{computer,software,information,document,retrieval,language,library,filtering}

Initial documents

D1:{computer, software, information, language}
D2:{computer, document, retrieval, library}
D3:{computer, information, filtering, retrieval}

Document vectors

D1: (1,1,1,0,0,1,0,0) D2: (1,0,0,1,1,0,1,0) D3: (1,0,1,0,1,0,0,1)

Document matrix

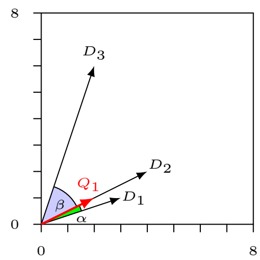
(typically a **sparse matrix**)

$$D = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Cosine Similarity

Cosine Method

- This is a popular approach to compare vectors.
- We calculate the cosine of the angle between vectors.
- If the angle is zero, then the cosine is 1.



$$\begin{aligned} \cos(D_1, Q_1) &= \cos(\alpha) \\ \cos(D_2, Q_1) &= \cos(0) = 1 \\ \cos(D_3, Q_1) &= \cos(\beta) \end{aligned}$$

Cosine Similarity: Formulas

General Formula

$$\cos(D_j, Q_k) = \frac{\Sigma_{i=1}^{N} D_{j,i} Q_{k,i}}{\sqrt{\Sigma_{i=1}^{N} D_{j,i}^2}\sqrt{\Sigma_{i=1}^{N} Q_{k,i}^2}} = \frac{D_j \cdot Q_k}{||D_j||_2 \, ||Q_k||_2}$$

If the vectors are normalised

$$\cos(D_j, Q_k) = \Sigma_{i=1}^{N} D_{j,i} Q_{k,i} = D_j \cdot Q_k$$

Vectorizing Jane Austen's "Emma"

```
#!/usr/bin/env python
import nltk
import numpy
emma_text =
    nltk.corpus.gutenberg.raw('austen-emma.txt')
emma_sentences =
    nltk.sent_tokenize(emma_text)
from sklearn.feature_extraction.text import
    TfidfVectorizer
from sklearn.metrics.pairwise import
    cosine_similarity
tfidf =
    TfidfVectorizer(stop_words='english',
    ngram_range=(1,2), min_df=1)
emma_sentences_as_vectors =
    tfidf.fit_transform(
    emma_sentences
)
print(emma_sentences_as_vectors.shape)
print(type(emma_sentences_as_vectors))
print(tfidf.get_feature_names_out()[1000:1005])
```

Making a search engine

```
query = input("Search for: ")
query_as_vector =
    tfidf.transform([query])
similarities =
    cosine_similarity(emma_sentences_as_vectors,
                      query_as_vector)
```

```
ranked_results =
    numpy.argsort(similarities,
    axis=0)[::-1]
match_found = False
for result_position in
    ranked_results[:3]:
    sentence_number =
        result_position[0]
    scoring =
        similarities[sentence_number]
    if scoring == 0.0: break
    match_found = True
    sentence =
        emma_sentences[sentence_number]
    print(sentence_number, scoring,
        sentence)
    if not match_found:
        print("No matches found")
```

Summary

- The NLTK library provides access to some public domain texts, and can tokenize words and sentences.
- Zipf's Law and Herdan's Law relate the number of words in a corpus with the number of distinct vocabulary items. These and other lexico-statistics can be used for author identification, and also let you estimate the size of the database index you will need for searching.
- When we say "words", that can mean almost anything.
- Byte-pair encoding is a way of getting word-like objects that you can use in other tasks.
- Bi-grams, tri-grams and n-grams are a quick hack that works quite well if you have a large volume of data to process and you want better search results without much effort.
- The bag-of-words and tf-idf vectorisation methods often work quite well, and produce easy-to-explain, easy-to-debug results.
- Stop words are words that you skip over (stop processing).
- One way of comparing two vectors is their cosine similarity

What's Next

Week 3

- Explainable methods
- Jurafsky and Martin: Chapter 5