Lecture 12

Natural Language Processing (NLP) and NLTK

Zoran B. Djordjević

Reference

- This lecture follows to a great measure:
  - *Natural Language Processing with Python*, 2nd Ed by Steven Bird, Ewan Klein and Edward Loper, O'Reilly 2015

  Note: One author of the above book, Steven Bird, appears to be the chief maintainer of the toolkit.

  Note: The 1st Edition is available on the Web. However, that edition contains references to a few functions that are not maintained and/or used any more.

- Somewhat useful reference if you want to test NLP with Spark:
  - *Advanced Analytics with Spark*, by Sandy Ryza et al., O'Reilly, 2015

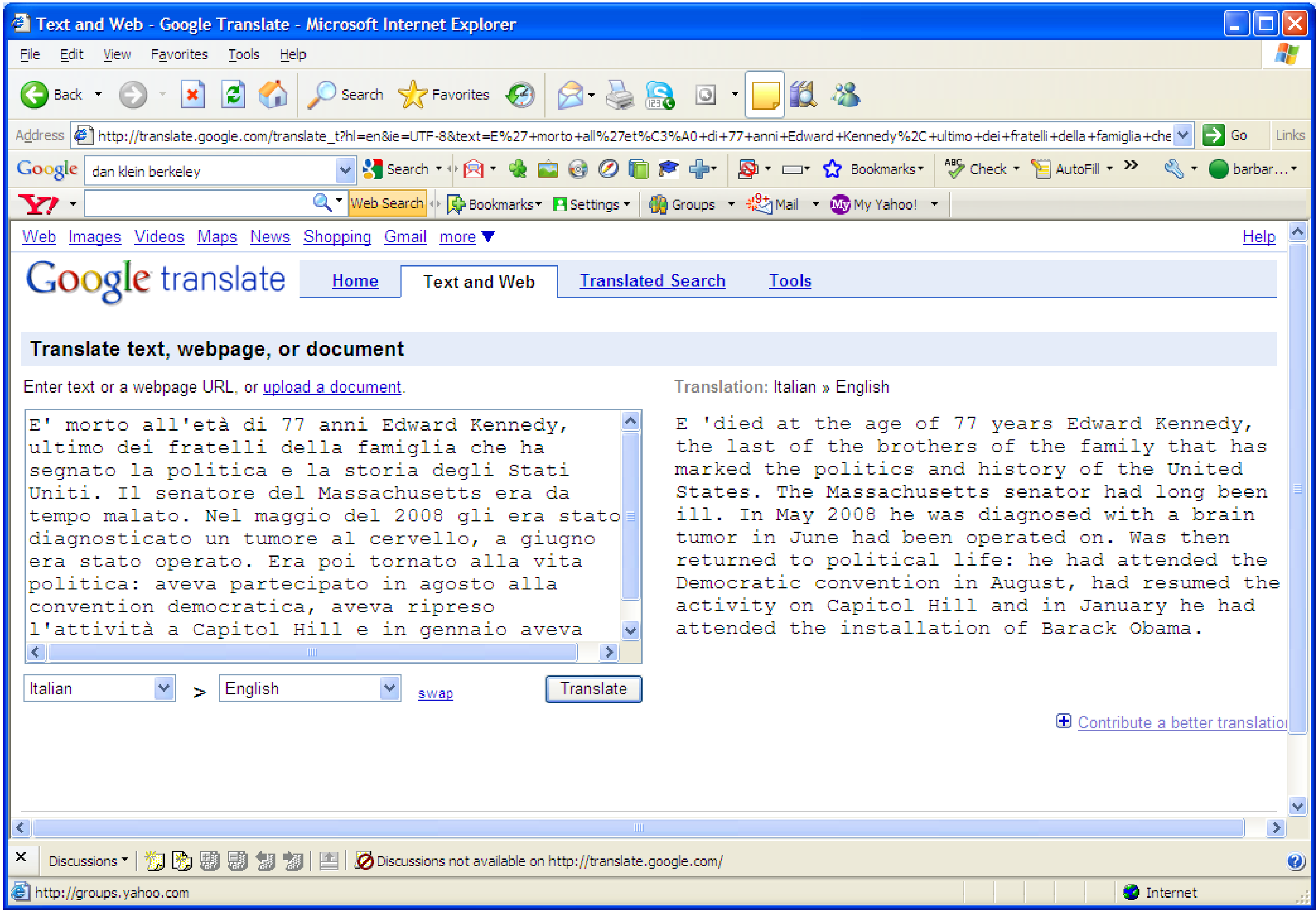
If you want to master NLP field, these are the essential books:

- *Foundations of Statistical Natural Language Processing*, Christopher Manning and Hinrich Schuetze, MIT Pess, 1999
- *Introduction to Information Retrieval*, Christopher D. Manning and Prabhakar Raghavan, Cambridge University Press, 2008
- *Speech and Language Processing, 2nd Ed.* Dan Jurafsky and James Martin, Prentice Hall, 2008.

Why NLP

- Large portion of (big) data software developers are asked to analyze are text data representing records in one of natural languages.
- Besides software developers, a wide range of people could benefit from having a working knowledge of NLP.
- This includes scientists working on human-computer interaction, business information analysts, marketing and political analysts, genomic scientists and web developers.
- Within academia, NLP is of greatest interest to scientiests in areas from humanities, computing and corpus linguistics through to computer science and artificial intelligence.
- To many people in academia, NLP is also known as “Computational Linguistics.”

Example: Machine Translation


Text and Web - Google Translate - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://translate.google.com/translate_t?hl=en&ie=UTF-8&text=E%27+morto+all%27et%C3%A0+di+77+anni+Edward+Kennedy%2C+ultimo+dei+fratelli+della+famiglia+che
dan klein berkeley
Web Images Videos Maps News Shopping Gmail more
Help
Google translate
Home
Text and Web
Translated Search
Tools

Translate text, webpage, or document

Enter text or a webpage URL, or upload a document.

```
E' morto all'età di 77 anni Edward Kennedy,
ultimo dei fratelli della famiglia che ha
segnato la politica e la storia degli Stati
Uniti. Il senatore del Massachusetts era da
tempo malato. Nel maggio del 2008 gli era stato
diagnosticato un tumore al cervello, a giugno
era stato operato. Era poi tornato alla vita
politica: aveva partecipato in agosto alla
convention democratica, aveva ripreso
l'attività a Capitol Hill e in gennaio aveva
```

Italian > English swap Translate

Translation: Italian » English

```
E 'died at the age of 77 years Edward Kennedy,
the last of the brothers of the family that has
marked the politics and history of the United
States. The Massachusetts senator had long been
ill. In May 2008 he was diagnosed with a brain
tumor in June had been operated on. Was then
returned to political life: he had attended the
Democratic convention in August, had resumed the
activity on Capitol Hill and in January he had
attended the installation of Barack Obama.
```

Contribute a better translatio

NLP applications

- Text Categorization
  - Classify documents by topics, language, author, spam filtering, information retrieval (relevant, not relevant), sentiment classification (positive, negative)
- Spelling & Grammar Corrections
- Information Extraction
- Speech Recognition
- Information Retrieval
  - Synonym Generation
- Summarization
- Machine Translation
- Question Answering
- Dialog Systems
  - Language generation

Why NLP is difficult

- A NLP system needs to answer the question “who did what to whom”
- Language is ambiguous
  - At all levels: lexical, phrase, semantic
  - Iraqi Head Seeks Arms
    - <u>Word sense</u> is ambiguous (head, arms)
  - Stolen Painting Found by Tree
    - <u>Thematic role</u> is ambiguous: tree is agent or location?
  - Ban on Nude Dancing on Governor’s Desk
    - <u>Syntactic structure (attachment</u>) is ambiguous: is the ban or the dancing on the desk?
  - Hospitals Are Sued by 7 Foot Doctors
    - <u>Semantics</u> is ambiguous : what is 7 foot?

Why NLP is difficult

- Language is flexible
  - New words, new meanings
  - Different meanings in different contexts
- Language is subtle
  - He arrived at the lecture
  - He chuckled at the lecture
  - He chuckled his way through the lecture
  - **He arrived his way through the lecture
- Language is complex!

Why NLP is difficult

- MANY hidden variables
  - Knowledge about the world
  - Knowledge about the context
  - Knowledge about human communication techniques
    - *Can you tell me the time?*
- Problem of scale
  - Many (infinite?) possible words, meanings, context
- Problem of sparsity
  - Very difficult to do statistical analysis, most things (words, concepts) are never seen before
- Long range correlations

Why NLP is difficult

- Key problems:
  - Representation of *meaning*
  - Language presupposes knowledge about the world
  - Language only reflects the surface of meaning
  - Language presupposes communication between people

Meaning

- What is meaning?
  - Physical referent in the real world
  - Semantic concepts, characterized also by relations.
- How do we represent and use meaning
  - I am *Italian*
    - *From lexical database* (WordNet)
    - *Italian =a native or inhabitant of Italy* → *Italy = republic in southern Europe [..]*
  - I am Italian
    - Who is “I”?
  - I know she is Italian/I think she is Italian
    - How do we represent “I know” and “I think”
    - Does this mean that I is Italian? What does it say about the “I” and about the person speaking?
  - I thought she was Italian
    - How do we represent tenses?

Why Python

- Python is a simple yet powerful programming language with excellent functionality for processing linguistic data.
- Python has no semicolons ☺. Python nests statements using tabs ☹
- Here is a five-line Python program that processes *file.txt* and prints all the words ending in `ing`:

```
>>> for line in open("file.txt"):
...     for word in line.split():
...         if word.endswith('ing'):
...                 print word
```

- More practical reason for using Python for NLP is in the fact that most books and software frameworks on the subject are published in Python lately.
- Do not despair, you could still find almost all of software needed for NLP in Java, C/C++ and even in C#.

Python NLTK

- NLTK defines an infrastructure that can be used to build NLP programs in Python.
- NLTK provides basic classes for representing data relevant to natural language processing; standard interfaces for performing tasks such as part-of-speech tagging, syntactic parsing, and text classification; and standard implementations for each task that can be combined to solve complex problems.
- NLTK comes with extensive documentation. In addition to many book, the website at *http://www.nltk.org/* provides API documentation that covers every module, class, and function in the toolkit, specifying parameters and giving examples of usage.
- The website also provides many HOWTOs with extensive examples and test cases, intended for Python developers.

Requirements

- *Python* as early as 2.4 has full NLTK. Perhaps what is coming with Python 3.4 is better.
- *NumPy* is a scientific computing library with support for multidimensional arrays and linear algebra, required for certain probability, tagging, clustering, and classification tasks.
- *Matplotlib* is a 2D plotting library for data visualization, and is useful for production of line graphs and bar charts.
- *NetworkX (optional)* is a library for storing and manipulating network structures consisting of nodes and edges.
- *Graphviz* (optional) is a library for visualizing semantic networks.
- *Prover9 (optional)* is an automated theorem prover for first-order and equational logic, used to support inference in language processing.
- *NLTK-Data is a* linguistic corpora (collection of documents) that are analyzed and processed in many examples and tests.

Natural Language Toolkit (NLTK)

- NLTK was originally created in 2001 as part of a computational linguistics course at the University of Pennsylvania. The following lists the most important modules.

Language processing task	NLTK modules	Functionality
Accessing corpora	nltk.corpus	Standardized interfaces to corpora and lexicons
String processing	nltk.tokenize nltk.stem	Tokenizers, Sentence stemmers
Collocation discovery	nltk.collocations	t-test chi-squared,point-wise, mutual information
Part-of-speech tagging	nltk.tag	n-gram, backoff, Brill HMM TnT
Classification	nltk.classify, nltk.cluster	Decision tree, maximum entropy, naive Bayes, EM, k-means
Chunking	nltk.chunk	Regular expression, n-gram, named entity
Parsing	nltk.parse	Chart feature-based, unification, probabilistic dependency
Semantic interpretation	nltk.sem, nltk.inference	Lambda calculu, first-order logic model, checking
Evaluation metrics	nltk.metrics	Precision recall, agreement coefficients
Probability and estimation	nltk.probability	Frequency distributions, smoothed probability distributions
Applications	nltk.app, nltk.chat	Graphical concordancer, parsers, WordNet browser, chatbots

Goals of Designers of NLTK

Simplicity

- To provide an intuitive framework along with substantial building blocks, giving users a practical knowledge of NLP without getting bogged down in the tedious house-keeping usually associated with processing annotated language data

Consistency

- To provide a uniform framework with consistent interfaces and data structures, and easily guessable method names

Extensibility

- To provide a structure into which new software modules can be easily accommodated, including alternative implementations and competing approaches to the same task

Modularity

- To provide components that can be used independently without needing to understand the rest of the toolkit.

The toolkit is not encyclopedic and does not include every imaginable functionality.

NLTK is not highly optimized and many tasks could be improved by creating efficient C or C++ routines.

Install NLTK

- Go to http://www.nltk.org/install.html
- Current version of NLTK requires Python 2.7 or 3.2+
- If you have Python run:

```
$ sudo pip install -U nltk
```

- On Windows download and run .exe installer. On other systems download appropriate tar.gz or zip file.
- Once NLTK is installed, open python and import nltk:

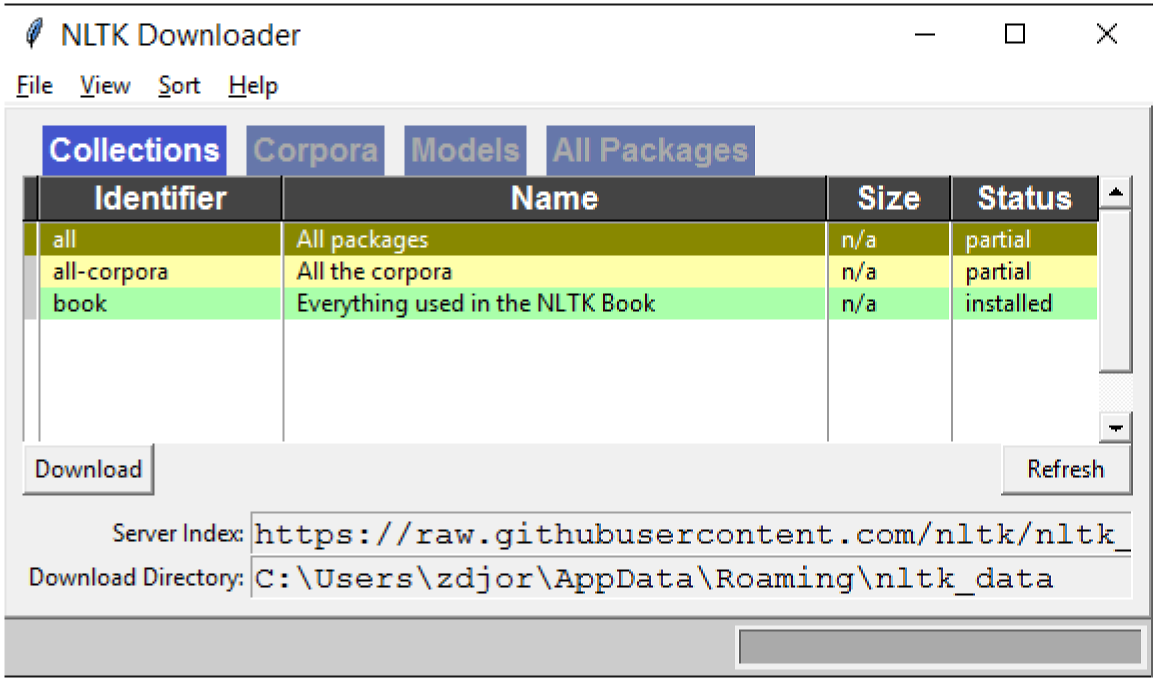
```
C:\> python
>>>  import nltk
>>>
```

Download NLTK Data

- Once you've installed NLTK, start up the Python interpreter as before, and install the sample data by typing the following two commands at the Python prompt:

```
>>> import nltk
>>> nltk.download()
```

- On the widget that pops-up select the book collection.



Identifier	Name	Size	Status
all	All packages	n/a	partial
all-corpora	All the corpora	n/a	partial
book	Everything used in the NLTK Book	n/a	installed

Server Index: https://raw.githubusercontent.com/nltk/nltk_

Download Directory: C:\Users\zdjor\AppData\Roaming\nltk_data

Importing book Collection

- Once the data is downloaded to your machine, you can load some of it using the Python interpreter. In the interpreter, type:

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
>>>
```

- Now you have some data to experiment with

Searching Text, Concordance

- There are many ways to examine the context of a text apart from simply reading it.
- A concordance view shows us every occurrence of a given word, together with some context. On Python prompt type:

```
>>> text1.concordance("monstrous")
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
that has survived the flood ; most monstrous and most mountainous ! That Himmal
they might scout at Moby Dick as a monstrous fable , or still worse and more de
th of Radney .'" CHAPTER 55 Of the Monstrous Pictures of Whales . I shall ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am strongly
ere to enter upon those still more monstrous stories of them which are to be fo
ght have been rummaged out of this monstrous cabinet there is no telling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u
```

- A concordance permits us to see words in context. For example, we saw that *monstrous* occurred in contexts such as *the* ___ *pictures* and *the* ___ *size*.

Searching Text, Similarity

- What other words appear in a similar range of contexts? We can find out by appending the term similar to the name of the text in question, then inserting the relevant word in parentheses:

```
text1.similar("monstrous")
Building word-context index...
subtly impalpable pitiable curious imperial perilous trustworthy
abundant untoward singular lamentable few maddens horrible loving lazy
mystifying christian exasperate puzzled

>>> text2.similar("monstrous")
Building word-context index...
very exceedingly so heartily a great good amazingly as sweet
remarkably extremely vast
```

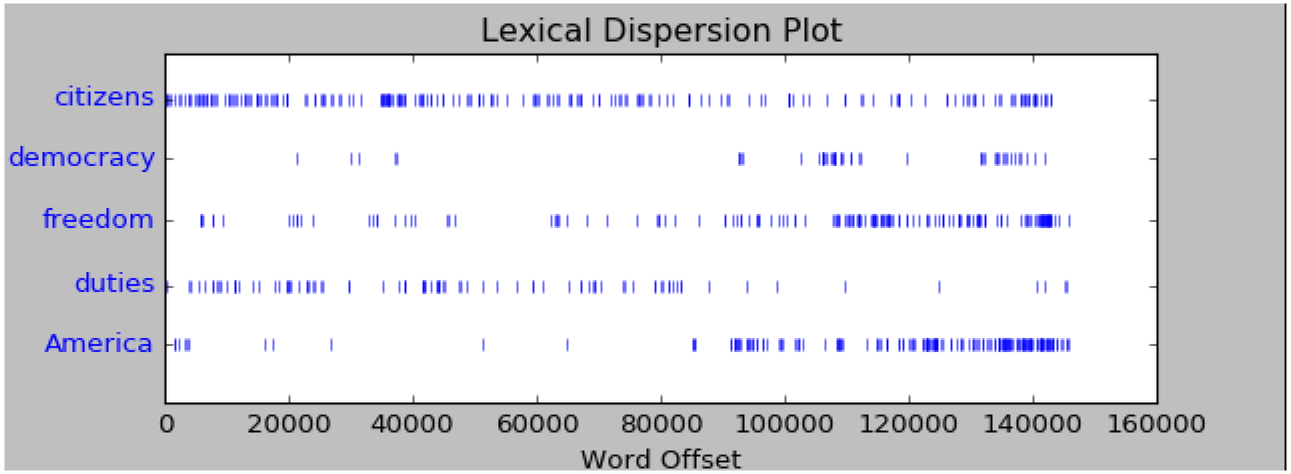
- We get different results for different texts. Austen uses this word quite differently from Melville; for her, *monstrous* has positive connotations, and sometimes functions as an intensifier like the word *very.*
- Function `common_contexts` allows us to examine just the contexts that are shared by two or more words, such as *monstrous* and *very*.

```
>>> text2.common_contexts(["monstrous", "very"])
be_glad is_pretty am_glad a_lucky a_pretty
>>> text1.common_contexts(["monstrous", "very"])
No common contexts were found
```

Dispersion Plot

- NLTK can also determine the *location* of a word in the text: how many words from the beginning it appears. This positional information can be displayed using a **dispersion plot**. Each stripe represents an instance of a word, and each row represents the entire text. Type:

```
>>> text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
```



Counting Vocabulary

- The most obvious fact about different texts that is that they differ in the vocabulary they use. NLTK can count the words in a text in a variety of useful ways.
- We find out the length of a text from start to finish, in terms of the words and punctuation symbols that appear by typing:

```
>>> len(text3)
44764
```

- So text3 has 44,764 words and punctuation symbols, or “tokens.”
- A **token** is the technical name for a sequence of characters.
- `len()` counts repeated word. `set()` will fetch only the words themselves

```
>>> sorted(set(text3))
['!', "'", '(', ')', ',', ',)', '.', '.)', ':', ';', ';)', '?', '?)',
'A', 'Abel', 'Abelmizraim', 'Abidah', 'Abide', 'Abimael', 'Abimelech',
'Abr', 'Abrah', 'Abraham', 'Abram', 'Accad', 'Achbor', 'Adah', ...]
>>> len(set(text3))
2789                          # there are only 2789 unique words
>>> >>> len(text3)/len(sorted(set(text3)))
16.050197203298673
>>>                           # a word is on average used 16 time in text3
```

Indexing Text

- Text in NLTK is a Python list `[ 'me','you',…]`
- `We can count the number of occurrences of each word:`

```
>>> tex1.count('heaven')
40
```

- We can find which work is on a particular position of the text:

```
>>> text1[5]
'Melville'
```

- Python permits us to access sublists as well, extracting manageable pieces of language from large texts, a technique known as **slicing**.

```
>>> text5[16715:16735]
['U86', 'thats', 'why', 'something', 'like', 'gamefly', 'is', 'so', 'good',
'because', 'you', 'can', 'actually', 'play', 'a', 'full', 'game', 'without',
'buying', 'it']
```

- We know that in Python indexes start from 0 not from 1.

```
>>> text4[:5]
['Fellow', '-', 'Citizens', 'of', 'the']
>>> text2[141525]
'among'
>>> text2[141525:]
['among', 'the', 'merits', 'and', 'the', 'happiness', 'of', 'Elinor', 'and',
'Marianne', ',', 'let', 'it', 'not', 'be', 'ranked', 'as', 'the', 'least',
'considerable', ',', 'that', 'though', 'sisters', ',', 'and', 'living', 'almost',
'within', 'sight', 'of', 'each', 'other', ',', 'they', 'could', 'live',
'without', 'disagreement', 'between', 'themselves', ',', 'or', 'producing',
'coolness', 'between', 'their', 'husbands', '.', 'THE', 'END']
```

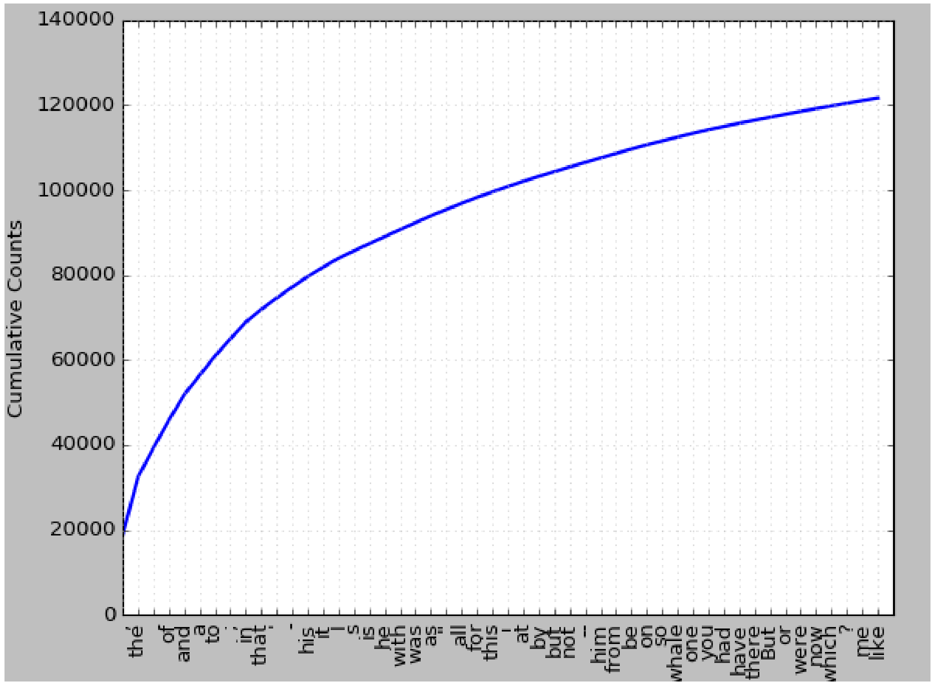
Frequency Distribution

- We would like to identify the words of a text that are most informative about the topic and genre of the text. Function `FreqDist()` does that for us:

```
>>> fdist1 = FreqDist(text1)
>>> fdist1
FreqDist({',': 18713, 'the': 13721, '.': 6862, 'of': 6536, 'and': 6024,
'a': 4569, 'to': 4542, ';': 4072, 'in': 3916, 'that': 2982, ...})
>>> fdist1.plot(50, cumulative=True)
```

- Only one word, *whale*, is slightly informative! It occurs over 900 times.
- The rest of the words tell us nothing about the text; they're just English "plumbing."
- What proportion of the text is taken up with such words? We can generate a cumulative frequency plot for these words, using `fdist1.plot(50, cumulative=True)`.
- Words occurring only once in the text:

```
>>> fdist1.hapaxes()
>>> len(fdist1.hapaxes)
9002
```



Find Long Words

- Let us say we want to find the most learned words in the text. Those must be the long ones:

```
>>> V = set(text1)
>>> long_words = [w for w in V if len(w) > 15]
>>> sorted(long_words)
['CIRCUMNAVIGATION', 'Physiognomically', 'apprehensiveness',
'cannibalistically',
'characteristically', 'circumnavigating', 'circumnavigation',
'circumnavigations',
'comprehensiveness', 'hermaphroditical', 'indiscriminately',
'indispensableness',
'irresistibleness', 'physiognomically', 'preternaturalness',
'responsibilities',
'simultaneousness', 'subterraneousness', 'supernaturalness',
'superstitiousness',
'uncomfortableness', 'uncompromisedness', 'undiscriminating',
'uninterpenetratingly']
```

Word Characterizing a Text

- Very long words frequently occur only once (there are a few of earned men) so they might not characterize a text well. Perhaps we should look for words that are not very short (longer than, for example 7 characters) and appear in a text more than a number of times (for example 5 times).

```
>>> text5
<Text: Chat Corpus>
>>>


>>> fdist5 = FreqDist(text5)
>>> sorted([w for w in set(text5) if len(w) > 7 and fdist5[w] > 7])
['#14-19teens', '#talkcity_adults', '((((((((((', '........', 'Question',
'actually', 'anything', 'computer', 'cute.-ass', 'everyone', 'football',
'innocent', 'listening', 'remember', 'seriously', 'something', 'together',
'tomorrow', 'watching']
```

Collocation and Bigrams

- A **collocation** is a sequence of words that occur together unusually often. Thus *red wine* is a collocation, whereas *the wine* is not. A characteristic of collocations is that they are resistant to substitution with words that have similar senses; for example, *blue wine* sounds very odd.
- To establish collocations, we start off by extracting from a text a list of word pairs, also known as **bigrams**. This is easily accomplished with the function bigrams():

```
>>> bigrams(['more', 'is', 'said', 'than', 'done'])
[('more', 'is'), ('is', 'said'), ('said', 'than'), ('than', 'done')]
>>>
```

- The pair of words *than-done* is a bigram, and we record it as `('than', 'done')`.
- Now, collocations are essentially just frequent bigrams that occur more often than we would expect based on the frequency of individual words.
- The `collocations()` function does this for us

```
>>> text4
<Text: Inaugural Address Corpus>
>>> text4.collocations()
Building collocations list
United States; fellow citizens; years ago; Federal Government; General
Government; American people; Vice President; Almighty God; Fellow
citizens; Chief Magistrate; Chief Justice; God bless; Indian tribes;
public debt; foreign nations; political parties; State governments;
```

- Collocations are characteristic of each text.

```
>>> text8
<Text: Personals Corpus>
>>> text8.collocations()
would like; medium build; social drinker; quiet nights; non smoker;
long term; age open; Would like; easy going; financially secure; fun
times; similar interests; Age open; weekends away; poss rship; well
presented; never married; single mum; permanent relationship; slim
build
```

Probabilistic Language Models

- Bigrams and similar constructs called n-grams are essential tools in computing the probability of a sentence or sequence of words:

$$P(W) = P(w_1, w_2, w_3, w_4, w_5 \ldots w_n)$$

- Related task: probability of an upcoming word:

  $P(w_5 | w_1, w_2, w_3, w_4)$

- A model that computes either of these:

  $P(W) \quad or \quad P(wn | w_1, w_2 \ldots wn\ _{-\ 1})$ is called a **language model**.

- Better: **the grammar** But **language model** or **LM** is standard
- We calculate above probabilities relying on so called chain rule.
- Recall the definition of conditional probabilities
- More variables:

$$P(A, B, C, D) = P(A)P(B|A)P(C|A, B)P(D|A, B, C)$$

- The Chain Rule in General

$$P(x_1, x_2, x_3, \ldots, x_n) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2) \ldots P(xn|x_1, \ldots, xn\ _{-\ 1})$$

Markov Assumption

- Markov Chain Assumption: we approximate each component in the product

$$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i \mid w_{i-k} \ldots w_{i-1})$$

- By a shorter chain immediately preceding every word

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-k} \ldots w_{i-1})$$

- The Simplest case is Unigram model which has no memory.

$$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i)$$

- A bit more precise is bigram model which knows what happened before every word:

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-1})$$

N-gram models

- The idea can be extend to trigrams, 4-grams, 5-grams
- In general this is an insufficient model of language
  - because language has **long-distance dependencies**:

  "The computer which I had just put into the machine room on the fifth floor crashed."
  (words computer and crashed are related and we know it)
- But we can often get away with N-gram models

- By the way you can get from Google actual, collected collection of n-grams (up to length 5).
- Python has a package to download that set:

```
$ pip install google-ngram-downloader
```

- On this site you can fetch ngram viewer
- http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

The Shannon Visualization Method

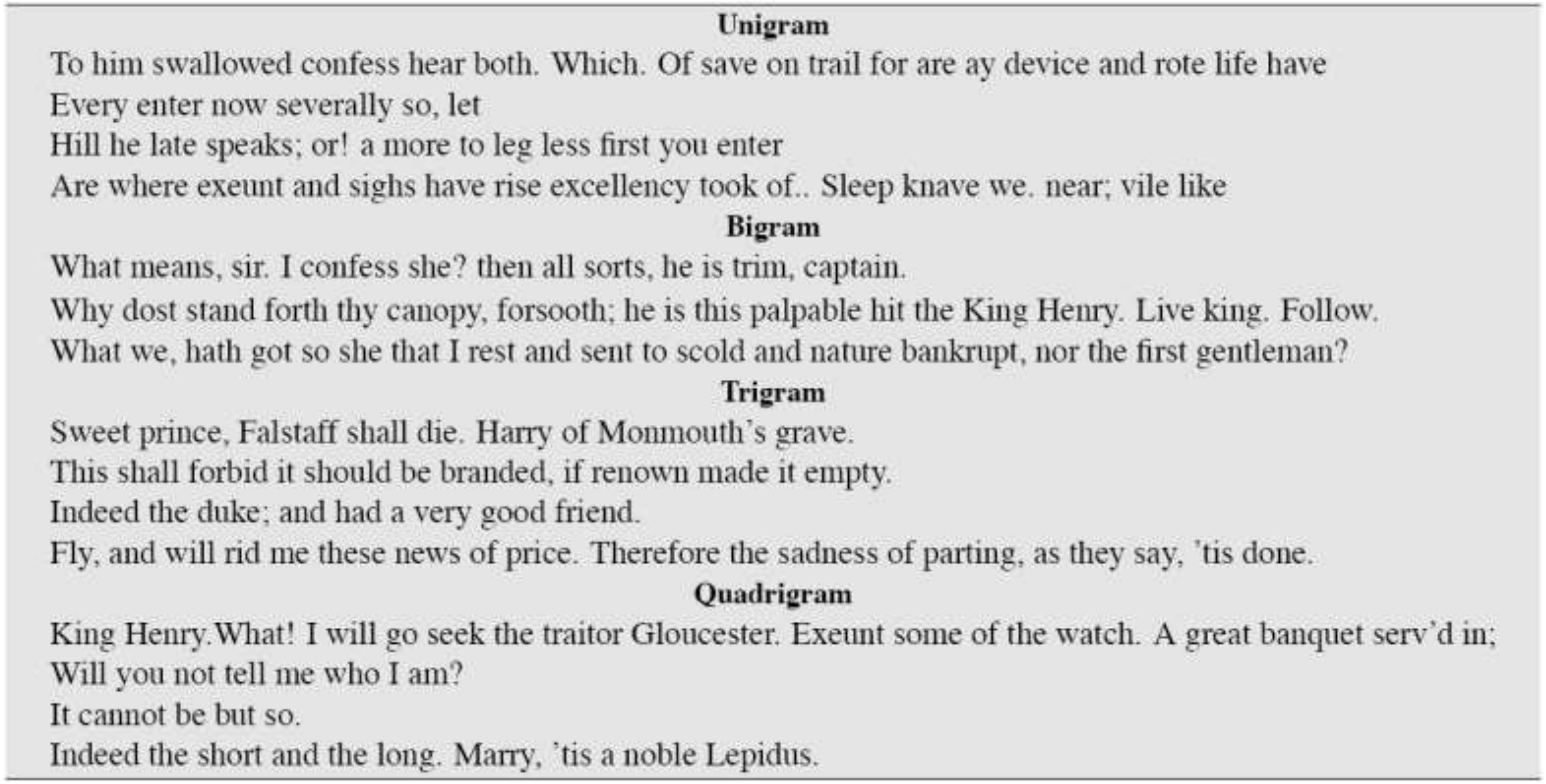
- Claude Shannon offered a systematic method for creating text using collections of n-grams and probabilities with which they occur.

- Choose a random bigram (<s>, w) according to its probability
- Now choose a random bigram (w, x) according to its probability
- And so on until we choose </s>
- Then string the words together

```
<s> I
    I want
      want to
           to eat
              eat Chinese
                  Chinese food
                          food  </s>
I want to eat Chinese food
```

Approximating Shakespeare

- The following are text generated using Shakespeare n-grams of different length.
- 4-grams (quadrigrams) are quite decent. You can use them in plays. Some people might not notice.



Unigram

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have
Every enter now severally so, let
Hill he late speaks; or! a more to leg less first you enter
Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like

Bigram

What means, sir. I confess she? then all sorts, he is trim, captain.
Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?

Trigram

Sweet prince, Falstaff shall die. Harry of Monmouth's grave.
This shall forbid it should be branded, if renown made it empty.
Indeed the duke; and had a very good friend.
Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.

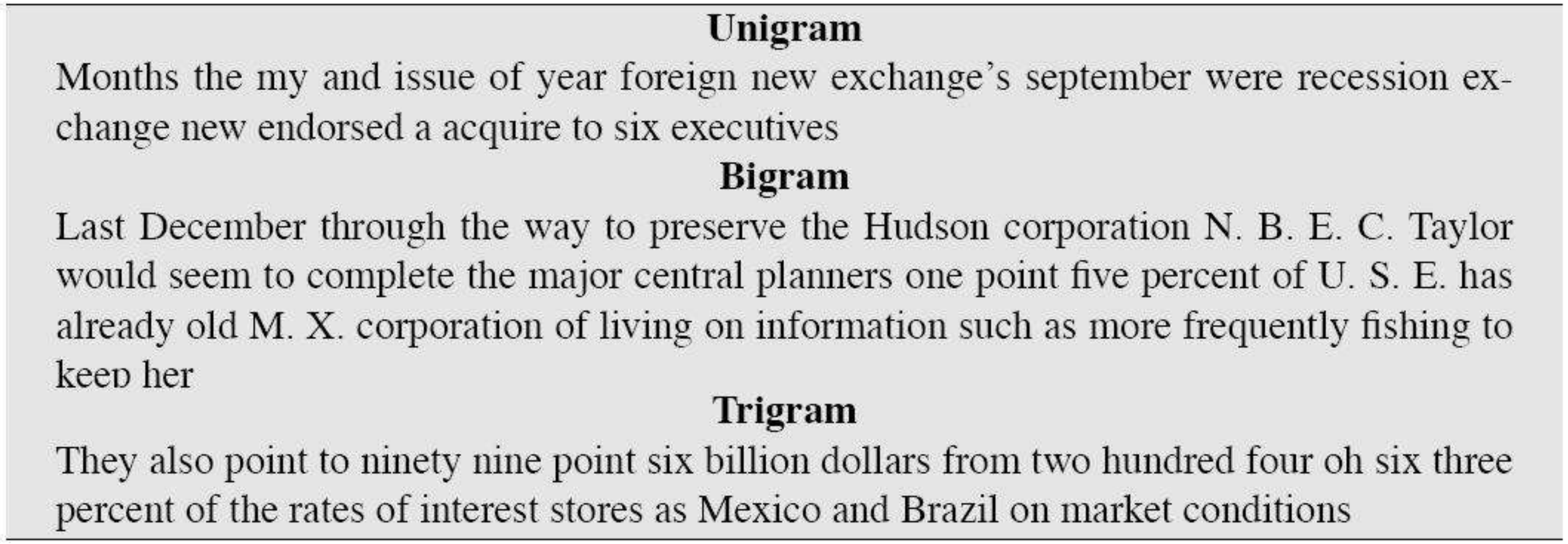
Quadrigram

King Henry.What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
Will you not tell me who I am?
It cannot be but so.
Indeed the short and the long. Marry, 'tis a noble Lepidus.

- Shakespeare corpus has N=884,647 tokens, V=29,066 unique words.
- Shakespeare produced 300,000 bigram types out of V^2= 844 million possible bigrams. So 99.96% of the possible bigrams were never seen (have zero entries in the table)

The Wall Street Journal is not Shakespeare

- Similar trick does not play that well with Wall Street Journal.



Unigram

Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives

Bigram

Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her

Trigram

They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Counting Other Things

- Counting words is useful, but we can count other things too. For example, we can look at the distribution of word lengths in a text, by creating a FreqDist out of a long list of numbers, where each number is the length of the corresponding word in the text:

```
>>> [len(w) for w in text1][:20]
[1, 4, 4, 2, 6, 8, 4, 1, 9, 1, 1, 8, 2, 1, 4, 11, 5, 2, 1, 7]
```

- `[len(w) for w in text1]` gives us the length of every word in the text. `[:20]` just displays the first 20 so we do not scroll forever. Next we establish the frequency distribution of those lengths

```
>>> fdist = FreqDist([len(w) for w in text1])
>>> fdist
FreqDist({3: 50223, 1: 47933, 4: 42345, 2: 38513, 5: 26597, 6: 17111, 7: 14399,
8: 9966, 9: 6428, 10: 3528, ...})
>>> fdist.keys()
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20])
>>>
>>> fdist.items()
[(3, 50223), (1, 47933), (4, 42345), (2, 38513), (5, 26597), (6, 17111), (7,
14399),
(8, 9966), (9, 6428), (10, 3528), (11, 1873), (12, 1053), (13, 567), (14, 177),
(15, 70), (16, 22), (17, 12), (18, 1), (20, 1)]
>>> fdist.max()
3
>>> fdist[3]       # the most frequent words are of length 3 (20% of the text)
50223
>>> fdist.freq(3)
0.19255882431878046
>>>
```

Functions defined for NLTK's frequency distributions

Example	Description
fdist = FreqDist(samples)	Create a frequency distribution containing the given samples
fdist.inc(sample)	Increment the count for this sample
fdist['monstrous']	Count of the number of times a given sample occurred
fdist.freq('monstrous')	Frequency of a given sample
fdist.N()	Total number of samples
fdist.keys()	The samples sorted in order of decreasing frequency
for sample in fdist:	Iterate over the samples in order of decreasing frequency
fdist.max()	Sample with the greatest count
fdist.tabulate()	Tabulate the frequency distribution
fdist.plot()	Graphical plot of the frequency distribution
fdist.plot(cumulative=True)	Cumulative plot of the frequency distribution
fdist1 < fdist2	Test if samples in fdist1 occur less frequently than in fdist2

Text Normalization

- Stemming
- Convert to lower case
- Identifying *non-standard words* including numbers, abbreviations, and dates, and mapping any such tokens to a special vocabulary.
  - For example, every decimal number could be mapped to a single token 0.0, and every acronym could be mapped to AAA. This keeps the vocabulary small and improves the accuracy of many language modeling tasks.
- Lemmatization
  - Make sure that the resulting form is a known word in a dictionary
  - WordNet lemmatizer only removes affixes if the resulting word is in its dictionary

Some comparison operators in Python

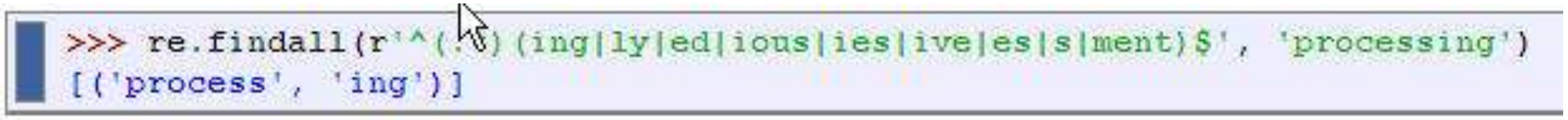
- In analysis of our texts we frequently perform various comparisons using functions and operators in the list bellow:

Function	Meaning
s.startswith(t)	Test if s starts with t
s.endswith(t)	Test if s ends with t
t in s	Test if t is contained inside s
s.islower()	Test if all cased characters in s are lowercase
s.isupper()	Test if all cased characters in s are uppercase
s.isalpha()	Test if all characters in s are alphabetic
s.isalnum()	Test if all characters in s are alphanumeric
s.isdigit()	Test if all characters in s are digits
s.istitle()	Test if s is titlecased (all words in s have initial capitals)
Operator	**Relationship**
<	Less than
<=	Less than or equal to
==	Equal to (note this is two "=" signs, not one)

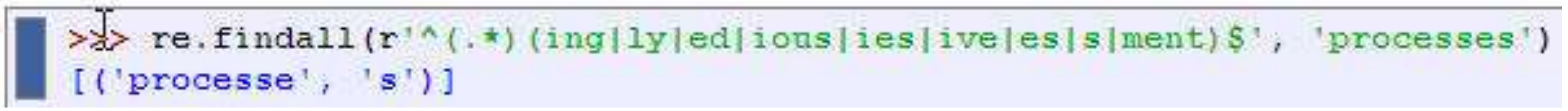
- Some examples of use of those are:

```
>>> sorted([w for w in set(text1) if w.endswith('ableness')])
['comfortableness', 'honourableness', 'immutableness',
'indispensableness', ...]
>>> sorted([term for term in set(text4) if 'gnt' in term])
['Sovereignty', 'sovereignties', 'sovereignty']
```

Regular Expressions for Stemming

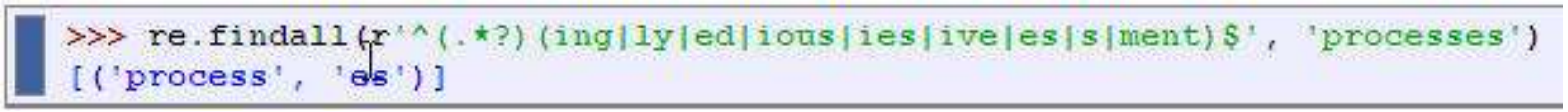


```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
```



```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
```

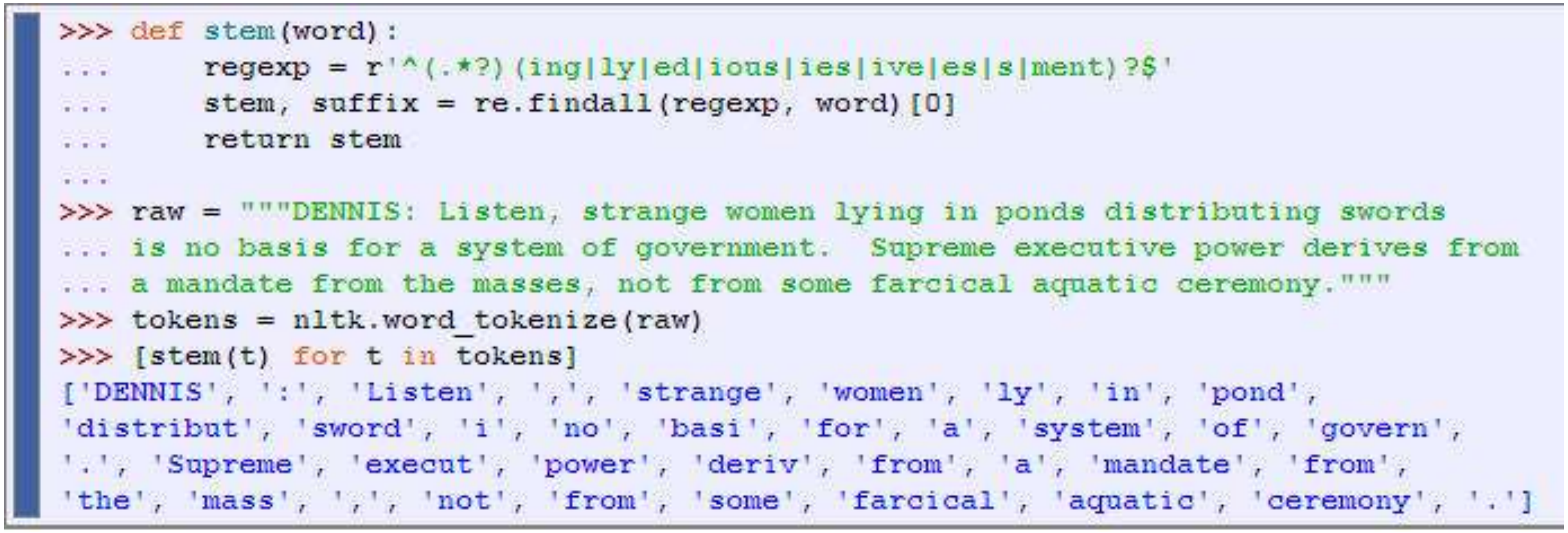
- Note: the star operator is "greedy" and the .* part of the expression tries to consume as much of the input as possible. If we use the "non-greedy" version of the star operator, written *?, we get what we want:



```
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
```

Regular Expressions for Stemming

- Let’s define a function to perform stemming, and apply it to a whole text

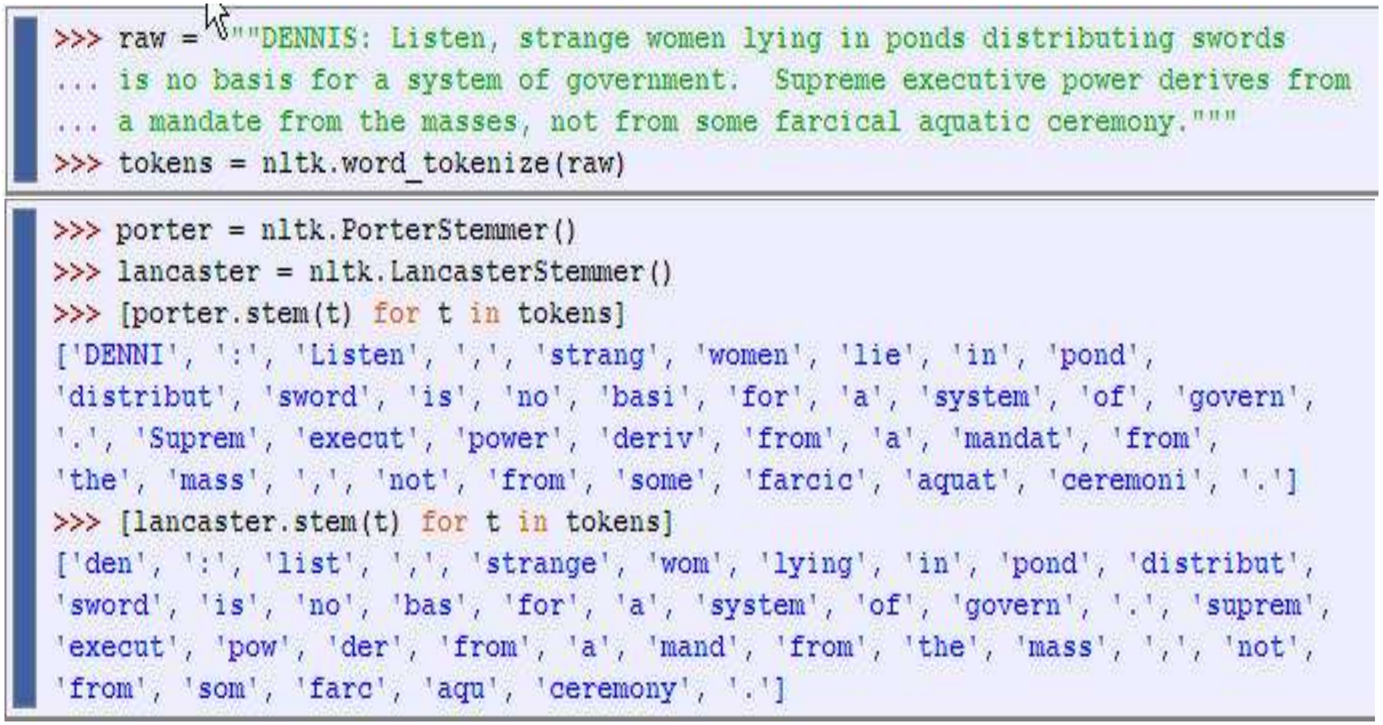


```
>>> def stem(word):
...     regexp = r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'
...     stem, suffix = re.findall(regexp, word)[0]
...     return stem
...
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government.  Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
>>> tokens = nltk.word_tokenize(raw)
>>> [stem(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'women', 'ly', 'in', 'pond',
'distribut', 'sword', 'i', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
'.', 'Supreme', 'execut', 'power', 'deriv', 'from', 'a', 'mandate', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcical', 'aquatic', 'ceremony', '.']
```

- The RE removed the *s* from *ponds* but also from *is* and *basis*. It produced some non-words like *distribut* and *deriv*, but these are acceptable stems in some applications.

NLTK Stemmers

- NLTK includes several off-the-shelf stemmers. The Porter and Lancaster stemmers follow their own rules for stripping affixes.
- Stemming is not a well-defined process, and we typically pick the stemmer that best suits the application we have in mind.



```
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government.  Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
>>> tokens = nltk.word_tokenize(raw)
```

```
>>> porter = nltk.PorterStemmer()
>>> lancaster = nltk.LancasterStemmer()
>>> [porter.stem(t) for t in tokens]
['DENNI', ':', 'Listen', ',', 'strang', 'women', 'lie', 'in', 'pond',
'distribut', 'sword', 'is', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
'.', 'Suprem', 'execut', 'power', 'deriv', 'from', 'a', 'mandat', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcic', 'aquat', 'ceremoni', '.']
>>> [lancaster.stem(t) for t in tokens]
['den', ':', 'list', ',', 'strange', 'wom', 'lying', 'in', 'pond', 'distribut',
'sword', 'is', 'no', 'bas', 'for', 'a', 'system', 'of', 'govern', '.', 'suprem',
'execut', 'pow', 'der', 'from', 'a', 'mand', 'from', 'the', 'mass', ',', 'not',
'from', 'som', 'farc', 'aqu', 'ceremony', '.']
```

Porter Stemmer

- Lexicon free stemmer
- Rewrite rules
  - ATIONAL → ATE (e.g. *relational, relate*)
  - FUL → ε (e.g. *hopeful, hope*)
  - SSES→ SS (e.g. *caresses, caress*)
- Errors of Commission
  - *Organization* → *organ*
  - *Policy* → *police*
- Errors of Omission
  - *Urgency* (not stemmed to *urgent*)
  - *European* (not stemmed to *Europe*)

NLTK Stemmers

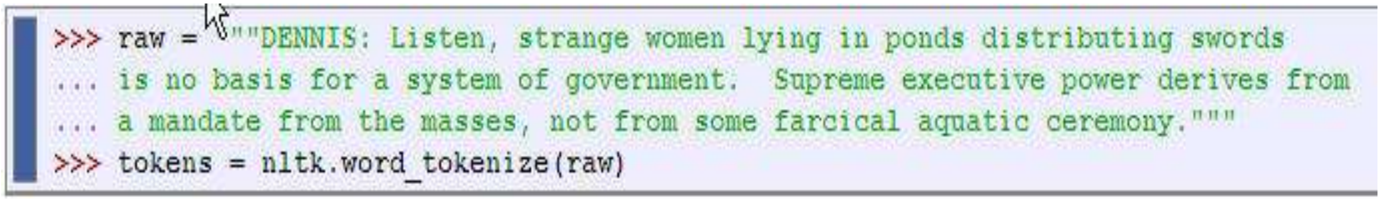
- nltk.wordnet.morphy
  - A slightly more sophisticated approach
  - Use an understanding of inflectional morphology
    - Use an Exception List for irregulars
    - Handle collocations in a special way
  - Do the transformation, compare the result to the WordNet dictionary
  - If the transformation produces a real word, then keep it, else use the original word.
  - For more details, see
    - http://wordnet.princeton.edu/man/morphy.7WN.html

Is stemming useful?

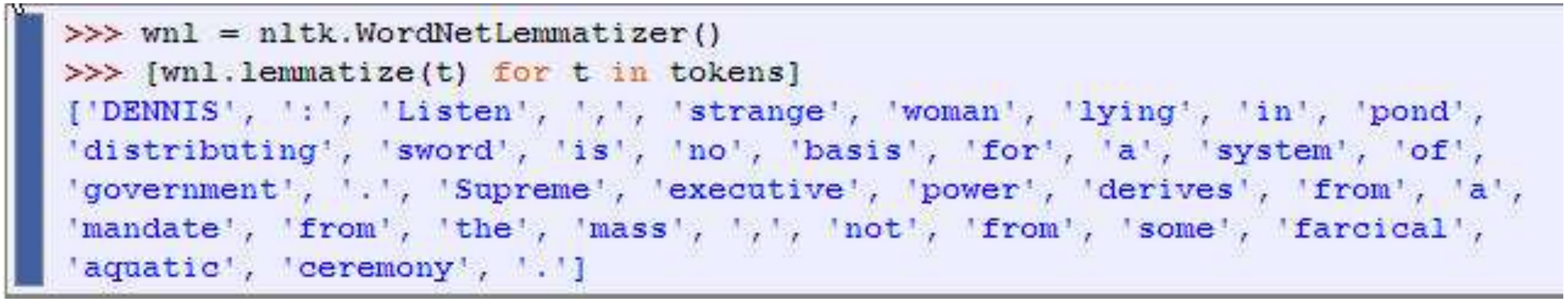
- For IR performance, some improvement (especially for smaller documents)
- May help a lot for some queries, but on average (across all queries) it doesn't help much (i.e. for some queries the results are worse)
  - Word sense disambiguation on query terms: *business* may be stemmed to *busy, saw (*the tool*) to see*
  - A truncated stem can be intelligible to users
  - Most studies for stemming for IR done for English (may help more for other languages)
  - The possibility of letting people interactively influence the stemming has not been studied much
- Since improvement is small, often IR engine usually don't use stemming
- More on this when we'll talk about IR

Lemmatization

- WordNet lemmatizer only removes affixes if the resulting word is in its dictionary
- The WordNet lemmatizer is a good choice if you want to compile the vocabulary of some texts and want a list of valid lemmas



```
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government.  Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
>>> tokens = nltk.word_tokenize(raw)
```



```
>>> wnl = nltk.WordNetLemmatizer()
>>> [wnl.lemmatize(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'woman', 'lying', 'in', 'pond',
'distributing', 'sword', 'is', 'no', 'basis', 'for', 'a', 'system', 'of',
'government', '.', 'Supreme', 'executive', 'power', 'derives', 'from', 'a',
'mandate', 'from', 'the', 'mass', ',', 'not', 'from', 'some', 'farcical',
'aquatic', 'ceremony', '.']
```

Notice that if doesn't handle *lying*, but it converts *women* to *woman*.

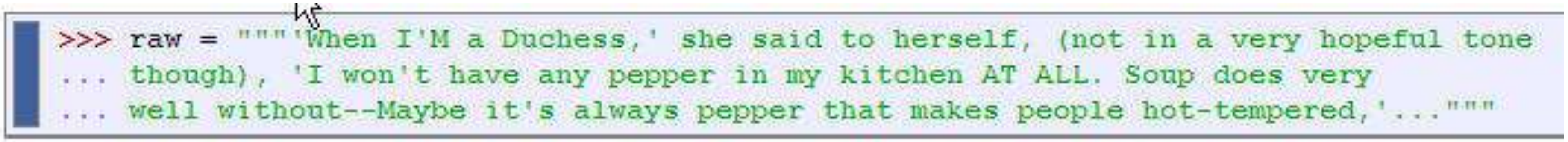
Tokenization

- Divide text into units called tokens (words, numbers, punctuations) (Manning)
- What is a word?
  - Graphic word: string of continuous alpha numeric character surrounded by white space
    - *$22.50*
  - Main clue (in English) is the occurrence of whitespaces
  - Problems
    - Periods: usually remove punctuation but sometimes it's useful to keep periods (*Wash*. → *wash*)
    - Single apostrophes, contractions (*isn't, didn't, dog's*: for meaning extraction could be useful to have 2 separate forms: *is* + *n't* or *not*)
    - Hyphenation:
      - Sometime best a single word: *co-operate*
      - Sometime best as 2 separate words: *26-year-old, aluminum-export ban*

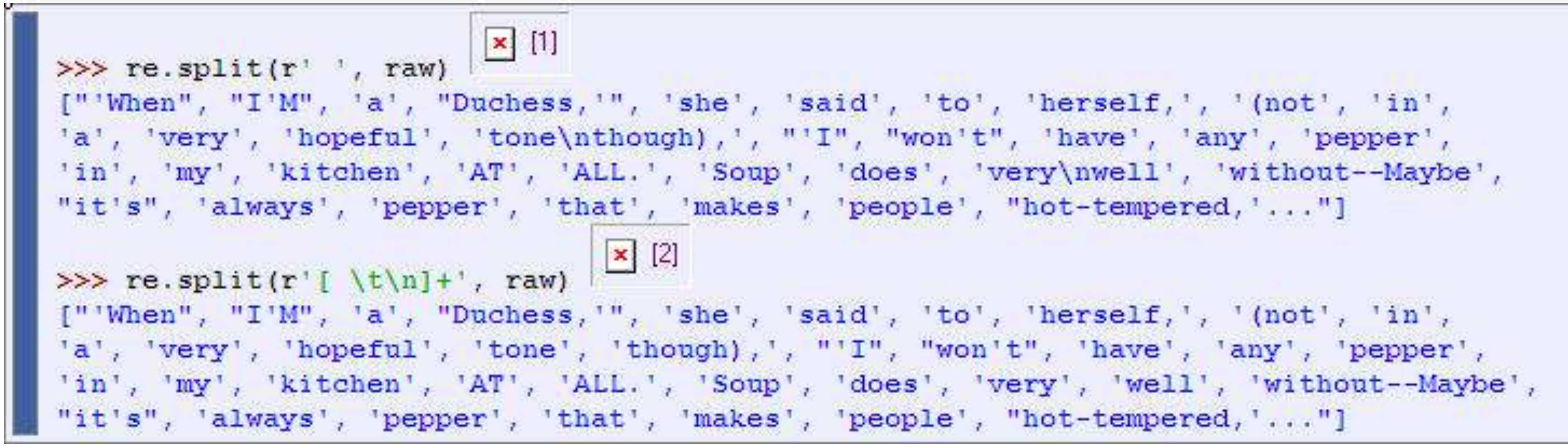
Tokenization

- Whitespace often do not indicate a word break: sometime we may want to lump together words that are separated by a white space (whitespace?) but that we want to regard as a single word
  - San Francisco
  - The New York-New Heaven railroad
  - Wake up, work out
    - I couldn’t *work* the answer *out*

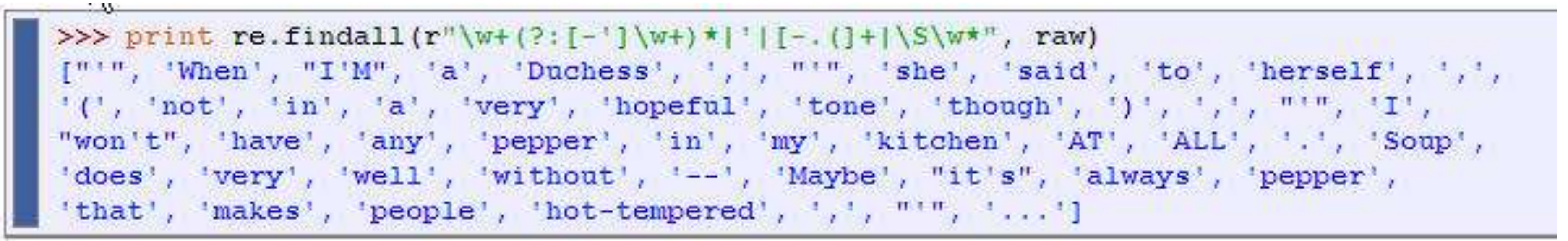
RE for Tokenizing Text



```
>>> raw = """'When I'M a Duchess,' she said to herself, (not in a very hopeful tone
... though), 'I won't have any pepper in my kitchen AT ALL. Soup does very
... well without--Maybe it's always pepper that makes people hot-tempered,'..."""
```

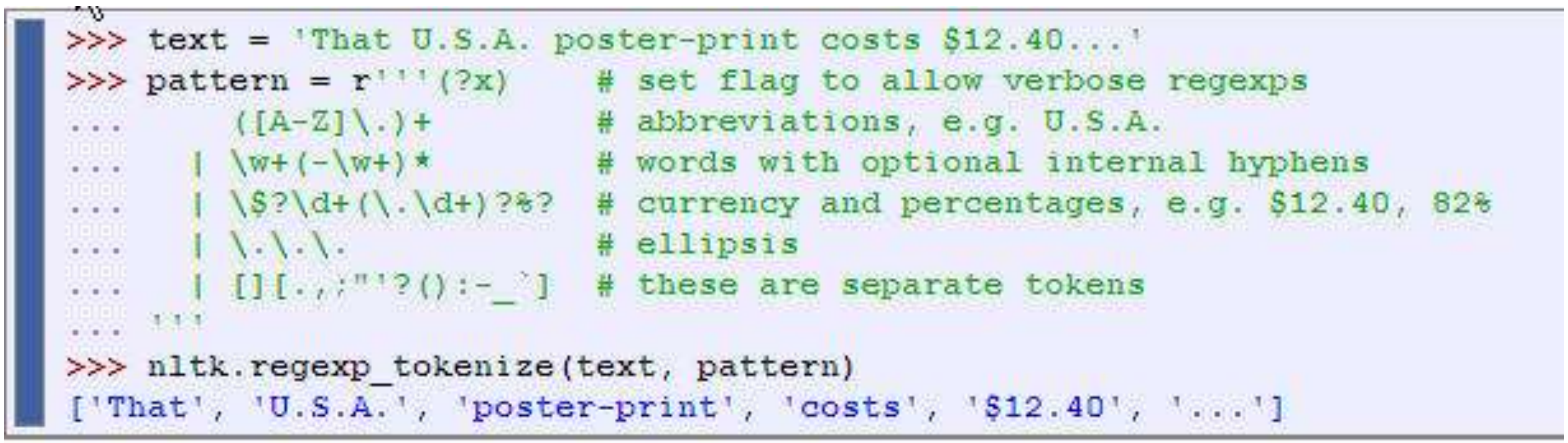


```
>>> re.split(r' ', raw)   [1]
["'When", "I'M", 'a', "Duchess,'", 'she', 'said', 'to', 'herself,', '(not', 'in',
'a', 'very', 'hopeful', 'tone\nthough),', "'I", "won't", 'have', 'any', 'pepper',
'in', 'my', 'kitchen', 'AT', 'ALL.', 'Soup', 'does', 'very\nwell', 'without--Maybe',
"it's", 'always', 'pepper', 'that', 'makes', 'people', "hot-tempered,'..."]

>>> re.split(r'[ \t\n]+', raw)   [2]
["'When", "I'M", 'a', "Duchess,'", 'she', 'said', 'to', 'herself,', '(not', 'in',
'a', 'very', 'hopeful', 'tone', 'though),', "'I", "won't", 'have', 'any', 'pepper',
'in', 'my', 'kitchen', 'AT', 'ALL.', 'Soup', 'does', 'very', 'well', 'without--Maybe',
"it's", 'always', 'pepper', 'that', 'makes', 'people', "hot-tempered,'..."]
```



```
>>> print re.findall(r"\w+(?:[-']\w+)*|'|[-.(]+|\S\w*", raw)
["'", 'When', "I'M", 'a', 'Duchess', ',', "'", 'she', 'said', 'to', 'herself', ',',
'(', 'not', 'in', 'a', 'very', 'hopeful', 'tone', 'though', ')', ',', "'", 'I',
"won't", 'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL', '.', 'Soup',
'does', 'very', 'well', 'without', '--', 'Maybe', "it's", 'always', 'pepper',
'that', 'makes', 'people', 'hot-tempered', ',', "'", '...']
```

NLTK's Regular Expression Tokenizer



```
>>> 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
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Tokenization

- Tokenization turns out to be a far more difficult task than you might have expected. No single solution works well across-the-board, and we must decide what counts as a token depending on the application domain.
- When developing a tokenizer it helps to have access to raw text which has been manually tokenized, in order to compare the output of your tokenizer with high-quality (or "gold-standard") tokens.
- The NLTK corpus collection includes a sample of Penn Treebank data, including the raw Wall Street Journal text (`nltk.corpus.treebank_raw.raw()`) and the tokenized version, `nltk.corpus.treebank.words()`

Segmentation

- Word segmentation
  - For languages that do not put spaces between words
    - Chinese, Japanese, Korean, Thai, German (for compound nouns)
- Sentence segmentation
  - Divide text into sentences
  - Why?

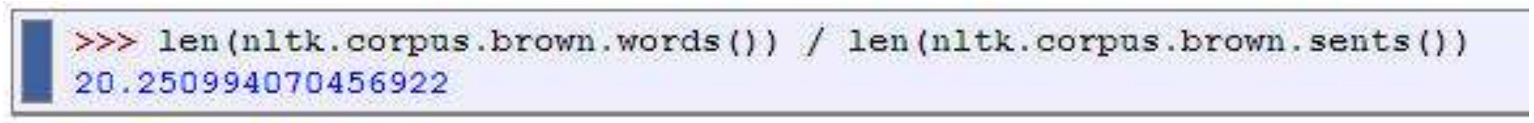
Sentence Segmentation

- Sentence:
  - Something ending with a .. ?, ! (and sometime also :)
  - “You reminded me,” she remarked, “of your mother.”
    - Nested sentences
    - Note the .”
- Sentence boundary detection algorithms
  - Heuristic (see Manning)
  - Statistical classification trees (Riley 1989)
    - Probability of a word to occur before or after a boundary, case and length of words
  - Neural network (Palmer and Hearst 1997)
    - Part of speech distribution of preceding and following words
  - Maximum Entropy (Mikheev 1998)

For reference see Manning

Sentence Segmentation NLTK tools

- Some corpora already provide access at the sentence level.
  - In the following example, we compute the average number of words per sentence in the Brown Corpus:



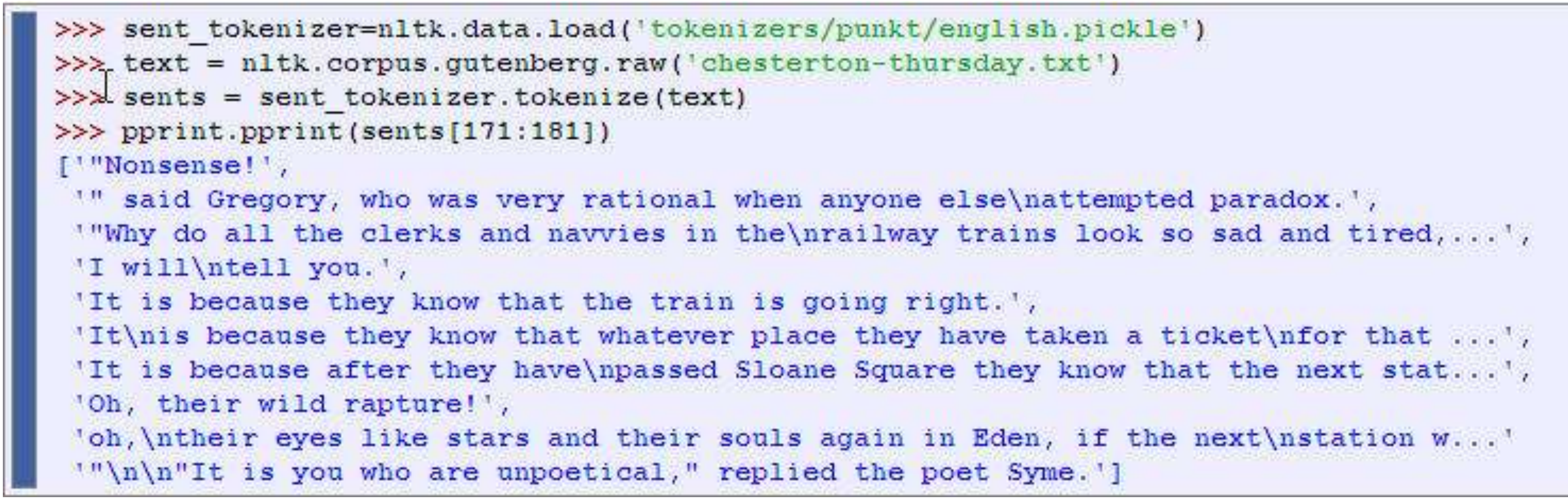
```
>>> len(nltk.corpus.brown.words()) / len(nltk.corpus.brown.sents())
20.250994070456922
```

Sentence Segmentation NLTK tools

- In other cases, the text is only available as a stream of characters. Before tokenizing the text into words, we need to segment it into sentences.
- NLTK facilitates this by including the Punkt sentence segmenter (Kiss & Strunk, 2006)* (*unsupervised language-independent, unsupervised approach to sentence boundary detection.)*
  - *It is based on the assumption that a large number of ambiguities in the determination of sentence boundaries can be eliminated once abbreviations have been identified. Instead of relying on orthographic clues, the proposed system is able to detect abbreviations with high accuracy using three criteria that only require information about the candidate type itself and are independent of context:*
    - *Abbreviations can be defined as a very tight collocation consisting of a truncated word and a final period*
    - *abbreviations are usually short,*
    - *abbreviations sometimes contain internal periods.*
  - **Example**
  - CELLULAR COMMUNICATIONS INC. sold 1,550,000 common shares at $21.75 each yesterday, according to lead underwriter L.F. Rothschild & Co.

*

Segmentation NLTK tools: Punkt sentence segmenter



```
>>> sent_tokenizer=nltk.data.load('tokenizers/punkt/english.pickle')
>>> text = nltk.corpus.gutenberg.raw('chesterton-thursday.txt')
>>> sents = sent_tokenizer.tokenize(text)
>>> pprint.pprint(sents[171:181])
['"Nonsense!',
 '" said Gregory, who was very rational when anyone else\nattempted paradox.',
 '"Why do all the clerks and navvies in the\nrailway trains look so sad and tired,...',
 'I will\ntell you.',
 'It is because they know that the train is going right.',
 'It\nis because they know that whatever place they have taken a ticket\nfor that ...',
 'It is because after they have\npassed Sloane Square they know that the next stat...',
 'Oh, their wild rapture!',
 'oh,\ntheir eyes like stars and their souls again in Eden, if the next\nstation w...'
 '"\n\n"It is you who are unpoetical," replied the poet Syme.']
```

*

Segmentation as classification

- Sentence segmentation can be viewed as a classification task for punctuation:
  - Whenever we encounter a symbol that could possibly end a sentence, such as a period or a question mark, we have to decide whether it terminates the preceding sentence.

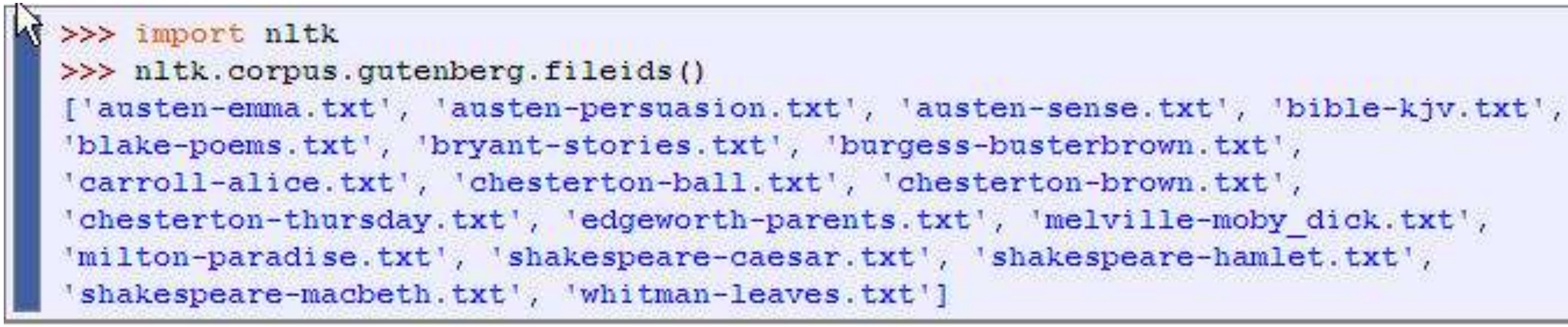
Corpora

- A text corpus is a large, structured collection of texts.
  - NLTK comes with many corpora

- The Open Language Archives Community (OLAC) provides an infrastructure for documenting and discovering language resource
  - OLAC is an international partnership of institutions and individuals who are creating a worldwide virtual library of language resources by:
    - (i) developing consensus on best current practice for the digital archiving of language resources, and
    - (ii) developing a network of interoperating repositories and services for housing and accessing such resources.
  - http://www.language-archives.org/

NLTK Corpora

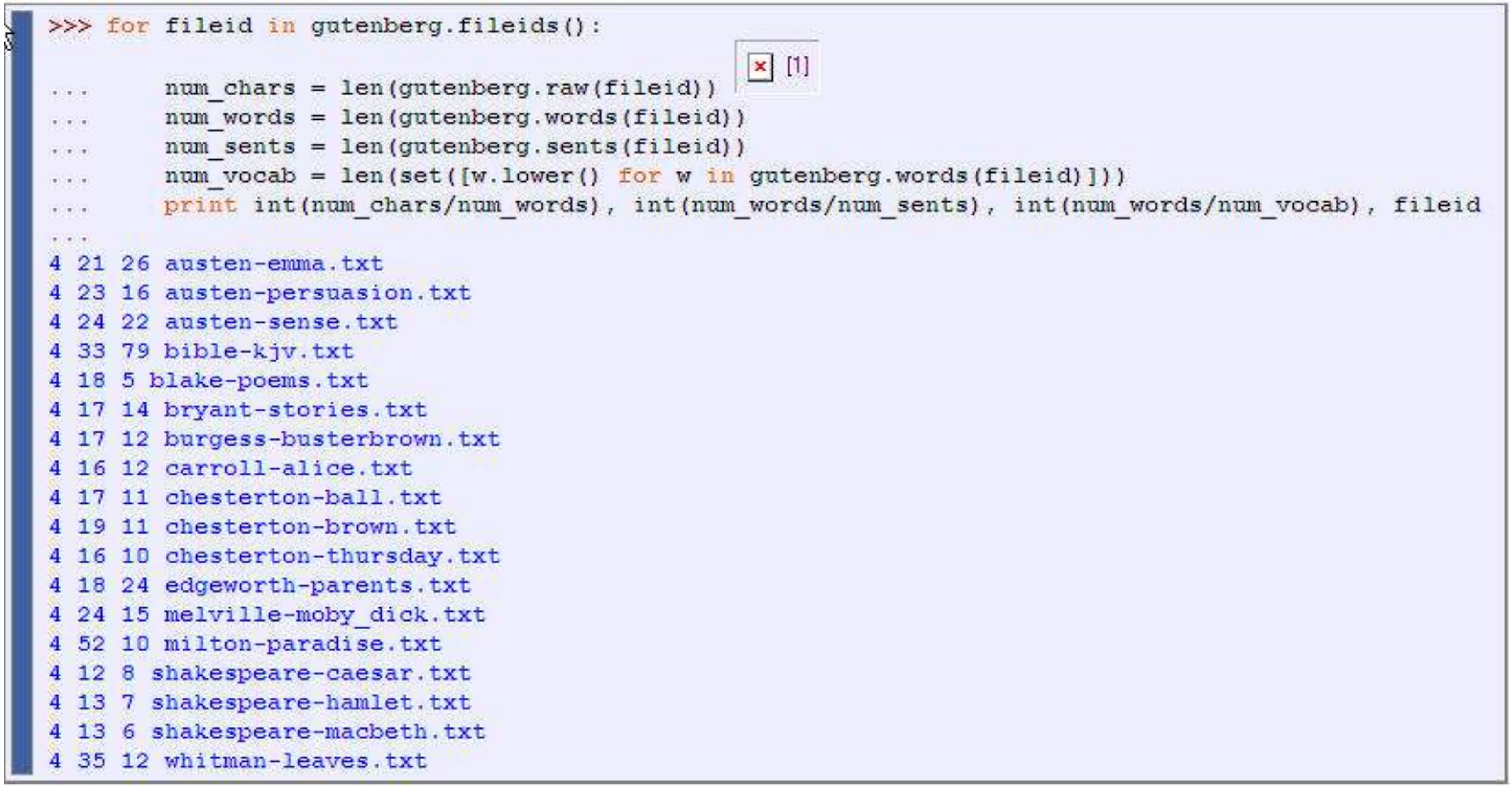
- **Gutenberg Corpus**
  - NLTK includes a small selection of texts from the Project Gutenberg electronic text archive (http://www.gutenberg.org), which contains some 25,000 free electronic books, and represents established literature
  - NLTK: we load the NLTK package, then ask to see the file identifiers in this corpus



```
>>> import nltk
>>> nltk.corpus.gutenberg.fileids()
['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', 'melville-moby_dick.txt',
'milton-paradise.txt', 'shakespeare-caesar.txt', 'shakespeare-hamlet.txt',
'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

NLTK Corpora

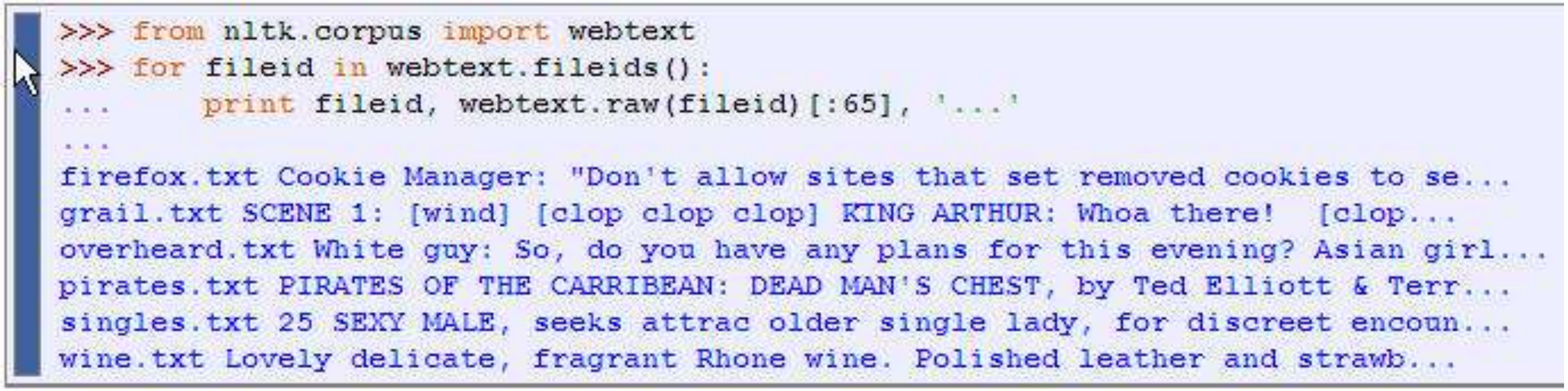
- Analyze the corpus!
  - Example: words(), raw(), and sents()
  - But also Conditional Frequency Distributions, Plotting and Tabulating Distributions



```
>>> for fileid in gutenberg.fileids():

...     num_chars = len(gutenberg.raw(fileid))
...     num_words = len(gutenberg.words(fileid))
...     num_sents = len(gutenberg.sents(fileid))
...     num_vocab = len(set([w.lower() for w in gutenberg.words(fileid)]))
...     print int(num_chars/num_words), int(num_words/num_sents), int(num_words/num_vocab), fileid
...
4 21 26 austen-emma.txt
4 23 16 austen-persuasion.txt
4 24 22 austen-sense.txt
4 33 79 bible-kjv.txt
4 18 5 blake-poems.txt
4 17 14 bryant-stories.txt
4 17 12 burgess-busterbrown.txt
4 16 12 carroll-alice.txt
4 17 11 chesterton-ball.txt
4 19 11 chesterton-brown.txt
4 16 10 chesterton-thursday.txt
4 18 24 edgeworth-parents.txt
4 24 15 melville-moby_dick.txt
4 52 10 milton-paradise.txt
4 12 8 shakespeare-caesar.txt
4 13 7 shakespeare-hamlet.txt
4 13 6 shakespeare-macbeth.txt
4 35 12 whitman-leaves.txt
```

Web and Chat Text

- NLTK contains less formal language as well; it's small collection of web text includes content from a Firefox discussion forum, conversations overheard in New York, the movie script of *Pirates of the Carribean*, personal advertisements, and wine reviews:



```
>>> from nltk.corpus import webtext
>>> for fileid in webtext.fileids():
...     print fileid, webtext.raw(fileid)[:65], '...'
...
firefox.txt Cookie Manager: "Don't allow sites that set removed cookies to se...
grail.txt SCENE 1: [wind] [clop clop clop] KING ARTHUR: Whoa there!  [clop...
overheard.txt White guy: So, do you have any plans for this evening? Asian girl...
pirates.txt PIRATES OF THE CARRIBEAN: DEAD MAN'S CHEST, by Ted Elliott & Terr...
singles.txt 25 SEXY MALE, seeks attrac older single lady, for discreet encoun...
wine.txt Lovely delicate, fragrant Rhone wine. Polished leather and strawb...
```

- There is also a corpus of instant messaging chat sessions with over 10,000 posts

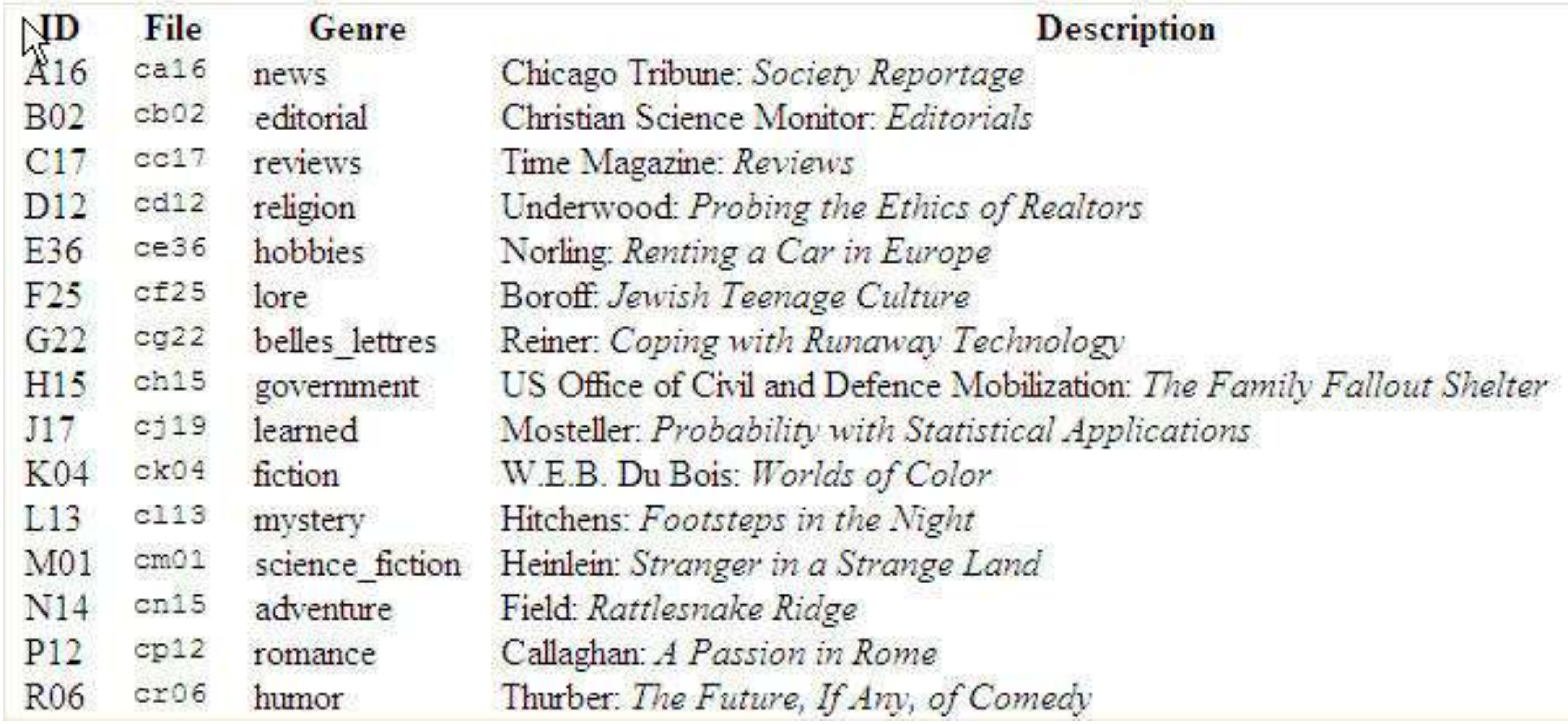
Annotated Text Corpora

- Many text corpora contain linguistic annotations, representing genres, POS tags, named entities, syntactic structures, semantic roles, and so forth.
- Not part of the text in the file; it explains something of the structure and/or semantics of text
- NLTK provides convenient ways to access several of these corpora
  - http://www.nltk.org/data
  - http://nltk.googlecode.com/svn/trunk/nltk_data/index.xml
  - Have a look!

Brown Corpus

- The Brown Corpus was the first million-word electronic corpus of English, created in 1961 at Brown University. This corpus contains text from 500 sources, and the sources have been categorized by genre, such as *news*, *editorial*, and so on.

Brown Corpus

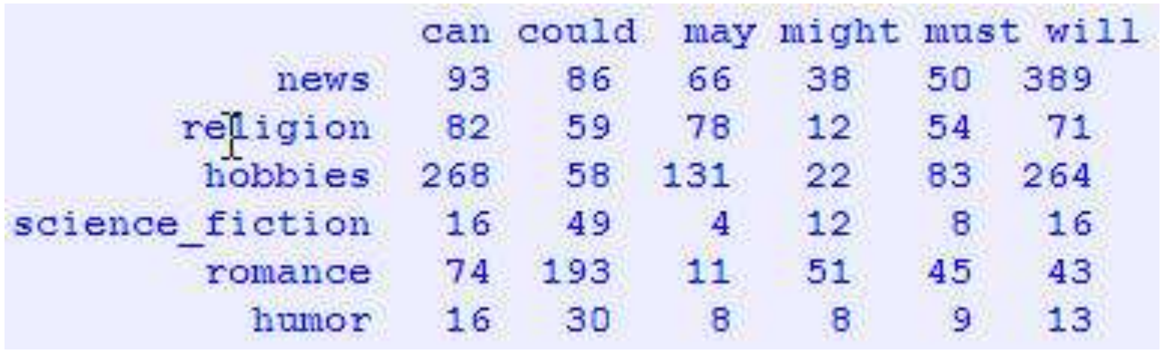


ID	File	Genre	Description
A16	ca16	news	Chicago Tribune: *Society Reportage*
B02	cb02	editorial	Christian Science Monitor: *Editorials*
C17	cc17	reviews	Time Magazine: *Reviews*
D12	cd12	religion	Underwood: *Probing the Ethics of Realtors*
E36	ce36	hobbies	Norling: *Renting a Car in Europe*
F25	cf25	lore	Boroff: *Jewish Teenage Culture*
G22	cg22	belles_lettres	Reiner: *Coping with Runaway Technology*
H15	ch15	government	US Office of Civil and Defence Mobilization: *The Family Fallout Shelter*
J17	cj19	learned	Mosteller: *Probability with Statistical Applications*
K04	ck04	fiction	W.E.B. Du Bois: *Worlds of Color*
L13	cl13	mystery	Hitchens: *Footsteps in the Night*
M01	cm01	science_fiction	Heinlein: *Stranger in a Strange Land*
N14	cn15	adventure	Field: *Rattlesnake Ridge*
P12	cp12	romance	Callaghan: *A Passion in Rome*
R06	cr06	humor	Thurber: *The Future, If Any, of Comedy*

- An example of each genre for the Brown Corpus
- (for a complete list, see http://icame.uib.no/brown/bcm-los.html)

Brown Corpus

- The Brown Corpus is a convenient resource for studying systematic differences between genres, a kind of linguistic inquiry known as **stylistics**.
- For example, we can compare genres in their usage of modal verbs:



	can	could	may	might	must	will
news	93	86	66	38	50	389
religion	82	59	78	12	54	71
hobbies	268	58	131	22	83	264
science_fiction	16	49	4	12	8	16
romance	74	193	11	51	45	43
humor	16	30	8	8	9	13

conditional frequency distributions of modal verbs conditioned on genre

Reuters Corpus

- The Reuters Corpus contains 10,788 news documents totaling 1.3 million words.
- The documents have been classified into 90 topics, and grouped into two sets, called "training" and "test“
  - This split is for training and testing algorithms that automatically detect the topic of a document
  - Unlike the Brown Corpus, categories in the Reuters corpus overlap with each other, simply because a news story often covers multiple topics.

Word Senses

- Words have multiple distinct meanings, or senses:
  - Plant: living plant, manufacturing plant, …
  - Title: name of a work, ownership document, form of address, material at the start of a film, …
- Many levels of sense distinctions
  - Homonymy: totally unrelated meanings (river bank, money bank)
  - Polysemy: related meanings (star in sky, star on tv, title)
  - Systematic polysemy: productive meaning extensions (metonymy such as organizations to their buildings) or metaphor
  - Sense distinctions can be extremely subtle (or not)
- Granularity of senses needed depends a lot on the task

Taken from

Word Sense Disambiguation (WSD)

- Determine which of the senses of an ambiguous word is invoked in a particular use of the word
- Example: living plant vs. manufacturing plant
- How do we tell these senses apart?
  - "Context"
    - The manufacturing plant which had previously sustained the town's economy shut down after an extended labor strike.
  - Maybe it's just text categorization
  - Each word sense represents a topic
- Why is it important to model and disambiguate word senses?
  - Translation
    - *Bank* → *banca* or *riva*
  - Parsing
    - For PP attachment, for example
  - information retrieval
    - To return documents with the right sense of *bank*

Adapted from

Resources

- WordNet
  - Hand-build (but large) hierarchy of word senses
  - Basically a hierarchical thesaurus
- SensEval
  - AWSD competition
  - Training / test sets for a wide range of words, difficulties, and parts-of-speech
  - Bake-off where lots of labs tried lots of competing approaches
- SemCor
  - A big chunk of the Brown corpus annotated with WordNet senses
- OtherResources
  - The Open Mind Word Expert
  - Parallel texts

Features

- Bag-of-words (use words around with no order)
  - *The manufacturing <u>plant</u> which had previously sustained the town's economy shut down after an extended labor strike.*
  - Bags of words = {*after, manufacturing, which, labor,* ..}
- Bag-of-words classification works ok for noun senses
  - 90% on classic, shockingly easy examples (line, interest, star)
  - 80% on senseval-1 nouns
  - 70% on senseval-1 verbs

Verb WSD

- Why are verbs harder?
  - Verbal senses less topical
  - More sensitive to structure, argument choice
  - Better disambiguated by their argument (subject-object): importance of local information
  - For nouns, a wider context likely to be useful
- Verb Example: “Serve”
  - [function] The tree stump serves as a table
  - [enable] The scandal served to increase his popularity
  - [dish] We serve meals for the homeless
  - [enlist] She served her country
  - [jail] He served six years for embezzlement
  - [tennis] It was Agassi's turn to serve
  - [legal] He was served by the sheriff

- Different types of information may be appropriate for different part of speech

Better features

- There are smarter features:
  - Argument selectional preference:
    - serve NP[meals] vs. serve NP[papers] vs. serve NP[country]
- Subcategorization:
  - [function] serve PP[as]
  - [enable] serve VP[to]
  - [tennis] serve <intransitive>
  - [food] serve NP {PP[to]}
  - Can capture poorly (but robustly) with local windows… but we can also use a parser and get these features explicitly
- Other constraints (Yarowsky 95)
  - One-sense-per-discourse
  - One-sense-per-collocation (pretty reliable when it kicks in:
  - manufacturing plant, flowering plant)

Various Approaches to WSD

- Unsupervised learning
  - We don’t know/have the labels
  - More than disambiguation is *discrimination*
    - Cluster into groups and discriminate between these groups without giving labels
    - Clustering
  - Example: EM (expectation-minimization), Bootstrapping (seeded with some labeled data)

- Indirect supervision
  - From thesauri
  - From WordNet
  - From parallel corpora

Supervised learning

- Supervised learning
  - When we know the truth (true senses) (not always true or easy)
  - Classification task
  - Most systems do some kind of supervised learning
  - Many competing classification technologies perform about the same (it's all about the knowledge sources you tap)
  - Problem: training data available for only a few words
  - Examples: **Bayesian classification**
    - **Naïve Bayes** (simplest example of Graphical models)
  - (We'll talk more about supervised learning/classification during the course)

Classification tasks

Assign the correct **class label** for a given input/object

In basic classification tasks, each input is considered in isolation from all other inputs, and the set of labels is defined in advance.

Examples:

Problem	Object	Label's categories
Tagging	Word	POS
Sense Disambiguation	Word	The word's senses
Information retrieval	Document	Relevant/not relevant
Sentiment classification	Document	Positive/negative
Text categorization	Document	Topics/classes
Author identification	Document	Authors
Language identification	Document	Language

Adapted from: Foundations of Statistical NLP (Manning et al)

Author identification

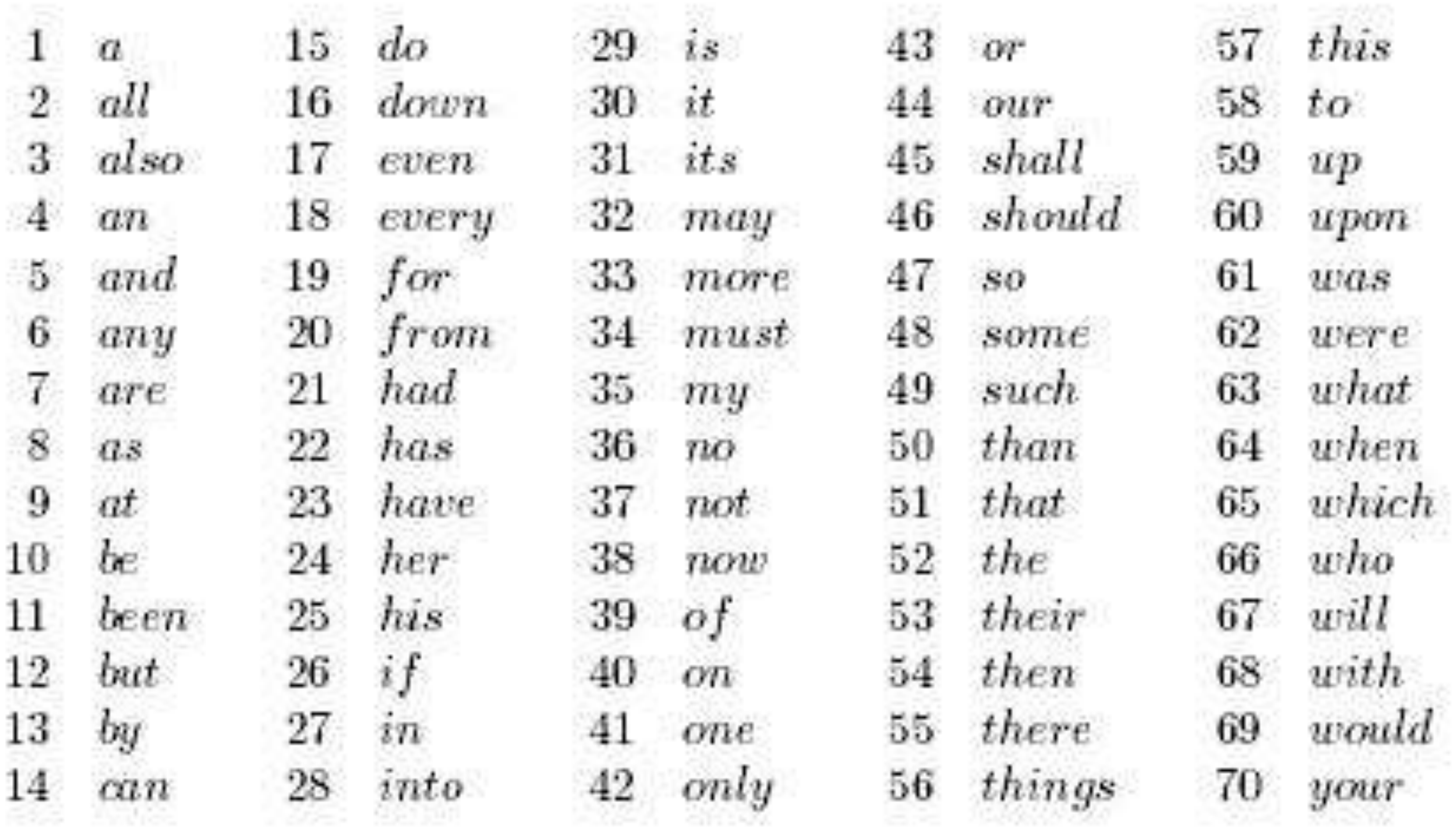
- They agreed that Mrs. X should only hear of the departure of the family, without being alarmed on the score of the gentleman's conduct; but even this partial communication gave her a great deal of concern, and she bewailed it as exceedingly unlucky that the ladies should happen to go away, just as they were all getting so intimate together.
- Gas looming through the fog in divers places in the streets, much as the sun may, from the spongey fields, be seen to loom by husbandman and ploughboy. Most of the shops lighted two hours before their time--as the gas seems to know, for it has a haggard and unwilling look. The raw afternoon is rawest, and the dense fog is densest, and the muddy streets are muddiest near that leaden-headed old obstruction, appropriate ornament for the threshold of a leaden-headed old corporation, Temple Bar.

Author identification

- Federalist papers
  - 77 short essays written in 1787-1788 by Hamilton, Jay and Madison to persuade NY to ratify the US Constitution; published under a pseudonym
  - The authorships of 12 papers was in dispute (*disputed papers*)
  - In 1964 Mosteller and Wallace* solved the problem
  - They identified 70 *function* words as good candidates for authorships analysis
  - Using statistical inference they concluded the author was Madison

Mosteller and Wallace 1964. Inference and Disputed Authorship: The Federalist.

Function words for Author Identification



1	*a*	15	*do*	29	*is*	43	*or*	57	*this*
2	*all*	16	*down*	30	*it*	44	*our*	58	*to*
3	*also*	17	*even*	31	*its*	45	*shall*	59	*up*
4	*an*	18	*every*	32	*may*	46	*should*	60	*upon*
5	*and*	19	*for*	33	*more*	47	*so*	61	*was*
6	*any*	20	*from*	34	*must*	48	*some*	62	*were*
7	*are*	21	*had*	35	*my*	49	*such*	63	*what*
8	*as*	22	*has*	36	*no*	50	*than*	64	*when*
9	*at*	23	*have*	37	*not*	51	*that*	65	*which*
10	*be*	24	*her*	38	*now*	52	*the*	66	*who*
11	*been*	25	*his*	39	*of*	53	*their*	67	*will*
12	*but*	26	*if*	40	*on*	54	*then*	68	*with*
13	*by*	27	*in*	41	*one*	55	*there*	69	*would*
14	*can*	28	*into*	42	*only*	56	*things*	70	*your*

Table 1: Function Words and Their Code Numbers

Function words for Author Identification

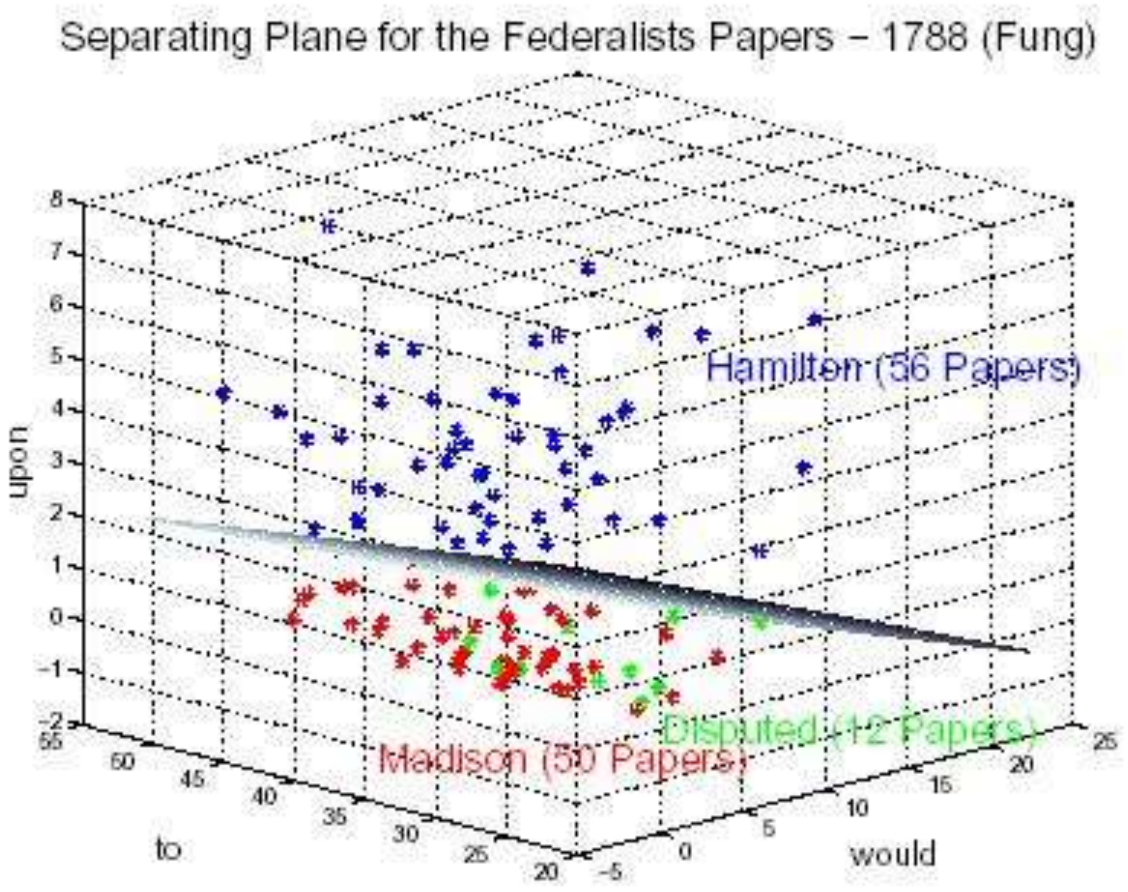


Figure 1: Obtained Hyperplane in 3 dimensions

Language identification

- Tutti gli esseri umani nascono liberi ed eguali in dignità e diritti. Essi sono dotati di ragione e di coscienza e devono agire gli uni verso gli altri in spirito di fratellanza.

- Alle Menschen sind frei und gleich an Würde und Rechten geboren. Sie sind mit Vernunft und Gewissen begabt und sollen einander im Geist der Brüderlichkeit begegnen.

Universal Declaration of Human Rights, UN, in 363 languages

Language identification

- égaux
- eguali
- iguales
- edistämään
- Ü
- ¿
- How to do determine, for a stretch of text, which language it is from?
- Turns out to be really simple
- Just a few character bigrams can do it (Sibun & Reynar 96)
  - Using special character sets helps a bit, but barely

Text categorization

- Topic categorization: classify the document into semantics topics

The U.S. swept into the Davis Cup final on Saturday when twins Bob and Mike Bryan defeated Belarus's Max Mirnyi and Vladimir Voltchkov to give the Americans an unsurmountable 3-0 lead in the best-of-five semi-final tie.

One of the strangest, most relentless hurricane seasons on record reached new bizarre heights yesterday as the plodding approach of Hurricane Jeanne prompted evacuation orders for hundreds of thousands of Floridians and high wind warnings that stretched 350 miles from the swamp towns south of Miami to the historic city of St. Augustine.

Text Categorization Applications

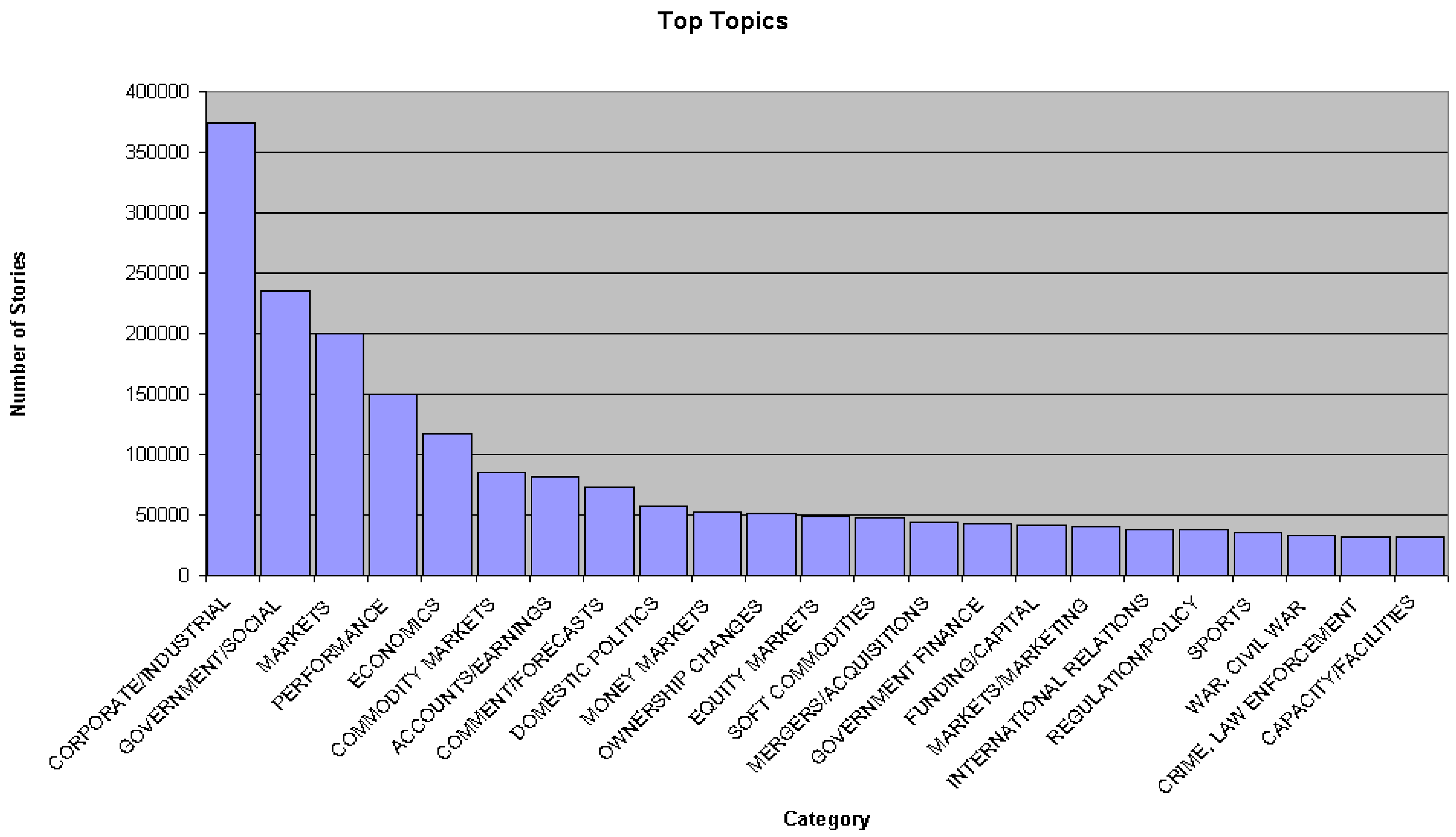
- Web pages organized into category hierarchies
- Journal articles indexed by subject categories (e.g., the Library of Congress, MEDLINE, etc.)
- Patents archived using *International Patent Classification*
- Patient records coded using international insurance categories
- E-mail message filtering
- Spam vs. anti-palm
- Customer service message classification
- News events tracked and filtered by topics

News topic categorization

- http://news.google.com/
- Reuters
  - Gold standard
  - Collection of (21,578) newswire documents.
  - For research purposes: a standard text collection to compare systems and algorithms
  - 135 valid topics categories

Reuters

- Top topics in Reuters



Reuters

<REUTERS TOPICS="YES" LEWISSPLIT="TRAIN" CGISPLIT="TRAINING-SET" OLDID="12981" NEWID="798">

<DATE> 2-MAR-1987 16:51:43.42</DATE>

<TOPICS><D>livestock</D><D>hog</D></TOPICS>

<TITLE>AMERICAN PORK CONGRESS KICKS OFF TOMORROW</TITLE>

<DATELINE>   CHICAGO, March 2 - </DATELINE><BODY>The American Pork Congress kicks off

tomorrow, March 3, in Indianapolis with 160 of the nations pork producers from 44 member states determining industry positions on a number of issues, according to the National Pork Producers Council, NPPC.

Delegates to the three day Congress will be considering 26 resolutions concerning various issues, including the future direction of farm policy and the tax law as it applies to the agriculture sector. The delegates will also debate whether to endorse concepts of a national PRV (pseudorabies virus) control and eradication program, the NPPC said.

A large trade show, in conjunction with the congress, will feature the latest in technology in all areas of the industry, the NPPC added. Reuter

&#3;</BODY></TEXT></REUTERS>

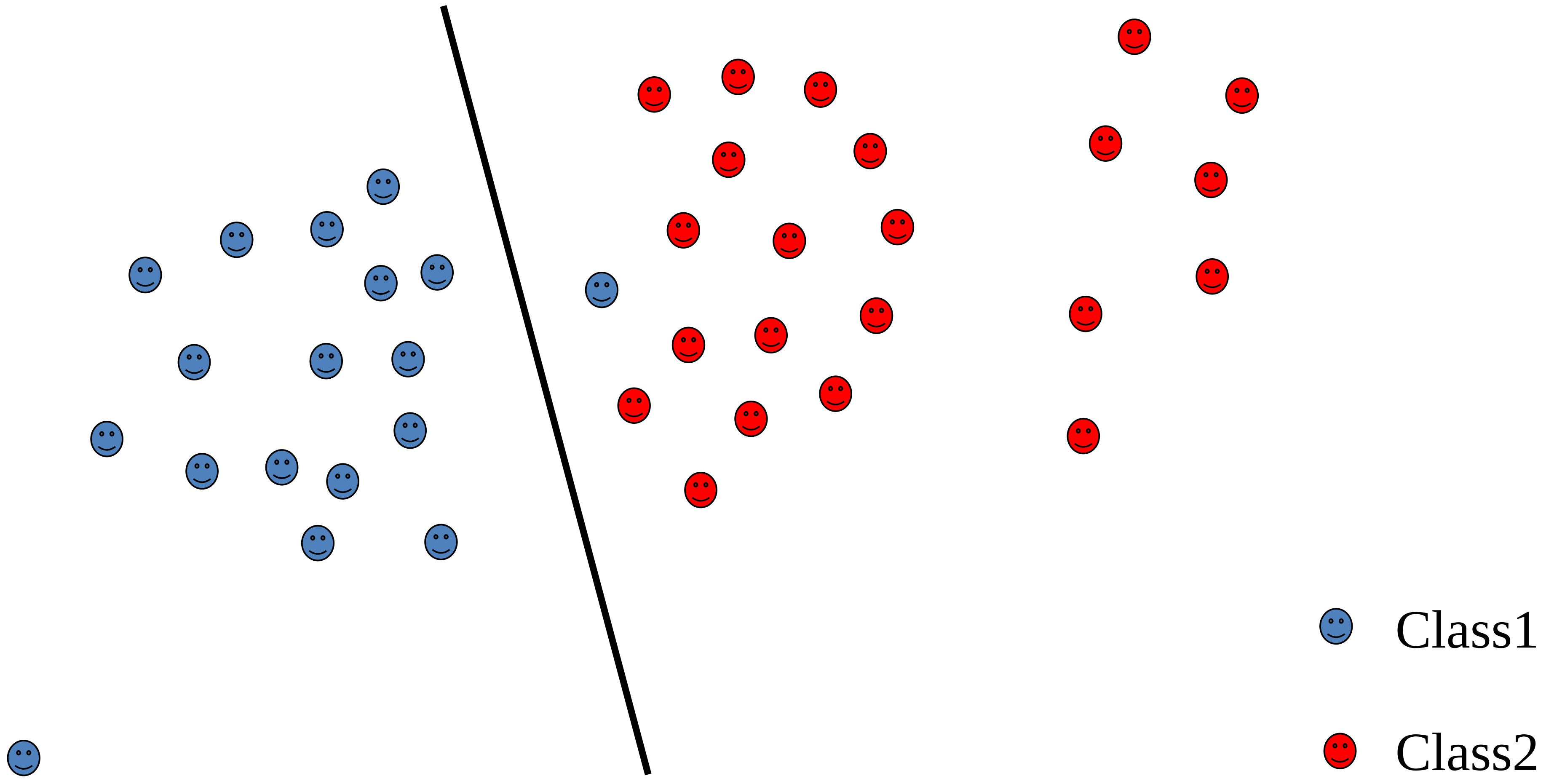
Outline

- Classification tasks
- Various issues regarding classification
  - Clustering vs. classification, binary vs. multi-way, flat vs. hierarchical classification, variants…
- Introduce the steps necessary for a classification task
  - Define classes (aka labels)
  - Label text
  - Define and extract features
  - Training and evaluation
- NLTK example

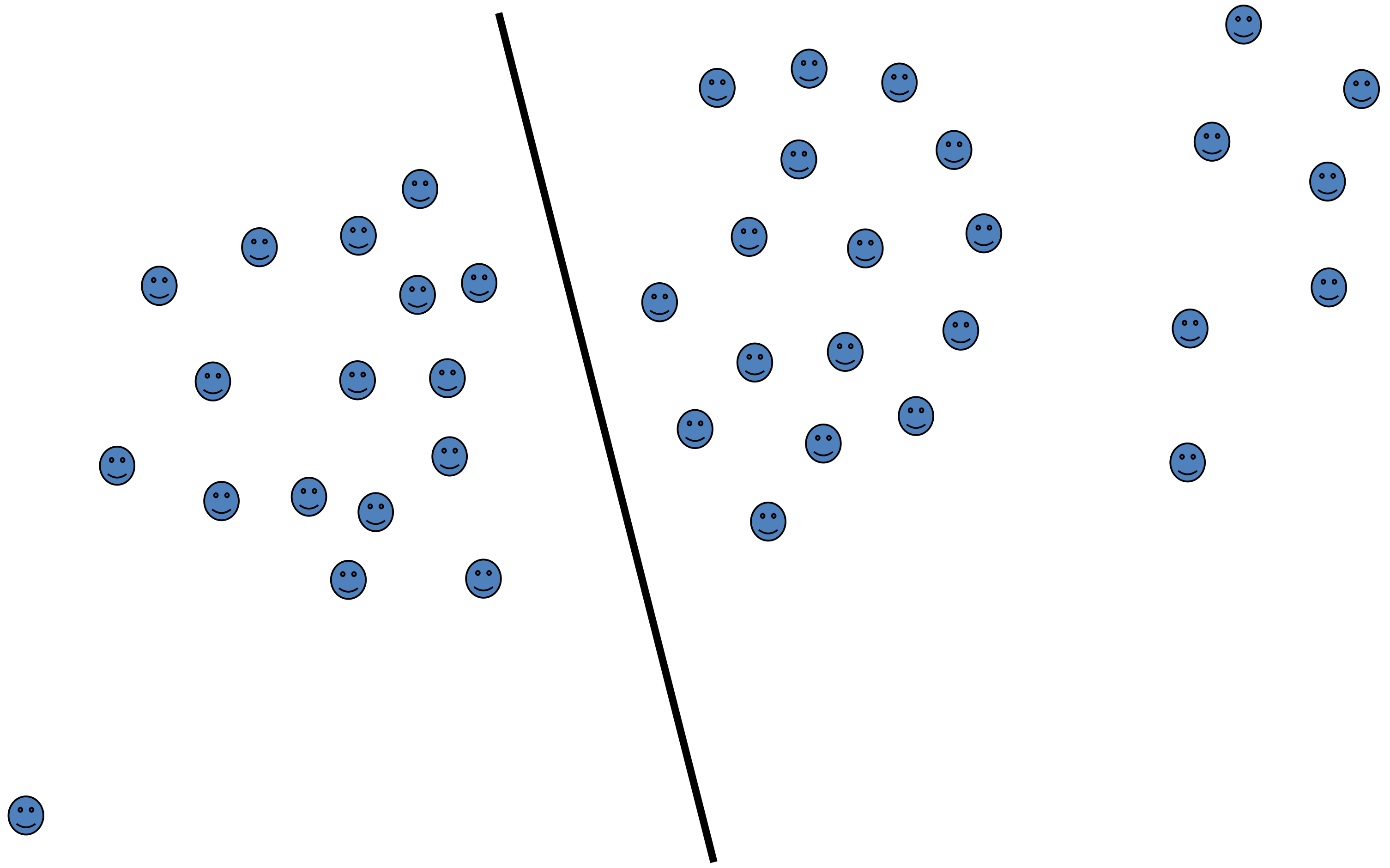
Classification vs. Clustering

- Classification assumes labeled data: we know how many classes there are and we have examples for each class (labeled data).
- **Classification is supervised**
- In Clustering we don’t have labeled data; we just assume that there is a natural division in the data and we may not know how many divisions (clusters) there are
- **Clustering is unsupervised**

Classification



Clustering



Binary vs. multi-way classification

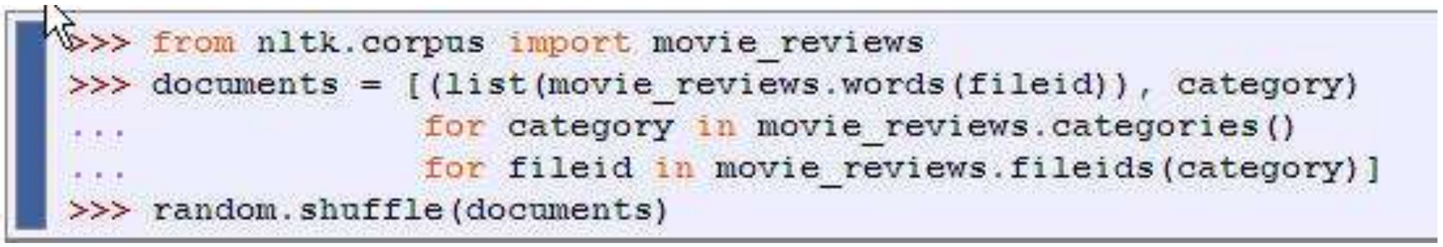
- Binary classification: two classes
- Multi-way classification: more than two classes
- Sometime it can be convenient to treat a multi-way problem like a binary one: one class versus all the others, for all classes

Flat vs. Hierarchical classification

- Flat classification: relations between the classes undetermined

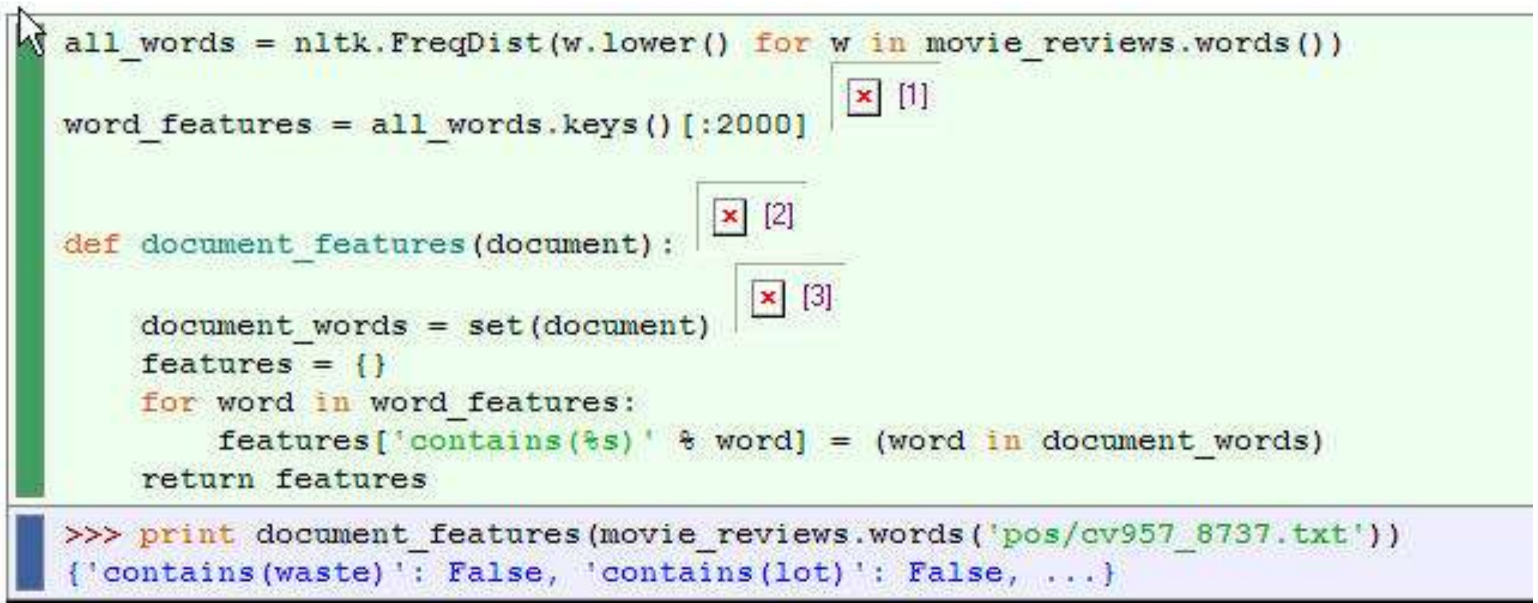
- Hierarchical classification: hierarchy where each node is the sub-class of its parent's node

Document classification NLTK example



```
>>> from nltk.corpus import movie_reviews
>>> documents = [(list(movie_reviews.words(fileid)), category)
...              for category in movie_reviews.categories()
...              for fileid in movie_reviews.fileids(category)]
>>> random.shuffle(documents)
```

- Define a feature extractor: a feature for each word, indicating whether the document contains that word.

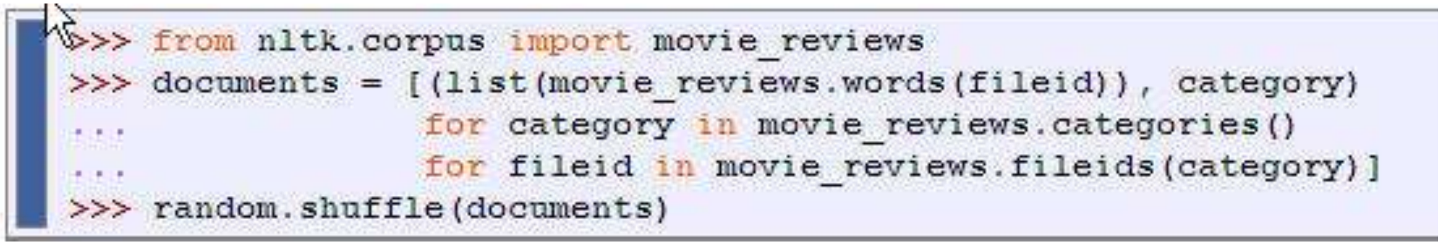


```
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
                                               [1]
word_features = all_words.keys()[:2000]

                                          [2]
def document_features(document):
                                             [3]
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains(%s)' % word] = (word in document_words)
    return features
```

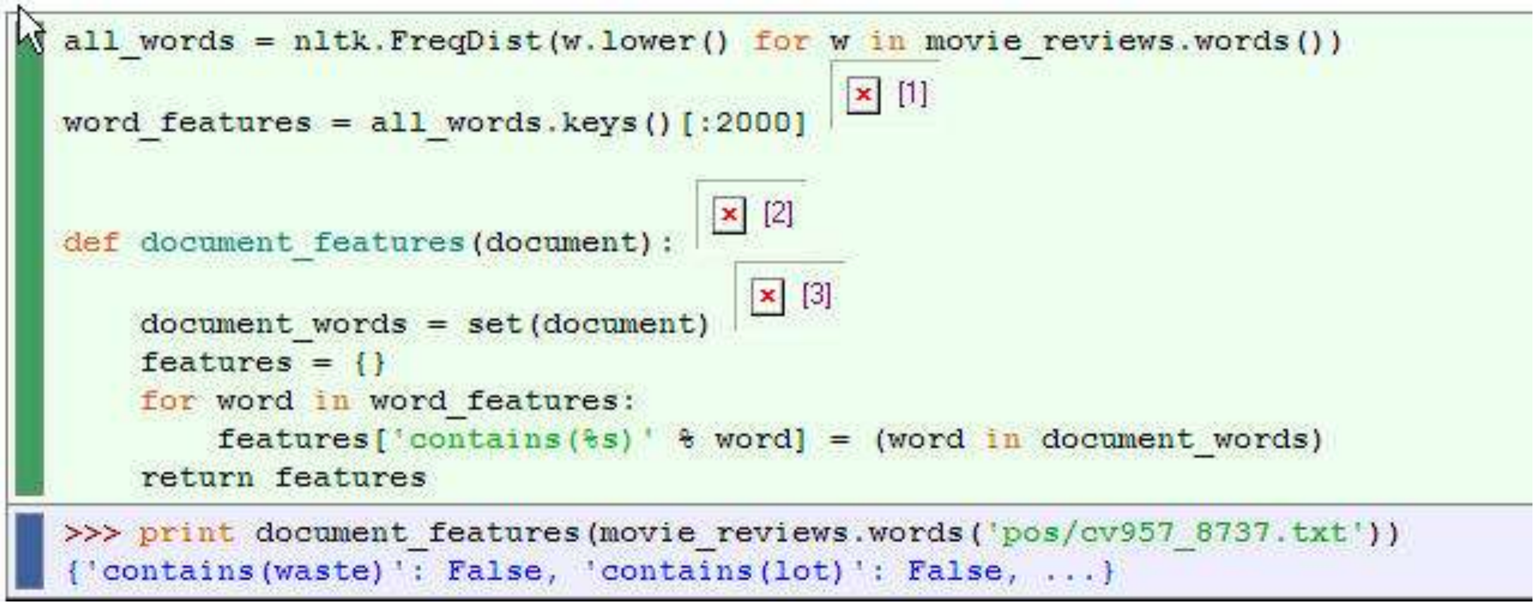
```
>>> print document_features(movie_reviews.words('pos/cv957_8737.txt'))
{'contains(waste)': False, 'contains(lot)': False, ...}
```

Document classification NLTK example

- Define a feature extractor: a feature for each word, indicating whether the document contains that word.



```
>>> from nltk.corpus import movie_reviews
>>> documents = [(list(movie_reviews.words(fileid)), category)
...              for category in movie_reviews.categories()
...              for fileid in movie_reviews.fileids(category)]
>>> random.shuffle(documents)
```

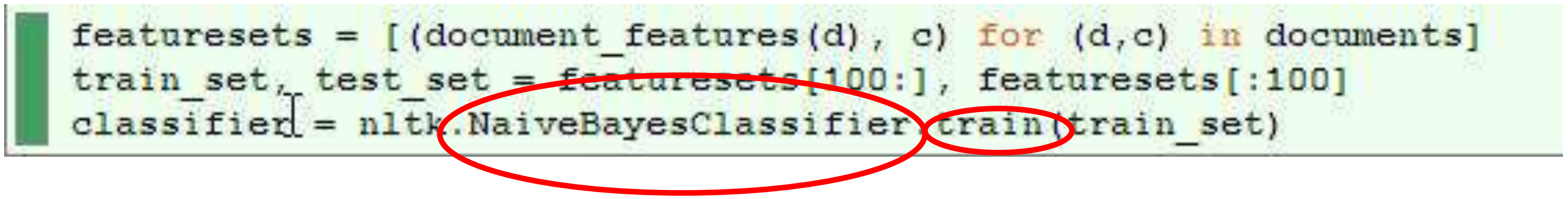


```
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())

word_features = all_words.keys()[:2000]     [1]


def document_features(document):     [2]

    document_words = set(document)     [3]
    features = {}
    for word in word_features:
        features['contains(%s)' % word] = (word in document_words)
    return features
```

```
>>> print document_features(movie_reviews.words('pos/cv957_8737.txt'))
{'contains(waste)': False, 'contains(lot)': False, ...}
```

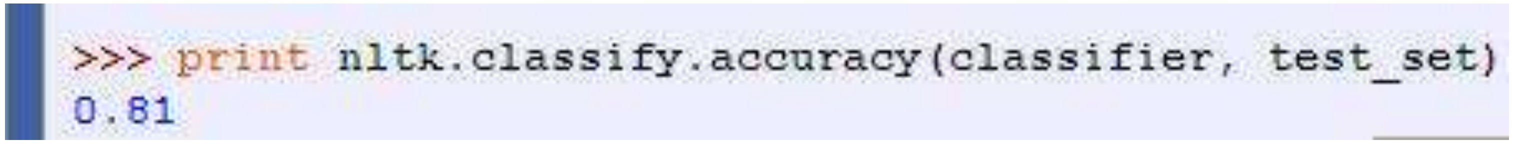
Document classification NLTK example

- Now that we've defined our feature extractor, we can use it to train a classifier.



```
featuresets = [(document_features(d), c) for (d,c) in documents]
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

- To check how reliable the resulting classifier is, we compute its accuracy on the test set



```
>>> print nltk.classify.accuracy(classifier, test_set)
0.81
```

Ranked retrieval

- Thus far, our queries have all been Boolean.
  - Documents either match or don’t.
- Good for expert users with precise understanding of their needs and the collection.
  - Also good for applications: Applications can easily consume 1000s of results.
- Not good for the majority of users.
  - Most users incapable of writing Boolean queries (or they are, but they think it’s too much work).
  - Most users don’t want to wade through 1000s of results.
    - This is particularly true of web search.

Problem with Boolean search: feast or famine

- Boolean queries often result in either too few (=0) or too many (1000s) results.
- Query 1: “*standard user dlink 650*” → 200,000 hits
- Query 2: “*standard user dlink 650 no card found*”: 0 hits
- It takes a lot of skill to come up with a query that produces a manageable number of hits.
  - AND gives too few; OR gives too many

Ranked retrieval models

- Rather than a set of documents satisfying a query expression, in ranked retrieval, the system returns an ordering over the (top) documents in the collection for a query
- Free text queries: Rather than a query language of operators and expressions, the user’s query is just one or more words in a human language
- In principle, there are two separate choices here, but in practice, ranked retrieval has normally been associated with free text queries and vice versa

Scoring as the basis of ranked retrieval

- We wish to return in order the documents most likely to be useful to the searcher
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in [0, 1] – to each document
- This score measures how well document and query “match”.

Query-document matching scores

- We need a way of assigning a score to a query/document pair
- Let’s start with a one-term query
- If the query term does not occur in the document: score should be 0
- The more frequent the query term in the document, the higher the score (should be)
- We will look at a number of alternatives for this.

Take 1: Jaccard coefficient

- Recall from Lecture 3: A commonly used measure of overlap of two sets *A* and *B*
- jaccard*(A,B)* = |*A* ∩ *B*| / |*A* ∪ *B*|
- jaccard*(A,A)* = 1
- jaccard*(A,B)* = 0 if *A* ∩ *B* = 0
- *A* and *B* don't have to be the same size.
- Always assigns a number between 0 and 1.

Issues with Jaccard for scoring

- It doesn't consider *term frequency* (how many times a term occurs in a document)
- Rare terms in a collection are more informative than frequent terms. Jaccard doesn't consider this information
- We need a more sophisticated way of normalizing for length
- Later in this lecture, we'll use
- . . . instead of |A ∩ B|/|A ∪ B| (Jaccard) for length normalization.

$$|A \cap B| / \sqrt{|A \cup B|}$$

Bag of words model

- Vector representation doesn't consider the ordering of words in a document
- *John is quicker than Mary* and *Mary is quicker than John* have the same vectors
- This is called the <u>bag of words</u> model.
- In a sense, this is a step back: The positional index was able to distinguish these two documents.
- We will look at "recovering" positional information later in this course.
- For now: bag of words model

Term frequency tf

- The term frequency $\mathrm{tf}_{t,d}$ of term *t* in document *d* is defined as the number of times that *t* occurs in *d*.
- We want to use tf when computing query-document match scores. But how?
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
  - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

Log-frequency weighting

- The log frequency weight of term t in d is
- 0 → 0, 1 → 1, 2 → 1.3, 10 → 2, 1000 → 4, etc.
- Score for a document-query pair: sum over terms *t* in both *q* and *d*:
- score

$$w_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{tf}_{t,d}, & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- The score is 0 if none of the query terms is present in the document.

$$= \sum\nolimits_{t \in q \cap d} (1 + \log \mathrm{tf}_{t,d})$$

Document frequency

- Rare terms are more informative than frequent terms
  - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric*
- → We want a high weight for rare terms like *arachnocentric*.

Document frequency, continued

- Frequent terms are less informative than rare terms
- Consider a query term that is frequent in the collection (e.g., *high, increase, line*)
- A document containing such a term is more likely to be relevant than a document that doesn't
- But it's not a sure indicator of relevance.
- → For frequent terms, we want high positive weights for words like *high, increase, and line*
- But lower weights than for rare terms.
- We will use document frequency (df) to capture this.

idf weight

- df$_t$ is the document frequency of t: the number of documents that contain t
  - df$_t$ is an inverse measure of the informativeness of t
  - df$_t$ $\leq N$
- We define the idf (inverse document frequency) of t by
  - We use log (N/df$_t$) instead of N/df$_t$ to "dampen" the effect of idf.

$$\mathrm{idf}_t = \log_{10}\,(N/\mathrm{df}_t)$$

Will turn out the base of the log is immaterial.

tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$\mathrm{w}_{t,d} = \log(1+\mathrm{tf}_{t,d}) \times \log_{10}(N/\mathrm{df}_t)$$

- Best known weighting scheme in information retrieval
  - Note: the "-" in tf-idf is a hyphen, not a minus sign!
  - Alternative names: tf.idf, tf x idf
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

Score for a document given a query

$$\text{Score}(q,d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

- There are many variants
  - How “tf” is computed (with/without logs)
  - Whether the terms in the query are also weighted
  - …

Binary → count → weight matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$

Documents as vectors

- So we have a |V|-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional: tens of millions of dimensions when you apply this to a web search engine
- These are very sparse vectors - most entries are zero.

Queries as vectors

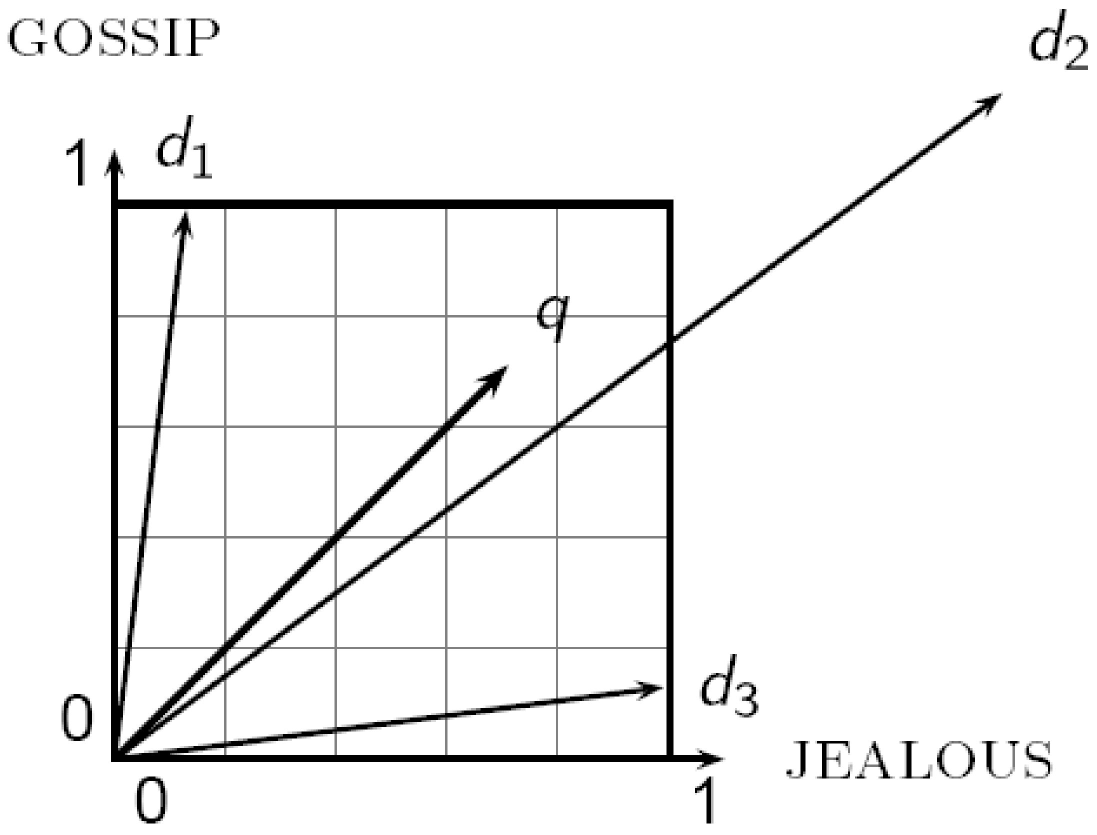
- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity ≈ inverse of distance
- Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model.
- Instead: rank more relevant documents higher than less relevant documents

Formalizing vector space proximity

- First cut: distance between two points
  - (= distance between the end points of the two vectors)
- Euclidean distance?
- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is large for vectors of different lengths.

Why distance is a bad idea

The Euclidean distance between $\vec{q}$ and $\vec{d_2}$ is large even though the distribution of terms in the query $\vec{q}$ and the distribution of terms in the document $\vec{d_2}$ are very similar.



Use angle instead of distance

- Thought experiment: take a document *d* and append it to itself. Call this document *d*′.
- “Semantically” d and d’ have the same content
- The Euclidean distance between the two documents can be quite large
- The angle between the two documents is 0, corresponding to maximal similarity.

- Key idea: Rank documents according to angle with query.