REVA UNIVERSITY
Bengaluru, India
Established as per the section 2(f) of the UGC Act, 1956,
Approved by AICTE, New Delhi

QS I·GAUGE GOLD

nirf

# Natural Language Processing

**Code: B22EA0602**

- **Course Objectives:**
- The objectives of this course are to:
- 1. Explain the fundamentals of natural language processing and python
- 2. Discuss how to access the text corpora and Lexical Resources
- 3. Demonstrate the writing the structured programs to process the raw text
- 4. Describe role of Classifiers in Text processing.
-
- **Course Outcomes:**
- On successful completion of this course, the student shall be able to:

- CO1: Summarize the fundamentals of natural language processing and python
- CO2: Learn how to access the text corpora and Lexical Resources
- CO3. Acquiring the skills for writing the structured programs to process the raw text
- CO4. Analyze the role of different classifiers in Text processing

**Course Contents:**

UNIT-1

**Language Processing and python, Accessing Text corpora and Lexical Analysis: Computing with language- Texts and words,** a closer look at python: texts as list of words, **computing with language**: simple statistics, Automatic natural language understanding; Accessing Text Corpora, Conditional Frequency Distributions, Lexical Resources, WordNet, Introduction to NLTK Tool. **Processing Raw Text:** Accessing Text from the Web and from Disk, Strings: Text Processing at the Lowest Level, Text Processing with Unicode, Regular Expressions for Detecting Word Patterns, Useful Applications of Regular Expressions, Normalizing Text, Regular Expressions for Tokenizing Text, Segmentation.

UNIT-2

**Categorizing and Tagging words:** Using a Tagger, using a Tagger, Mapping Words to Properties Using Python Dictionaries, Automatic Tagging, N-Gram Tagging, Transformation-Based Tagging, How to Determine the Category of a Word. **Classifying Text:** Supervised Classification: Examples, Evaluation; Decision Trees, Naive Bayes Classifiers.

UNIT-3

Text Summarization: Text Summarization and Information Extraction, Important concepts, Keyphrase Extraction, Topic Modelling, Automated Document Summarization. Text Similarity and Clustering: Information Retrieval, Feature Engineering, Text Similarity, Unsupervised machine learning algorithms, Analyzing Term Similarity, Analyzing Document Similarity, Document Clustering.

UNIT-4

Semantic and Sentiment Analysis: Semantic Analysis, Exploring WordNet, Word Sense Disambiguation, Named Entity Recognition, Analyzing Semantic Representation, Sentiment Analysis, Sentiment analysis of IMDb Movie Reviews.

**Self-Learning Component**:

Extracting information from Text, Exploring the 20 Newsgroups with Text Analysis Algorithms, Stock Price prediction with Regression Algorithms, Best Practices: i) Data preparation stage ii)Training sets generation stage iii) Model training, evaluation and selection stage.

## *Recommended Learning Resources: (Text Books)*

**1.Steven Bird, Ewan Klein and Edward Loper, —Natural Language Processing with Python, First Edition, OReilly Media, 2009.**

2.Yuxi (Hayden) Liu, - Python Machine Learning by Example, First edition, Packt publisher, 2017.

## Contents

- Introduction
- History
- Basics of python for NLP
- Texts as Lists of Words
- Computing with Language: Simple Statistics

# What is NLP?

- **Natural language processing** (**NLP**) is a subfield of linguistics, computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human (natural) languages, in particular how to program computers to process and analyse large amounts of natural language data.

- Computer assisted text analysis of human language
- Goal : To enable machines to understand human language and extract meaning from it.
- Field of study: Machine Learning / computational linguistics

# History

- 1948 :
  - 1st NLP application
  - Dictionary look-up system (developed at Brikbeck college, London) - designed to search and retrieve definitions, meanings, translations, synonyms, and other linguistic information from a dictionary database.
- 1949:
  - American interest
  - WWII (World War II (1939–1945)) code breaker Warren Weaver
  - He viewed German as English in code (Warren Weaver, an American mathematician and WWII codebreaker, proposed in 1949 that language translation could be approached like code-breaking. He suggested that translating German to English was similar to decrypting an encoded message. This idea laid the foundation for early machine translation research.)
- 1966 : over promised – under delivered
  - In the early days of artificial intelligence, researchers dreamed of teaching machines to understand human language. They promised wonders—instant translations, intelligent conversations, and seamless communication. But when the first attempts at machine translation emerged, they converted words without grasping meaning, leading to strange and useless results. Disappointed, funding agencies lost faith, cutting research budgets and leaving NLP struggling for survival. Long before AI had its name, NLP had already tarnished its reputation, casting a shadow over the field for years to come.

- 1980 – Return of NLP
  - A revolution in NLP( Due to steady increase of computational power, and the shift to Machine Learning algorithms)
  - Throughout the 1980s, IBM was responsible for the development of several successful, complicated statistical models.
- 1990 – statistical models for NLP (Uses probability-based models for language understanding.
  - N-Grams
  - LSTM recurrent neural network models
- After 2000 –
  - 2001- a feed forward neural network model
  - 2011- Apple's **siri**
  - Machine learning techniques (In 2001, **feed-forward neural networks** were used for NLP, marking a shift toward deep learning. By 2011, **Apple's Siri** introduced AI-driven voice assistants, leveraging **machine learning techniques** for speech recognition and natural language understanding).

# Language Processing and Python



- What we learn?
  - What can we achieve by combining simple programming techniques with large quantities of text?
  - How can we automatically extract key words and phrases that sum up the style and content of a text?
  - What tools and techniques does the Python programming language provide for such work?
  - What are some of the interesting challenges of natural language processing?

# Computing with Language: Texts and Words



- Text – raw data

- NLTK
- Install Anaconda python 3
- Download nltk
  - >>> import nltk
  - >>> nltk.download() - **download necessary NLTK resources** from the internet .

  Ex: nltk.download('wordnet') - WordNet database

  nltk.download('omw-1.4') - Open Multilingual WordNet

  nltk.download('averaged_perceptron_tagger') - POS tagging model, used to determine a word's part of speech for better lemmatization.

  - >>> from nltk.book import * - imports all the text samples from the NLTK (Natural Language Toolkit) book module, which includes several classic texts for NLP analysis.

# Computing with Language: Texts and Words



- Lists - sequence of words and punctuation:
- At one level, it is a sequence of symbols on a page,
- At another level, it is a sequence of chapters, made up of a sequence of sections, where each section is a sequence of paragraphs, and so on.
- However, for our purposes, we will think of a text as nothing more than a sequence of words and punctuation.
  - >>> sent1 = ['Call', 'me', 'Ishmael', '.' ’Vishwa’]
- >>> sent1
  - ['Call', 'me', 'Ishmael', ‘.’ ,’Vishwa’]
- >>> len(sent1)
  - 5
- >>> lexical_diversity(sent1)
  - 1.0
  - Lexical Diversity=Number of Unique Words (Types)/ Total Number of Words (Tokens)

- >>> sent2
  - ['The', 'family', 'of', 'Dashwood', 'had', 'long','been', 'settled', 'in', 'Sussex', '.']
- >>> sent3
  - ['In', 'the', 'beginning', 'God', 'created', 'the','heaven', 'and', 'the', 'earth', '.']
- >>> sent4
  - ['Fellow', '-', 'Citizens', 'of', 'the', 'Senate', 'and', 'of', 'the','House', 'of', 'Representatives', ':']

## Adding two lists (‘ + ‘)

- >>> sent4 + sent1
  - ['Fellow', '-', 'Citizens', 'of', 'the', 'Senate', 'and', 'of', 'the’, 'House', 'of', 'Representatives', ':', 'Call', 'me', 'Ishmael', '.’]

- **Appending - append()**
- >>> sent1.append("Some")
- >>> sent1
  - ['Call', 'me', 'Ishmael', '.', 'Some']

## Indexing Lists

- Count –
  - total number of words
  - Number of occurrences
- Examples: The number that represents this position is the item's **index**.
- >>> text4[173]
  - 'awaken‘
- We can do the converse; given a word, find the index of when it first occurs:
- >>> text4.index('awaken')
  - 173

## slicing

- 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']
- >>> **text6[1600:1625]**
  - ['We', """, 're', 'an', 'anarcho', '-', 'syndicalist', 'commune', '.', 'We','take', 'it', 'in', 'turns', 'to', 'act', 'as', 'a', 'sort', 'of', 'executive','officer', 'for', 'the', 'week']

# Variables

- *variable = expression*
- >>> sent1 = ['Call', 'me', 'Ishmael', '.']
- >>> my_sent = ['Bravely', 'bold', 'Sir', 'Robin', ',', 'rode',
  - ... 'forth', 'from', 'Camelot', '.']
- >>> noun_phrase = my_sent[1:4]
- >>> noun_phrase
  - ['bold', 'Sir', 'Robin']
- >>> words = sorted(noun_phrase)
- >>> words
  - ['Robin', 'Sir', 'bold']

# Strings

- **Assign a string to a variable**
  - >>> name = 'Monty‘
- **Index a string**
  - >>> name[0]
    - 'M'
- **Slice a string**
  - >>> name[:4]
    - 'Mont‘

- **Multiplication and addition with strings:**
- >>> name * 2
  - 'MontyMonty'
- >>> name + '!‘
  - 'Monty!‘
- **join the words of a list to make a single string, or split a string into a list**
- >>> ' '.join(['Monty', 'Python'])
  - 'Monty Python'
- >>> 'Monty Python'.split()
  - ['Monty', 'Python']

# Computing with Language: Simple Statistics

- In this section, we pick up the question of what makes a text distinct, and use automatic methods to find characteristic words and expressions of a text.
- what output do you expect here?
- >>> saying = ['After', 'all', 'is', 'said', 'and', 'done', 'more', 'is', 'said', 'than', 'done']
- >>> tokens = set(saying) - Removes duplicates, Stores elements in an unordered way.
- Ex: print(tokens)

**Output:** {'After', 'all', 'is', 'said', 'and', 'done', 'more', 'than'}

- >>> tokens = sorted(tokens)
- >>> tokens[-2:]
- Ex: tokens = ["I", "love", "natural", "language", "processing"]

  print(tokens[-2:])

**Output:** ['language', 'processing']

# Frequency Distributions

- Imagine how you might go about finding the 50 most frequent words of a book?
  - keep a tally for each vocabulary item – needs thousands of rows (its laborious process)
  - Frequency distribution- frequency of each vocabulary item in the ‘text’
  - Words like **"the" (most frequent)** appear **more times**, showing their **high importance** in the corpus.
  - Less frequent words (e.g., **"persevere"**) may have **lower significance**.
  - **"the"** has **21 marks** (5 + 5 + 5 + 1) **"been"** has **11 marks** (5 + 5 + 1)

Word Tally

| | |
|---|---|
| the | 𝍸 𝍸 𝍸 \|\|\|\| |
| been | 𝍸 𝍸 \| |
| message | \|\|\|\| |
| persevere | \| |
| nation | 𝍸 \|\|\| |

- **Frequency distributions** is needed very often in language processing, **NLTK** provides built-in support for them – '**FreqDist**'
- from nltk import FreqDist
- >>> fdist1 = FreqDist(text1)
- >>> fdist1
  - <FreqDist with 260819 outcomes>
  - Means that the frequency distribution is analyzing 260,819 unique tokens from a dataset.
- The expression **keys()** gives us a list of all the distinct types in the text and the first 50 of these by slicing the list.
- >>> vocabulary1 = fdist1.keys()
- >>> vocabulary1[:50]
  - [',', 'the', '.', 'of', 'and', 'a', 'to', ';', 'in', 'that', '"', '-','his', 'it', 'I', 's', 'is', 'he', 'with', 'was', 'as', '"', 'all', 'for', 'this', '!', 'at', 'by', 'but', 'not', '--', 'him', 'from', 'be', 'on', 'so', 'whale', 'one', 'you', 'had', 'have', 'there', 'But', 'or', 'were', 'now', 'which', '?', 'me', 'like']

- >>> fdist1['whale']
  - 906 (It occurs over 900 times)

    - If the frequent words don’t help us, how about the words that occur once only, the so called **hapaxes**? View them by typing **fdist1.hapaxes().**

    This list contains *lexicographer*, *cetological*, *contraband*, *expostulations*, and about 9,000 others.

    - Lexicographer → A person who compiles, writes, or edits dictionaries.
    - Cetological → Related to cetology, the study of whales, dolphins, and porpoises.
    - Expostulations refers to strong objections, protests, or expressions of disagreement.

    It seems that there are too many rare words, and without seeing the context we probably can’t guess what half of the hapaxes mean in any case! Since neither frequent nor infrequent words help, we need to try something else.

**Fine-Grained Selection of Words :** It means filtering words based on specific linguistic properties such as word frequency, length, part of speech (POS), stopwords, or semantic.

- *long* words of a text – more characteristic and informative
- To find the words from the vocabulary of the text that are more than 15 characters long.
- Let's call this property *P*, so that *P*(*w*) is true if and only if *w* is more than 15 characters long.

**a. *[w for w in V if p(w)]*** (it produces a list, not a set, which means that duplicates are possible.)

- lists allow repeated values.
- Maintains order of elements in V.
- V = set(text1)
- long_words = [w for w in V if len(w) > 15]
- print(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']
  - Sorted(long_words)

For each word w in the vocabulary V, we check whether len(w) is greater than 15; all other words will be ignored.

**b. {*w* | *w* ∈ *V* & *P*(*w*)}** (the set of all *w* such that *w* is an element of *V* (the vocabulary) and *w* has property *P*.)

- Eliminates duplicates (since sets cannot have duplicates).
- Order is not guaranteed (sets are unordered).

Ex: {w | w∈V and |w|>5}

- V = {"hello", "world", "beautiful", "great", "Python", "AI", "fantastic"}  # Vocabulary
- filtered_words = {w for w in V if len(w) > 5}
- print(filtered_words)

**Output :** {'beautiful', 'fantastic', 'Python'}

- All words from the chat corpus that are longer than seven characters, that occur more than seven times:
- >>> 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']

- Used two conditions:
  - len(w) > 7 - words are longer than seven letters
  - fdist5[w] > 7 - words occur more than seven times

# Collocations 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;
- Example - ***maroon wine*** sounds very odd.

  Fast food (allowed) vs Quick food (not allowed)

  Heavy rain (allowed) vs  Strong rain (not allowed)

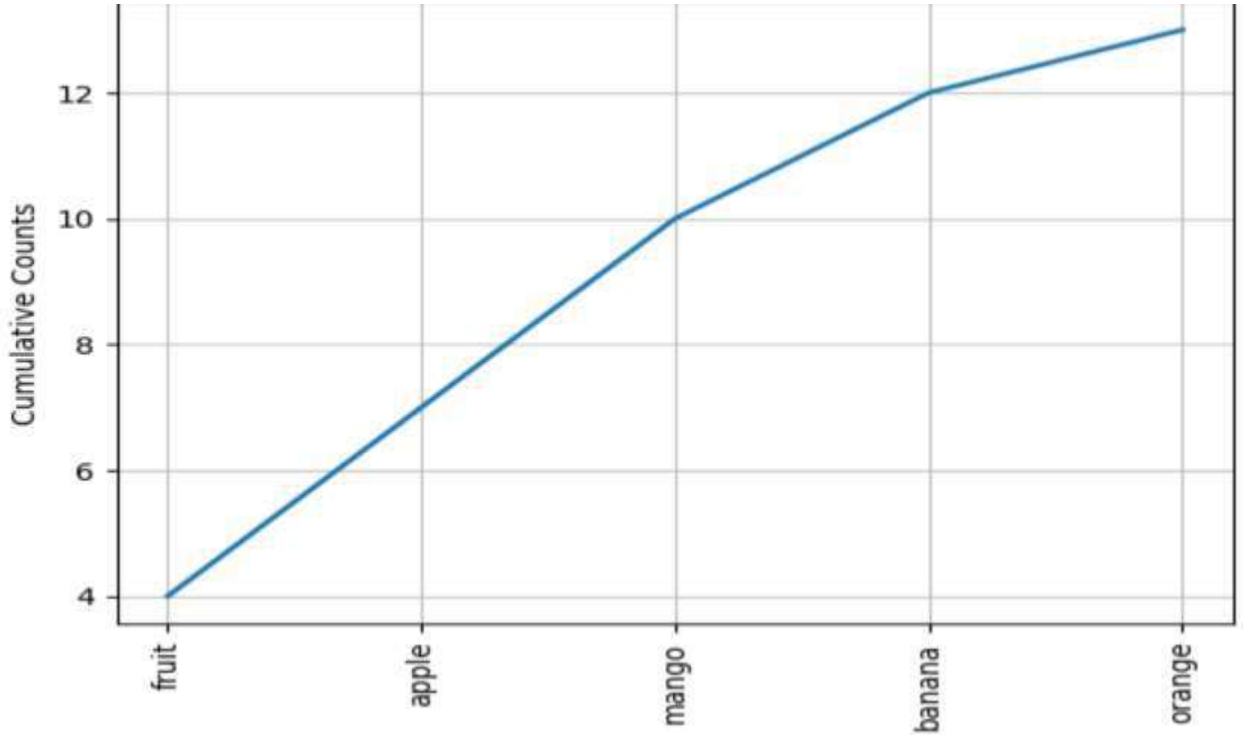
- Extracting from a text a list of word pairs, also known as **bigrams - bigrams()**
- >>> bigrams(['more', 'is', 'said', 'than', 'done'])
  - [('more', 'is'), ('is', 'said'), ('said', 'than'), ('than', 'done')]

- Collocations are essentially just frequent bigrams.
- We want to find bigrams that occur more often than we would expect based on the frequency of individual words.
- >>> 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; National Government; United Nations; public money
- >>> text8.collocations()
- Building collocations list
  - medium build; social drinker; quiet nights; long term; age open; financially secure; fun times; similar interests; Age open; poss rship; single mum; permanent relationship; slim build; seeks lady; Late 30s; Photo pls; Vibrant personality; European background; ASIANLADY; country drives

# *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` |

```
import nltk
nltk.download('punkt_tab')
from nltk.probability import FreqDist
from nltk.tokenize import word_tokenize
import matplotlib.pyplot as plt
# Sample text
text = "apple banana apple orange banana apple mango mango mango fruit fruit fruit fruit"
tokens = word_tokenize(text.lower())  # Tokenizing words
# Create frequency distribution
fdist = FreqDist(tokens)
# Plot cumulative frequency distribution
fdist.plot(10, cumulative=True)  # Top 10 words
# Show plot
plt.show()text = "apple banana apple orange banana apple mango mango mango fruit fruit fruit fruit"
```




The X-axis represents the most frequent words.

The Y-axis represents the cumulative count.

The curve increases as it sums up word occurrences.

**2. fdistt.tabulate() –**

Ex: text = "apple banana apple orange banana **apple mango mango mango“**

- **apple mango banana orange**

  3 3 2 1

# Automatic Natural Language Understanding

- Search engines have been crucial to the growth and popularity of the Web, but have some shortcomings.
- It takes skill, knowledge, and some luck, to extract answers to such questions as:
- *What tourist sites can I visit between Philadelphia and Pittsburgh on a limited budget? What do experts say about digital SLR cameras?*
- *What predictions about the steel market were made by credible commentators in the past week?*
- Getting a computer to answer them automatically involves a range of language processing tasks, including **information extraction, inference, and summarization**, and would need to be carried out on a scale and with a level of robustness that is still beyond our current capabilities.
- A long-standing challenge within artificial intelligence has been **to build intelligent machines**, and a major part of intelligent behavior is **understanding language**.

**Some language understanding technologies with interesting challenges**

**i.** Word Sense Disambiguation

**ii.** Pronoun Resolution

**iii.** Generating Language Output

**iv.** Machine Translation

**v.** Spoken Dialogue Systems

**vi.** Textual Entailment

**i. Word Sense Disambiguation**: It is the process of determining the correct meaning (sense) of a word based on its context. Many words are lexically ambiguous, meaning they have multiple meanings depending on how they are used in a sentence.

- which sense of a word was intended in a given context
- Consider the ambiguous words *serve* and *dish*:
  - a. *serve*: help with food or drink(ex: She served dinner to the guests); hold an office (ex: He served as mayor for five years); put ball into play (ex: She served the ball powerfully)
  - b. *dish*: plate(ex: The soup was in a deep dish); course of a meal (ex: Pasta is my favorite dish); communications device (ex: They installed a satellite dish for better reception)s
- consider the word *by*, which has several meanings, for example, *the book by Chesterton* (agentive (refers to a linguistic and grammatical concept describing a person or thing that performs an action.)—Chesterton was the author of the book); *the cup by the stove* (locative—the stove is where the cup is); and *submit by Friday* (temporal—Friday is the time of submitting).
  - a. The lost children were found by the ***searchers*** (agentive)
  - b. The lost children were found by the ***mountain*** (locative)
  - c. The lost children were found by the ***afternoon*** (temporal)

**ii. Pronoun Resolution:** It is also known as coreference resolution, is the task of determining which noun a pronoun (he, she, it, they, etc.) refers to in a sentence or document.

- A deeper kind of language understanding is to work out "who did what to whom," i.e., to detect the subjects and objects of verbs.
- Consider three possible following sentences in and try to determine what was **sold, caught, and found** (one case is ambiguous).
  - a. The thieves stole the paintings. **They** were subsequently ***sold***.
  - b. The thieves stole the paintings. **They** were subsequently ***caught***.
  - c. The thieves stole the paintings. **They** were subsequently ***found***.
- Answering this question involves finding the **antecedent** (is the word, phrase, that a pronoun refers to in a sentence) of the pronoun ***they*****,** either thieves or paintings. Computational techniques for tackling this problem include **anaphora resolution(refers back to a previously mentioned word )**—identifying what a pronoun or noun phrase refers to—and **semantic role labeling**—identifying how a noun phrase relates to the verb (as agent, patient, instrument, and so on).

# iii. Generating Language Output

If we can automatically solve such problems of language understanding, we will be able to move on to the tasks that involve generating language output, such as **question answering** and **machine translation.**

- **Question Answering :** A machine should be able to answer a user's questions relating to collection of texts.
  - a. *Text:* ... The thieves stole the paintings. They were subsequently sold. ...
  - b. *Human:* Who or what was sold?
  - c. *Machine:* The paintings.
- The machine's answer demonstrates that it has correctly worked out that *they* refers to paintings and not to thieves
- **Machine translation :** The machine should be able to translate the text into another language, accurately conveying the meaning of the original text.
- In translating the example text into French, we are forced to choose the gender of the pronoun in the second sentence: *ils* (masculine) if the thieves are sold, and *elles* (feminine) if the paintings are sold.

- Correct translation actually depends on correct understanding of the pronoun.
  - a. The thieves stole the paintings. They were subsequently found.
  - b. Les voleurs ont volé les peintures. Ils ont été trouvés plus tard. (the thieves)
  - c. Les voleurs ont volé les peintures. Elles ont été trouvées plus tard. (the paintings)

- To clarify whether "they" refers to thieves or paintings in French, you can explicitly mention the noun instead of using a pronoun:

**iv Machine translation continuation**

The program submits a sentence for translation into a specified language, then submits the resulting sentence for translation back into English. It stops after 12 iterations, or if it receives a translation that was produced already.

- NLTK Babelizer, a tool used for machine translation.
- babelize_shell() - starts the Babelizer interactive shell in NLTK.
- NLTK Babelizer: type 'help' for a list of commands.

  Babel> how long before the next flight to Alice Springs?

  Babel> german

  Babel> run

  - 0> how long before the next flight to Alice Springs?
  - 1> wie lang vor dem folgenden Flug zu Alice Springs?
  - 2> how long before the following flight to Alice jump?
  - 3> wie lang vor dem folgenden Flug zu Alice springen Sie?

- 4> how long before the following flight to Alice do you jump?
- 5> wie lang, bevor der folgende Flug zu Alice tun, Sie springen?
- 6> how long, before the following flight to Alice does, do you jump?
- 7> wie lang bevor der folgende Flug zu Alice tut, tun Sie springen?
- 8> how long before the following flight to Alice does, do you jump?
- 9> wie lang, bevor der folgende Flug zu Alice tut, tun Sie springen?
- 10> how long, before the following flight does to Alice, do do you jump?
- 11> wie lang bevor der folgende Flug zu Alice tut, Sie tun Sprung?
- 12> how long before the following flight does leap to Alice, does you?

- Observe that the system correctly translates *Alice Springs* from English to German (in the line starting 1>), but on the way back to English, this ends up as *Alice jump* (line 2). The preposition *before* is initially translated into the corresponding German preposition *vor*, but later into the conjunction *bevor* (line 5). After line 5 the sentences become non-sensical (but notice the various phrasings indicated by the commas, and the change from *jump* to *leap*). The translation system did not recognize when a word was part of a proper name, and it misinterpreted the grammatical structure.

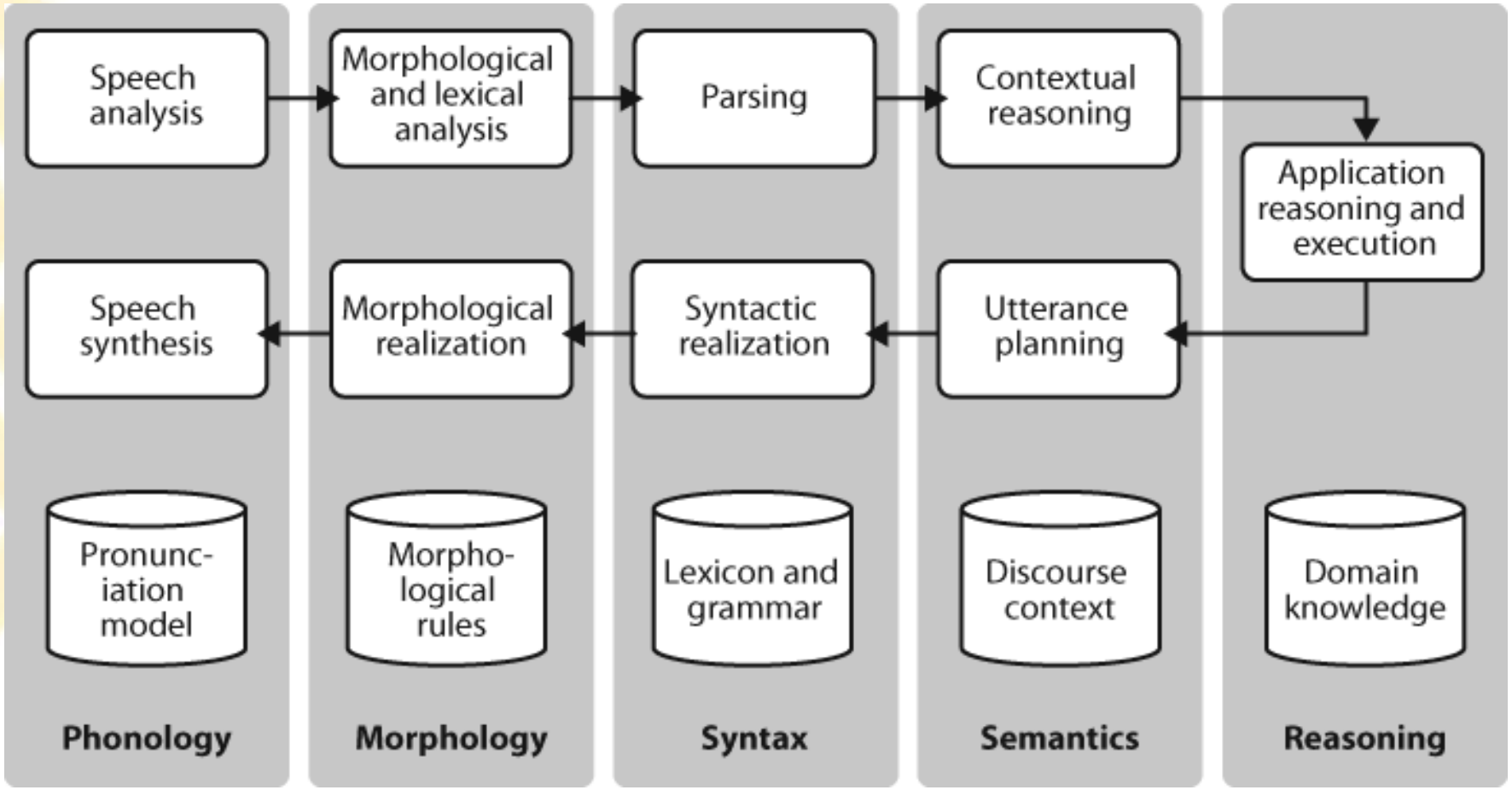
- >>> babelize_shell()
- Babel> The pig that John found looked happy
- Babel> german
- Babel> run
  - 0> The pig that John found looked happy
  - 1> Das Schwein, das John fand, schaute gl?cklich
  - 2> The pig, which found John, looked happy
- Machine translation is difficult because a given word could have several possible translations (depending on its meaning), and because word order must be changed in keeping with the grammatical structure of the target language.

**Probable Solution**: Given a document in German and English, and possibly a bilingual dictionary, we can automatically pair up the sentences, a process called **text alignment**. Once we have a million or more sentence pairs, we can detect corresponding words and phrases, and build a model that can be used for translating new text.

# v. Spoken Dialogue Systems

- The chief measure of intelligence has been a linguistic one, namely the **Turing Test**: can a dialogue system, responding to a user's text input, perform so naturally that we cannot distinguish it from a human-generated response? In contrast, today's commercial dialogue systems are very limited, but still perform useful functions in narrowly defined domains, as we see here:
  - S: How may I help you?
  - U: When is **Saving Private Ryan** playing?
  - S: For what theater?
  - U: The Paramount theater.
  - S: Saving Private Ryan is not playing at the Paramount theater, but it's playing at the Madison theater at 3:00, 5:30, 8:00, and 10:30.
- You could not ask this system to provide driving instructions or details of nearby restaurants unless the required information had already been stored and suitable question answer pairs had been incorporated into the language processing system.

# *pipeline architecture for a spoken dialogue system*



# Vi. Textual Entailment (Interpretation)

- Recognizing Textual Entailment (RTE) - finding evidence to support the hypothesis.
- Suppose you want to find evidence to support the hypothesis:

  a. Text: David Golinkin is the editor or author of 18 books, and over 150 responsa, articles, sermons and books

  b. Hypothesis: Golinkin has written 18 books

- In order to determine whether the hypothesis is supported by the text, the system needs the following background knowledge:

  (i) if someone is an author of a book, then he/ she has written that book;

  (ii) if someone is an editor of a book, then he/she has not written (all of) that book

  (iii) if someone is editor or author of 18 books, then one cannot conclude that he/she is author of 18 books.

# Chapter -2 of Unit -1
# Accessing Text Corpora and Lexical Resources

Practical work in NLP typically uses large bodies of linguistic data, or corpora. The goal of this chapter is to answer the following questions:

- What are some useful text corpora and lexical resources, and how can we access them with Python?
- Which Python constructs are most helpful for this work?
- How do we avoid repeating ourselves when writing Python code?

This chapter continues to present programming concepts by example, in the context of a linguistic processing task. We will wait until later before exploring each Python construct systematically.

# Accessing Text Corpora

- A text corpus is a large body of text.
- This section examines a variety of text corpora. It shows how to select individual texts, and how to work with them.

**Gutenberg Corpus:**

- NLTK includes a small selection of texts from the Project Gutenberg electronic text archive, which contains some 25,000 free electronic books.
- Start getting the Python interpreter to load the NLTK package, then ask to see nltk.corpus.gutenberg.fileids(), the file identifiers in this corpus:
- import nltk
- nltk.download('gutenberg')
- 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']

- Let’s pick out the first of these texts—*Emma* by Jane Austen—and give it a short name, emma, then find out how many words it contains:
- >>> emma = nltk.corpus.gutenberg.words ('austen-emma.txt')
- >>> len(emma)
- 192427
- When we defined emma, we invoked the words() function of the gutenberg object in NLTK’s corpus package. But since it is cumbersome to type such long names all the time, Python provides another version of the import statement, as follows:
- >>> from nltk.corpus import gutenberg
- >>> gutenberg.fileids()

['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', ...]

- >>> emma = gutenberg.words('austen-emma.txt')

- Let's write a short program to display other information about each text, by looping over all the values of fileid corresponding to the gutenberg file identifiers listed earlier and then computing statistics for each text.
- For a compact output display, we will make sure that the numbers are all integers, using int().

```
for fileid in gutenberg.fileids():
    num_chars = len(gutenberg.raw(fileid))                # Total number of characters
    num_words = len(gutenberg.words(fileid))              # Total number of words
    num_sents = len(gutenberg.sents(fileid))              # Total number of sentences
    num_vocab = len(set([w.lower() for w in gutenberg.words(fileid)])) # Unique words (vocabulary size)
    print (int(num_chars/num_words), int(num_words/num_sents),
    int(num_words/num_vocab), fileid)
```

- num_chars: Total **characters** in the text.
- num_words: Total **words** in the text.
- num_sents: Total **sentences** in the text.
- num_vocab: Number of **unique words** (vocabulary size, case insensitive).

- num_chars / num_words: Average **word length** (characters per word).
- num_words / num_sents: Average **sentence length** (words per sentence).
- num_words / num_vocab: **Lexical diversity** (words per unique word).

This means, for example, in **austen-emma.txt**:

- Average **word length** = **4** characters
- Average **sentence length** = **24** words
- **Lexical diversity** = 26 (i.e., each unique word is used about 26 times)

- 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

- This program displays three statistics for each text:

- Average word length,

- Average sentence length,

- The number of times each vocabulary item appears in the text on average (our lexical diversity score).

- Observe that average word length appears to be a general property of English, since it has a recurrent value of *4*.

- By contrast average sentence length and lexical diversity appear to be characteristics of particular authors.

- The previous example also showed how we can access the “raw” text of the book , not split up into tokens.

- The raw() function gives us the contents of the file without any linguistic processing. So, for example, len(gutenberg.raw('blake-poems.txt') tells us how many *letters* occur in the text, including the spaces between words.

- The sents() function divides the text up into its sentences, where each sentence is a list of words:
- >>> macbeth_sentences = gutenberg.sents('shakespeare-macbeth.txt')
- >>> macbeth_sentences

  [['[', 'The', 'Tragedie', 'of', 'Macbeth', 'by', 'William', 'Shakespeare’,

  '1603', ']'], ['Actus', 'Primus', '.'], ...]
- >>> macbeth_sentences[1037]

  ['Double', ',', 'double', ',', 'toile', 'and', 'trouble', ';', 'Fire', 'burne', ',', 'and', 'Cauldron’, 'bubble']
- >>> longest_len = max([len(s) for s in macbeth_sentences]) #Finds the maximum sentence length (i.e., the longest sentence in terms of word count).
- >>> [s for s in macbeth_sentences if len(s) == longest_len]

  [['Doubtfull', 'it', 'stood', ',', 'As', 'two', 'spent', 'Swimmers', ',', 'that',

  'doe', 'cling', 'together', ',', 'And', 'choake', 'their', 'Art', ':', 'The', 'mercilesse', 'Macdonwald', ...], ...]

## Web and Chat Text

- Although Project Gutenberg contains thousands of books, it represents established literature.
- It is important to consider less formal language as well. NLTK'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 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, originally collected by the Naval Postgraduate School for research on automatic detection of Internet predators.
- The corpus contains over 10,000 posts, anonymized by replacing usernames with generic names of the form “UserNNN”, and manually edited to remove any other identifying information.
- The corpus is organized into 15 files, where each file contains several hundred posts collected on a given date, for an age-specific chatroom (teens, 20s, 30s, 40s, plus a generic adults chatroom).
- The filename contains the date, chat- room, and number of posts; e.g., 10-19-20s_706posts.xml contains 706 posts gathered from the 20s chat room on 10/19/2006.
- >>> from nltk.corpus import nps_chat
- >>> chatroom = nps_chat.posts('10-19-20s_706posts.xml')
- >>> chatroom[123]
- ['i', 'do', "n't", 'want', 'pics', 'of', 'a', 'female', ',', 'I', 'can', 'look', 'in', 'a', 'mirror', '.']

# 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.
- Table 2-1 gives an example of each genre (for a complete list, see *http://icame.uib.no/brown/bcm-los.html*).

- *Table 2-1. Example document for each section of the 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 |

- We can access the corpus as a list of words or a list of sentences (where each sentence is itself just a list of words). We can optionally specify particular categories or files to read:
- from nltk.corpus import brown
- brown.categories()
- ['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance', 'science_fiction']
- brown.words(categories='news')
- ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
- brown.words(fileids=['cg22'])
- ['Does', 'our', 'society', 'have', 'a', 'runaway', ',', ...]
- brown.sents(categories=['news', 'editorial', 'reviews']) [['The', 'Fulton', 'County'...], ['The', 'jury', 'further'...], ...]

- The Brown Corpus is a convenient resource for studying systematic differences between genres, a kind of linguistic inquiry known as **stylistics**.

Let's compare genres in their usage of modal verbs.

- The first step is to produce the counts for a particular genre.
- Remember to import nltk before doing the following:
- >>> from nltk.corpus import brown
- >>> news_text = brown.words(categories='news')
- >>> fdist = nltk.FreqDist([w.lower() for w in news_text])
- >>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
- >>> for m in modals:
- ... print (m + ':', fdist[m])
- can: 94 could: 87 may: 93 might: 38 must: 53 will: 389

- Choose a different section of the Brown Corpus, and adapt the preceding example to count a selection of *wh* words, such as *what*, *when*, *where*, *who* and *why*.
- Next, we need to obtain counts for each genre of interest. We'll use NLTK's support for conditional frequency distributions.
- These are presented systematically as follows
- **>>> cfd = nltk.ConditionalFreqDist(**

  **(genre, word)**

  **for genre in brown.categories()**

  **for word in brown.words(categories=genre))**
- **>>> genres = ['news', 'religion', 'hobbies', 'science_fiction', 'romance', 'humor']**
- **>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']**
- **>>> cfd.tabulate(conditions=genres, samples=modals)**

| Genre | can | could | may | might | must | will |
|---|---|---|---|---|---|---|
| news | 94 | 87 | 79 | 38 | 53 | 389 |
| religion | 82 | 59 | 78 | 12 | 54 | 72 |
| hobbies | 268 | 58 | 22 | 5 | 83 | 264 |
| science_fiction | 16 | 49 | 4 | 12 | 8 | 16 |
| romance | 74 | 193 | 11 | 51 | 45 | 43 |
| humor | 16 | 30 | 8 | 8 | 9 | 13 |

- Observe that the most frequent modal in the news genre is *will,* while the most frequent modal in the romance genre is *could.*

## Reuters Corpus

- Provides a collection of financial and business news articles that are useful for natural language processing (NLP).
- 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”; thus, the text with fileid 'test/14826' is a document drawn from the test set.
- This split is for training and testing algorithms that automatically detect the topic of a document
- >>> from nltk.corpus import reuters
- >>> reuters.fileids()
- ['test/14826', 'test/14828', 'test/14829', 'test/14832', ..., 'training/9888', 'training/9889']
- >>> reuters.categories()
- ['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa’,

  'coconut', 'coconut-oil', 'coffee', 'copper', 'copra-cake', 'corn’,

  'cotton', 'cotton-oil', 'cpi', 'cpu', 'crude', 'dfl', 'dlr', ...]

- Unlike the Brown Corpus, categories in the Reuters Corpus overlap with each other, simply because a news story often covers multiple topics.
- We can ask for the topics covered by one or more documents, or for the documents included in one or more categories.
- For convenience, the corpus methods accept a single fileid or a list of fileids.
- >>> reuters.categories('training/9865')

  ['barley', 'corn', 'grain', 'wheat']
- >>> reuters.categories(['training/9865', 'training/9880'])

  ['barley', 'corn', 'grain', 'money-fx', 'wheat']
- >>> reuters.fileids('barley')

  ['test/15618', 'test/15649', 'test/15676', 'test/15728', 'test/15871', ...]
- >>> reuters.fileids(['barley', 'corn'])

  ['test/14832', 'test/14858', 'test/15033', 'test/15043', 'test/15106', 'test/15287', 'test/15341', 'test/15618', 'test/15618', 'test/15648', ...]

- Similarly, we can specify the words or sentences we want in terms of files or categories. The first handful of words in each of these texts are the titles, which by convention are stored as uppercase
- >>> reuters.words('training/9865')[:14]

  ['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', 'BIDS',

  'DETAILED', 'French', 'operators', 'have', 'requested', 'licences', 'to', 'export']
- >>> reuters.words(['training/9865', 'training/9880'])

  ['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
- >>> reuters.words(categories='barley')

  ['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
- >>> reuters.words(categories=['barley', 'corn'])

  ['THAI', 'TRADE', 'DEFICIT', 'WIDENS', 'IN', 'FIRST', ...]

# Inaugural Address Corpus

- Contains a collection of **presidential inaugural speeches** from 1789 to the present.
- The corpus is a collection of 55 texts, one for each presidential address. An interesting property of this collection is its time dimension:
- Each filename corresponds to a U.S. presidential inaugural address, formatted as:

  [Year]-[President].txt
- >>> from nltk.corpus import inaugural
- >>> inaugural.fileids()

  ['1789-Washington.txt', '1793-Washington.txt', '1797-Adams.txt', ...]
- >>> [fileid[:4] for fileid in inaugural.fileids()]

  ['1789', '1793', '1797', '1801', '1805', '1809', '1813', '1817', '1821', ...]
- To get the year out of the filename, we extracted the first four characters, using fileid[:4].

- Let's look at how the words *America* and *citizen* are used over time
- The following code converts the words in the Inaugural corpus to lowercase using w.lower(), then checks whether they start with either of the "targets" America or citizen using startswith().
- Thus it will count words such as *American's* and *Citizens*
- >>> cfd = nltk.ConditionalFreqDist(

  (target, fileid[:4])

  for fileid in inaugural.fileids()

  for w in inaugural.words(fileid)
- >>> cfd.plot()

| Feature | `FreqDist` | `ConditionalFreqDist` |
|---|---|---|
| **Definition** | A simple frequency distribution that counts occurrences of words (or other elements) in a single dataset. | A conditional frequency distribution that counts word occurrences **grouped by categories** (conditions). |
| **Use Case** | Counts word frequency in a **single** text. | Counts word frequency **across multiple categories** (e.g., genres, years). |
| **Data Structure** | A dictionary `{word: count}`. | A nested dictionary `{category: {word: count}}`. |
| **Key-Value Structure** | `{word: count}` | `{condition: {word: count}}` |
| **Example Corpus** | Analyzing frequency of words in one speech. | Comparing word usage across different speeches. |
| **Visualization** | `.plot()` for a single dataset. | `.plot()` for comparing trends across |

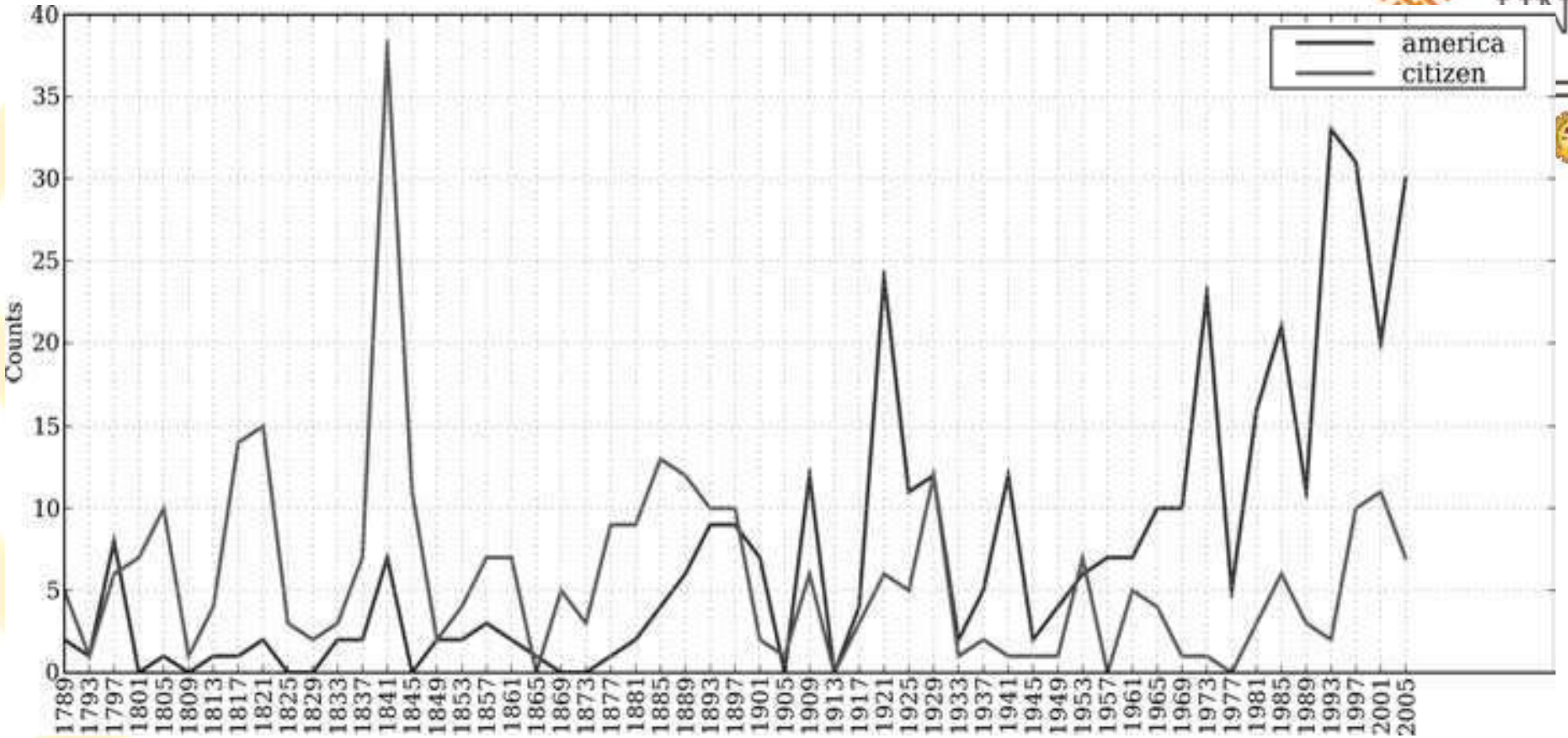



*Figure 2-1. Plot of a conditional frequency distribution: All words in the Inaugural Address Corpus that begin with* america *or* citizen *are counted; separate counts are kept for each address; these are plotted so that trends in usage over time can be observed; counts are not normalized for document length.*

# Annotated Text Corpora

- Many text corpora contain **linguistic annotations, representing part-of-speech tags, named entities, syntactic structures, semantic roles**, and so forth.
- NLTK provides convenient ways to access several of these corpora, and has data packages containing corpora and corpus samples, freely downloadable for use in teaching and research.
- Table 2-2 lists some of the corpora. For information about downloading them, see *http://www.nltk.org/data*.
- *Table 2-2. Some of the corpora and corpus samples distributed with NLTK*

| Corpus | Compiler | Contents |
|---|---|---|
| Brown Corpus | Francis, Kucera | 15 genres, 1.15M words, tagged, categorized |
| CESS Treebanks | CLiC-UB | 1M words, tagged and parsed (Catalan, Spanish) |
| Chat-80 Data Files | Pereira & Warren | World Geographic Database |
| CMU Pronouncing Dictionary | CMU | 127k entries |
| CoNLL 2000 Chunking Data | CoNLL | 270k words, tagged and chunked |

## Corpora in Other Languages

- NLTK comes with corpora for many languages, though in some cases you will need to learn how to manipulate character encodings in Python before using these corpora
- >>> nltk.corpus.cess_esp.words()

  ['El', 'grupo', 'estatal', 'Electricit\xe9_de_France', ...]
- >>> nltk.corpus.floresta.words()

  ['Um', 'revivalismo', 'refrescante', 'O', '7_e_Meio', ...]
- >>> nltk.corpus.indian.words('hindi.pos')

  ['\xe0\xa4\xaa\xe0\xa5\x82\xe0\xa4\xb0\xe0\xa5\x8d\xe0\xa4\xa3','\xe0\xa4\xaa\xe0\xa5\x8d\xe0\xa4\xb0\xe0\xa4\xa4\xe0\xa4\xbf\xe0\xa4\xac\xe0\xa4\x82\xe0\xa4\xa7', ...]
- >>> nltk.corpus.udhr.fileids()
- ['Abkhaz-Cyrillic+Abkh', 'Abkhaz-UTF8', 'Achehnese-Latin1', 'Achuar-Shiwiar-Latin1', 'Adja-UTF8', 'Afaan_Oromo_Oromiffa-Latin1', 'Afrikaans-Latin1', 'Aguaruna-Latin1', 'Akuapem_Twi-UTF8', 'Albanian_Shqip-Latin1', 'Amahuaca', 'Amahuaca-Latin1', ...]
- >>> nltk.corpus.udhr.words('Javanese-Latin1')[11:] [u'Saben', u'umat', u'manungsa', u'lair', u'kanthi', ...]

# Universal Declaration of Human Rights (UDHR)

- The last of these corpora, udhr, contains the **Universal Declaration of Human Rights** in over 300 languages.
- The fileids for this corpus include information about the character encoding used in the file, such as UTF8 or Latin1.
- Let's use a conditional frequency distribution to examine the differences in word lengths for a selection of languages included in the udhr corpus.
- The output is shown in Figure 2-2 (run the program yourself to see a color plot). Note that True and False are Python's built-in Boolean values.
- >>> from nltk.corpus import udhr
- >>> languages = ['Chickasaw', 'English', 'German_Deutsch', 'Greenlandic_Inuktikut', 'Hungarian_Magyar', 'Ibibio_Efik']
- >>> cfd = nltk.ConditionalFreqDist(

  (lang, len(word))

  for lang in languages

  for word in udhr.words(lang + '-Latin1'))
- >>> cfd.plot(cumulative=True)

# Text Corpus Structure

- We have seen a variety of corpus structures so far; these are summarized in Figure 2-3.
- The simplest kind lacks any structure: it is just a collection of texts.
- Often, texts are grouped into categories that might correspond to genre, source, author, language, etc.
- Sometimes these categories overlap, notably in the case of topical categories, as a text can be relevant to more than one topic.
- Occasionally, text collections have temporal structure, news collections being the most common example.
- NLTK's corpus readers support efficient access to a variety of corpora, and can be used to work with new corpora. Table 2-3 lists functionality provided by the corpus readers.

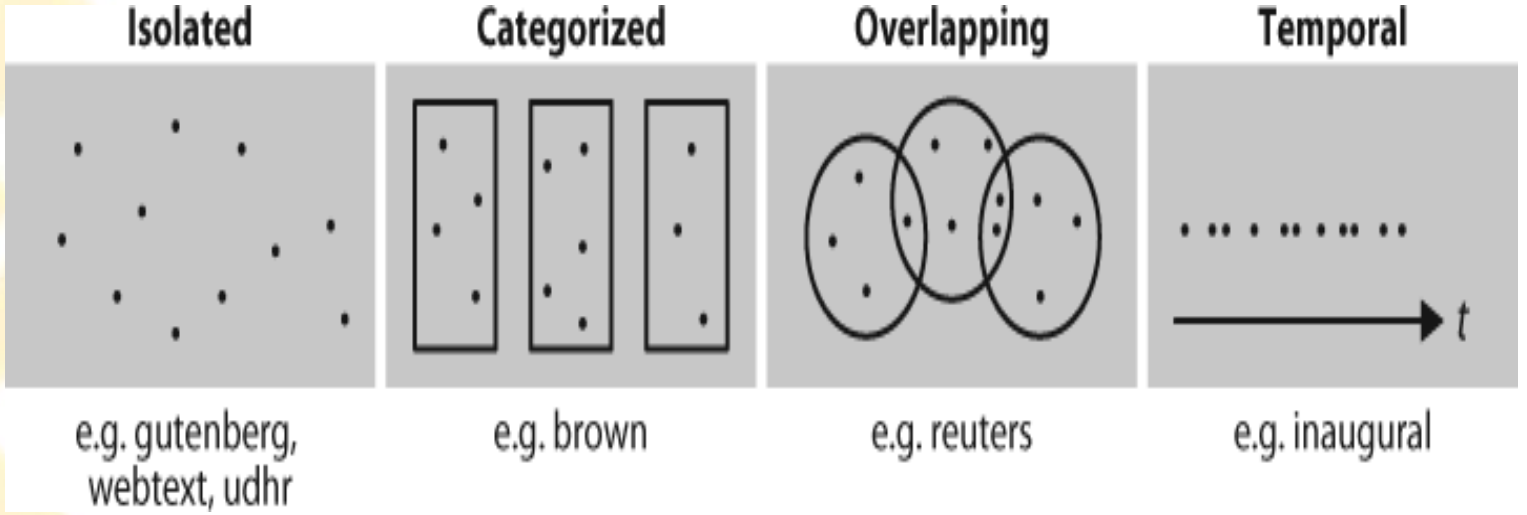



*Figure 2-3. Common structures for text corpora: The simplest kind of corpus is a collection of isolated texts with no particular organization; some corpora are structured into categories, such as genre (Brown Corpus); some categorizations overlap, such as topic categories (Reuters Corpus); other corpora represent language use over time (Inaugural Address Corpus).*

*Table 2-3. Basic corpus functionality defined in NLTK: More documentation can be found using help(nltk.corpus.reader) and by reading the online Corpus HOWTO at http://www.nltk.org/howto.*

| Example | Description |
|---|---|
| fileids() | The files of the corpus |
| fileids([categories]) | The files of the corpus corresponding to these |
| categories categories() | The categories of the corpus |
| categories([fileids]) | The categories of the corpus corresponding to these files |
| raw() | The raw content of the corpus |
| raw(fileids=[f1,f2,f3]) | The raw content of the specified files |
| raw(categories=[c1,c2]) | The raw content of the specified categories |
| words() | The words of the whole corpus |
| words(fileids=[f1,f2,f3]) | The words of the specified fileids |
| words(categories=[c1,c2]) | The words of the specified categories |
| sents() | The sentences of the specified categories |
| sents(fileids=[f1,f2,f3]) | The sentences of the specified fileids |
| sents(categories=[c1,c2]) | The sentences of the specified categories |
| abspath(fileid) | The location of the given file on disk |
| encoding(fileid) | The encoding of the file (if known) |
| open(fileid) | Open a stream for reading the given corpus file |
| root() | The path to the root of locally installed corpus |
| readme() | The contents of the README file of the corpus |

- We illustrate the difference between some of the corpus access methods here:
- >>> raw = gutenberg.raw("burgess-busterbrown.txt")
- >>> raw[1:20]
- 'The Adventures of B'
- >>> words = gutenberg.words("burgess-busterbrown.txt")
- >>> words[1:20]

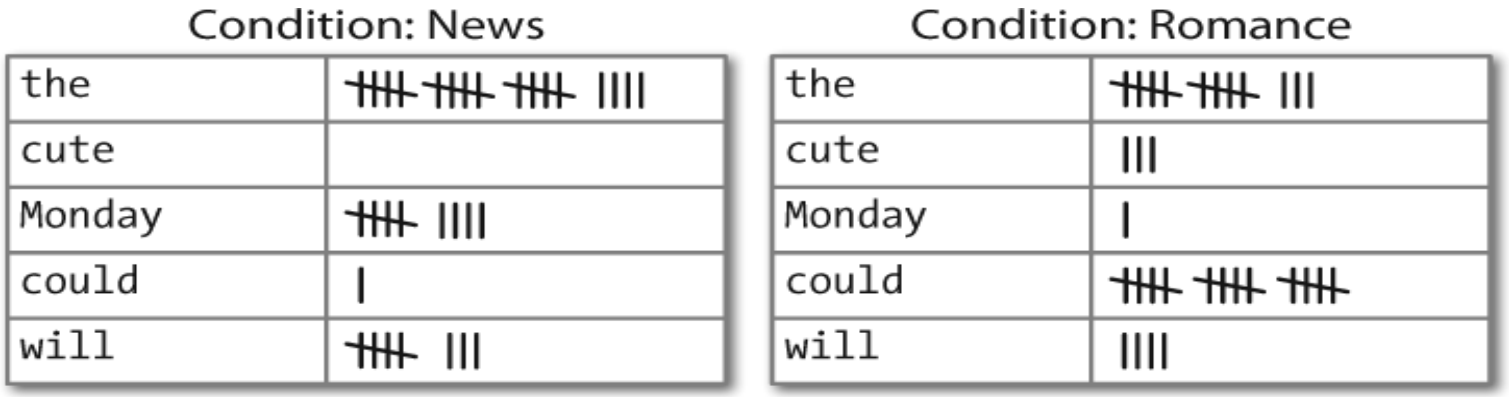
  ['The', 'Adventures', 'of', 'Buster', 'Bear', 'by', 'Thornton', 'W', '.',
  'Burgess', '1920', ']', 'I', 'BUSTER', 'BEAR', 'GOES', 'FISHING', 'Buster',
  'Bear']
- >>> sents = gutenberg.sents("burgess-busterbrown.txt")
- >>> sents[1:20]

  [['I'], ['BUSTER', 'BEAR', 'GOES', 'FISHING'], ['Buster', 'Bear', 'yawned', 'as',
  'he', 'lay', 'on', 'his', 'comfortable', 'bed', 'of', 'leaves', 'and', 'watched',
  'the', 'first', 'early', 'morning', 'sunbeams', 'creeping', 'through', ...], ...]

- Loading Your Own Corpus
- If you have your own collection of text files that you would like to access using the methods discussed earlier, you can easily load them with the help of NLTK's Plain textCorpusReader.
- Check the location of your files on your file system; in the following example, we have taken this to be the directory */usr/share/dict*.
- Whatever the location, set this to be the value of corpus_root .
- The second parameter of the PlaintextCor pusReader initializer can be a list of fileids, like ['a.txt', 'test/b.txt'], or a pattern that matches all fileids, like '[abc]/.*\.txt' (see Section 3.4 for information about regular expressions).
- >>> from nltk.corpus import PlaintextCorpusReader
- >>> corpus_root = '/usr/share/dict' >>> wordlists = PlaintextCorpusReader(corpus_root, '.*')
- >>> wordlists.fileids()
- ['README', 'connectives', 'propernames', 'web2', 'web2a', 'words']
- >>> wordlists.words('connectives')
- ['the', 'of', 'and', 'to', 'a', 'in', 'that', 'is', ...]

# • 2.2 Conditional Frequency Distributions

- We introduced frequency distributions in Section 1.3. We saw that given some list mylist of words or other items, FreqDist(mylist) would compute the number of occurrences of each item in the list. Here we will generalize this idea.
- When the texts of a corpus are divided into several categories (by genre, topic, author, etc.), we can maintain separate frequency distributions for each category.
- This will allow us to study systematic differences between the categories.
- In the previous section, we achieved this using NLTK's ConditionalFreqDist data type. A **conditional frequency distribution** is a collection of frequency distributions, each one for a different "condition."
- The condition will often be the category of the text. Figure 2-4 depicts a fragment of a conditional frequency distribution having just two conditions, one for news text and one for romance text



Condition: News

| | |
|---|---|
| the | 𝍸 𝍸 𝍸 \|\|\|\| |
| cute | |
| Monday | 𝍸 \|\|\|\| |
| could | \| |
| will | 𝍸 \|\|\| |

Condition: Romance

| | |
|---|---|
| the | 𝍸 𝍸 \|\|\| |
| cute | \|\|\| |
| Monday | \| |
| could | 𝍸 𝍸 𝍸 |
| will | \|\|\|\| |

*Figure 2-4. Counting words appearing in a text collection (a conditional frequency distribution).*

## Conditions and Events

A frequency distribution counts observable events, such as the appearance of words in a text. A conditional frequency distribution needs to pair each event with a condition. So instead of processing a sequence of words , we have to process a sequence of pairs :

```
>>> text = ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
>>> pairs = [('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ...]
```

Each pair has the form (*condition*, *event*). If we were processing the entire Brown Corpus by genre, there would be 15 conditions (one per genre) and 1,161,192 events (one per word).

- **Counting Words by Genre**
- In Section 2.1, we saw a conditional frequency distribution where the condition was the section of the Brown Corpus, and for each condition we counted words. Whereas FreqDist() takes a simple list as input, ConditionalFreqDist() takes a list of pairs.
- >>> from nltk.corpus import brown
- >>> cfd = nltk.ConditionalFreqDist(
- ... (genre, word)
- ... for genre in brown.categories()
- ... for word in brown.words(categories=genre))
- Let's break this down, and look at just two genres, news and romance. For each genre , we loop over every word in the genre , producing pairs consisting of the genre and the word
- >>> genre_word = [(genre, word)
- ... for genre in ['news', 'romance']
- ... for word in brown.words(categories=genre)]
- >>> len(genre_word)
- 170576

- So, as we can see in the following code, pairs at the beginning of the list genre_word will be of the form ('news', *word*) , whereas those at the end will be of the form ('romance', *word*) .



```
>>> genre_word[:4]
[('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ('news', 'Grand')]
```

- >>> genre_word[-4:]
- [('romance', 'afraid'), ('romance', 'not'), ('romance', """), ('romance', '.')]
- We can now use this list of pairs to create a ConditionalFreqDist, and save it in a variable cfd. As usual, we can type the name of the variable to inspect it and verify it has two conditions
- >>> cfd = nltk.ConditionalFreqDist(genre_word)
- >>> cfd
- <ConditionalFreqDist with 2 conditions>
- >>> cfd.conditions() ['news', 'romance']

- Let’s access the two conditions, and satisfy ourselves that each is just a frequency distribution:
- >>> cfd['news']
- <FreqDist with 100554 outcomes>
- >>> cfd['romance']
- <FreqDist with 70022 outcomes>
- >>> list(cfd['romance'])
- [',', '.', 'the', 'and', 'to', 'a', 'of', '``', """, 'was', 'I', 'in', 'he', 'had',
- '?', 'her', 'that', 'it', 'his', 'she', 'with', 'you', 'for', 'at', 'He', 'on', 'him',
- 'said', '!', '--', 'be', 'as', ';', 'have', 'but', 'not', 'would', 'She', 'The', ...]
- >>> cfd['romance']['could'] 193

- **Plotting and Tabulating Distributions**
- Apart from combining two or more frequency distributions, and being easy to initialize, a ConditionalFreqDist provides some useful methods for tabulation and plotting.
- The plot in Figure 2-1 was based on a conditional frequency distribution reproduced in the following code. The condition is either of the words *america* or *citizen*
- the counts being plotted are the number of times the word occurred in a particular speech.
- It exploits the fact that the filename for each speech—for example, *1865-Lincoln.txt*—contains the year as the first four characters
- The following code generates the pair ('america', '1865') for every instance of a word whose lowercased form starts with *america*—such as *Americans*—in the file *1865-Lincoln.txt*.

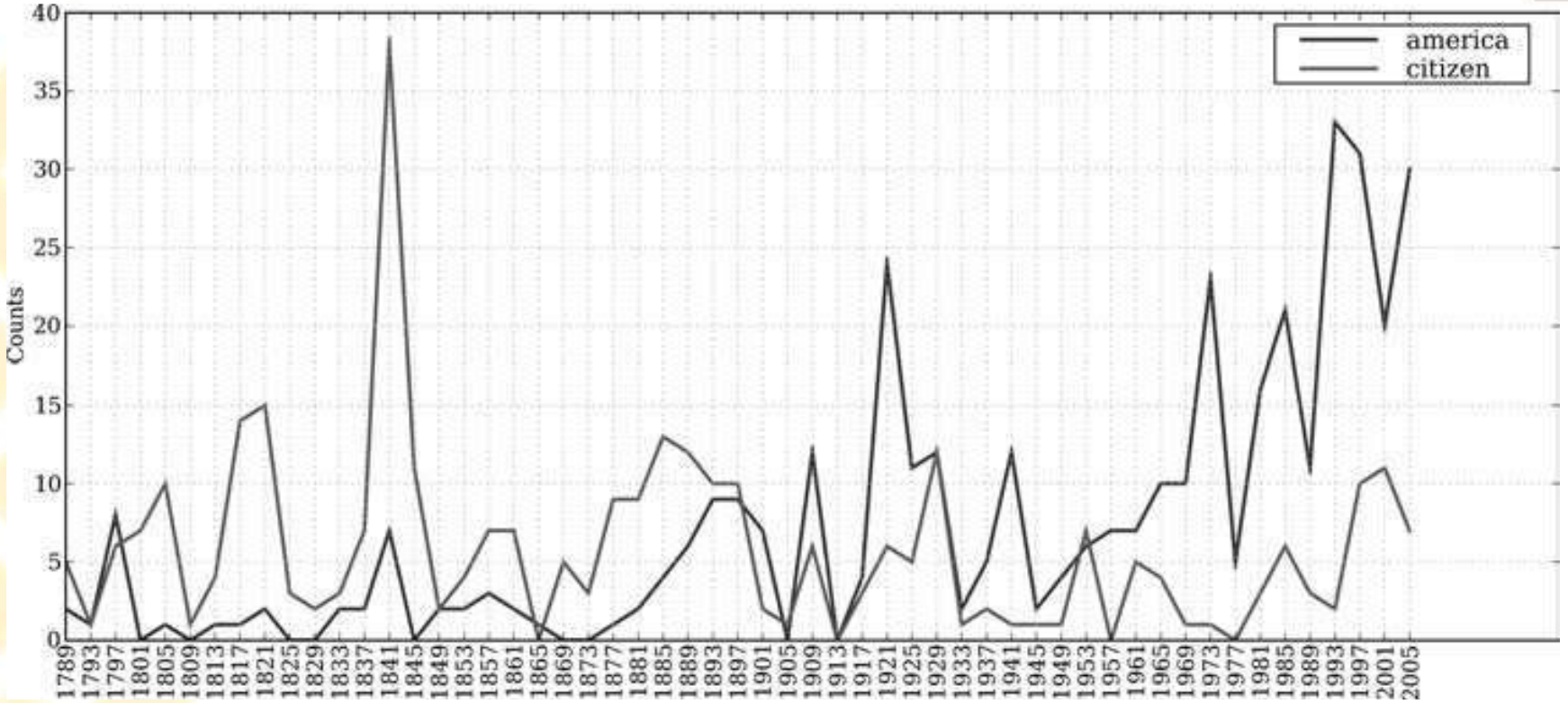



*Figure 2-1. Plot of a conditional frequency distribution: All words in the Inaugural Address Corpus that begin with* america *or* citizen *are counted; separate counts are kept for each address; these are plotted so that trends in usage over time can be observed; counts are not normalized for document length.*

- >>> from nltk.corpus import inaugural
- >>> cfd = nltk.ConditionalFreqDist(
- ... (target, fileid[:4])
- ... for fileid in inaugural.fileids()
- ... for w in inaugural.words(fileid)
- ... for target in ['america', 'citizen'] ...
- if w.lower().startswith(target))
- The plot in Figure 2-2 was also based on a conditional frequency distribution, reproduced in the following code. This time, the condition is the name of the language, and the counts being plotted are derived from word lengths It exploits the fact that the filename for each language is the language name followed by '-Latin1' (the character encoding).

- >>> from nltk.corpus import udhr
- >>> languages = ['Chickasaw', 'English', 'German_Deutsch',
- ... 'Greenlandic_Inuktikut', 'Hungarian_Magyar', 'Ibibio_Efik']
- >>> cfd = nltk.ConditionalFreqDist(
- ... (lang, len(word))
- ... for lang in languages
- ... for word in udhr.words(lang + '-Latin1'))

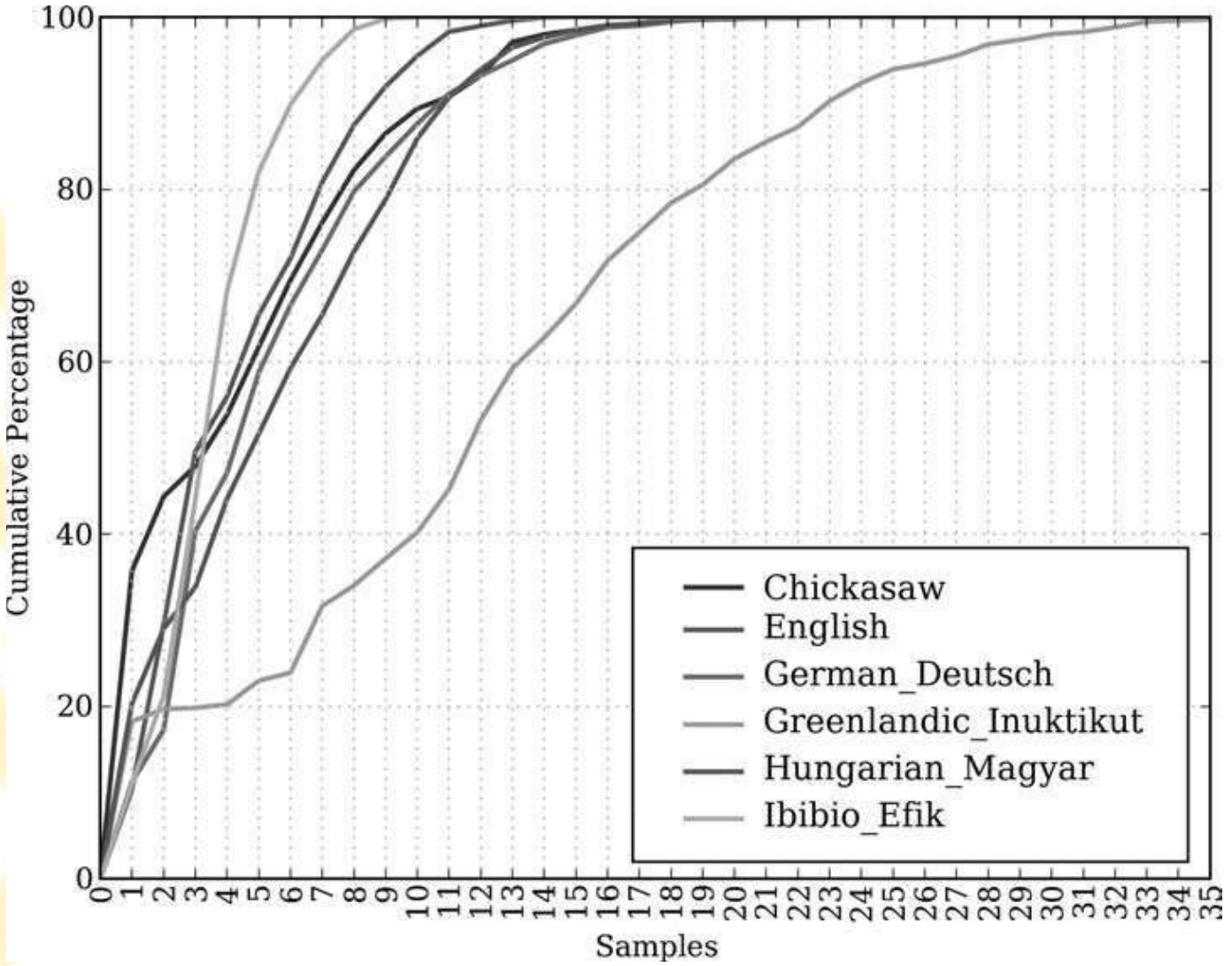



*Figure 2-2. Cumulative word length distributions: Six translations of the Universal Declaration of Human Rights are processed; this graph shows that words having five or fewer letters account for about 80% of Ibibio text, 60% of German text, and 25% of Inuktitut text.*

- **Generating Random Text with Bigrams**
- We can use a conditional frequency distribution to create a table of bigrams
- The bigrams() function takes a list of words and builds a list of consecutive word pairs:
- >>> sent = ['In', 'the', 'beginning', 'God', 'created', 'the', 'heaven',
- ... 'and', 'the', 'earth', '.']
- >>> nltk.bigrams(sent)
- [('In', 'the'), ('the', 'beginning'), ('beginning', 'God'), ('God', 'created'),
- ('created', 'the'), ('the', 'heaven'), ('heaven', 'and'), ('and', 'the'), ('the', 'earth'), ('earth', '.')]

- *Example 2-1. Generating random text: This program obtains all bigrams from the text of the book of Genesis, then constructs a conditional frequency distribution to record which words are most likely to follow a given word; e.g., after the word* living*, the most likely word is* creature*; the* generate_model() *function uses this data, and a seed word, to generate random text.*
- def generate_model(cfdist, word, num=15):
- for i in range(num):
- print word,
- word = cfdist[word].max()
- text = nltk.corpus.genesis.words('english-kjv.txt')
- bigrams = nltk.bigrams(text)
- cfd = nltk.ConditionalFreqDist(bigrams)
- >>> print cfd['living']
- <FreqDist: 'creature': 7, 'thing': 4, 'substance': 2, ',': 1, '.': 1, 'soul': 1>
- >>> generate_model(cfd, 'living')
- living creature that he said , and the land of the land of the land

- *Table 2-4. NLTK's conditional frequency distributions: Commonly used methods and idioms for defining, accessing, and visualizing a conditional frequency distribution of counters*

| Example | Description |
|---|---|
| cfdist = ConditionalFreqDist(pairs) | Create a conditional frequency distribution from a list of pairs |
| cfdist.conditions() | Alphabetically sorted list of conditions |
| cfdist[condition] | The frequency distribution for this condition |
| cfdist[condition][sample] | Frequency for the given sample for this condition |
| cfdist.tabulate() | Tabulate the conditional frequency distribution |
| cfdist.tabulate(samples, conditions) | Tabulation limited to the specified samples and conditions |
| cfdist.plot() | Graphical plot of the conditional frequency distribution |
| cfdist.plot(samples, conditions) | Graphical plot limited to the specified samples and conditions |
| cfdist1 < cfdist2 | Test if samples in cfdist1 occur less frequently than in cfdist2 |

# • 2.3 Lexical Resources:

- A lexicon, or lexical resource, is a collection of words and/or phrases along with associated information, such as part-of-speech and sense definitions.
- Lexical resources are secondary to texts, and are usually created and enriched with the help of texts.
- For example, if we have defined a text my_text, then
- vocab = sorted(set(my_text)) builds the vocabulary of my_text,
- whereas word_freq = FreqDist(my_text) counts the frequency of each word in the text. Both **vocab** and **word_freq** are simple lexical resources.
- A **lexical entry** consists of a **headword** (also known as a **lemma**) along with additional information, such as the part-of-speech and the sense definition. *Two distinct words having the same spelling are called* ***homonyms****.*

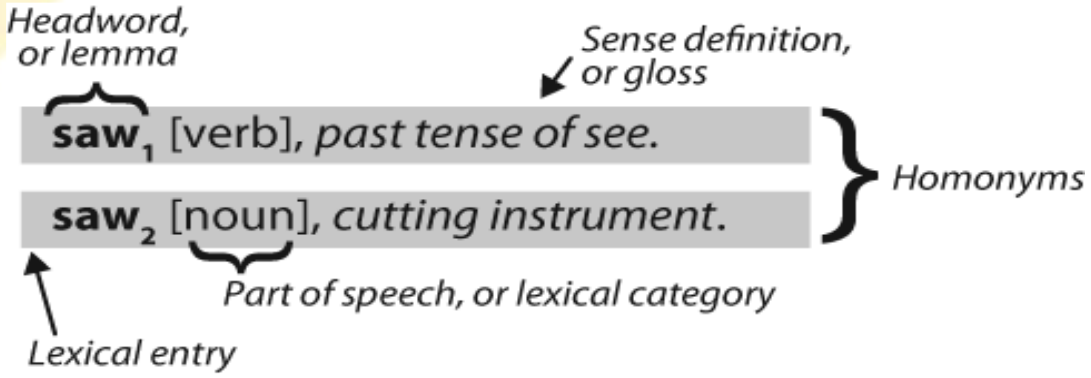



*Figure : Lexicon terminology: Lexical entries for two lemmas having the same spelling (homonyms), providing part-of-speech and gloss information.*

- **Wordlist Corpora**
- NLTK includes some corpora that are nothing more than wordlists. The Words Corpus is the */usr/dict/words* file from Unix, used by some spellcheckers. We can use it to find unusual or misspelled words in a text corpus, as shown in Example
- *Example - Filtering a text: This program computes the vocabulary of a text, then removes all items that occur in an existing wordlist, leaving just the uncommon or misspelled words.*
- def unusual_words(text):
- text_vocab = set(w.lower() for w in text if w.isalpha()) english_vocab = set(w.lower() for w in nltk.corpus.words.words()) unusual = text_vocab.difference(english_vocab)
- return sorted(unusual)
- >>> unusual_words(nltk.corpus.gutenberg.words('austen-sense.txt'))
- ['abbeyland', 'abhorrence', 'abominably', 'abridgement', 'accordant', 'accustomary', 'adieus', 'affability', 'affectedly', 'aggrandizement', 'alighted', 'allenham', 'amiably', 'annamaria', 'annuities', 'apologising', 'arbour', 'archness', ...]

- >>> unusual_words(nltk.corpus.nps_chat.words())
- ['aaaaaaaaaaaaaaaaa', 'aaahhhh', 'abou', 'abourted', 'abs', 'ack', 'acros',
- 'actualy', 'adduser', 'addy', 'adoted', 'adreniline', 'ae', 'afe', 'affari', 'afk',
- 'agaibn', 'agurlwithbigguns', 'ahah', 'ahahah', 'ahahh', 'ahahha', 'ahem', 'ahh', ...]

- There is also a corpus of **stopwords**, that is, high-frequency words such as *the*, *to*, and *also* that we sometimes want to filter out of a document before further processing.
- Stopwords usually have little lexical content, and their presence in a text fails to distinguish it from other texts.
- >>> from nltk.corpus import stopwords
- >>> stopwords.words('english')
- ['a', "a's", 'able', 'about', 'above', 'according', 'accordingly', 'across',
- 'actually', 'after', 'afterwards', 'again', 'against', "ain't", 'all', 'allow',
- 'allows', 'almost', 'alone', 'along', 'already', 'also', 'although', 'always', ...]

- Let’s define a function to compute what fraction of words in a text are *not* in the stop- words list:
- >>> def content_fraction(text):
- ...  stopwords = nltk.corpus.stopwords.words('english')
- ...  content = [w for w in text if w.lower() not in stopwords]
- ...  return len(content) / len(text)
- ...
- >>> content_fraction(nltk.corpus.reuters.words())
- 0.6599769539328526 1
- Thus, with the help of stopwords, we filter out a third of the words of the text. Notice that we’ve combined two different kinds of corpus here, using a lexical resource to filter the content of a text corpus.

- **A Pronouncing Dictionary**
- A slightly richer kind of lexical resource is a table (or spreadsheet), containing a word plus some properties in each row. NLTK includes the CMU Pronouncing Dictionary for U.S. English, which was designed for use by speech synthesizers.
- >>> entries = nltk.corpus.cmudict.entries()
- >>> len(entries) 127012
- >>> for entry in entries[39943:39951]:
- ... print entry
- ...
- ('fir', ['F', 'ER1'])
- ('fire', ['F', 'AY1', 'ER0'])
- ('fire', ['F', 'AY1', 'R'])
- ('firearm', ['F', 'AY1', 'ER0', 'AA2', 'R', 'M'])
- ('firearm', ['F', 'AY1', 'R', 'AA2', 'R', 'M'])
- ('firearms', ['F', 'AY1', 'ER0', 'AA2', 'R', 'M', 'Z'])
- ('firearms', ['F', 'AY1', 'R', 'AA2', 'R', 'M', 'Z'])
- ('fireball', ['F', 'AY1', 'ER0', 'B', 'AO2', 'L'])

- For each word, this lexicon provides a list of phonetic codes—distinct labels for each contrastive sound—known as *phones*. Observe that *fire* has two pronunciations (in U.S. English): the one-syllable F AY1 R, and the two-syllable F AY1 ER0.

- Each entry consists of two parts, and we can process these individually using a more complex version of the for statement. Instead of writing for entry in entries:, we replace entry with *two* variable names, word, pron Now, each time through the loop, word is assigned the first part of the entry, and pron is assigned the second part of the entry:

- >>> for word, pron in entries:
- ... if len(pron) == 3:
- ... ph1, ph2, ph3 = pron
- ... if ph1 == 'P' and ph3 == 'T':
- ... print word, ph2,
- ...
- pait EY1 pat AE1 pate EY1 patt AE1 peart ER1 peat IY1 peet IY1 peete IY1 pert ER1 pet EH1 pete IY1 pett EH1 piet IY1 piette IY1 pit IH1 pitt IH1 pot AA1 pote OW1 pott AA1 pout AW1 puett UW1 purt ER1 put UH1 putt AH1

- The program just shown scans the lexicon looking for entries whose pronunciation consists of three phones.
- If the condition is true, it assigns the contents of pron to three new variables: ph1, ph2, and ph3. Notice the unusual form of the statement that does that work.
- Here's another example of the same for statement, this time used inside a list comprehension. This program finds all words whose pronunciation ends with a syllable sounding like *nicks*. You could use this method to find rhyming words.
- >>> syllable = ['N', 'IH0', 'K', 'S']
- >>> [word for word, pron in entries if pron[-4:] == syllable]
- ["atlantic's", 'audiotronics', 'avionics', 'beatniks', 'calisthenics', 'centronics', 'chetniks', "clinic's", 'clinics', 'conics', 'cynics', 'diasonics', "dominic's",

  'ebonics', 'electronics', "electronics'", 'endotronics', "endotronics'", 'enix', ...]
- Notice that the one pronunciation is spelled in several ways: *nics*, *niks*, *nix*, and even *ntic's* with a silent *t*, for the word *atlantic's*.

- Let’s look for some other mismatches between pronunciation and writing. Can you summarize the purpose of the following examples and explain how they work?
- >>> [w for w, pron in entries if pron[-1] == 'M' and w[-1] == 'n'] ['autumn', 'column', 'condemn', 'damn', 'goddamn', 'hymn', 'solemn']
- >>> sorted(set(w[:2] for w, pron in entries if pron[0] == 'N' and w[0] != 'n')) ['gn', 'kn', 'mn', 'pn']
- The phones contain digits to represent primary stress (1), secondary stress (2), and no stress (0). As our final example, we define a function to extract the stress digits and then scan our lexicon to find words having a particular stress pattern.
- def stress(pron):
- ... return [char for phone in pron for char in phone if char.isdigit()]
- >>> [w for w, pron in entries if stress(pron) == ['0', '1', '0', '2', '0']] ['abbreviated', 'abbreviating', 'accelerated', 'accelerating', 'accelerator', 'accentuated', 'accentuating', 'accommodated', 'accommodating', 'accommodative', 'accumulated', 'accumulating', 'accumulative', 'accumulator', 'accumulators', ...]
- >>> [w for w, pron in entries if stress(pron) == ['0', '2', '0', '1', '0']] ['abbreviation', 'abbreviations', 'abomination', 'abortifacient', 'abortifacients', 'academicians', 'accommodation', 'accommodations', 'accreditation', 'accreditations', 'accumulation', 'accumulations', 'acetylcholine', 'acetylcholine', 'adjudication', ...]

- We can use a conditional frequency distribution to help us find minimally contrasting sets of words. Here we find all the *p* words consisting of three sounds and group them according to their first and last sounds .
- >>> p3 = [(pron[0]+'-'+pron[2], word
- ... for (word, pron) in entries
- ... if pron[0] == 'P' and len(pron) == 3]
- >>> cfd = nltk.ConditionalFreqDist(p3)
- >>> for template in cfd.conditions():
- ... if len(cfd[template]) > 10:
- ... words = cfd[template].keys()
- ... wordlist = ' '.join(words)
- ... print template, wordlist[:70] + "..."
- ...
- P-CH perch puche poche peach petsche poach pietsch putsch pautsch piche pet...

- P-K pik peek pic pique paque polk perc poke perk pac pock poch purk pak pa...
- P-L pil poehl pille pehl pol pall pohl pahl paul perl pale paille perle po...
- P-N paine payne pon pain pin pawn pinn pun pine paign pen pyne pane penn p...
- P-P pap paap pipp paup pape pup pep poop pop pipe paape popp pip peep pope...
- P-R paar poor par poore pear pare pour peer pore parr por pair porr pier...
- P-S pearse piece posts pasts peace perce pos pers pace puss pesce pass pur...
- P-T pot puett pit pete putt pat purt pet peart pott pett pait pert pote pa...
- P-Z pays p.s pao's pais paws p.'s pas pez paz pei's pose poise peas paiz p...
- Rather than iterating over the whole dictionary, we can also access it by looking up particular words.
- We look up a dictionary by specifying its name, followed by a **key** (such as the word 'fire') inside square brackets

- >>> prondict = nltk.corpus.cmudict.dict()
- >>> prondict['fire']
- [['F', 'AY1', 'ER0'], ['F', 'AY1', 'R']]
- >>> prondict['blog']
- Traceback (most recent call last):
- File "<stdin>", line 1, in <module> KeyError: 'blog'
- >>> prondict['blog'] = [['B', 'L', 'AA1', 'G']]
- >>> prondict['blog'] [['B', 'L', 'AA1', 'G']]

- We can use any lexical resource to process a text, e.g., to filter out words having some lexical property (like nouns), or mapping every word of the text. For example, the following text-to-speech function looks up each word of the text in the pronunciation dictionary:
- >>> text = ['natural', 'language', 'processing']
- >>> [ph for w in text for ph in prondict[w][0]]
- ['N', 'AE1', 'CH', 'ER0', 'AH0', 'L', 'L', 'AE1', 'NG', 'G', 'W', 'AH0', 'JH', 'P', 'R', 'AA1', 'S', 'EH0', 'S', 'IH0', 'NG']

- **Comparative Wordlists**
- Another example of a tabular lexicon is the **comparative wordlist**. NLTK includes so-called **Swadesh wordlists**, lists of about 200 common words in several languages. The languages are identified using an ISO 639 two-letter code.
- >>> from nltk.corpus import swadesh
- >>> swadesh.fileids()
- ['be', 'bg', 'bs', 'ca', 'cs', 'cu', 'de', 'en', 'es', 'fr', 'hr', 'it', 'la', 'mk', 'nl', 'pl', 'pt', 'ro', 'ru', 'sk', 'sl', 'sr', 'sw', 'uk']
- >>> swadesh.words('en')
- ['I', 'you (singular), thou', 'he', 'we', 'you (plural)', 'they', 'this', 'that', 'here', 'there', 'who', 'what', 'where', 'when', 'how', 'not', 'all', 'many', 'some',
- 'few', 'other', 'one', 'two', 'three', 'four', 'five', 'big', 'long', 'wide', ...]
- We can access cognate words from multiple languages using the entries() method, specifying a list of languages. With one further step we can convert this into a simple dictionary

- >>> fr2en = swadesh.entries(['fr', 'en'])
- >>> fr2en
- [('je', 'I'), ('tu, vous', 'you (singular), thou'), ('il', 'he'), ...]
- >>> translate = dict(fr2en)
- >>> translate['chien'] 'dog'
- >>> translate['jeter'] 'throw'
- We can make our simple translator more useful by adding other source languages. Let's get the German-English and Spanish-English pairs, convert each to a dictionary using dict(), then *update* our original translate dictionary with these additional mappings:
- >>> de2en = swadesh.entries(['de', 'en'])  # German-English
- >>> es2en = swadesh.entries(['es', 'en'])  # Spanish-English
- >>> translate.update(dict(de2en))
- >>> translate['Hund'] 'dog'
- >>> translate['perro'] 'dog'

- We can compare words in various Germanic and Romance languages:
- >>> languages = ['en', 'de', 'nl', 'es', 'fr', 'pt', 'la']
- >>> for i in [139, 140, 141, 142]:
- ...  print swadesh.entries(languages)[i]
- ...
- ('say', 'sagen', 'zeggen', 'decir', 'dire', 'dizer', 'dicere')
- ('sing', 'singen', 'zingen', 'cantar', 'chanter', 'cantar', 'canere')
- ('play', 'spielen', 'spelen', 'jugar', 'jouer', 'jogar, brincar', 'ludere')
- ('float', 'schweben', 'zweven', 'flotar', 'flotter', 'flutuar, boiar', 'fluctuare')

# • WordNet

- **WordNet** is a semantically oriented dictionary of English, similar to a traditional thesaurus but with a richer structure.
- NLTK includes the English WordNet, with 155,287 words and 117,659 synonym sets.
- We'll begin by looking at synonyms and how they are accessed in WordNet.
- **Senses and Synonyms**
- Consider the sentence in (1a). If we replace the word *motorcar* in (1a) with *automobile*, to get (1b), the meaning of the sentence stays pretty much the same:
    - a. Benz is credited with the invention of the motorcar.

      b. Benz is credited with the invention of the automobile.

- Since everything else in the sentence has remained unchanged, we can conclude that the words *motorcar* and *automobile* have the same meaning, i.e., they are **synonyms**.
- We can explore these words with the help of WordNet:
- >>> from nltk.corpus import wordnet as wn
- >>> wn.synsets('motorcar') [Synset('car.n.01')]
- Thus, *motorcar* has just one possible meaning and it is identified as car.n.01, the first noun sense of *car*.
- The entity car.n.01 is called a **synset**, or “synonym set,” a collection of synonymous words (or “lemmas”):
- >>> wn.synset('car.n.01').lemma_names
- ['car', 'auto', 'automobile', 'machine', 'motorcar']

- Each word of a synset can have several meanings, e.g., *car* can also signify a train carriage, a gondola, or an elevator car. However, we are only interested in the single meaning that is common to all words of this synset.
- Synsets also come with a prose definition and some example sentences:
- >>> wn.synset('car.n.01').definition
- 'a motor vehicle with four wheels; usually propelled by an internal combustion engine'
- >>> wn.synset('car.n.01').examples ['he needs a car to get to work']
- Although definitions help humans to understand the intended meaning of a synset, the *words* of the synset are often more useful for our programs. To eliminate ambiguity, we will identify these words as car.n.01.automobile, car.n.01.motorcar, and so on.
- This pairing of a synset with a word is called a lemma. We can get all the lemmas for a given synset, look up a particular lemma , get the synset corresponding to a lemma and get the “name” of a lemma:

- >>> wn.synset('car.n.01').lemmas
- [Lemma('car.n.01.car'), Lemma('car.n.01.auto'), Lemma('car.n.01.automobile'), Lemma('car.n.01.machine'), Lemma('car.n.01.motorcar')]
- >>> wn.lemma('car.n.01.automobile')
- Lemma('car.n.01.automobile')
- >>> wn.lemma('car.n.01.automobile').synset
- Synset('car.n.01')
- >>> wn.lemma('car.n.01.automobile').name 'automobile‘
- Unlike the words *automobile* and *motorcar*, which are unambiguous and have one synset, the word *car* is ambiguous, having five synsets:
- >>> wn.synsets('car')
- [Synset('car.n.01'), Synset('car.n.02'), Synset('car.n.03'), Synset('car.n.04'), Synset('cable_car.n.01')]
- >>> for synset in wn.synsets('car'):
- ... print synset.lemma_names
-

- ['car', 'auto', 'automobile', 'machine', 'motorcar'] ['car', 'railcar', 'railway_car', 'railroad_car'] ['car', 'gondola']
- ['car', 'elevator_car']
- ['cable_car', 'car']

- The WordNet Hierarchy:
- WordNet synsets correspond to abstract concepts, and they don't always have corresponding words in English. These concepts are linked together in a hierarchy. Some concepts are very general, such as *Entity*, *State*, *Event*; these are called **unique beginners** or root synsets. Others, such as *gas guzzler* and *hatchback*, are much more specific. A small portion of a concept hierarchy is illustrated in Figure:

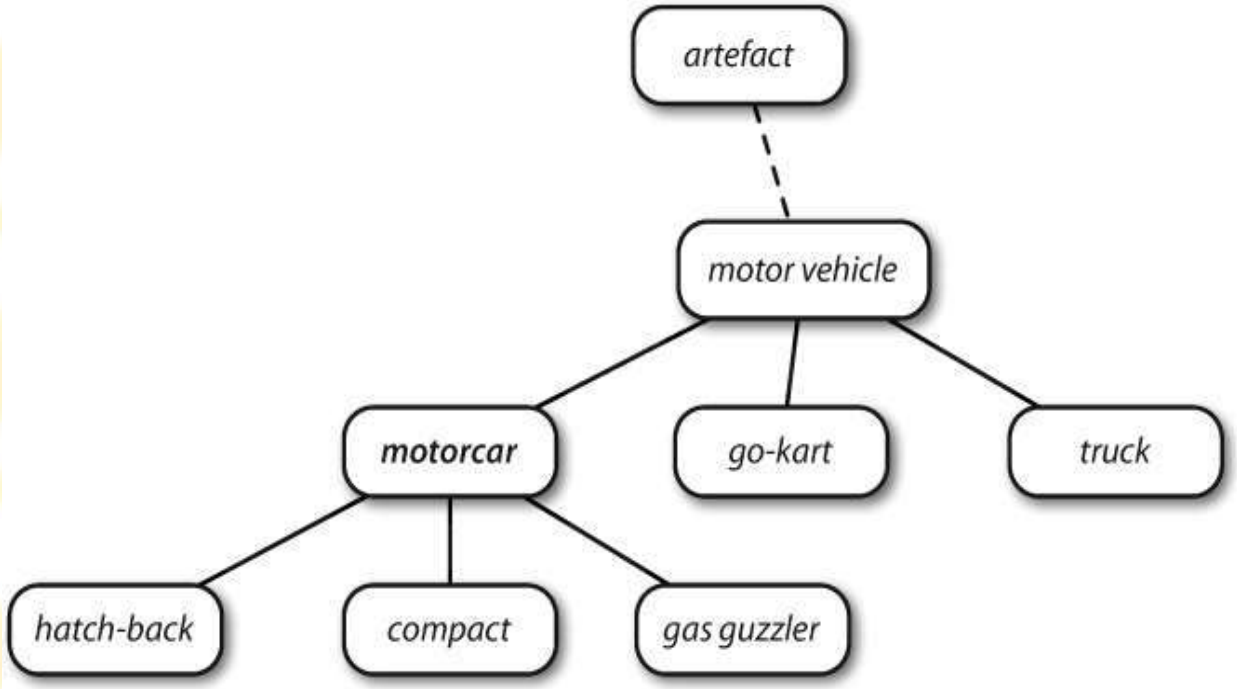



*Figure 2-8. Fragment of WordNet concept hierarchy: Nodes correspond to synsets; edges indicate the hypernym/hyponym relation, i.e., the relation between superordinate and subordinate concepts.*

- WordNet makes it easy to navigate between concepts. For example, given a concept like *motorcar*, we can look at the concepts that are more specific— the (immediate) **hyponyms**.
- >>> motorcar = wn.synset('car.n.01')
- >>> types_of_motorcar = motorcar.hyponyms()
- >>> types_of_motorcar[26] Synset('ambulance.n.01')
- >>> sorted([lemma.name for synset in types_of_motorcar for lemma in synset.lemmas])
- ['Model_T', 'S.U.V.', 'SUV', 'Stanley_Steamer', 'ambulance', 'beach_waggon', 'beach_wagon', 'bus', 'cab', 'compact', 'compact_car', 'convertible', 'coupe', 'cruiser', 'electric', 'electric_automobile', 'electric_car', 'estate_car', 'gas_guzzler', 'hack', 'hardtop', 'hatchback', 'heap', 'horseless_carriage', 'hot-rod', 'hot_rod', 'jalopy', 'jeep', 'landrover', 'limo', 'limousine', 'loaner', 'minicar', 'minivan', 'pace_car', 'patrol_car', 'phaeton', 'police_car', 'police_cruiser', 'prowl_car', 'race_car', 'racer', 'racing_car', 'roadster', 'runabout', 'saloon', 'secondhand_car', 'sedan', 'sport_car', 'sport_utility', 'sport_utility_vehicle', 'sports_car', 'squad_car', 'station_waggon', 'station_wagon', 'stock_car', 'subcompact', 'subcompact_car', 'taxi', 'taxicab', 'tourer', 'touring_car', 'two-seater', 'used-car', 'waggon', 'wagon']

- We can also navigate up the hierarchy by visiting hypernyms. Some words have multiple paths, because they can be classified in more than one way. There are two paths between car.n.01 and entity.n.01 because wheeled_vehicle.n.01 can be classified as both a vehicle and a container.
- >>> motorcar.hypernyms() [Synset('motor_vehicle.n.01')]
- >>> paths = motorcar.hypernym_paths()
- >>> len(paths) 2
- >>> [synset.name for synset in paths[0]]
- ['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01', 'instrumentality.n.03', 'container.n.01', 'wheeled_vehicle.n.01',
- 'self-propelled_vehicle.n.01', 'motor_vehicle.n.01', 'car.n.01']
- >>> [synset.name for synset in paths[1]]
- ['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01', 'instrumentality.n.03', 'conveyance.n.03', 'vehicle.n.01', 'wheeled_vehicle.n.01', 'self-propelled_vehicle.n.01', 'motor_vehicle.n.01', 'car.n.01']
- We can get the most general hypernyms (or root hypernyms) of a synset as follows:
- >>> motorcar.root_hypernyms() [Synset('entity.n.01')]

- **More Lexical Relations**
- Hypernyms and hyponyms are called **lexical relations** because they relate one synset to another. These two relations navigate up and down the "is-a" hierarchy. Another important way to navigate the WordNet network is from items to their components (**meronyms**) or to the things they are contained in (**holonyms**). For example, the parts of a *tree* are its *trunk*, *crown*, and so on; these are the part_meronyms(). The *substance* a tree is made of includes *heartwood* and *sapwood*, i.e., the substance_meronyms(). A collection of trees forms a *forest*, i.e., the member_holonyms():
- >>> wn.synset('tree.n.01').part_meronyms()
- [Synset('burl.n.02'), Synset('crown.n.07'), Synset('stump.n.01'), Synset('trunk.n.01'), Synset('limb.n.02')]
- >>> wn.synset('tree.n.01').substance_meronyms()
- [Synset('heartwood.n.01'), Synset('sapwood.n.01')]
- >>> wn.synset('tree.n.01').member_holonyms()
- [Synset('forest.n.01')]

- To see just how intricate things can get, consider the word *mint*, which has several closely related senses. We can see that mint.n.04 is part of mint.n.02 and the substance from which mint.n.05 is made.
- >>> for synset in wn.synsets('mint', wn.NOUN):
- ... print synset.name + ':', synset.definition
- ...
- batch.n.02: (often followed by `of') a large number or amount or extent mint.n.02: any north temperate plant of the genus Mentha with aromatic leaves and
- small mauve flowers
- mint.n.03: any member of the mint family of plants mint.n.04: the leaves of a mint plant used fresh or candied mint.n.05: a candy that is flavored with a mint oil
- mint.n.06: a plant where money is coined by authority of the government
- >>> wn.synset('mint.n.04').part_holonyms() [Synset('mint.n.02')]
- >>> wn.synset('mint.n.04').substance_holonyms() [Synset('mint.n.05')]

- There are also relationships between verbs. For example, the act of *walking* involves the act of *stepping*, so walking **entails** stepping. Some verbs have multiple entailments:
- >>> wn.synset('walk.v.01').entailments() [Synset('step.v.01')]
- >>> wn.synset('eat.v.01').entailments() [Synset('swallow.v.01'), Synset('chew.v.01')]
- >>> wn.synset('tease.v.03').entailments() [Synset('arouse.v.07'), Synset('disappoint.v.01')]
- Some lexical relationships hold between lemmas, e.g., **antonymy**:
- >>> wn.lemma('supply.n.02.supply').antonyms() [Lemma('demand.n.02.demand')]
- >>> wn.lemma('rush.v.01.rush').antonyms() [Lemma('linger.v.04.linger')]
- >>> wn.lemma('horizontal.a.01.horizontal').antonyms() [Lemma('vertical.a.01.vertical'), Lemma('inclined.a.02.inclined')]
- >>> wn.lemma('staccato.r.01.staccato').antonyms() [Lemma('legato.r.01.legato')]

- **Semantic Similarity**
- We have seen that synsets are linked by a complex network of lexical relations. Given a particular synset, we can traverse the WordNet network to find synsets with related meanings. Knowing which words are semantically related is useful for indexing a col- lection of texts, so that a search for a general term such as *vehicle* will match documents containing specific terms such as *limousine*.
- Recall that each synset has one or more hypernym paths that link it to a root hypernym such as entity.n.01. Two synsets linked to the same root may have several hypernyms in common (see Figure 2-8). If two synsets share a very specific hypernym—one that is low down in the hypernym hierarchy—they must be closely related.

- >>> right = wn.synset('right_whale.n.01')
- >>> orca = wn.synset('orca.n.01')
- >>> minke = wn.synset('minke_whale.n.01')
- >>> tortoise = wn.synset('tortoise.n.01')
- >>> novel = wn.synset('novel.n.01')
- >>> right.lowest_common_hypernyms(minke) [Synset('baleen_whale.n.01')]
- >>> right.lowest_common_hypernyms(orca) [Synset('whale.n.02')]
- >>> right.lowest_common_hypernyms(tortoise) [Synset('vertebrate.n.01')]
- >>> right.lowest_common_hypernyms(novel) [Synset('entity.n.01')]

- Of course we know that *whale* is very specific (and *baleen whale* even more so), whereas *vertebrate* is more general and *entity* is completely general. We can quantify this concept of generality by looking up the depth of each synset:
- >>> wn.synset('baleen_whale.n.01').min_depth() 14
- >>> wn.synset('whale.n.02').min_depth() 13
- >>> wn.synset('vertebrate.n.01').min_depth() 8
- >>> wn.synset('entity.n.01').min_depth()
- Similarity measures have been defined over the collection of WordNet synsets that incorporate this insight. For example, path_similarity assigns a score in the range 0–1 based on the shortest path that connects the concepts in the hypernym hierarchy (-1 is returned in those cases where a path cannot be found). Comparing a synset with itself will return 1. Consider the following similarity scores, relating *right whale* to *minke whale*, *orca*, *tortoise*, and *novel*. Although the numbers won't mean much, they decrease as we move away from the semantic space of sea creatures to inanimate objects.
- >>> right.path_similarity(minke) 0.25
- >>> right.path_similarity(orca) 0.16666666666666666
- >>> right.path_similarity(tortoise) 0.07692307692307692
- >>> right.path_similarity(novel) 0.043478260869565216