Lecture 08

Natural Language Processing

Zoran B. Djordjević

Natural Language Tool Kit (NLTK)

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 2017

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

  A version of the book for Python 3 can be found at: http://www.nltk.org/book/

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

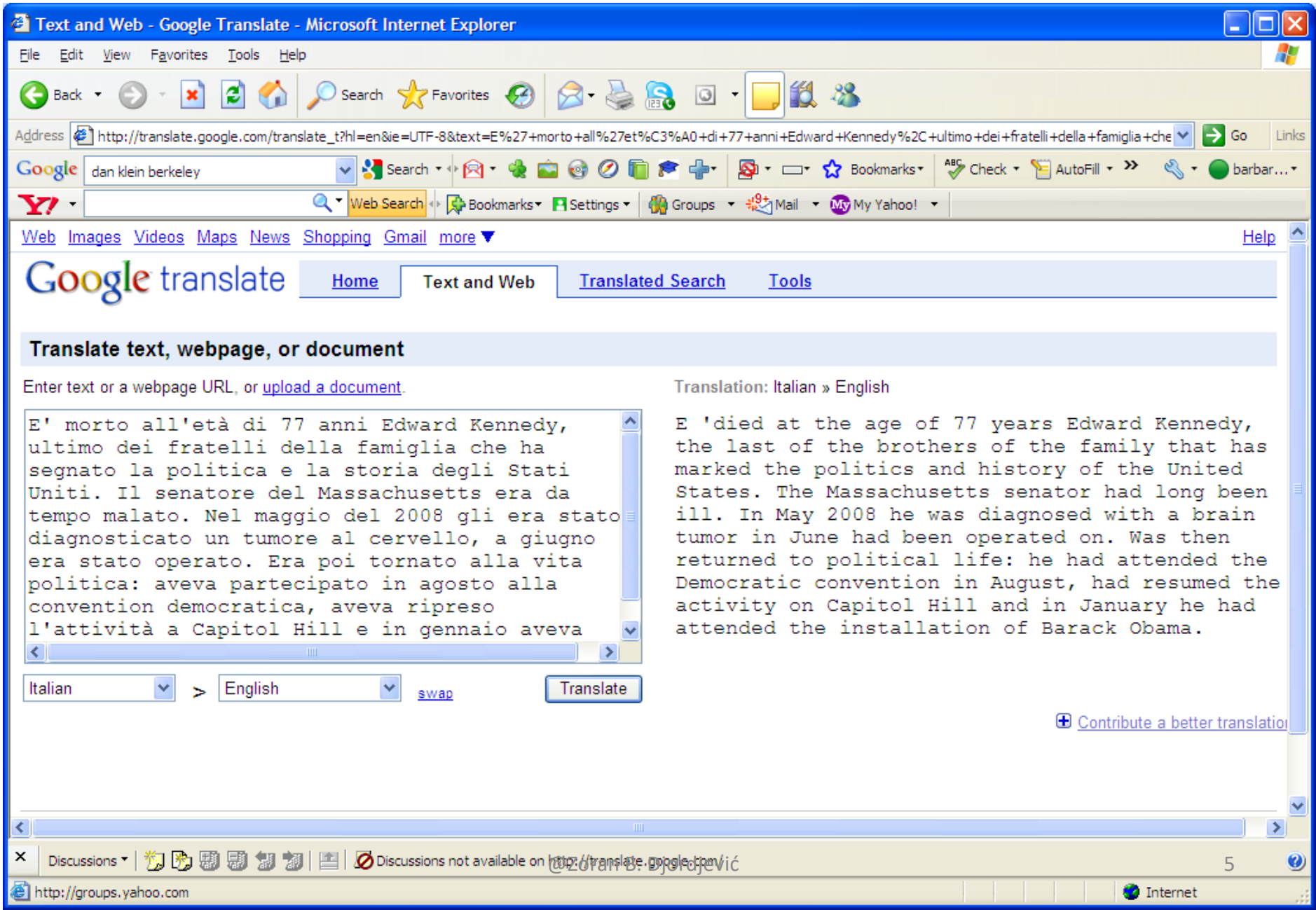
If you want to master NLP field, these are the essential books:

- *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 we 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 scientists in areas from humanities, computing and corpus linguistics to computer science and artificial intelligence.
- To many people in academia, NLP is also known as “Computational Linguistics.”

Example: Machine Translation



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
    - Word sense is ambiguous (head, arms)
  - Stolen Painting Found by Tree
    - Thematic role is ambiguous: tree is agent or location?
  - Ban on Nude Dancing on Governor's Desk
    - Syntactic structure (attachment) is ambiguous: is the ban or the dancing on the desk?
  - Hospitals Are Sued by 7 Foot Doctors
    - Semantics 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 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 ?
  - A word might represent a physical object 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 recent books and software frameworks on the subject are published in Python.
- Do not despair, you could find almost all of software you might need for NLP in Java, R, C/C++ , C# and a few other languages.

Python NLTK

- NLTK is a fairly popular framework that provides tools that could be used to build NLP programs in Python.
- NLTK provides
  - basic classes relevant to natural language processing;
  - standard interfaces for performing tasks such as part-of-speech tagging,
  - syntactic parsing, and
  - text classification;
- NLTK provides standard implementations for above and other tasks that could 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.
- This is not the best designed package in the World.
- NLTK has many deficiencies but it does many useful things.

Requirements

- *Python* as early as 2.4 has full NLTK. NLTK with Python 3.6 is better, one presumes.
- *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 calculus, 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 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 or even better Python 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.

- If you are using Anaconda Python, NLTK is already in.

- Once NLTK is installed, open python and import nltk:

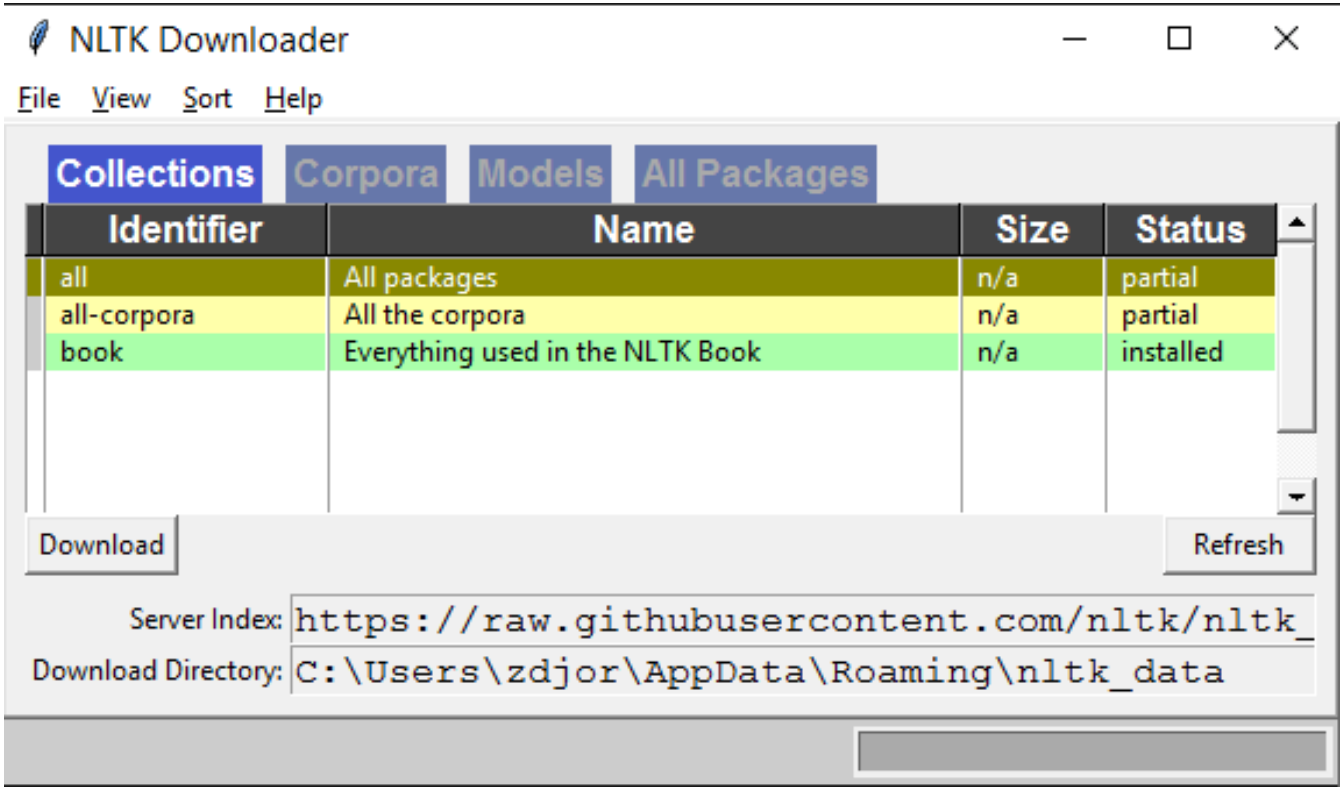
```
C:\> python
  import nltk
```

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.



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

What NLTK can do for you

- Tokenize and tag some text:

```
import nltk
sentence = """At eight o'clock on Thursday morning
        Arthur didn't feel very good."""
tokens = nltk.word_tokenize(sentence)
tokens
['At', 'eight', "o'clock", 'on', 'Thursday', 'morning',
'Arthur', 'did', "n't", 'feel', 'very', 'good', '.']
tagged = nltk.pos_tag(tokens)
tagged[0:6]
[('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]
```

Part-of-speech tags, Penn Treebank Project

- https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

- CC | Coordinating conjunction |
- CD | Cardinal number |
- DT | Determiner |
- EX | Existential *there* |
- FW | Foreign word |
- IN | Preposition or subordinating conjunction |
- JJ | Adjective |
- JJR | Adjective, comparative |
- JJS | Adjective, superlative |
- LS | List item marker |
- MD | Modal |
- NN | Noun, singular or mass |
- NNS | Noun, plural |
- NNP | Proper noun, singular |
- NNPS | Proper noun, plural |
- PDT | Predeterminer |
- POS | Possessive ending |
- PRP | Personal pronoun |
- PRP$ | Possessive pronoun |
- RB | Adverb |
- RBR | Adverb, comparative |
- RBS | Adverb, superlative |
- RP | Particle |
- SYM | Symbol |
- TO | *to* |
- UH | Interjection |
- VB | Verb, base form |
- VBD | Verb, past tense |
- VBG | Verb, gerund or present participle |
- VBN | Verb, past participle |
- VBP | Verb, non-3rd person singular present |
- VBZ | Verb, 3rd person singular present |
- WDT | Wh-determiner |
- WP | Wh-pronoun |
- WP$ | Possessive wh-pronoun |
- WRB | Wh-adverb |

Searching Text, Concordance

- There are many ways to examine the context of a text apart from simply reading it.
- A `concordance` view shows every occurrence of a given word, together with some surrounding text.
- 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 similar contexts?
- We can find out by appending the method `similar()` to the name of the text we are analyzing, then inserting the relevant word in quotes and 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
```

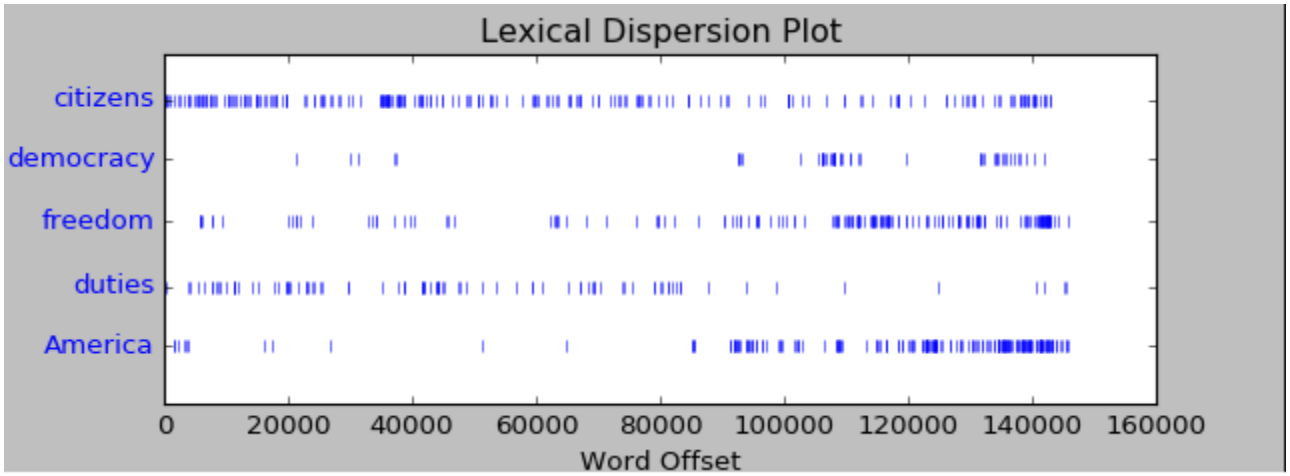
- We get different results for different texts. Jane Austen uses word `monstrous` quite differently from Melville; for her, *`monstrous`* has positive connotations, and sometimes functions as an intensifier like the word *very.*
- Function `common_contexts` allows us to examine just the contexts that 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"])
```



- `text4` is the corpus of the inaugural addresses

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. Function `set()` will fetch only the words themselves without counting repeated words

```
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 word 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.

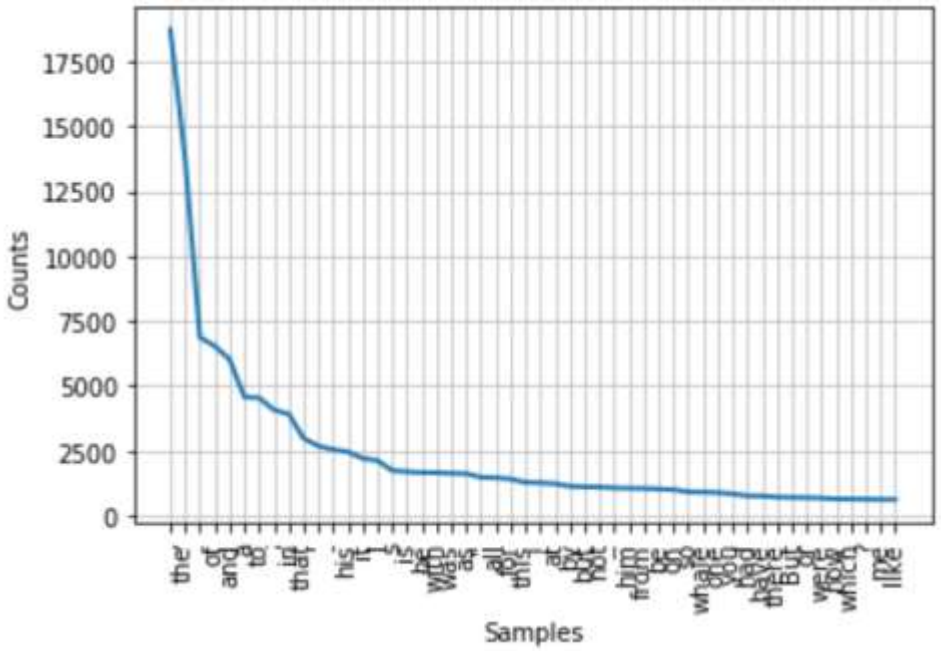
```
text4[:5]
['Fellow', '-', 'Citizens', 'of', 'the']
text2[141525]
'among'
text2[141525:]
['among', 'the', 'merits', 'and', 'the', 'happiness', 'of', 'Elinor', 'and',
'Marianne', ',', 'let', 'it', 'not', 'be', 'ranked', 'as', 'the', 'least',
'considerable', ',', 'that', 'though', 'sisters', ',', 'and', '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:

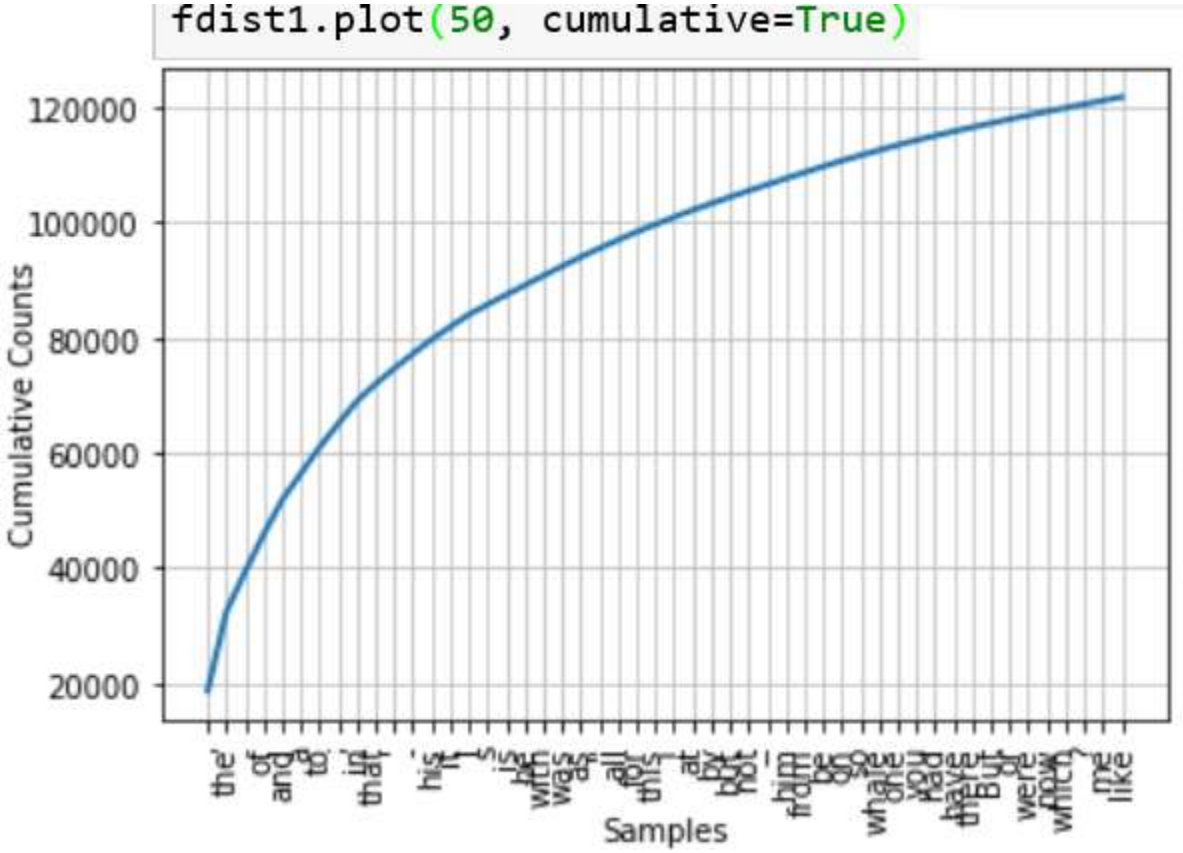
```
fdist1 = FreqDist(text1)
fdist1
FreqDist({',': 18713, 'the': 13721, '.': 6862, 'of': 6536, 'and': 6024,
'a': 4569, 'to': 4542, ';': 4072, 'in': 3916, 'that': 2982, ...})
fdist1.plot(50, cumulative=False)
```

- Only one word in this list, *whale*, is slightly informative! It occurs over 900 times. `text1.count('whale')>906`
- The rest of the words tell us nothing about the text; they're just English "plumbing."



Cumulative Distribution

- What proportion of the text is taken up with such “empty” words? We can generate a cumulative frequency plot for these words, using command: `fdist1.plot(50, cumulative=True)`.
- We see that 50 “stop” words, up to “like” consume 120,000 out of `len(text1) = 260,819` words in the novel.



Words Occurring Only Once

- Words occurring only once in the text:

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

[u'funereal', u'unscientific', u'prefix', u'plaudits', u'woody', u'disobeying', u'Westers', u'DRYDEN', u'Untried', u'superficially', u'vesper', u'Western', u'Spurn', u'treasuries', u'powders', u'tinkerings', u'bolting', u'stabbed', u'elevations', u'ferreting', u'wooded', u'songster', u'Saco', u'clings', u'Winding', u'Sands', u'spindle', u'ornamental', u'charter', u'puissant', u'miller', u'cordially', u'railing', u'mail', u'Hecla', u'compliance', u'haughtily', u'relieving', u'BERMUDAS', u'contributed', u'shamble', u'fossil', u'unconsciousness', u'vacation', u'distinguishing', u'vacuity', u'majestically', u'1729', u'1726', u'flipped', u'Sorrows', u'unflattering', u'Zoology', u'Kant', u'yearly', u'snuffling', u'loveliest', u'hollowing', u'LANTERNS', u'glutinous', u'ants', u'Hogarthian', u'Comparing', u'anti', u'geese', u'rollings', u'Pious', u'cranium', u'vacuum', u'inferiors', u'snare', u'61', u'clad', u'wordless',

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)
```

[u'CIRCUMNAVIGATION',
u'Physiognomically',
u'apprehensiveness',
u'cannibalistically',
u'characteristically',
u'circumnavigating',
u'circumnavigation',
u'circumnavigations',
u'comprehensiveness',
u'hermaphroditical',
u'indiscriminately',
u'indispensableness',
u'irresistibleness',
u'physiognomically',
u'preternaturalness',
u'responsibilities',
u'simultaneousness',
u'subterraneousness',
u'supernaturalness',
u'superstitiousness',
u'uncomfortableness',
u'uncompromisedness',
u'undiscriminating',
u'uninterpenetratingly']

Word Characterizing a Text

- Very long words frequently occur only once (there are a few learned 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. *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 accomplished with the following operations:

```
from nltk.collocations import *
bigram_measures = nltk.collocations.BigramAssocMeasures()
finder = BigramCollocationFinder.from_words(['more', 'is',
'said', 'than', 'done'])
finder.nbest(bigram_measures.pmi,10)
```

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

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

Collocation and Bigrams

- Function `collocations()` finds bigrams on an arbitrary text. For example:

```
text4
<Text: Inaugural Address Corpus>
text4.collocations()
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 a sequence of words:

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

- Related task: probability of an upcoming word:

  $P(w_5|w_1, w_2, w_3, w_4)$, probability that 5th word is w_5 given that first 4 words are w_1, w_2, w_3, and w_4

- A model that computes either of these:

  $P(W)$ *or* $P(w_n|w_1, w_2 \dots w_{n-1})$ is called **a language model**.

- It would be better: **a grammar ,** but the **language model** or **LM** is standard
- We calculate above probabilities relying on so called chain rule.
- Recall the definition of conditional probability: $P(A, B) = P(A)P(B|A)$
- Or, in case of several 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 for n events reads:

  $P(x_1, x_2, x_3, \dots, x_n) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2) \dots P(x_n|x_1, \dots, x_{n-1})$

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 accounts for the presence of the previous word only:

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

N-gram models

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

  "The computer which I had just put into the machine room on the fifth floor crashed." (words computer and crashed are 15 positions away one from another but related and we understand that.)
- However, we can often get away with N-gram models

- By the way, you can get from Google, the actual, Google 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.
- The result often sounds like a real sentence.

```
<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 list of Shakespeare's bigrams)

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 a list containing a length of every word in the text. `[:20]` displays the first 20 word so we do not scroll forever . Next we establish the frequency distribution of those lengths

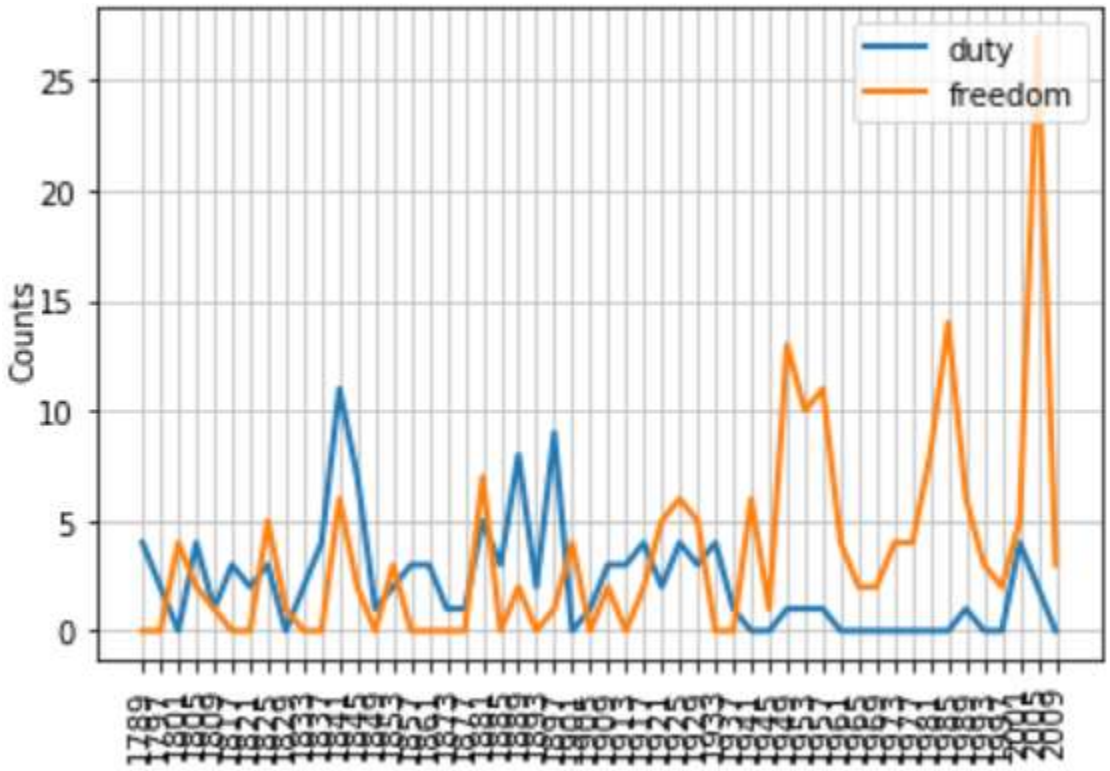
```
fdist = FreqDist([len(w) for w in text1])
fdist
FreqDist({3: 50223, 1: 47933, 4: 42345, 2: 38513, 5: 26597, 6: 17111, 7: 14399,
8: 9966, 9: 6428, 10: 3528, ...})
fdist.keys()
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20])
>>>
fdist.items()
[(3, 50223), (1, 47933), (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

Curiosity, Correlation of duty and freedom

```
from nltk.corpus import inaugural
inaugural.fileids()
['1789-Washington.txt', '1793Washington.txt', '1797-Adams.txt', '1801-
Jefferson.txt', '1805-Jefferson.txt', '1809-Madison.txt', . ..]
cfd = nltk.ConditionalFreqDist((target, fileid[:4])
     for fileid in inaugural.fileids()
     for w in inaugural.words(fileid)
     for target in ['duty','freedom']
     if w.lower()
       .startswith(target))
cfd.plot()
```



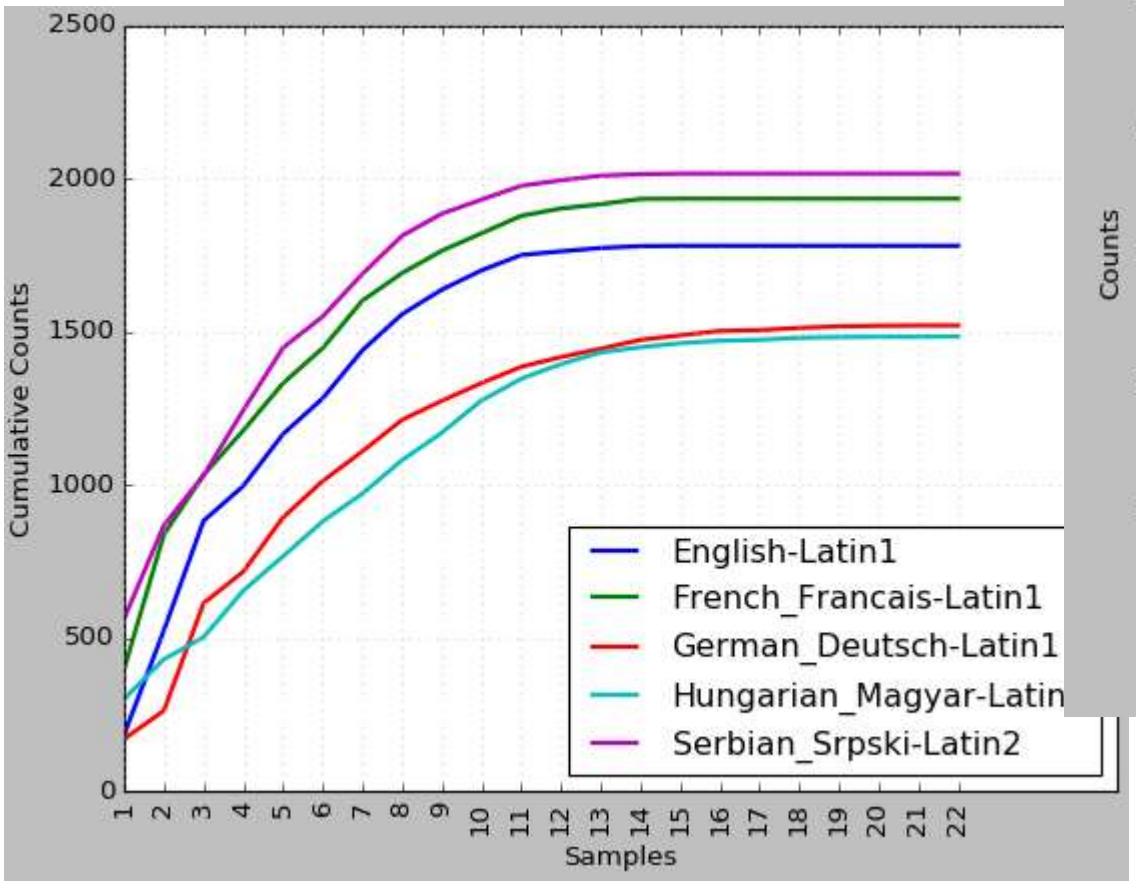
Annotated Text Corpora

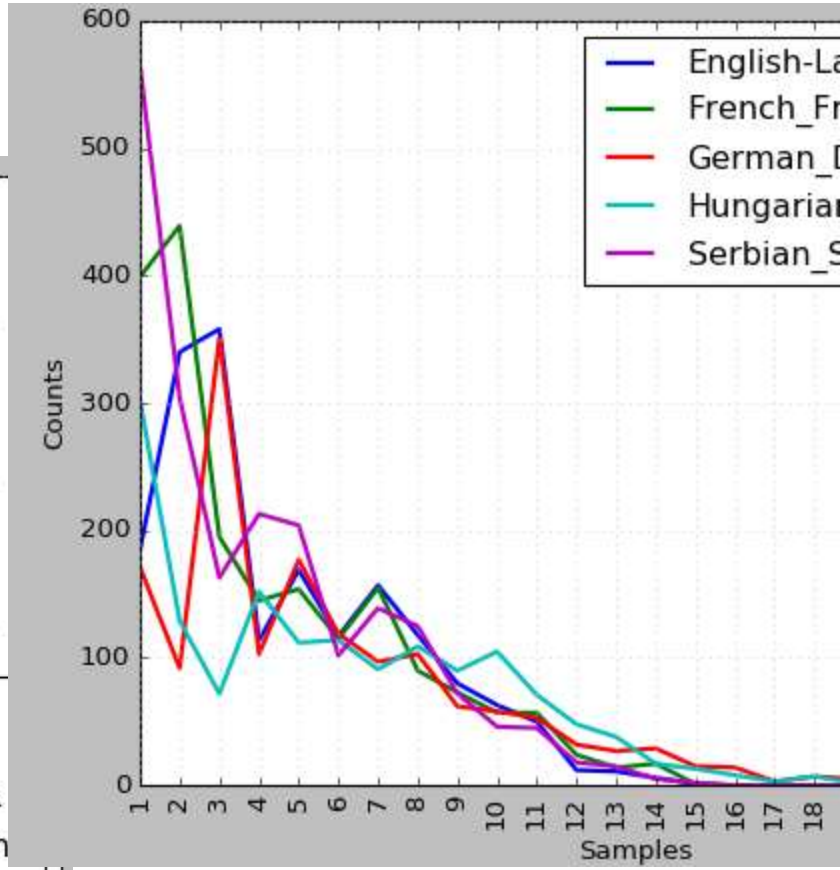
- Many text corpora contain linguistic annotations, representing POS tags, named entities, syntactic structures, semantic roles, and so forth.
- NLTK provides convenient ways to access several of these corpora, and has data packages containing corpora and corpus samples, freely downloadable for use in teaching and research.
- For information about downloading them, see http://nltk.org/data.
- For more examples of how to access NLTK corpora, please the Corpus HOWTO at http://nltk.org/howto.

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
CoNLL 2002 Named Entity	CoNLL	700k words, pos- and named-entity-tagged (Dutch, Spanish)
CoNLL 2007 Dependency Treebanks (sel)	CoNLL	150k words, dependency parsed (Basque, Catalan)
Dependency Treebank	Narad	Dependency parsed version of Penn Treebank sample
FrameNet	Fillmore, Baker et al	10k word senses, 170k manually annotated sentences
Floresta Treebank	Diana Santos et al	9k sentences, tagged and parsed (Portuguese)
Gazetteer Lists	Various	Lists of cities and countries
Genesis Corpus	Misc web sources	6 texts, 200k words, 6 languages
Gutenberg (selections)	Hart, Newby, et al	18 texts, 2M words
Inaugural Address Corpus	CSpan	US Presidential Inaugural Addresses (1789-present)
Indian POS-Tagged Corpus	Kumaran et al	60k words, tagged (Bangla, Hindi, Marathi, Telugu)
MacMorpho Corpus	NILC, USP, Brazil	1M words, tagged (Brazilian Portuguese)
Movie Reviews	Pang, Lee	2k movie reviews with sentiment polarity classification
Names Corpus	Kantrowitz, Ross	8k male and female names
NIST 1999 Info Extr (selections)	Garofolo	63k words, newswire and named-entity SGML markup
Nombank	Meyers	115k propositions, 1400 noun frames
NPS Chat Corpus	Forsyth, Martell	10k IM chat posts, POS-tagged and dialogue-act tagged

Universal Declaration of Human Rights

```
from nltk.corpus import udhr
languages = ['Serbian_Srpski-Latin2', 'English-Latin1', 'German_Deutsch-Latin1',
'French_Francais-Latin1', 'Hungarian_Magyar-Latin1']
cfd = nltk.ConditionalFreqDist(
...             (lang, len(word))
...             for lang in languages
...             for word in udhr.words(lang))
cfd.plot(cumulative=True)
```





Text Normalization

- Stemming
- Converting 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

- You can 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', '.']
```

- Above 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, Relevance

- 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*)
- 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 unintelligible 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

NLTK Lemmatizer

- NLTK Uses WordNet Lemmatizer
- Lemmatization is a 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. (http://wordnet.princeton.edu/man/morphy.7WN.html)

WordNet

- *WordNet® is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. The resulting network of meaningfully related words and concepts can be navigated with the browser. WordNet is also freely and publicly available for download. WordNet's structure makes it a useful tool for computational linguistics and natural language processing.*
- *WordNet superficially resembles a thesaurus, in that it groups words together based on their meanings. However, there are some important distinctions. First, WordNet interlinks not just word forms—strings of letters—but specific senses of words. As a result, words that are found in close proximity to one another in the network are semantically disambiguated. Second, WordNet labels the semantic relations among words, whereas the groupings of words in a thesaurus does not follow any explicit pattern other than meaning similarity.*

WordNet 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

```
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()
wordnet_lemmatizer.lemmatize('dogs')
u'dog'
wordnet_lemmatizer.lemmatize('are')
'are'
wordnet_lemmatizer.lemmatize('are',pos='v')
u'be‘
```

- The default pos tag is NOUN (n). Lemmatization might does output the correct lemma for a verb, unless the pos tag is explicitly specified as VERB (v).

Word2Vec

Representations of Text

Representation of text is very important for performance of many real-world applications. The most common techniques are:

- Local representations
  - N-grams
  - Bag-of-words
  - 1-of-N coding
- Continuous representations
  - Latent Semantic Analysis
  - Latent Dirichlet Allocation
  - **Distributed Representations**

Vector Space Model or Bag of Words Model

- We have seen previously that classical language processing technologies including the original Google' Page Rank system treat documents and queries as vectors in a very high dimensional space

$$d_j = \left(w_{1,j}, w_{2,j}, \ldots\ldots, w_{t,j}\right) \ , \quad q \ = (w_{1,q}, w_{2,q}, \ldots\ldots, w_{n,q})$$

- Where d_j is the vector representing document with index j. The document has t distinct terms.
- q represents a query string. That string has n distinct terms (words of significance).

- Each dimension corresponds to a separate term. If a term occurs in the document, its value in the vector is nonzero. Several different ways of computing these values, also known as (term) weights, have been developed. One of the best known schemes is $tf - idf$ weighting.

- The definition of term depends on the application. Typically terms are single words, keywords, or longer phrases.
- If words are chosen to be the terms, the dimensionality of the vector is the number of words in the vocabulary, i.e. the number of distinct words (terms) occurring in the corpus.
- This model is frequently called Bag of Words model since ordering of words is considered unimportant.

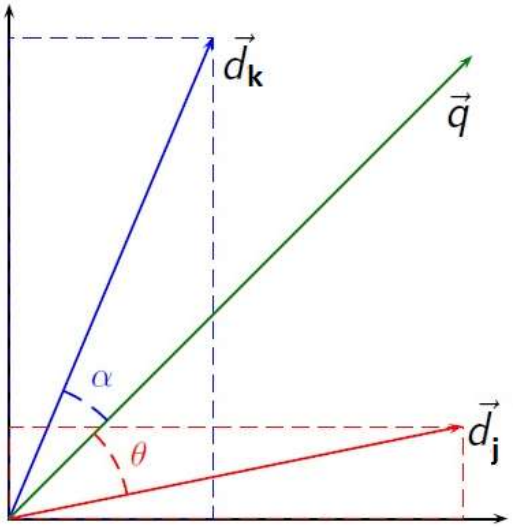
Comparing Documents and Queries

- Whether a document is “relevant” as an answer to a query string can be determined in a variety of ways. On the basic level we usually look for the similarity of the query string and the document string.
- Since both the query and the documents are “vectors”, similarity or alignment of those two vectors could conveniently be expressed by the cosine of the angle between two vectors

$$\cos(\theta) = \frac{d_j \cdot q}{\|d_j\| \|q\|}$$

- The dot product of two vectors is calculated as:

$$d_j \cdot q = \sum_{m=1}^{t} w_{m,j}\, w_{m,q}$$



- In the diagram on the right vector q is more similar
- to vector d_k than to the vector d_j , since angle α is smaller than angle θ, i. e. $\cos(\alpha) > \cos(\theta)$)
- Norms of two vectors are standard Euclidian norms

$$\|q\| = \sqrt[2]{\sum_{i=1}^{n} q_i^{\,2}}$$

tf-idf weights

- In the classic vector space model proposed by Salton, Wong and Yang the term specific weights in the document vectors are products of local and global parameters. The model is known as term frequency- inverse document frequency model.
- Weights $w_{t,d}$ in vector representing document $d_j = (w_{1,j}, w_{2,j}, \ldots, w_{t,j})$ are calculated as:

$$\boldsymbol{w_{t,d}=tf_{t,d} \cdot log \frac{|D|}{|\{d' \epsilon D | t \epsilon d'\}|}}$$

- $tf_{t,d}$ is the term frequency, i.e. the number of times term t appears in document d (a local parameter).
- $log \frac{|D|}{|\{d' \epsilon D | t \epsilon d'\}|}$ is called inverse document frequency.
- $|D|$ is the total number of documents in the document set (corpus).
- $|\{d' \in D | t \in d'\}|$ is the number of documents containing term t. Notice that if a term is present in a smaller number of documents d', its weight is greater (denominator of the logarithmic expression is smaller). That term has a higher discriminatory power.

- This model, while it might appear too simplistic, is extensively used and is not unsuccessful. Most of information retrieval applications build so far use this model.

Advantages and Limitations of Vector Space Model

- The vector space model has some advantages:
    1. Simple model based on linear algebra
    2. Term weights not binary
    3. Allows computing a continuous degree of similarity between queries and documents
    4. Allows ranking documents according to their possible relevance
    5. Allows partial matching
- The model has a few limitations
    1. Enormous dimensionality of any language makes documents very sparse.
    2. Long documents are poorly represented because they have poor similarity values (a small scalar product and a large dimensionality)
    3. Search keywords must precisely match document terms; word substrings might result in a "false positive match“
    4. Semantic sensitivity; documents with similar context but different term vocabulary won't be associated.
    5. The order in which the terms appear in the document is lost in the vector space representation.
    6. All terms are statistically independent.
    7. Weighting is intuitive but not very formal.

Distributional Semantics

- One of the ideas to improve on Vector Space Model is called Distributional Semantics.
- The basic idea of distributional semantics is in the so called distributional hypothesis:

  "linguistic items with similar distributions have similar meanings".
- Another way to express this hypothesis is to state:

  "words that are used and occur in the same contexts tend to purport similar meanings." or

  "a word is characterized by the company it keeps"
- In recent years, the distributional hypothesis has provided the basis for the theory of similarity based generalization in language learning: the idea that children can figure out how to use words they've rarely encountered before by generalizing about their use from distributions of similar words.
- The distributional hypothesis suggests that the more semantically similar two words are, the more distributionally similar they will be in turn, and thus the more that they will tend to occur in similar linguistic contexts.
- Whether or not this suggestion holds has significant implications for both the data sparsity problem in computational modeling, and for the question of how children are able to learn language so rapidly given relatively impoverished input.

Applications of Distributional Semantics

- Distributional semantic models were successfully applied for the following tasks:
  - language translation
  - finding semantic similarity between words and multiword expressions;
  - word clustering based on semantic similarity;
  - automatic creation of thesauri and bilingual dictionaries;
  - lexical ambiguity resolution;
  - expanding search requests using synonyms and associations;
  - defining the topic of a document;
  - document clustering for information retrieval;
  - data mining and named entities recognition;
  - creating semantic maps of different subject domains;
  - paraphrasing;
  - sentiment analysis;
  - modeling selectional preferences of words.
- Semantics is the study of meaning. It investigates questions such as: What is meaning? How come words and sentences have meaning?

Linguistic Context

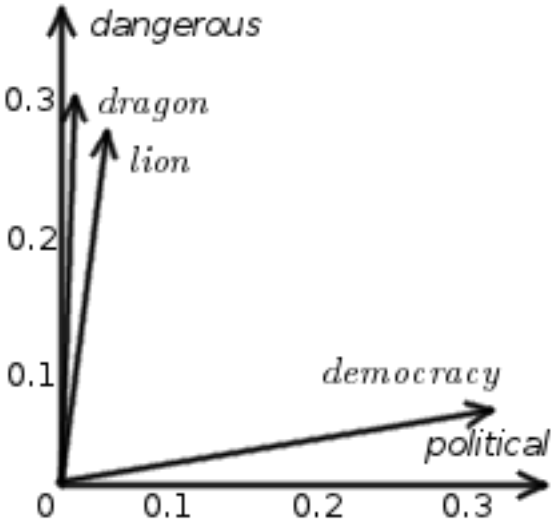
- The linguistic context of a word is simply the collection of other words that appear next to it. So for example, the following might be a context for ‘coconut’:

found throughout the tropic and subtropic area, the . . . . . is known for its great versatility

- Let’s now assume that we collect every single instance of the word ‘coconut’ in Wikipedia, and start counting how many times it appears next to ‘tropic’, ‘versatility’, ‘the’, ‘subatomic’, etc.
- We would probably find that coconuts appear many more times next to ‘tropic’ than next to ‘subatomic’: this is a good indication of what coconuts are about.
- But counting is not enough. The word ‘coconut’ appears thousands of times next to ‘the’, which doesn’t mean that ‘the’ tells us very much about what coconuts are.
- So usually, the raw counts are combined into statistical measures which help us define more accurately which words are ‘characteristic’ for a particular term. Pointwise Mutual Information is one such measure, which gives higher values to words that are really defining for a term: for instance, ‘tropic’, ‘food’, ‘palm’ for ‘coconut’.
- Once we have such measures, we can build a simulation of how the words in a particular language relate to each other. This is done in a so-called ‘semantic space’ which, mathematically, corresponds to a vector space. The dimensions of the vector space are context words like ‘tropic’ or ‘versatility’ and words are defined as points (or vectors) in that space. Semantic space is typically of a lower dimension than previously discussed Term Vector Space.

Similarity in Semantic Space

- Here is a very simplified example, where I define the words ‘dragon’, ‘lion’ and ‘democracy’ with respect to only two dimensions: ‘dangerous’ and ‘political’.
  - Because dragons and lions have much to do with being dangerous but very little with political systems, their vectors have a high value along the ‘dangerous’ axis and a small value along the ‘political’ axis.
  - The opposite is true for ‘democracy’. Note that very naturally, the vectors for ‘dragon’ and ‘lion’ have clustered together in the space, while ‘democracy’ is much further from them.
  - By calculating the distance between two vectors in a semantic space, we can deduce the similarity of two words: this is what allows us to say that cats and dogs are more alike than cats and coconuts.
  - In regular usage dimensionality of semantic space is several hundreds or several thousands, but not hundreds or thousands or millions.



Word2Vec

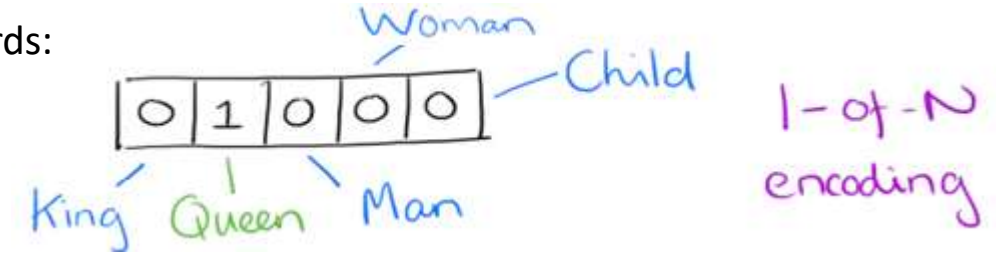
- Google's researchers (Tomas Mikolov et al.) created a tool called **Word2Vec** which efficiently implements ideas of Distributed Semantics.
- Word2Vec is a two-layer neural net that processes text. Its input is a text corpus and its output is a set of vectors: feature vectors for words in that corpus.
- While Word2vec is not a deep neural network, it turns text into a numerical form that deep nets can understand.
- The **Word2vec** tool first constructs a vocabulary from the training text data and then learns vector representation of words. The resulting word vector file can be used as features in many natural language processing and machine learning applications.
- A simple way to investigate the learned representations is to find the closest words for a user-specified word. The **distance** tool serves that purpose. For example, if you enter 'france', **distance** will display the most similar words and their distances to 'france', which should look like:

Word	**Cosine distance**
spain	0.678515
belgium	0.665923
netherlands	0.652428
italy	0.633130
switzerland	0.622323

Word	**Cosine distance**
luxembourg	0.610033
portugal	0.577154
russia	0.571507
germany	0.563291
catalonia	0.534176

One-hot Encoding vs. Word2Vec

- In a simple 1-of-N (or 'one-hot') encoding every element in the vector is associated with a word in the vocabulary. The encoding of a given word is simply the vector in which the corresponding element is set to one, and all other elements are zero.
- Suppose our vocabulary has only five words:
- King, Queen, Man, Woman, and
- Child. We could encode the word
- 'Queen' as:

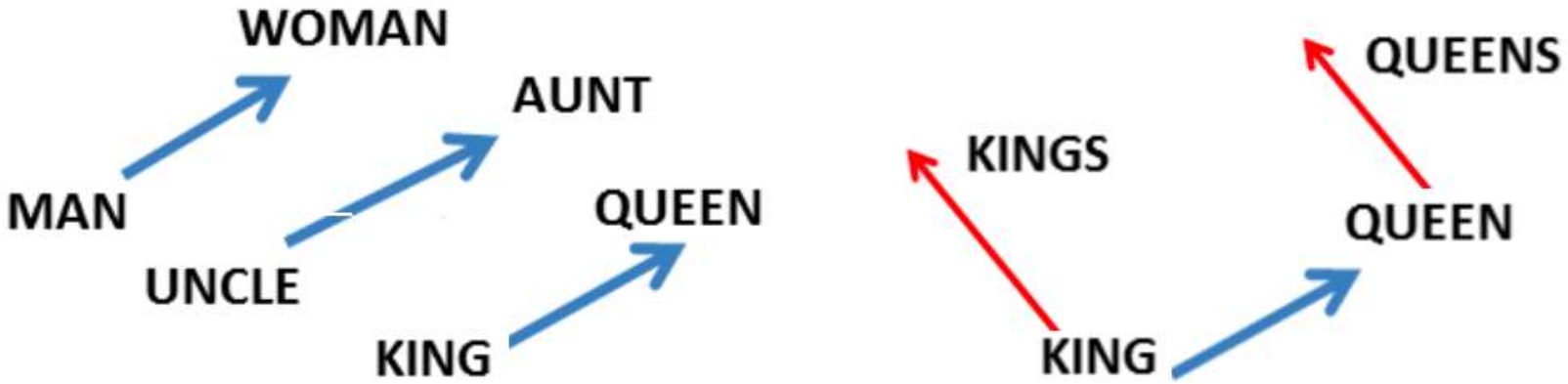


- In word2vec, a "*distributed*" representation of a word is used. The vector space has several hundred dimensions (say 300 or 1000). Each word is represented by a distribution of weights across those dimensions. So instead of a one-to-one mapping between an element in the vector and a word, the representation of a word is spread across all of the elements in the vector, and each element in the vector contributes to the definition of many words.
- If I label the dimensions in a hypothetical word vector (there are no such pre-assigned labels in the algorithm of course), it might look a bit like this:

- Such vectors represent in an abstract way the 'meaning' of words. By examining a large corpus it's possible to learn word vectors that are able to capture the relationships between words in a surprisingly expressive way.

Reasoning with word vectors

- We find that the learned word representations capture meaningful syntactic and semantic regularities in a very simple way. Specifically, the regularities are observed as constant vector offsets between pairs of words sharing a particular relationship.
- For example, if we denote the vector for word i as x_i, and focus on the singular/plural relation, we observe that $x_{apple} - x_{apples} \approx x_{car} - x_{cars}, x_{family} - x_{families} \approx x_{car} - x_{cars}$, and so on. Perhaps more surprisingly, we find that this is also the case for a variety of semantic relations.
- The vectors are very good at answering analogy questions of the form *a* is to *b* as *c* is to *?*. For example, *man* is to *woman* as *uncle* is to *?* (*aunt*) using a simple vector offset method based on cosine distance.
- For example, here are vector offsets for three word pairs illustrating the gender relation and plural-singular relation:



Algebraic Operations on Vectors

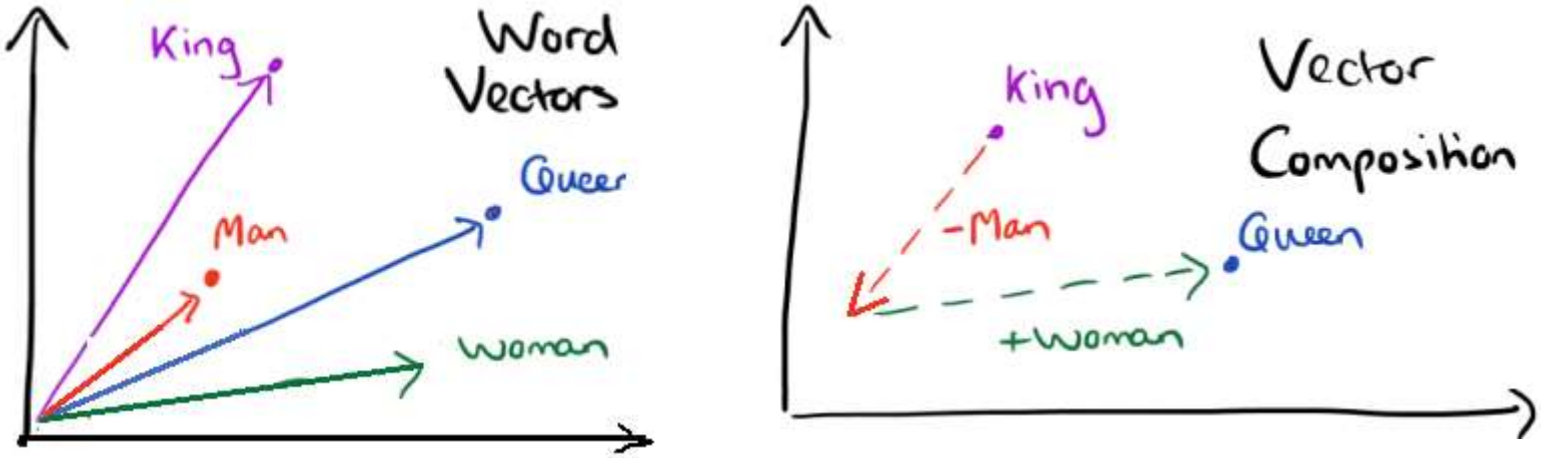
- This kind of vector composition also lets us answer “King – Man + Woman = ?” question and arrive at the result “Queen” ! All of which is truly remarkable when you think that all of this knowledge simply comes from looking at lots of word in context (as we’ll see soon) with no other information provided about their semantics.
- It was found that similarity of word representations goes beyond simple syntactic regularities. Using a word offset technique where simple algebraic operations are performed on the word vectors.
- We will see that we can perform vector algebra on “words” and calculate expressions like:

$$vector(\text{“}King\text{”}) - vector(\text{“}Man\text{”}) + vector(\text{“}Woman\text{”})$$

- The result will be a vector that is very close to the vector representation of the word ”Queen”.

Vector Subtraction and Addition

- If you recall vector operations, you can subtract one vector from another. Subtraction is the same as adding a negative vector. Similarly you can add one vector to another.
- So, if we subtract vector for word “man” from the vector for word “King” and then add to the resulting vector a vector for work “woman”, we will arrive at the vector for word “Queen”. Those operations are schematically presented bellow:



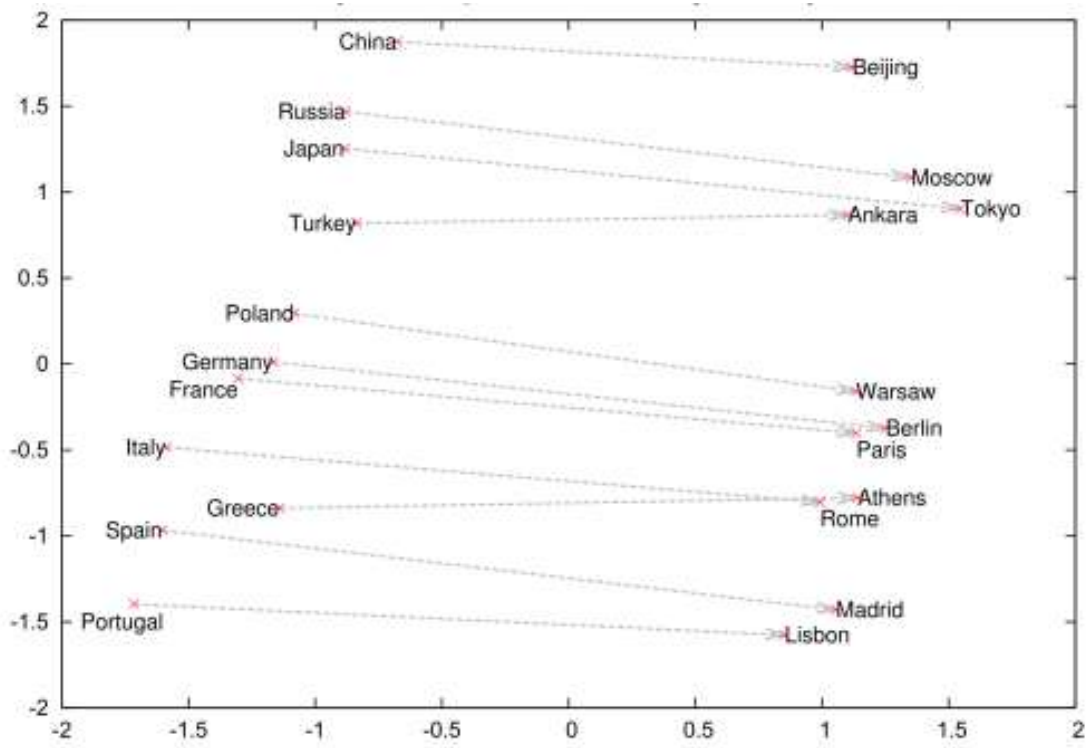
Examples of Word-Pair Relationships

- Examples of the world pair relationships using the best word vectors (skip-gram model trained on 783 M words in space of 300 dimensions).
- For example, if you say relationship is France – Paris, what is the corresponding word for Italy, the answer will be Rome. Similarly, if the basic relationship is Japan – sushi, and we ask for the word corresponding to Germany we will get bratwurst

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Countries vs. Capitals, PCA 2-d Projection

- Two dimensional (PCA) projection of the 1000 dimensional Skip-gram vectors of countries and their capitals. This illustrates the ability of the model to automatically organize concepts and learn implicitly the relationships between them. During the training no supervised information was provided suggesting what is the meaning of the capital city.
  - Perhaps because of the Christmas, Turkey and Ankara are connected by a flat vector different from most other country-capital pairs.
  - That of course does not explain the same flat angle between Greece and Athens.
  - It might be too fetched to argue that Greece and Turkey are neighbors.



Analogical Reasoning

- Here are some more examples of the ‘a is to b as c is to ?’ style questions answered by word vectors. The goal is to compute the fourth phrase using the first three. The claim that they could achieve 72% accuracy on a test set of 3218 examples.

Newspapers			
New York	New York Times	Baltimore	Baltimore Sun
San Jose	San Jose Mercury News	Cincinnati	Cincinnati Enquirer
NHL Teams			
Boston	Boston Bruins	Montreal	Montreal Canadiens
Phoenix	Phoenix Coyotes	Nashville	Nashville Predators
NBA Teams			
Detroit	Detroit Pistons	Toronto	Toronto Raptors
Oakland	Golden State Warriors	Memphis	Memphis Grizzlies
Airlines			
Austria	Austrian Airlines	Spain	Spainair
Belgium	Brussels Airlines	Greece	Aegean Airlines
Company executives			
Steve Ballmer	Microsoft	Larry Page	Google
Samuel J. Palmisano	IBM	Werner Vogels	Amazon

Element-wise Addition of vector elements

- We can also use element-wise addition of vector elements to ask questions such as ‘German + airlines’ and by looking at the closest tokens to the composite vector come up with impressive answers:

Czech + currency	Vietnam + capital	German + airlines	Russian + river	French + actress
koruna	Hanoi	airline Lufthansa	Moscow	Juliette Binoche
Check crown	Ho Chi Minh City	carrier Lufthansa	Volga River	Vanessa Paradis
Polish zolty	Viet Nam	flag carrier Lufthansa	upriver	Charlotte Gainsbourg
CTK	Vietnamese	Lufthansa	Russia	Cecile De

- Vector compositionality using element-wise addition. Four closest tokens to the sum of two vectors are shown, using the best Skip-gram model.
- Word vectors with such semantic relationships could be used to improve many existing NLP applications, such as machine translation, information retrieval and question answering systems, and may enable other future applications yet to be invented.

Semantic-Syntactic

- The Semantic - Syntatic word relationship tests for understanding of a wide variety of relationships as shown below. Using 640-dimensional word vectors, a skip-gram trained model achieved 55% semantic accuracy and 59%

Model Architecture	Semantic-Syntactic Word Relationship test set		MSR Word Relatedness Test Set [20]
	Semantic Accuracy [%]	Syntactic Accuracy [%]	
RNNLM	9	36	35
NNLM	23	53	47
CBOW	24	64	61
Skip-gram	55	59	56

- Comparison of architectures using models trained on the same data, with 640-dimensional word vectors. The accuracies are reported on Semantic-Syntactic World Relationship test set and on a syntactic test set.

Linguistic Regularity in Word Vector Space

Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

Rare Words, Examples of nearest neighbors

- Word vectors from neural networks were previously criticized for their poor performance on rare words
- Scaling up training data set size helps to improve performance on rare words

Method Used	Redmond	Havel	graffiti	capitulate
Collobert NNLM	conyers lubbock keene	plauen dzerzhinsky osterreich	cheesecake gossip dioramas	abdicate accede rearm
Turian NNLM	McCarthy Alston Cousins	Jewell Arzu Ovitz	gunfire emotion impunity	- - -
Mnih NNLM	Podhurst Harlang Agarwal	Pontiff Pinochet Rodionov	anaesthetics monkeys Jews	Mavericks planning hesitated
Skip-gram (phrases)	Redmond Wash. Redmond Washington Microsoft	Vaclav Havel president Vaclav Havel Velvet Revolution	spray paint grafitti taggers	capitulation capitulated capitulating

Forming Phrases from Words

- Example query:

  restaurants in mountain view that are not very good
- Forming the phrases:

  restaurants in (mountain view) that are (not very good)
- Adding the vectors:

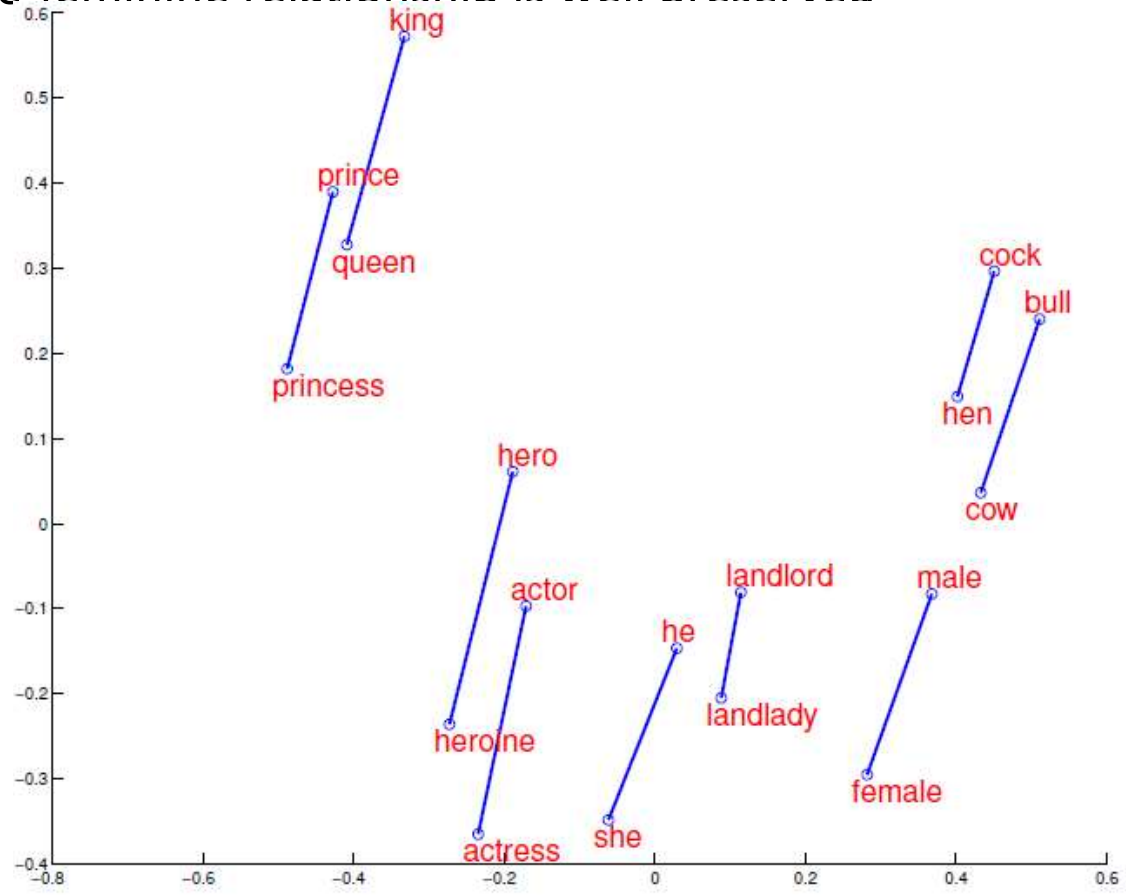
  restaurants + in + (mountain view) + that + are + (not very good)
- Very simple and efficient
- Unfortunately, does not work well for long sentences or documents

Compositionality by Vector Addition

Expression	*Nearest tokens*
Czech + currency	koruna, Czech crown, Polish zloty, CTK
Vietnam + capital	Hanoi, Ho Chi Minh City, Viet Nam, Vietnamese
German + airlines	airline Lufthansa, carrier Lufthansa, flag carrier Lufthansa
Russian + river	Moscow, Volga River, upriver, Russia
French + actress	Juliette Binoche, Vanessa Paradis, Charlotte Gainsbourg

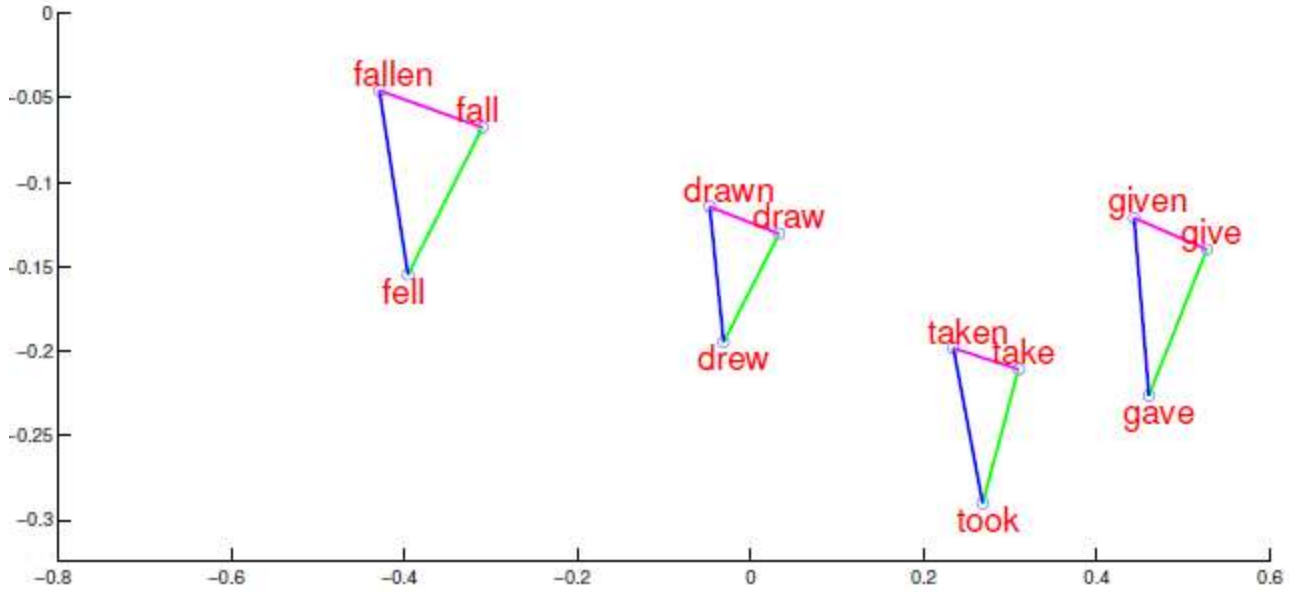
Visualization of Regularities in Word Vector Space

- We can visualize the word vectors by projecting them to 2D space
- PCA can be used for dimensionality reduction
- Although a lot of information is lost, the regular structure is often visible
- Masculine-feminine relationship is well preserved



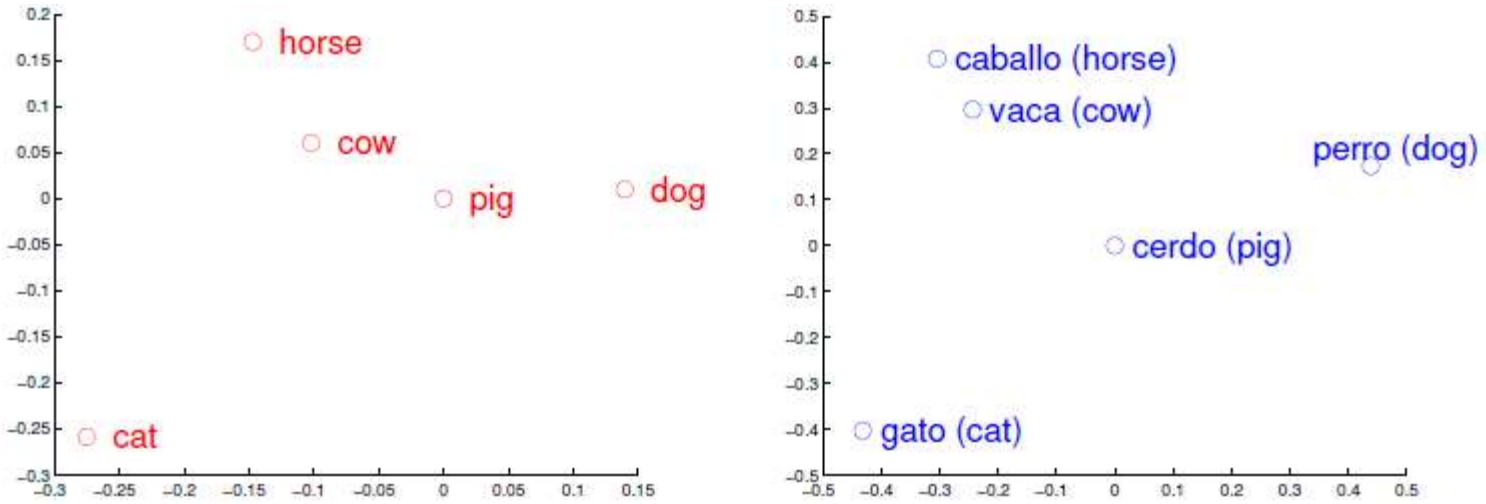
Visualization of Regularity in Word Vector Spaces

- Relationships between verb tenses are well maintained.



Machine Translation

- Word vectors should have similar structure when trained on comparable corpora
- This should hold even for corpora in different languages

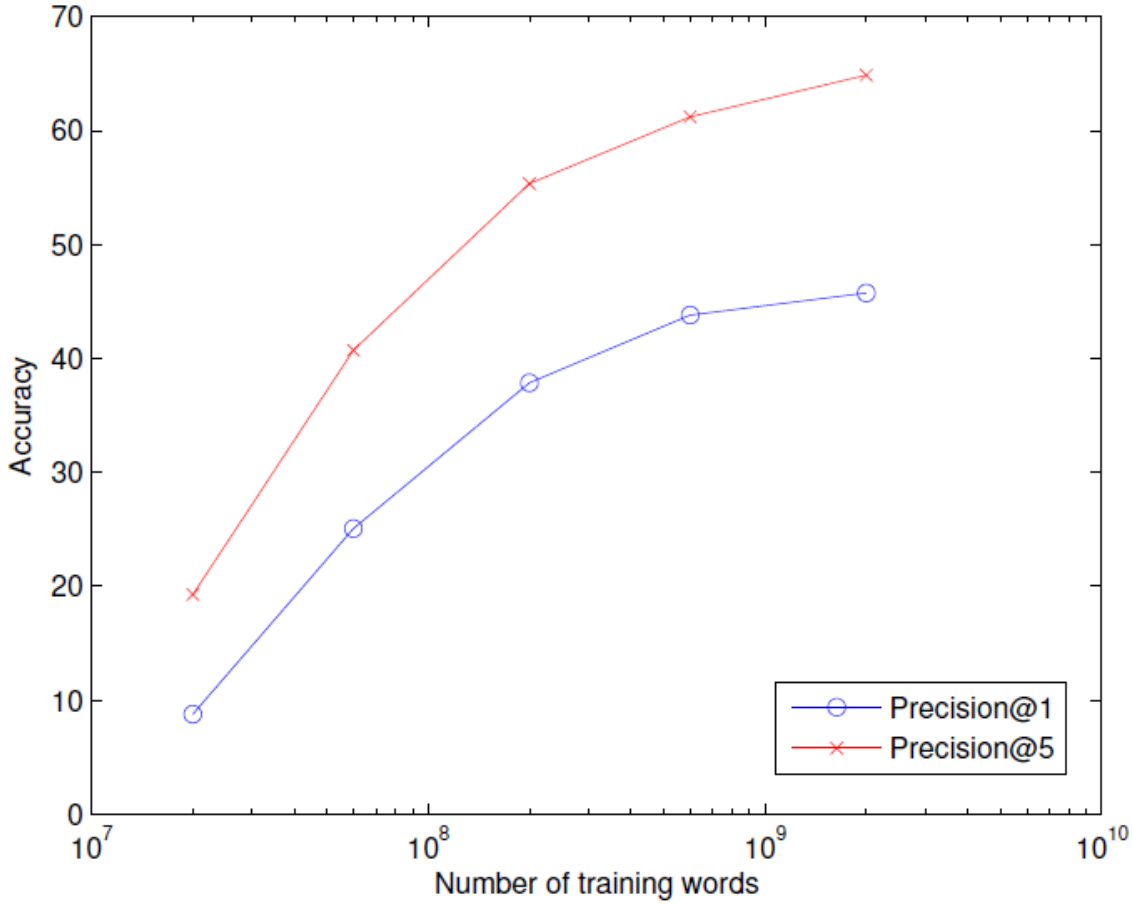


- The figures were manually rotated and scaled.

- For translation from one vector space to another, we need to learn a linear projection (will perform rotation and scaling)
- Small starting dictionary can be used to train the linear projection
- Then, we can translate any word that was seen in the monolingual data

Accuracy of English to Spanish Translation

- When applied to English to Spanish word translation, the accuracy is above 90% for the most confident translations.
- Can work for any language pair (T. Mikolov tried English to Vietnamese)
- More details in paper: "Exploiting similarities among languages for machine translation"



Learning Word Vectors

- Tomas Mikolov et al. weren't the first to use continuous vector representations of words, but they did show how to reduce the computational complexity of learning such representations – making it practical to learn high dimensional word vectors on a large amount of data.
- For example, "We have used a Google News corpus for training the word vectors. This corpus contains about 6B tokens. We have restricted the vocabulary size to the 1 million most frequent words…"
- The complexity in neural network language models (feedforward or recurrent) comes from the non-linear hidden layer(s).
- While this is what makes neural networks so attractive, Mikolov et al. decided to explore simpler models that might not be able to represent the data as precisely as neural networks, but could be trained efficiently on much more data.
- Google's team introduced 2 architectures for calculation of word vectors:
  - *Continuous Bag-of-Words* model, and a
  - *Continuous Skip-gram* model

Continuous Bag-of-Words (CBOW) Model

- Consider a piece of text such as “The recently introduced continuous Skip-gram model is an efficient method for learning high-quality distributed vector representations that capture a large number of precise syntactic and semantic word relationships.”
- Imagine a sliding window over the text, that includes the central word currently in focus, together with the four words that precede it, and four words that follow it:

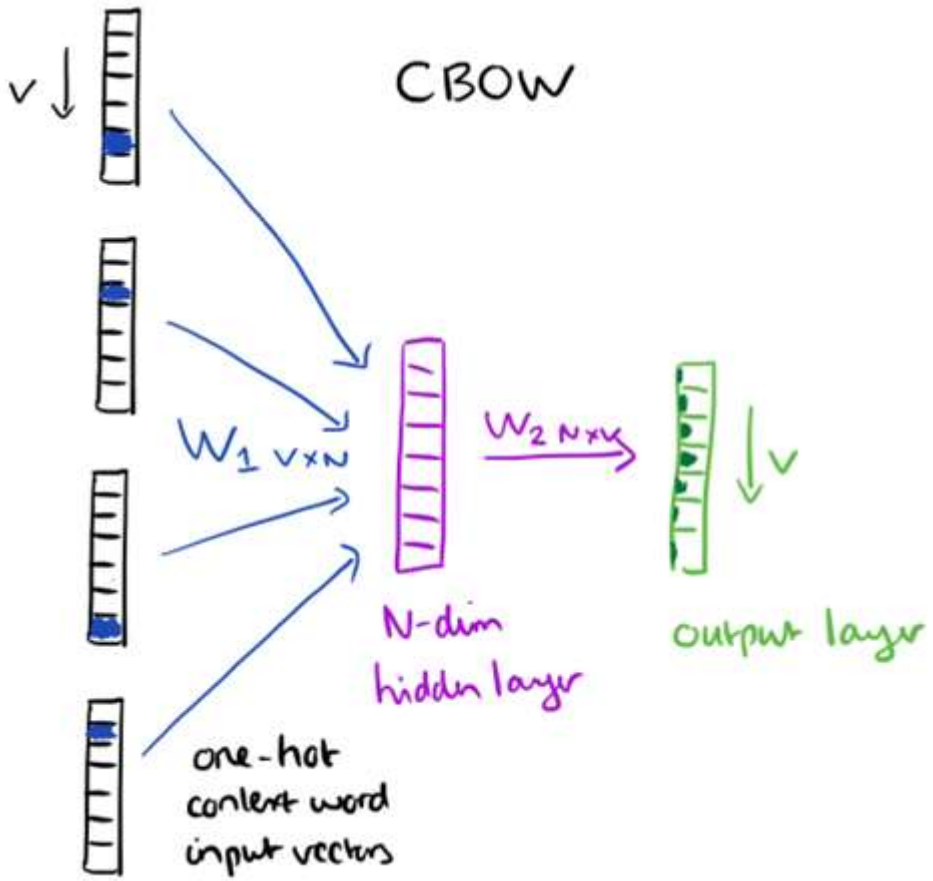
an efficient method for learning high-quality distributed vector

-------- context --------- learning -------- context ----------

- The context words form the input layer. Each word is encoded in one-hot form, so if the vocabulary size is *V* these will be V-dimensional vectors with just one of the elements set to one, and the rest all zeros. There is a single hidden layer and an output layer.

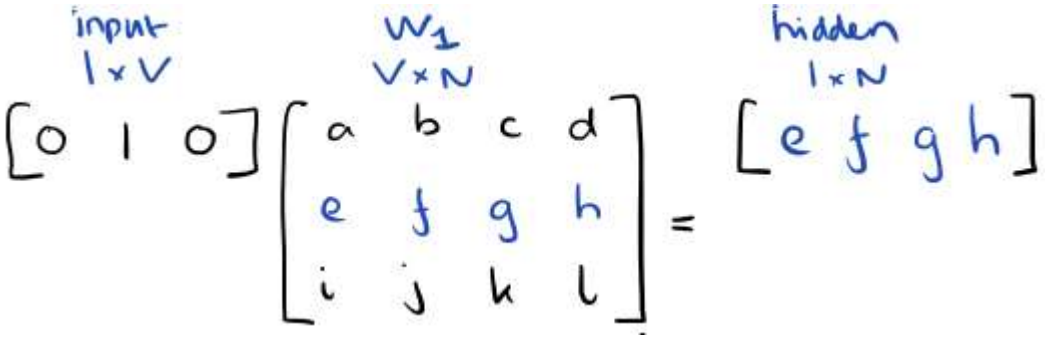
CBOW

- The training objective is to maximize the conditional probability of observing the actual output word (the focus word) given the input context words, with regard to the weights.
- In our example, given the input (“an”, “efficient”, “method”, “for”, “high”, “quality”, “distributed”, “vector”) we want to maximize the probability of getting “learning” as the output.



Structure of CBOW Layers

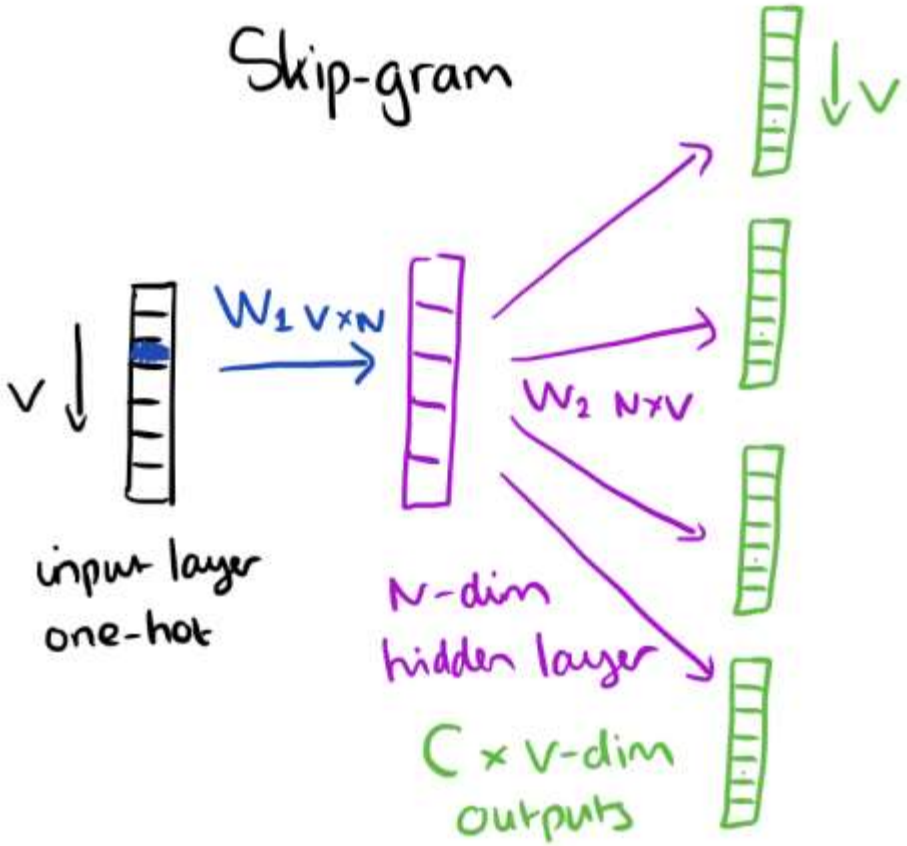
- Since our input vectors are one-hot, multiplying an input vector by the weight matrix $\mathbf{W_1}$ amounts to simply selecting a row from $\mathbf{W_1}$.



- Given *C* input word vectors, the activation function for the hidden layer **h** amounts to simply summing the corresponding 'hot' rows in W_1, and dividing by C to take their average.
- *This implies that the link (activation) function of the hidden layer units is simply linear (i.e., directly passing its weighted sum of inputs to the next layer).*
- From the hidden layer to the output layer, the second weight matrix $\mathbf{W_2}$can be used to compute a score for each word in the vocabulary, and softmax can be used to obtain the posterior distribution of words.

Skip-gram Model

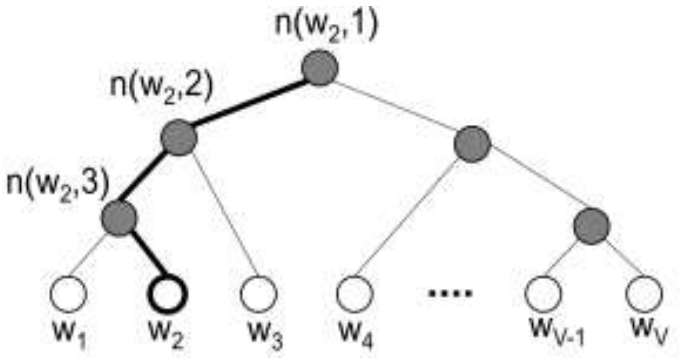
- The **skip-gram** model is the opposite of the CBOW model. It is constructed with the focus word as the single input vector, and the target context words are now at the output layer.
  - The activation function for the hidden layer simply amounts to copying the corresponding row from the weights matrix $\mathbf{W_1}$ (linear) as we saw before.
  - At the output layer, we now output *C* multinomial distributions instead of just one.
  - The training objective is to minimize the summed prediction error across all context words in the output layer. In our example, the input would be “learning”, and we hope to see (“an”, “efficient”, “method”, “for”, “high”, “quality”, “distributed”, “vector”) at the output layer



Optimization of the Learning Process

- Having to update every output word vector for every word in a training instance is very expensive….
- *To solve this problem, an intuition is to limit the number of output vectors that must be updated per training instance. One elegant approach to achieving this is hierarchical softmax; another approach is through sampling.*
- Hierarchical softmax uses a binary tree to represent all words in the vocabulary. The words themselves are leaves in the tree. For each leaf, there exists a unique path from the root to the leaf, and this path is used to estimate the probability of the word represented by the leaf. "We define this probability as the probability of a r the root ending at the leaf in question."

- An example binary tree for the hierarchical softmax model. White units are words in the vocabulary.
- The main advantage is that instead of evaluating V output nodes in the neural network to obtain the probability distribution, it is needed to evaluate only about $\log_2(V)$ words…
- Mikolov et al. use a binary Huffman tree, as it assigns short codes to the frequent words which results in fast training.



Negative Sampling

- Negative Sampling is simply the idea that we only update a sample of output words per iteration. The target output word should be kept in the sample and gets updated, and we add to this a few (non-target) words as negative samples. “A probabilistic distribution is needed for the sampling process, and it can be arbitrarily chosen… One can determine a good distribution empirically.”
- Milokov et al. also use a simple subsampling approach to counter the imbalance between rare and frequent words in the training set (for example, “in”, “the”, and “a” provide less information value than rare words).
- Each word in the training set is discarded with probability $P(w_i)$ where

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

$f(w_i)$ is the frequency of word w_i and t is a chosen threshold, typically around 10^{-5}

Installing Word2Vec

- Original Google site resides at https://code.google.com/archive/p/word2vec/
- The site tells you that you need the Subversion, version control system. Go to:
- https://subversion.apache.org/ and select binary package for your operating system. Once Subversion is installed, run the `checkout` command:

```
$ svn checkout http://word2vec.googlecode.com/svn/trunk/
svn: E170013: Unable to connect to a repository at URL
```

- This is a bad sign, suggesting that Google is not paying much attention to this technology any more. It has been forgotten once it was open-sourced.
- However, the code could be downloaded from the archive:

https://code.google.com/archive/p/word2vec/source/default/source

- Once you expand the archive, you will see a bunch of C language source files: `word2vec.c, word2phrase.c, word-analogy.co, distance.c and compute-accuracy.c` and a bunch of demo `sh` scripts that will run various examples.
- C code needs to be compiled before you could us it. I tried using Windows' Visual Studio `nmake` to compile the code but gave up and used `gcc` compiler and the `make` that I installed with Cygwin. Your Cygwin will also need `wget`.
- On the command prompt, in the directory with C files and the `makefile`, type:

```
$ make
```

- There were a few warnings. Hopefully nothing important.

demo-word.sh, Test Installation

- The script `demo-word.sh` downloads a small (97.7 MB) text corpus from the web, and trains a small word vector model. After the training is finished, the user can interactively explore the similarity of the word.

```
make
if [ ! -e text8 ]; then
  wget http://mattmahoney.net/dc/text8.zip -O text8.gz
  gzip -d text8.gz -f
fi
time ./word2vec -train text8 -output vectors.bin -cbow 1 -size 200 -window 8 -
negative 25 -hs 0 -sample 1e-4 -threads 20 -binary 1 -iter 15
./distance vectors.bin
```

- The script is testing whether file `text8` exists locally. If not, it downloads it using `wget` utility from http://mattmahoney.net/dc/text8.zip as a `gz` archive. Subsequently, script expands `gz` archive into file `text8`.
- Then `demo-word.sh` calls `word2vec` executable to train CBOW model with dimensionality of 200, the size of the context window of 8, using negative sampling algorithm with threashold of `E-4`. Result of the analysis, i.e word vectors, are placed in file `vectors.bin`. Options for cbow parameter are `1 (cbow)` and `0 (skip-gram)`.
- Creation of word vectors is called “word embedding”
- Finally, script invokes executable `distance` on file `vectors.bin`

demo-word.sh

```
$ ./demo-word.sh
Starting training using file text8
Vocab size: 71291
Words in train file: 16718843
Alpha: 0.000005  Progress: 100.10%  Words/thread/sec: 84.13k
real    6m38.680s
user    49m52.500s
sys     0m2.686s
Enter word or sentence (EXIT to break): day in berkshire  # in executable distance.exe
Word: day  Position in vocabulary: 137
Word: in  Position in vocabulary: 5
Word: berkshire  Position in vocabulary: 26294
                                              Word       Cosine distance
------------------------------------------------------------------------
                                           england              0.472332
                                      thanksgiving              0.439126
                                            monday              0.424840
                                       christchurch             0.420473
                                         yorkshire              0.418178
                                        eastbourne              0.404624
                                       huddersfield             0.404338
                                      staffordshire             0.397926
                                   gloucestershire              0.394107
                                              fife              0.393847
                                           hogmanay             0.392669
```

distance.exe, What it does

- The **word2vec** tool takes a text corpus as input and produces the word vectors as output. It first constructs a vocabulary from the training text data and then learns vector representation of words.
- The resulting word vector file can be used as features in many natural language processing and machine learning applications.
- A simple way to investigate the learned representations is to find the closest words for a user-specified word. The **distance** tool serves that purpose. For example, if you enter 'france', **distance** will display the most similar words and their distances to 'france', which should look like:

```
$ ./distance.exe vectors.bin
Enter word or sentence (EXIT to break): france

Word: france  Position in vocabulary: 303

                                              Word       Cosine distance
------------------------------------------------------------------------
                                             spain              0.673869
                                             italy              0.637653
                                            french              0.604296
                                       netherlands              0.601854
                                           commune              0.584286
                                           belgium              0.573884
                                           germany              0.565837
                                          provence              0.558092
```

word-analogy.sh

- We stated previously that the word vectors capture many linguistic regularities, for example vector operations `vector('Paris') - vector('France') + vector('Italy')` results in a vector that is very close to `vector('Rome')`, and `vector('king') - vector('man') + vector('woman')` is close to `vector('queen')`.
- We could try out a simple demo by running `word-analogy.exe`. We will use a bigger word vector file trained on 3 billion word collection of text with 3 million words in vocabulary and word space of dimension 300. The file could be find here: https://github.com/mmihaltz/word2vec-GoogleNews-vectors. Type:

```
$ ./word-analogy.exe GoogleNews-vectors-negative300.bin
Enter three words (EXIT to break): king man woman
Word: king  Position in vocabulary: 187
Word: man  Position in vocabulary: 243
Word: woman  Position in vocabulary: 1012
                                                  Word              Distance
------------------------------------------------------------------------
                                                 girl              0.529294
                                               loving              0.453044
                                                 sexy              0.432399
                                            tenderness             0.426713
                                               femme               0.424321
                                               wonder              0.423229
```

- It appears that the age of kings and queens has passed. We did not find the `queen` we expected.

word-analogy.exe

- Let us try one more analogy:

```
Enter three words (EXIT to break): france paris russia
Word: france  Position in vocabulary: 225534
Word: paris  Position in vocabulary: 198365
Word: russia  Position in vocabulary: 294451
```

Word	Distance
north_korea	0.471760
tom_cruise	0.449580
lohan	0.448753
joel	0.445634
lindsay_lohan	0.445479
heidi	0.440514
megan_fox	0.438607
britney	0.429737
russians	0.429315
moscow	**0.428812**
natalie_portman	0.426762

- Moscow is found, indeed, but much behind more important notions and subjects like Tom Cruise, Lindsay Lohan and Megan Fox. This vocabulary was built reading Google News and this results clearly tell you that reality shows and movies are much more important that the classical geopolitics.
- You can even say that reality shows are shaping geopolitics.
- To observe strong regularities in the word vector space, it is needed to train the models on large data set, with sufficient vector dimensionality. Using the **word2vec** tool, it is possible to train models on huge data sets (up to hundreds of billions of words). This is not sufficient evidence to link Lindsay Lohan to V. Putin.

demo-phrases.exe

- In certain applications, it is useful to have vector representation of larger pieces of text. For example, it is desirable to have only one vector for representing `'san francisco'`. This can be achieved by pre-processing the training data set to form the phrases using the `word2phrase` tool, as is shown in the example script `./demo-phrases.sh`.
- The script apparently creates a new word vector dictionary which includes phrases

```
# make
if [ ! -e news.2012.en.shuffled ]; then
    wget http://www.statmt.org/wmt14/training-monolingual-news-
crawl/news.2012.en.shuffled.gz
    gzip -d news.2012.en.shuffled.gz -f
fi
sed -e "s/’/'/g" -e "s/’/'/g" -e "s/''/ /g" < news.2012.en.shuffled | \
   tr -c "A-Za-z'_ \n" " " > news.2012.en.shuffled-norm0
time ./word2phrase -train news.2012.en.shuffled-norm0 -output \
         news.2012.en.shuffled-norm0-phrase0 -threshold 200 -debug 2
time ./word2phrase -train news.2012.en.shuffled-norm0-phrase0 -output  \
         news.2012.en.shuffled-norm0-phrase1 -threshold 100 -debug 2
tr A-Z a-z < news.2012.en.shuffled-norm0-phrase1 > news.2012.en.shuffled-norm1-phrase1
time ./word2vec -train news.2012.en.shuffled-norm1-phrase1 -output \
   vectors-phrase.bin -cbow 1 -size 200 -window 10 -negative 25 -hs 0 \
   -sample 1e-5 -threads 20 -binary 1 -iter 15
./distance vectors-phrase.bin
```

demo-phrases.sh, takes a long time

```
$ ./demo-phrases.sh
gcc word-analogy.c -o word-analogy -lm -pthread -O3 -march=native -Wall -funroll-loops -Wno-unused-result
word-analogy.c: In function ‘main’:
word-analogy.c:20:8: warning: unused variable ‘ch’ [-Wunused-variable]
   char ch;
        ^
--2017-04-22 14:19:17--  http://www.statmt.org/wmt14/training-monolingual-news-
crawl/news.2012.en.shuffled.gz
Resolving www.statmt.org... 129.215.197.100
Connecting to www.statmt.org|129.215.197.100|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 786717767 (750M) [application/x-gzip]
Saving to: ‘news.2012.en.shuffled.gz’
news.2012.en.shuffled.gz      100%[===================================>] 750.27M  4.10MB/s    in 3m 49s
2017-04-22 14:23:07 (3.27 MB/s) - ‘news.2012.en.shuffled.gz’ saved [786717767/786717767]
Starting training using file news.2012.en.shuffled-norm0
Words processed: 296900K     Vocab size: 33198K
Vocab size (unigrams + bigrams): 18838711
Words in train file: 296901342
Words written: 296900K
real    23m3.658s
user    16m16.781s
sys     6m45.405s
Starting training using file news.2012.en.shuffled-norm0-phrase0
Words processed: 280500K     Vocab size: 38761K
Vocab size (unigrams + bigrams): 21728781
Words in train file: 280513979
Words written: 280500K
real    25m51.083s
user    16m29.250s
sys     9m20.358s
Starting training using file news.2012.en.shuffled-norm1-phrase1
Vocab size: 681320
Words in train file: 283545447
Alpha: 0.002176  Progress: 95.65%  Words/thread/sec: 113.92k
```

- One has an impression that this process indexes (embeds) not only individual words but bigrams as well.

Once finished, Test

```
Alpha: 0.000005  Progress: 100.00%  Words/thread/sec: 113.89k
real    84m30.377s
user    625m16.890s
sys     0m46.625s
Enter word or sentence (EXIT to break): san_francisco
Word: san_francisco  Position in vocabulary: 1908
```

Word	Cosine distance
bay_area	0.768280
seattle	0.739046
los_angeles	0.737351
san_francisco's	0.719936
san_diego	0.716911
san_jose	0.652228
northern_california	0.651532
redwood_city	0.645263
new_york	0.642389
oakland	0.639925

Stop words are not thrown out

- If you enter any “stop word”, you get something:

```
$ ./distance vectors-phrase.bin
Enter word or sentence (EXIT to break): the

Word: the  Position in vocabulary: 1

                                              Word       Cosine distance
------------------------------------------------------------------------
                                                in              0.745736
                                                of              0.735179
                                             which              0.725551
                                              from              0.629848
                                            within              0.626430
                                                by              0.611502
                                                 a              0.606961
                                                on              0.603177
                                              over              0.598575
                                               its              0.589629
                                               for              0.584841
                                              also              0.572342
                                               and              0.562596
                                               has              0.555843
                                                as              0.555592
                                                to              0.551339
                                              only              0.543157
                                             under              0.531530
```

./distance.exe vector-phrases.bin

- The example output with the closest tokens to **'san_francisco'** should looks like:

```
$ ./distance.exe vector-phrases.bin
Word Cosine distance
los_angeles 0.666175
golden_gate 0.571522
oakland 0.557521
california 0.554623
san_diego 0.534939
pasadena 0.519115
seattle 0.512098
taiko 0.507570
houston 0.499762
chicago_illinois 0.491598
```

- The linearity of the vector operations seems to weakly hold also for the addition of several vectors, so it is possible to add several word or phrase vectors to form representation of short sentences

Pre-trained word and phrase vectors

- Google published pre-trained vectors trained on part of Google News dataset (about 100 billion words). The model contains 300-dimensional vectors for 3 million words and phrases. The phrases were obtained using a simple data-driven approach described in [2]. The archive is available here: `GoogleNews-vectors-negative300.bin.gz`.
- An example output of `./distance GoogleNews-vectors-negative300.bin`:

```
Enter word or sentence (EXIT to break): Chinese river
Word Cosine distance
Yangtze_River 0.667376
Yangtze 0.644091
Qiantang_River 0.632979
Yangtze_tributary 0.623527
Xiangjiang_River 0.615482
Huangpu_River 0.604726
Hanjiang_River 0.598110
Yangtze_river 0.597621
Hongze_Lake 0.594108
Yangtse 0.593442
```

- The above example will average vectors for words 'Chinese' and 'river' and will return the closest neighbors to the resulting vector. More examples that demonstrate results of vector addition are presented in [2]. Note that more precise and disambiguated entity vectors can be found in the dataset that uses Freebase naming.
- [2] *Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.*

Word Clustering

- The word vectors can be also used for deriving word classes from huge data sets. This is achieved by performing K-means clustering on top of the word vectors.
- The script that demonstrates this is `./demo-classes.sh`.
- The output is a vocabulary file with words and their corresponding class IDs, such as:

```
carnivores 234
carnivorous 234
cetaceans 234
cormorant 234
coyotes 234
crocodile 234
crocodiles 234
crustaceans 234
cultivated 234
danios 234
. . .
acceptance 412
argue 412
argues 412
arguing 412
argument 412
arguments 412
belief 412
believe 412
challenge 412
claim 412
```

Performance

- The training speed can be significantly improved by using parallel training on multiple-CPU machine (use the switch '-threads N').
- The hyper-parameter choice is crucial for performance (both speed and accuracy), however varies for different applications. The main choices to make are:
  - architecture: skip-gram (slower, better for infrequent words) vs CBOW (fast)
  - the training algorithm: hierarchical softmax (better for infrequent words) vs negative sampling (better for frequent words, better with low dimensional vectors)
  - sub-sampling of frequent words: can improve both accuracy and speed for large data sets (useful values are in range 1e-3 to 1e-5)
  - dimensionality of the word vectors: usually more is better, but not always
  - context (window) size: for skip-gram usually around 10, for CBOW around 5

How to measure quality of the word vectors

- Several factors influence the quality of the word vectors: * amount and quality of the training data * size of the vectors * training algorithm
- The quality of the vectors is crucial for any application. However, exploration of different hyper-parameter settings for complex tasks might be too time demanding. Thus, we designed simple test sets that can be used to quickly evaluate the word vector quality.
- For the word relation test set described in [1], see **./demo-word-accuracy.sh**, for the phrase relation test set described in [2], see **./demo-phrase-accuracy.sh**. Note that the accuracy depends heavily on the amount of the training data; our best results for both test sets are above 70% accuracy with coverage close to 100%.
- [1] *Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.*
- [2] *Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.*

Training, Embedding, Data

The quality of the word vectors increases significantly with amount of the training data.

For research purposes, you can consider using data sets that are available on-line:

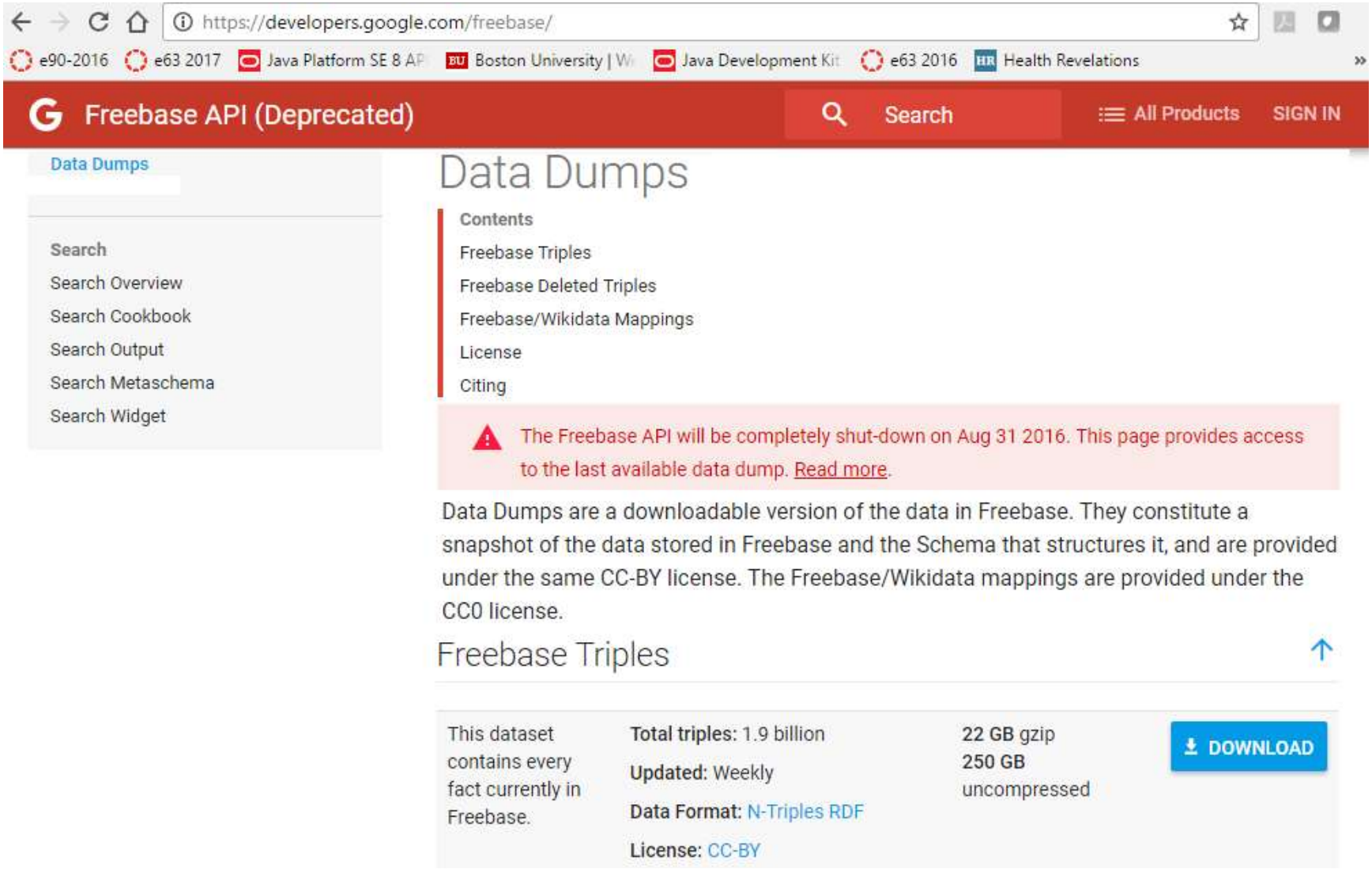
- First billion characters from wikipedia (use the pre-processing perl script from the bottom of Matt Mahoney's page)
- Latest Wikipedia dump Use the same script as above to obtain clean text. Should be more than 3 billion words.
- WMT11 site: text data for several languages (duplicate sentences should be removed before training the models)
- Dataset from "One Billion Word Language Modeling Benchmark" Almost 1B words, already pre-processed text.
- UMBC webbase corpus Around 3 billion words, more info here. Needs further processing (mainly tokenization).
- Text data from more languages can be obtained at statmt.org and in the Polyglot project.

Pre-trained entity vectors with Freebase naming

- Google is also offering more than 1.4M pre-trained entity vectors with naming from Freebase.
- This is especially helpful for projects related to knowledge mining.
- Entity vectors trained on 100B words from various news articles: freebase-vectors-skipgram1000.bin.gz
- Entity vectors trained on 100B words from various news articles, using the deprecated /en/ naming (more easily readable); the vectors are sorted by frequency: freebase-vectors-skipgram1000-en.bin.gz
- An example output of `./distance freebase-vectors-skipgram1000-en.bin`:

```
Enter word or sentence (EXIT to break): /en/geoffrey_hinton
Word Cosine distance
/en/marvin_minsky 0.457204
/en/paul_corkum 0.443342
/en/william_richard_peltier 0.432396
/en/brenda_milner 0.430886
/en/john_charles_polanyi 0.419538
/en/leslie_valiant 0.416399
/en/hava_siegelmann 0.411895
/en/hans_moravec 0.406726
/en/david_rumelhart 0.405275
/en/godel_prize 0.405176 ```
```

Freebase.com



Word2Vec in Python

- There appear to be several implementations of word2vec in Python.
- One implementation could be found at https://github.com/danielfrg/word2vec
- This is code written by Daniel Rodriquez. Code installs with `pip` utility:

```
$ pip install word2vec
```

- To train word data, i.e. create word vectors we need a file with space, tab and EOL separated words. If we expand previously downloaded file `text8.zip` into directory `Downloads` we will get file `text8`.
- We could run training process with the following commands:

```
>> %load_ext autoreload
>> %autoreload 2
>> import word2vec
>>  word2vec.word2vec('Downloads/text8','Downloads/text8.bin', tail size=100, verbose=True)
```

- After a few minutes `text8.bin` file with 100-dimensional vectors will be created. `text8.bin` is a binary file and is not readable.
- Generated word space representation of our vocabulary can be loaded into the memory with the command:

```
>> binary = word2vec.load('Downloads/text8.bin', kind='bin')
>> binary.vocab      # binary.vocab is a numpy array
array(['</s>', 'the', 'of', ..., 'koba', 'skirting', 'selectors'], dtype='<U78')
```

Transform to Text

- You can transform binary word vector file into a readable .txt file with command

```
>> text = word2vec.load('Downloads/text8.txt', kind='txt')
```

- And see its vocabulary with

```
>> text.vocab
array(['</s>', 'the', 'of', ..., 'koba', 'skirting', 'selectors'], dtype='<U78')
```

- You can see the actual coordinates of you vectors by doing head or tail from the command prompt:

```
$ tail text8.txt
skirting -0.018014 0.000921 -0.000209 0.003462 0.006218 -0.016255 -0.001128 -0.013593 -
0.010376 -0.011174 0.014719 0.001978 -0.012098 0.008415 -0.012701 0.008280 0.001361 -
0.003232 -0.000120 0.007181 -0.009247 -0.012890 -0.003461 -0.003768 0.021957 0.008478
0.001966 0.003011 0.001607 0.003924 0.001586 0.000059 -0.006806 0.001657 0.017555
0.013210 0.014977 -0.004028 -0.004557 -0.009443 -0.013524 -0.000591 0.005661 -0.001083
-0.006900 -0.012541 0.006005 -0.012217 0.019680 -0.000015 0.007053 -0.006741 -0.002487
0.001939 0.010716 -0.009277 0.029898 -0.013014 0.011603 0.013440 -0.001314 0.009390 -
0.000274 -0.012617 0.006387 0.021958 -0.003516 0.011438 0.003820 -0.011352 -0.013571
0.010020 0.000926 0.006217 0.013828 -0.003189 -0.026646 0.009708 -0.021804 0.002526
0.003636 -0.010022 -0.004468 0.001599 -0.000996 -0.004156 -0.019251 0.010191 -0.000180
0.017261 -0.018199 -0.001550 0.017765 -0.002069 0.004417 -0.012223 -0.006329 0.002732
0.008535 -0.003271
selectors -0.015117 0.006820 0.022063 -0.006469 0.005228 -0.001816 0.000132 0.018086 -
0.002442 0.000760 -0.011336 -0.008832 -0.006574 0.014347 0.008112 -0.005247 0.018120
0.001398 -0.001796 -0.004904 -0.021957 0.003600 -0.003638 -0.000655 0.002697 -0.010504
0.013091 0.003310 0.016441 -0.001614 -0.004995 0.001907 0.005268 0.000272 -0.014656
0.004964 0.013869 -0.014529 -0.016712 -0.004493 0.015696 -0.001692 0.013926 0.014830
0.003024 0.006204 -0.009322 0.002514 -0.000330 0.006635 0.005237 0.015940 -0.004621
0.007583 -0.001822 0.005997 0.002037 -0.010255 -0.013393 0.008003 -0.003274 0.009850
0.007925 0.000884 -0.008354 -0.011012 -0.001289 0.002487 -0.014844 0.008174 0.006669
0.000716 -0.016517 -0.010141 0.016993 -0.008798 0.015384 0.003189 -0.016148 -0.020089
0.014786 0.013022 -0.001284 -0.003411 0.005129 -0.016009 0.001533 -0.020321 -0.008155
0.006476 0.006971 -0.014797 -0.001563 -0.017390 -0.006061 -0.004409 -0.005829 -0.003257
0.004168 -0.004749
```

- The above are 2 vectors for the last two words in the vocabulary `skirting` and `selectors`

Create bigram phrases, and clusters

- To create bigram phrases we could use `word2vec.word2phrase()` method

```
>>  word2vec.word2phrase('Downloads/text8','Downloads/text8-phrases', verbose=True)
```

- This will create file `text8-phrases` that we can use as a better input for word2vec.
- Note that you could also skip this previous step and use the original data as input for word2vec.
- Train the model using the `word2phrase` output.

```
>>  word2vec.word2vec('Downloads/text8-phrases','Downloads/text8.bin',size=100, verbose=True)
b'Starting training using file Downloads/text8-
phrases\r\n'b'100K\r200K\r300K\r400K\r500K\r600K\r700K\r800K\r900K\r1000K\r1100K\r1200K\r1300
K\r1400K\r1500K\r1600K\r1700K\r1800K\r1900K\r2000K\r2100K\r2200K\r2300K\r2400K\r2500K\r2600K\
```

- That generated new `text8.bin` file containing the word vectors in a binary format.
- Do the clustering of the vectors based on the trained model.

```
>> word2vec.word2clusters('Downloads/text8','Downloads/text8-clusters.txt',100,verbose=False)
```

- That created file `text8-clusters.txt` with the cluster for every word in the vocabulary:

```
$ tail text8-clusters.txt
breen 81
contreras 55
sideboard 2
ngaio 12
angelman 31
renaldo 51
paavo 59
koba 31
skirting 91
selectors 47
```

Predictions

```
>> import word2vec
```

- Import the word2vec binary file created above

```
>> model = word2vec.load('Downloads/text8.bin')
```

- We can take a look at the vocabulary as a numpy array

```
>> model.vocab
```

- Or take a look at the whole matrix

```
>> model.vectors.shape
(98331, 100)
>> model.vectors
array([[-0.16299246, -0.1238264 , -0.11257625, ..., 0.08034179, -0.12345738, -
0.11461086], [-0.13303199, 0.02959067, -0.14775175, ..., -0.05474622, 0.13290612,
0.04335521], [ 0.05310937, 0.08245376, -0.05846839, ..., 0.10106612, 0.1371658 ,
0.04797614],
```

- We can retrieve the vectors of individual words

```
>> model['dog'].shape
(100,)
>> model['dog'][:10]
array([-0.0962669 , -0.03933436, 0.02695419, -0.2129041 , -0.04581217, -0.01077986, -
0.06551255, -0.21367645, 0.10262867, 0.07369183])
```

Predictions

- We can do simple queries to retrieve words similar to "socks" based on cosine similarity:

```
>> indexes, metrics = model.cosine('socks')indexes, metrics
(array([12054, 19940, 20496, 26617, 5749, 6055, 7242, 8472, 79225, 39642],
dtype=int64), array([ 0.74813582, 0.74571054, 0.73818754, 0.73195643, 0.7308529 ,
0.72381697, 0.72291172, 0.72244098, 0.72178341, 0.72154959]))
```

- This returned a tuple with 2 items:
  - numpy array with the indexes of the similar words in the vocabulary
  - numpy array with cosine similarity to each word
- Its possible to get the words of those indexes

```
>> model.vocab[indexes]
array(['haired', 'candy', 'hairy', 'bluebird', 'winged', 'pat_hingle',
'list_firstnode', 'flask', 'leopard', 'jacket'], dtype='<U78'). . .
```

- There is a helper function to create a combined response: a numpy [record array]

```
>> model.generate_response(indexes, metrics)
rec.array([('haired', 0.77271406), ('candy', 0.76402519), ('hairy', 0.76043951), ('bluebird',
0.75748381), ('winged', 0.75214367), ('pat_hingle', 0.75027125), ('list_firstnode',
0.74869653), ('flask', 0.7422336 ), ('leopard', 0.74166617), ('jacket', 0.73788907)],
dtype=[('word', '<U78'), ('metric', '<f8')])
```

- Is easy to make that numpy array a pure python response

```
>> model.generate_response(indexes, metrics).tolist()


[('haired', 0.7727140594897487), ('candy', 0.7640251928941824), ('hairy',
0.7604395098383087), ('bluebird', 0.7574838136196931), ('winged', 0.7521436707175413),
('pat_hingle', 0.7502712506932621), ('list_firstnode', 0.7486965338917364), ('flask',
0.7422336037434836), ('leopard', 0.7416661668935238), ('jacket', 0.7378890706885178)]
```

Phrases

- Since we trained the model with the output of word2phrase we can ask for similarity of "phrases"

```
>> indexes, metrics = model.cosine('los_angeles')
>> model.generate_response(indexes, metrics).tolist()
[('san_francisco', 0.8605505616714487),
('kansas_city', 0.8432142846793768),
 ('chicago', 0.8243280851057059),
('boston', 0.8193195994427888),
('detroit', 0.814421968950594),
('california', 0.802739016000159),
('cincinnati', 0.7985432923087663),
('cleveland', 0.7880383284377004),
 ('brooklyn', 0.7880002484825327),
 ('new_jersey', 0.7869173687547675)]
```

Analogies

- Its possible to do more complex queries like analogies such as: king - man + woman = queen This method returns the same as cosine the indexes of the words in the vocab and the metric

```
>> indexes,metrics=model.analogy(pos=['king','woman'],neg=['man'], n=10)
>> indexes, metrics
(array([ 648, 7530, 1145, 1335, 1087, 6756, 1006, 1770, 344, 3138],
dtype=int64), array([ 0.29872462, 0.2917552 , 0.28828292, 0.28677072,
0.28659093, 0.286144 , 0.28366104, 0.2810565 , 0.27920366, 0.27897164]))

>> model.generate_response(indexes, metrics).tolist()
[('emperor', 0.2987246166697374),
('empress', 0.2917551978662613),
('prince', 0.2882829150365351),
('wife', 0.2867707222238017),
('queen', 0.2865909335112263),
('roman_emperor', 0.2861440046814332),
('daughter', 0.2836610386140423 5),
('pope', 0.2810565032569355 4),
('son', 0.2792036589389082 5),
('monarch', 0.2789716380448376)]
```

Clusters

- Load clusters

```
>> clusters = word2vec.load_clusters('Downloads/text8-clusters.txt')
```

- We can see get the cluster number for individual words
- We can see get the cluster number for individual words

```
>> clusters[b'king']
```

- We can see get all the words grouped on an specific cluster

```
>> clusters.get_words_on_cluster(90)[:10]
array([b'had', b'became', b'took', b'came', b'died', b'wrote', b'went',
b'appeared', b'married', b'returned'], dtype=object)
```

- We can add the clusters to the word2vec model and generate a response that includes the clusters

```
>> model.clusters = clusters
>> indexes, metrics = model.analogy(pos=['paris','germany'],neg=['france'],n=10)
model.generate_response(indexes, metrics).tolist()
[(u'berlin', 0.32333651414395953, 20),
(u'munich', 0.28851564633559, 20),
(u'vienna', 0.2768927258877336, 12),
(u'leipzig', 0.2690537010929304, 91),
(u'moscow', 0.26531859560322785, 74),
(u'st_petersburg', 0.259534503067277, 61),
(u'prague', 0.25000637367753303, 72),
(u'dresden', 0.2495974800117785, 71),
(u'bonn', 0.24403155303236473, 8),
(u'frankfurt', 0.24199720792200027, 31)]
```

gensim Package

- Genism is another popular Python package for word2vec. If you have Anaconda3.5 installed, it is already there. Otherwise you would do:

```
$ (sudo) pip install –upgrade genism  # note - –upgrade
```

- Gensime could use `vectors.bin` or similar file with trained word vectors, so just copy it to the directory where you want to run a gensim script.
- To run a `king – man + woman` like query, write a script with this code:

```
# import modules & set up logging
import gensim, logging
# define logging level.  We only care about warnings
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
level=logging.WARNING)
# Import already existing vectors from previous example
model = gensim.models.KeyedVectors.load_word2vec_format('vectors.bin',
binary=True)
# Get word similarity using out of the box task
most_similar = model.most_similar(positive=['emperor', 'woman'],
negative=['man'])
print(most_similar)
```

```
[('empress', 0.6104894876480103), ('theodora', 0.5391228199005127),
('emperors', 0.5160053968429565), ('chlorus', 0.5084700584411621),
('saimei', 0.4915323853492737), ('montferrat', 0.47812795639038086),
('suiko', 0.4621468484401703), ('comnena', 0.4601095914840698),
('faustina', 0.4584055542945862), ('constantius', 0.4577965438365936)]
```

If you doubt technology

Female names returned by the previous query are all empresses or princesses (daughters of emperors and empresses):

- `Saimei` ((655–661) and `Ashikaga Okiko` were Japanese Empresses.
- `Faustina` was the Empress and wife of Roman Emperor Antoninus Pius.
- Anna `Comnena` (Komnena) was a Byzantine princes and a daughter of an Emperor and an Empress.

You can find all of this in Wikipedia. One presumes that Google's techies include Wikipedia into their corpuses ☺

GloVe: Global Vectors for Word Representation

- Pennington et al. argue that the online scanning approach used by word2vec is suboptimal since it doesn't fully exploit statistical information regarding word co-occurrences. They demonstrate a *Global Vectors* (GloVe) model which combines the benefits of the word2vec skip-gram model when it comes to word analogy tasks, with the benefits of matrix factorization methods that can exploit global statistical information.
- The GloVe model…
- *… produces a vector space with meaningful substructure, as evidenced by its performance of 75% on a recent word analogy task. It also outperforms related models on similarity tasks and named entity recognition.*
- The source code for the model, as well as trained word vectors can be found at **http://nlp.stanford.edu/projects/glove/**

References

- *[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.*
- *[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.*
- *[3] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In Proceedings of NAACL HLT, 2013.*
- *Tomas Mikolov, Quoc V. Le and Ilya Sutskever. Exploiting Similarities among Languages for Machine Translation. We show how the word vectors can be applied to machine translation. Code for improved version from Georgiana Dinu here.*
- *Word2vec in Python by Radim Rehurek in gensim (plus tutorial and demo that uses the above model trained on Google News).*
- *Word2vec in Java as part of the deeplearning4j project. Another Java version from Medallia here.*
- *Word2vec implementation in Spark MLlib.*
- *Comparison with traditional count-based vectors and cbow model trained on a different corpus by CIMEC UNITN.*