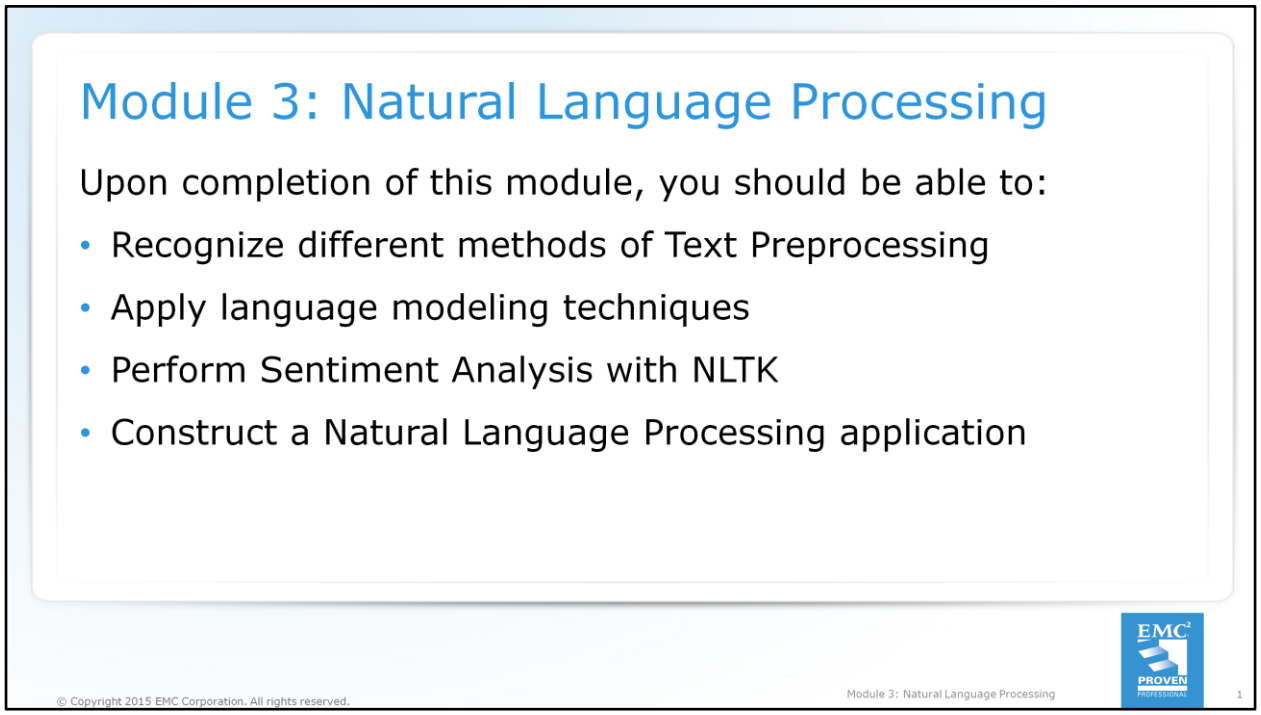
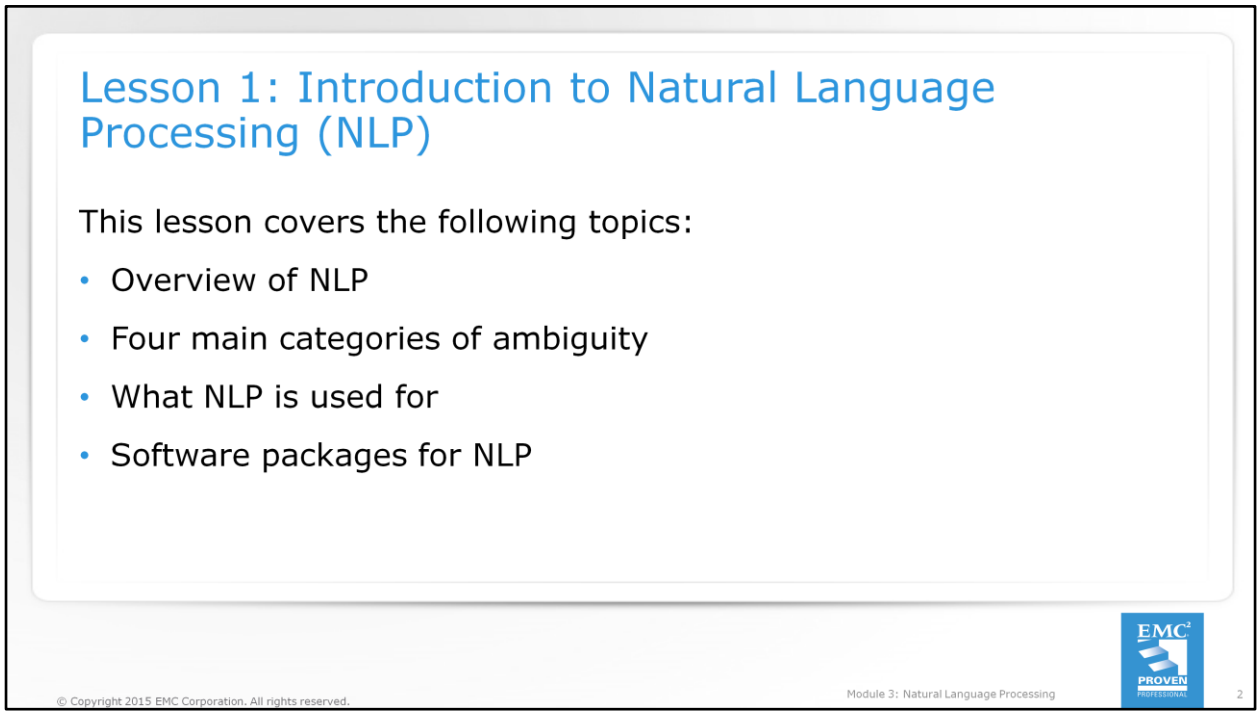


This module focuses on Natural Language Processing.



In this lesson, we will take a look at some basic techniques of NLP and various levels of text ambiguity.

Overview of NLP

- Interdisciplinary Field
- Goal: Enable computers to perform useful tasks involving human languages
- Example tasks:
  - Conduct sentiment analysis on the web, to gauge positive or negative feeling related to a specific topic
  - Machine transcription and translation of presidential speech
  - Summarize news articles of a certain topic
  - Automated scoring of student essays

Natural language processing is an interdisciplinary field. It sometimes bears another name, such as human language technology, speech and language processing, and computer linguistics. Natural language processing is related to many fields of study, including computer science, artificial intelligence (especially machine learning), linguistics, information retrieval, human-computer interaction, and even psychology.

The goal of natural language processing is to enable computers to perform useful tasks that involve human natural languages. It is concerned with all levels of interactions between humans, or humans and computers.

Some sample tasks of natural language processing include:

- Normalizing text with tokenization and stemming prior to the analysis
- Extracting useful information from unstructured text with disambiguation and Part-of-Speech (POS) tagging
- Accessing and processing a few popular linguistic data sets and corpora
- Conducting sentiment analysis on the web to gauge positive or negative feelings related to a specific topic
- Machine transcription and translation of presidential speech
- Understanding the meaning of the text including parsing and semantic analysis
- Summarizing news articles of a certain topic
- Automated scoring of student essays (in the SAT and GRE tests)

Next, we will show an example related to understanding the meaning of texts in order to illustrate the ambiguity issues that NLP has to face.

NLP Has to Deal with Ambiguity

"A computer that understands you like your mother."
-- 1985 McDonnell-Douglas ad

What does it mean?

- The computer understands you as well as your mother understands you.
- The computer understands that you like your mother.
- The computer understands you as well as it understands your mother.

(Lee 2004)

The 1985 McDonnel-Douglas Ad uses the slogan "A computer that understands you like your mother." Without any context such as what the ad is for and what McDonnel-Douglas is, this sentence can be interpreted in at least three different ways, as shown on the slide.

The ad is intended for the first interpretation, that the computer understands you as well as your mother understands you. Without any context, it would be very difficult for computers to guess what the sentence actually means.

Reference:

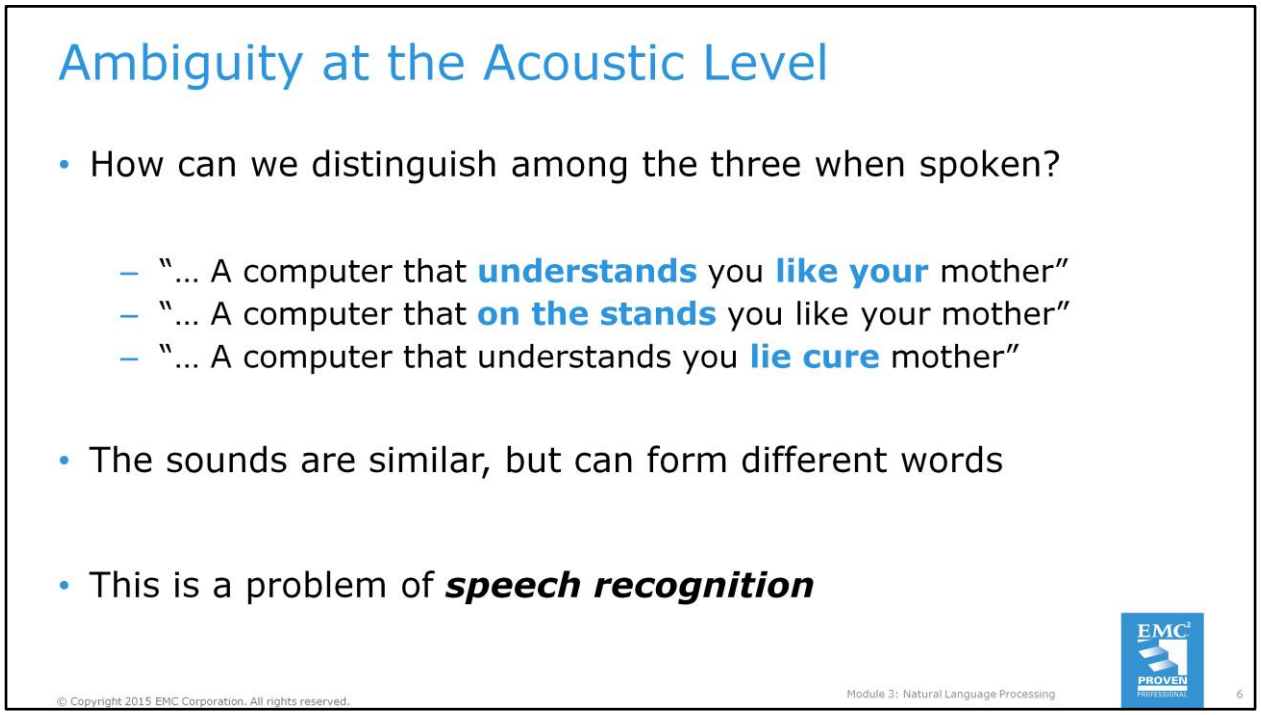
Lillian Lee (2004). "I'm sorry Dave, I'm afraid I can't do that": Linguistics, Statistics, and Natural Language Processing circa 2001. *Computer Science: Reflections on the Field, Reflections from the Field.* Report of the National Academies' study on the Fundamentals of Computer Science, pp. 111—118.
http://arxiv.org/pdf/cs/0304027.pdf

Four Main Categories of Ambiguity

- Four main categories of ambiguity
  - Acoustic ambiguity (Sound)
  - Semantic ambiguity (Meaning)
  - Syntactic ambiguity (Syntax)
  - Discourse ambiguity (Reference)
- There are established methods in NLP for identifying and resolving these kinds of ambiguity

The field of natural language processing often has to deal with ambiguity in human languages, regardless of which human language we are talking about. The ambiguity in NLP can be classified into four categories: acoustic ambiguity, semantic ambiguity, syntactic ambiguity, and discourse ambiguity.

Most of the techniques discussed in this module will help address some of these ambiguity issues.



At the acoustic level, words that have similar pronunciation may cause ambiguity. This is a common problem for automatic speech recognition (ASR) systems. In the example, the acoustic sound of "a computer that understands you like your mother" may be mistakenly transcribed by an ASR system as "a computer that on the stands you like your mother" or "a computer that understands you lie cure mother."

The following paper elaborates on some of the difficulties with ASR systems:

Markus, Forsberg (2003). "Why is speech recognition difficult." *Chalmers University of Technology.* http://www.speech.kth.se/~rolf/gslt_papers/MarkusForsberg.pdf

Ambiguity in ASR systems is on-going research. One approach to address the issue can be found in the following literature:

H. Sak, F. Beaufays, K. Nakajima, and C. Allauzen (2013). "Language model verbalization for automatic speech recognition." *In Proceedings of Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8262-8266. http://research.google.com/pubs/archive/41158.pdf

Ambiguity at the Semantic Level

- Semantic refers to the meaning of the words in a given context

moth·er \'mə-thər\ *n*
1, a female parent.
2, a woman in authority.
3, something that is an extreme or ultimate example of its kind especially in terms of scale.
4, a slimy membrane composed of yeast and bacterial cells that develops on the surface of alcoholic liquids undergoing acetous fermentation and is added to wine or cider to produce vinegar ---*adj.* of, relating to, or being a mother ---*v.* to give birth to; to care for or protect like a mother

- How do we know which meaning of "mother" it refers to in a given context?
- This is a problem of ***word sense disambiguation***

Many words have several, very different meanings. For example, a simple word like "mother" could mean a female parent or a slimy membrane composed of yeast and bacterial cells. How can an NLP system know the correct meaning when parsing text that contains the word mother?

Ambiguity at the semantic level is a common problem of an NLP subtask called Word Sense Disambiguation (WSD). We will discuss WSD later in this module.

Ambiguity at the Syntactic Level

- Syntactic ambiguity refers to a sentence that may be interpreted in multiple ways in its meaning
  - I was thinking of our cat **driving** home.

Photo:
http://www.stockvault.net/photo/132819/woman-driving-a-car

Photo:
http://catsdrivingthings.tumblr.com/

Syntactic ambiguity arises when the meaning of a sentence can be interpreted in multiple ways. In the example shown , syntactically speaking the sentence could be interpreted as "I am driving home" or "Our cat is driving home."

Ambiguity at the Syntactic Level (Cont.)

Photo Caption:

Candles lighted in memory of the Boston victims **burned** at Olympic Forest Park in Beijing on Tuesday.

- Candles … burned?
- Victims … burned?

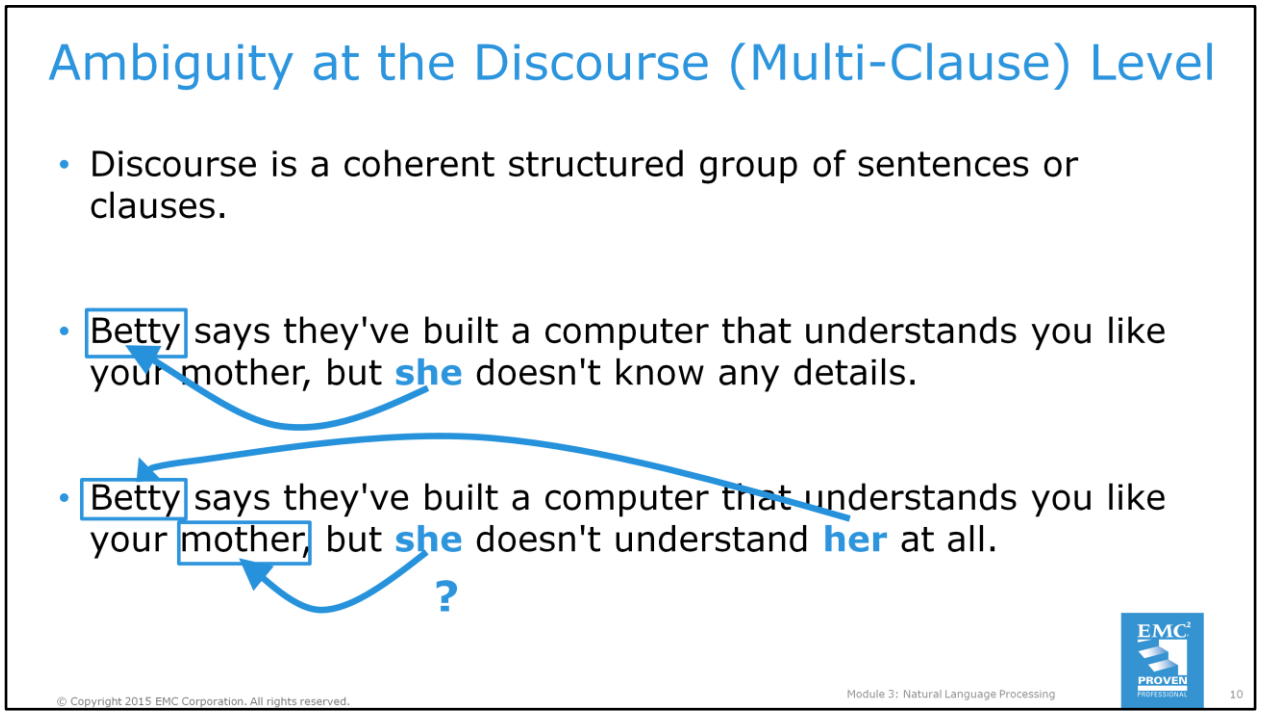
Ambiguity at the syntactic level is commonly seen in newspapers. For example, the sentence shown on the slide is taken from a New York Times article. Grammar-wise, this sentence can be interpreted in two different ways: "the candles burned" or "the victims burned."[1]

One approach to tackle this issue is to extract lists of lexical associations with prepositions.[2]

References:

[1]Chris Buckley (April 17, 2013). "Grad Student With Eye on Career in Finance Is Mourned in China." *nytimes.com.* http://www.nytimes.com/2013/04/18/world/asia/china-mourns-the-death-of-student-in-boston-blast.html

[2]Donald Hindle and Mats Rooth (March 1993). "Structural ambiguity and lexical relations." *Computational Linguistics*, 19, 1, pp. 103-120.



Discourse is a coherent structured group of textual units, such as sentences or clauses. Important tasks in processing a discourse include:

- Discourse segmentation
- Determining coherence relations
- Anaphora resolution

The example on the slide is an instance of anaphora, where "she" co-referees to some other discourse entity.

Human languages are spoken or written as a collection of sentences. Although discourse is a very challenging task, we cannot simply ignore it. If we treat a sentence in isolation, syntactically and/or semantically without considering discourse, the sentence would not be fully understood.

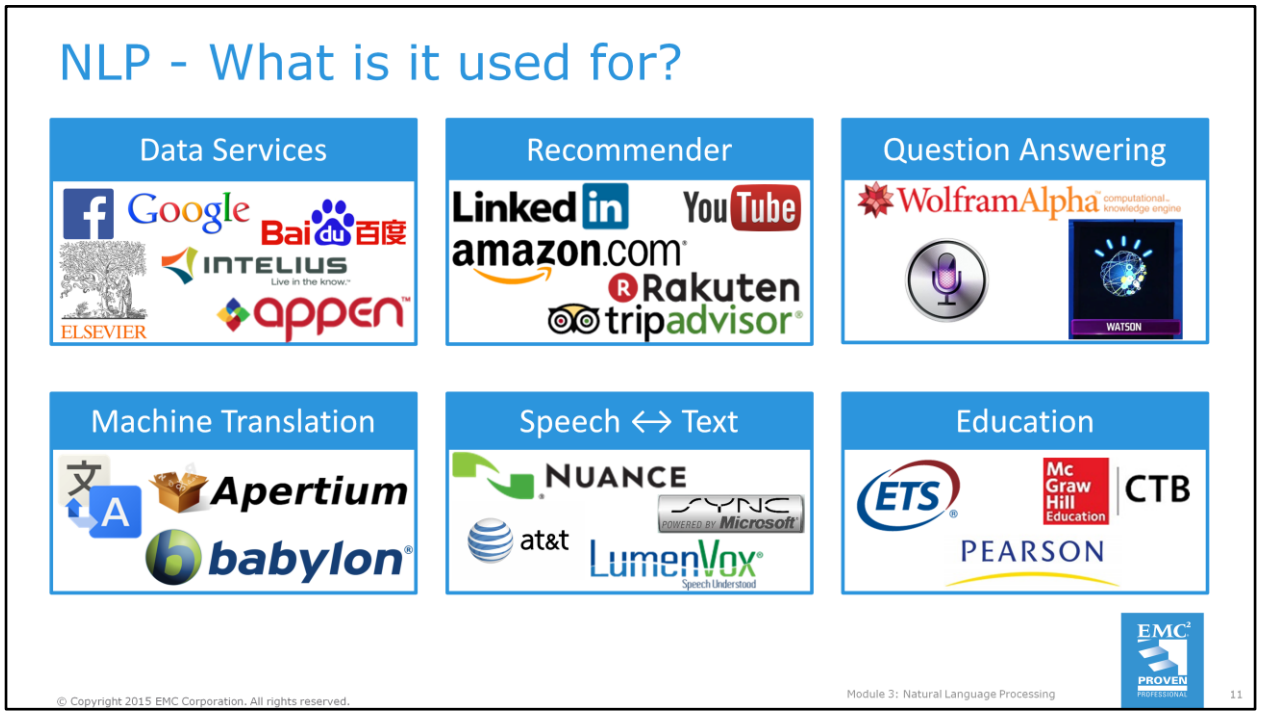
Consider the following example:

> Today was Jack's birthday. Penny and Janet went to the store. They were going to get presents. Janet decided to get a kite. "Don't do that," said Penny. "Jack has a kite. *He will make you take it back."*

An AI engine designed to understand stories can figure out facts such as the presents are for Jack, the kite is a present, etc. These problems can be partly resolved by storing information in a data structure called a birthday party object. This object includes information such as, at birthday parties people give presents to the person being honored, people generally buy presents at stores, etc.

This solves part of the problem, but what about the underlined "it" in the last sentence? According to the grammar, the AI engine can refer back to the last mentioned "kite" – the kite Jack owns. But we know this kite is not the one that Jack will make Janet take back. It will be the new kite that goes back. Any 4-year old will get this right. But such an observation is very difficult for the computer to obtain.

"Anaphora (linguistics)." *Wikipedia: the free encyclopedia.* October 2, 2014. http://en.wikipedia.org/wiki/Anaphora_(linguistics)



The slide shows some example applications where NLP is used.

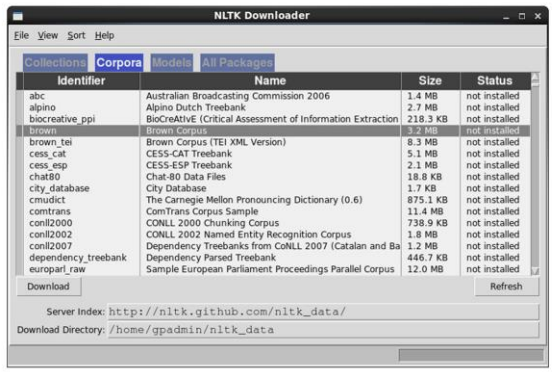
- Data services: Facebook, Google search, Google Ads, Baidu.com, Intelius's People Search http://www.intelius.com/people-search.html, Elsevier's Fingerprint Engine http://www.elsevier.com/online-tools/research-intelligence/products-and-services/elsevier-fingerprint-engine, and Appen http://www.appen.com/.
- Recommender: LinkedIn's "people you may know" widget, the "customers who bought this also bought that" feature provided by many online retailers such as Amazon and Rakuten, YouTube's "recommended channels" or TripAdvisor's "recommended trips."
- Question answering systems: WolframAlpha knowledge engine, Apple Siri, and IBM Watson
- Machine translation: Google Translate, Apertium http://www.apertium.org/ and Babylon http://translation.babylon.com/.
- Speech-to-text or text-to-speech: Nuance Dragon, Ford SYNC http://www.ford.com/technology/sync/, AT&T Speech API http://developer.att.com/apis/speech, and LumenVox http://www.lumenvox.com/.
- Education: ETS, Pearson, and McGrawHill Education. One example of using NLP is to design education applications that include spell correction, grammar checking, and auto-grading.

NLP is a broad field. In this course, we will only focus on text analysis.

Python NLTK

- A leading platform for building Python programs to work with human language data
- Website: http://www.nltk.org/

```
[user@machine ~]$ python
Python 2.6.6 (r266:84292, Jan 22
2014, 09:42:36)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-
4)] on linux2
Type "help", "copyright", "credits"
or "license" for more information.
>>> import nltk
>>> nltk.__version__
'2.0.4'
>>> nltk.download()
```



NLTK Downloader

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources, such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning. Readers can refer to the website http://www.nltk.org/ for resources such as tutorials, API documentation, release announcements, and downloads.

This module will use Python and NLTK to demonstrate most NLP tasks. We will be using NTLK because it is easy to use, especially if you have just started learning NLP. Programmers of other languages, other than Python, may alternatively use the following packages:

- OpenNLP (Java) https://opennlp.apache.org/
- StanfordNLP (Java) http://nlp.stanford.edu/
- LingPipe (Java) http://alias-i.com/lingpipe/
- UIMA (Java) http://uima.apache.org/
- Mallet (Java) http://mallet.cs.umass.edu/
- ScalaNLP (Scala) http://www.scalanlp.org/
- Haskell (Haskell, built-in) http://nlpwp.org/book
- NLP (R) http://cran.r-project.org/web/packages/NLP/
- tm (R) http://cran.r-project.org/web/packages/tm/

Note that Python and Java are the more popular choices for NLP. Both are efficient and both provide many existing NLP packages to choose from. **R is usually not a good choice when working with any non-trivially-sized linguistic data.** R holds all data in your active workspace in RAM. If you are running R on a 32-bit system, for example, you only have a 4 GB limit to the RAM R can access. There are two implications of this: (1) NLP data needs to be stored in memory-efficient objects, and (2) there is a hard limit on how much linguistic data you can work on at one time.

After NTLK is installed on your machine, enter `nltk.download()` in the Python prompt to launch the NLTK Downloader where you can install additional corpora from NLTK's repository.

Reference:

B. D. Ripley and D. J. Murdoch. "2.9 There seems to be a limit on the memory it uses!" *R for Windows FAQ: Version for R-3.1.2, Dec 12, 2012.* http://cran.r-project.org/bin/windows/base/rw-FAQ.html#There-seems-to-be-a-limit-on-the-memory-it-uses_0021

Check Your Knowledge

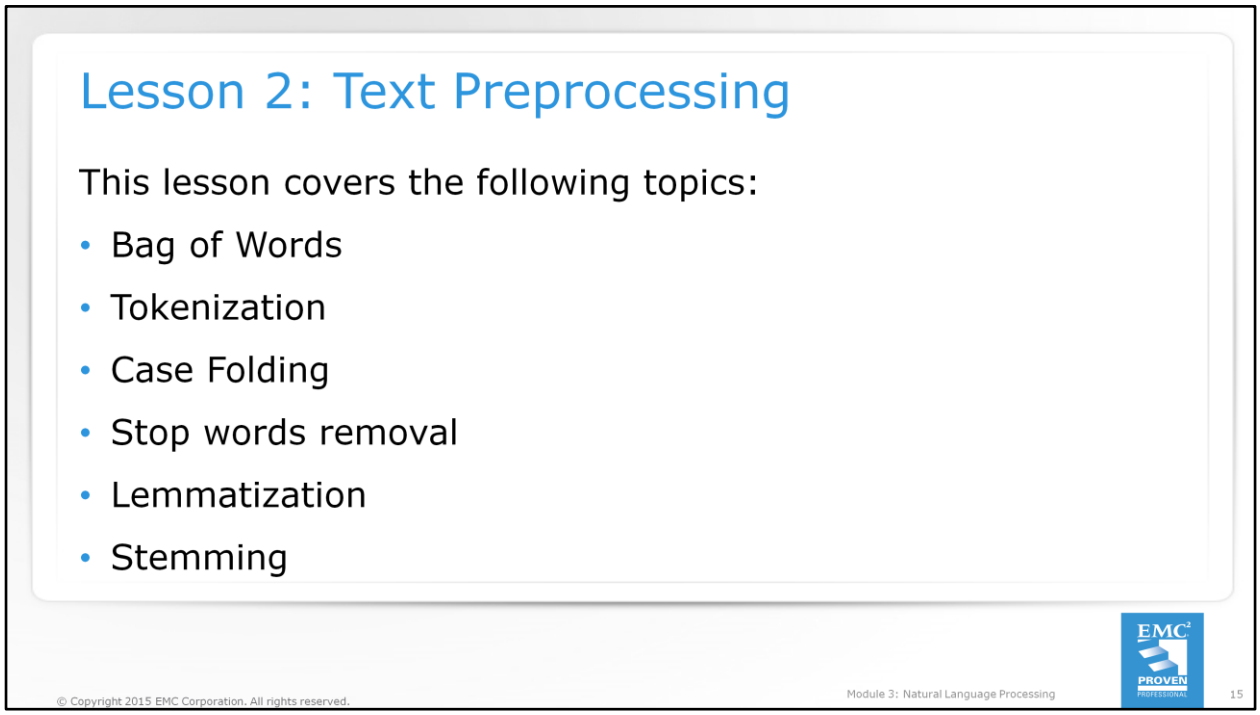
- What is NLP?
- What ambiguity issues does NLP deal with?
- Which NLP task addresses the semantic level of ambiguity?
- What is NLP used for?
- What is Python NLTK?

Write your answers here.

Lesson 1: Summary

During this lesson the following topics were covered:

- Overview of NLP
- Four main categories of ambiguity
- What NLP is used for
- Software packages for NLP



In this lesson, we will take a look at different methods of text preprocessing, such as Bag of Words, Tokenization, Case Folding, Stop Words Removal, Lemmatization, and Stemming.

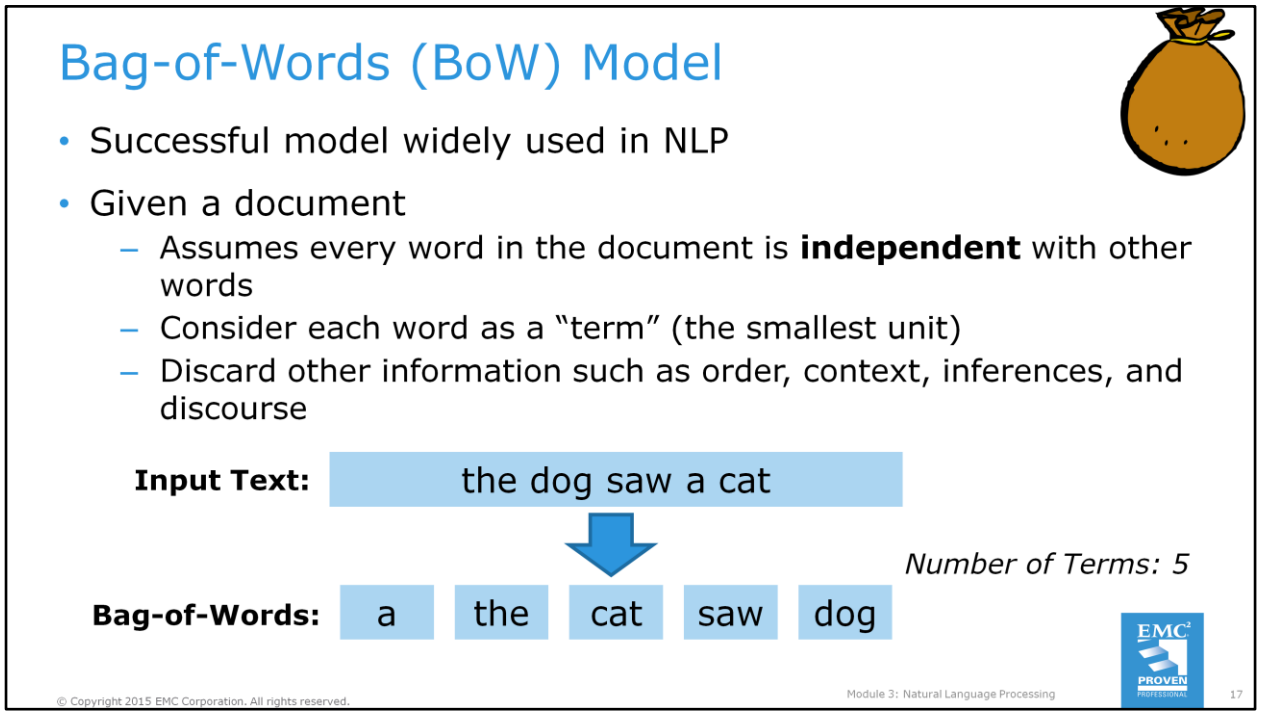
Text Preprocessing

- Text preprocessing is a broad term that contains various tasks, and is often the first step of an NLP application
  - With bag-of-words
  - Without bag-of-words
- Sample uses of text preprocessing
  - Tokenize or segment words from Twitter firehose
  - Normalize word formats, dates, locations
  - Identify the part-of-speech of a sentence
  - Identify the names and locations in the text

Text preprocessing is a broad term which includes many tasks that add some structures or logistics to the raw text. It is often the first step of most NLP tasks, and can be used with or without bag-of-words.

Text preprocessing makes it easy for subsequent steps, such as information extraction and text mining, to process the text and discover insights.

The slide includes a few examples of text preprocessing. Next, we will look at how text preprocessing may work with or without the bag-of-words model.

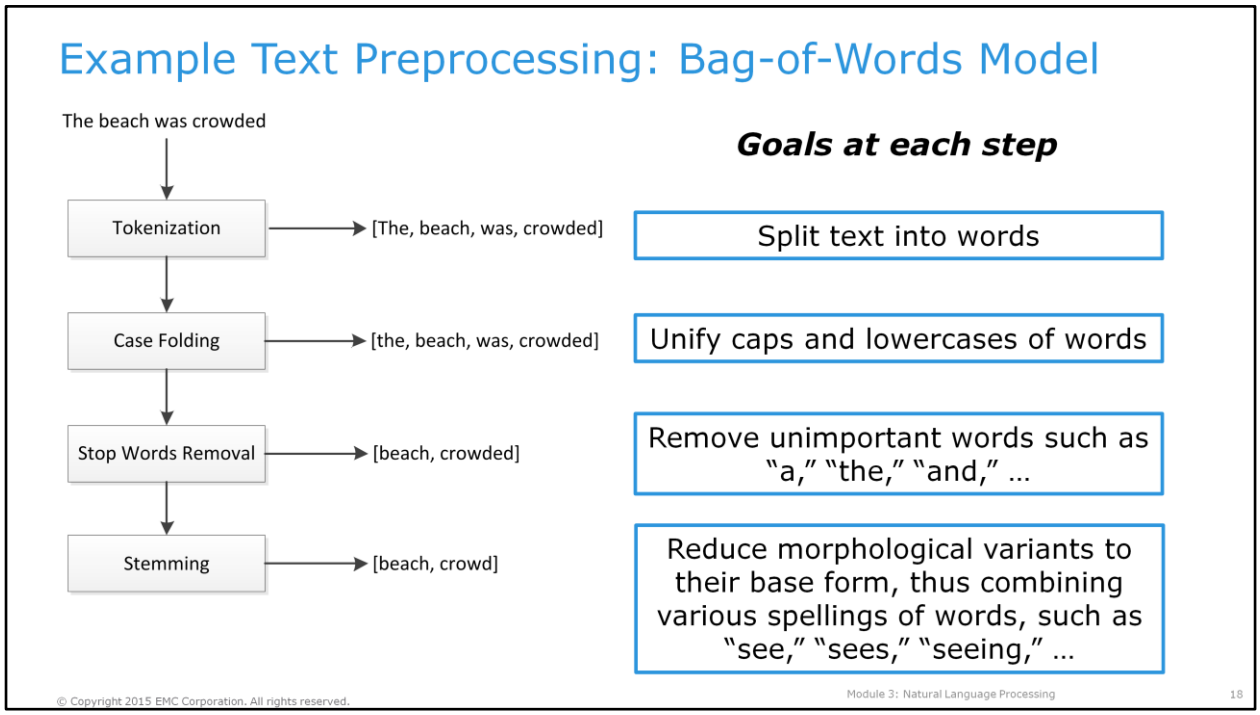


A simple yet widely used approach to represent text is called bag-of-words (BoW). Given a document, BoW represents the document as a set of terms, ignoring information such as order, context, inferences, and discourse. Each word is considered a term or token, which is often the smallest unit for the analysis.

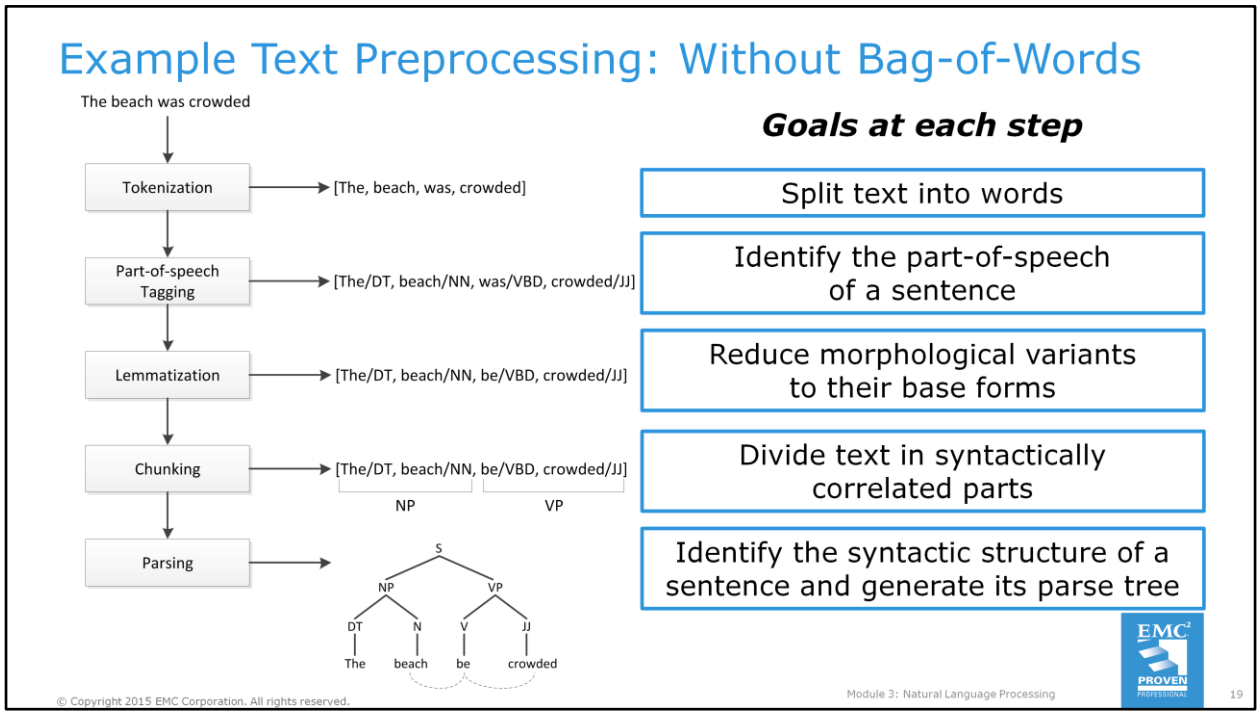
In many cases, BoW additionally assumes every term in the document is independent. The document then becomes a vector with one dimension for every distinct term in the space, and the terms are unordered. The permutation D* of document D contains the same words exactly the same number of times but in a different order. Therefore, using the BoW representation, document D and its permutation D* would share the same representation.

The BoW representation can be implemented with data types such as `dict` in Python or `hashmap` in Java. In Python for example, the built-in `dict` type or a more efficient alternative `collections.defaultdict` type can be used to store unordered `<word, frequency>` pairs.

```
>>> s = "the dog saw a cat"
>>> from collections import defaultdict
>>> dt = defaultdict(int)
>>> for w in s.split():
        dt[w] += 1
>>> dt
defaultdict(<type 'int'>, {'a': 1, 'the': 1, 'saw': 1, 'dog': 1, 'cat': 1})
```



Bag-of-words is perhaps the simplest model for text preprocessing. The slide shows an example workflow of text preprocessing that includes some sample tasks of the bag-of-words model. This lesson will discuss how NLP tasks such as tokenization, case folding, stop words removal, and stemming may function within the bag-of-words model.

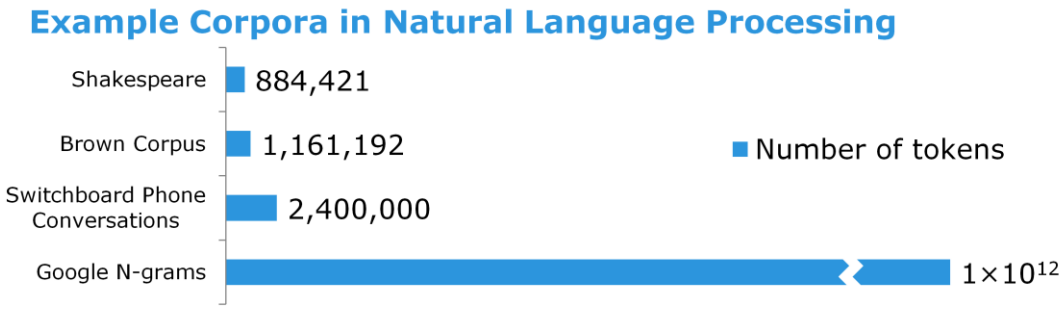


This slide shows an example of text preprocessing without the bag-of-words. This workflow shares some tasks as those in the bag-of-words, such as tokenization and lemmatization. Lemmatization is similar to stemming. Later in the lesson we will discuss their differences. However, the workflow contains other tasks including part-of-speech tagging, chunking and parsing that preserves the word order and helps identify other information, including context, inferences, and discourse.

Part-of-speech tagging scans through a sentence and tags each word with its part-of-speech. In the example, "The" is tagged as a determiner (DT), "beach" as a noun (NN), "was" as a past tense verb (VBD), and "crowded" as an adjective (JJ). We will discuss part-of-speech tagging later in the lesson.

Tokenization Splits Raw Text into Tokens

- Tokenization is the process of converting raw text into collections of tokens.
- By default, **1 token = 1 word**



Tokenization is the NLP task of separating out words from the body of text. After the tokenization, raw text is converted into collections of tokens, where each token is generally a word.

A **corpus** (plural: **corpora**) is a large collection of texts used for various purposes in natural language processing. The chart shows the token counts of a few example corpora that are commonly used in NLP research. The smallest corpus in the list, the complete works of Shakespeare, contains about 0.88 million words. In contrast, the Google n-gram corpus contains one trillion words from publicly accessible web pages. Text analysis often has to deal with textual data that is complex. In fact, the high dimensionality of text is an important issue, and has a direct impact on the complexities of many text analysis tasks.

References:

- Shakespeare (http://shakespeare.mit.edu/)
  - Word count: 0.88 million
  - Domain: Written
- Brown Corpus (http://icame.uib.no/brown/bcm.html)
  - Word count: 1 million
  - Domain: Written
- Switchboard Phone Conversations (http://catalog.ldc.upenn.edu/LDC97S62)
  - Word count: 3 million
  - Domain: Spoken
- Google N-grams (http://catalog.ldc.upenn.edu/LDC2006T13)
  - Word count: 1 trillion
  - Domain: Written

Python NLTK Offers Multiple Tokenization Methods

LMack @lauriemackaye24
I've been feeling awful all day. I think I **caught** the **flu** from my brother. :(
Expand Reply Retweet Favorite More

```
>>> from nltk.tokenize import *
>>> tweet = "I've been feeling awful all day. I think I caught the flu from my
brother. :("
>>> tweet.split()
["I've", 'been', 'feeling', 'awful', 'all', 'day.', 'I', 'think', 'I', 'caught', 'the',
'flu', 'from', 'my', 'brother.', ':(']
>>> wordpunct_tokenize(tweet)
['I', "'", 've', 'been', 'feeling', 'awful', 'all', 'day', '.', 'I', 'think', 'I',
'caught', 'the', 'flu', 'from', 'my', 'brother', '.', ':(']
>>> word_tokenize(tweet)
['I', "'ve", 'been', 'feeling', 'awful', 'all', 'day.', 'I', 'think', 'I', 'caught',
'the', 'flu', 'from', 'my', 'brother.', ':', '(']
>>> regexp_tokenize(tweet, pattern='\w+')
['I', 've', 'been', 'feeling', 'awful', 'all', 'day', 'I', 'think', 'I', 'caught', 'the',
'flu', 'from', 'my', 'brother']
>>> sent_tokenize(tweet)
["I've been feeling awful all day.", 'I think I caught the flu from my brother.', ':(']
```


Here we show how to perform tokenization using NTLK's built-in tokenizers.

A common approach of tokenizing English text is based on spaces. This is achieved by using either `tweet.split()` or `SpaceTokenizer().tokenize(tweet)`. Note that word "I" is not separated from the token "I've." This is the result of only using space as the separator. Tokenizing the text based on punctuation marks using `wordpunct_tokenize(tweet)` may somewhat alleviate the problem.

However, it may be more preferable to tokenize contractions such as "I've" into two tokens [I, 've] instead of three tokens [I, ', ve]. The `word_tokenize(tweet)` uses NLTK's recommended word tokenizer to tokenize words in a given sentence.

If none of the previous tokenizers produces preferable results, you may want to use the `regexp_tokenize()` function and specify a regular expression of the pattern you want. In the given example, pattern='\w+' tells the `regexp_tokenize()` function to only produce words and remove all punctuations.

Sometimes you may prefer to first tokenize the text into sentences using the `sent_tokenize()` function, and then tokenize each sentence into words using the most preferable word tokenizer.

Tokenization Issues in English

- Should we map these words to the same token?
  - Résumé → Résumé résumé resume ?
  - Ph.D → Ph.D PhD Ph.D. PHD phd ph.d ph.d. D.Phil. Dphil ?
  - Jimmy's Pizza → Jimmy s Jimmys Jimmy's ?
  - What're, I'm, isn't → What are, I am, is not ?
  - Lowercase → lower-case lowercase lower case ?
- How many tokens?
  - State-of-the-art, 50-year-old → State of the art, 50 year old ?
  - B-Tree, Wi-Fi, Coca-Cola → One token or two ?
  - San Francisco → one token or two ?
  - O'Neil → One token or two ?
  - Robertson IV → One token or two ?

Tokenization is a much more difficult task than one may expect. For example, should we map words like Résumé, résumé, and resume to the same token? Should words like state-of-the-art, Wi-Fi, and San Francisco be considered one token or more?

Tokenization Issues in French and German

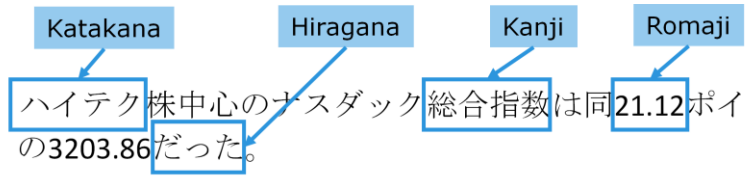
- French
  - L'ensemble: One token or two?
- German compound nouns are not segmented
  - *Rechtsschutzversicherungsgesellschaften*
  - Translation: Legal protection insurance companies
  - Need to use German compound splitting

Tokenization is even more difficult beyond English. French suffers a similar tokenization issue as in English. In German, there are many unsegmented compound nouns.

Tokenization Issues in Chinese and Japanese

- Chinese: no spaces between words
  - 日文章鱼怎么说?
    - 日文　　章鱼　　怎么 说 ?
    - Japanese octopus how say → How to say octopus in Japanese?

    - 日　　文章　　鱼　　怎么　　说 ?
    - Sun　　article　　fish　　how　　say
- Japanese has multiple alphabets intermingled



  - ハイテク株中心のナスダック総合指数は同21.12ポイント安の3203.86だった。
  - End-user can express query entirely in hiragana

In Chinese, there are no spaces between words. Segmenting sentences in the wrong places will result in very different meanings.

Japanese has four alphabets intermingled: hiragana, katakana, kanji, and romaji. End-users can express a query entirely in hiragana.

It's safe to say that there is no single tokenizer that will work in every scenario. The Data Science team needs to decide what counts as a token depending on the domain of the task and select an appropriate tokenization technique that fits most situations well. In reality, it's common to pair a standard tokenization technique with a lookup table to address the contractions and terms that should not be tokenized.

Case Folding

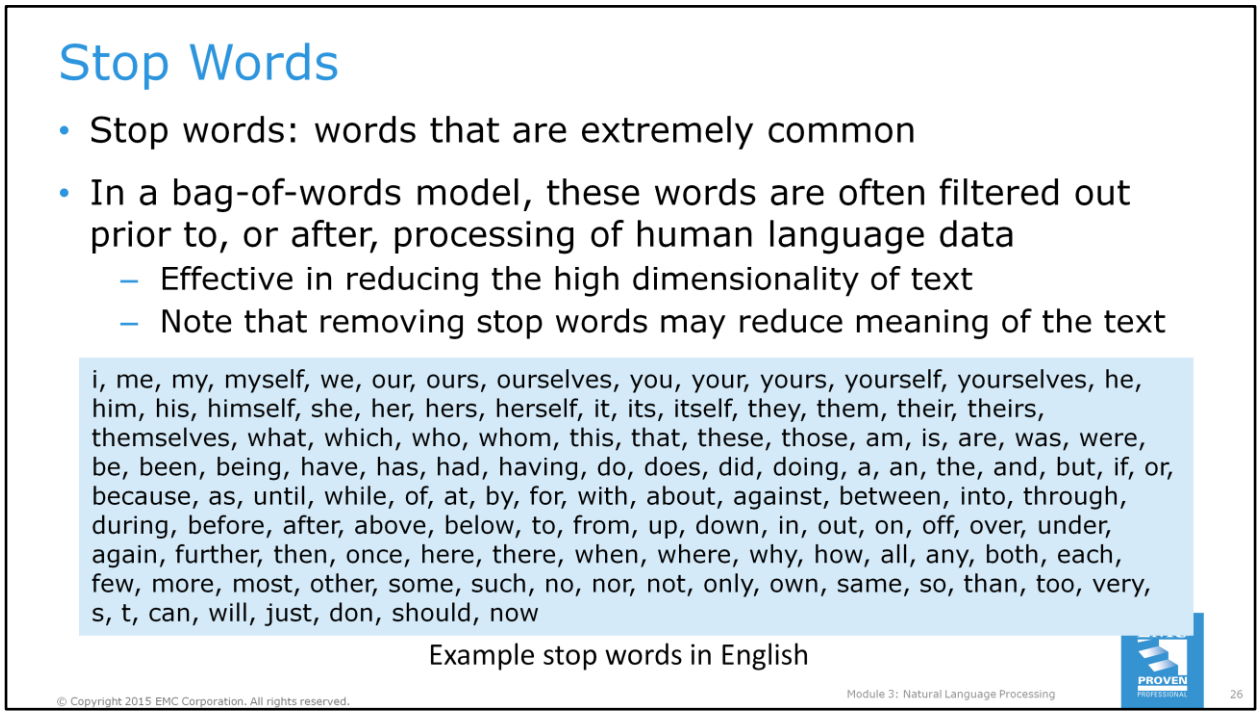
- Reduce all letters to lower case
  - Since most words tend to be in lower case
  - Possible exception: upper case in mid-sentence?
    - e.g., General Motors
    - Fed vs. fed
    - WHO vs. who
- Case folding…
  - Generally well suited for: document indexing, search engines, …
  - Use caution when applying to… Information extraction, sentiment analysis, machine translation
    - For example, US and us

Case folding reduces all letters to lower case. Since most words tend to be in lower case, case folding offers an effective way to quickly reduce the high dimensions in the text. Different instances, such as automobile, Automobile and AUTOMOBILE, can be mapped to the same word. This will help information retrieval systems, such as a web search engines, to fetch more related documents.

However, case folding can equate words that should not be equated, such as company names like *General Motors*, government organizations like *the Fed*, and acronyms such as *WHO*.

Therefore, you need to be cautious applying case folding to tasks such as information extraction, sentiment analysis, and machine translation. If implemented incorrectly, case folding may reduce or change the meaning of the text and create additional noise.

If case folding must be present, one way to address its problems is to create a lookup table of words not to be case folded. Alternatively, you can come up with some heuristics or rules-based strategies for the case folding. For example, the program can be taught to ignore words that have uppercase in the middle of a sentence.



For the purpose of reducing dimensionality, not all the words from a given language need to be included in the term frequency vector.

In English, for example, it is common to remove words such as *the*, *a*, *of*, *and*, *to*, and other articles that may not contribute to semantic understanding. These common words are called **stop words**.

Lists of stop words, or stop lists, are available in various languages for automating the identification of stop words. The Snowball's stop list contains stop words in more than ten languages. Shown here is the Snowball's stop list in English, containing 127 stop words. There are other stop lists available, such as the SMART stop list, which contains 570 words.

Note that removing the stop words may reduce the meaning of the text. Web search engines generally do not discard stop words. Also, for tasks such as discourse and information extraction, it's better to keep the stop words.

References:

- Snowball's stop list in English http://snowball.tartarus.org/algorithms/english/stop.txt
- SMART stop list http://jmlr.csail.mit.edu/papers/volume5/lewis04a/a11-smart-stop-list/english.stop

Stop Words Removal in NLTK

- Define the following function to remove stop words

```
def remove_stopwords(text):
    import nltk
    stopwords = nltk.corpus.stopwords.words('english')
    return [w for w in text.split() if w.lower() not in stopwords]
```

```
>>> print remove_stopwords('the beach was crowded')

['beach', 'crowded']

>>> print remove_stopwords('The U.S. economy rebounded more strongly than initially thought
in the second quarter with more of the growth being driven by domestic demand and less by
restocking by businesses.')

['U.S.', 'economy', 'rebounded', 'strongly', 'initially', 'thought', 'second', 'quarter',
'growth', 'driven', 'domestic', 'demand', 'less', 'restocking', 'businesses.']
```


The Snowball's stop lists are included in the standard distribution of NLTK.

Method `nltk.corpus.stopwords.words('english')` imports all the stop words in English. The code defines a function `remove_stopwords(`*`text`*`)` to remove all the stop words from a string passed as a parameter.

The lab at the end of this lesson will provide more practice on how to remove stop words from texts.

Reduce Words to Base Form

- We often want to reduce inflections or variant forms to their base forms
  - cat, cats, cat's, cats' → cat
  - organize, organizes, and organizing → organize
  - democracies, democratic, and democratization → democracy
- Simply removing the –s, –es and –ing suffixes may create additional problems
  - Removing the suffix of *using* becomes *us*
  - Removing the suffix of *clothes* becomes *clothe* or *cloth*
- Solutions
  - Lemmatization
  - Stemming

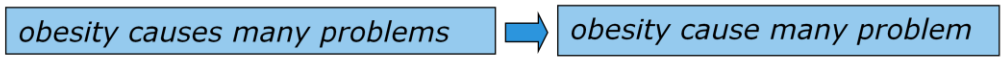
In text preprocessing, we often want to reduce inflections or variant forms to its base form as a way to reduce the high dimensionality in the text. For example, we may want to map words such as *play*, *plays*, *played*, and *playing* to the same term.

We could simply remove the –s, -es and –ing suffixes, but this may create additional problems. For example, if we removing the suffix of *using* it becomes *us*, and *clothes* becomes *clothe* or *cloth*.

A more elegant alternative is to use **lemmatization** or **stemming**. Lemmatization and stemming are two different techniques that reduce the number of dimensions and reduce inflections or variant forms to their base forms.

Lemmatization Finds the Dictionary Base Form

- Reduce inflections or variant forms to the base forms
- Usually with the use of a human language dictionary
- **The output base form always exists in the dictionary**
- Example:
  - am, are, is → be
  - cat, cats, cat's, cats' → cat
  - the company's jobs are posted→ the company job be post
- Works on the entire text



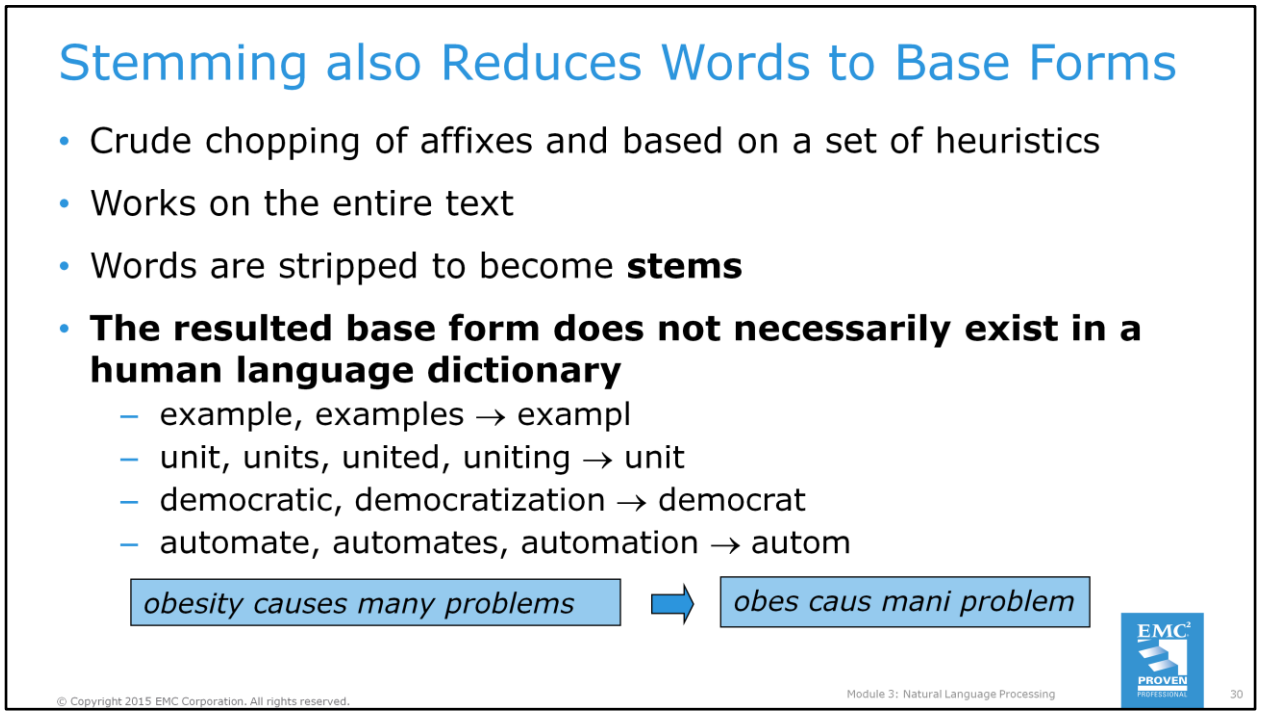
With the use of a given dictionary, **lemmatization** finds the correct dictionary base form of a word. For example, given the sentence

- obesity causes many problems

the output of lemmatization would be

- obesity cause many problem

Note that lemmatization requires the use of a human language dictionary, such as Webster, WordNet, etc. The output base forms always exist in the dictionary, unless the original word is not defined in the dictionary, in which case lemmatization usually returns the word in its original form.



Different from lemmatization, **stemming** does not need a dictionary, and it usually refers to a crude process of stripping affixes based on a set of heuristics with the hope of correctly achieving the goal to reduce inflections or variant forms. After the process, words are stripped to become **stems**. A stem is not necessarily an actual word defined in the human language, but it is sufficient to differentiate itself from the stems of other words.

Stems are the core meaning-bearing units. **Affixes**, including prefixes, suffixes, infixes, and circumfixes, are the bits and pieces that adhere to stems. Both stems and affixes constitute **morpheme**, the smallest meaningful unit in the grammar of a language.

Lemmatization and stemming may also differ in that stemming most commonly collapses derivationally related words, whereas lemmatization commonly only collapses the different inflectional forms of a lemma.

Reference:

"Stemming and Lemmatization." *nlp.stanford.edu.* Cambridge University Press (2008). http://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

Porter's Stemming Algorithm

- Most widely used stemming algorithm
- View any word as the form $[C](VC)^m\,[V]$
  - *C* is a sequence of one or more consonants
  - *V* is a sequence of one or more vowels
  - *m* will be called the measure of any word or word part when represented in this form

Step 1a

sses → ss	caresses → caress
ies → i	ponies → poni
ss → ss	caress → caress
s → ø	cats → cat

Step 1b

(*v*)ing → ø	motoring → motor
	sing → sing
(*v*)ed → ø	plastered → plaster
…	

Step 2

ational→ ate	relational → relate
ization→ ize	vietnamization → vietnamize
ation→ ate	predication → predicate
…	

Step 3-5 (other scenarios)

ative → ø	formative → form
al → ø	revival → reviv
able → ø	adjustable → adjust
…	

A well-known rule-based stemming algorithm is Porter's stemming algorithm. It defines a set of production rules to iteratively transform words into their stems. The Porter stemmer uses m to represent the length of a word or word part. If *C* is a sequence of one or more consonants, and *V* a sequence of one or more vowels, any word part has the form

$$[C](VC)^m\,[V],$$

which is to be read as an optional *C*, followed by *m* repetitions of *VC*, followed by an optional *V*.

Reference:

"The Porter stemming algorithm." *snowball.tartarus.org.* accessed November 5, 2014. http://snowball.tartarus.org/algorithms/porter/stemmer.html

Lemmatization and Stemming with NLTK

- The **`nltk.stem.wordnet`** module has a **`WordNetLemmatizer`** class
- Lemmatize using WordNet's built-in morphy function
- If an input word cannot be found in WordNet, it is returned "as is"

```
>>> from nltk.stem.wordnet import WordNetLemmatizer
>>> lmtzr = WordNetLemmatizer()
>>> s = 'obesity causes many problems'
>>> print[lmtzr.lemmatize(w) for w in s.split()]
['obesity', u'cause', 'many', u'problem']
```

- The **`PorterStemmer`** class implements the Porter stemming algorithm

```
>>> from nltk.stem import PorterStemmer
>>> porter = PorterStemmer()
>>> s = 'obesity causes many problems'
>>> print [porter.stem(w) for w in s.split()]
[u'obes', u'caus', u'mani', u'problem']
```


Here we show how to perform lemmatization and stemming in NLTK.

The `nltk.stem.wordnet` module has a `WordNetLemmatizer` class, which provides functionality to lemmatize using WordNet's built-in morphy function. If an input word cannot be found in WordNet, the word is returned in its original form, or "as is". The example shows how lemmatization and stemming output different results over the same sentence "obesity causes many problems." For lemmatization, the sentence is turned into four tokens: "obesity," "cause," "many," and "problem." The "u" before "cause" and "problem" stands for Unicode. When lemmatizing a token, the WordNetLemmatizer automatically converts it into Unicode to better support non-Latin languages.

The `PorterStemmer` class defined in the `nltk.stem` module implements the Porter stemming algorithm. For stemming, the four words in the sentence are turned into stems: "obes," "caus," "mani," and "problem."

Lab Exercise 7: Basic Text Processing with NLTK

- This lab introduces you to natural language process using Python NLTK.
- After completing the tasks in this lab you should be able to:
  - Understand basic concepts of text processing
  - Perform basic tasks with Python NLTK

Check Your Knowledge

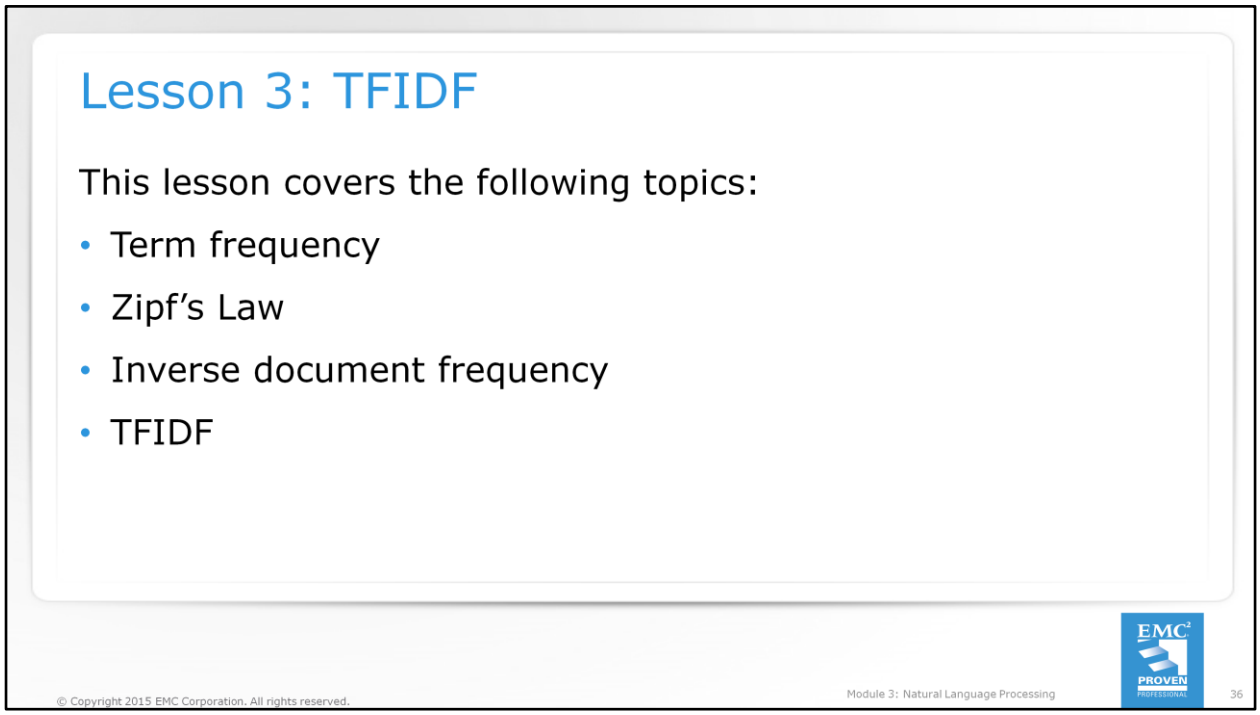
- What is a bag-of-words model?
- What is tokenization?
- What issues might tokenization run into?
- What is case folding?
- What are stop words?
- What information does bag-of-words discard?
- What is the difference between lemmatization and stemming?

Write your answers here.

Lesson 2: Summary

During this lesson the following topics were covered:

- Bag of Words
- Tokenization
- Case Folding
- Stop words removal
- Lemmatization
- Stemming



In this lesson, we will revisit TFIDF, a simple statistical measure to identify meaningful words without any background knowledge of what the corpus is about.

What Kind of Problem Do I Need to Solve? How Do I Solve It?

The Problem to Solve	The Category of Techniques	Example Model Covered in the module
I want to identify the important words in raw text.	Information Retrieval	TFIDF (Lesson 3)
I want to model my corpus and predict the next word my user will enter.	Language Modeling	Trigram language model (Lesson 5)
I want to extract names, dates and places, or distinguish nouns from verbs.	Named Entity Recognition, Part-of-speech tagging	Trigram Hidden Markov Models (Lesson 6)
I want to measure user sentiments toward my product.	Sentiment Analysis	Naïve Bayes (Lesson 7)
I want to quickly obtain a list of topics as a summary of an article.	Topic models	Latent Dirichlet Allocation (Lesson 7)

Here is a list of some NLP problems, their categories of techniques, and the example models we will discuss in the next several lessons.

Why These Example Techniques?

- Most popular, frequently used:
  - Provide the foundation for NLP skills on which to build
- Easy to understand & comprehend
- Applicable to a broad range of problems in several verticals

Over the next several lessons, we will revisit TFIDF and naïve Bayes, which were introduced in the previous course. This module will also present first and second order Markov processes, trigram hidden Markov models, and topic models.

The reasons for which these techniques are chosen among all the available techniques are listed on this slide.

Term Frequency (TF)

- Assign each term in a document a weight for that term
- The weight of a term *t* in a document *d* is a function of the number of times *t* appears in *d*
  - $TF_1(t,d) = \sum_{i=1}^{n} f(t,t_i) \qquad t_i \in d; |d| = n$ where
  - $f(t,t') = \begin{cases} 1, & \text{if } t = t' \\ 0, & \text{otherwise} \end{cases}$

Document

there is a man with a telescope

there	1
is	1
a	2
man	1
with	1
telescope	1

Given a term *t* and a document $d = \{t_1, t_2, t_3, \ldots t_n\}$ containing *n* terms, the simplest form of term frequency of *t* in *d* can be defined as the number of times *t* appears in *d*. Given the document "there is a man with a telescope," the corresponding term frequencies are shown on the slide.

The term frequency function can be logarithmically scaled. Recall from the previous course, we learned that the logarithm can be applied to distribution with a long tail in order to enable more data detail. Similarly, the logarithm can be applied to word frequencies whose distribution also contains a long tail:

$$TF_2(t,d) = \log\,[TF_1(t,d) + 1]$$

Because longer documents contain more terms, they tend to have higher term frequency values. They also tend to contain more distinct terms. These factors can conspire to raise the term frequency values of longer documents and lead to undesirable bias favoring longer documents. To address this problem, the term frequency can be normalized. For example, the term frequency of term *t* in document *d* can be normalized based on the number of terms in *d*:

$$TF_3(t,d) = \frac{TF_1(t,d)}{n} \qquad |d| = n$$

Besides the three common definitions mentioned earlier, there are other less common variations of term frequency.

In practice, one needs to choose the term frequency definition that is the most suitable to the data and the problem to be solved.

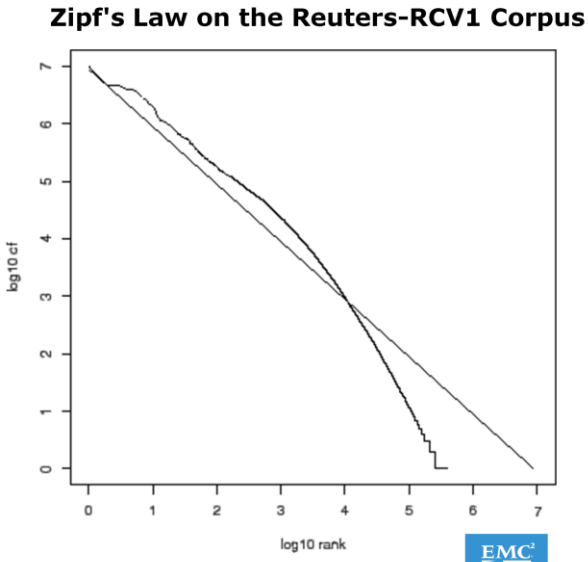
Reference:

"Document and query weighting schemes." *nlp.stanford.edu.* Cambridge University Press (2008). http://nlp.stanford.edu/IR-book/html/htmledition/document-and-query-weighting-schemes-1.html.

Zipf's Law

- A commonly used model of the distribution of terms in a collection
- The collection frequency (cf) of the i-th most common term is proportional to $1/i$

$$cf_i \propto i^{-1}$$



Zipf's law models the size, (or frequency), of an occurrence of an event relative to it's rank i. It is named after George Kingsley Zipf, a Harvard linguistics professor, who sought to determine the size of the 3rd, or 8th, or 100th most common word.

Zipf's Law is useful to estimate the frequency of a word in a large corpus. For example, you can randomly select the subset of the corpus, and rank the words by the number of times they appear. The frequencies of these words are proportional to the popularity over the entire corpus.

This graph shows the frequencies of words from the Reuters-RCV1 corpus that follow the Zipf's law. The Reuters-RCV1 corpus consists of 800,000 manually categorized newswire stories from Reuters. In the graph, frequency is plotted as a function of the frequency rank for the terms in the corpus. The line in the graph is the distribution predicted by Zipf's law.

Zipf's law is a special case of the power law. Power law appears widely in various fields. The next module on social network analysis will discuss more on power law.

Reference:

D. D. Lewis, Y. Yang, T. G. Rose, and F. Li (2004)., "RCV1: A new benchmark collection for text categorization research." *The Journal of Machine Learning Research*, *5*, pp. 361-397.

Inverse Document Frequency (IDF)

- The number of documents that contain *t*
  - $IDF_1(t) = \log\frac{N}{DF(t)}$
  - *N*: Number of documents in the corpus
- $DF(t)$ defines the document frequency of t
  - $DF(t) = \sum_{i=1}^{N} f'(t, d_i)$ $\quad d_i \in D; |D| = N$ where
  - $f'(t, d') = \begin{cases} 1, & \text{if } t \in d' \\ 0, & \text{otherwise} \end{cases}$
- IDF measures term uniqueness in a corpus
- Avoids a division-by-zero:
  - $IDF_2(t) = \log\frac{N}{DF(t)+1}$

Term frequency by itself suffers a critical problem; it regards the stand-alone document as the entire world. The importance of a term is solely based on its presence in this particular document. Stop words, such as the, and, and a, could be inappropriately considered the most important because they have the highest frequencies in every document. Besides stop words, words that are more general in meaning tend to appear more often, thus having higher term frequencies. In an article about consumer telecommunications, the word phone would be likely to receive a high term frequency.

A quick fix for the problem is to introduce an additional variable that has a broader view of the world—considering the importance of a term not only in a single document but in a collection of documents, or in a corpus. The additional variable should reduce the effect of the term frequency as the term appears in more documents.

Indeed, that is the intention of the inverse document frequency (IDF). The IDF inversely corresponds to the document frequency (DF), which is defined to be the number of documents in the corpus that contain a term. The basic definition of IDF is given on the slide.

If the term is not in the corpus, it leads to a division-by-zero. A quick fix is to add 1 to the denominator:

$$IDF_2(t) = \log\frac{N}{DF(t)+1}$$

The precise base of the logarithm is not material to the ranking of a term. Mathematically, the base constitutes a constant multiplicative factor towards the overall result.

Words with higher IDF tend to be more meaningful over the entire corpus. In other words, the IDF of a rare term would be high, and the IDF of a frequent term would be low.

TFIDF

- Term frequency – inverse document frequency (tf-idf or TFIDF) of term t in document d:
  - $TFIDF(t,d) = TF(t,d) \times IDF(t)$
- Considers both the prevalence of a term within a document (TF) and the scarcity of the term over the entire corpus (IDF)
- Scores words higher that appear more often in a document but occur less often across all documents in the corpus
- Because the TF values may vary, the same term is likely to receive different TFIDF scores in different documents

Despite the fact that IDF encourages words that are more meaningful, it comes with a caveat. Because the total document count of a corpus (N) remains a constant, IDF solely depends on the DF. All words having the same DF value therefore receive the same IDF value. IDF scores words higher that occur less frequently across the documents. Those words that score the lowest DF receive the same highest IDF. In many cases, it is useful to distinguish between two words that appear in an equal number of documents. Methods to further weight words should be considered to refine the IDF score.

The TFIDF, or TF-IDF, is a measure that considers both the prevalence of a term within a document (TF) and the scarcity of the term over the entire corpus (IDF). The TFIDF of a term *t* in a document *d* is defined as the term frequency of *t* in *d* multiplying the document frequency of *t* in the corpus.

TFIDF scores words higher that appear more often in a document but occur less often across all documents in the corpus. Note that TFIDF applies to a term in a specific document. The same term is likely to receive different TFIDF scores in different documents, because the TF values may be different.

TFIDF is efficient in that the calculations are simple and straightforward, and it does not require knowledge of the underlying meanings of the text. But this approach also reveals little of the inter-document or intra-document statistical structure. Later in this lesson we will show how first and second order Markov processes, trigram hidden Markov models, and topic models can address this shortcoming of TFIDF.

Lab Exercise 8: TFIDF

- This lab demonstrates how to compute TFIDF over the Enron email corpus
- After completing the tasks in this lab you should be able to:
  - Understand notions of TF, IDF, and TFIDF
  - Compute TFIDF over any text collections

Check Your Knowledge

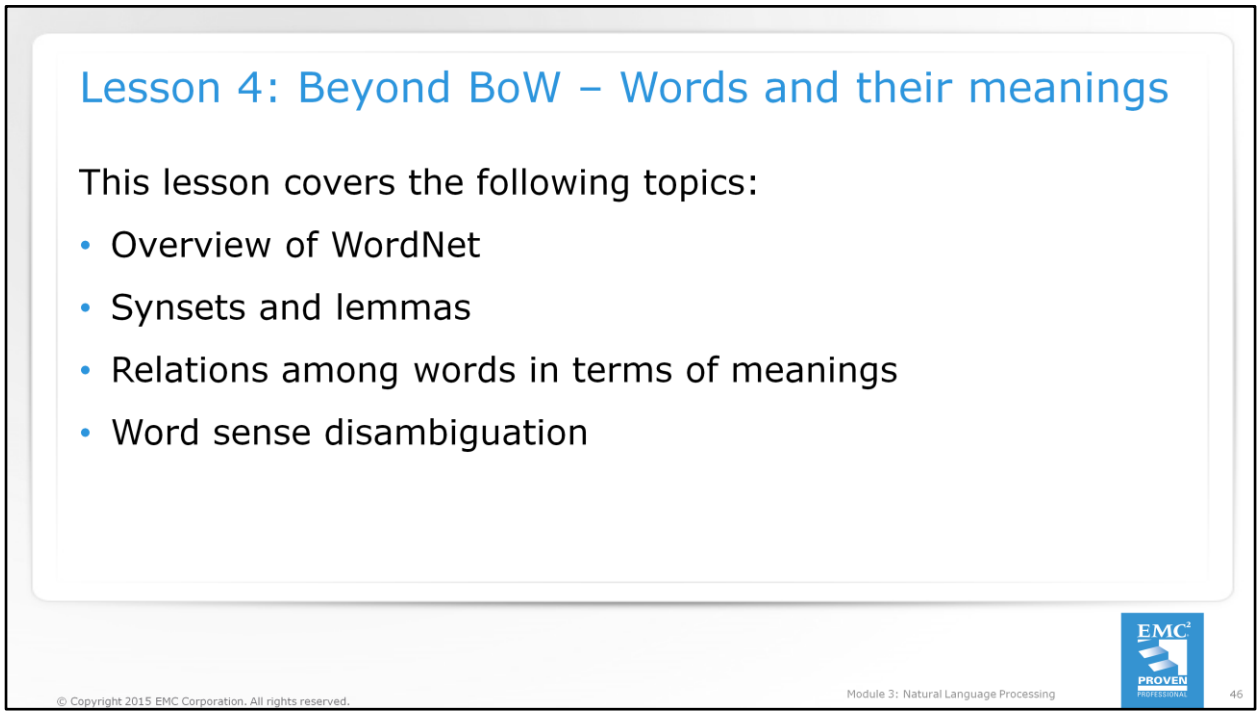
- What is Zipf's law?
- How does TFIDF work?
- What is the benefit of using TFIDF over TF standalone?

Write your answers here.

Lesson 3: Summary

During this lesson the following topics were covered:

- Term frequency
- Zipf's Law
- Inverse document frequency
- TFIDF



In this lesson, we will discuss scenarios where a bag-of-words model is not sufficient and we need to go beyond that. More particularly, we look at words and their meanings. We see how tools such as WordNet can help extract relations among words, thus can be useful for NLP tasks such as word sense disambiguation.

Why is Bag-of-Words not enough?

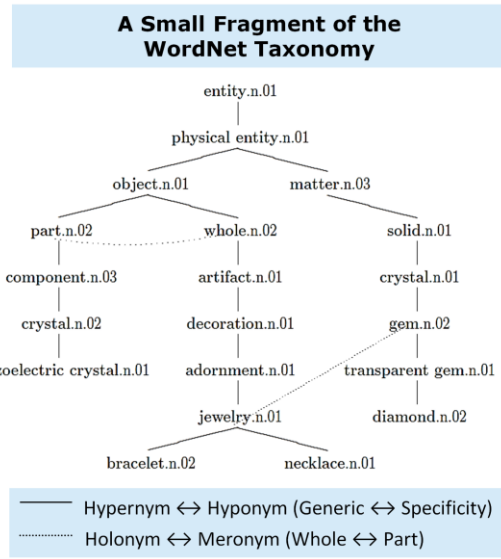
- Word order is important to consider sometimes
  - Only I ate bread yesterday
  - I only ate bread yesterday
  - I ate bread only yesterday
- Certain tasks are difficult with BoW
  - Which word pair has a closer meaning?
    - <bird, rooster> v.s. <bird, roster>
  - Write a summary of a long article
  - Translate from English to Japanese
  - Question-answering
    - IBM Watson in the jeopardy game
- Often necessary to consider word meanings
  - WordNet helps

So far we have only focused on bag-of-words model. BoW model is easy to understand, efficient, and often sufficient for simple NLP tasks. However, in many cases, we need to go beyond that. The slide list a few tasks that are very difficult to do using BoW.

It's often necessary to consider word meanings. Next, we'll see how WordNet can aid in associate words in terms of their meanings.

WordNet: A Large Lexical Database of English

- Combines dictionary with thesaurus
- Words are grouped to synonym sets (synsets)
- Synsets interlinked through semantic and lexical relations
- Synset is in the form of *x.y.z*
  - *x*: word
  - *y*: noun (*n*), verb (*v*), or adjective (*a*)
  - *z*: corresponds to a sense (definition) of word *x*
- Also available in other languages



WordNet (http://wordnet.princeton.edu/wordnet/) is a publicly available English lexical database, created in the Cognitive Science Laboratory of Princeton University under the direction of psychology professor George Miller starting in 1985. WordNet groups nouns, verbs, adjectives, and adverbs into sets of cognitive synonyms (synsets), each expressing a distinct concept. WordNet is often regarded as an ontology for English. WordNet 3.0 contains a total of 117,659 synsets: 82,115 nouns, 13,767 verbs, 18,156 adjectives, and 3,621 adverbs.

Each synset is formatted as `x.y.z`, where `x` is a word, `y` could be a noun (n) or verb (v), and `z` corresponds to a sense of word `x`. For example, synset `crystal.n.02` corresponds to the second definition of the word "crystal" as a noun. Synsets are interlinked through semantic and lexical relations.

The WordNet taxonomy can be loosely regarded as a tree, where the root node is the entity synset. The deeper a synset's position in the tree, the more specific it is. The slide shows a fragment of the WordNet taxonomy. The graph only includes two types of relations. The hypernym-hyponym (generic-specificity) relation is visualized as solid lines and the holonym-meronym (whole-part) relation is visualized as dotted lines.

WordNet is used for tasks such as word sense disambiguation, automatic text classification, automatic text summarization, machine translation and even automatic crossword puzzle generation.

Besides English, WordNet is also available in many other languages. Check out the Global WordNet Association's website (http://globalwordnet.org/) for more detail.

Access WordNet in NLTK

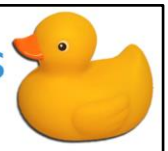
- NTLK provides the WordNet interface as a corpus reader
- Look up a word with `synsets()`, which returns
  - All the senses of the word
  - Synonyms

```
>>> from nltk.corpus import wordnet as wn
>>> wn.synsets('duck')     # synonym sets of duck
[Synset('duck.n.01'), Synset('duck.n.02'), Synset('duck.n.03'),
Synset('duck.n.04'), Synset('duck.v.01'), Synset('duck.v.02'),
Synset('dip.v.10'), Synset('hedge.v.01')]
>>> print(wn.synset('duck.n.01').definition())
small wild or domesticated web-footed broad-billed swimming bird usually
having a depressed body and short legs
>>> print(wn.synset('duck.n.02').definition())
(cricket) a score of nothing by a batsman
>>> print(wn.synset('duck.v.01').definition())
to move (the head or body) quickly downwards or away
```

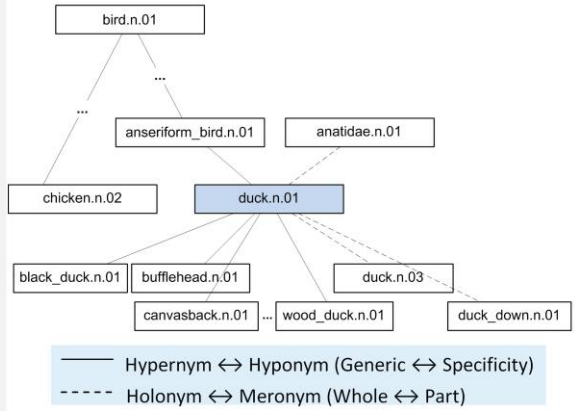

The English WordNet is included in the NLTK, accessed as a corpus reader. The first line of code shows how to import Wordnet into the current workspace as variable *wn*.

The `synsets()` function lookups a word and fetches all the synonym sets (synsets). The x.y.z format of a synset can be passed as a string to the `wn.synset()` function to return a synset object. You can lookup the definition of synset using the `definition()` method.

Look Up Words Through Semantic Relations



```
>>> from nltk.corpus import wordnet as wn
>>> duck = wn.synset('duck.n.01')    # define variable
duck
>>> duck.hypernyms()    # words more generic than duck
[Synset('anseriform_bird.n.01')]
>>> duck.hyponyms()    # words more specific than duck
[Synset('black_duck.n.01'), Synset('bufflehead.n.01'),
Synset('canvasback.n.01'), Synset('dabbling_duck.n.01'),
Synset('diving_duck.n.01'), Synset('drake.n.02'),
Synset('duckling.n.02'), … Synset('wood_duck.n.01')]
>>> duck.part_meronyms()    # components of duck
[Synset('duck.n.03'), Synset('duck_down.n.01')]
>>> duck.member_holonyms()    # words that names the
whole of duck
[Synset('anatidae.n.01')]
>>> # the following code returns the lowest common
ancestor of duck and chicken in the WordNet taxonomy
>>>
duck.lowest_common_hypernyms(wn.synset('chicken.n.02'))
[Synset('bird.n.01')]
```



Useful for semantic analysis

Given a synset, you can look for the synsets that are more generic (hypernyms) or more specific (hyponyms). In the WordNet taxonomy, hypernyms and hyponyms correspond to the parents and children of the synset.

Another way to navigate the WordNet taxonomy is to go from synsets to their components (meronyms) or to the things that they are contained in (holonyms). Meronyms and holonyms can be accessed via the `part_meronyms()` and `member_holonyms()` functions respectively.

The `lowest_common_hypernyms()` function fetches the most immediate, or lowest common ancestor of two synsets in the WordNet taxonomy. This method is very useful for semantic analysis. The example shows that synset `bird.n.01` is the lowest common ancestor of `duck.n.01` and `chicken.n.02`.

Lemmas() with NLTK

- A base word form that is indexed in WordNet
- A synset contains one or more lemmas
  - In other words, one or more words can represent the same sense

```
>>> from nltk.corpus import wordnet as wn
>>> wn.synset('gem.n.01').lemmas()
[Lemma('gem.n.01.gem'), Lemma('gem.n.01.treasure')]
>>> wn.synset('chicken.n.01').lemmas()
[Lemma('chicken.n.01.chicken'), Lemma('chicken.n.01.poulet'),
Lemma('chicken.n.01.volaille')]
>>> wn.synset('dark.n.01').lemmas()
[Lemma('dark.n.01.dark'), Lemma('dark.n.01.darkness')]

>>> wn.lemma('dark.n.01.dark').antonyms()
[Lemma('light.n.09.light')]
>>> wn.lemma('good.a.01.good').antonyms()
[Lemma('bad.a.01.bad')]
>>> wn.lemma('vocal.a.01.vocal').pertainyms()
[Lemma('voice.n.02.voice')]
```


Lemma is defined as the lexical entry for a single morphological form of a sense-disambiguated word. A lemma in NLTK in the format of <word>.<POS>.<number>.<lemma>:

- <word> is the morphological stem identifying the synset
- <POS> is the part-of-speech of the word
- <number> is the sense number
- <lemma> is the morphological form of interest

A synset may contain one or more lemmas. Recall that each synset corresponds to a sense of the word. Therefore, a synset having multiple lemmas means that multiple words can represent the same sense.

For example, synset `gem.n.01` corresponds to the first sense of the word gem as a noun. This `gem.n.01` synset has two lemmas: `gem.n.01.gem` and `gem.n.01.treasure`. Therefore both gem and treasure can represent the `gem.n.01` synset.

Lemmas can also have relations. The `antonyms()` function fetches those words of opposite meanings. The `pertainyms()` function returns words of related meanings.

Word Sense Disambiguation (WSD)

- Refers to the ability to computationally identify the exact sense of a word in a specific context
- Difficult: often considered as an AI-complete problem
- Examples:

I saw one **duck**.
I saw two **duck**s.

I can hear **bass** sounds.
I like grilled **bass**.

I need to get a statement from the **bank**.
Water smacked the river **bank**s in large waves.

- Can be viewed as a classification task

Word Sense Disambiguation (WSD) is the ability to computationally determine which sense of a word is activated by its use in a given context. It's often considered as an AI-complete, or AI-hard, problem; that is, a task whose solution is at least as hard as the most difficult problems in artificial intelligence—making computers as intelligent as people.

The slide shows three example sentence pairs that need WSD:

(Example # 1) Without more context, it's hard to tell whether the word "duck" in the first sentence "I saw one duck" is a verb (moving quickly downward) or a noun (bird). It's more evident that "ducks" in the second sentence refers to the bird.

(Example # 2) The word "bass" in the two sentences clearly denotes different meanings: low-frequency tones and a type of fish, respectively.

(Example # 3) The word "bank" in the two sentences also denotes different meanings: bank-money or bank-riverside, respectively.

WSD can be viewed as a classification task. Word senses are the class labels. A classification method can be used to assign each occurrence of a word to one or more classes based on the evidence from the context and from external knowledge sources. Depending on the classification methods being used, WSD can be unsupervised, semi-supervised or supervised.

Reference:

Nancy Ide and Jean Véronis (March 1998). "Introduction to the special issue on word sense disambiguation: the state of the art." *Computational Linguistics*, 24, 1, pp. 2-40. http://www.aclweb.org/anthology/J98-1001.

Check Your Knowledge

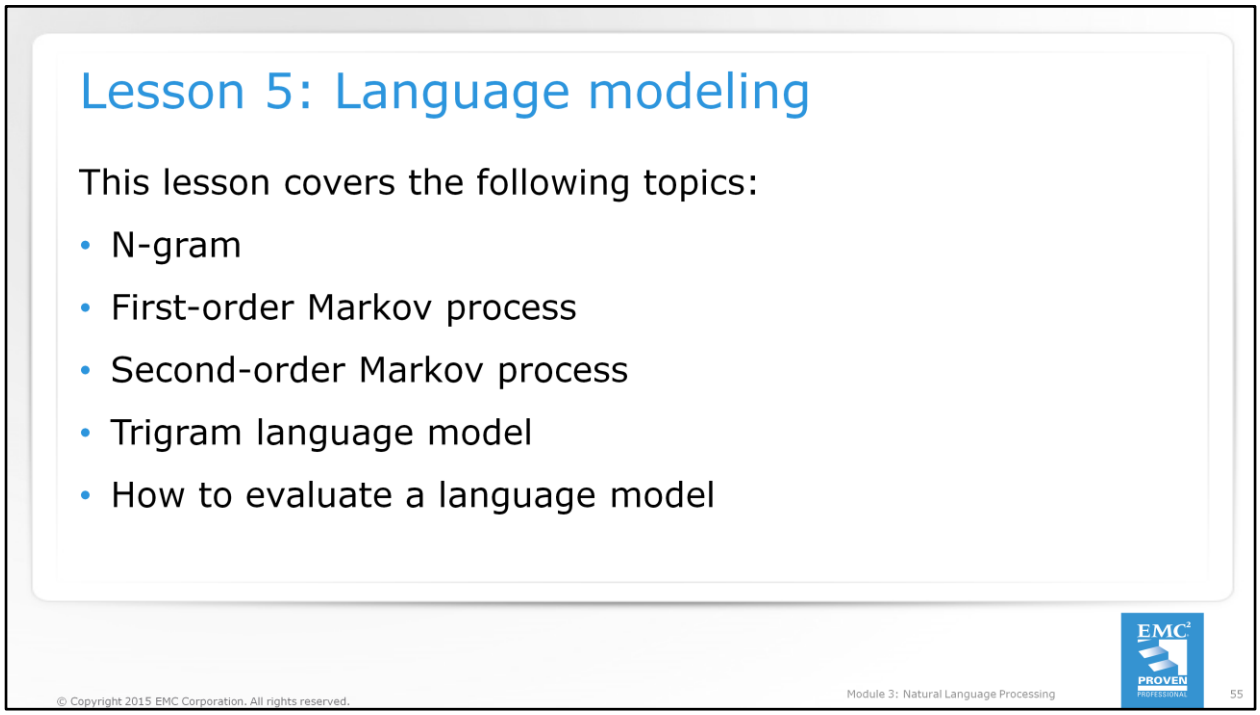
- What is WordNet?
- What is a synset? What is the form of a synset in WordNet?
- What is the lowest common hypernym?
- What is a lemma?
- What is word sense disambiguation?

Write your answers here.

Lesson 4: Summary

During this lesson the following topics were covered:

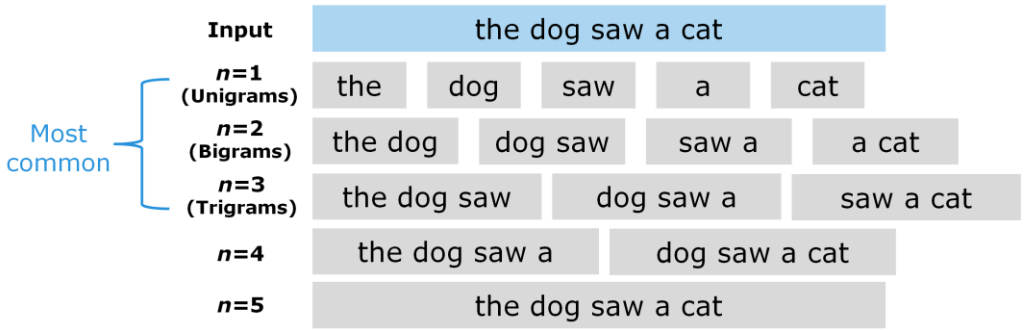
- Overview of WordNet
- Synsets and lemmas
- Relations among words in terms of meanings
- Word sense disambiguation



In this lesson, we will look at the notion of n-grams. We will discuss what language modeling is, and look at how a trigram language mode works based on first-order and second-order Markov processes. We will also show how to evaluate and compare language models.

N-gram

- A contiguous sequence of *n* items from a given sequence of text or speech
- Useful for building statistical models of word sequences (i.e., **language modeling**)
- Essential for tasks such as **word prediction**



A contiguous sequence of n items from a given sequence of text or speech is called an **n-gram**. N-grams are useful for building statistical models of word sequences for word prediction. Identifying which next word would most likely occur is equal to computing the probability of the next word. This is closely related to computing the probability of a sequence of words.

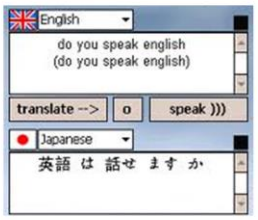
For example, given a simple input text such as "the dog saw a cat," the shown n-grams ($n \in [1,5]$ in this case) can be extracted. After modeling millions of input text, it may be identified that "dog" is more commonly followed by "saw" than "the," therefore "saw" is more likely to be the next word of "dog" than "the." Note that in related research, it's common to set $n \leq 3$.

N-grams are essential for tasks that have to identify words in noisy, ambiguous input, such as speech recognition, handwriting recognition, machine translation, and spelling correction. Readers can refer to Chapter 4 of the following book for further detail:

Daniel Jurafsky, and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*. 2nd edition. Prentice-Hall.
http://web.mit.edu/6.863/www/fall2012/readings/ngrampages.pdf

Language modeling: Examples

- Example applications that employ language models
  - Augmentative communication systems
  - Automated speech recognition
  - Handwriting recognition
  - Optical character recognition
  - Spelling correction
  - Machine translation
  - Gene prediction



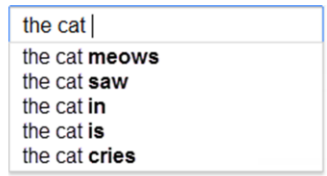
Shown are some example applications that use language modeling.

Language modeling is useful in augmentative communication systems for the disabled. People who are unable to use speech to communicate, like Steven Hawking, can use simple body movements to select words from an augmentative communication system. On such a system, users can select from a list of words that are suggested by word prediction using language models and the system will pronounce the words for them.

Other applications of language modeling include automated speech recognition, handwriting recognition, optical character recognition, spelling correction, machine translation, and gene prediction. These application try to answer the question: what's likely to appear next based on what I've seen so far? – a perfect place for the use of language modeling.

Language modeling: Next Word Prediction

What word will the user enter next given the words entered so far?



- Define *V* to be a *finite* set of all words in English
  - {the, cat, in, the, hat, lady, … }
- A sentence can be represented as a sequence of words $x_1 x_2 \ldots x_n$
  - $x_i \in V$ for $i \in [1, n-1]$
  - x_n is a special symbol (END) assuming $x_n \notin V$
- Define V^* to be the *infinite* set of all sentences with *V*
  - the cat meows END
  - the cat meowed at me END
  - the cat saw a dog END
  - …

Next, let's take a look at how the word prediction as a language modeling problem can benefit from n-grams.

The goal of word prediction is to find which next word is most likely to occur next given the words seen so far. Take a search engine for example, as a user types in the search query, the search engine can use word prediction to suggest the next word that the user is most likely to enter.

A few notations need to be defined before we present the language modeling.

Let *V* be a set of all the words in English. This set can be very large but it's still finite. A sentence in turn can be represented as a sequence of words $x_1 x_2 \ldots x_n$ where the first *n*-1 words are from *V* and the last word x_n is a special symbol END assuming END $\notin V$.

Define V^* to be a set of all sentences with *V*. This set is infinite because sentences can have various lengths.

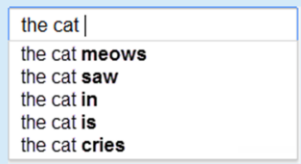
Language modeling: Next Word Prediction

- Assume we have a training set of sentences
- The goal is to infer a probability distribution P that

$$\sum_{\langle x_1 x_2 \dots x_n \rangle \in V^*} P(x_1 x_2 \dots x_n) = 1$$

$$P(x_1 x_2 \dots x_n) \geq 0 \text{ for any } \langle x_1 x_2 \dots x_n \rangle \in V^*$$

- For example:
  - $P(the\ END) = 10^{-15}$
  - $P(the\ cat\ END) = 10^{-12}$
  - $P(the\ cat\ meows\ END) = 2 \times 10^{-8}$
  - $P(the\ cat\ saw\ END) = 10^{-8}$
  - $P(the\ cat\ in\ END) = 3 \times 10^{-9}$
  - $P(the\ cat\ is\ END) = 10^{-9}$
  - …



This example can be considered as identifying $\arg\max_{x_3} P(x_3 | x_1 = the, x_2 = cat)$

- This is formally known as the Markov process (or Markov chain)

Assume we have a training set of sentences $\subset V^*$. The goal of language modeling is to infer a probability distribution *P* that produces the probability score of a sentence $\langle x_1 x_2 \dots x_n \rangle \in V^*$. All the probabilities should sum up to 1.

The slide has presented an example where the user has entered the and cat for a query to the search engine. The system needs to suggest the next word that the user is most likely to enter. The problem is essentially finding the x_3 that maximizes $P(x_3 | x_1 = the, x_2 = cat)$. From parsing the training set, we may find out the probability scores of some sentences as shown on the slide. Based on just the 6 scores in the slide, we can see that $P(x_3 | x_1 = the, x_2 = cat)$ is maximized when $x_3 = meows$. Unless another word receives a higher probability score, the system should choose $meows$ as the next predicted word. The search engine should suggest a list of words sorted by their probabilities.

The process described above is formally known as the **Markov process**.

Markov Models: First-Order Markov Process

- Consider a sequence of random variables $X_1, X_2, \ldots X_n$, and $X_i = x_i$, (where $x_i \in V, i \in [1, n]$)

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n) = P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_1 = x_1, \ldots X_{i-1} = x_{i-1})$$

- First-order Markov process assumes

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n) = P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_{i-1} = x_{i-1})$$

  - That is,
    - For any $i \in [2, n]$
    - $P(X_i = x_i | X_1 = x_1, \ldots X_{i-1} = x_{i-1}) = P(X_i = x_i | X_{i-1} = x_{i-1})$

That is, probability of the next word only depends on the current word.

Consider a sequence of random variables $X_1, X_2, \ldots X_n$, and $X_i = x_i$, (where each x_i is a word that $x_i \in V, i \in [1, n]$)

From the chain rule of probabilities, we know that

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$$

$$= P(X_1 = x_1) P(X_2 = x_2 | X_1 = x_1) P(X_3 = x_3 | X_1 = x_1, X_2 = x_2) \cdots$$
$$P(X_n = x_n | X_1 = x_1, X_2 = x_2, \cdots X_{n-1} = x_{n-1})$$

$$= P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_1 = x_1, \ldots X_{i-1} = x_{i-1})$$

The first-order Markov process makes the following assumption, which considerably simplifies the model:

$$P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_1 = x_1, \ldots X_{i-1} = x_{i-1})$$

$$= P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_{i-1} = x_{i-1})$$

This is called the **first-order Markov assumption**. It assumes that word x_i at the *i-th* position in the sequence only depends on word of the previous position, x_{i-1}.

Markov Models: Second-Order Markov Process

Probability of the next word only depends on the current word and the previous word.

- Second-order Markov assumption

$$P(X_1 = x_1, X_2 = x_2, \dots X_n = x_n)$$
$$= P(X_1 = x_1) \cdot P(X_2 = x_2 | X_1 = x_1)$$
$$\cdot \prod_{i=3}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$
$$= \prod_{i=1}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

- Assumes $x_{-1} = x_0 = \#$ (a special symbol) and $\# \notin V$

The second-order Markov process is similar to the first-order, but it makes a slightly different assumption. It assumes that word x_i at the *i-th* position in the sequence only depends on the previous two words, x_{i-2} and x_{i-1}. That is,

$$P(X_i = x_i | X_1 = x_1, X_2 = x_2, \dots X_{i-1} = x_{i-1}) = P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

Probability of the entire sequence is:

$$P(X_1 = x_1, X_2 = x_2, \dots X_n = x_n) = \prod_{i=1}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

We assume $x_{-1} = x_0 = \#$ where $\#$ is a special symbol and $\# \notin V$

Trigram Language Model

- Based on Markov models
- Consists of
  - A finite set V
  - A parameter $q(w|u,v)$ for trigram (u,v,w) such that $u,v \in V \cup \{\#\}$ and $w \in V \cup \{\text{END}\}$
    - $q(w|u,v)$: The probability of seeing word w immediately after bigram (u,v)
- Probability of sentence $x_1 x_2 \dots x_n$ where $x_i \in V, i \in [1, n-1]$ and $x_n = \text{END}$ is defined by

$$p(x_1 x_2 \dots x_n) = \prod_{i=1}^{n} q(x_i | x_{i-2}, x_{i-1})$$

assuming $x_{-1} = x_0 = \#$

The trigram language model is based on the second-order Markov process, and it is a direct application of Markov models. A trigram language model consists of a finite set *V*, and a parameter $q(w|u,v)$ for any trigram (u,v,w) such that $u,v \in V \cup \{\#\}$ and $w \in V \cup \{\text{END}\}$. The $q(w|u,v)$ represents the probability of seeing word w immediately after bigram (u,v).

Similar to the first-order and second-order Markov processes, the trigram language model represents each sentence as a sequence of words $x_1 x_2 \dots x_n$, where $x_i \in V, i \in [1, n-1]$ and $x_n = \text{END}$. Assuming $x_{-1} = x_0 = \#$ where $\#$ is a special symbol and $\# \notin V$, the trigram language model defines the probability of the sequence $x_1 x_2 \dots x_n$ as

$$p(x_1 x_2 \dots x_n) = \prod_{i=1}^{n} q(x_i | x_{i-2}, x_{i-1})$$

Trigram Language Model (Cont.)

- Consider sentence

  the cat meows END

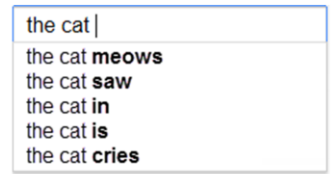
- Then

$$p(the\ cat\ meows\ END) = q(the|\#,\#) \cdot q(cat|\#,the) \cdot q(meows|the,cat) \cdot q(END|cat,meows)$$

- **Maximum-likelihood** estimate: Pick the next word w that maximizes

$$q(w|u,v) = \frac{Count(u,v,w)}{Count(u,v)}$$

- The above estimate is usually applied with **smoothing**



Here we show a concrete example of computing the probability of a sentence under the trigram language model. We know that according to the trigram language model, the probability of a sentence is defined as follows, assuming $x_i \in V, i \in [1, n-1]$, $x_n = \text{END}$ and $x_{-1} = x_0 = \#$:

$$p(x_1 x_2 \dots x_n) = \prod_{i=1}^{n} q(x_i|x_{i-2}, x_{i-1})$$

Therefore, for the example sentence "the cat meows END," its probability is:

$$p(the\ cat\ meows\ END) = q(the|\#,\#) \cdot q(cat|\#,the) \cdot q(meows|the,cat) \cdot q(END|cat,meows)$$

We use **maximum-likelihood** to pick the next predicted word. Given a sentence ending with bigram (u,v), the predicted next word w is the word that maximizes

$$q(w|u,v) = \frac{Count(u,v,w)}{Count(u,v)}$$

We define $Count(u,v,w)$ to be the number of times that the trigram (u,v,w) is seen in the training corpus, and $Count(u,v)$ to be the number of times that the bigram (u,v) is seen in the corpus. For example, the estimate for $q(meows|the,cat)$ is

$$q(meows|the,cat) = \frac{Count(the,cat,meows)}{Count(the,cat)}$$

Note that this estimate has a serious issue. The data is sparse, and the trigram language model has many parameters. So many counts would be zero. This has two implications: (1) many of the $q(w|u,v)$ would be zero which potentially underestimates many trigram probabilities, and (2) the denominator $Count(u,v)$ may be zero, causing a division by zero error. In practice, it's common to replace the estimate with a **smoothed estimation method**, such as linear interpolation.

Linear interpolation defines:

$$q(w|u,v) = \lambda_1 \frac{Count(u,v,w)}{Count(u,v)} + \lambda_2 \frac{Count(v,w)}{Count(v)} + \lambda_3 \frac{Count(w)}{Count()}$$

where parameters $\lambda_1 \geq 0$, $\lambda_2 \geq 0$, $\lambda_3 \geq 0$, $\lambda_1 + \lambda_2 + \lambda_3 = 1$, and $Count()$ is the total number of words in the corpus.

Evaluation of a Language Model with Perplexity

- Consider we have m sentences as the testing set $s_1 s_2 \ldots s_m$
- **Perplexity** evaluates the performance of a language model

$$Perplexity = 2^{-k}, \text{ where } k = \frac{1}{M}\sum_{i=1}^{m} \log_2 p(s_i)$$

  - $p(s_i)$: Probability the model assigns to the *i*-th test sentence
  - M: Total number of words in $s_1 s_2 \ldots s_m$
- A well-performed language model is robust over unseen sentences

Perplexity measures the quality of a language model. Assume we have m sentences $s_1 s_2 \ldots s_m$ as the testing set. Like any testing set, these sentences should be the held-out data. That means, they are new, unseen sentences that are not part of the corpus used to build the language model.

Each sentence s_i is a sequence of words. Let the total number of words in $s_1 s_2 \ldots s_m$ be M, and $p(s_i)$ denotes the probability the language model assigns to the *i*-th test sentence.

Perplexity is defined as $perplexity = 2^{-k}$, where $k = \frac{1}{M}\sum_{i=1}^{m} \log_2 p(s_i)$.

The idea is that, if a language model performs well, it should be able to model unseen sentence. That is, each $p(s_i)$ should receive a high quantity, resulting in a high k and small perplexity.

Evaluation of a Language Model: An Exercise

- Assume we have a vocabulary V and $N = |V \cup \{END\}|$
- In a dumb model that simply predicts the uniform distribution

$$q(w|u,v) = \frac{1}{N}$$

for all $u, v \in V \cup \{\#\}$ and $w \in V \cup \{\text{END}\}$

- Q: What is the perplexity of this dumb model?
- A: $\text{perplexity} = N$.

> Ideally, your language model should receive
> $$\text{perplexity} \ll N$$
>
> The smaller the value of perplexity, the better the language model is at modeling unseen data.

In this dumb model,

$$k = \frac{1}{M}\sum_{i=1}^{m} \log_2 p(s_i) = \frac{1}{M}\log_2 \prod_{i=1}^{m} p(s_i) = \frac{1}{M}\log_2 \left[\left(\frac{1}{N}\right)^{|s_1|} \cdot \left(\frac{1}{N}\right)^{|s_2|} \cdots \left(\frac{1}{N}\right)^{|s_m|}\right] = \frac{1}{M}\log_2 \left(\frac{1}{N}\right)^{|s_1|+|s_2|+\cdots|s_m|}$$
$$= \frac{1}{M}\log_2 \left(\frac{1}{N}\right)^{M} = \log_2 \frac{1}{N}$$

Therefore, the perplexity of this dumb model is $2^{-\log_2 \frac{1}{N}} = N$.

Check Your Knowledge

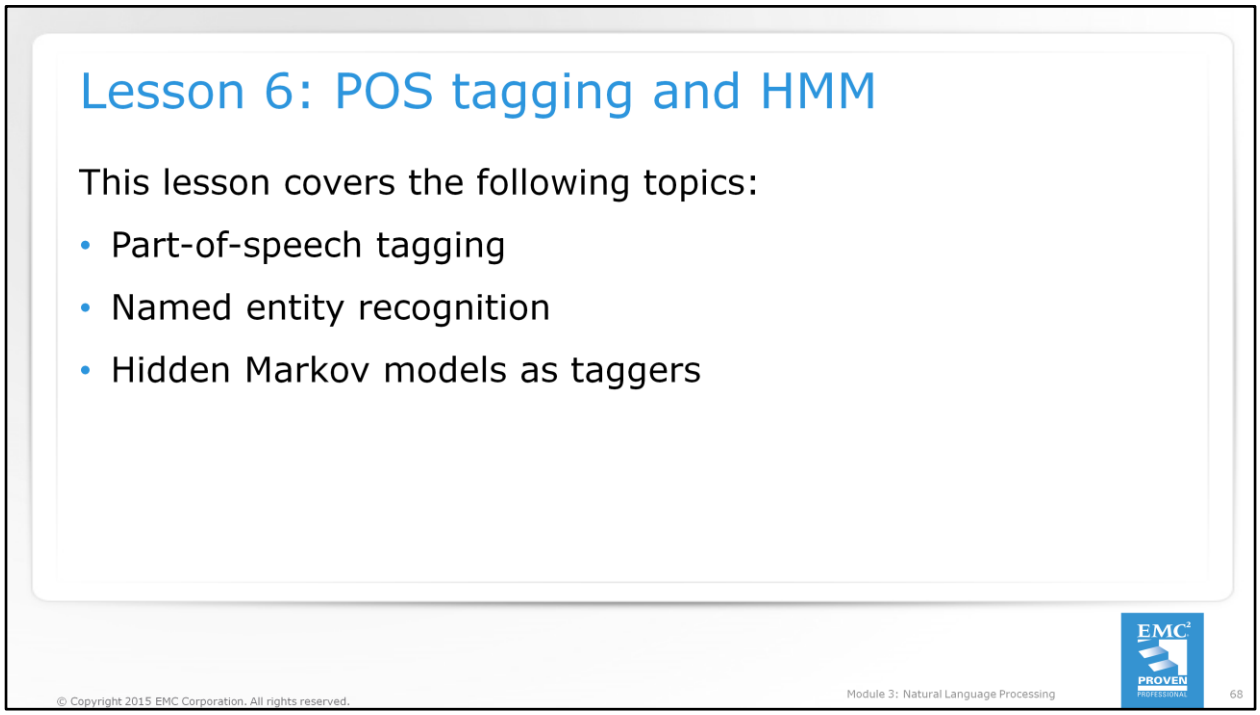
- What is language modeling?
- What is a first-order Markov process?
- What is a second-order Markov process?
- How does a trigram language model work?
- What is perplexity?

Write your answers here.

Lesson 5: Summary

During this lesson the following topics were covered:

- N-gram
- First-order Markov process
- Second-order Markov process
- Trigram language model
- How to evaluate a language model

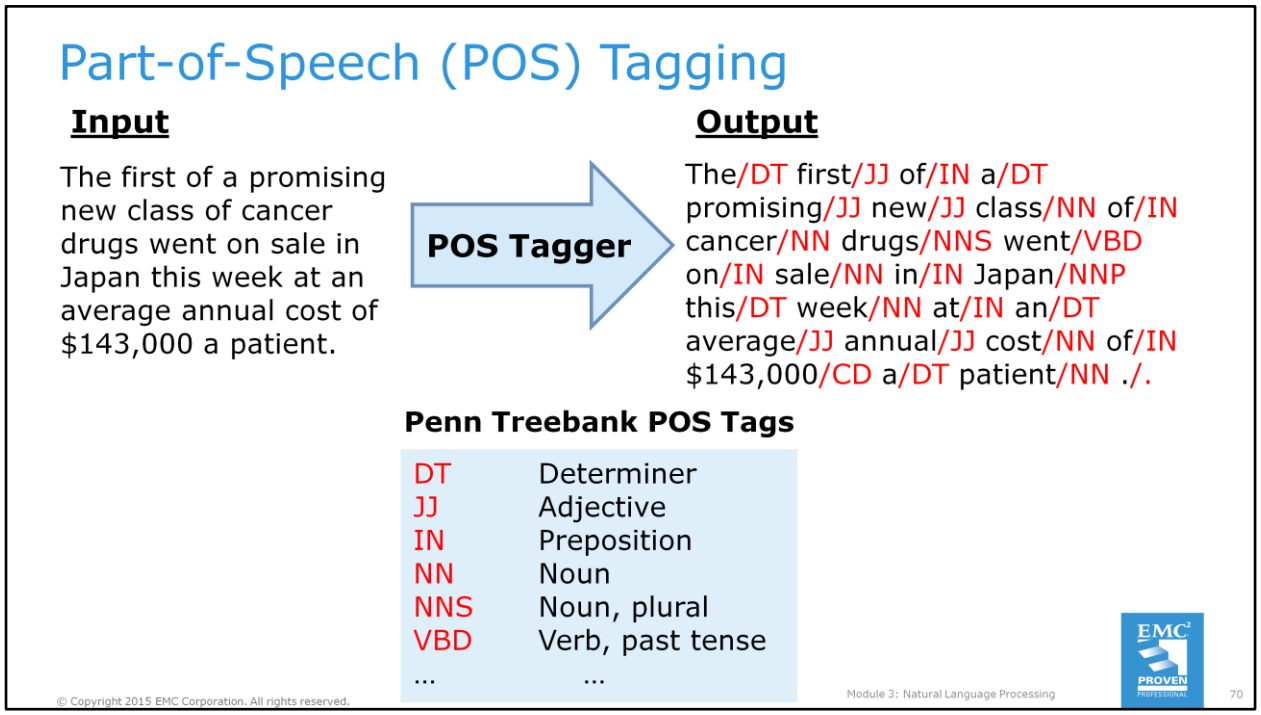


In this lesson, we will look at NLP tasks such as Part-of-Speech Tagging and Named Entity Recognition. We will discuss how the Hidden Markov Model can work as a part-of-speech tagger.

Part-of-Speech (POS) Tagging

- The process of assigning a part-of-speech to each word in a sentence
- Text-to-speech
  - lead \ˈlēd\ or \ˈled\
  - object \ˈäb-jikt\ or \əb-ˈjekt\
  - content \ˈkän-ˌtent\ or \kən-ˈtent\
  - overflow \ˈō-vər-ˌflō\ or \ˌō-vər-ˈflō\
- Word sense disambiguation
  - Noun or verb? Verb or adjective?
    - The cat saw a duck — The cat saw a woman duck
    - Pressing buttons — Pressing issue
  - Useful for tasks such as semantic analysis
- Information extraction
  - Finding names, locations, relations etc.

Part-of-speech tagging, or POS tagging, is the process of assigning a part-of-speech to each word in a sentence. It is often the first step of many practical NLP tasks, acting as a preprocessing step of parsing. POS tagging is useful in tasks such as text-to-speech, word sense disambiguation, and information retrieval.



Here we show an example of how POS tagging works. Before the POS tagging, we need to choose a standard set of POS tags, and allocate one tag for each part of speech. We could pick a very coarse tagset such as N, V, Adj, Adv, Prep., ... But more commonly used tagsets are finer-grained. One such popular tagset is the **Penn Treebank Tagset** which consists of 36 POS tags.

Once the POS tagset is defined, we need to build a POS tagger. It's common to treat POS tagging as a classification task. First, build the POS tagger from a training set of documents that contains pre-tagged sentences. Then, test the POS tagger over a testing set of unseen documents.

The POS tags in the output include:

- CD Cardinal number
- DT Determiner
- IN Preposition
- JJ Adjective
- NN Noun
- NNP Proper noun, singular
- NNS Noun, plural
- VBD Verb, past tense

Reference:

Alphabetical list of part-of-speech tags used in the Penn Treebank Project:
https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

POS Tagging with NLTK

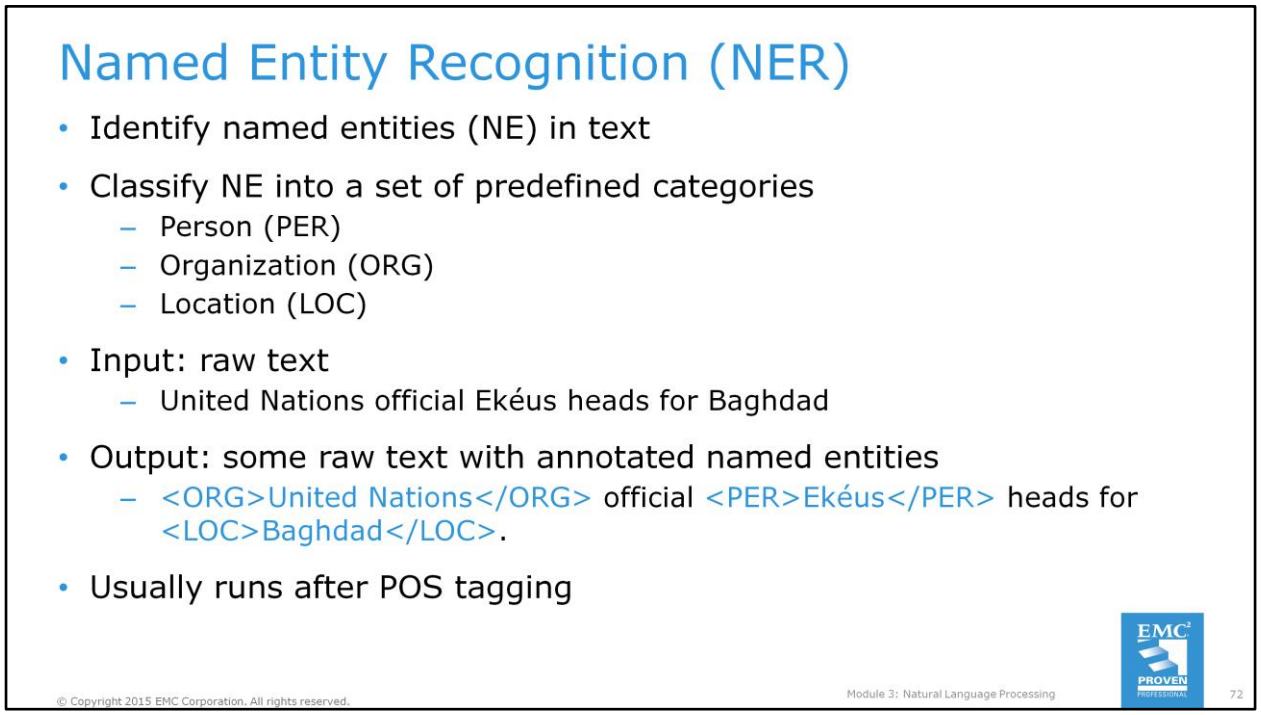
- Python NLTK has a built-in POS tagger `nltk.pos_tag()`
- Based on a naïve Bayes classifier

```
>>> import nltk
>>> text = nltk.word_tokenize('What is the air-speed velocity of an unladen
swallow')
>>> nltk.pos_tag(text)
[('What', 'WP'), ('is', 'VBZ'), ('the', 'DT'), ('air-speed', 'JJ'),
('velocity', 'NN'), ('of', 'IN'), ('an', 'DT'), ('unladen', 'JJ'), ('swallow',
'NN')]

>>> text = nltk.word_tokenize('He is content that the article has good
content')
>>> nltk.pos_tag(text)
[('He', 'PRP'), ('is', 'VBZ'), ('content', 'JJ'), ('that', 'IN'), ('the',
'DT'), ('article', 'NN'), ('has', 'VBZ'), ('good', 'JJ'), ('content', 'NN')]
```


NLTK has a built-in POS tagger that is based on a naïve Bayes classifier. First, tokenize the raw text into a list of tokens. Then, call the `nltk.pos_tag()` function to perform POS tagging.

For more information, readers can refer to Chapter 5 of the NLTK book, which is available at: http://www.nltk.org/book/ch05.html.



Named entities (NE) are words or phrases that contain names of persons, organizations and locations. Named entity recognition (NER) is a task that uses algorithms to identify named entities in raw text. It usually runs after the POS tagging.

For example, given raw text "United Nations official Ekéus heads for Baghdad" as the input, NER outputs the raw text with annotated NE -- United Nations is an organization, Ekéus is a person's name, and Baghdad is a location.

NER plays an important role in information extraction systems. Popular NER methods can be categorized into the following:

- **Supervised**: Probabilistic graphical models that requires training and testing. These methods often requires large annotated datasets and they are sensitive to the domains of the topics.

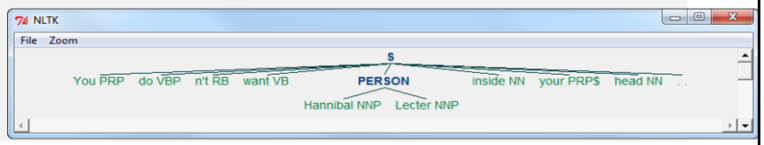
- **Semi-supervised**: Mutual bootstrap methods that propagate seed tuples by identifying extraction patterns in the text. Words are clustered by their meanings. Unknown words are assigned labels based on their similar and known words. Error may get accumulated and magnified after many iterations.

- **Unsupervised**: Annotated training data is not required. These methods use external resources like web queries or Wikipedia to identify named entities.

NER with NLTK

- NLTK's default NE chunker uses maximum entropy (MaxEnt)

```
>>> import nltk
>>> tokens = nltk.word_tokenize("You don't want Hannibal Lecter inside your head.")
>>> tokens = nltk.pos_tag(tokens)
>>> print(nltk.ne_chunk(tokens))
(S
  You/PRP
  do/VBP
  n't/RB
  want/VB
  (PERSON Hannibal/NNP Lecter/NNP)
  inside/NN
  your/PRP$
  head/NN
  ./.)
>>> tree = nltk.ne_chunk(tokens)
>>> tree.draw()
```



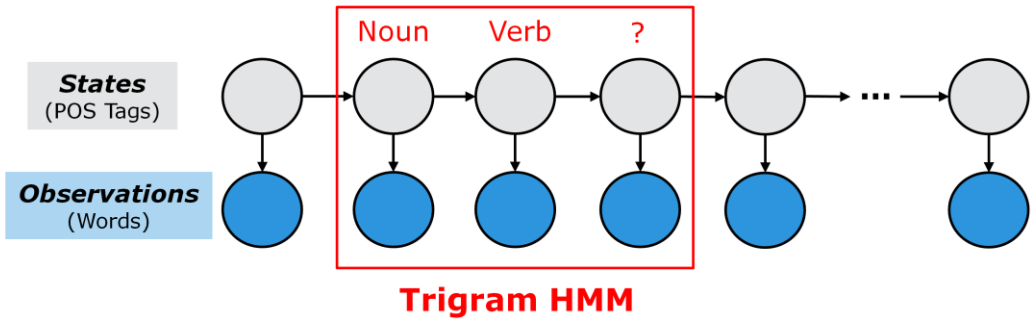
NLTK comes with a named entities (NE) chunker. The default NE chunker is a maximum entropy (MaxEnt), aka multinomial logistic regression, chunker trained on the ACE corpus (http://catalog.ldc.upenn.edu/LDC2005T09). Multinomial logistic regression will be covered in Module 5.

The slide demonstrates how to use the default NE chunker to perform NER. It is able to recognize Hannibal Lecter as a person's name.

Note that like any classifier, the naïve Bayes for POS tagging and the maximum entropy for NE chunking cannot guarantee a 100% accuracy. Therefore it's possible that the default POS tagger or NE chunker in NLTK may label the wrong POS tags, fail to recognize named entities, or label the wrong named entities. Other classifiers such as hidden Markov models (HMM) can be used instead of naïve Bayes or MaxEnt and receive a better performance on POS tagging and NER.

POS Tagging with Hidden Markov Models (HMM)

- Each observation is a word
- Each state can be viewed as a POS tag of the corresponding word



For Markov models such as first-order and second-order Markov processes, the output symbols are the same as the states, i.e., a one-to-one correspondence. But for POS tagging, the output symbols, that is, the observations, are the words and the states are the POS tags. This is not one-to-one correspondence, but many-to-many. We do not know what states we have given the observations. The goal is to defer the POS tags for a sequence of words. In other words, observations are what you already know and states are what you need to know.

A **hidden Markov model** (HMM) is an extension of a Markov model in which the observations are not the same as the states. HMM is a probabilistic language model. Each gray circle indicates a state. Each arrow connecting two states indicates their probabilistic dependency. Each state corresponds to an observation (blue circle). Using an HMM to do POS tagging is a special case of Bayesian inference.

A **trigram hidden Markov model** can be set up as a POS tagger. From a training set, the trigram HMM can estimate parameters such as the probability of seeing a certain tag r immediately after bigram tags (p,q), and the probability of observing a word x given a tag t. Such parameters can be used to identify the most likely sequence of tags for any given sentence in the testing set. The tag sequence can be found recursively by using an algorithm called the Viterbi algorithm.

Besides POS tagging, HMM can be used for other tasks such as named entity recognition, speech recognition, machine translation, and gene prediction.

HMM with NLTK

- The **`nltk.tag`** package includes a module **`nltk.tag.hmm`**
  - HiddenMarkovModelTagger
  - HiddenMarkovModelTrainer
- Need to specify tags and words (symbols and states)
- Train supervised or unsupervised
- Test the NLTK HMM module:

```
import nltk
# HMM probability calculation demo
nltk.tag.hmm.demo()
# HMM POS tagging demo with supervised training
nltk.tag.hmm.demo_pos()
```


HMM as well as the Viterbi algorithm are already provided by NLTK so you don't need to implement them. This slide shows how to import and use the `nltk.tag.hmm` module. You can find the API documentation on the `nltk.tag` package at: http://www.nltk.org/api/nltk.tag.html.

The HMM in NLTK can be trained supervised or unsupervised. If trained unsupervised, it uses the Baum-Welch algorithm to maximize the probability of the tag sequence. Baum-Welch algorithm works by assigning initial probabilities to all the parameters. Then until the training converges, it adjusts the probabilities of the HMM's parameters so as to increase the probability the model assigns to the training set.

The lab will show you how to use the HMM provided by NLTK to perform POS tagging.

Lab Exercise 9: Part-of-Speech Tagging

- This lab introduces you to perform part-of-speech (POS) tagging using NLTK's built-in functions.

- After completing the tasks in this lab you should be able to:
  - Perform POS tagging with a
    - unigram tagger
    - trigram tagger
    - HMM trigram tagger
  - Compare the performances of these taggers with measures such as precision, recall, and F1-score

Check Your Knowledge

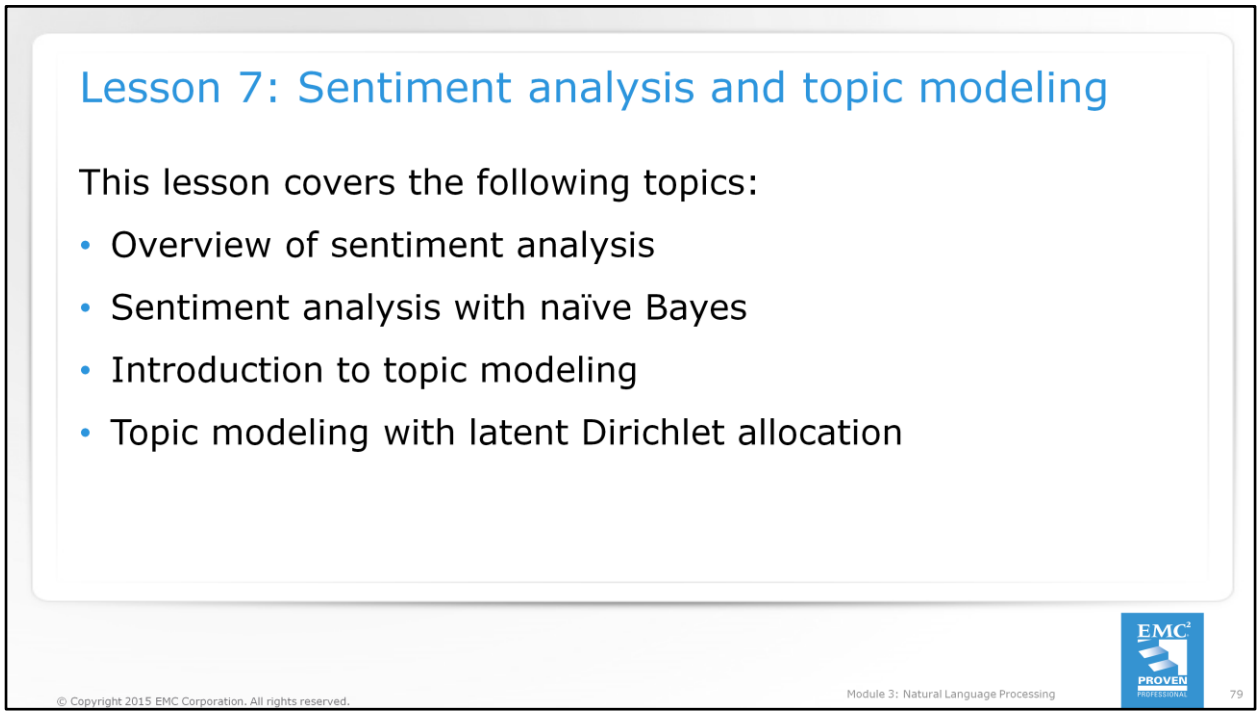
- What is POS tagging?
- What is named entity recognition?
- What named entities does NER usually recognize?
- How does HMM work as a POS tagger?

Write your answers here.

Lesson 6: Summary

During this lesson the following topics were covered:

- Part-of-speech tagging
- Named entity recognition
- Hidden Markov models as taggers



In this lesson, we will discuss sentiment analysis and topic modeling.

Sentiment Analysis: An Intuitive Approach

- Uses statistics and NLP to identify and extract subjective information from texts
- Consider movie reviews
  - Collect from websites like IMDb or Rotten Tomatoes
  - Detect if the reviewers liked the movies
- Early research
  - Used words that may be indicative of sentiments
  - Accuracy around 60%

	Proposed word lists	Accuracy	Ties
Human 1	positive: *dazzling, brilliant, phenomenal, excellent, fantastic* negative: *suck, terrible, awful, unwatchable, hideous*	58%	75%
Human 2	positive: *gripping, mesmerizing, riveting, spectacular, cool, awesome, thrilling, badass, excellent, moving, exciting* negative: *bad, cliched, sucks, boring, stupid, slow*	64%	39%

Sentiment analysis refers to a group of tasks that use statistics and natural language processing to mine opinions in order to identify and extract subjective information from texts.

Early work on sentiment analysis focused on detecting the polarity of product reviews from Epinions and movie reviews from the Internet Movie Database (IMDb) at the document level. Later work concentrated on sentiment analysis at the sentence level. More recently, the focus has shifted to phrase-level and short-text forms in response to the popularity of micro-blogging services such as Twitter.

Intuitively, to conduct sentiment analysis, one can manually construct lists of words with positive sentiments, such as brilliant, awesome, and spectacular, and negative sentiments, such as awful, stupid, and hideous. Related work has pointed out that such an approach can be expected to achieve an accuracy of around 60%,[1] and it is likely to be outperformed by an examination of corpus statistics.[2]

In Pang et al.'s work, the authors asked two volunteers, Human 1 and Human 2, to compile a list of words they believed to have positive or negative sentiments. Next, the authors used these lists of words to classify if new documents exhibited positive or negative sentiments. The result showed that lists from Human 1 received an accuracy of 58% and lists from Human 2 received an accuracy of 64%. In addition, 75% of the time Human 1's word lists couldn't be used to tell if a review was positive or negative (a tie) and 39% the times Human 2's word lists couldn't be used to tell if a review was positive or negative.[1]

References:

[1] B. Pang, L. Lee, and S. Vaithyanathan (2002). "Thumbs Up? Sentiment Classification Using Machine Learning Techniques." Proceedings of *EMNLP,* pp. 79–86.

[2] B. Pang and L. Lee (2008). "Opinion Mining and Sentiment Analysis." *Foundations and Trends in Information Retrieval*, vol. 2, no. 1–2, pp. 1–135.

Sentiment Analysis as a Text Classification Task

- Many classifiers can be used
  - Maximum entropy (multinomial logistic regression), naïve Bayes, support vector machines, HMM, …
  - Can reach ~ 80% accuracy
  - Can be applied to unstructured data such as movie reviews or even tweets
- Treat it as regular supervised learning
  - Provide training and testing sets

Classification methods such as naïve Bayes, maximum entropy (MaxEnt, or multinomial logistic regression), and support vector machines (SVM) are often used to extract corpus statistics for sentiment analysis. Related research has found out that these classifiers can score around 80% accuracy[1,2,3] on sentiment analysis over unstructured data. One or more of such classifiers can be applied to unstructured data such as movie reviews or even tweets.

Depending on the classifier, it may require the data to be split into training and testing sets. As you probably learned from the previous course, a useful rule of the thumb for splitting data is to produce a training data set much bigger than the testing data set. For example, an 80/20 split would produce 80% of the data as the training set and 20% as the testing set.

Next, one or more classifiers are trained over the training set to learn the characteristics or patterns residing in the data. The sentiment tags in the testing data are hidden away from the classifiers. After the training, classifiers are tested over the testing set to infer the sentiment tags. Finally, the result is compared against the original sentiment tags to evaluate the overall performance of the classifier.

References:

[1] B. Pang, L. Lee and S. Vaithyanathan (2002). "Thumbs Up? Sentiment Classification Using Machine Learning Techniques." *Proceedings of EMNLP*, pp. 79–86.

[2] A. Go, R. Bhayani, and L. Huang, (2009). "Twitter Sentiment Classification Using Distant Supervision." *CS224N Project Report.* Stanford, pp. 1–12.

[3] A. Pak and P. Paroubek (2010). "Twitter as a Corpus for Sentiment Analysis and Opinion Mining." *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10),* pp. 19–21.

Sentiment Analysis with NLTK (1 of 3)

- 2,000 movie reviews collected by Pang et al. (EMNLP 2002)
  - Manually pre-tagged, 1,000 as positive and 1,000 as negative
  - Included in NLTK's **nltk.corpus** package

```
import nltk.classify.util
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import movie_reviews
from collections import defaultdict
import numpy as np

# define a 80/20 split for train/test
SPLIT = 0.8
# file IDs for the positive and negative reviews
posids = movie_reviews.fileids('pos')
negids = movie_reviews.fileids('neg')

def word_feats(words):
    feats = defaultdict(lambda: False)
    for word in words:
        feats[word] = True
    return feats
```

Code continues in the next slide

The movie review corpus by Pang et al. includes 2,000 movie reviews collected from the IMDb archive of the rec.arts.movies.reviews newsgroup. These movie reviews have been manually tagged into 1,000 positive reviews and 1,000 negative reviews.

The Python code shows how to use NLTK to perform sentiment analysis over the movie review corpus. Function `word_feats()` accepts a list of tokens (naming *`words`*) as the input parameter, and return a `defaultdict` *`feat`*, with every token in *`words`* as a key whose value is `True`.

Reference:

B. Pang, L. Lee and S. Vaithyanathan (2002). "Thumbs up? Sentiment classification using machine learning techniques." *Proceedings of EMNLP*, pp. 79-86.

Sentiment Analysis with NLTK (2 of 3)

```
posfeats = [(word_feats(movie_reviews.words(fileids=[f])), 'pos') for f in posids]
negfeats = [(word_feats(movie_reviews.words(fileids=[f])), 'neg') for f in negids]
cutoff = int(len(posfeats) * SPLIT)
trainfeats = negfeats[:cutoff] + posfeats[:cutoff]
testfeats = negfeats[cutoff:] + posfeats[cutoff:]

print 'Train on %d instances' % len(trainfeats)
print 'Test on %d instances' % len(testfeats)
classifier = NaiveBayesClassifier.train(trainfeats)
print 'Accuracy:', nltk.classify.util.accuracy(classifier, testfeats)

classifier.show_most_informative_features()

# prepare confusion matrix
pos = np.array([classifier.classify(fs) for (fs,l) in posfeats[cutoff:]])
neg = np.array([classifier.classify(fs) for (fs,l) in negfeats[cutoff:]])
print 'Confusion matrix:'
print '\t'*2, 'Predicted class'
print '-'*40
print '|\t %d (TP) \t|\t %d (FN) \t| Actual class' % (
          (pos == 'pos').sum(), (pos == 'neg').sum())
print '-'*40
print '|\t %d (FP) \t|\t %d (TN) \t|' % (
          (neg == 'pos').sum(), (neg == 'neg').sum())
print '-'*40
```


The rest of the Python program is shown on the slide. The *posfeats* stores the word features for all 1,000 positive reviews and the *negfeats* stores the word features for all 1,000 negative reviews. Value of the *cutoff* is 800.

The *trainfeats* includes the first 800 positive reviews and the first 800 negative reviews as the training set. The *testfeats* includes the last 200 positive reviews and the last 200 negative reviews as the testing set.

The *classifier* stores the naïve Bayes classifier that has been trained over the *trainfeats*. Function `nltk.classify.util.accuracy()` returns the accuracy of the classifier. The `show_most_informative_features()` method prints out a list of most informative words that contribute to either positive or negative sentiments. Method `classifier.classify()` uses the trained *classifier* to classify unseen data from the testing set.

Sentiment Analysis with NLTK (3 of 3)

- Program output

```
Train on 1600 instances
Test on 400 instances
Accuracy: 0.735
Most Informative Features
             outstanding = True              pos : neg    =     13.9 : 1.0
               insulting = True              neg : pos    =     13.7 : 1.0
              vulnerable = True              pos : neg    =     13.0 : 1.0
               ludicrous = True              neg : pos    =     12.6 : 1.0
             uninvolving = True              neg : pos    =     12.3 : 1.0
              astounding = True              pos : neg    =     11.7 : 1.0
                  avoids = True              pos : neg    =     11.7 : 1.0
             fascination = True              pos : neg    =     11.0 : 1.0
               animators = True              pos : neg    =     10.3 : 1.0
               affecting = True              pos : neg    =     10.3 : 1.0
Confusion matrix:
        Predicted class
----------------------------------------
|     195 (TP)      |     5 (FN)       | Actual class
----------------------------------------
|     101 (FP)      |     99 (TN)      |
----------------------------------------
```

Q: What are the precision, recall, and F1-score of this classifier?

Here we show the output of the previous Python program. The naïve Bayes classifier is trained over 1600 movie reviews and then tested over 400 reviews, receiving an accuracy of 0.735.

Some of the most informative positive words include *outstanding*, *vulnerable*, *astounding*, *avoids*, *fascination*, *animators* and *affecting*. Some of the most informative negative words include *insulting*, *ludicrous*, and *uninvolving*.

The confusion matrix shows the classifier receives 195 true positives, 5 false negatives, 101 false positives and 99 true negatives. We can see that there are many false positives.

Precision is defined as the percentage of positive classifications that are correctly classified as positive, that is, TP/(TP+FP).

Recall is the percentage of actual positives that are correctly classified as positive, that is, TP/(TP+FN).

The **F1-score** (or F-score) considers both precision and recall:

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

The F1-score can be viewed as a weighted average of the precision and recall, and it reaches its best value at 1 and worst value at 0.

For the naïve Bayes classifier we've built for sentiment analysis, the precision is 195/(195+101)≈0.659, the recall is of 195/(195+5)=0.975, and the F1-score is 2*0.659*0.975/(0.659+0.975) ≈ 0.786.

Sentiment Analysis: Cautions

- Classifiers determine sentiments solely based on the datasets on which they are trained
  - Be aware of domain shift!
    - For example, "unpredictable" could be a good thing for movies, but not for dishwashers
  - Receiving high accuracy on one testing set doesn't guarantee high accuracy on another testing set
- Options to enhance the analysis
  - Include data from the new domain in the training set
  - Use emoticons such as :) and :(
    - Unsupervised or semi-supervised
  - Crowdsource the tedious "tagging" work (Amazon Mechanical Turk)

The movie review corpus only contains 2,000 reviews, and therefore it is relatively easy to manually tag each review. For sentiment analysis based on larger amounts of streaming data such as millions or billions of tweets, it is less feasible to collect and construct data sets of tweets that are big enough or manually tag each of the tweets to train and test one or more classifiers. There are two popular ways to cope with this problem. The first way is to construct pre-tagged data[1,2] by applying supervision and use emoticons such as :) and :(to indicate if a tweet contains positive or negative sentiments. In addition, words from these tweets can be used as clues to classify the sentiments of future tweets. Go et al. used classification methods including naïve Bayes, MaxEnt and SVM over the training and testing datasets to perform sentiment classifications. Their demo is available at http://www.sentiment140.com.

Emoticons make it easy and fast to detect sentiments of millions or billions of tweets; however, using emoticons as the sole indicator of sentiments sometimes can be misleading as emoticons may not necessarily correspond to the sentiments in the accompanied text. For example, a tweet saying "I have an awful feeling I'm going to fail my exams :)" has the :) emoticon but the text does not express a positive sentiment.

One way to address this problem in related research is to use Amazon Mechanical Turk (MTurk) to collect human-tagged reviews. MTurk is a crowdsourcing Internet marketplace that enables individuals or businesses to coordinate the use of human intelligence to perform tasks that are difficult for computers to do. In many cases, MTurk has been shown to collect human input much faster compared to traditional channels such as door-to-door surveys. For example, you can publish a list of untagged tweets to MTurk as Human Intelligence Tasks (HITs), and ask human workers to tag each tweets as positive, neutral or negative. The result can be used to train one or more classifiers or test the performances of classifiers.

References:

[1] A. Go, R. Bhayani and L. Huang (2009). "Twitter sentiment classification using distant supervision." *CS224N Project Report*. Stanford, pp. 1-12.

[2] A. Pak and P. Paroubek (2010). "Twitter as a corpus for sentiment analysis and opinion mining." Proceedings of *the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, pp. 19-21.

Categorize Documents by Topics

- Can be achieved through clustering or classification
- **Topic modeling** is a more feasible and prevalent approach
- Process of topic modeling
  - Uncover the hidden topical patterns within a corpus
  - Annotate documents according to these topics
  - Use annotations to organize, search, and summarize texts
- **Topic:** a distribution over a fixed vocabulary of words
- A simple topic model -- **Latent Dirichlet Allocation** (LDA)

Often, we want to categorize documents according to the topics. Document grouping can be achieved with clustering methods such as k-means clustering or classification methods such as naïve Bayes, support vector machines, or *k* nearest neighbors. However, a more feasible and prevalent approach is to use **topic modeling**. Topic modeling provides tools to automatically organize, search, understand, and summarize from vast amounts of information. **Topic models** are statistical models that examine words from a set of documents, determine the themes over the text, and discover how the themes are associated or change over time. The process of topic modeling can be simplified to the following:

1. Uncover the hidden topical patterns within a corpus.

2. Annotate documents according to these topics.

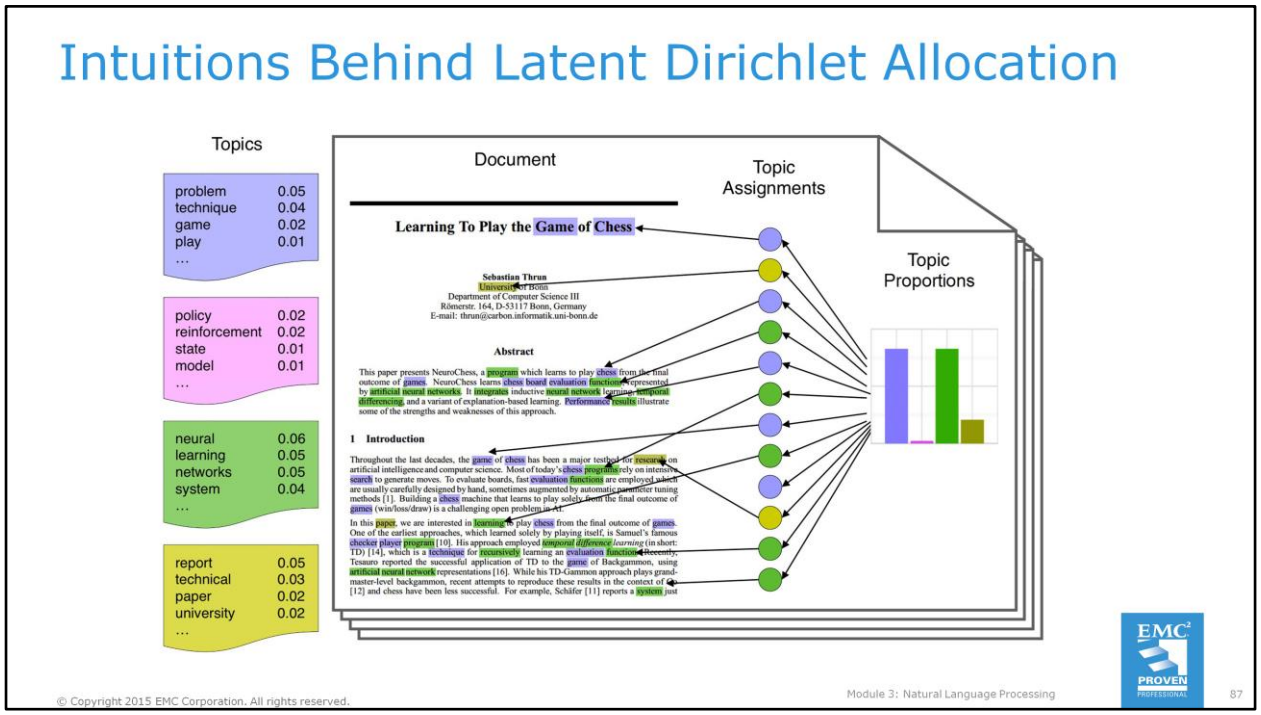
3. Use annotations to organize, search, and summarize texts.

A **topic** is formally defined as a distribution over a fixed vocabulary of words. Different topics have different distributions over the same vocabulary. A topic can be viewed as a cluster of words with related meanings, and each word has a corresponding weight inside this topic. Note that a word from the vocabulary can reside in multiple topics with different weights. Topic models do not necessarily require prior knowledge of the texts. The topics can emerge solely based on analyzing the text.

The simplest topic model is **latent Dirichlet allocation** (LDA), a generative probabilistic model of a corpus proposed by David M. Blei and two other researchers. In generative probabilistic modeling, data is treated as the result of a generative process that includes hidden variables. LDA assumes that there is a fixed vocabulary of words, and that the number of the latent topics is predefined and remains constant. LDA assumes that each latent topic follows a Dirichlet distribution over the vocabulary, and each document is represented as a random mixture of latent topics.

Reference:

D. M. Blei (2012). "Probabilistic Topic Models." *Communications of the ACM,* vol. 55, no. 4, pp. 77–84.



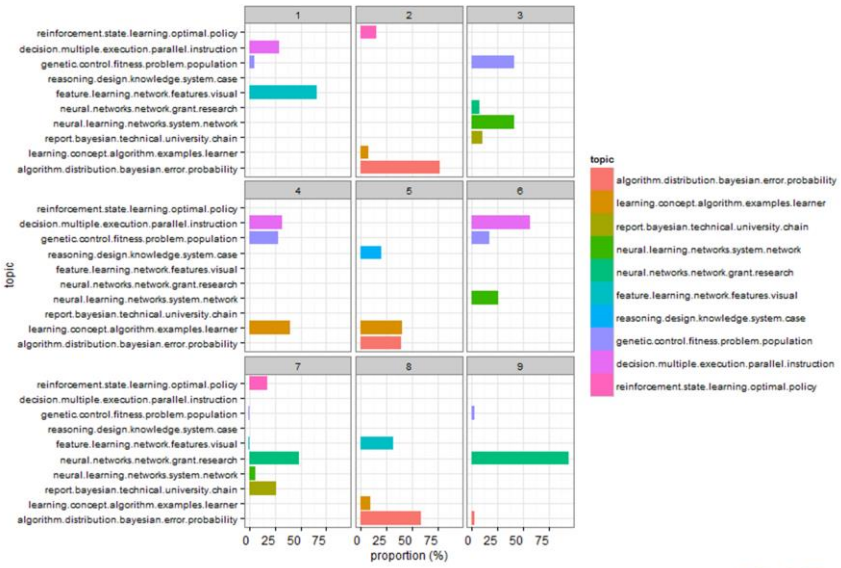
The figure illustrates the intuitions behind LDA. The left side of the figure shows four topics built from a corpus, where each topic contains a list of the most important words from the vocabulary. The four example topics are related to problem, policy, neural, and report. For each document, a distribution over the topics is chosen, as shown in the histogram on the right. Next, a topic assignment is picked for each word in the document, and the word from the corresponding topic (colored discs) is chosen. In reality, only the documents (as shown in the middle of the figure) are available. The goal of LDA is to infer the underlying topics, topic proportions, and topic assignments for every document.

Reference:

D. M. Blei, A. Y. Ng, and M. I. Jordan (2003), "Latent Dirichlet Allocation." *Journal of Machine Learning Research,* vol. 3, pp. 993–1022.

LDA in R

- Many programming tools provide LDA
  - R has a `lda` package
- The graph shows 10 topics on 9 scientific documents randomly drawn from a dataset



Many programming tools provide software packages that can perform LDA over datasets. R comes with an `lda` package that has built-in functions and sample datasets. The `lda` package was developed by David M. Blei's research group. This figure shows the distributions of ten topics on nine scientific documents randomly drawn from the `cora` dataset of the `lda` package. The `cora` dataset is a collection of 2,410 scientific documents extracted from the Cora search engine. The figure was generated by the following R code.

```
require("ggplot2")
require("reshape2")
require("lda")

# load documents and vocabulary
data(cora.documents)
data(cora.vocab)
theme_set(theme_bw())

# Define number of topic clusters and number of documents to display
K <- 10
N <- 9

result <- lda.collapsed.gibbs.sampler(cora.documents, K, cora.vocab, 25, 0.1,
                                      0.1, compute.log.likelihood=TRUE)

# Get the top words in the cluster
top.words <- top.topic.words(result$topics, 5, by.score=TRUE)

# build topic proportions
topic.props <- t(result$document_sums) / colSums(result$document_sums)
document.samples <- sample(1:dim(topic.props)[1], N)
topic.props <- topic.props[document.samples,]
topic.props[is.na(topic.props)] <-  1 / K
colnames(topic.props) <- apply(top.words, 2, paste, collapse=" ")

topic.props.df <- melt(cbind(data.frame(topic.props), document=factor(1:N)),
                       variable.name="topic", id.vars = "document")

qplot(topic, value*100, fill=topic, stat="identity",
      ylab="proportion (%)", data=topic.props.df,
      geom="histogram") +
  theme(axis.text.x = element_text(angle=0, hjust=1, size=12)) +
  coord_flip() +
  facet_wrap(~ document, ncol=3)
```

Check Your Knowledge

- What is sentiment analysis? How is naïve Bayes used for sentiment analysis?
- How can emoticons be used for sentiment analysis?
- What is topic modeling?
- What is the name of a simple topic model?

Further Reading

- Michael Collins (2013). Language Modeling. *Course notes for NLP*. http://www.cs.columbia.edu/~mcollins/lm-spring2013.pdf
- Daniel Jurafsky and James H. Martin (2009). *Speech and Language Processing 2nd Edition*. Upper Saddle River: Prentice-Hall, Inc. http://www.cs.colorado.edu/~martin/slp.html
- Christopher D. Manning and Hinrich Schütze (1999). *Foundations of Statistical Natural Language Processing*. Cambridge: MIT Press. http://nlp.stanford.edu/fsnlp/
- Steven Bird, Ewan Klein, and Edward Loper (2009). *Natural Language Processing with Python 1st Edition*. O'Reilly Media, Inc. http://www.nltk.org/book/
- Jacob Perkins (2010). *Python Text Processing with NLTK 2.0 Cookbook*. Packt Publishing. https://www.packtpub.com/application-development/python-text-processing-nltk-20-cookbook

Lesson 7: Summary

During this lesson the following topics were covered:

- Overview of sentiment analysis
- Sentiment analysis with naïve Bayes
- Introduction to topic modeling
- Topic modeling with latent Dirichlet allocation

Module 3: Summary

Key points covered in this module:

- Recognize different methods of Text Preprocessing
- Apply language modeling techniques
- Perform Sentiment Analysis with NLTK
- Construct a Natural Language Processing application

This slide intentionally left blank.