Basic NLP Concepts

Common misunderstandings

- How keyword analysis, n-grams, co-occurrence, stemming, and other techniques from a previous generation of NLP tools are no longer the best approaches to use
- That NLP work leading into AI applications is either fully automated or something which requires a huge amount of manual work
- NLP requires Big Data tooling and use of clusters

So, before we fully vest ourselves into neural network methods for NLP it is worthwhile to review some of the above traditional NLP concepts and methods.

- We will touch on
  - Keyword analysis, n-grams, stemming
  - How statistical parsing works
  - How resources such as WordNet enhance text mining
  - How to extract more than just a list of keywords from a text
  - How to summarize and compare a set of documents

Some of the steps in the above tasks are

- Prepare texts for parsing, e.g., how to handle difficult Unicode
- Parse sentences into annotated lists, structured as JSON output
- Perform keyword ranking using TF-IDF, while filtering stop words
- Calculate a Jaccard/cosine similarity measure to compare texts
- Leverage probabilistic data structures to perform the above more efficiently

As an example of what might be required in the first bullet above, let's look at a raw HTML document:

Extracting text from HTML

Suppose we have an HTML document organized like the following (see Appendix 2 for HTML syntax):

```
<!DOCTYPE html>
<html lang="en">
<head>
 <title>Foo Bar</title>
</head>
<body>
<div id="article-body">
<p>Nullum gratuitum prandium. Phasellus dictum urna sed metus
aliquet, quis vehicula ex rhoncus. In ante urna, imperdiet in
placerat non, elementum a libero.</p>
<p>Nam eu sem metus. Interdum et malesuada fames ac ante ipsum
primis in faucibus. Quisque et hendrerit massa.</p>
</div>
<div id="boiler-plate">
<p>Copyright ©2015 Fuberz. All rights reserved.</p>
</div>
</body>
</html>
```

Extracting text from HTML

We want the text in the **<p/>** elements enclosed by the ** ** element with **id="article-body"**

```
<!DOCTYPE html>
<html lang="en">
<head>
 <title>Foo Bar</title>
</head>
<body>
<div id="article-body">
<p>Nullum gratuitum prandium. Phasellus dictum urna sed metus
aliquet, quis vehicula ex rhoncus. In ante urna, imperdiet in
placerat non, elementum a libero.</p>
<p>Nam eu sem metus. Interdum et malesuada fames ac ante ipsum
primis in faucibus. Quisque et hendrerit massa.</p>
</div>
<div id="boiler-plate">
<p>Copyright ©2015 Fuberz. All rights reserved.</p>
</div>
</body>
</html>
```

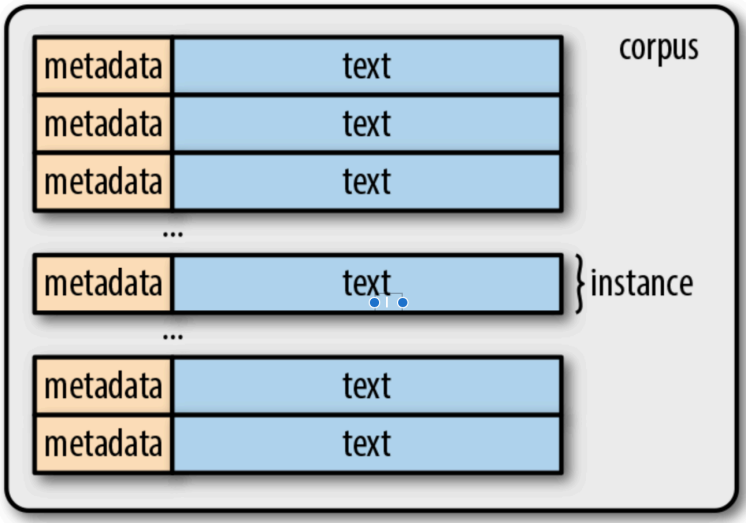
The steps to extract the text and process it further are in the notebook. We will continue with the general notions steps usually followed in NLP tasks.

Corpora, Tokens, and Types

- All NLP methods, be they classic or modern, begin with a text dataset, also called a corpus (plural: corpora).

- A corpus usually contains raw text (in ASCII or UTF-8) and any metadata associated with the text. The raw text is a sequence of characters (bytes), but most times it is useful to group those characters into contiguous units called tokens. In English, tokens correspond to words and numeric sequences separated by white-space characters or punctuation.

- The metadata could be any auxiliary piece of information associated with the text, like identifiers, labels, and timestamps. In machine learning parlance, the text along with its metadata is called an instance or data point. The corpus as collection of instances (i.e. dataset).



The process of breaking a text down into tokens is called tokenization.

For example, there are six tokens in the Esperanto sentence “Maria frapis la verda sorĉistino.”

Tokenizing tweets involves preserving hashtags and @handles, and segmenting smilies such as :-) and URLs as one unit.

Tokenization

- This is the first step in an NLP pipeline, so it can have a big impact on the rest of your pipeline.

- A tokenizer breaks unstructured data, natural language text, into chunks of information that can be counted as discrete elements. These counts of token occurrences in a document can be used directly as a vector representing that document. This immediately turns an unstructured string (text document) into a numerical data structure suitable for machine learning.

- Types are unique tokens present in a corpus

    - The set of all types in a corpus is its vocabulary or lexicon.
    - Words can be distinguished as content words and stopwords. Stopwords such as articles and prepositions serve mostly a grammatical purpose, like filler holding the content words.

Unigrams, Bigrams, Trigrams, ..., N-grams

- N-grams are fixed length (n) consecutive token sequences occurring in the text. A bigram has two tokens, a unigram one. Generating n-grams from a text is straightforward enough.

- N-grams enrich the vocabulary you build and help retain some of the word order information in a document.

Lemmas and Stems

- Sometimes, it might be useful to reduce the tokens to their lemmas to keep the dimensionality of the vector representation low. This reduction is called lemmatization
- Lemmas are root forms of words.
- Consider the verb fly. It can be inflected into many different words —flow, flew, flies, flown, flowing, and so on—and fly is the lemma for all of these seemingly different words.
- Stemming is poor-man's lemmatization. It involves use of handcrafted rules to strip endings of words to reduce them to a common form called stems

- Popular stemmer implemented in open source packages is the Porter Stemmer (Python implementation: https://github.com/jedijulia/porter-stemmer/blob/master/stemmer.py)

- Lemmatization is a potentially more accurate way to normalize a word than stemming or case normalization (e.g. all lower case) because it takes into account a word's meaning.

- A lemmatizer uses a knowledge base of word synonyms and word endings to ensure that only words that mean similar things are consolidated into a single token.

- WordNet® is a large lexical database of English which helps the lemmatizers. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets (n=117K) are interlinked by means of conceptual-semantic and lexical relations.

- Some lemmatizers use the word's part of speech (POS) tag in addition to its spelling to help improve accuracy. The POS tag for a word indicates its role in the grammar of a phrase or sentence.

Lexeme	PoS	Stem	Lemma
interact	VB	interact	interact
comments	NNS	comment	comment
provides	VBZ	provid	provide

- One example of part-of-speech (POS) "parser" is the Penn Treebank which POS tags are given on next page

- PoS from spaCy (with tags and their meaning): https://github.com/explosion/spaCy/blob/master/spacy/glossary.py

Tag	Description	Example	Tag	Description	Example
CC	Coordin. Conjunction	*and, but, or*	SYM	Symbol	+,%, &
CD	Cardinal number	*one, two, three*	TO	"to"	*to*
DT	Determiner	*a, the*	UH	Interjection	*ah, oops*
EX	Existential 'there'	*there*	VB	Verb, base form	*eat*
FW	Foreign word	*mea culpa*	VBD	Verb, past tense	*ate*
IN	Preposition/sub-conj	*of, in, by*	VBG	Verb, gerund	*eating*
JJ	Adjective	*yellow*	VBN	Verb, past participle	*eaten*
JJR	Adj., comparative	*bigger*	VBP	Verb, non-3sg pres	*eat*
JJS	Adj., superlative	*wildest*	VBZ	Verb, 3sg pres	*eats*
LS	List item marker	*1, 2, One*	WDT	Wh-determiner	*which, that*
MD	Modal	*can, should*	WP	Wh-pronoun	*what, who*
NN	Noun, sing. or mass	*llama*	WP$	Possessive wh-	*whose*
NNS	Noun, plural	*llamas*	WRB	Wh-adverb	*how, where*
NNP	Proper noun, singular	*IBM*	$	Dollar sign	*$*
NNPS	Proper noun, plural	*Carolinas*	#	Pound sign	*#*
PDT	Predeterminer	*all, both*	"	Left quote	' or "
POS	Possessive ending	*'s*	"	Right quote	' or "
PRP	Personal pronoun	*I, you, he*	(	Left parenthesis	[, (, {, <
PRP$	Possessive pronoun	*your, one's*	)	Right parenthesis	],), }, >
RB	Adverb	*quickly, never*	,	Comma	,
RBR	Adverb, comparative	*faster*	.	Sentence-final punc	. ! ?
RBS	Adverb, superlative	*fastest*	:	Mid-sentence punc	: ; ... – -
RP	Particle	*up, off*			

Figure 5.6 Penn Treebank part-of-speech tags (including punctuation).

Categorizing Spans: Chunking and Named Entity Recognition

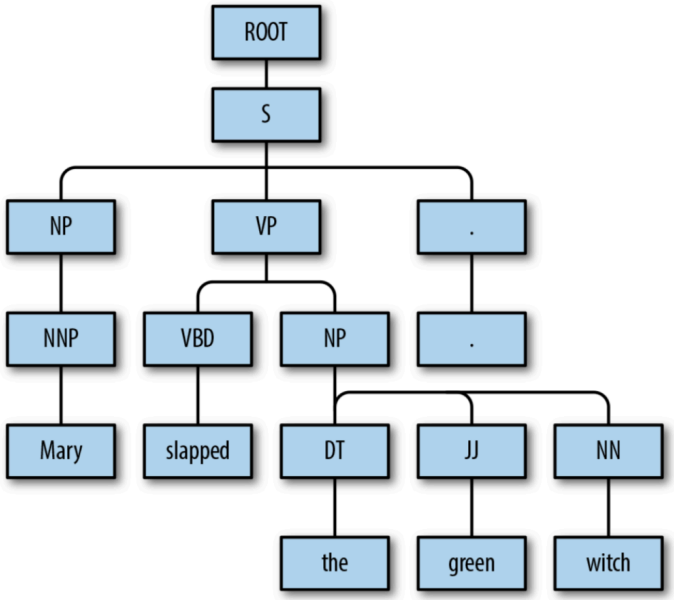
- Often, we need to label a span of text; that is, a contiguous multi-token boundary. For example, consider the sentence, "Mary slapped the green witch." We might want to identify the noun phrases (NP) and verb phrases (VP) in it, as shown here:

```
[NP Mary] [VP slapped] [the green witch].
```

- Another type of span that's useful is the named entity. A named entity is a string mention of a real world concept like a person, location, organization, drug name, and so on. Here's an example:

Structure of Sentences

- Whereas shallow parsing identifies phrasal units, the task of identifying the relationship between them is called parsing. You might recall from elementary English class diagramming sentences like in this example



For Deeper Understanding read in Jurafsky & Martin's book

Chapter 2 (Finite Automata & Regular Expressions)
Chapter 3: 3.1-3.3, 3.8-3.9 (Morphology/Parsing)
Chapter 4: 4.1-4.2 (N-grams)
Chapter 5: 5.1-5.5 (PoS, Penn Treebank)

Chapter 12: 12.1-12.3.2 (Context Free Grammars, Parsing)
Chapter 13: 13.1-13.4, 13.5.1
Chapter 14: 14.1-14.4 (Statistical Parsing)

Chapter 17: 17.1-17.3 (Semantics)
Chapter 19: 19.1-19.4
Chapter 20: 20.1-20.5

Chapter 22.1, 23.1-23.3 (Applications such as Information Retrieval)

Regular expressions (RE)

- Regular expressions use a special kind of formal language grammar called a regular grammar. Regular grammars have predictable, provable behavior, and yet are flexible enough to power some of the most sophisticated dialog engines and chatbots on the market.

- Amazon Alexa and Google Now are mostly pattern-based engines that rely on regular grammars. Deep, complex regular grammar rules can often be expressed in a single line of code called a regular expression.

Quick Review of RE

- The search string can consist of a single character (like /!/) or a sequence of characters (like /urgl/); The *first* instance of each match to the regular expression is underlined below (although a given application might choose to return more than just the first instance):

RE	Example Patterns Matched
`/woodchucks/`	"interesting links to woodchucks and lemurs"
`/a/`	"Mary Ann stopped by Mona's"
`/Claire␣says,/`	"Dagmar, my gift please," Claire says,"
`/DOROTHY/`	"SURRENDER DOROTHY"
`/!/`	"You've left the burglar behind again!" said Nori

- Regular expressions are case sensitive. To help searches where letter can appear as both lower and upper case use [and]:

RE	Match	Example Patterns
`/[wW]oodchuck/`	Woodchuck or woodchuck	"Woodchuck"
`/[abc]/`	'a', 'b', *or* 'c'	"In uomini, in soldati"
`/[1234567890]/`	any digit	"plenty of 7 to 5"

Figure 2.1 The use of the brackets `[ ]` to specify a disjunction of characters.

RE	Match	Example Patterns Matched
`/[A-Z]/`	an uppercase letter	"we should call it 'Drenched Blossoms'"
`/[a-z]/`	a lowercase letter	"my beans were impatient to be hoed!"
`/[0-9]/`	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.2 The use of the brackets `[ ]` plus the dash `-` to specify a range.

RE	Match (single characters)	Example Patterns Matched
`[^A-Z]`	not an uppercase letter	"Oyfn pripetchik"
`[^Ss]`	neither 'S' nor 's'	"I have no exquisite reason for't"
`[^\.]`	not a period	"our resident Djinn"
`[e^]`	either 'e' or '^'	"look up ^ now"
`a^b`	the pattern 'a^b'	"look up a^ b now"

Figure 2.3 Uses of the caret `^` for negation or just to mean `^`.

RE	Match	Example Patterns Matched
`woodchucks?`	woodchuck or woodchucks	"woodchuck"
`colou?r`	color or colour	"colour"

Figure 2.4 The question-mark `?` marks optionality of the previous expression.

- Sometimes we need regular expressions that allow repetitions of things. For example, consider the language of (certain) sheep, which consists of strings that look like the following:

baa!
baaa!
baaaa!
baaaaa!
baaaaaa!
. . .

- This language consists of strings with a *b*, followed by at least two *a*s, followed by an exclamation point. The set of operators that allow us to say things like "some number of *a*s" are based on the asterisk or *, commonly called the **Kleene *** (pronounced "cleany star") and means "zero or more occurrences of the immediately previous character or regular expression".
- So /a*/ means "any string of zero or more *a*s". This will match *a* or *aaaaaa* but it will also match *Off Minor*, since the string *Off Minor* has zero *a*s. So the regular expression for matching one or more *a* is /aa*/
- Thus one way to specify the sheep language is: /baaa*!/

There is more serious reason to consider RE:

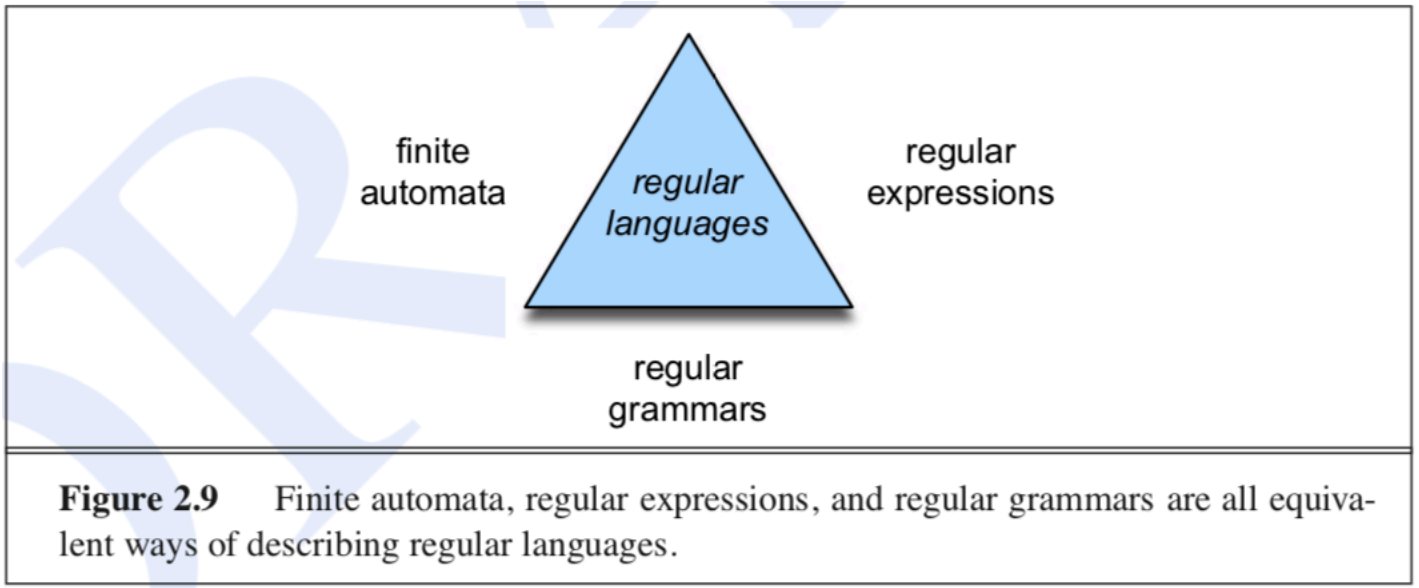


Figure 2.9 Finite automata, regular expressions, and regular grammars are all equivalent ways of describing regular languages.

More in Appendix 3 and Jurafsky & Martin …

Some NLP tasks

- Categorizing or classifying documents is probably one of the earliest applications of NLP. The TF and TF-IDF representations are described below and are immediately useful for classifying and categorizing longer chunks of text such as documents or sentences.
- The TF portion considers how frequently a term appears in a document, normalized by the DF portion which considers how frequently terms appear across all documents. Using the following notation

  t – term (word), d – document, D – corpus (all documents)

  $TF(t,d)$ = (# of times term t appears in document d) / (Total # of terms in document d)

  $IDF(t,D)$ = log(Total # of documents / # of documents with term t in it) = $log(|D|/DF(t,D))$

$$=> \ \mathrm{TF-IDF}(t,d,D) = TF(t,d) * IDF(t,D)$$

Example. Consider a document containing 100 words wherein the word *cat* appears 3 times. The term frequency (i.e., TF) for *cat* is then (3 / 100) = 0.03. Now, assume we have 10 million documents and the word *cat* appears in 1,000 of these. Then, the inverse document frequency (i.e., IDF) is calculated as $\log_{10}(10{,}000{,}000 \,/\, 1{,}000) = 4$. Thus, the $TF - IDF$ weight is the product of these quantities: $0.03 * 4 = 0.12$.

- TF-IDF can be used to calculate the similarity of text. By representing the documents with their TF-IDFs in a vector space and using for example a cosine distance (to compute similarity), we can compare documents
- It's fast and works well when documents are large and/or have lots of overlap.
- It looks for exact matches, so at the very least you should use a lemmatizer to take care of the plurals. When comparing short documents with limited-term variety — such as search queries — there is a risk that you will miss semantic relationships where there isn't an exact word match.

- Because of the last point, methods using word embeddings (gensim, GloVe – see Appendix 1) are used when overlap between texts is limited, such as if you need 'fruit and vegetables' to relate to 'tomatoes'.
- Methods for semantic similarity using word embeddings are more flexible but there is a lot more computation involved. This scales well, but running a single query is slow.
- Most words have some degree of similarity to other words, so almost all documents will have some non-zero similarity to other documents.
- Semantic similarity is good for ranking content in order, rather than making specific judgements about whether a document is or is not about a specific topic.

See the notebook for use of the above.

- Other NLP problems such as assigning topic labels, predicting sentiment of reviews, filtering spam emails, language identification, and email triaging can be framed as supervised document classification problems.

- Summarization is an NLP tasks which historically, has been handled in a few ways, including with DL (https://mike.place/2016/summarization/) One simple approach to do it with the means we mentioned above is

    - parse the text to create a feature vector
    - calculate the semantic similarity b/w the feature vector and each sentence
    - select the N top-ranked sentences, listed in order, to summarize

Appendix 1: Encoding Categorical Features (HOML, Ch 13)

Using One-Hot Vectors

- Consider the ocean_proximity feature in the California housing dataset - it is a categorical feature with 5 possible values: "<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", and "ISLAND". We need to encode this feature before we feed it to a neural network. Since there are very few categories, we can use one-hot encoding. For this, we first need to map each category to its index (0 to 4), which can be done using a lookup table:

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)

table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

- We first define the *vocabulary*: this is the list of all possible categories.
- Then we create a tensor with the corresponding indices (0 to 4).

- Next, we create an initializer for the lookup table, passing it the list of categories and their corresponding indices. In this example, we already have this data, so we use a KeyValueTensorInitializer; but if the categories were listed in a text file (with one category per line), we would use a TextFileInitializer instead.

- In the last two lines, we create the lookup table, giving it the initializer and specifying the number of *out-of-vocabulary* (oov) buckets. If we look up a category that does not exist in the vocabulary, the lookup table will compute a hash of this category and use it to assign the unknown category to one of the oov buckets.

Now, let's use the lookup table to encode a small batch of categorical features to one-hot vectors:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
>>> cat_one_hot
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
```

Using Embeddings

- An embedding is a trainable dense vector that represents a category. By default, embeddings are initialized randomly, so for example the "NEAR BAY" category could be represented initially by a random vector such as $[0.131, 0.890]$, while the "NEAR OCEAN" category might be represented by another random vector such as $[0.631, 0.791]$.

- In this example, we use 2D embeddings, but the number of dimensions is a hyperparameter you can tweak. Since these embeddings are trainable, they will gradually improve during training; and as they represent fairly similar categories, Gradient Descent will certainly end up pushing them closer together, while it will tend to move them away from the "INLAND" category's embedding (see Figure 13-4). Indeed, the better the representation, the easier it will be for the neural network to make accurate predictions, so training tends to make embeddings useful representations of the categories.

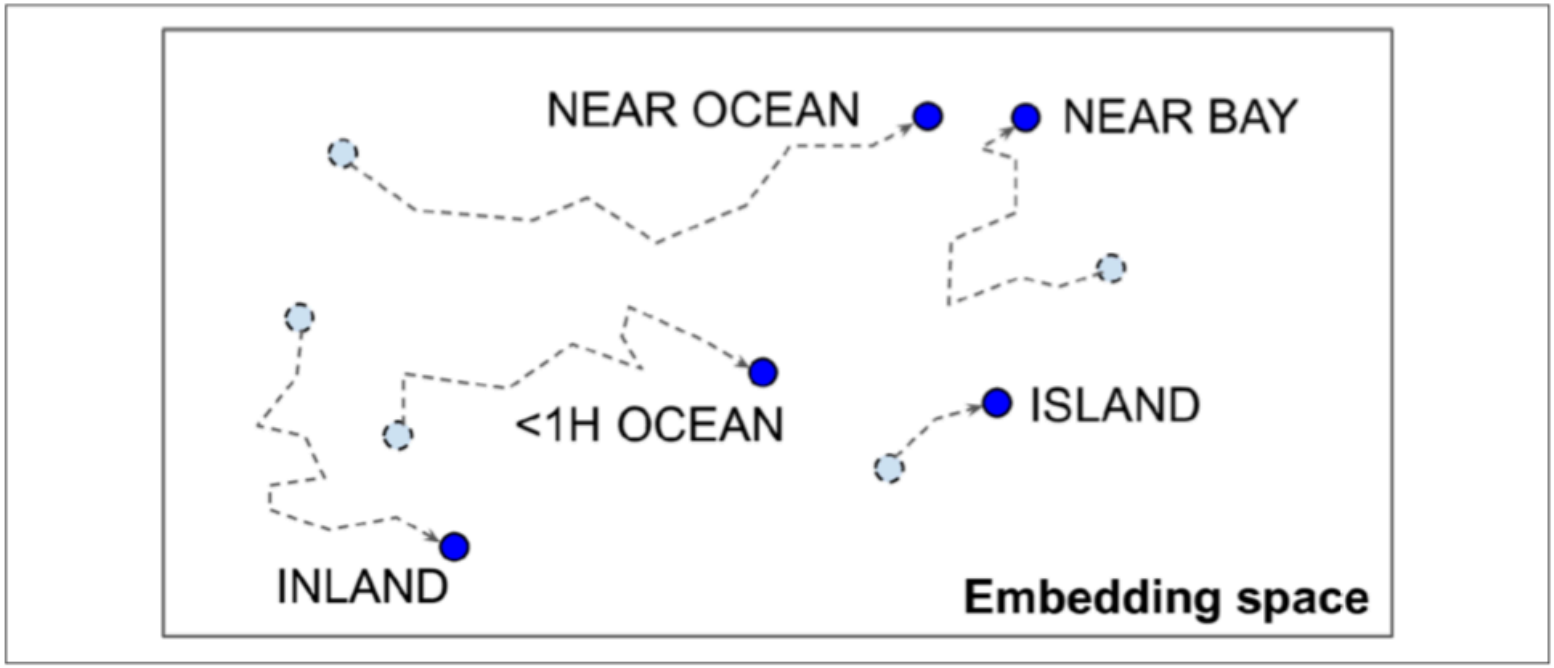


Figure 13-4. Embeddings will gradually improve during training

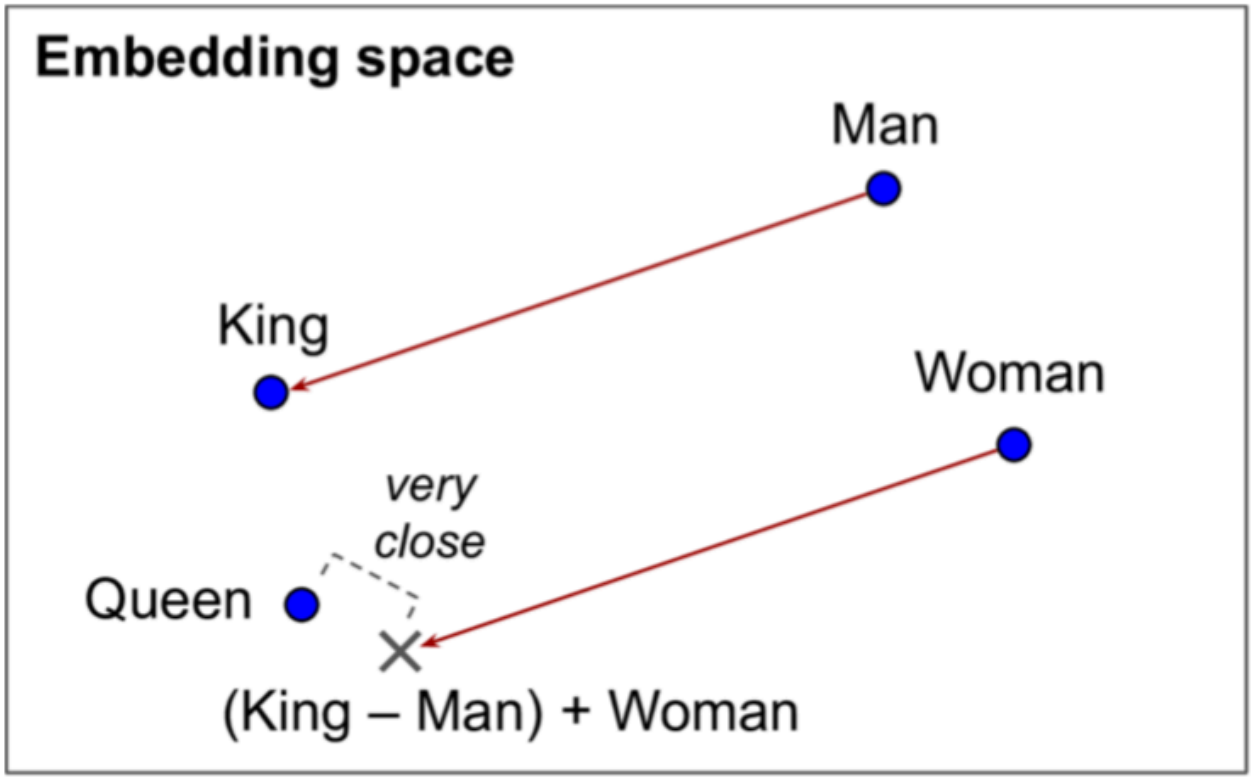


Figure 13-5. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts

Appendix 2:

Common HTML elements and attributes

- `<html>..</html>`—Demarcates the entire HTML document.
- `<title>..</title>`—The title of the document. This title appears in a web browser's title bar but not in the browser's rendered contents.
- `<head>..</head>`—The head of the document. The information in the head is not intended to appear in the browser's rendered contents.
- `<body>..</body>`—The body of the document. The information in the body is intended to appear in a browser's rendered contents.
- `<h1>..</h1>`—A header in the document. It is generally rendered in large, bold letters.
- `<h2>..</h2>`—A header in the document whose formatting slightly differs from `<h1>`.
- `<p>..</p>`—A single paragraph in the document.
- `<p id="unique_id">..</p>`—A single paragraph in the document containing a unique `id` attribute that is not shared by any other document elements.
- `<a href="http://www.manning.com">..</a>`—A clickable text hyperlink. Clicking the text sends a user to the URL specified in the `href` attribute.
- `<ul>..</ul>`—An unstructured list composed of individual list items that appear as bullet points in a browser's rendered contents.
- `<li>..</li>`—An individual list item in an unstructured list.
- `<div>..</div>`—A division demarcating a specific subsection of the document.
- `<div class="category_class">..</div>`—A division demarcating a specific subsection of the document. The division is assigned a category class by way of the `class` attribute. Unlike a unique ID, this class can be shared across other divisions in the HTML.

Appendix 3:

Formal mathematical explanation of formal languages

Kyle Gorman describes programming languages this way:

- Most (if not all) programming languages are drawn from the class of context-free languages.
- Context-free languages are parsed with context-free grammars, which provide efficient parsing.
- The regular languages are also efficiently parsable and used extensively in computing for string matching.
- String matching applications rarely require the expressiveness of context-free.
- There are a number of formal language classes, a few of which are shown here (in decreasing complexity):[a]
  - Recursively enumerable
  - Context-sensitive
  - Context-free
  - Regular

Natural languages:

- Are not regular[b]
- Are not context-free[c]
- Can't be defined by any formal grammar[d]

- [a] See the web page titled "Chomsky hierarchy - Wikipedia" (https://en.wikipedia.org/wiki/Chomsky_hierarchy).
- [b] "English is not a regular language" (http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=20) by Shuly Wintner.
- [c] "Is English context-free?" (http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=24) by Shuly Wintner.
- [d] See the web page titled "1.11. Formal and Natural Languages — How to Think like a Computer Scientist: Interactive Edition" (http://interactivepython.org/runestone/static/CS152f17/GeneralIntro/FormalandNaturalLanguages.html).