Cambridge Elements

Quantitative and Computational Methods for the Social Sciences

Text Analysis in Python for Social Scientists

Discovery and Exploration

Dirk Hovy

ISSN 2398-4023 (online)
ISSN 2514-3794 (print)

Cambridge Elements

Elements in Quantitative and Computational Methods for the Social Sciences
edited by
R. Michael Alvarez
California Institute of Technology
Nathaniel Beck
New York University

TEXT ANALYSIS IN PYTHON FOR SOCIAL SCIENTISTS

Discovery and Exploration

Dirk Hovy
Bocconi University

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre,
New Delhi – 110025, India

79 Anson Road, #06–04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781108819824
DOI: 10.1017/9781108873352

First published 2020

A catalogue record for this publication is available from the British Library.

ISBN 978-1-108-81982-4 Paperback
ISSN 2398-4023 (online)
ISSN 2514-3794 (print)

Text Analysis in Python for Social Scientists

Discovery and Exploration

Elements in Quantitative and Computational Methods for the Social Sciences

DOI: 10.1017/9781108873352
First published online: December 2020

Dirk Hovy
Bocconi University

Author for correspondence: Dirk Hovy, dirk.hovy@unibocconi.it

Abstract: Text is everywhere, and it is a fantastic resource for social scientists. However, because it is so abundant, and because language is so variable, it is often difficult to extract the information we want. A whole subfield of AI is concerned with text analysis (natural language processing). Many of the basic analysis methods developed are now readily available as Python implementations. This Element will teach you when to use which method, the mathematical background of how it works, and the Python code to implement it.

Keywords: text analysis; natural language processing; computational linguistics; exploration
JEL classifications: A12, B34, C56, D78, E90

ISBNs: 9781108819824 (PB), 9781108873352 (OC)
ISSNs: 2398-4023 (online), 2514-3794 (print)

Contents

Introduction

Today, text is an integral part of our lives and one of the most abundant sources of information. On an average day, we read about 9000 words, comprising emails, text messages, the news, blog posts, reports, tweets, and things like street names and advertisements. Throughout a lifetime of reading, this gets us to about 200 million words. It sounds impressive (and it is), and yet, we can store this amount of information in less than half a gigabyte: we could carry a life's worth of reading around on a USB stick. At the time of writing this, the internet contains an estimated minimum of more than 1200 TB of text, or 2.5 million lives' worth of reading. A large part of that text is now in the form of social media: microblogs, tweets, Facebook statuses, Instagram posts, online reviews, LinkedIn profiles, YouTube comments, and many others. However, text is abundant even offline – in quarterly earnings reports, patent filings, questionnaire responses, written correspondence, song lyrics, poems, diaries, novels, parliamentary proceedings, meeting minutes, and thousands of other forms that can be (and are) used in social science research and data mining.

Text is an excellent source of information not just because of its scale and availability. It is also (relatively) permanent, and – most importantly – it encodes language. This one human faculty reflects (indirectly and sometimes even directly) a wide range of sociocultural and psychological constructs: trust, power, beliefs, fears. Text analysis has consequently been used to measure sociocultural constructs such as trust (Niculae, Kumar, Boyd-Graber, & Danescu-Niculescu-Mizil, 2015) and power (Prabhakaran, Rambow, & Diab, 2012). Language encodes the age, gender, origin, and many other demographic factors of the author (Labov, 1972; Pennebaker, 2011; Trudgill, 2000). Text can therefore be used to measure the attitudes of society toward those target concepts over time (see Garg, Schiebinger, Jurafsky, & Zou, 2018; Hamilton, Leskovec, & Jurafsky, 2016; Kulkarni, Al-Rfou, Perozzi, & Skiena, 2015).

However, this avalanche of great data can quickly become overwhelming, and dealing with it can feel daunting. Text is often called **unstructured data**, meaning it does not come in a spreadsheet, neatly ordered into categories. It has varying lengths and can not readily be fed into your favorite statistical analysis tool without formatting it first. As we will see, though, "unstructured" is a bit of a misnomer. Text is by no means without any structure – it follows very regular structures, governed by syntactic rules. And if you know about these, making sense of text becomes a lot easier.

There are even simpler regularities in text that we can explore to get information. From simple counts to probabilities, we can do a lot to weed out the noise and gather some insights into the data. And then there are the heavy lifters: machine learning tools that can crunch almost limitless amounts of data and extract precisely the information you want – well, as precisely as these tools can manage.

This Element is aimed at the working social scientist who wants to use text in their exploratory analysis. It is therefore first and foremost a practical Element, giving concrete examples. Each section contains Python 3.6 code using the latest available Python packages to execute the examples we will discuss. This Element assumes enough familiarity with Python to follow those examples. It does not serve as an introduction to programming.

Using off-the-shelf libraries, rather than coding the algorithms from first principles, strips away some of the complexity of the task, which makes it a lot more approachable. It also allows programming novices to use efficient code to try out a research idea, rather than spending days to make the code scale to the problem and to debug it. At the same time, using packages hides some of the inner workings of the algorithms. However, I would argue that we do not need to implement the algorithms from scratch to use these tools to their full potential. It is essential, though, to know how these tools work. Knowing what is going on under the hood helps us make the right choices and understand the possibilities and limitations. For each example, we will look at the general idea behind the algorithm as well as the mathematical underpinnings. If you would like to dive deeper into a particular technique, or need to modify it to suit your needs, I will provide pointers to detailed tutorials and background reading. All the code in this Element can be downloaded from https://github.com/dirkhovy/text_analysis_for_social_science.

We do not cover text classification in this Element, such as sentiment analysis and other supervised machine learning techniques. Those topics merit a separate treatment, especially in the light of recent advances in deep learning techniques for natural language processing (NLP).

NLP, the branch of AI that applies machine learning methods to text, is becoming ubiquitous in social sciences. There is a growing number of overview articles and tutorials on text analysis and in several disciplines. These works cover the most relevant terms and models and use cases for the respective fields. It makes sense to consult them to see what people in a specific field have been using. Some of the more recent examples include Grimmer and Stewart (2013) for political science, Evans and Aceves (2016) for sociology, Humphreys and Wang (2017) and Hartmann, Huppertz, Schamp, and Heitmann (2018) for marketing, and Gentzkow, Kelly, and Taddy (2017) for economics.

The best starting point from which to dive into NLP in all its depth and glory is the textbook by Jurafsky and Martin (2014), which is constantly updated (important for such a dynamic field). The book by Manning and Schütze (1999) is still a good complement, especially on the fundamentals, despite its age and lack of some of the latest machine learning approaches. The latest textbook is Eisenstein (2019). If you are interested in specific topics, you can search the anthology of the Association for Computational Linguistics (ACL), the official organization of most NLP research. All publications in this field are peer-reviewed conference proceedings (with acceptance rates around 20–25 percent), and the entire catalog of all papers ever published is available for free at www.aclweb.org/anthology/. The field borrows heavily from machine learning, especially deep learning. For an excellent overview of the most relevant deep learning methods in NLP, see the book by Goldberg (2017) or the freely available primer (Goldberg, 2016) on which it is based. To explore these topics in Python, the books by Marsland (2015) and Chollet (2017), are very practically oriented and beginner-friendly. For a very readable general overview of machine learning, see Murphy (2012).

This Element is structured into two halves. In the first half, we will look at some of the basic properties of text and language – the levels of linguistic analysis, grammatical and semantic components, and how to describe them. We will also discuss what to remove and what to keep for our analyses and how to compute simple, useful statistics.

In the second half, we will look at exploration, the discovery of latent structure in the data. We will go from simple statistics to more sophisticated machine learning methods, such as topic models, word embeddings, and dimensionality reduction.

BACKGROUND

1 Prerequisites

This Element assumes some basic familiarity with Python. However, it does not replace an introduction to programming. If you are looking to learn Python, or to refresh the basics, the official Python website lists a number of excellent online tutorials (https://wiki.python.org/moin/IntroductoryBooks) and textbooks (https://wiki.python.org/moin/BeginnersGuide/NonProgrammers).

To develop Python programs, one of the easiest environments is **Jupyter notebooks**, which allow for code, text, and images to be stored in a platform-independent way. It is an excellent way to visualize and store intermediate steps. To use these notebooks, you will have to install Anaconda Python and make sure it is running. Taking the following steps is the easiest way to do so:

1. Download Anaconda with Python 3.7 or higher for your operating system from www.anaconda.com/download/.
2. Install Anaconda and set it as your system-default Python.
3. Open the Jupyter Notebook app either by
   (a) opening the Anaconda Navigator app and clicking on "Jupyter Notebook"; or
   (b) opening a terminal window and typing "`jupyter notebook`."
4. This will give you a browser window with a file structure. Make sure you know where on your computer this folder is!
5. Open a new notebook window and copy the code from Code 1 into it.
6. Execute the code in the notebook by either clicking the Play button or hitting `SHIFT+ENTER`.

```
garble = 'C#LoBln{;gFZr>!a%LtVQu!(l&Ya!zt/;i iofKn: sg_,gh %
    gyGroUquJ` jeaY!r:-e]m $taIZlbCr "e;Fa{Yd,hy>] _xog*nb{ |Ky*
    PoY)uTKr&K %)wipa@ny{g QEtMloXy %apWsr,foO\'g+RryBaD\'mT-m*}
    iIrnE;gy=!>t'
print(''.join([garble[c] for c in range(0, len(garble), 3)]))
```

Code 1 Our first Jupyter Notebook code.

You should see an encouraging message if all is working correctly.

In order to work with text, we will use the following libraries:

1. **`spacy`** (including at least the English-language model)
2. **`nltk`** (including the data sets it provides)
3. **`gensim`** (for topic models)

The `spacy` library should come preinstalled in Anaconda, but you will have to download the models for various languages. You can check this by executing the following code in a notebook cell:

```
import spacy
spacy.load('en')
```

Code 2 Loading the English-language model of `spacy`.

If you get an error, install at least the English model (additional languages are available). See https://spacy.io/usage/models#section-install for details on how to do this.

You can install the gensim and nltk libraries through Anaconda – which we will need for embeddings and topic models – either through the Anaconda Navigator *or* through the terminal by executing the following lines:

```
conda install nltk
conda install gensim
```

In order to install the NLTK data sets, you can simply execute the instructions in Code 3 in a notebook:

```
import nltk
nltk.download('all')
```

Code 3 Downloading the data sets of `nltk`.

2 What's in a Word

Say we have collected a large number of companies' self-descriptions from their websites. We want to start finding out whether there are any systematic differences, what the big themes are, and how the companies typically act. The question is, *How do we do all that*? Does it matter that firms *buy*, *bought*, and *are buying* – or do we only want to know that there is any buying event, no matter when? Do we care about prepositions (e.g., *in*, *off*, *over*, *about*), or are they just distracting us from what we really want to know? How do we tell the program that "Facebook acquires Whatsapp" and "Whatsapp acquired by Facebook" mean the same thing, but "Whatsapp acquires Facebook" does not (even though it uses the same words as the first example)?

Before we dive into the applications, let's take a look at the subject we are working with: language. In this section, we will look at the terminology to describe some of its basic elements (morphology, syntax, and semantics) and their equivalents in text (characters, words, sentences). To choose the right method for any text-related research question we have, it makes sense to think about what language is and what it is not, how it is structured, and how it "works." This section does not replace an introduction to linguistics, but it gives us a solid starting point for our purposes. If you are interested in reading more on this, there are many excellent introductory textbooks to linguistics and its diverse subfields. One of the most entertaining ones is Fromkin, Rodman, and Hyams (2018). For a more focused overview of English and its history, see Crystal (2003).

Language often encodes information redundantly, i.e., we say the same thing in several ways: through the meanings of the words, their positions, the context, and many other cues. Words themselves consist of different components, which are the focus of different linguistic disciplines: their meaning (**semantics**), their function in a sentence (**syntax**), and the prefixes and endings (**morphology**). Not all words have all of this information. And when we work with textual data, we might not be interested in all of this information. In fact, it can be beneficial to remove some of the information we do not need.

When we work with text, the unit we are interested in depends strongly on the problem we are investigating. Traditionally, this was a report or an article. However, when we work with social media, it can also refer to a user's entire posting history, to a single message, or even to an individual sentence. Throughout this Element, we will refer to all these units of text as **documents**. It should be clear from the context what size a document has. Crucially, one document always represents one observation in our data. To refer to the entire collection of documents/observations, we use the word **corpus** (plural *corpora*).

The set of all the unique terms in our data is called the **vocabulary** (V). Each element in this set is called a **type**. Each *occurrence* of a type in the data is called a **token**. So the sentence "*a good sentence is a sentence that has good words*" has 10 tokens but only 7 types (namely, "*a*," "*good*," "*sentence*," "*is*," "*that*," "*has*," and "*words*"). Note that types can also include punctuation marks and multiword expressions (see more in Section 4).

2.1 Word Descriptors

2.1.1 Tokens and Splitting

Imagine we are looking at reports and want to filter out short sentences because they don't contain anything of interest. The easiest way to do so is by defining a cutoff for the number of words. However, it can be surprisingly tricky to define what a **word** is. How many words are there in "She went to Berlin" and in "She went to San Luis Obispo"? The most general definition used in many languages is any string of characters delimited by white space. However, note that Chinese, for example, does not use white space between words. Unfortunately, not all words are surrounded by white space. Words at the beginning of a line have no white space beforehand. Words at the end of a phrase or sentence might have a punctuation symbol (commas, full stops, exclamation or question marks, etc.) directly attached to them. Quotation marks and brackets complicate things even more.

Of course, we can separate these symbols by introducing extra white space. This process is called **tokenization** (because we make each word and punctuation mark a separate token). A different process, called **sentence splitting**, separates a document into sentences and works similarly. The problem there is to decide when a dot is part of a word (as in titles like *Mr.* or abbreviations like *abbr.*) or a full stop at the end of a sentence. Both tokenization and sentence splitting are prediction tasks, for which there exist reliable machine learning models. We will not discuss the inner workings of these tools in detail here but instead will rely on the available Python implementations in the `spacy` library.

We need to load the library and create an instance for the language we work with (here, English), and then we can use it on any input string.

```
import spacy
nlp = spacy.load('en')

documents = "I've been 2 times to New York in 2011, but did not
    have the constitution for it. It DIDN'T appeal to me. I
    preferred Los Angeles."
tokens = [[token.text for token in sentence] for sentence in nlp
    (documents).sents]
```

Code 4 Applying sentence splitting and tokenization to a set of documents.

Executing the example in Code 4 gives us the following results for `tokens`:

```
[['I', "'ve", 'been', '2', 'times', 'to', 'New', 'York', 'in', '
    2011', ',', 'but', 'did', 'not', 'have', 'the', '
    constitution', 'for', 'it', '.'],
 ['It', "DIDN'T", 'appeal', 'to', 'me', '.'],
 ['I', 'preferred', 'Los', 'Angeles', '.']]
```

Notice that the capitalized word *DIDN'T* was not tokenized properly, but the first, lowercased version was. This difference suggests a common and simple solution, namely, converting everything to lowercase as a preprocessing step.

2.1.2 Lemmatization

Suppose we have a large corpus of articles from the business section of various newspapers. We are interested in how often and which companies acquire each other. We have a list of words, but words come in different forms, depending on the tense and aspect, so we might have to look for "acquire," "acquires," "acquired," and "acquiring" (since we are dealing with newspaper articles, they might be referring to recent events or to future plans, and they might quote people). We could try to formalize this pattern by looking only for the endings *–e*, *–es*, *–ed*, and *–ing*, but that only works for **regular verbs** ending in *–re*. If we are interested in **irregular verbs**, we might see forms like "go," "goes," "went," "gone," or "going." And it does not stop there: the firms we are looking for might acquire just one "subsidi*ary*" or several "subsidi*aries*," one "comp*any*" or several "comp*anies*." We need a more principled way to deal with this variation.

When we look up a word in a dictionary, we usually just look for the **base form** (in the previous example, that would be the infinitive "go"). This dictionary base form is called the **lemma**. All the other forms don't change the core meaning of this lemma but add further information (such as temporal and other aspects). Many of these inflections are required by the syntax, i.e., the context and word order that make a sentence grammatical. When we work with text and are more interested in the meaning, rather than the morphology or syntax, it can be useful to replace each word with its lemma. This step reduces the amount of variation in the data and makes it easier to collect meaningful statistics. Rather than getting a single count for each of "go," "goes," "went," "gone," and "going," we would simply record having seen the form "go" five times in our data. This reduction does lose some of the temporal and other syntactic information, but that information might be irrelevant for our purposes. Many words can be lemmatized by undoing certain patterns, based on the type of word (e.g., remove "–ed" from the end of a verb like "walked" but remove "–ier" from the end of an adjective like "happier"). For the exceptions (for example, "to go"), we have to have a lookup table.

Luckily, **lemmatization** is already built into `spacy`, so we can make use of it. Applying lemmatization to our example sentences from above (Code 5), we get

```
lemmas = [[token.lemma_ for token in sentence] for sentence in
    nlp(documents).sents]
```

Code 5 Applying lemmatization to a set of documents.

```
[['-PRON-', 'have', 'be', '2', 'time', 'to', 'new', 'york', 'in'
    , '2011', ',', 'but', 'do', 'not', 'have', 'the', '
    constitution', 'for', '-PRON-', '.'],
 ['-PRON-', "didn't", 'appeal', 'to', '-PRON-', '.'],
 ['-PRON-', 'prefer', 'los', 'angeles', '.']]
```

Note that as part of the process, all words are converted to lowercase (if we have not done so already ourselves), while pronouns are all conflated as a special token `-PRON-` (see also Section 2.2). Again, the word *DIDN'T* was not lemmatized correctly but simply changed to lowercase.

2.1.3 Stemming

Maybe we are searching legal texts for political decisions and are generally interested in anything that has to do with the constitution. So the words "constitution," "constitutions," "constitutional," "constitutionality," and "constitutionalism" are all of interest to us, despite having different parts of speech.

Lemmatization will not help us here, so we need another way to group the words across word classes.

An even more radical way to reduce variation is **stemming**. Rather than reducing a word to the lemma, we strip away everything but the irreducible morphological core (the **stem**). For example, for a word like "anticonstitutionalism," which can be analyzed as "anti+constitut+ion+al+ism," we remove everything but "constitut." The most famous and commonly used stemming tool is based on the algorithm developed by Porter (1980). For each language, it defines a number of **suffix**es (i.e., word endings) and the order in which they should be removed or replaced. By repeatedly applying these actions, we reduce all words to their stems. In our example, all words derive from the stem "constitut–" by attaching different endings.

Again, a version of the Porter stemmer is already available in Python, in the `nltk` library (Loper & Bird, 2002), but we have to specify the language (Code 6).

```python
from nltk import SnowballStemmer
stemmer = SnowballStemmer('english')
stems = [[stemmer.stem(token) for token in sentence] for
    sentence in tokens]
```

Code 6 Applying stemming to a set of documents.

This gives us

```
[['i', 've', 'been', '2', 'time', 'to', 'new', 'york', 'in', '
    2011', ',', 'but', 'did', 'not', 'have', 'the', 'constitut',
     'for', 'it', '.'],
 ['it', "didn't", 'appeal', 'to', 'me', '.'],
 ['i', 'prefer', 'los', 'angel', '.']]
```

While this is extremely effective in reducing variation in the data, it can make the results harder to interpret. One way to address this problem is to keep track of all the original words that share a stem and how many times each of them occurred. We can then replace the stem with the most common derived form (in our example, this would most likely be "constitution"). Note that this can conflate word classes, say, nouns and verbs, so we need to decide whether to use this depending on our goals.

2.1.4 n-Grams

Looking at words individually can be a good start. Often, though, we want to look also at the immediate context. In our example, we have two concepts that span two words, "New York" and "Los Angeles," and we do not capture them by looking at each word in turn.

Instead of looking at each word in turn, we can use a sliding window of *n* words to examine the text. This window is called an ***n*-gram**, where *n* can have any size. Individual words are also called **unigrams**, whereas combinations of two or three words are called **bigrams** and **trigrams**, respectively. For larger *n*, we simply write the number, e.g., "4-grams."

We can extract *n*-grams with a function from `nltk`:

```python
from nltk import ngrams
bigrams = [gram for gram in ngrams(tokens[0], 2)]
```

Code 7 Extracting bigrams from a sentence.

This gives us

```
[('I', "'ve"),
 ("'ve", 'been'),
 ('been', '2'),
 ('2', 'times'),
 ('times', 'to'),
 ('to', 'New'),
 ('New', 'York'),
 ('York', 'in'),
 ('in', '2011'),
 ('2011', ','),
 (',', 'but'),
 ('but', 'did'),
 ('did', 'not'),
 ('not', 'have'),
 ('have', 'the'),
 ('the', 'constitution'),
 ('constitution', 'for'),
 ('for', 'it'),
 ('it', '.')]
```

We will see later how we can join frequent bigram expressions that are part of the same "word" (see Section 4).

2.2 Parts of Speech

Let's say we have collected a large corpus of restaurant menus, including both fine-dining and fast-food joints. We want to know how each of them describes the food. Is it simply "grilled," or is it "juicy," "fresh," or even "artisanal"? All of these food descriptors are **adjectives**, and by extracting them and correlating them with the type of restaurant who use them, we can learn something about the restaurants' self-representation – and about the correlation with price.[1]

At a very high level, words denote things, actions, and qualities in the world. These categories correspond to **parts of speech**, e.g., nouns, verbs, and adjectives (most languages have more categories than just these three). They are jointly called **content words** or **open-class words** because we can add new words to each of these categories (for example, new nouns, like "tweet," or verbs, like "twerking"). There are other word classes: determiners, prepositions, etc. (see below). They don't "mean" anything (i.e., they are not referring to a concept) but help to structure a sentence and to make it grammatical. They are therefore referred to jointly as **function words** or **closed-class words** (it is very unlikely that anybody comes up with a new preposition any time soon). Partially because function words are so short and ubiquitous, they are often overlooked. While we have a rough idea of how many times we have seen the noun "class" in the last few sentences, it is almost impossible to consciously notice how often we have seen, say, "in." [2]

Languages differ in the way they structure sentences. Consequentially, there was little agreement about the precise number of these grammatical categories, beyond the big three of content words (and even those are not always sure). The need for NLP tools to work across languages recently spawned efforts to come up with a small set of categories that applies to a wide range of languages (Petrov, Das, & McDonald, 2011). It is called the universal part-of-speech tag set (see https://universaldependencies.org/u/pos/). In this Element, we will use this set of 15 parts of speech.

Open-class words:

- ADJ: adjectives. They modify nouns to specify their properties. Examples: *awesome*, *red*, *boring*
- ADV: adverbs. They modify verbs, but also serve as question markers. Examples: *quietly*, *where*, *never*
- INTJ: interjections. Exclamations of some sort. Examples: *ouch*, *shhh*, *oi*

[1] This example is based on the book The Language of Food: A Linguist Reads the Menu, by Dan Jurafsky. He did exactly this analysis and found that the kinds of adjectives used are a good indicator of the price the restaurant charges (though maybe not necessarily of the quality).

[2] Even though we do not notice function words, we tend to have individual patterns for using them. This property makes them particularly interesting for psychological analysis and forensic linguistics. The book The Secret Life of Pronouns: What Our Words Say about Us by James Pennebaker examines this quality of function words. They can therefore be used to identify kidnappers, predict depression, or assess the stability of a marriage.

- NOUN: nouns. Entities in the world. Examples: *book*, *war*, *shark*
- PROPN: proper nouns. Names of entities, a subclass of nouns. Examples: *Rosa*, *Twitter*, *CNN*
- VERB: full verbs. Events in the world. Examples: *codes*, *submitted*, *succeed*

Closed-class words:

- ADP: adpositions. Prepositions or postpositions, markers of time, place, beneficiary, etc. Examples: *over*, *before*, *(get) down*
- AUX: auxiliary and modal verbs. Used to change time or modality. Examples: *have (been)*, *could (do)*, *will (change)*
- CCONJ: coordinating conjunctions. Link together parts of sentences with equal importance. Examples: *and*, *or*, *but*
- DET: determiners. Articles and quantifiers. Examples: *a*, *they*, *which*
- NUM: numbers. Exactly what you would think it is . . .
- PART: particles. Possessives and grammatical markers. Examples: *'s*
- PRON: pronouns. Substitutions for nouns. Examples: *you*, *her*, *his*, *myself*
- SCONJ: subordinating conjunctions. Link together parts of sentences with one part being more important. Examples: *since*, *if*, *that*

Other

- PUNCT: punctuation marks. Examples: *!*, *?*, *–*
- SYM: symbols. Word-like entities, often special characters, including emojis. Examples: *%*, *$*, *:)*
- X: other. Anything that does not fit into any of the above. Examples: *pffffrt*

Automatically determining the parts of speech and syntax of a sentence are known as **POS tagging** and **parsing**, two of the earliest and most successful NLP applications. We can again use the POS tagger in spacy:

```
pos = [[token.pos_ for token in sentence] for sentence in nlp(
    documents).sents]
```

Code 8 POS tagging a set of documents.

This gives us

```
[['PRON', 'VERB', 'VERB', 'NUM', 'NOUN', 'ADP', 'PROPN', 'PROPN'
    , 'ADP', 'NUM', 'PUNCT', 'CCONJ', 'VERB', 'ADV', 'VERB', '
    DET', 'NOUN', 'ADP', 'PRON', 'PUNCT'],
 ['PRON', 'PUNCT', 'VERB', 'ADP', 'PRON', 'PUNCT'],
 ['PRON', 'VERB', 'PROPN', 'PROPN', 'PUNCT']]
```

Because POS tagging was one of the first successful NLP applications, a lot of research has gone into it. By now, POS taggers are more accurate, more consistent, and definitely much faster than even the best-trained linguists.

2.3 Stopwords

If we want to assess the general topics in product reviews, we usually do not care whether a review refers to "*the* price" or "*a* price," and can remove the determiner. Similarly, if we are looking at the political position of parties in their manifestos, we know who wrote each manifesto. So we do not need to keep their names when they mention themselves in the document. In both cases, we have a set of words that occur often, but do not contribute much to our task, so it can be beneficial to remove them. The set of these ignorable words is called **stopwords**.

As we have seen above, many words in a text belong to the set of function words. In fact, the most-frequent words in any language are predominantly function words. The ten most common words in English (according to the Oxford English Corpus), with their parts of speech are: *the* (DET), *be* (VERB/AUX), *to* (ADP/PART), *of* (ADP), *and* (CCONJ), *a* (DET), *in* (ADP), *that* (CCONJ), *have* (VERB/AUX), *I* (PRON). While these are all very useful words when building a sentence, they do not really mean much on their own, i.e., without context. In fact, for most applications (e.g., topic models; see Boyd-Graber, Mimno, & Newman, 2014), it is better to ignore them altogether.[3]

We can either exclude stopwords based on their part of speech (see above), or by using a list. The former is more general, but it risks throwing out some unintended candidates. (If we exclude prepositions, we risk losing the noun "round" if it gets incorrectly labeled.) The latter is more task-specific (e.g., we can use a list of political party names), but it risks leaving out words we were not aware of when compiling the list. Often, it can therefore be beneficial to use a combination of both. In `spacy`, we can use the `is_stop` property of a token, which checks it against a list of common stopwords. The list of English stopwords is in Appendix A.

For our running example, if we exclude common stopwords, and filter out noncontent words (all parts of speech except NOUN, VERB, PROPN, ADJ, and ADV):

```python
content = [[token.text for token in sentence
           if token.pos_ in {'NOUN', 'VERB', 'PROPN', 'ADJ', 'ADV'}
```

[3] See the exception for profiling above, though.

```
        and not token.is_stop]
        for sentence in nlp(documents).sents]
```

Code 9 Selecting content words based on POS.

This gives us

```
[["'ve", 'times', 'New', 'York', 'constitution'],
 ['appeal'],
 ['preferred', 'Los', 'Angeles']]
```

Another way to reduce variation is to replace any numbers with a special token, rather than to remove them. There are infinitely many possible numbers, and in many contexts, we do not care about the exact amount they denote (unless, of course, we are interested in prices). We will see an example of how to do this most efficiently in Section 3, when we learn about regular expressions.

2.4 Named Entities

Say we are interested in the changing patterns of tourism and want to find the most popular destinations over time in a corpus of travel blogs. We would like a way to identify the names of countries, cities, and landmarks. Using POS tagging, we can easily identify all proper names, but that still does not tell us with what kind of entity we are dealing. These semantic categories of words are referred to as **named entities**.

Proper names refer to entities in the real world, such as companies, places, persons, or something else, e.g., *Apple*, *Italy*, *George Oscar Bluth*, *LAX* or *Mercedes*. While implementing a **named entity recognizer** (NER) is outside the scope of this work, it generally works by using a list of known entities that are unlikely to change (for example, country names or first names). However, names are probably the most open-ended and innovative class of words, and there are plenty of entities that are not listed anywhere. Luckily, we can often determine what kind of entity we have by observing the context in which they occur. For example, company names might be followed by “Inc.” or “LLC.” Syntax can give us more cues – if an entity *says* something, *eats*, *sleeps*, or is associated with other verbs that denote human activities, we can label it as a person. (Of course, this gets trickier when we have metaphorical language, e.g., “parliament says . . .”).

Luckily, `spacy` provides a named entity recognizer that can help us identify a wide range of types: PERSON, NORP (Nationality OR Religious or Political group), FAC (facility), ORG (organization), GPE (GeoPolitical Entity), LOC (locations, such as seas or mountains), PRODUCT, EVENT (in sports, politics,

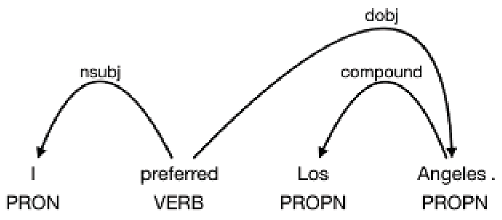


Figure 1 Example of a dependency parse.

history, etc.), WORK_OF_ART, LAW, LANGUAGE, DATE, TIME, PERCENT, MONEY, QUANTITY, ORDINAL (ordinal numbers such as “third”), and CARDINAL (regular numbers).

For our running example, we can use the following code to extract the words and their types:

```python
entities = [[(entity.text, entity.label_)
            for entity in nlp(sentence.text).ents]
            for sentence in nlp(documents).sents]
```

Code 10 Named entity recognition.

This gives us

```
[[('2', 'CARDINAL'), ('New York', 'GPE'), ('2011', 'DATE')],
 [],
 [('Los Angeles', 'GPE')]]
```

2.5 Syntax

Let’s say we are interested in firm acquisitions. Imagine we want to filter out all the business transactions (who bought whom) from a large corpus of newswire text. We define a list of verbs that express acquisitions and want to find all the nouns associated with these verbs. That is, who is doing the acquiring (the subjects), or who is being acquired (the objects). We want to end up with NOUN-VERB-NOUN triples that give us the names of the firms and other entities involved in the acquisitions, as well as the verb.

We have already seen that words can have different grammatical forms, i.e., parts of speech (nouns, verbs, etc.). However, a sentence can have many words with the same POS, and what a specific word means in a sentence depends in part on its **grammatical function**. Sentence structure, or **syntax** is an essential field of study in linguistics, and explains how words in a sentence hang together.

One of the most important cues to a word's function in English is its position in the sentence. In English, grammatical function is mainly determined by word order, while other languages use markers on the words. Changing the word order changes the grammatical function of the word. Consider the difference between "Swimmer eats fish," and "Fish eats swimmer." The **subject** (the entity doing the eating) and the **object** (the entity being eaten) are now switched. We have already seen verbs, subjects, and objects as examples of syntactic functions, but there are many other possible **dependency relations** that can hold between words. The idea in dependency grammar is that the sentence "hangs" off the main verb like a mobile. The links between words describe how the words are connected.

Naturally, these connections can vary quite a bit between languages, but similar to the Universal POS tags, there have been efforts to unify and limit the set of possibilities (McDonald et al., 2013; Nivre et al., 2015, 2016, inter alia). Here is a list of the ones used in the Universal Dependencies project (again, see https://universaldependencies.org):

- `acl`, clausal modifier of a noun (adjectival clause)
- `advcl`, adverbial clause modifier
- `advmod`, adverbial modifier
- `amod`, adjectival modifier
- `appos`, appositional modifier
- `aux`, auxiliary verb
- `case`, case marker
- `cc`, coordinating conjunction
- `ccomp`, clausal complement
- `clf`, classifier
- `compound`, compound
- `conj`, conjunction
- `cop`, copula
- `csubj`, clausal subject
- `dep`, unspecified dependency
- `det`, determiner
- `discourse`, discourse element
- `dislocated`, dislocated elements
- `dobj`, direct object
- `expl`, expletive

- `fixed`, fixed multiword expression
- `flat`, flat multiword expression
- `goeswith`, goes with
- `iobj`, indirect object
- `list`, list
- `mark`, marker
- `nmod`, nominal modifier
- `nsubj`, nominal subject
- `nummod`, numeric modifier
- `obj`, object
- `obl`, oblique nominal
- `orphan`, orphan
- `parataxis`, parataxis
- `pobj`, prepositional object
- `punct`, punctuation
- `reparandum`, overridden disfluency
- `root`, the root of the sentence, usually a verb
- `vocative`, vocative
- `xcomp`, open clausal complement

Many of these might never or rarely occur in the language we study, and for many applications, we might only want to look at a small handful of relations, e.g., the core arguments, `nsubj`, `obj`, `iobj`, `pobj`, or `root`. We can see an example of a dependency parse in Figure 1. Why is this relevant? Because knowing where a word stands in the sentence and what other words are related to it (and in which function), helps us make sense of a sentence.

In Python, we can use the parser from the `spacy` library to get

```
[[(c.text, c.head.text, c.dep_) for c in nlp(sentence.text)]
 for sentence in nlp(documents).sents]
```

Code 11 Parsing.

For the first sentence, this gives us

```
[('I', 'been', 'nsubj'),
  ("'ve", 'been', 'aux'),
  ('been', 'been', 'ROOT'),
  ('2', 'been', 'npadvmod'),
  ('times', '2', 'quantmod'),
  ('to', '2', 'prep'),
  ('New', 'York', 'compound'),
  ('York', 'to', 'pobj'),
('in', 'been', 'prep'),
('2011', 'in', 'pobj'),
(',', 'been', 'punct'),
('but', 'been', 'cc'),
('did', 'have', 'aux'),
('not', 'have', 'neg'),
('have', 'been', 'conj'),
('the', 'constitution', 'det'),
('constitution', 'have', 'dobj'),
('for', 'have', 'prep'),
('it', 'for', 'pobj'),
('.', 'been', 'punct')]
```

We can see how the verb "been" is the root node that everything else hangs off of. For our example with the firm acquisitions, we would want to extract anything labeled with `nsubj`, `dobj`, or `pobj`.

2.6 Caveats – What If It's Not English?

Say you have a whole load of Italian data that you want to work with, doing some of the things we have done in the previous sections. What are your options?

`spacy` comes with support for a number of other languages, including German (`de`), Spanish (`es`), French (`fr`), Italian (`it`), Dutch (`nl`), and Portuguese (`pt`). All you have to do is load the correct library:

```python
import spacy
nlp = spacy.load('it')
```

However, you need to download these language models separately and make sure they are in the right path. They mostly offer all the same functionalities (lemmatization, tagging, parsing). However, their performance can vary. Because NLP has been so focused on English, there is often much less training data for other languages. Consequently, the statistical models are often much less precise.

NLP was developed predominantly in the English-speaking world, by people working on English texts. However, English is not like many other languages. It is a pidgin language between Celtic, old Germanic languages, Franco-Latin, and Norse, and at some point it lost most of its inflections (the word endings that we learn in declension tables). However, word endings allow you to move the word order around: you still know from the endings what the subject and what the object is, no matter where they occur. To still distinguish subject from object, English had to adopt a fixed word order. Because of this historical development, *n*-gram methods work well for English: we see the same order many times and can get good statistics over word combinations. We might not see the same *n*-gram more than once in languages that are highly inflected or have variable word order – no matter how large our corpus is. Finnish, for example, has 15 cases, i.e., many more possible word endings, and German orders subordinate clauses differently from main clauses. For those languages, lemmatization becomes even more important, and we need to rely on syntactic *n*-grams rather than sequences.

Alternatively, `nltk` offers support to stem text in Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, and Swedish. This method is, of course, much coarser than lemmatization, but it is an efficient way to reduce the noise in the data.

There are programs out there to tag and parse more languages, most notably the TreeTagger algorithm, which can be trained on data from the Universal POS project. However, those programs involve some manual installation and command-line running that go beyond the scope of this Element.

If you are working with other languages than English, you can, in many cases, still lemmatize the data. However, you might have fewer or no options for POS tagging, parsing, and NER. For the applications in the following sections, though, adequate preprocessing is often more important than being able to perform all types of linguistic analyses. However, if your analysis relies on these tools, take a look at the NLP literature on low-resource language processing, a thankfully growing subfield in NLP.

Now that we have seen what the basic building blocks of linguistic analysis are, we can start looking at extracting information out of the text. We will see how to search for flexible patterns in the data to identify things like email addresses, company names, or numerical amounts in Section 3. In Section 5.2.3, we will see how to find the most meaningful and important words in a corpus (given that pure frequency is not a perfect indicator). And in Section 4, we will see how to capture that some words occur much more frequently together than on their own, and how to define this notion.

3 Regular Expressions

Say you are working with a large sample of employment records from Italian employees, with detailed information about when they worked at which company, and a description of what they did there. However, the names of the

Table 1 Examples of simple matching

Sequence	**Matches**
`e`	any single occurrence of e
`at`	`at, rat, mat, sat, cat, attack, attention, later`

companies are contained in the text, rather than in a separate entry. You want to extract the names to get a rough overview of how many different companies there are and to group the records accordingly. Unfortunately, the NER tool for Italian does not work as well as you would like. You do know, though, that Italian company names often end in several abbreviations (similar to “Co.,” “Ltd.,” or “Inc.”). There are only two you want to consider for now (“s.p.a.” and “s.r.l.”), but they come in many different spelling variations (e.g., *SPA*, *spa*, *S.P.A.*). How can you identify these markers and extract the company names right before them?

In the same project, you have collected survey data from a large sample of employees. Some of them have left their email address as part of the response, and you would like to spot and extract the email addresses. You could look for anything that comes before or after an @ sign, but how do you make sure it is not a Twitter handle? And how do you make sure it is a valid email address?[4]

String variation is a textbook application of **regular expressions** (or RegEx). They are flexible **patterns** that allow you to specify what you are looking for in general, and how it can vary. In order to do so, you simply write the pattern you want to match. Table 1 shows examples of simple patterns and their matches.

In Python, we can specify regular expressions and then apply them to text with either `search()` or `match()`. The former checks whether a pattern is *contained* in the input, the latter checks whether the pattern *completely matches* the input:

```python
import re
pattern = re.compile("at")

re.search(pattern, 'later') # match at position (1,3)
re.match(pattern, 'later') # no match
```

[4] With “valid,” we don’t mean whether it actually exists (you could only find out by emailing them), but only whether it is correctly formatted (so that you actually *can* email them).

Table 2 Examples of quantifiers

Quantifier	Means	Example	Matches
?	0 or 1	`fr?og`	`fog, frog`
*	0 or more	`cooo*l`	`cool, coool`
+	1 or more	`hello+`	`hello, helloo, hellooooooo`

Table 3 Examples of special characters

Character	Means	Example	Matches
`.`	any single character	`.el`	`eel, Nel, gel`
`\n`	newline character (line break)	`\n+`	one or more line breaks
`\t`	a tab stop	`\t+`	one or more tabs
`\d`	a single digit [0-9]	`B\d`	`B0, B1, ..., B9`
`\D`	a nondigit	`\D.t`	`' t, But, eat`
`\w`	any alphanumberic character	`\w\w\w`	`top, WOO, bee,`...
`\W`	nonalphanumberic character		
`\s`	a white space character		
`\S`	a non–white space character		
`\`	"escapes" special characters to match them	`.+\.com`	`abc.com, united.com`
`^`	the beginning of the input string	`^...`	first three-letter word in line
`$`	the end of the input string	`^\n$`	empty line

Sometimes, we know that a specific character will be in our pattern, but not how many times. We can use special **quantifiers** to signal that the character right before them occurs any number of times, including zero (∗), exactly one or zero times (?), or one or more times (+). See the examples in Table 2.

In Python, we simply incorporate them into the patterns:

```python
pattern1 = re.compile("fr?og")
pattern2 = re.compile("hello+")
pattern3 = re.compile("cooo*l")
```

REs also provide a whole set of special characters to match certain types of characters or positions (see Table 3).

Table 4 Examples of character classes

Class	Means	Example	Matches
`[abc]`	match any of `a, b, c`	`[bcrms]at`	`bat, cat, rat, mat, sat`
`[^abc]`	match anything BUT `a, b, c`	`te[^ ]+s`	`tens, tests, teens, texts, terrors...`
`[a-z]`	match any lowercase character	`[a-z][a-z]t`	`act, ant, not, ... wit`
`[A-Z]`	match any uppercase character	`[A-Z]...`	`Ahab, Brit, In a, ..., York`
`[0-9]`	match any digit	`DIN A[0-9]`	`DIN A0, DIN A1, ..., DIN A9`

Table 5 Examples of groups

Group	Means	Example	Matches
`(abc)`	match sequence `abc`	`.(ar).`	`hard, cart, fare...`
`(ab|c)`	match `ab` OR `c`	`(ab|C)ate`	`abate, Cate`

However, maybe we don't want really *any* old character in a wildcard position, only one of a small number of characters. For this, we can define **classes** of allowed characters.

In Python, we can use character classes to define our patterns (see also the examples in Table 4):

```python
pattern = re.compile('[bcr]at')
```

So far, we have only defined single characters and their repetitions. However, RegExes also allow us to specify entire **groups**, as shown in Table 5.

In order to apply REs to strings in Python, we have several options. We can `match()` input strings, which will only succeed if it can cover the entire string. If we are only interested in whether the pattern is contained as a substring, we can use `search()` instead.

```python
document = 'the batter won the game'
matches = re.match(pattern, document) # returns None
searches = re.search(pattern, document) # matches 'bat'
```

Either of these operations return an object that is `None` if the pattern could not be applied. Otherwise, it will provide us with several properties of the results: `span()` gives us a tuple of the substring positions that match the pattern (and which can be used as indices for slicing), while `group()`returns the matched substring. For our example above, `searches.span()` will return `(4, 7)`, while `searches.group()` returns `'bat'`. If we define several groups in our pattern, we can access them via `groups()`, either jointly or individually.

```
word = 'preconstitutionalism'
affixes = re.compile('(...).+(...)')
results = re.search(affixes, word)
```

Here, `results.groups()` will return the prefix and the suffix found in the input, i.e., `('pre', 'ism')`.

Lastly, we can also use the RegEx to replace elements of the input string that match the pattern via `sub()`. We have mentioned earlier how we can replace the digits of any number with a special token, for example `0`. The following code does this:

```
numbers = re.compile('[0-9]')
re.sub(numbers, '0', 'Back in the 90s, when I was a 12-year-old,
     a CD cost just 15,99EUR!')
```

Code 12 Replacing patterns in a string.

The code in 12 will return

`'Back in the 00s, when I was a 00-year-old, a CD cost just 00,00EUR!'`.

A **warning**: regular expressions are extremely powerful, but you need to know when to stop. A well-written RegEx can save you a lot of work, and accomplish quite a bit in terms of extracting the right bits of text. However, RegExes can quickly become extremely complicated when you add more and more variants that you would like to account for. It’s very easy to write a RegEx that captures way too much (false positives), or is so specific that it captures way too little (false negatives). It is therefore always a good idea to write a couple of positive and negative examples, and make sure the RegEx you develop matches what you want.

4 Pointwise Mutual Information

Intuitively, we would like to treat terms like “social media,” “New York,” or “greenhouse emission” as a single concept, rather than treating them as individual units delimited by white space. However, just looking at the frequency of the two words together is not sufficient. The word pair “of the” is extremely frequent, but nobody would argue that this is a single meaningful concept. To complicate matters more, each part of these often also occurs by itself. Think of the elements in “New York” (while “York” is probably not too frequent, “New” definitely is). Leaving these words apart or joining them will have a huge impact on our findings. How do we join the right words together?

Word is a squishy concept, though. We have seen that white spaces are more of an optional visual effect than a meaningful word boundary (e.g., Chinese does not use them). Even the (in)famous German noun compounds (“Nahrungsmittelettikierungsgesetz,” or “*law for the labeling of food stuffs*”) can be seen as just a more white-space-efficient way of writing the words.

Even in English, we have concepts that include several words, but that we would not (anymore) analyze as separate components. Think of place names like “Los Angeles” and “San Luis Obispo,” but also concepts like “social media.” (If you disagree on the latter, that is because the process, called **grammaticalization**, is gradual, and in this case not fully completed.) While it is still apparent what the parts mean, they have become inextricably linked with each other. These word pairs are called **collocations**.

When we work with text, it is often better to treat collocations as one entity, rather than as separate units. To do this, we usually just replace the white space between the words with an underscore (i.e., we would use `los_angeles`). There are some “general” collocations (like “New York” or “Los Angeles”). However, nearly every domain includes specialized concepts relevant to the context of our analysis. For example, “quarterly earnings” in earnings reports, or “great service” if we are analyzing online reviews.

We can compute the probability of each word occurring on its own, but this does not help much: “New” is much more likely to occur than “York.” We need to relate these quantities to how likely the two words are to occur together. If either (or all) of the candidate words occur in the context of the others, we have reason to believe it is a collocation, e.g., “Angeles” in the case of “Los Angeles.” If, on the other hand, either word is just as likely to occur independently, we probably do not have a collocation, e.g., “of” and “the” in “of the.”

To put numbers to this intuition, we use the **joint probability** of the entire n-gram occurring in a text and normalize it by the individual probabilities (for more on probabilities, see Appendix B). Since the probabilities can get fairly small, we typically use the logarithm of the result. So, for a bigram, we have

$$PMI(x;y) = \log \frac{P(x,y)}{P(x)P(y)}$$

Joint probabilities can be rewritten as conditional probabilities, i.e.,

$$P(x,y) = P(y) \times P(x|y) = P(x) \times P(y|x)$$

The last two parts are the same because we don't care about the order of x and y. So if it's more convenient, we can transform our PMI calculation into

$$PMI(x;y) = \log \frac{P(x|y)}{P(x)}$$

or

$$PMI(x;y) = \log \frac{P(y|x)}{P(y)}$$

Essentially, the latter tells us for a bigram "xy" how likely we are to see x followed by y (i.e., $P(y|x)$), normalized by how likely we are to see y in general ($P(y)$).

Applying collocation detection to an example corpus in Python is much easier, since there are some functions implemented in `nltk` that save us from computing all the probability tables:

```python
from nltk.collocations import BigramCollocationFinder,
    BigramAssocMeasures
from nltk.corpus import stopwords

stopwords_ = set(stopwords.words('english'))

words = [word.lower() for document in documents for word in
    document.split()
          if len(word) > 2
          and word not in stopwords_]
finder = BigramCollocationFinder.from_words(words)
bgm = BigramAssocMeasures()
collocations = {bigram: pmi for bigram, pmi in finder.
    score_ngrams(bgm.mi_like)}
collocations
```

Code 13 Finding collocations with `nltk`.

Table 6 MI values and rank for a number of collocations in Moby Dick

Rank	Collocation	*MI(x; y)*
1	moby_dick	83.000000
2	sperm_whale	20.002847
3	mrs_hussey	10.562500
4	mast_heads	4.391153
5	sag_harbor	4.000000
6	vinegar_cruet	4.000000
7	try_works	3.794405
8	dough_boy	3.706787
9	white_whale	3.698807
10	caw_caw	3.472222

The code first creates a long list of strings from the corpus, removing all stop-words and words shorter than two characters (thereby removing punctuation). It then creates a `BigramCollocationFinder` object, initialized with the word list. From this object, we can extract the statistics. Here, we use a variant of PMI, called `mi_like`, which does not take the logarithm, but uses an exponent.

Table 6 shows the results for the text sample from Moby Dick. We can then set a threshold above which we concatenate the words.

All of the last few sections are examples of **preprocessing**, to get rid of noise and unnecessary variation, and to increase the amount of signal in the data.

All of these steps can be combined, so we get a much more homogenous, less noisy version of our input corpus. However, there is no general rule of what steps to exclude and which to keep. For topic models (see Section 9), removing stopwords is crucial. However, for other tasks, such as author attribute prediction, the number and choice of stopwords, such as pronouns, is essential. We might not need numbers for sentiment analysis, so we can either remove them or replace them with a generic number token. However, we would definitely want to keep all numbers if we are interested in tracking earnings.

Ultimately, which set of preprocessing steps to apply is highly dependent on the task and the goals. It makes sense to try out different combinations on a separate, held-out development set, and observe which one gives the best results. More importantly, preprocessing decisions always change the text, and thereby affect the inferences we can make. We need to be aware of this effect, and we should justify and document all preprocessing steps and decisions we have taken (cf. Denny and Spirling, 2018).

5 Representing Text

Say you have collected a large corpus of texts on political agendas and would like to work with it. As a human, you can just print them out, take a pen, and start to take notes. But how do you let a computer do the same?

To work with texts and gain insights, we need to represent the documents in a way for the computer to process it. We cannot simply take a Microsoft Word document and work with that. (Word documents are great for writing but terrible to analyze, since they store a lot of formatting information along with the content.) Instead, we need to get it into a form that computers can understand and manipulate. Typically, this is done via representing documents as **feature vectors**. Vectors are essentially lists of numbers, but they have several useful properties that make them easy to work with. The domain of working with vectors and matrices is **linear algebra**. We will use many of the concepts from linear algebra, so it can be good to refresh your knowledge. A good place to start is the YouTube channel 3Blue1Brown, which has a dedicated series on the topic (www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab). Lang (2012) is a great textbook. We will only cover the basics here. Vectors can either represent a collection of **discrete features**, or a collection of **continuous dimensions**. Discrete features characterize a word or a document (in which case they are called **discrete representations**). The dimensions of **distributed representations** represent the word or document holistically, i.e., their position relative to all other words or documents in our corpus.

In discrete feature vectors, each dimension of the vector represents a feature with a specific meaning (e.g., the fifth dimension means whether the word contains a vowel or not). The values in the vector can either be binary indicator variables (0 or 1) or some form of counts. Discrete representations are often also referred to as **sparse** vectors, because not every document will have all the features. For the most part, these vectors are empty (e.g., a document might not contain the word `tsk!`, so the corresponding position in the vector will be empty). We will see discrete and sparse vectors when we look at bags of words, counts, and probabilities in Section 5.2.

In distributed or continuous representations, vectors as a whole represent points in a high-dimensional space, i.e., by a vector of fixed dimensionality with continuous-valued numbers. The individual dimensions of the vectors do not mean anything anymore, i.e., they can not be interpreted. Instead, they describe

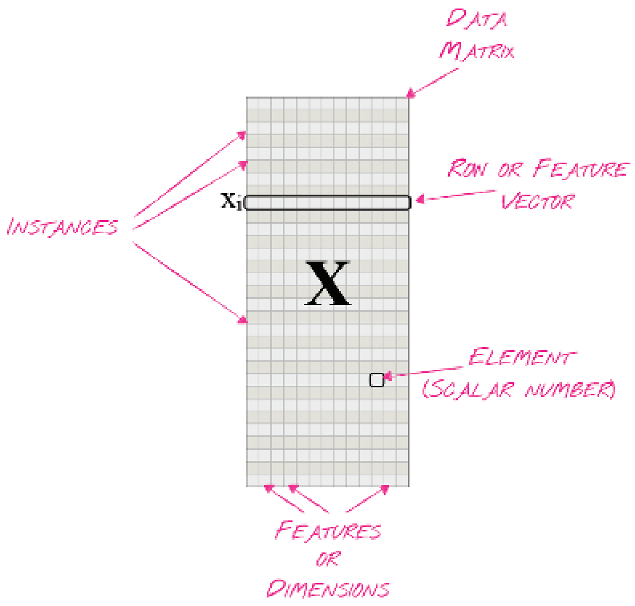


Figure 2 Elements of the data matrix.

the coordinates of a document relative to all other documents in the vector space. These vector representations are also called **dense**, because we do not have any empty positions in the vectors, but will always have some value in each position. When the vectors represent words or documents, these high-dimensional, dense vector representations are called **embeddings** (because they *embed* words or documents in the high-dimensional space). We will see distributed representations when we look at word and document embeddings in Section 5.3.

5.1 Enter the Matrix

After you choose how to represent each document in your corpus, you end up with a bunch of vectors. Put those vectors on top of each other, and you have a matrix. Let's talk about some of the terminology.

Irrespective of the type of representation we choose (sparse or dense), it makes sense to think of the data in terms of **matrices and vectors**. A vector is an ordered collection of D dimensions with numbers in them. Each dimension is a feature in discrete vectors. A matrix is a collection of N vectors, where each row represents one document, and the columns represent the features mentioned above, or the dimensions in the embedding space. This matrix is also sometimes referred to as **term-document matrix** (Manning & Schütze, 1999), reflecting the two elements involved in its construction. When we refer to the **data matrix** in the following, we mean a matrix with one row for each document, and as many columns as we have features or dimensions in our representation. If we want to look at one particular feature or dimension, we can thus extract the corresponding **column vector**. And when we refer to a document, unless specified otherwise, we refer to a **row vector** in the data matrix. See Figure 2 for an illustration of these elements.

We generally refer to **inputs** to our algorithms with X, to **outputs** with Y. We will use these uppercase symbols to refer to vectors. If the input is a single **scalar** number, we use lowercase x or y. If we refer to a specific element of a vector, we will use an index to refer to the position in the vector, e.g., x_i, where i is an integer $\in D$ denoting the specific dimension. For matrices, we will use bold-faced uppercase $\mathbf{X}$, and denote individual elements with their row and column index: $x_{i,j}$. Generally, we will use $i \in 1..N$ to iterate over the documents, and $j \in 1..D$ to iterate over the dimensions.

Thinking of the data this way allows us to use linear algebra methods. We can now compute the linear combination of the data with some coefficients. Or we can decompose the data, reduce their dimensionality, transform them, etc. Linear algebra also allows us to combine the data with other kinds of information. For example, networks can be represented as matrices, where each row and column represents a node, and each cell tells us whether two nodes are connected (and if so, how strongly). Luckily, Python allows us to derive matrix representations from text, and to use them in various algorithms.

5.2 Discrete Representations

Suppose we have a corpus of documents, and have preprocessed it so that it contains a vocabulary of 1,000 words. We want to represent each document as the number of times we have seen each of these 1,000 words in a document. How do we collect the counts to make each document a 1,000-dimensional vector?

The simplest way of quantifying the importance of a particular word is to count its occurrences in a document. Say we want to track the importance of a particular issue (say, "energy") for a company in their quarterly reports. Getting the **term frequency** of the (lemmatized) word can give us a rough impression of its importance over time. There is evidence that our brain keeps track of how often you have seen or heard a word as well. Well, kind of. You can immediately tell whether you have seen the word "dog" more frequently than the word "platypus," but maybe not *how* often.

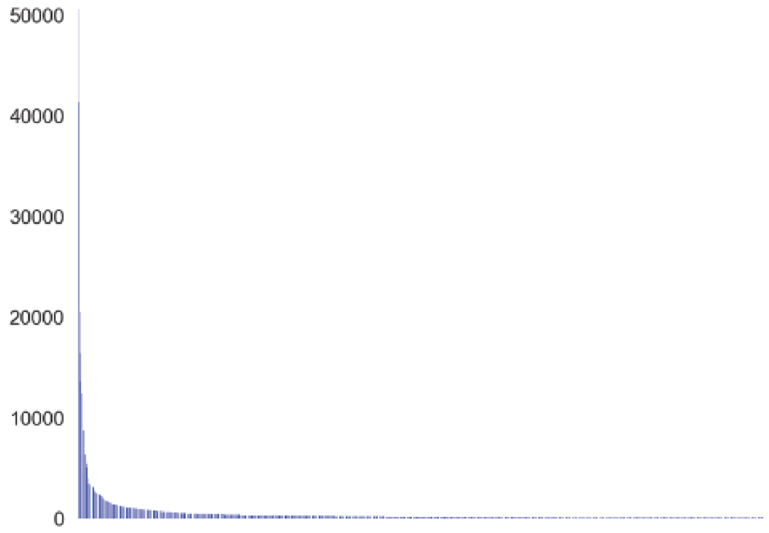


Figure 3 Frequency distribution of the top 1,000 words in a random sample of tweets, following Zipf's law.

There are other frequency effects in language: not all words occur nearly at the same frequency. One thing to notice for word counts is their overall distribution. The most frequent word in any language is typically several times more frequent than the second most frequent word. That word, in turn, is magnitudes more frequent than the third most frequent word, and so on until we reach the "tail" of the distribution. That tail is a long list of words that occur only once (see Figure 3). This phenomenon (which is a power-law distribution) was first observed by Zipf (1935) and is therefore often referred to as **Zipf's law**. It also holds for city sizes, income, and many other sociocultural phenomena. It is, therefore, essential to keep in mind that methods that assume a normal distribution do not work well for many, if not most, linguistic problems. Zipf's law also means that a small minority of word types account for the majority of word tokens in a corpus.[5]

One of the most important choices we have to make when working with texts is how to represent our input X. The most straightforward way to represent a document is simply to create a count vector. Each dimension represents one word in our vocabulary, so D has the size of all words in our vocabulary V,

[5] More than half of all word tokens in a newspaper belong to a tiny group of word types. Unfortunately, those are usually not the informative word types. This is why learning vocabulary is such an essential part of learning any language.

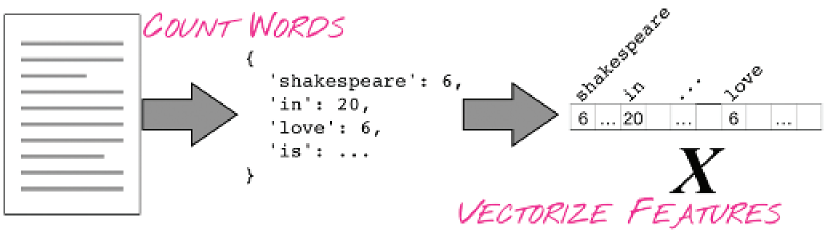


Figure 4 Schematic of a bag-of-words representation.

i.e., $D = |V|$. We then simply count for each document how often each word occurs in it. This representation (a **count vector**) does not pay any attention to where in the document a word occurs, what its grammatical role is, or other structural information. It simply treats a document as a bunch of word counts, and is therefore called a **bag of words** (or BOW; see Figure 4). The result of this application to our corpus is a **count matrix**.

In Python, we can collect counts with a function in `sklearn`:

```python
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(analyzer='word')

X = vectorizer.fit_transform(documents)
```

Code 14 Extracting *n*-gram count representations.

5.2.1 *n*-gram Features

In addition to individual words, we would also like to account for *n*-grams of lengths 2 and 3 to capture both word combinations and some grammatical sequence effects in our representations. We want to be able to see the influence of “social media” as well as “social” and “media.” (We have seen in Section 4 how to preprocess such terms to make them a single token.)

N-grams are sequences of tokens (either words or characters), and the easiest way to represent a text is by keeping track of the frequencies of all the *n*-grams in the data. Luckily, `CountVectorizer` allows us to extract all *n*-gram counts within a specified range.

```python
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(analyzer='word',
                             ngram_range=(1, 3),
                             min_df=10,
                             max_df=0.75,
                             stop_words='english')

X = vectorizer.fit_transform(documents)
```

Code 15 Extracting *n*-gram count representations.

In the code in snippet 15, we create a vectorizer that collects all word unigrams, bigrams, and trigrams. To be included, these *n*-grams need to occur in at least ten documents, but not in more than 75 percent of all documents. We exclude any words that are contained in a list of stopwords.

The exact *n*-gram range, minimum and maximum frequency, and target entity ('`word`' or '`char`', for character) depend on the task, but sensible starting settings are

```
word_vectorizer = CountVectorizer(analyzer='word',
                                  ngram_range=(1, 2),
                                  min_df=10,
                                  max_df=0.75,
                                  stop_words='english')

char_vectorizer = CountVectorizer(analyzer='char',
                                  ngram_range=(2, 6),
                                  min_df=10,
                                  max_df=0.75)
```

Using character *n*-grams can be useful when we look at individual words as documents, or if we are dealing with multiple languages in the same corpus. *N*-grams also help account for neologisms (new word creations), and misspellings. From a linguistic perspective, this approach may seem counterintuitive. However, from an algorithmic perspective, character *n*-grams have many advantages. Many character combinations occur across languages (e.g., "en"), so we can make them comparable even if we cannot use the appropriate lemmatization. *N*-grams capture overlap and similarities between words better than a whole-word approach. This property can help us with both spelling mistakes and new variations of words because they account for the overlap with the original word. Compare the words *definitely* and its frequent misspelling *definately*. They do have significant overlap, both in terms of form and meaning. Trigrams, for example, capture this fact: seven of the nine unique trigrams we can extract from the two words are shared. Their character count vectors will be similar – despite the spelling variation. In a word-based approach, these two tokens would not be similar. We usually only need to store a limited number of all possible character combinations, because many do not occur in our data (e.g., “qxq”). *N*-grams are, therefore, also a much more efficient representation computationally: there are much fewer character combinations than there are word combinations.

5.2.2 Syntactic Features

One problem with *n*-grams is that we only capture direct sequences in the text. For English, this is fine, since word order is fixed (as we have discussed), and grammatical constituents like subjects and verbs are usually close together. As a result, there are enough word *n*-grams that frequently occur to capture most of the variation. The fact that NLP was first developed for English and its strict word order might explain the enduring success of *n*-grams.

However, as we have seen, word order in many languages is much freer than it is in English. That means that subjects and verbs can be separated by many more words, and occur at different positions in the sentence. For example, in German,

> “Sie sagte, dass [der Koch]$_{nominative}$ [dem Gärtner]$_{dative}$ [die Bienen]$_{accusative}$ gegeben hatte”

(*She said that [the chef]*$_{nominative}$ *had given [the bees]*$_{accusative}$ *to [the gardener]*$_{dative}$, noun cases marked with subscript) could also be written as:

> “Sie sagte, dass [dem Gärtner]$_{dative}$ [die Bienen]$_{accusative}$ [der Koch]$_{nominative}$ gegeben hatte”

> “Sie sagte, dass [die Bienen]$_{accusative}$ [dem Gärtner]$_{dative}$ [der Koch]$_{nominative}$ gegeben hatte”

and various other orderings. Not all of them will sound equally natural to a German speaker (they might seem grammatically **marked**). Still, they will all be grammatically valid and possible. In those cases, it is hard to collect sufficient statistics via *n*-grams, which means that similar documents might look very different as representations. Instead, we would want to collect statistics/counts over the pairs of words that are *grammatically* connected, no matter where they are in the sentence.

In Python, we can use the dependency parser from `spacy` to extract words and their grammatical parents as features, as we have seen before. We simply extract a string of space-separated word pairs (joined by an underscore), and then pass that string to the `CountVectorizer`, which treats each of these pairs as a word:

```
from sklearn.feature_extraction.text import CountVectorizer

features = [' '.join(["{}_{}".format(c.lemma_, c.head.lemma_)
    for c in nlp(s.text)])
 for s in nlp(documents).sents]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(features)
```

Code 16 Extracting syntactic feature count representations.

For example, the counts for the first sentence in our examples (“I’ve been 2 times to New York in 2011, but did not have the constitution for it”) would be

```
{'have_be': 2,
'-PRON-_be': 1,
'be_be': 1,
'2_be': 1,
'time_2': 1,
'to_2': 1,
'new_york': 1,
'york_to': 1,
'in_be': 1,
'2011_in': 1,
',_be': 1,
'but_be': 1,
'do_have': 1,
'not_have': 1,
'the_constitution': 1,
'constitution_have': 1,
'for_have': 1,
'-PRON-_for': 1,
'._be': 1}
```

5.2.3 TF-IDF Counts

Count vectors are a good start, but they give equal weight to all terms with the same frequency. However, some terms are just more “interesting” than others, even if they have the same or fewer occurrences. Let us say we have 500 companies, and for each of them several years’ worth of annual reports. The collected reports of each company are one document. We will not be surprised to find the words “report” or “company” among the most frequent terms, since they are bound to be used in every document. However, if only some of those companies use “sustainability,” but do so consistently in every report, this would be a very interesting signal worth discovering. How could we capture this notion?

When we first look at a corpus, we usually want to know what it is generally about, and what terms we might want to pay special attention to. Frequency alone is not sufficient to find important words: many of the stopwords we purposely remove are extremely frequent. However, there is something about word frequency that carries information. It is just not the raw frequency of each word.

The answer is to weigh the raw frequency (its **term frequency**, or TF) by the number of documents the word occurs in (its **document frequency**, or DF). This way, words that occur frequently, but in every document (such as function words), will receive very low scores. So will words that are rare, but could occur everywhere (say, "president" in our example: it is not frequent, but will be mentioned by every report). Words that are moderately frequent but occur in only a few documents get the highest score, and these words are what we are after. We can get both of these quantities from a discrete count matrix. The TF is the sum over the corresponding column of the matrix, while the DF is the count of the nonzero entries in that column (see Figure 5).

There are many ways to compute the weighting factor. In the most common one, we check how many documents a term occurs in, and take its **inverse**, or the fraction of all documents in our corpus that contain the word. (This quantity can be obtained by counting the nonzero rows in the respective column of our count matrix.)

$$IDF(w) = log\frac{N}{DF(w)}$$

This is – quite sensibly – called the **inverse document frequency** (or IDF). Because we often have large corpora and terms occur in only a few, the value

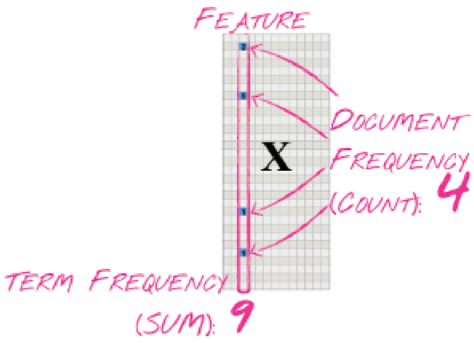


Figure 5 Extracting term and document frequency from a count matrix.

of the IDF can get very small, risking an underflow. To prevent this, we take the logarithm.[6] The idea was first introduced by Spärck Jones (1972).

By multiplying the two terms together, we get the TF-IDF score of a word:

$$TFIDF(w) = TF(w) \cdot IDF(w)$$

There are several variants to compute both TF and IDF. We can use Laplace smoothing, i.e., we add 1 to each count, to prevent division by 0 (in case a term is not in the corpus). Another possibility is to take the logarithm of the frequency to dampen frequency effects. TF can be computed as a binary measure (i.e., the word is present in the document or not), as the raw count of the word, the relative count (divided by the length of each document), or a smoothed version:

$$TF_{smoothed} = log(TF(w) + 1)$$

A smoothed version of the IDF is

$$\log \frac{N}{DF(w) + 1} + 1$$

A common variant that implements smoothing and log discounting is

$$TFIDF(w) = (\log TF(w) + 1) \cdot \frac{1}{\log(1 + \frac{N}{DF(w)})}$$

By plotting the IDF versus the TFIDF values, and scaling by the TF, we can see how the nonstopwords in a corpus are distributed. Figure 6 shows the content words in Moby Dick. The five words with the highest TFIDF scores are labeled. While the prevalence of "whale" is hardly surprising (the TFIDF is dominated by the high TF, but then, it's a book about whales), we can see a couple of interesting entries to the right, including the doomed protagonist Ahab, and the archaic form "ye" (for "you").

In Python, we can use the `TfidfVectorizer` in `sklearn`, which functions very much like the `CountVectorizer` we have already seen:

```python
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(analyzer='word',
                                   min_df=0.001,
                                   max_df=0.75,
                                   stop_words='english',
```

[6] You might also see the IDF as $-log\frac{DF(w)}{N}$, which comes out to the same.

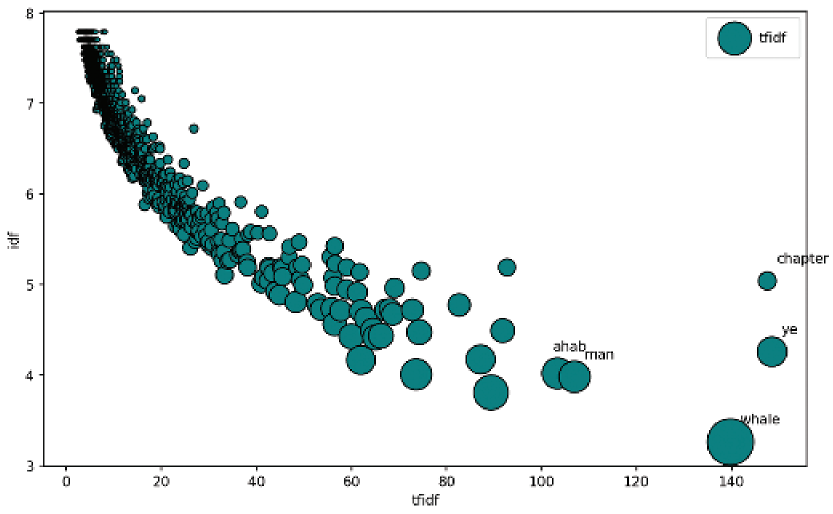


Figure 6 Scatterplot of TFIDF and IDF values, scaled by TF, for content words in Moby Dick.

```
                                  sublinear_tf=True)

X = tfidf_vectorizer.fit_transform(documents)
```

Code 17 Extracting TFIDF values from a corpus.

This returns a data matrix with a row for each document, and the transformed values in the cells. We can get the IDF values from the `idf_` property of the vectorizer. In order to get the TFIDF values of a certain feature for the entire corpus, we can sum over the all rows/documents. The `A1` property turns the sparse matrix into a regular array:

```
idf = tfidf_vectorizer.idf_
tfidf = X.sum(axis=0).A1
```

Code 18 computing IDF and TFIDF values for features.

5.2.4 Dictionary Methods

In many cases, we already have prior knowledge about the most important words in a domain, and can use those as features. This knowledge usually comes in the form of word lists, codebooks, or dictionaries, mapping from a word to the associated categories. Several sources and tools exist for applying such **dictionary methods** to text, especially in the field of psychology, such as

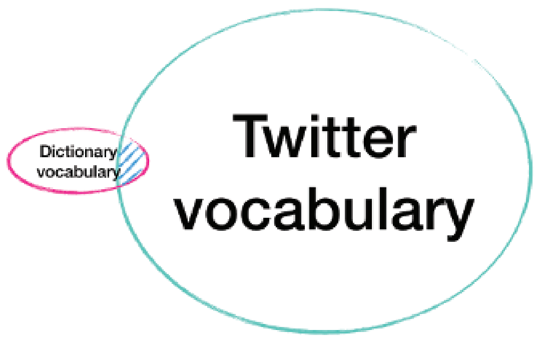


Figure 7 Vocabulary overlap between dictionary-based methods and target domain.

LIWC (Pennebaker, Francis, & Booth, 2001). They come with additional benefits, such as the functionality of scoring texts along several dimensions, such as emotions or psychological traits.

While it can be handy to use such a dictionary to get started, it is essential to note that dictionary-based methods have inherent limitations. Their vocabulary and the vocabulary we find in the data will only partially overlap (see Figure 7). In practice, this means that whatever *is* in the dictionary is a perfect candidate (we have high precision in our selection). However, we might miss out on many indicative words because they were not included in the dictionary (our selection has a low recall of all the possible features). Given how flexible language is, it is challenging to cover all possible expressions in a dictionary (Schwartz et al., 2013).

A handy approach is expanding a dictionary-based method with some of the methods discussed here, to expand the dictionary. You can, for example, use the Wordify tool (https://wordify.unibocconi.it) to find more terms related to the dictionary categories. This approach will give us bona fide features and will pick up data-specific features. We can substantially extend our initial word list and "customize" it to the data at hand by choosing automatically derived features that strongly correlate with the dictionary terms.

5.3 Distributed Representations

Many times, we would like to capture the meaning similarity between words, such as for **synonyms** like "flat" and "apartment." We might also be interested in their general relatedness in semantic terms. For example, "doctor," "syringe," and "hospital" all tend to appear in the same context. For example, to capture language change, we might be interested in capturing the most similar words to "mouse" over time (Hamilton et al., 2016; Kulkarni et al., 2015). Or we might want to measure stereotypes by seeing how similar to each other "nurse" is with "man" and "woman," respectively (Garg et al., 2018).

This semantic relatedness of context (called **distributional semantics**) was recognized early on in linguistics by Firth (1957), who stated that

> "You shall know a word by the company it keeps."

So far, we have treated each word as a separate entity, i.e., "expenditure" and "expenditures" are treated as two different things. We have seen that this is why preprocessing is so important (in this case, lemmatization) because it solves this problem. However, preprocessing does not adequately capture how similar a word is to other words. Distributed representations allow us to do all of these things, by projecting words (or documents) into a high-dimensional space where their relative position and proximity to each other mean something. That is, words that are similar to each other should be close together in that space. In discrete representations, we defined the dimensions/features, hoping that they capture semantic similarity. In distributed representations, we use contextual co-occurrence as the criterion to arrange words in the given space. Similarity is now an emerging property of proximity in the space. First, though, we need to be more specific by what we mean with *similar to each other*.

5.3.1 Cosine Similarity

When we are searching for "flats in milan" online, we will often get results that mention "apartments in milan" or "housing in milan." We have not specified these two other words, but they are obviously useful and relevant. We often get results that express the same thing, but in different words. Intuitively, these words are more similar to each other than to, say, the word "platypus." If we want to find the nearest neighbors, we need a way to express this intuition of semantic similarity between words in a more mathematical way.

For all of the above, the key question is, what does it mean for words to be "similar to each other"? Remember that our words are vectors. A vector is nothing but a point specified in the embedding space. So in distributed representations, our words are points in a D-dimensional space. We can draw a line from the **origin** (i.e., the 0 coordinates) of the space to each point. This property allows us to compute the angle between two words, using the **cosine similarity** of two vectors. The angle is a measure of the distance between the two vectors.

To get the cosine similarity, we first take the dot product between the two vectors, and normalize it by the norm of the two vectors:

$$\cos(A, B) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^{D} a_i b_i}{\sqrt{\sum_{i=1}^{D} a_i^2} \sqrt{\sum_{i=1}^{D} b_i^2}}$$

The dot product is simply the sum of all pairs of elements multiplied together, and the norm is the square root of the sum over all squared elements. The reason we first square each element and then compute the root is that this gets rid of negative values. So when we sum the numbers up, we end up with a positive number.

Cosine similarity gives us a number between −1.0 and 1.0. A negative value means that the vectors point in diametrally opposite directions. In contrast, a value of 1.0 implies that the two vectors are the same. A value close to 0.0 means the two vectors are perpendicular to each other, or uncorrelated with each other. It does *not* say that the words mean the opposite of each other!

We can use cosine similarity to compute the **nearest neighbors** of a word, i.e., to find out which words are closest to a target word in the embedding space. We do this by sorting the vectors of all other words by their cosine similarity with the target. Ideally, we would want the cosine similarity between words with similar meaning (“flat” and “apartment”) and between related words (“doctor,” “syringe,” and “hospital”) to be high. The library we will use has a function to do so, but in some cases (e.g., when averaging over several models), we will need to implement it ourselves:

```python
import numpy as np
def nearest_neighbors(query_vector, vectors, n=10):
    # compute cosine similarities
    ranks = np.dot(query_vector, vectors.T) / np.sqrt(np.sum(
    vectors**2, 1))
    # sort by similarity, reverse order, and get the top N
    neighbors = [idx for idx in ranks.argsort()[::-1]][:n]
    return neighbors
```

Code 19 Implementation of nearest neighbors in vector space.

Now we have clarified what it means for words to be similar to each other. The only questions left are, *How do we get all the words into a high-dimensional space? and How do we arrange them by cosine similarity*?

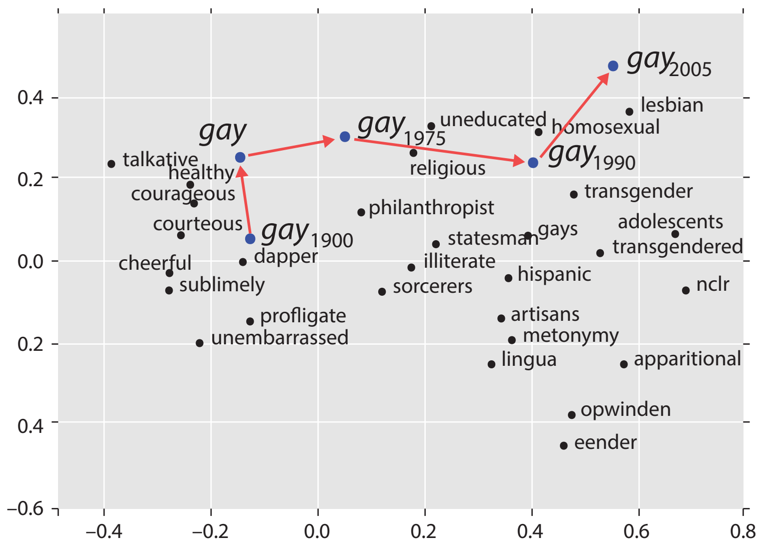


Figure 8 Position and change over time of the word "gay" and its nearest neighbors, from Kulkarni et al. (2015).

5.3.2 Word Embeddings

Let's say we are interested in the changing meaning of words over time. What did people associate with a word in the early 1900s? Does that differ from its present-day meaning (e.g., Kulkarni et al. 2015, see Figure 8)? What were people associating with gender roles, ethnic stereotypes, and other psychological biases during different decades (Garg et al., 2018; Bhatia, 2017)? A distributed representation of words from different times allows us to discover these relations by finding the nearest neighbors to a word under different settings.

Following the distributional hypothesis, for words to be similar to each other, they should occur in similar contexts. That is, if we often see the same two words within a few steps from each other, we can assume that they are **semantically related**. For example, "doctor" and "hospital" will often occur together in the same documents, sometimes within the same sentence, and are therefore semantically similar. If we see two words in the same slot of a sentence, i.e., if we see them interchangeably in the same context, we can assume they are **semantically similar**. For example, look at the slots in the sentences "I only like the ___ gummy bears" and "There was a vase with ___ flowers." Both can be filled with "red" or "yellow" (or "blue," or ...), indicating that colors are syntactically similar. Unfortunately, the same is true for **antonyms**. "I had a ___ weekend" can be completed by a wide range of adjectives, but that does not mean that "boring" and "amazing" mean similar things.

Traditionally, distributed representations were collected by defining several **context words**. For each target word in the vocabulary, we would then count how often it occurred in the same document as each of the context words. This approach is known as the **vector space model**. The result was essentially a discrete count matrix. The counts were usually weighted by TF-IDF, to bring out latent similarities, and then transformed by keeping only the *k* highest eigenvalues from Singular Value Decomposition. The method to produce these dense representations is called **Latent Semantic Analysis** or LSA (Deerwester, Dumais, Furnas, Landauer, & Harshman, 1990). Distributional semantic models like LSA used the entire document as context. That is, we would count the co-occurrence of the target with context words anywhere in the entire text. In contrast, modern embedding models define context as a local window to either side of the target word.

Some years ago, a new tool, `word2vec` (Mikolov, Sutskever, Chen, Corrado, & Dean, 2013), made headlines for being fast and efficient in projecting words into vector space, based on a large enough corpus. Rather than relying on a predefined set of context words to represent our target word, this algorithm uses the natural co-occurrence of all words to compute the vectors. ("Co-occurrence" means the words occur within a particular window size, rather than the whole document.) We only specify the number of dimensions we want to have in our embedding space, and press go.

The intuitive explanation of how the algorithm works is simple. You can imagine the process similar to arranging a bunch of word magnets on a giant fridge. Our goal is to arrange the word magnets so that proximity matches similarity. That is, words in similar contexts are close to each other, and dissimilar words are far from each other. We initially place all words randomly on the fridge. We then look at word pairs: if we have seen them in the same sentence somewhere, we move them closer together. If they do not occur in the same sentence, we increase the distance. Naturally, as we adjust a word to be close to one of its neighbors, we move it further away from other neighbors. So we need to iterate a couple of times over our corpus before we are satisfied with the configuration.

On a fridge, we only have two dimensions in which to arrange the words, which gives us fewer options for sensible configurations. In `word2vec`, we can choose a higher number, which makes the task easier, as we have many more degrees of freedom to find the right distances between all words. Initial proposals included 50 or 100 dimensions, but for some unknown reason, 300

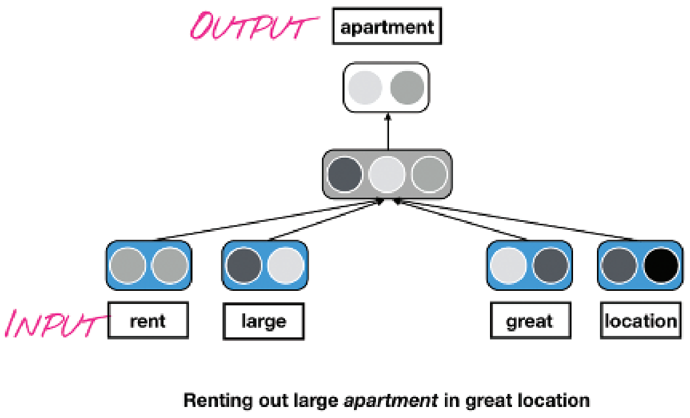


Figure 9 Schematic of `Word2vec` model with CBOW architecture for five words with two dimensions. Context word vectors in blue, target word vector in white.

dimensions seem to work well for most purposes (Landauer & Dumais, 1997; Lau & Baldwin, 2016).

There are two main architectures to implement this intuitive "moving magnets on a fridge" and updating the resulting positions of the vectors: CBOW and the skipgram model. The algorithm iterates over each document in the corpus, and carries out a prediction task. It selects an input word *w* and a second, output word *c* from the vocabulary. It then tries to predict the output from the input. In practice, we can either predict the output word from the input of context words (this is called the **continuous bag of words (CBOW) model**; see Figure 9), or predict the context word output from the target word input, which is called **skipgram model** (see Figure 10). Skipgram is somewhat slower, but considered more accurate for low-frequency words.

In either case, we keep separate matrices for the target and context versions of each word, and discard the output matrix in the end. That is, in the CBOW model, we delete the target word matrix, in the skipgram model, we remove the context word matrix. For more details on this, see the explanations by Rong (2014) or Goldberg and Levy (2014).

To update the output word matrices, we need to consider the whole vocabulary, which can be quite large (up to hundreds of thousands of words). There are two algorithms to make this efficient: **negative sampling** and **hierarchical softmax**. Both help us update the vast number of words in our vocabulary in an efficient manner.

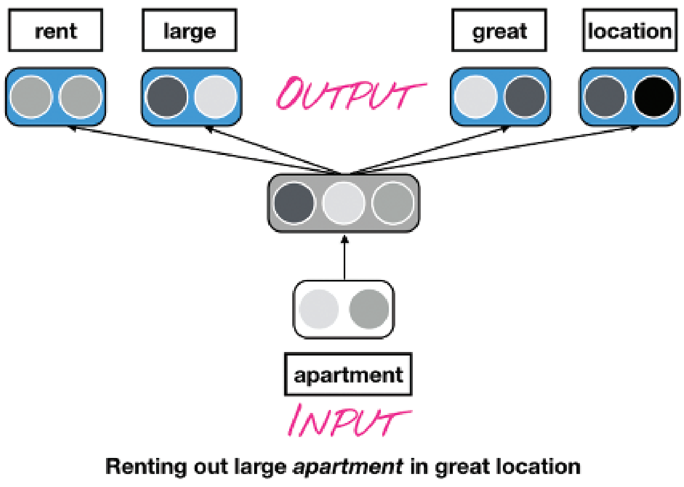


Figure 10 Schematic of `Word2vec` model with skipgram architecture for five words with two dimensions. Context word vectors in blue, target word vector in white.

In negative sampling, the model chooses a random sample of context words c. These words are labeled either 1 or -1, depending on whether they occur within a fixed window either side of the target word w. The model computes the dot product of the input and output vectors and checks the sign of the result. If the prediction is correct, i.e., the sign is positive, and c does indeed occur in the context of w, or if the sign is negative and v does indeed *not* occur in the context of w, nothing happens. If the prediction is incorrect, i.e., the predicted sign differs from the label, the model updates the selected vectors. If we iterate this process often enough, words that occur within a certain window size of each other get drawn closer and closer together in the vector space.

Given an input word, we can check for a few sampled words, whether they occur in that word's context. However, we can instead also predict the output probability of *all* vocabulary words. Then we choose the word with the highest probability as the prediction. This probability distribution is called a **softmax**. Unfortunately, though, our vocabulary is usually quite large. Computing the softmax for every input word over all output words involves an exponential number of comparisons. The time complexity of this computation becomes quickly prohibitive. The hierarchical softmax addresses this problem. We build a tree on top of our vocabulary, starting with each vocabulary word as a leaf node. We then combine leaf nodes to form higher layers of nodes until we have a single root node. We can use this tree to get the prediction probabilities, and only need to update the paths through this tree to compute the probabilities. This structure cuts the number of necessary computations by a large factor (the logarithm of the vocabulary size). For a much more in-depth explanation and derivation of the methods, see Rong (2014).

The quality of the resulting word vectors depends – as always – on the quality of the input data. One common form of improving quality is to throw out stop words or, rather, to keep only content words. We can achieve this through POS information. Adding POS also disambiguates the different uses of homonyms, e.g., "show_NOUN" versus "show_VERB."

There are many pretrained word embeddings available, such as GloVe (Pennington, Socher, & Manning, 2014) or Fasttext (Grave, Bojanowski, Gupta, Joulin, & Mikolov, 2018), among others. They are based on vast corpora, and in some cases, embeddings exist for several languages in the same embeddings space, making it possible to compare across languages. If we are interested in general societal attitudes, or if we have tiny data sets, these **pretrained embeddings** are a great starting point.

In other cases, we might want to train embeddings on our corpus ourselves, to capture the similarities which are specific to this data collection. In the latter case, we can use the `word2vec` implementation in the `gensim` library. The model takes a list of list of strings as input.

```python
from gensim.models import Word2Vec
from gensim.models.word2vec import FAST_VERSION

# initialize model
w2v_model = Word2Vec(size=300,
                     window=5,
                     hs=0,
                     sample=0.000001,
                     negative=5,
                     min_count=10,
                     workers=-1,
                     iter=5000
)

w2v_model.build_vocab(corpus)

w2v_model.train(corpus, total_examples=w2v_model.corpus_count,
    epochs=w2v_model.epochs)
```

Code 20 Initializing and training a `word2vec` model.

We need to pay attention to the parameter settings. The **window size** determines whether we pick up more on semantic or syntactic similarity. The **negative sampling rate** and the number of negative samples affect how much frequent words influence the overall result (hs=0 tells the model to use negative sampling). As before, we can specify a minimum number of times a word needs to occur to be embedded. The number of threads on which to train (workers=-1 tells the model to use all available cores to spawn off threads, which speeds up training). The number of iterations controls how often we adjust the words in the embeddings space.

Note that there is no guarantee that we find the best possible arrangement of words, since word embeddings start randomly and are changed piecemeal. Because of this stochastic property, it is recommended to train several embedding models on the same data, and then average the resulting embedding vectors for each word over all of the models (Antoniak & Mimno, 2018).

We can access the trained word embeddings via the wv property of the model.

```
word1 = "clean"
word2 = "dirty"

# retrive the actual vector
w2v_model.wv[word1]

# compare
w2v_model.wv.similarity(word1, word2)

# get the 3 most similar words
w2v_model.wv.most_similar(word1, topn=3)
```

Code 21 Retrieving word vector information in the trained Word2vec model.

The resulting word vectors have several neat semantic properties (similarity of meaning), which allow us to do **vector space semantics**. The most famous example of this is the gender analogy of $king - man + woman = queen$. If we add the vectors of "king" and "woman" and then subtract the vector of "man," we get a vector that is quite close to the vector of the word "queen." (In practice, we need to find the nearest actual word vector of the resulting vector.) Other examples show the relationship between countries and their capitals, or the most typical food of countries. These tasks are known as **relational similarity prediction**.

```
# get the analogous word
w2v_model.wv.most_similar(positive=['king', 'woman'], negative=[
    'man'])
```

Code 22 Doing vector semantics with the trained word2vec model.

5.3.3 Document Embeddings

Imagine we want to track the language variation in a country city by city: how does the language in A differ from that in B? We have data from each city, and would like to represent the cities on a continuous scale (dialects in any language typically change smoothly, rather than abruptly). We can think of document embeddings as a linguistic profile of a document, so we could achieve our goal if we manage to project each city into a higher-dimensional space based on the words used there.[7]

Often, we are not interested in representing individual words, but in representing an entire document. Previously, we have used discrete feature vectors in a BOW approach to do so. However, BOW vectors only count how often each word was present in a document, no matter where they occur. The equivalent in continuous representations is taking the aggregate over the word embeddings in a document. We concatenate the word vectors row-wise into a matrix D, and compute either the mean or the sum over all rows:

$$e(D) = \sum_{w \in D} e(w)$$

or

$$e(D) = \frac{\sum_{w \in D} e(w)}{|D|}$$

However, we can also use a similar algorithm to what we have used before to learn separate **document embeddings**. Le and Mikolov (2014) introduced a model similar to `word2vec` to achieve this goal. The original paper called the model **paragraph2vec**, but it is now often referred to with the name of the successful `gensim` implementation, `doc2vec`.

Initially, we represent each word and each document randomly in the same high-dimensional vector space, albeit with separate matrices for the documents and words. We can now either predict target words based on a combined input of their context plus the document vector (similar to the CBOW of `word2vec`). This model architecture is called the Distributed Memory version of Paragraph Vector (PV-DM). Alternatively, we can predict context words based solely on the document vectors (this is called Distributed Bag of Words version of Para-

[7] This is precisely the idea behind the paper by Hovy and Purschke (2018), who find that the resulting city representations not only capture the dialect continuum (which looks nice on a map), but that they are also close in embedding space to local terms and can be used in prediction tasks.

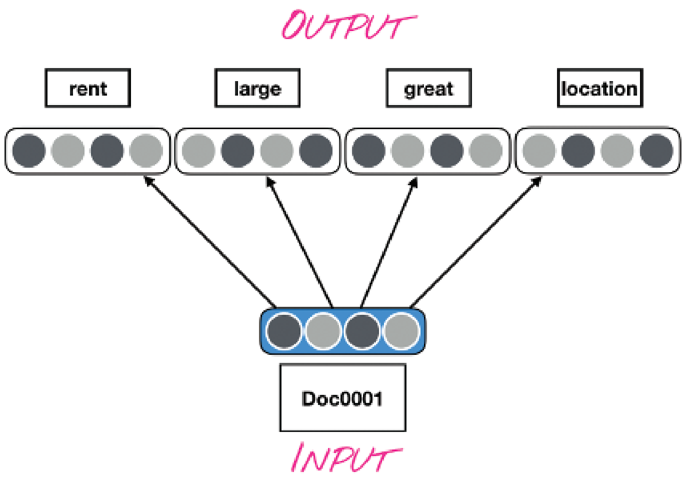


Figure 11 `Doc2Vec` model with PV-DBOW architecture.

graph Vector, or PV-DBOW, see Figure 11). The latter is much faster in practice and more memory-efficient, but the former structure is considered more accurate.

The Python implementation in `gensim`, `doc2vec`, is almost identical to the `word2vec` implementation. The main difference is that the input documents need to be represented as a special object called `TaggedDocument`, which has separate entries for `words` and `tags` (the document labels). Note that we can use more than one label! If we do not know much about our documents, we can simply give each of them a unique ID (in the code below, we use a 4-digit number with leading zeroes, i.e., `0002` or `9324`). However, if we know more about the documents (for example, the political party who wrote them, or the city they originated from, or their general sentiment), we can use this.

```python
from gensim.models import Doc2Vec
from gensim.models.doc2vec import FAST_VERSION
from gensim.models.doc2vec import TaggedDocument

corpus = []
for docid, document in enumerate(documents):
    corpus.append(TaggedDocument(document.split(), tags=["
    {0:0>4}".format(docid)]))

d2v_model = Doc2Vec(size=300,
                    window=5,
                    hs=0,
                    sample=0.000001,
                    negative=5,
                          min_count=10,
                          workers=-1,
                          iter=5000,
                          dm=0,
                          dbow_words=1)

d2v_model.build_vocab(corpus)

d2v_model.train(corpus, total_examples=d2v_model.corpus_count,
     epochs=d2v_model.epochs)
```

Code 23 Training a `Doc2vec` model.

The parameters are the same as for `word2vec`, with the addition of `dm`, which decides the model architecture (1 for distributed memory (PV-DM), 0 for distributed bag of words (PV-DBOW)), and `dbow_words`, which controls whether the word vectors are trained (1) or not (0). Not training the word vectors is faster. `doc2vec` stores separate embedding matrices for words and documents, which we can access via `wv` (for the word vectors) and `docvecs`.

```
target_doc = '0002'

# retrieve most similar documents
d2v_model.docvecs.most_similar(target_doc, topn=5)
```

In general, `doc2vec` provides almost the same functions as `word2vec`.

Projecting both word and document embeddings into the same space allows us to compare not just words to each other, which can tell us about biases and conceptualizations. It also allows us to compare words to documents, which can tell us what a specific document is about (see example in Figure 12). And of course, we can compare documents to documents, which can tell us which documents are talking about similar things. For all of these comparisons, we use the nearest neighbor algorithm (see Code 19).

5.4 Discrete versus Continuous

Which representation we choose depends on several things, not least the application.

Discrete representations are interpretable, i.e., they can assist us in finding a (causal) explanation of the phenomenon we investigate. The dimensions are either predefined by us, or based on the (most frequent or informative) words in the corpus. Each dimension means something, but many of them will be empty. A discrete data matrix is a bit like a Swiss cheese.

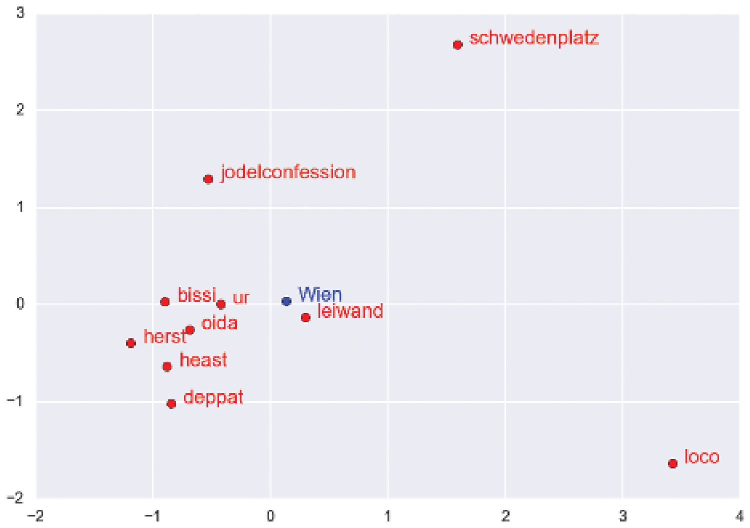


Figure 12 Example of documents (cities) and words in the same embedding space from Hovy and Purschke (2018). The words closest to the city vector (in blue) are dialect terms.

Continuous representations are only interpretable as a whole. The central criterion is similarity (as measure by distance), the individual dimensions have no meaning any more – they are chosen by us, but filled by the algorithm. Almost every cell will be nonzero, so a continuous data matrix is more like a brie than a block of Swiss cheese.

The difference between representations is also similar to the intended use, mirrored in two schools of thought: **rationalism** and **empiricism**.[8] In the first, we would like a model that helps us *understand* how the world works, and make sure the model is *interpretable*. In the second case, we want a model that *performs* well on a task, even if we do not understand how it does so. Ultimately, though, we need both approaches to make scientific progress.

EXPLORATION: FINDING STRUCTURE IN THE DATA

We will now look at ways to visualize the information we have (Section 6) and to group texts together into larger clusters (Section 7), before we turn to probabilistic models. We discuss how to spot unusual or creative sentences (and potentially generate our own) in Section 8; and how to find the most important themes and topics (via topic models) in Section 9.

[8] See also Peter Norvig's excellent blog post about the difference in scientific approaches to machine learning: www.norvig.com/chomsky.html.

6 Matrix Factorization

Let's say you have run `Doc2Vec` on a corpus and created an embedding for each document. You would like to get a sense of how the documents as a whole relate to each other. Since embeddings are points in space, a graph sounds like the most plausible idea. However, with 300 dimensions, this is a bit hard. How can you reduce the 300 down to 2 or 3? You would also like to visualize the similarity between documents using colors: documents with similar content should receive similar colors. Is there a way to translate meaning into color?

The data we collect from our count or TFIDF vectorizations is a sample of actual language use. These are the correlations we have observed. However, language is complex and has many **latent dimensions**. Various factors influence how we use words, such as societal norms, communicative goals, broader topics, discourse coherence, and many more. If we think of the word and document representations as mere results of these underlying processes, we can try and reverse-engineer the underlying dimensions.

Finding these latent factors is what **matrix factorization** tries to do: the methods we will see try to represent the documents and the terms as separate entities, connected through the latent dimensions. We can interpret the latent dimensions as a variety of things, such as coordinates in a two- or three-dimensional space, RGB color values, or (if we have more than three) as "topics."

This view was incredibly popular in NLP in the 1990 and into the early 2000s, thanks to a technique called **latent semantic analysis** (LSA), based on matrix decomposition. The resulting word and document matrices were used for all kinds of semantic similarity computations (between words, between documents), for topics, and for visualization. However, there are now specialized techniques that provide a more concise and flexible solution for everything LSA was used for. Word embeddings (see Section 5.3), document embeddings (ibd.), topic models (see Section 9), and visualization methods like t-SNE (see below). However, it is still worth understanding the principles behind it, as they can provide a quick and efficient way to analyze data. Matrix factorization is still widely used in **collaborative filtering** for recommender systems. For example, to suggest movies people should watch based on what they and people similar to them have watched before.

6.1 Dimensionality Reduction

So far, the dimensionality D of our data matrix X was defined by one of two factors. Either by the size of the vocabulary in our data (in the case of sparse vectors), or by a prespecified number of dimensions (in the case of distributed vectors).

However, there are good reasons to reduce this number. The first is that 300 dimensions are incredibly hard to imagine and impossible to visualize. However, to get a sense of what is going on in the data, we often *do* want to plot it. So having a way to project the vectors down to two or three dimensions is very useful for **visualization**.

The other reason is performance and efficiency: we might suspect that there is a smaller number of **latent dimensions** that is sufficient to explain the variation in the data. Reducing the dimensionality to this number can help us do a better analysis and prediction. The patterns that emerge can even tell us what those latent dimensions were.

In the following, we will look at two dimensionality-reduction algorithms. We can represent the input data as either sparse BOW vectors or dense embedding vectors. Note that using dimensionality reduction to k factors on a D-dimensional embedding vector is *not* the same as learning a k-dimensional embedding vector in the first place. There is no guarantee that the k embedding dimensions are the same as the latent dimensions we get from dimensionality reduction. Matrix decomposition relies on feature-based input to find a smaller subset. A feature of this approach on document representation matrices is that earlier dimensions are more influential than later ones. At the same time, the embedding-based techniques make use of the entire set of dimensions.

6.1.1 Singular Value Decomposition

Singular Value Decomposition (SVD) is one of the most well-known matrix factorization techniques. It is an exact decomposition technique based on **eigenvalues**. The goal is to decompose the matrix into three elements:

$$\mathbf{X}_{n\times m} = \mathbf{U}_{n\times k}\mathbf{S}_{k\times k}(\mathbf{V}_{m\times k})^T$$

where $\mathbf{X}$ is a data matrix of n documents and m terms, $\mathbf{U}$ is a lower-dimensional matrix of n documents and k latent concepts, $\mathbf{S}$ (sometimes also confusingly written with the Greek letter for it, Σ, which can look like a summation) is a k-by-k diagonal matrix with nonnegative numbers on the diagonal (the principal components, eigenvalues, or "strength" of each latent concept) and 0s everywhere else, and $\mathbf{V}^T$ is a matrix of m terms and k latent concepts.[9]

If we matrix-multiply the elements together, we can get a matrix $\mathbf{X}'$. This matrix has the same number of columns and rows as $\mathbf{X}$, but it is filled with new values,

[9] Note that officially, the decomposition is $\mathbf{U}_{n\times\mathbf{n}}\mathbf{S}_{\mathbf{n}\times\mathbf{m}}(\mathbf{V}_{m\times\mathbf{m}})^T$, which is then reduced to k by sorting the eigenvalues and setting all after the kth to 0. However, this is not aligned with the Python implementation.

reflecting the latent dimensions. That is, $\mathbf{X}'$ contains as many features as our chosen number of latent dimensions, and the rest comprises zeros.

In Python, SVD is very straightforward:

```python
from sklearn.decomposition import TruncatedSVD
k = 10

svd = TruncatedSVD(n_components=k)
U = svd.fit_transform(X)
S = svd.singular_values_
V = svd.components_
```

Code 24 SVD decomposition into 10 components.

SVD is related to another common decomposition approach, called **principal components analysis** (PCA), where we try to find the main dimensions of variation in the data. SVD is one way to do PCA, since the singular values can be used to find the dimensions of variation. PCA can be done in other ways, too, but SVD has the advantage over other methods that it does not require us to have a square matrix (i.e., the same number of rows and columns).

6.1.2 Nonnegative Matrix Factorization

Nonnegative matrix factorization (NMF) fulfills a similar role as SVD. However, it assumes that the observed n-by-m matrix $\mathbf{X}$ is composed of only two matrices, $\mathbf{W}$ and $\mathbf{H}$, which approximate $\mathbf{X}$ when multiplied together. $\mathbf{W}$ is a n-by-k matrix, i.e., it gives us a lower-dimensional view of the documents, and is therefore very similar to what we got from SVD's $\mathbf{U}$. $\mathbf{H}$ is a k-by-m matrix, i.e., it gives us a lower-dimensional view of the terms. It is, therefore, very similar to what we got from SVD's $\mathbf{V}$. The big differences from SVD are that there is no matrix of eigenvalues, nor any negative values. NMF is also not exact, so the two solutions will not be the same.

In Python, NMF is again very straightforward.

```python
from sklearn.decomposition import NMF

nmf = NMF(n_components=k, init='nndsvd', random_state=0)
W = nmf.fit_transform(X)
H = nmf.components_
```

Code 25 Applying NMF to data.

The initialization has a certain impact on the results, and using Nonnegative Double Singular Value Decomposition (NNDSVD) helps with sparse inputs.

6.2 Visualization

Once we have decomposed our data matrix into the lower-dimensional components, we can use them for visualization. If we project down into two or three dimensions, we can visualize them in 2D or 3D. An example are the clouds in Figure 13: each point represents a song, colored by the genre. The graph immediately shows that there is little to no semantic overlap between country and hip-hop. The function for plotting this is given in Code 26 below.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import colors
```

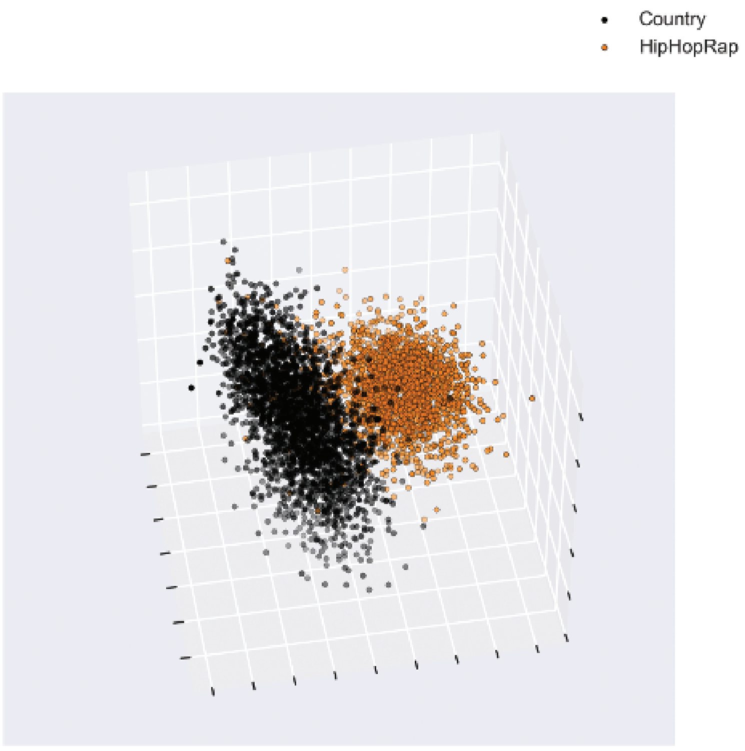


Figure 13 Scatterplot of document embeddings for song lyrics in 3D.

```

def plot_vectors(vectors, title='VIZ', labels=None, dimensions
    =3):
    # set up graph
    fig = plt.figure(figsize=(10,10))

    # create data frame
    df = pd.DataFrame(data={'x':vectors[:,0], 'y': vectors
    [:,1]})
    # add labels, if supplied
    if labels is not None:
        df['label'] = labels
        print(df.label)
    else:
        df['label'] = [''] * len(df)

    # assign colors to labels
    cm = plt.get_cmap('afmhot') # choose the color palette
    n_labels = len(df.label.unique())
    label_colors = [cm(1. * i/n_labels) for i in range(n_labels)
    ]
    cMap = colors.ListedColormap(label_colors)

    # plot in 3 dimensions
    if dimensions == 3:
        # add z-axis information
        df['z'] = vectors[:,2]
        # define plot
        ax = fig.add_subplot(111, projection='3d')
        frame1 = plt.gca()
        # remove axis ticks
        frame1.axes.xaxis.set_ticklabels([])
        frame1.axes.yaxis.set_ticklabels([])
        frame1.axes.zaxis.set_ticklabels([])

        # plot each label as scatter plot in its own color
        for l, label in enumerate(df.label.unique()):
            df2 = df[df.label == label]
            ax.scatter(df2['x'], df2['y'], df2['z'], c=
    label_colors[l], cmap=cMap, edgecolor=None, label=label,
    alpha=0.3, s=100)

    # plot in 2 dimensions
    elif dimensions == 2:
        ax = fig.add_subplot(111)
        frame1 = plt.gca()
        frame1.axes.xaxis.set_ticklabels([])
        frame1.axes.yaxis.set_ticklabels([])

        for l, label in enumerate(df.label.unique()):
            df2 = df[df.label == label]
            ax.scatter(df2['x'], df2['y'], c=label_colors[l],
    cmap=cMap, edgecolor=None, label=label, alpha=0.3, s=100)

    else:
        raise NotImplementedError()

    plt.title(title)
    plt.show()
```

Code 26 Making scatterplots of vectors in 2D or 3D, colored by label.

We can also use a projection into three dimensions to color our data: by interpreting each dimension as a **color channel** in an RGB scheme, with the first denoting the amount of red, the second the amount of green, and the third the amount of blue. We then can assign a unique color triplet to each document. These triplets can then be translated into a color (e.g., $(0, 0, 0)$ is black, since it contains no fraction of either red, green, or blue). Documents with similar colors can be interpreted as similar to each other. Since the RGB values have to be between 0 and 1, NMF works better for this application, since there will be no negative values. However, the columns still have to be scaled.

This approach was taken in Hovy and Purschke (2018), where the input matrix consisted of document embeddings of text from different cities. By placing the cities on a map and coloring them with the RGB color corresponding to the three-dimensional representation of each document vector, they created the map in Figure 14. It accurately shows the dialect gradient between Austria and Germany, with Switzerland different from either of them.

6.2.1 t-SNE

While the previous two techniques work reasonably well for visualization, they are essentially by-products of the matrix factorization. There is another dimensionality reduction technique that was specifically developed for the visualization of high-dimensional data. It is called **t-SNE** (t-distributed stochastic neighbor embedding; Maaten & Hinton, 2008).

The basic intuition behind t-SNE is that we would like to reduce the number of dimensions to something plottable. At the same time, we want to maintain the similarities between neighborhoods of points in the original high-dimensional space. That is, if two instances were similar to each other in the original space, we want them to be similar to each other in the new space. Also, they should be similar to their respective neighbors in the original space. To achieve this, t-SNE computes the probability distributions over all neighbors for a point in both dimensions. It then tries to minimize the difference between these two distributions. This effort amounts to minimizing the **Kullback-Leibler (KL) divergence**, which measures how different two probability distributions are from each other.

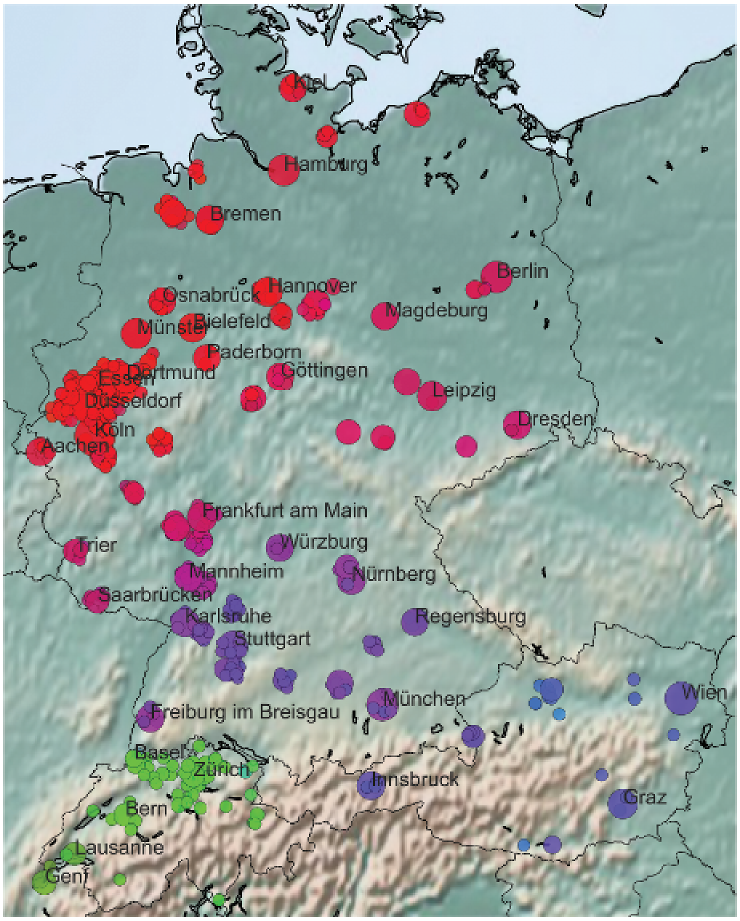


Figure 14 Visualization of documents from various German-speaking cities, colored by RGB values (Hovy & Purschke, 2018).

In practice, t-SNE can produce much more "coherent" visualizations that respect the inherent structure of the input data. For example, the widely used MNIST data set is a collection of actual hand-written numbers from letters, which is used to train visual models. We would like to have all examples of hand-written 2s close together, no matter their slant or size of the curve. At the same time, we want to have them separate from all examples of 7s, which in turn should be close to each other. t-SNE can accomplish that. However, the way t-SNE works is stochastic, so no two runs of the algorithm are guaranteed to give us the same result. It also involves several parameters that influence the outcome. It therefore requires some tuning and experimentation before we have a good visualization.

6.3 Word Descriptors

In both matrix factorization algorithms we have seen, the input data is divided into at least two elements. First, a n-by-k matrix that can be thought of as the representation of the document representation in lower-dimensional space. In SVD, this was **U**, in NMF **W**. Second, there was a k-by-n matrix that can be thought of as the representation of the words in the lower-dimensional space, if transposed. However, if we are not transposing it, we can think of it as the affinity of each word with each of the rows of k latent concepts. In SVD, this was **V**, in NMF **H**.

We can retrieve the five highest scoring columns for each of the k dimensions. The words corresponding to these five columns "characterize" each of the k latent dimensions. These word lists are similar to a probabilistic topic model. In fact, they were its precursors. However, again, these lists are only a by-product of the factorization. They are not specifically designed with the structure of language in mind. The decomposition solution determines the topics. If we do not like them, we do not have much flexibility in changing them (other than rerunning the decomposition). Probabilistic topic models, by contrast, offer all kinds of knobs and dials to adjust the outcome. Nonetheless, it is a fast and easy way to get the gist of the latent dimensions, and can help us interpret the factorization.

```python
def show_topics(a, vocabulary, topn=5):
    topic_words = ([[vocabulary[i] for i in np.argsort(t)[:-topn
    -1:-1]]
                        for t in a])
    return [', '.join(t) for t in topic_words]
```

Code 27 Retrieving the word descriptors for latent dimensions.

Here, `a` is the k-by-n matrix, `vocabulary` is the list of words (typically from the vectorizer). For Moby Dick and $k = 10$, this returns

```
['ahab, captain, cried, captain ahab, cried ahab',
 'chapter, folio, octavo, ii, iii',
 'like, ship, sea, time, way',
 'man, old, old man, look, young man',
 'oh, life, starbuck, sweet, god',
 'said, stubb, queequeg, don, starbuck',
 'sir, aye, let, shall, think',
 'thou, thee, thy, st, god',
 'whale, sperm, sperm whale, white, white whale',
 'ye, look, say, ye ye, men']
```

Table 7 Comparison of SVD and NMF

	SVD	**NMF**
Negative values (embeddings) as input?	yes	no
Number of components	3: **U**, **S**, **V**	2: **W**, **H**
Document view?	yes: **U**	yes: **W**
Term view?	yes: **V**	yes: **H**
Strength ranking?	yes: **S**	no
Exact?	yes	no
"Topic" quality	mixed	better
Sparsity	low	medium

Table 8 Comparison of SVD and NMF

	Discrete features	**Embeddings**
Latent topics	NMF	not applicable
RGB translation	NMF	SVD + scaling
Plotting	SVD	t-SNE

6.4 Comparison

The two algorithms are very similar, with only minor differences. Let's directly compare them side by side. Table 7 compares them on a variety of dimensions.

What to ultimately use depends partially on the input and the desired application/output we want, as shown in Table 8.

7 Clustering

Imagine you have a bunch of documents and suspect that they form larger groups, but you are not sure which and how. For example, say you have texts from various cities and suspect that these cities group together in dialect and regiolect areas. How can you test this? And how do you know how good your solution is?[10]

Clustering is an excellent way to find patterns in the data even if we do not know much about it, by grouping documents together based on their similarities. Imagine, for example, that you have self-descriptions of a large number

[10] This is exactly the setup in Hovy and Purschke (2018). Spoiler alert: the automatically induced clusters match existing dialect areas to almost 80 percent.

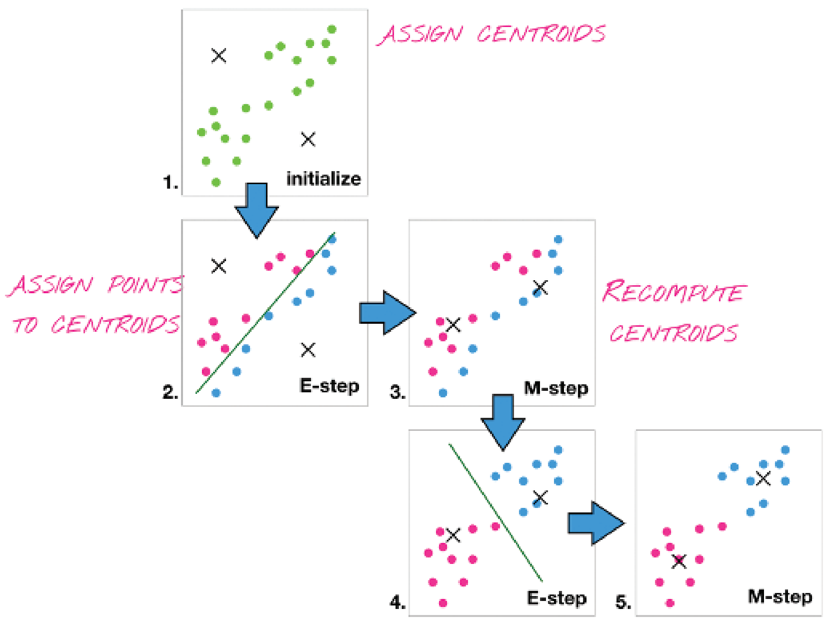


Figure 15 *k*-Means clustering.

of companies. You are trying to categorize them into a manageable number of industry sectors (technology, biology, health care, etc.). If you suspect that the language in the documents reflects these latent properties, you can use clustering to assign the companies to the sectors.

Both clustering and matrix factorization try to find latent dimensions. However, the critical difference is that matrix factorization is reversible (i.e., we can reconstruct the input), while clustering is not.

7.1 *k*-Means

***k*-Means** is one of the simplest clustering algorithms. It chooses *k* **centroids** of clusters (the means of all the document vectors associated with that cluster). It then assigns each document a score according to how similar it is to each mean.

However, we have a bit of a circular problem here: To know the cluster centroids, we need to assign the documents to the closest clusters. But to know which clusters the documents belong to, we need to compute their centroids. This situation is an example of where the EM algorithm (Dempster, Laird, & Rubin, 1977) comes in handy.

Here is how we can solve it (see Figure 15):

1. We start by randomly placing cluster centroids on the graph.

2. Then, we assign each data point to the closest cluster, by computing the distance between the cluster centroids and each point.
3. Lastly, we compute the centers of those new clusters and move the centroids to that position.

We repeat the last two steps until the centroids stop moving around.

In `sklearn`, clustering with *k*-means is very straightforward, and it is easy to get both the cluster assignment for each instance as well as the coordinates of the centroids.

```python
from sklearn.cluster import KMeans

km = KMeans(n_clusters=10, n_jobs=-1)
clusters = km.fit_predict(X)
centroids = km.cluster_centers_

```

Code 28 *k*-Means clustering with 10 clusters, using all cores.

k-Means is quite fast and scales well to large numbers of instances. However, it has the distinct disadvantage of being stochastic. Because we set the initial centroids at random, we will get slightly different outcomes every time we run the clustering. If we happen to know something about the domain, we can instead use these points as initial cluster centroids. For example, if we know that certain documents are good exemplars of a suspected cluster, we can let the algorithm start from them.

7.2 Agglomerative Clustering

Rather than dividing the space into clusters and assigning data points wholesale, we can also build clusters from the bottom up. We start with each data point in its own cluster and then merge them, again and again, to form ever-larger groups, until we have reached the desired number of clusters. This procedure is exactly what **agglomerative clustering** does, and it often mirrors what we are after in social sciences, where groups emerge, rather than spontaneously/randomly form.

The question here is, of course: *how do we decide which two clusters to merge at each step*? This condition is called the **linkage criterion**. There are several conditions, but the most common is called **Ward clustering**, where we choose the pair of clusters that minimizes the variance. Other choices include minimizing the average distance between clusters (called **average linkage criterion**), or minimizing the maximum distance between clusters (called **complete linkage criterion**). In `sklearn`, the default setting is Ward linkage.

```python
from sklearn.cluster import AgglomerativeClustering

agg = AgglomerativeClustering(n_clusters=10, n_jobs=-1)
clusters = agg.fit_predict(X)
```

Code 29 Agglomerative clustering of 10 clusters with Ward linkage.

Agglomerative clustering does not give us cluster centroids, so if we need them, we have to compute them ourselves:

```python
import numpy as np
centroids = np.array([X[clusters == c].mean(axis=0).A1 for c in
    range(k)])
```

Code 30 Computing cluster centroids for agglomerative clustering.

We can further influence the clustering algorithm by providing information about how similar the data points are to each other before clustering. If we do that, clusters (i.e., data points) that are more similar to each other than others are merged earlier. Providing this information makes sense if our data points represent entities that are comparable to each other. For example, similar documents (i.e., about the same topics), people that share connections, or cities that are geographically close to each other. This information is encoded in a **connectivity matrix**. We can express the connectivity in binary form, i.e., either as 1 or 0. This hard criterion is telling the algorithm to either merge or never merge two data points. This notion is captured by an **adjacency matrix**: two countries either share a border, or they don't. If instead, we use continuous values in the connectivity matrix, we express the degree of similarity. Gradual similarity is the case, for example, in expressing the distance between cities.

In `sklearn`, we can supply the connectivity matrix as a `numpy` array when initializing the clustering algorithm, using `connectivity`.

```python
agg = AgglomerativeClustering(n_clusters=10, n_jobs=-1,
    connectivity=C)
clusters = agg.fit_predict(X)
```

Code 31 Agglomerative clustering of 10 clusters with Ward linkage and adjacency connectivity.

The only thing we need to ensure is that the connectivity matrix is square (i.e., it has as many rows as columns), and the dimensionality of both is equal to the number of documents, i.e., the connectivity matrix is a n-by-n matrix.

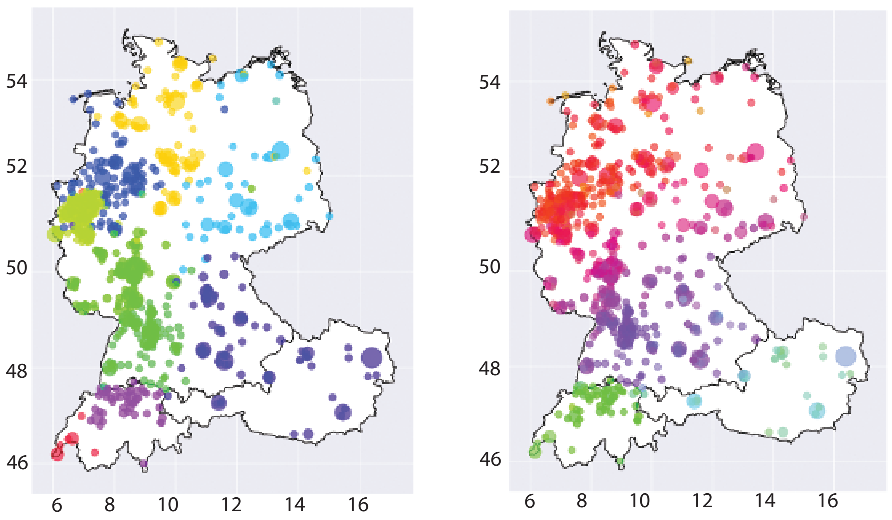


Figure 16 Example of clustering document representations of German-speaking cities with (a) *k*-means and (b) agglomerative clustering with a distance matrix into 11 clusters.

Table 9 Comparison of *k*-means and agglomerative clustering

	***k*-Means**	**Agglomerative**
Scalable	yes	no (up to about 20,000)
Repeatable result	no	yes
Include external info	sort-of (initialization)	yes
Good on dense clusters?	no	yes

Figure 16 shows the effect of different clustering solutions on a data set from Hovy and Purschke (2018).

7.3 Comparison

There are very clear trade-offs between the two clustering algorithms, as shown in Table 9.

Under most conditions, agglomerative clustering is the better choice. That is, until we have too many instances (say, more than 20,000). In those cases, it can become prohibitively slow, due to the many comparisons we have to make in each step. If we run into this problem, we might use *k*-means. However, we can initialize it with the cluster centroids derived via agglomerative clustering from a subset of the data. That way, we get a bit of the best of both worlds: stable, repeatable results, and fast convergence.

7.4 Choosing the Number of Clusters

So far, we have only compared which *method* is better for the data we have, but have quietly assumed that some outside factor determines the number of clusters. In many cases, though, the exact number of clusters is yet another parameter we need to choose.

To choose the optimal number of clusters, we can use various methods, each with their pros and cons. Here, we will focus on the **Silhouette method**. We fit different models with increasing numbers of clusters and measure the effect of the resulting solution on the Silhouette score. We select the number of clusters with the highest score. This score is composed of two parts. The first of them measures how compact each cluster is (i.e., the average distance of each instance in a cluster to all of its other members). The other score measures how well separated the individual clusters are (i.e., how far away each instance is on average from the nearest example in another cluster). The final metric is the average of this score across all cases in our data.

In practice, we can implement this as follows:

```python
from sklearn.metrics import silhouette_score

scores = []
for n_clusters in range(20):
    agg = AgglomerativeClustering(n_clusters=n_clusters, n_jobs
    =-1)
    clusters = agg.fit_predict(X)

    scores.append(silhouette_score(X, clusters))
```

Code 32 Computing the Silhouette score for 20 different models.

We can then select the number of clusters corresponding to the highest score.

7.5 Evaluation

To test the performance of our clustering solutions, we can compare it to some known grouping (if available). In that case, we are interested in how well the clustering lines up with the actual groups. We can look at this overlap from two sides, and there are two metrics to do so: **homogeneity** and **completeness**.

Homogeneity measures whether a cluster contains only data points from a single gold standard group. Completeness measures how many data points of a gold standard group are in the same cluster. The difference is the focus. For the example of matching dialect regions, homogeneity is high if a cluster occurs only in one region. At the same time, completeness is high if the region contains

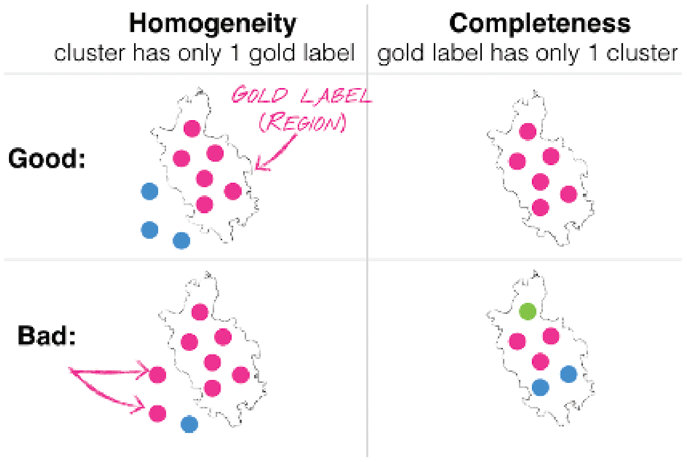


Figure 17 Homogeneity and completeness metrics for clustering solutions compared to a gold standard.

only points from one cluster. Figure 17 shows an example of matching clusters to dialect regions.

We can also take a harmonic mean between the two metrics to report a single score. This metric is called the **V-score**. It corresponds to the precision/recall/F1 scores commonly used in classification tasks.

```python
from sklearn.metrics import homogeneity_completeness_v_measure

h, c, v = homogeneity_completeness_v_measure(known, clusters)
```

Code 33 Computing homogeneity, completeness, and V-score for a clustering solution against the known grouping.

8 Language Models

Imagine a copywriter has produced several versions of a text advertisement. We want to investigate the theory that creative directors pick the most creative version of an ad for the campaign. We have the different draft versions, we have the final campaign, but how do we measure creativity? This is a big question, but if we just want a simple measure to compare texts, we could rank which of the texts are most unusual, compared to some standard reference. This is what **language models** do: for any text we give them, they return a probability of how likely/unusual this text is (with respect to some standard). In our example, we can use language models to assign each ad version a probability. A lower probability roughly corresponds to a more creative text (or at least one that is unlike most things we have seen). We can now check whether the texts with the lowest probabilities are indeed the ones that are picked most often by the creative director for the campaign, which would give us a way of measuring and quantifying "creativity."

The interesting question is of course: "How do we compute the probability of a sentence?" There are indefinitely many possible sentences out there: you probably have produced several today that have never been uttered before in the history of ever. How would a model assign a probability to something infinitely large? We cannot just count and divide: after all, any number divided by infinity is going to be 0. In order to get around this problem, we divide a sentence into a sequence of smaller parts, compute their probabilities, and combine them. Formally, a language model is a probability function that takes any document S and a reference corpus θ and returns a value between 0 and 1, denoting how likely S is to be produced under the parameters θ.

Language models have a long history in NLP and speech processing (Jelinek & Mercer, 1980; Katz, 1987), because they are very useful in assessing the output quality of a system: it gives us a fast and easy way to rank the different possible outputs produced by a chat bot, or the best translation from a machine translation system, or a speech recognizer. Today, the best language models are based on neural networks (Bengio, Ducharme, Vincent, & Jauvin, 2003; Mikolov, Karafiát, Burget, Černockỳ, & Khudanpur, 2010, inter alia), but they tend to require immense amounts of data. For most of our purposes here, a probability-based *n*-gram language model will work well.

More specifically, we will build a trigram LM based on **Markov chains**. Markov chains are a special kind of **weighted directed graphs** that describe how a system changes over time. Each state in the system is typically a point in discrete "time," so in the case of language, each state is a word. Markov chains have been used for weather prediction, market development, generational changes, speech recognition, NLP, and many other applications. They are also the basis of HMMs (hidden Markov models) and CRFs (conditional random fields), which are widely used in speech recognition and NLP. Lately, they have been replaced in many of these applications by neural language models, which produce better output. However, probabilistic language models based on Markov chains are a simple and straightforward model to assess the likelihood of a document, and to generate short texts. In general, they assume that the probability of seeing a sequence (e.g., a sentence) can be decomposed into the product of a series of smaller probabilities:

$$P(w_1, w_2, ..., w_n) = \prod_{i}^{N} P(w_i|w_1, ...w_{i-1})$$

That is, the probability of seeing a sentence is the probability of seeing each word in the sentence preceded by all others before it.

8.1 The Markov Assumption

In reality, this is a good idea. Each state or word depends on a long history of previous states: the weather yesterday and the day before influences the weather today, and the words you have read up to this one influence how you interpret the whole sentence. *Ideally*, we want to take the entire history up to the current state into account.

However, this quickly leads to exponentially complex models as we go on (just imagine the amount of words we would have to keep in memory for the last words in this sentence). The **Markov assumption** limits the history of each state to a fixed number of previous states. This limited history is called the **Markov order**. In a zeroth-order model, each state only depends on itself. In a first order model, each state depends on its direct predecessor. In a second order model, each state depends on its two previous states.

Of course this is a very simplifying assumption, and in practice, the higher the order, the better the model, but higher orders also produce sparser probability tables. And while there are some cases of long-range dependencies (for example, when to open and close brackets around a very long sentence like this one), most of the important context is in the context words right before our target (at least in English).

8.2 Trigram LMs

In a trigram LM, we use a second order Markov chain. We first extract trigram counts from a reference corpus and then estimate trigram probabilities from them. In practice, it is enough to store the raw counts and compute the probabilities "on the fly."

A trigram (i.e., an n-gram with $n = 3$) is a sequence of 3 tokens, e.g., "eat more pie." Or, more general: $u\ v\ w$. A *trigram language model* (or trigram model) takes a trigram and predicts the last word in it, based on the first two (the history), i.e., $P(w|u, v)$. In general, the history or order of an n-gram model is $n - 1$ words. Other common ns for language models are 5 or 7. The longer the history, the more accurate the probability estimates.

We will follow the notation in Mike Collins' lecture notes.[11] We are going to build a trigram language model that consists of a vocabulary $\mathcal{V}$ and a parameter $P(w|u, v)$ for each trigram $u\ v\ w$. We will define that the last token in a sentence

[11] www.cs.columbia.edu/∼mcollins/lm-spring2013.pdf.

X is STOP, i.e., x_n=STOP. We will also prepend $n-1$ dummy token to each sentence, so we can count the first word, i.e., in a trigram model, x_0 and x_{-1} are set to $*$.

We first have to collect the necessary building blocks (the trigram counts) for the LM from a corpus of sentences. This corpus determines how good our LM is: bigger corpora give us better estimates for more trigrams. However, if we are only working within a limited domain (say, all ads from a certain agency), and want to rank sentences within that, the size of our corpus is less important, since we will have seen all the necessary trigrams we want to score.

Take the following sentence: `* * eat more pie STOP` The 3-grams we can extract are

```
[('*', '*', 'eat'),
 ('*', 'eat', 'more'),
 ('eat', 'more', 'pie'),
 ('more', 'pie', 'STOP')]
```

The $*$ symbols allow us to estimate the probability of a word starting a sentence ($P(w|*,*)$), and the `STOP` symbol allows us to estimate what words end a sentence.

8.3 Maximum Likelihood Estimation (MLE)

The next step is to get from the trigram counts to a probability estimate for a word given its history, i.e., $P(w|u,v)$. We will here use the most generic solution and use **maximum likelihood estimation**. We estimate the parameters of the statistical model that maximize the likelihood of the observed data (which in practice simply translates into: count and divide). We define

$$P(w|u,v) = \frac{c(u,v,w)}{c(u,v)}$$

where $c(u,v,w)$ is the count of the trigram and $c(u,v)$ the number of times the history bigram is seen in the corpus.

Instead of actually storing both trigram and bigram counts, we can make use of the fact that the bigrams are a subset of the trigrams. We can **marginalize out** the count of a bigram from all the trigrams it occurs in, by summing over all those occurrences, using the following formula:

$$P(w|u,v) = \frac{c(u,v,w)}{c(u,v)} = \frac{c(u,v,w)}{\sum_z c(u,v,z)}$$

For example, in order to get the probability of $P(STOP|more,pie)$, we count *all* trigrams starting with “more pie.”

8.4 Probability of a Sentence

Now that we have the necessary elements of a trigram language model (i.e., the trigram counts (u, v, w)) and a way to use these counts to compute the necessary probabilities $P(w|u, v)$, we can use the model to calculate the probability of an entire sentence based on the Markov assumption. The probability of a sentence is the product of all the trigram probabilities involved in it:

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i|x_{i-2}, x_{i-1})$$

That is, we slide a trigram window over the sentence, compute the probability for each of them, and multiply them together. If we have a long sentence, we will get very small numbers, so it is better to use the logarithm of the probabilities again:

$$\log P(x_1, ..., x_n) = \sum_{i=1}^{n} \log P(x_i|x_{i-2}, x_{i-1})$$

Note that when using logarithms, products become sums, and division becomes subtraction, so the logarithm of the trigram probability can be in turn computed as

$$\log P(w|u, v) = \log c(u, v, w) - \log \sum_{z} c(u, v, z)$$

8.5 Smoothing

If we apply the LM only to the sentences in the corpus we used to collect the counts, we are done at this point. However, sometimes we want to compute the probability of a sentence that is *not* in our original corpus (for example, if we want to see how likely the sentences of another ad agency are).

Because language is so creative and any corpus, no matter how big, is limited, some perfectly acceptable three-word sequences will be missing from our model. This means that a large number of the possible n-grams will have a probability of 0 under our model, even though they should really get a nonzero probability. This is problematic, because any 0 count for a single trigram in our product calculation above would make the probability of the entire sentence 0 from thereon out, since any number multiplied by 0 is, well, 0:

$$P(x_1, ..., x_n) = P(x_1| * *) \cdot P(x_2| * x_1) \cdot 0 \cdot ... = 0.0$$

Therefore, we need to modify the MLE probability computation to assign some probability mass to new, unseen n-grams. This is called **smoothing**. There are numerous approaches to smoothing, i.e., **discounting** (reducing the actual observed frequency) or **linear interpolation** (computing several n-grams and combining their weighted evidence). There is a whole paper on the various methods (Chen & Goodman, 1996), and it is still an area of active research.

The easiest smoothing, however, is **Laplace-smoothing**, also known as "add-1 smoothing." We simply assume that the count of any trigram, seen or unseen, is 1, plus whatever evidence we have in the corpus. It is the easiest and fastest smoothing to implement, and while it lacks some desirable theoretical properties, it deals efficiently with new words. So, for a trigram that contains an unseen word x, our probability estimate becomes

$$P(w|u,x) = \frac{c(u,x,w)+1}{c(u,x)+1} = \frac{0+1}{\sum_z c(u,x,z)+1}$$

Here, the resulting probability is 1, which seems a bit high for an unseen trigram. We can reduce that effect by choosing a smaller smoothing factor, say, 0.001, i.e., pretending we have seen each word only a fraction of the time. As a result, unseen trigrams will have a much smaller probability.

Putting everything above together in code, we can estimate probabilities for any new sentence. Smoothing can be efficiently implemented by using the `defaultdict` data structure, which returns a predefined value for any unseen key. By storing the trigrams in a nested dictionary, we can easily sum over them to compute the bigram counts:

```python
from collections import defaultdict
import numpy as np
import nltk

smoothing = 0.001
counts = defaultdict(lambda: defaultdict(lambda: smoothing))

for sentence in corpus:
    tokens = ['*', '*'] + sentence + ['STOP']
    for u, v, w in nltk.ngrams(tokens, 3):
        counts[(u, v)][w] += 1

def logP(u, v, w):
    return np.log(counts[(u, v)][w]) - np.log(sum(counts[(u, v)
    ].values()))

def sentence_logP(S):
    tokens = ['*', '*'] + S + ['STOP']
    return sum([logP(u, v, w) for u, v, w in nltk.ngrams(tokens,
     3)])
```

Code 34 A simple trigram LM.

This code can be easily extended to deal with higher-order n-grams.

Coming back to our initial question about creativity, we could use a trained language model to assign probabilities to all the ad versions, and measure whether there is a correlation between unusual, low-probability versions and their eventual success.

8.6 Generation

Typically, Markov chain language models are used to predict future states, but we can also use it to **generate text** (language models and Markov Chains are generative graphical models). In fact, Andrey Markov first used Markov chains to predict the next letter in a novel. We will do something similar and use a Markov chain process to predict the next word, i.e., to generate sentences.

This technique has become a bit infamous, since it was used by some people to generate real-looking, but meaningless, scientific papers (that actually got accepted!), and is still used by spammers in order to generate their emails. However, there are also much more innocent use cases, such as generating real-looking nonce stimuli for an experiment, or for a lighthearted illustration of the difference between two political candidates, by letting their respective LMs "finish" the same sentence.

We can use the counts from our language model to compute a probability distribution over the possible next words, and then sample from that distribution according to the probability (i.e., if we sampled long enough from the distribution of a word $P(w|u, v)$, we would get the true distribution over the trigrams).

```python
import numpy as np

def sample_next_word(u, v):
    keys, values = zip(*counts[(u, v)].items())
    values = np.array(values)
    values /= values.sum()
    return keys[np.argmax(np.random.multinomial(1, values))]

def generate():
    result = ['*', '*']
    next_word = sample_next_word(result[-2], result[-1])
    result.append(next_word)
    while next_word != 'STOP':
        next_word = sample_next_word(result[-2], result[-1])
        result.append(next_word)

    return ' '.join(result[2:-1])
```

Code 35 A simple trigram sentence generator.

This generates simple sentences. A trigram model is not very powerful, and in order to generate real-sounding sentences, we typically want a higher order, which then also means a much larger corpus to fit the model on.

9 Topic Models

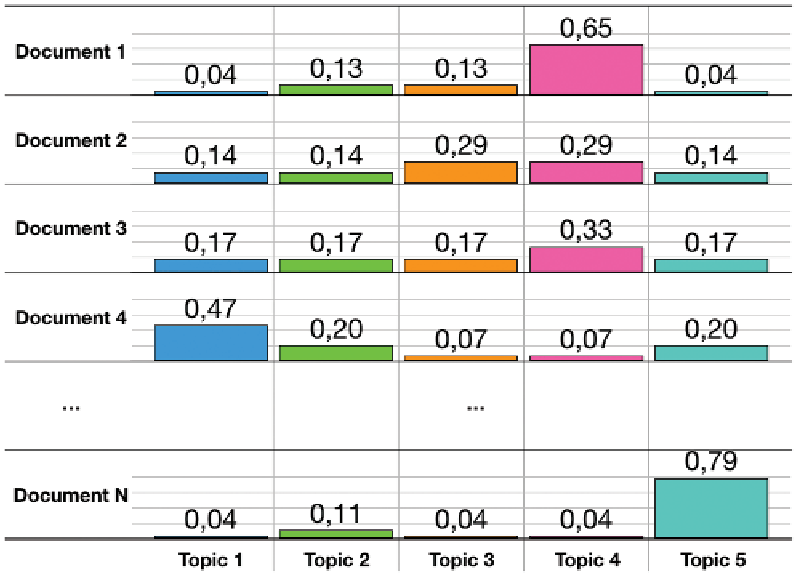
Imagine we have a corpus of restaurant reviews and want to learn what people are mostly talking about: what are their main complaints (price, quality, hygiene?), what types of restaurants (upscale, street food) are they discussing, and what cuisines (Italian, Asian, French, etc.) do they prefer? TF-IDF terms can give us individual words and phrases that hint at these things, but they do not group together into larger constructs. What we want is a way to automatically find these larger trends.

Topic models are one of the most successful and best-known applications of NLP in social sciences (not in small part due to its availability as a package in R). Their advantage is that they allow us to get a good high-level overview of the things that are being discussed in our corpus. What are the main threads, issues, and concerns in the texts?

What we ultimately want is a distribution over topics for each document. For example, our first document is 80 percent topic 1, 12 percent topic 2, and 8 percent topic 3. In order to interpret these topics, we want for each topic a distribution over words. For example, the five words most associated with topic 1 are "pasta," "red_wine," "pannacotta," "aglio," and "ragù." Based on these words, we can give it a descriptive label that sums up the words (here, for example, "Italian cuisine"). This also gives us the basic elements: words, documents, and topics.

The most straightforward way to express the intuition above mathematically is through probabilities. The three elements can be related to each other in two conditional probability tables: one that tells us how likely each word is for each topic ($P(\text{word}|\text{topic})$), and one that tells us how likely each topic is for a specific document ($P(\text{topic}|\text{document})$). In practice, the distribution of words over topics is usually just called z, and the distribution over topics for each document is called θ. We also have a separate distribution θ_i for every document, rather than one that is shared for the entire corpus. After all, we do not expect that a document about a fast-food joint hits upon the same issues as a review of the swankiest three-star Michelin place in town. Figure 18 shows the two quantities as graphical distributions.

The term "topic model" covers a whole class of **generative probabilistic models**, with plenty of variations, but the most well known is **Latent Dirichlet**



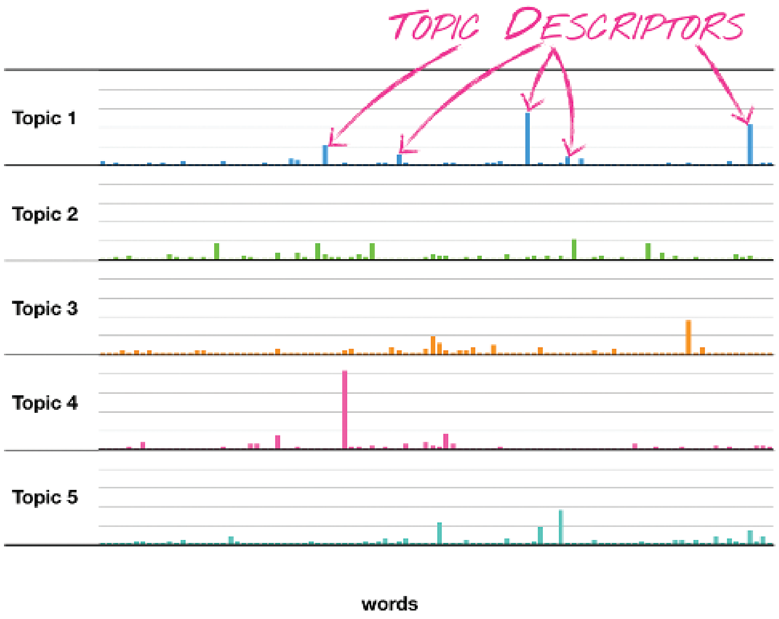


Figure 18 Graphical representation of the topic model parameters $\theta = P(\text{topic}|\text{document})$ (top) and $z = P(\text{word}|\text{topic})$ (bottom). Topic descriptors are the top k words with the highest probability for each topic in z.

Allocation (LDA) (Blei, Ng, & Jordan, 2003). So in the following, we will use the two terms interchangeably.

The operative word here is "generative": probabilistic models can be used to describe data, but they can just as well be used to *generate* data, as we have seen with language models. Topic models are very similar. In essence, a topic model is a way to imagine how a collection of documents was generated word by word. They have a Markov horizon of 0 (i.e., each word is independent from all the ones before), but instead each word is conditioned on a topic. In other words: with a topic model, rather than conditioning on the previous words (i.e., the history), we first choose a topic and then select a word based on the topic. How this is supposed to have happened is described in the **generative process**. This is why LDA is a "generative model." The steps in this generative story for how a document came to be are fairly simple: for each word in a document, we first choose the topic of that slot, and then look up which word from the topic should fill the slot. Or, more precisely, for a document d_i,

1. Set the number of words N by drawing from a **Poisson distribution**
2. Draw a multinomial topic distribution θ over k topics from a **Dirichlet distribution**
3. For $j \in N$,
   - Pick a topic t from the multinomial distribution θ
   - Choose word w_j based on its probability to occur in the topic according to the conditional probability $z_ij = P(w_j|t)$

No, this is of course not how documents are generated in real life, but it is a useful abstraction for our purposes. Because if we assume that this *is* indeed how the document was generated, then we can reverse-engineer the whole process to find the parameters that did it. And those are the ones we want. The generative story is often depicted in **plate notation** (Figure 19).
We have already seen θ and z. The first describes the probability of selecting a particular topic when sampling from a document. The second describes the chance of selecting a particular word when sampling from a given topic. Each of those probability distributions have a **hyperparameter** attached to them, α and β, respectively. α is the **topic prior**: it is the parameter for the Dirichlet process (a process that generates probability distributions) and controls how many topics we expect each document to have on average. If we expect every document to have several or all of the topics present, we choose an α value at 1.0 or above, leading to a rather uniform distribution. If we know or suspect that there are only a few topics per document (for example because the documents are quite short), we use a value smaller than 1.0, leading to a very peaked

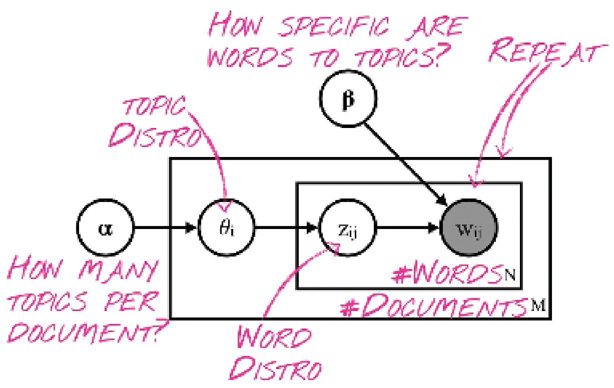

HOW SPECIFIC ARE WORDS TO TOPICS?
REPEAT
TOPIC DISTRO
β
α
θi
zij
wij
HOW MANY TOPICS PER DOCUMENT?
#WORDS N
#DOCUMENTS M
WORD DISTRO

Figure 19 Plate notation of LDA model.

distribution. β is the **word prior**: it controls how topic-specific we expect the words to be. If we think that words are very specific to a particular topic, use a small β value (0.01 or less), if we think they are general, we use values up to 1.0 We can imagine both of these parameters a bit like entropy: the lower the value, the more peaked the underlying distribution.

Now, we do not have the two tables yet, so we need to infer them from the data. This is usually done using either **Gibbs sampling** or **expectation maximization** (Dempster et al., 1977). We do not have the space here to go into depth how these algorithms work (gentle introductions to them are Resnik & Hardisty, 2010, and Hovy, 2010, respectively). Both are essentially guaranteed to find better and better parameter settings the longer we run them (i.e., the models become more and more likely to have produced the data we see). The question is of course: where do we start? In the first round, we essentially set all parameters to random values, and see what happens. Then we update the parameters in order to improve the fit with the data and run again. We repeat this until we have reached a certain number of iterations, or the parameters stop changing much from iteration to iteration, i.e., the model fits the data more and more. We can measure the fit of the model to the data (also called the **data likelihood**) by multiplying together all the probabilities involved in generating the corpus.

9.1 Caveats

The results depend strongly on the initialization and the parameter settings. There are a couple of best practices and rules of thumb (see also Boyd-Graber et al., 2014):

1. thoroughly preprocess your data: lowercase the text, use lemmatization, remove stopwords, replace numbers and user names, and join collocations (and document your choices).
2. use a minimum document frequency of words: exclude all words that are rare (a word that we have seen five times in the data could just be a name or a common misspelling). Depending on your corpus size, a document frequency of 10, 20, 50, or even 100 is a good starting point.
3. maximum document frequency of words: a word that occurs in more than 50 percent of all documents has very little discriminative power, so you might as well remove it. However, in order to get coherent topics, it can be good to go as low as 10 percent.
4. choose a good number of iterations: when is the model done fitting? While the model will always improve the data likelihood with more iterations, we do not want to wait indefinitely. Choosing a number that is too small, however, will result in an imperfect fit. After you are happy with the other parameter choices, extend the number of iterations and see whether the results improve.

In either case, you will have to look at your topics and decide whether they make sense. This can be nontrivial: a topic might perfectly capture a topic you are not aware of (e.g., the words "BLEU, Bert, encoder, decoder, transformer" might look like random gibberish, but they perfectly encapsulate papers on machine translation).[12] Because topics are so variable from run to run, they are not stable indicators for dependent variables: we cannot use the output of a topic model to predict something like ratings.

9.2 Implementation

In order to implement LDA in Python, there are two possibilities, `gensim` and `sklearn`. We will look at the `gensim` implementation here, and run it on a sample of wine descriptions to find out what sommeliers talk about.

```python
from gensim.models import LdaMulticore, TfidfModel
from gensim.corpora import Dictionary
import multiprocessing
```

Code 36 Code for loading topic models from `gensim`.

```python
dictionary = Dictionary(instances)
dictionary.filter_extremes(no_below=100, no_above=0.1)
```

[12] Thanks to Hanh Nguyen for the example.

```

ldacorpus = [dictionary.doc2bow(text) for text in instances]

tfidfmodel = TfidfModel(ldacorpus)

model_corpus = tfidfmodel[ldacorpus]
```

Code 37 Extracting and limiting the vocabulary and transforming the data into TF-IDF representations of the vocabulary.

```
num_passes = 10
num_topics = 20
# find chunksize to make about 200 updates
chunk_size = len(model_corpus) * num_passes/200

model = LdaMulticore(model_corpus,
                     id2word=dictionary,
                     num_topics=num_topics,
                     workers=min(10, multiprocessing.cpu_count()
    -1),
                     passes=num_passes,
                     chunksize=chunk_size
                    )
```

Code 38 Training the LDA model on the corpus on multiple cores.

This will give us a model we can then use on the corpus in order to get the topic proportions for each document:

```
topic_corpus = model[model_corpus]
```

Code 39 Transforming the corpus into topic proportions.

We can also examine the learned topics:

```
model.print_topics()
```

Code 40 Showing the topic words.

Unfortunately, these are formatted in a way that is a bit hard to read:

```
[(0,
  '0.011*"rose" + 0.010*"fruity" + 0.010*"light" + 0.009*"orange
    " + 0.009*"pink" + 0.008*"lively" + 0.008*"balance" +
    0.008*"mineral" + 0.008*"peach" + 0.007*"color"'),
  ...
]
```

We can use regular expressions to pretty this up a bit:

```
import re
topic_sep = re.compile(r"0\.[0-9]{3}\*")

model_topics = [(topic_no, re.sub(topic_sep, '', model_topic).
    split(' + ')) for topic_no, model_topic in
                model.print_topics(num_topics=num_topics,
    num_words=5)]

descriptors = []
for i, m in model_topics:
    print(i+1, ", ".join(m[:5]))
    descriptors.append(", ".join(m[:2]).replace('"', ''))
```

Code 41 Enumerate the topics and print out the top five words for each.

This gives us a much more readable format:

```
1 "rose", "fruity", "light", "orange", "pink"
2 "pretty", "light", "seem", "mix", "grow"
3 "tobacco", "close", "leather", "dark", "earthy"
4 "lime", "lemon", "green", "zesty", "herbal"
5 "pepper", "licorice", "clove", "herb", "bright"
6 "toast", "vanilla", "grill", "meat", "jam"
7 "jammy", "raspberry", "chocolate", "heavy", "cola"
8 "age", "structure", "year", "wood", "firm"
9 "pear", "apple", "crisp", "white", "attractive"
```

9.3 Selection and Evaluation

The first question with topic models is of course: *How many topics should I choose?* This is a good question, and in general, more likely models should also have more coherent and sensible topics, but that is a rough and tenuous equivalence. Which brings us to the question of **topic coherence**. Ideally, we want topics that are “self-explanatory,” i.e., topics that are easy to interpret by anyone who sees them. A straightforward way to test this is through an **intrusion test**: take the top five words of a topic, replace one with a random word, and show this version to some people. If they can pick out the random intruder word, then the other words must have been coherent enough as a topic to make the intruder stick out. By asking several people and tracking how many guessed correctly, we can compute a score for each topic. Alternatively, we can use easy-to-compute measures that correlate with coherence for all pairs of the topic descriptor words, to see how likely we are to see these words together, as opposed to randomly distributed.

There are many measures we can use (Stevens, Kegelmeyer, Andrzejewski, & Buttler, 2012), but here, we will focus on two: the UMass evaluation score (Mimno, Wallach, Talley, Leenders, & McCallum, 2011), which uses the log probability of word co-occurrences, and the C_v score (Röder, Both, & Hinneburg, 2015), which uses the normalized pointwise mutual information and cosine similarity of the topic words. We can use these measures to compute a coherence score for each number of topics on a subset of the data, and then choose the number of topics that gave us the best score on both (or on the one we prefer).

In Python, we can choose these with the `CoherenceModel`:

```python
from gensim.models import CoherenceModel

coherence_values = []
model_list = []
for num_topics in range(5, 21):
    print(num_topics)
    model = LdaMulticore(corpus=sample, id2word=dictionary,
    num_topics=num_topics)
    model_list.append(model)
    coherencemodel_umass = CoherenceModel(model=model, texts=
    test_sample, dictionary=dictionary, coherence='u_mass')

    coherencemodel_cv = CoherenceModel(model=model, texts=
    test_sample, dictionary=dictionary, coherence='c_v')

    coherence_values.append((num_topics, coherencemodel_umass.
    get_coherence(), coherencemodel_cv.get_coherence()))
```

Code 42 Choosing topic number via coherence scores.

That will give us a relatively good idea of the best number of topics, as we can see in Figure 20.

Even after we settle on a number of topics, there is still quite a lot of variation. If we run five topic models with the same number of topics on the same data, we will get five slightly different topic models, because we initialize the models randomly. Some topics will be relatively stable between these models, but others might vary greatly. So how do we know which model to pick? The data likelihood of a model can also be used to calculate the **entropy** of the model, which in turn can be used to compute the **perplexity**. We can compare these numbers and select the model with the best perplexity. Entropy and perplexity are model-inherent measures, i.e., they only depend on the parameters of the model. That means we can use a grid search over all the possible combination of parameter values to find the best model.

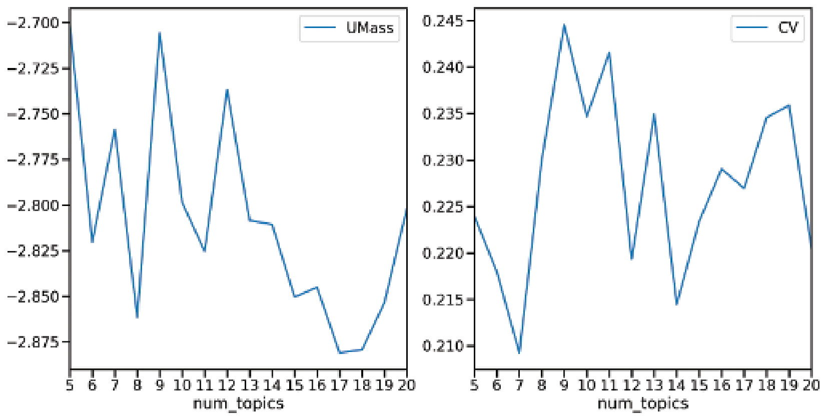


Figure 20 Coherence scores for different numbers of topics.

9.4 Adding Structure

A popular variant of LDA is the **structural topic model** (STM) by Roberts et al. (2013). This model has become very popular in computational social science, since we often have access to external factors (covariates) associated with the text, such as the author of the text, when it was written, or which party published it. More often than not we are interested in the effects of these factors on our topics. A simple way is to aggregate the topic distributions by each of these factors. However, a more principled approach is to algorithmically model the influence of the external factors on the topic distributions.

The original STM is again only available as an `R` package, but `gensim` added a version of the structural topic model as **author topic model**. The interface is essentially the same as for the regular LDA model, with one addition: we need to provide a dictionary that maps from each value of our external factor to the documents associated with it. In the original context, this is a mapping from scientific authors to the papers they were involved in writing. In our example, let’s use the country of origin for each wine as covariate: that way, we can learn what the dominant flavor profiles are for each country (see Figure 21). As we can see, each category receives a dominant topic.

The implementation in Python is very similar to the regular LDA model. The only thing we need to do before we can start training, is to map each covariate (here, a country) to the index of all documents it is associated with. In our case, each document has only one covariate, but we can easily associate each covariate with a number of documents (for example, if we have scientific papers written by several authors).

```python
from collections import defaultdict
```

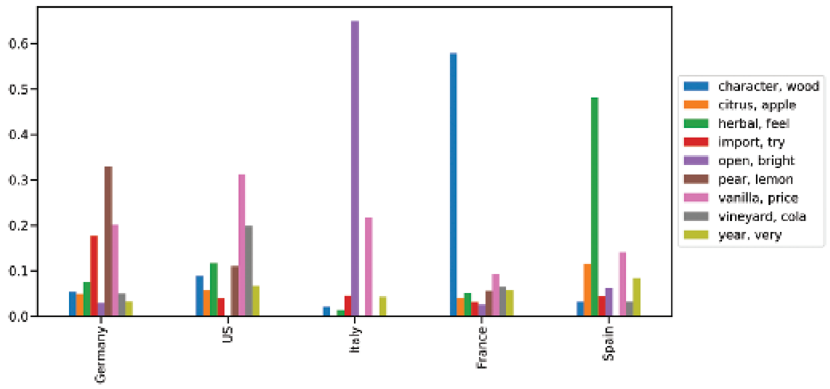


Figure 21 Topic distribution by country.

```python
author2doc = defaultdict(list)

for i, country in enumerate(df.country):
    author2doc[country].append(i)

```

Code 43 Creating the mapping from covariates to document indices.

Once we have this mapping, we can run the model.

```python
from gensim.models import AuthorTopicModel
from gensim.test.utils import temporary_file

with temporary_file("serialized") as s_path:
    author_model = AuthorTopicModel(
        model_corpus,
        author2doc=author2doc,
        id2word=dictionary,
        num_topics=9,
        serialized=True,
        serialization_path=s_path,
        alpha=0.5
    )

    author_model.update(model_corpus, author2doc)
```

Code 44 Running the Author LDA model.

The results we get are similar to the regular LDA model (again, we use regular expressions to clean things up a bit):

```
1 "raspberry", "not", "make", "vanilla", "chocolate"
2 "tone", "bright", "back", "velvety", "smooth"
3 "open", "feel", "tobacco", "deliver", "close"
4 "character", "age", "structure", "wood", "fruity"
5 "green", "apple", "zesty", "herbal", "body"
6 "peach", "honey", "lemon", "apple", "pear"
7 "flesh", "bouquet", "mouth", "mingle", "rind"
8 "pair", "mouthfeel", "white", "easy", "exotic"
9 "cheerful", "guava", "mouthful", "underripe", "blast"
```

As we have seen, if we visualize these topic proportions per country (see Figure 21), the countries each get a different profile, corresponding to their dominant styles of wine.

9.5 Adding Constraints

Even when setting the hyperparameters well and having enough data to fit, there is a good chance that topics which we know should be separate are merged by the model. One way to keep them apart is to tinker with z, by adding priors to some of the words. This will nudge the model toward some distributions and away from others. In practice, we are preventing the model from exploring some parameter configurations that we know will not lead to a solution we want. Jagarlamudi, Daumé III, and Udupa (2012) introduced a straightforward implementation of such a **guided LDA**.

Blodgett, Green, and O'Connor (2016) used a fixed prior distribution over topics based on the known distribution of different ethnicities in the cities they investigated. By using the same number of topics as there were ethnicities in the data, they were able to capture ethnolects and regional terms.

9.6 Topics versus Clusters

A relatively frequent question at this point is, Why should I use a topic model if I could just cluster word embeddings? The two approaches do indeed produce similar results: a separation of the data into larger semantic groups. The differences are somewhat subtle: word embeddings capture semantic similarity within the defined window size, topic models take the entire document into account. If the documents are short (e.g., tweets, reviews) and the window size of the embedding model is sufficiently wide, the two are equivalent. However, with longer document types, word embeddings capture only general semantic fields (i.e., what are all the things people talk about in the data), not any specific to a particular document (i.e., what is the distribution of things people talk about in each document). On the other hand, clustering embeddings is relatively stable and will produce the same results every time, whereas the output of topic models varies with every run.

Lately, there has been an increased interest in neural (rather than probabilistic) topic models. They seem to be more flexible, result in more coherent topics, and are applicable to multiple languages (Das, Zaheer, & Dyer, 2015; Srivastava & Sutton, 2017; Dieng, Ruiz, & Blei, 2019; Bianchi, Terragni, & Hovy, 2020). However, there are no widely available Python libraries — yet.

Appendix A

English Stopwords

a, about, above, across, after, afterwards, again, against, all, almost, alone, along, already, also, although, always, am, among, amongst, amount, an, and, another, any, anyhow, anyone, anything, anyway, anywhere, are, around, as, at, back, be, became, because, become, becomes, becoming, been, before, beforehand, behind, being, below, beside, besides, between, beyond, both, bottom, but, by, ca, call, can, cannot, could, did, do, does, doing, done, down, due, during, each, eight, either, eleven, else, elsewhere, empty, enough, even, ever, every, everyone, everything, everywhere, except, few, fifteen, fifty, first, five, for, former, formerly, forty, four, from, front, full, further, get, give, go, had, has, have, he, hence, her, here, hereafter, hereby, herein, hereupon, hers, herself, him, himself, his, how, however, hundred, i, if, in, indeed, into, is, it, its, itself, just, keep, last, latter, latterly, least, less, made, make, many, may, me, meanwhile, might, mine, more, moreover, most, mostly, move, much, must, my, myself, name, namely, neither, never, nevertheless, next, nine, no, nobody, none, noone, nor, not, nothing, now, nowhere, of, off, often, on, once, one, only, onto, or, other, others, otherwise, our, ours, ourselves, out, over, own, part, per, perhaps, please, put, quite, rather, re, really, regarding, same, say, see, seem, seemed, seeming, seems, serious, several, she, should, show, side, since, six, sixty, so, some, somehow, someone, something, sometime, sometimes, somewhere, still, such, take, ten, than, that, the, their, them, themselves, then, thence, there, thereafter, thereby, therefore, therein, thereupon, these, they, third, this, those, though, three, through, throughout, thru, thus, to, together, too, top, toward, towards, twelve, twenty, two, under, unless, until, up, upon, us, used, using, various, very, via, was, we, well, were, what, whatever, when, whence, whenever, where, whereafter, whereas, whereby, wherein, whereupon, wherever, whether, which, while, whither, who, whoever, whole, whom, whose, why, will, with, within, without, would, yet, you, your, yours, yourself, yourselves

Appendix B

Probabilities

In some cases, we will represent words as probabilities. In the case of topic models or language models, this allows us to reason under uncertainty. However, there is an even simpler reason to use probabilities: keeping track of frequencies is often not too practical for our purposes. In one of our previous examples, some quarterly reports might be much longer than others, so seeing "energy" mentioned more often in those longer reports than in shorter ones is not too informative by itself. We need to somehow account for the length of the documents.

For this, normalized counts, or probabilities, are a much better indicator. Of course, probabilities are nothing more than counts normalized by a **Z-score**. Typically, those Z-scores are the counts of all words in our corpus,

$$P(w) = \frac{count(w)}{N}$$

where N is the total count of words in our corpus.

A **probability distribution** over a vocabulary therefore assigns a probability between 0.0 and 1.0 to each word in the vocabulary. And if we were to sum up all probabilities in the vocabulary, the result would be 1.0 (otherwise, it is not a probability distribution).

Technically, probabilities are continuous values. However, since each probability is a discrete feature (i.e., it "means" something) in a feature vector, we cover them here under discrete representations.

Say we have a corpus of 1,000 documents, x is "natural" and occurs in 20 documents, and y is "language" and occurs in 50 documents.

The probability $\mathbf{P(x)}$ of seeing a word x if we reach into a big bag with all words in our vocabulary is therefore simply the number of documents that contain x, say, "natural" (here, 20), divided by the number of all documents in our corpus (1,000):

$$P(natural) = \frac{count(documents\ w.\ \text{“}natural''\text{)}}{number\ of\ all\ documents} = \frac{20}{1000} = \frac{1}{50} = 0.02$$

$$P(language) = \frac{count(documents\ w.\ \text{“}language''\text{)}}{number\ of\ all\ documents} = \frac{50}{1000} = \frac{1}{20} = 0.05$$

Note that probabilities are not percentages (even though they are often used that way in everyday use)! In order to express a probability in percent, you need to multiply it by 100. We therefore have a 2 percent probability of randomly drawing the word "natural" and a 5 percent chance of drawing "language."

As we have seen, though, word frequencies follow a Zipf distribution, so the most frequent words get a lot of the probability mass, whereas most of the words get a very small probability. Since this can lead to vanishingly small numbers that computers sometimes can not handle, it is common to take the logarithm of the probabilities (also called **log-probabilities**). Note that in log-space, multiplication becomes addition, and division becomes subtraction. Therefore, an alternative way to compute the (log) probability of a word is to subtract the log-count of the total number of words from the log-count of the word count:

$$\log P(w) = \log count(w) - \log N$$

To get a normal probability back, we have to exponentiate, but this can cause a loss of precision. It is therefore usually better to store raw counts and convert them to log-probabilities as needed:

$$\log P(natural) = \log 20 - \log 1000 = \log 1 - \log 50 = -3.912023005428146$$

B1 Joint Probability

$P(x, y)$ is a **joint probability**, i.e., how likely is it that we see x and y together, that is, the words in one document ("natural language," "language natural," or separated by other words, since order does not matter in this case). Say we have 10 documents that contain both words; then

$$P(natural, language) = \frac{count(documents\ w.\ \text{``}natural\ and\ \text{``}language)}{number\ of\ all\ documents}$$
$$= \frac{10}{1000} = \frac{1}{100} = 0.01$$

B2 Conditional Probability

Words do not occur in isolation but in context, and in certain contexts, they are more likely than in others. If you hear the sentence "Most of the constituents mistrusted the ...," you expect only a small number of words to follow. "Politician" is much more likely than, say, "handkerchief." That is, the probability of "politician" in this context is much higher than that of "handkerchief." Or, to put it in mathematical notation,

$$P(politician|CONTEXT) > P(handkerchief|CONTEXT)$$

$P(y|x)$ is a **conditional probability**, i.e., how likely we are to see y after having seen x.

Let's reduce the context to one word for now and look at the words in our previous example, i.e., "language" in a document that contains "natural." We can compute this as

$$P(language|natural) = \frac{P(x,y)}{P(x)} = \frac{\frac{count(documents\ w.\ \text{``}natural\ and\ \text{``}language)}{number\ of\ all\ documents}}{\frac{count(documents\ w.\ \text{``}natural)}{number\ of\ all\ documents}}$$

$$= \frac{count(documents\ w.\ \text{``}natural\ and\ \text{``}language)}{count(documents\ w.\ \text{``}natural)}$$

$$= \frac{10}{20} = \frac{1}{2} = 0.5$$

Note that order *does* matter in this case! So the corresponding probability formulation for seeing "natural" in a document that contains "language," or $P(x|y) = \frac{P(x,y)}{P(y)}$, is something completely different (namely, 0.2).

B3 Probability Distributions

So far, we have treated probabilities as normalized counts, and for the most part, that works well. However, there is a more general way of looking at probabilities, and that is as functions. They take as input an event (a word or some other thing) and return a number between 0 and 1. In the case of normalized counts, the function itself was a lookup table, or maybe a division. However, we can also have probability functions that are defined by an equation and that take a number of parameters other than the event we are interested in.

The general form of these probability functions is

$$f(x;\theta)$$

where x is the event we are interested in and θ is just shorthand for any of the additional parameters.

What makes these functions probability distributions are the facts that they

1. cover the entire sample space (for example all words)
2. sum to 1.0 (if we add up the probabilities of all words)

In the case of texts, we mostly work with **discrete probability distributions**, i.e., there is a defined number of events, for example, words. When we graph these distributions, we usually use bar graphs. There are also continuous probability distributions, the most well known of which is the normal or Gaussian distribution. However, for the sake of this Element, we will not need them. In the following, we will look at some of the most common discrete probability distributions.

B3.1 Uniform distribution

The simplest distribution is the one where all events are equally likely, so all we have to return is 1 divided by the number of outcomes:

$$U(x; N) = \frac{1}{N}$$

A common example is a die roll: all numbers are equally likely to come up. This is also a good distribution to use if we do not know anything about the data and do not want to bias our model unduly.

B3.2 Bernoulli distribution

This is the simplest possible discrete distribution: it only has two outcomes and can be described with a single parameter q, which is the chance of success. The other outcome is simply $1 - q$:

$$Pr(x; q) = \begin{cases} 1 - q & if\, x = 0 \\ q & if\, x = 1 \end{cases}$$

If $q = 0.5$, the Bernoulli distribution is also a uniform distribution.

B3.3 Multinomial Distribution

Most often, we will deal with distributions that are much larger than two outcomes and much more irregular than uniform. Formally, it is parameterized by a single parameter θ, which is a vector with all probabilities. The mathematical definition is

$$P(x; \theta) = \prod_{j=1}^{1} \theta_j^{I(x_j)=1}$$

which is a very complicated way of writing "use the value at the vector position for x."

A good example is the distribution over characters in a sample of text. In Figure 22, we can see how different characters have very different probabilities.

In fact, if we sort them by their likelihood, we can see that they rapidly decline. This is another example of Zipf's law, and can be modeled with a Zeta distribution, which we will not go into here.

B3.4 Dirichlet Distribtution

A **Dirichlet distribution** is a distribution over multinomial distributions and has only one parameter, called α. We can imagine a Dirichlet distribution as a generator function that gives us multinomial distributions over k outcomes for

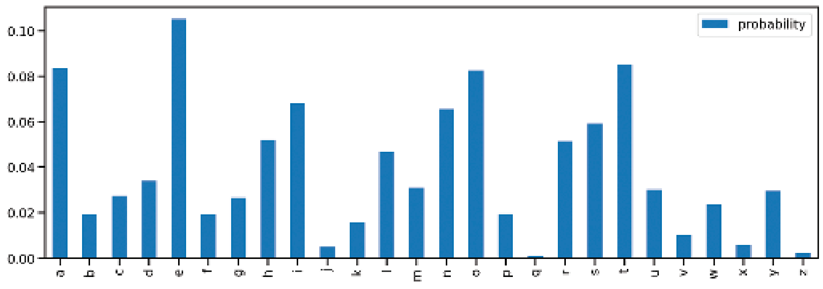


Figure 22 Multinomial distribution over lowercase characters.

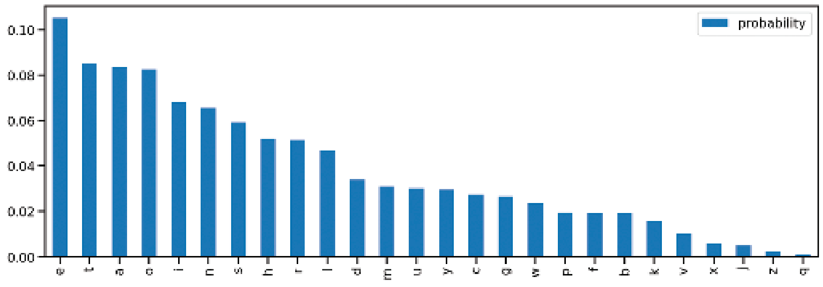


Figure 23 Sorted multinomial distribution over lowercase characters.

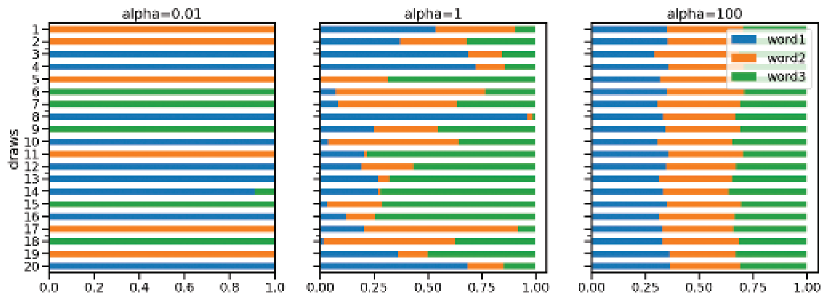


Figure 24 Twenty multinomial distributions over three outcomes drawn from Dirichlets differing in α.

each document. The parameter α controls how peaked or uniform the multinomial distributions are: if α is close to 0, the distributions are very peaked, i.e., one outcome will have (almost) all the probability mass. The larger α gets, the more uniform the distributions become. Figure 24 shows the effect over 20 draws from a Dirichlet over three outcomes with different values for α.

References

Antoniak, M., & Mimno, D. (2018). Evaluating the stability of embedding-based word similarities. *Transactions of the Association for Computational Linguistics, 6*, 107–119.

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3, 1137–1155.

Bhatia, S. (2017). Associative judgment and vector space semantics. *Psychological Review, 124*(1), 1.

Bianchi, F., Terragni, S., & Hovy, D. (2020). *Pre-training is a hot topic: Contextualized document embeddings improve topic coherence*. arXiv preprint arXiv:2004.03974.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3, 993–1022.

Blodgett, S. L., Green, L., & O'Connor, B. (2016). Demographic dialectal variation in social media: A case study of African-American English. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics. Stroudsburg, PA. (pp. 1119–1130).

Boyd-Graber, J., Mimno, D., & Newman, D. (2014). Care and feeding of topic models: Problems, diagnostics, and improvements. In E. M. Airoldi, D. Blei, E. A. Erosheva, & S. E. Fienberg (Eds.), *Handbook of mixed membership models and their applications.* Boca Raton, FL: CRC Press, pp. 225–254

Chen, S. F., & Goodman, J. (1996). *An empirical study of smoothing techniques for language modeling*. Paper presented at the 34th annual meeting of the Association for Computational Linguistics. Retrieved from `http://aclweb.org/anthology/P96-1041`

Chollet, F. (2017). *Deep learning with Python.* Manning, Shelter Island, NY.

Crystal, D. (2003). *The Cambridge encyclopedia of the English language* (3rd ed.). Cambridge, England: Cambridge University Press.

Das, R., Zaheer, M., & Dyer, C. (2015). Gaussian LDA for topic models with word embeddings. In *Proceedings of the 53rd annual meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing: Vol. 1. Long papers*. Association for Computational Linguistics. Stroudsburg, PA. (pp. 795–804).

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science, 41*(6), 391–407.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological), 39(1)*, 1–22.

Denny, M. J., & Spirling, A. (2018). Text preprocessing for unsupervised learning: Why it matters, when it misleads, and what to do about it. *Political Analysis, 26*(2), 168–189.

Dieng, A. B., Ruiz, F. J., & Blei, D. M. (2019). *Topic modeling in embedding spaces*. arXiv preprint arXiv:1907.04907.

Eisenstein, J. (2019). *Introduction to natural language processing*. Cambridge, MA: MIT Press.

Evans, J. A., & Aceves, P. (2016). Machine translation: Mining text for social theory. *Annual Review of Sociology, 42*, 21–50.

Firth, J. R. (1957). A synopsis of linguistic theory, 1930–1955. *Studies in Linguistic Analysis*. Basil Blackwell, Oxford. pp 1–32. Volume 1

Fromkin, V., Rodman, R., & Hyams, N. (2018). *An introduction to language*. Cengage Learning. Wadsworth. Boston, MA.

Garg, N., Schiebinger, L., Jurafsky, D., & Zou, J. (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences, 115*(16), E3635–E3644.

Gentzkow, M., Kelly, B. T., & Taddy, M. (2017). *Text as data* (technical report). Washington, DC: National Bureau of Economic Research.

Goldberg, Y. (2016). A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research, 57*, 345–420.

Goldberg, Y. (2017). Neural network methods for natural language processing. Edited by Graeme Hirst. Morgan & Claypool. San Rafael, CA, *Synthesis Lectures on Human Language Technologies, 10*(1), 1–309.

Goldberg, Y., & Levy, O. (2014). *word2vec Explained: Deriving Mikolov et al.'s negative-sampling word-embedding method*. arXiv preprint arXiv: 1402.3722.

Grave, E., Bojanowski, P., Gupta, P., Joulin, A., & Mikolov, T. (2018). *Learning word vectors for 157 languages*. Paper presented at the International Conference on Language Resources and Evaluation (LREC 2018).

Grimmer, J., & Stewart, B. M. (2013). Text as data: The promise and pitfalls of automatic content analysis methods for political texts. *Political Analysis, 21*(3), 267–297.

Hamilton, W. L., Leskovec, J., & Jurafsky, D. (2016). Diachronic word embeddings reveal statistical laws of semantic change. In *Proceedings of the 54th Meeting of the Association for Computational Linguistics* (pp. 1489–1501).

Hartmann, J., Huppertz, J., Schamp, C., & Heitmann, M. (2018). Comparing automated text classification methods. Association for Computational Linguistics. Stroudsburg, PA. *International Journal of Research in Marketing. 36*(1), pp. 20–38.

Hovy, D. (2010). *An evening with… EM* (technical report). University of Southern California. Online tech report.

Hovy, D., & Purschke, C. (2018). Capturing regional variation with distributed place representations and geographic retrofitting. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. Stroudsburg, PA. (pp. 4383–4394).

Humphreys, A., & Wang, R. J.-H. (2017). Automated text analysis for consumer research. *Journal of Consumer Research, 44*(6), 1274–1306.

Jagarlamudi, J., Daumé, H., III, & Udupa, R. (2012). Incorporating lexical priors into topic models. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics. Stroudsburg, PA (pp. 204–213).

Jelinek, F., & Mercer, R. (1980). Interpolated estimation of Markov source parameters from sparse data. In *Proceedings Workshop Pattern Recognition in Practice* (pp. 381–397).

Jurafsky, D. (2014). *The language of food: A linguist reads the menu.*. North Holland Publishing Company, Amsterdam. New York: W. W. Norton.

Jurafsky, D., & Martin, J. H. (2014). *Speech and language processing* (3rd ed.). London: Pearson.

Katz, S. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing, 35*(3), 400–401.

Kulkarni, V., Al-Rfou, R., Perozzi, B., & Skiena, S. (2015). Statistically significant detection of linguistic change. In *Proceedings of the 24th International Conference on the World Wide Web*, Association for Computing Machinery. New York, NY. (pp. 625–635).

Labov, W. (1972). *Sociolinguistic patterns.* Philadelphia, PA: University of Pennsylvania Press.

Landauer, T. K., & Dumais, S. T. (1997). A solution to Plato’s problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review, 104*(2), 211–240.

Lang, S. (2012). *Introduction to linear algebra.* New York: Springer Science & Business Media.

Lau, J. H., & Baldwin, T. (2016). An empirical evaluation of doc2vec with practical insights into document embedding generation. In (p. 78–86). Proceedings of the 1st Workshop on Representation Learning for NLP. Association for Computational Linguistics. Stroudsburg, PA.

Le, Q., & Mikolov, T. (2014). Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. Association for Computing Machinery. New York, NY. (pp. 1188–1196).

Loper, E., & Bird, S. (2002). *NLTK: The Natural Language Toolkit*. Paper presented at the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics.

Maaten, L. v. d., & Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research, 9*, 2579–2605.

Manning, C. D., & Schütze, H. (1999). *Foundations of statistical natural language processing.* Cambridge, MA: MIT Press.

Marsland, S. (2015). *Machine learning: An algorithmic perspective* (2nd ed.). New York: Chapman and Hall/CRC.

McDonald, R., Nivre, J., Quirmbach-Brundage, Y., Goldberg, Y., Das, D., Ganchev, K., et al. (2013). Universal dependency annotation for multilingual parsing. In *Proceedings of the 51st annual meeting of the Association for Computational Linguistics: Vol. 2. Short Papers*. Association for Computational Linguistics. Stroudsburg, PA. (pp. 92–97).

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., & Khudanpur, S. (2010). *Recurrent neural network based language model*. Paper presented at the 11th annual conference of the International Speech Communication Association.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems. Neural Information Processing Systems Foundation. San Diego, CA.* (pp. 3111–3119).

Mimno, D., Wallach, H., Talley, E., Leenders, M., & McCallum, A. (2011). Optimizing semantic coherence in topic models. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. Stroudsburg, PA. (pp. 262–272).

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective.* Cambridge, MA: MIT Press.

Niculae, V., Kumar, S., Boyd-Graber, J., & Danescu-Niculescu-Mizil, C. (2015). Linguistic harbingers of betrayal: A case study on an online strategy game. In *Proceedings of the 53rd annual meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing: Vol. 1. Long Papers*. Association for Computational Linguistics. Stroudsburg, PA. (pp. 1650–1659).

Nivre, J., Agic, Ž., Aranzabe, M. J., Asahara, M., Atutxa, A., Ballesteros, M., et al. (2015). Universal Dependencies Consortium. No address: https://universaldependencies.org/ *Universal dependencies 1.2*.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajič, J., Manning, C. D., et al. (2016, May). Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)* (pp. 1659–1666). Portorož, Slovenia: European Language Resources Association (ELRA). Retrieved from www.aclweb.org/anthology/L16-1262

Pennebaker, J. W. (2011). *The secret life of pronouns: What our words say about us*. New York: Bloomsbury Press.

Pennebaker, J. W., Francis, M. E., & Booth, R. J. (2001). *Linguistic inquiry and word count: LIWC 2001*. Mahwah, NJ: Lawrence Erlbaum, 2001.

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics. Stroudsburg, PA. (pp. 1532–1543).

Petrov, S., Das, D., & McDonald, R. (2011). A universal part-of-speech tagset. In *Proceedings of LREC*. European Language Resources Association. Paris.

Porter, M. F. (1980). An algorithm for suffix stripping. *Program, 14*(3), 130–137.

Prabhakaran, V., Rambow, O., & Diab, M. (2012). Predicting overt display of power in written dialogs. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics. Stroudsburg, PA. (pp. 518–522).

Resnik, P., & Hardisty, E. (2010). *Gibbs sampling for the uninitiated* (technical report). College Park, MD: University of Maryland Institute for Advanced Computer Studies.

Roberts, Molly Roberts, Brandon Stewart, Dustin Tingley, Edoardo Airoldi M. E., Stewart, B. M., Tingley, D., Airoldi, E. M., et al. (2013). The structural topic model and applied social science. In *Advances in neural information processing systems workshop on topic models: Computation, application, and evaluation. Neural Information Processing Systems Foundation. San Diego, CA*. (pp. 1–20).

Röder, M., Both, A., & Hinneburg, A. (2015). Exploring the space of topic coherence measures. In *Proceedings of the 8th ACM International Conference on Web Search and Data Mining*. Association for Computing Machinery. New York, NY. (pp. 399–408).

Rong, X. (2014). *word2vec parameter learning explained*. arXiv preprint arXiv:1411.2738.

Schwartz, H. A., Eichstaedt, J., Blanco, E., Dziurzynski, L., Kern, M., Ramones, S., et al. (2013). Choosing the right words: Characterizing and reducing error of the word count approach. In *Second Joint Conference on Lexical and Computational Semantics (* SEM): Vol. 1. Proceedings of the main conference and the shared task: Semantic textual similarity*. Association for Computational Linguistics. Stroudsburg, PA. (pp. 296–305).

Spärck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation, 28*(1), 11–21.

Srivastava, A., & Sutton, C. (2017). *Autoencoding variational inference for topic models*. arXiv preprint arXiv:1703.01488.

Stevens, K., Kegelmeyer, P., Andrzejewski, D., & Buttler, D. (2012, July). Exploring topic coherence over many models and many topics. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning* (pp. 952–961). Jeju Island, Korea: Association for Computational Linguistics. Retrieved from www.aclweb.org/anthology/D12-1087

Trudgill, P. (2000). *Sociolinguistics: An introduction to language and society*. London: Penguin.

Zipf, G. K. (1935). The psycho-biology of language: An introduction to dynamic philology. Houghton Mifflin. Boston, MA.

Acknowledgments

I would like to thank several people. First, I thank all the participants in my first NLP class at Bocconi for their detailed feedback on the first drafts of this material. I am also grateful to the reviewers for their thoughtful and constructive feedback for improving this work. Thanks also to Tony Bertelli, who put me in touch with Cambridge University Press, and to Mike Alvarez and Neal Beck for being diligent and patient editors, explaining the process and exploring the options for this new endeavor. Many thanks also to Robert Dreesen and Cambridge for taking on this project.