CHAPTER .1

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 and start to take notes. But how do you let a computer do the same?

In order to really work with text 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, we need to get it into a form that computers can understand and manipulate. Typically, this is done via representing document as **feature vectors**. Vectors are essentially list of numbers, but they have a number of 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. We will only cover the basics here. Vectors can either represent a collection of **discrete features** that characterize a word or a document (in which case they are called **discrete representations**), or they can be **distributed representations** with **continuous dimensions**, which represent the word or document holistically, in relation to all other words or documents in our corpus.

feature vectors

linear algebra

discrete features

discrete representations

distributed representations

continuous dimensions

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, and for the most part, these vectors are empty (e.g., a document might not contain the word `tsk!`, and 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 2.

sparse

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, the position of a document with respect to all other documents in the vector space is the important part. 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 representations are called **embeddings** (because the *embed* a word or document in the high-dimensional space). We will see distributed representations when we look at word and document embeddings in Section 3.

dense

embeddings

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.

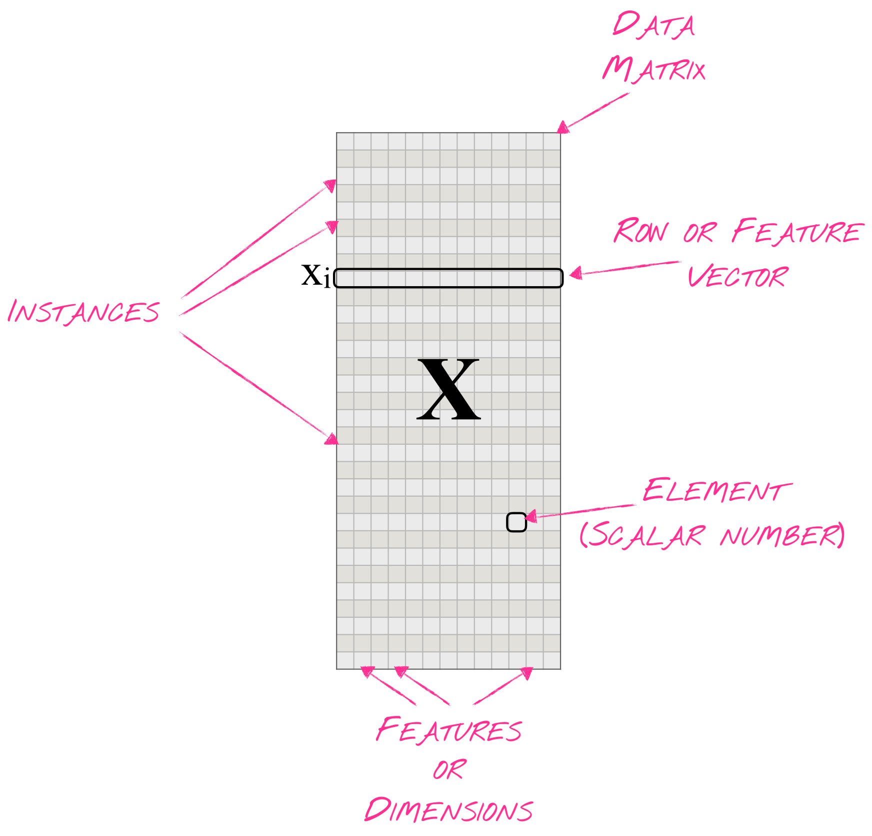

FIGURE 1. Elements of the data matrix.

matrices and vectors

data matrix

row vector

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 above-mentioned features or dimensions in the embedding space. 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. So when we refer to a document, unless specified otherwise, we refer to a **row vector** in the data matrix.

inputs

outputs

scalar

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 to take linear combination of the data with coefficients, to decompose the data, reduce their dimensionality, transform it, etc. It also allows us to combine the data with other kinds of information, for example networks (which 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 these representations from text, and to use them in various algorithms.

2. Discrete Representations

Suppose we have a corpus of documents, and have preprocessed it so that it contains a vocabulary of 1000 words. We want to represent each document as the number of times we have seen each of these 1000 words in a document. How do we collect the counts to make each document a 1000-dimensional vector?

The simplest way of quantifying the importance of a particular word is to count its occurrences in a document. When we want to track the importance of a particular issue (say, "energy") for a company in their quarterly reports, getting the **term frequency** (after lemmatization) gives us a rough impression of its importance over time. In fact, 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 definitely immediately tell whether you have seen the word "dog" more frequently than the word "platypus".

term frequency

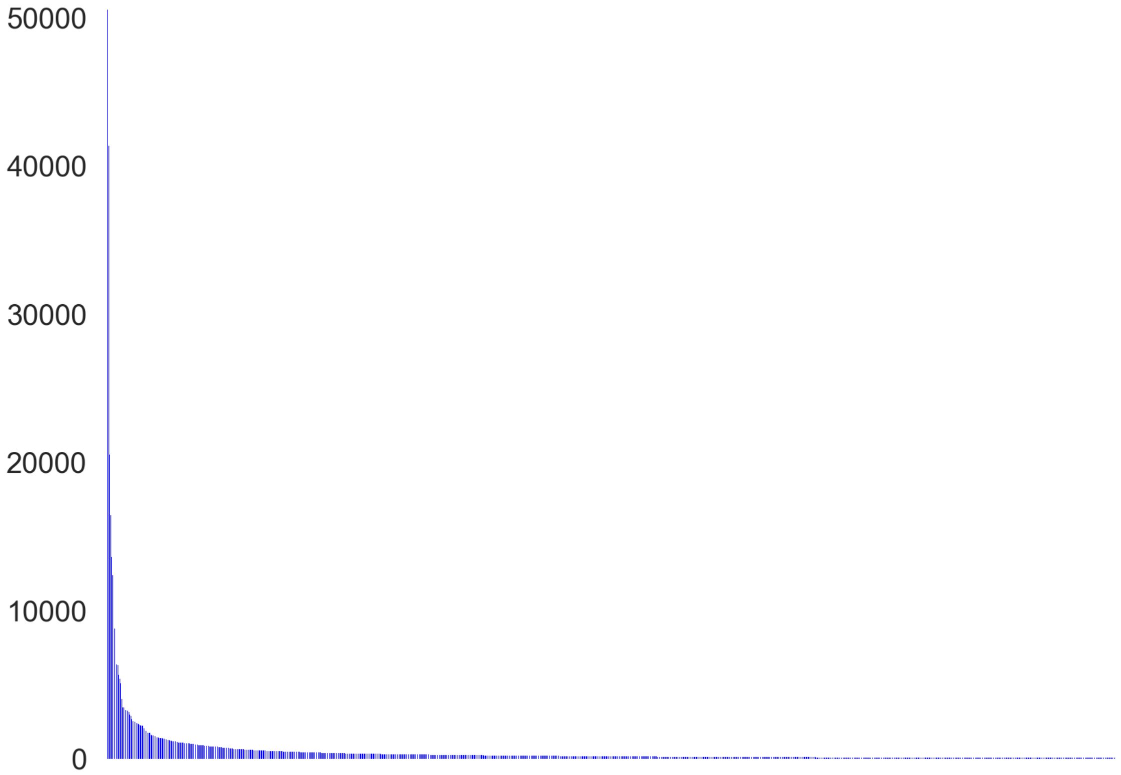

FIGURE 2. Frequency distribution of the top 1000 words in a random sample of tweets, following Zipf's law.

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 a language is typically several times more frequent than the second most frequent word, which in turn is magnitudes more frequent than the third most frequent word, etc., until we reach the "tail" of the distribution, where we will have a long list of words that occur only once (see Figure 2). 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 socio-cultural phenomena. It is therefore important 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.

Zipf's law

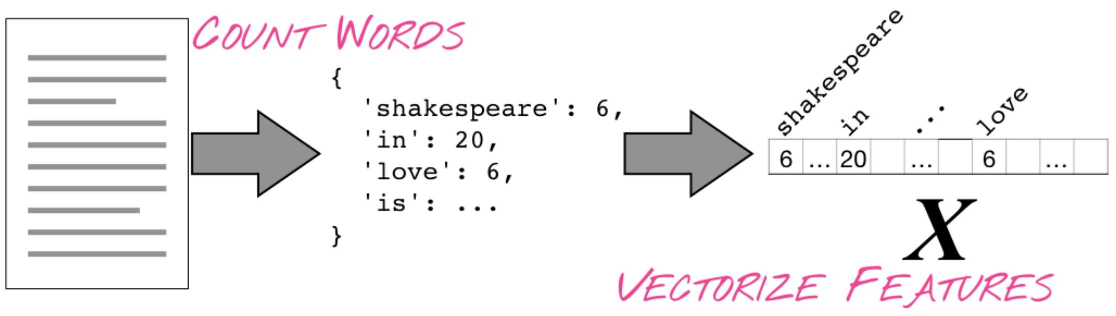

FIGURE 3. Schematic of a bag-of-words representation

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 is to simply create a vector where each dimension represents one word in our vocabulary, so D has the size of all words in our vocabulary V, 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 3). The result of this application to our corpus is a **count matrix**.

count vector

bag of words
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 1. Extracting n-gram count representations.

2.1. n-gram Features.

In addition to individual words, we would also like to account for n-grams of lengths 2 and 3, in order 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 will see later, in chapter **??** how to preprocess such terms so as 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 2. Extracting n-gram count representations.

In the code snippet 3, we create a vectorizer that collects all word unigrams, bigrams, and trigrams that occur in at least 10 documents, but not in more than 75% of all documents, and only if the words are not 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 either look at individual words as documents, or if we are dealing with multiple languages in the same corpus (when we cannot use the appropriate mode for lemmatization), neologisms (new word creations), and misspellings. From a linguistic perspective, this may seem counterintuitive, but from an algorithmic perspective, character n-grams address all of these issues. Many character n-grams are shared across languages (e.g., "en"), and we usually only need a limited number of all possible combinations, because many are never seen 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. N-grams capture overlap and similarities between words better than a full-word approach. This 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: 7 out of the 9 unique trigrams we can extract from them are shared, and so the character count-vectors of the two will be similar – despite the spelling variation. IN a word based approach, these two words would be much less similar.

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 sufficiently many word n-grams that occur frequently enough to allow us to capture most of the variation (the fact English was the first language NLP was developed for might also explain the enduring success of n-grams).

However, as we have seen, word order in many languages is much more free 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**), but 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.

marked

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:

```python
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 3. 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}
```

3. Distributed Representations

There are still many cases where we would like to capture the meaning similarity between words (such as for **synonyms**, e.g., "flat" and "apartment"), or their general relatedness in semantic terms, such as "doctor", "syringe", and "hospital", which all tend to appear in the same context. For example, in order to capture language change, we might be interested in seeing what the nearest words to "mouse" are over time (Kulkarni et al., 2015; Hamilton et al., 2016), or we might want to measure stereotypes by seeing how similar to each other "nurse" is with "man" and "woman" (Garg et al., 2018).

synonyms

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

distributional semantics

"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 (in this case lemmatization) is so important, because it solves this problem. However, it 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 (i.e., words that are similar to each other be close together in that space). First, though, we need to be more specific by what we mean with *similar to each other*.

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 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.

The key question in all of the above is of course what it means to be "similar to each other". In distributed representations, our words are points in a D-dimensional space. Remember that a vector is nothing but a point specified in the vector, and that we can draw a line from the **origin** (0 coordinates) of the space to that point. This property

origin

allows us to measure the distance between two vectors by computing the angle between them. This angle is computed by the **cosine similarity** of two vectors.

cosine similarity

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 take the root is that this gets rid of negative values, so when we sum them 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, whereas a value of 1.0 means 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.

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:

nearest neighbors

```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 and get the top N
    neighbors = [idx for idx in ranks.argsort()[::-1]][:n]
    return neighbors
```

CODE 4. Implementation of nearest neighbors in vector space.

After we have clarified what it means to be similar to each other, The only question left is now: *how do we get all the words into a high-dimensional space and arrange them by cosine similarity*?

3.2. Word Embeddings.

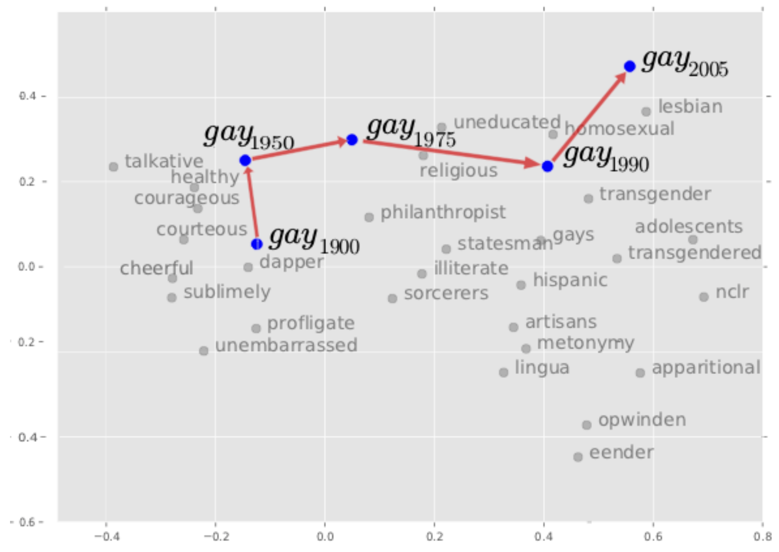

FIGURE 4. Position and change over time of the word "gay" and its nearest neighbors, from Kulkarni et al. (2015)

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, and how has that changed with respect to its present-day meaning (Kulkarni et al., 2015; Hamilton et al., 2016)? What are 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.

For words to be similar to each other, they should occur in similar contexts. I.e., 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, the slots in the sentences "I only like the — gummy bears" and "There was a vase with — flowers" can be filled with "red" or "yellow" (or "blue", or...), indicating that these colors are syntactically similar. Unfortunately, the same is true for **antonyms**: "I had a — weekend" can be filled by a wide range of adjectives, but that does not mean that "boring" and "amazing" mean similar things.

semantically related

semantically similar

antonyms: *words that mean the opposite*

context words

vector space model

Traditionally, word embeddings were collected by defining a number of **context words**, and then counting for each target word in the vocabulary how often it occurred in the same sentence or document as the context words. This approach is known as the **vector space model**. The result was essentially a discrete count matrix. In order to bring out latent similarities, the counts were usually weighted by TF-IDF and then transformed by keeping only the k highest eigenvalues from Singular Value Decomposition. The resulting dense representations are called **Latent Semantic Analysis** or LSA (Deerwester et al., 1990).

Latent Semantic Analysis

Some years ago, though, a new tool, **`word2vec`** (Mikolov et al., 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 against by which we define our target word, this algorithm uses the natural co-occurrence of all words in a corpus (within a certain window to either side of the target word) to compute the vectors. We only specify the number of dimensions we want to have in our embedding space, and press go.

`word2vec`

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 words so that words in similar contexts are close to each other, and words that are dissimilar 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 dimensions seem to work well for most purposes (Landauer and Dumais, 1997; Lau and Baldwin, 2016).

The algorithm iterates over all words w in the vocabulary and carries out a prediction task. It selects a second word c from the context, which wither does (labeled as 1) or does not (labeled as -1) occur within a certain window to either side of w. This process of choosing words that are not from the context is called **negative sampling**. The models computes the dot product of the two vectors and checks the sign of the result. If the prediction is correct, i.e., the sign is positive and v 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 vector for w. We will get into the details of this updating operation in the later half of the book, where we look at prediction and neural networks. If this process is iterated often enough, words that occur within a certain window size of each other get drawn closer and closer together in the vector space.

negative sampling

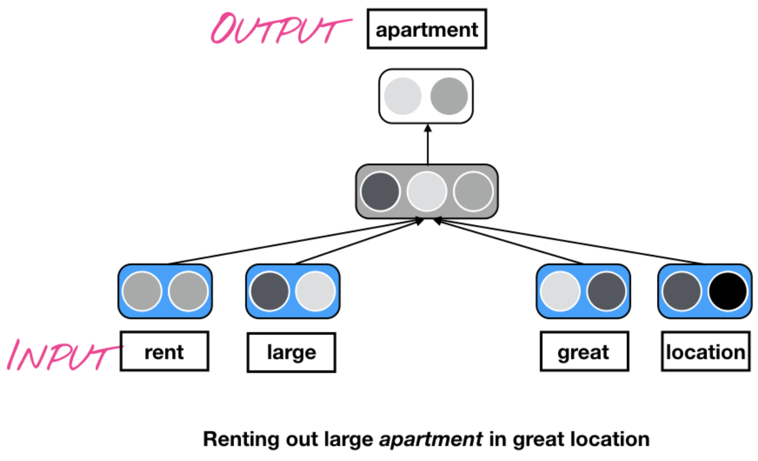

FIGURE 5. `Word2vec` model with CBOW architecture. Context word vectors in blue, target word vector in white.

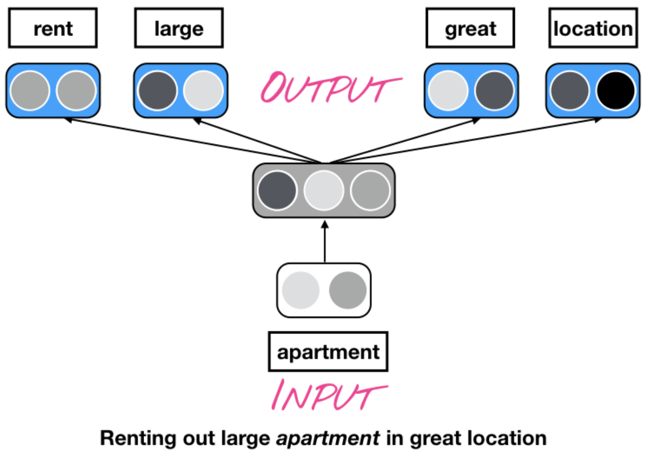

FIGURE 6. `Word2vec` model with skipgram architecture. Context word vectors in blue, target word vector in white.

In practice, we can either predict the target word from the contexts (this is called the **continuous bag of words (CBOW) model**, see Figure 5), or predict the context words from the target word, which is called **skipgram model** (see Figure 6). 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, but discard the context matrix in the end. For more details on this, see the explanation by Goldberg and Levy (2014).

continuous bag of words (CBOW) model
skipgram model

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 only keep content words. We can do this by adding POS. The width of the There are a number of pre-trained word embeddings, which are based on extremely large corpora, and in some cases embeddings exist for several languages in the same embeddings space, making it possible to compare across languages (Grave et al., 2018).

If we are interested in general attitudes, these **pre-trained embeddings** are a great starting point. In other cases, we might want to train embeddings on our corpus ourselves, to capture the similarities 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.

pre-trained embeddings

```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 5. 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, and the **negative sampling rate** and 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

window size
negative sampling rate

minimum number of times a word needs to occur in order 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.

Important: since word embeddings start out randomly and are changed piecemeal, there is no guarantee that we find the optimal solution, i.e., the best possible arrangement of words. 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 and 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 6. Retrieving word vector information in the trained `Word2vec` model.

vector space semantics

The resulting word vectors have a number of nice semantic properties (similarity of meaning), which allow us to do **vector space semantics**. The most famous example of this is the gender analogy of $king\breve{\ }man + woman = queen$. If we add the vectors of the words "king" and "woman" and then subtract the vector of the word "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 relation between countries and their capitals, or the most typical food of countries. These tasks are known as **relational similarity prediction**.

relational similarity prediction

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

CODE 7. Doing vector semantics with the trained `word2vec` model.

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 (dialect in one 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.[1]

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, simply by counting how often each word was present in a document, no matter where they occur. In continuous representations, we can still treat documents as collections of words, by taking the word embeddings we already have, putting them together as a document matrix D, and computing an aggregate over the rows of that matrix as either the mean or the sum over all word embeddings:

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

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`**.

document embeddings
paragraph2vec
`doc2vec`

[1]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.

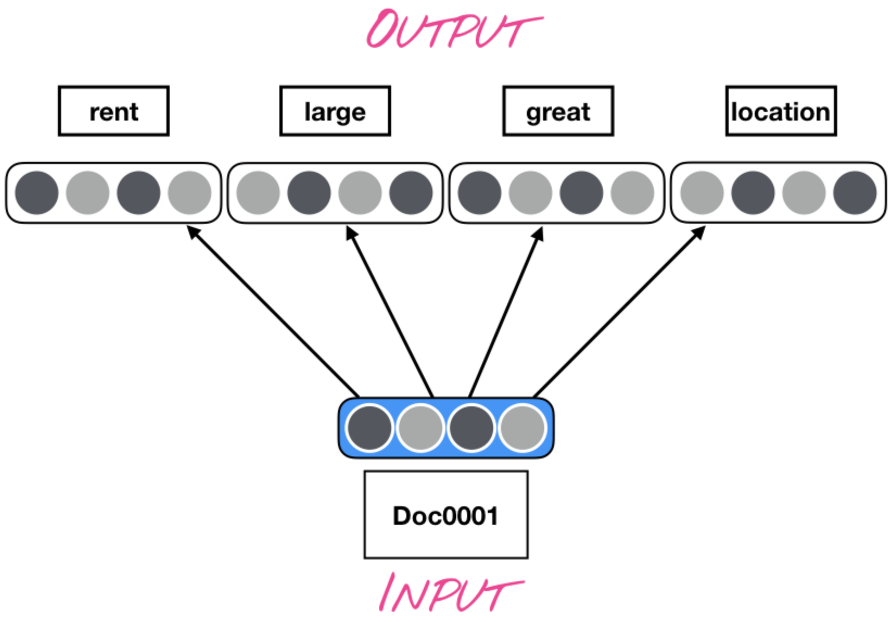

FIGURE 7. `Doc2Vec` model with PV-DBOW architecture

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 Paragraph Vector, or PV-DBOW). 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 8. 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.

```python
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.

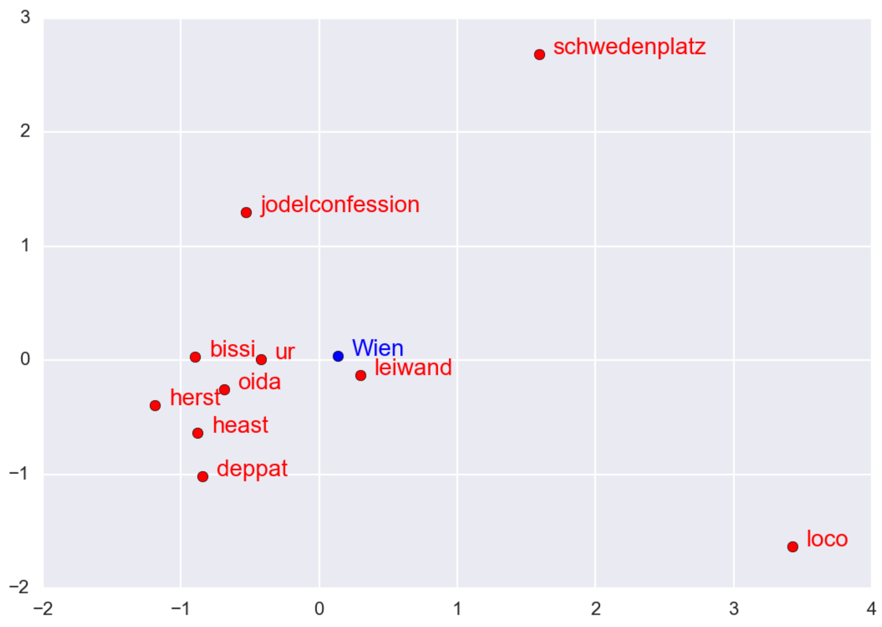

FIGURE 8. Example of documents (cities) and words in the same embedding space from Hovy and Purschke (2018). The words closest to the city vector are dialect terms.

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), but also words to documents (which can tell us what a specific document is about), and of course documents to documents (which can tell us which documents are talking about similar things). For all of these, we use the nearest neighbor algorithm (see Code 4).

Bibliography

Maria Antoniak and David Mimno. 2018. Evaluating the stability of embedding-based word similarities. *Transactions of the Association for Computational Linguistics*, 6:107–119.

Sudeep Bhatia. 2017. Associative judgment and vector space semantics. *Psychological review*, 124(1):1.

Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407.

John R Firth. 1957. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*.

Nikhil Garg, Londa Schiebinger, Dan Jurafsky, and James Zou. 2018. Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences*, 115(16):E3635–E3644.

Yoav Goldberg and Omer Levy. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*.

Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning word vectors for 157 languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*.

William L Hamilton, Jure Leskovec, and Dan Jurafsky. 2016. Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change. In *Proceedings of the 54th Meeting of the Association for Computational Linguistics*, pages 1489–1501. Association for Computational Linguistics.

Dirk Hovy and Christoph Purschke. 2018. Capturing regional variation with distributed place representations and geographic retrofitting. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4383–4394.

Vivek Kulkarni, Rami Al-Rfou, Bryan Perozzi, and Steven Skiena. 2015. Statistically significant detection of linguistic change. In *Proceedings of the 24th International Conference on World Wide Web*, pages 625–635. International World Wide Web Conferences Steering Committee.

Thomas K Landauer and Susan T Dumais. 1997. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, 104(2):211.

Jey Han Lau and Timothy Baldwin. 2016. An empirical evaluation of doc2vec with practical insights into document embedding generation. page 78.

Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

George K Zipf. 1935. The psycho-biology of language: an introduction to dynamic philology.