

P R I F Y S G O L
BANGOR
U N I V E R S I T Y



ICP-4721: Natural Language Processing

Laboratory 4: Understanding and Creating Word

Embeddings

Bill Teahan

EXERCISES

For this module's laboratory sessions, you will need to produce a report that includes the answers to each of the lab exercises and then submit the report to Blackboard. You will have two weeks to complete the report. Marks (out of 100) are shown at the top of each exercise. Try to complete the exercises within the two lab sessions otherwise you will need to complete the exercises in your own time. There will be four labs for this module, so as the lab weighting is 50%, this means that each lab report is worth 12.5%. Feedback on the previous week's lab will be provided the following week so have your lab report and programs ready for checking.

Please do not include the questions in the lab report – just the answers – as this will affect the similarity index for your report for the TurnItIn plagiarism software.

ACADEMIC INTEGRITY

You will be personally asked questions for every lab report you submit. These questions will be individualised based on the answers you have provided in the report. If you fail to answer the questions satisfactorily, this may result in a reduction in your mark, and/or may result in you being referred to a school panel if there are any AI (Academic Integrity) concerns. Ultimately, this may result in you being assigned a 0 mark for the assessment. **To avoid this, please make sure the work you submit is 100% your own! Do not use Generative AI to generate answers to any of the questions below.**

SUBMISSION DEADLINE

All lab work will be due at midnight on Tuesday of the week when it is due (i.e. the night before the next lab session).

DOCUMENTATION

Official Python website – Python.org:

<http://www.python.org/>

For further information and documentation, you can also try the Python Wiki:

<http://wiki.python.org/moin/>

O'Reilly Media – “Learning Python” :

<http://oreilly.com/catalog/9780596513986/>

UNDERSTANDING AND CREATING WORD EMBEDDINGS

Today's lab is based on a tutorial written by Avery Blankenship, Sarah Connell, and Quinn Dombrowski which can be found using the following URL:

<https://programminghistorian.org/en/lessons/understanding-creating-word-embeddings>

Some of the material below has also been taken from a tutorial on word embeddings at:

<https://www.nlplanet.org/course-practical-nlp/01-intro-to-nlp/11-text-as-vectors-embeddings>

Word embeddings allow you to analyze the usage of different terms in a corpus of texts by capturing information about their contextual usage. This lab is designed to get you started with word embedding models. You will learn how to prepare your corpus, read it into your Python session, and train a model. You will explore how word vectors work, how to interpret them, and how to perform some exploratory queries using them. You will be provided with some introductory code to get you started with word vectors, but the main focus will be on equipping you with fundamental knowledge and core concepts to use word embedding models for your own research.

While word embeddings can be implemented in many different ways using varying algorithms, this lesson does not aim to provide an in-depth comparison of word embedding algorithms (though we may at times make reference to them). It will instead focus on the **word2vec** algorithm, which has been used in a range of digital humanities and computational social science projects.¹

This lab uses as its case study a relatively small [corpus of nineteenth-century recipes](#)². This particular case study was chosen to demonstrate some of the potential benefits of a corpus that is tightly constrained, as well as to highlight some of the specific considerations to keep in mind when working with a small corpus.

There are many further potential research applications for trained models. For example, *Programming Historian's* advanced [lesson on word embeddings](#) explains how to cluster and visualize documents using word embedding models. The [Women Writers Project](#) has also published a [series of tutorials in R and Python](#) that cover the basics of running code, training and querying models, validating trained models, and producing exploratory visualizations.

By the end of this lab, you will have learned:

- What word embedding models and word vectors are, and what kinds of questions we can answer with them.
- How to create and interrogate word vectors using Python.
- What to consider when putting together the corpus you want to analyze using word vectors.
- The limitations of word vectors as a methodology for answering common questions.

PREREQUISITES / SYSTEM REQUIREMENTS

To run the code, you can use the lesson's [Jupyter notebook](#) on your own computer. If you prefer to download the notebook alongside a structured environment with folders for sample data and related files, you can also access [this release](#).

The code included in this lab uses [Python 3.8.3](#) and [Gensim 4.2.0](#). [Gensim](#) is an open-source Python library developed by Radim Řehůřek which allows you to represent a corpus as vectors.

The particular word vector implementation used by Gensim is [word2vec](#), which is an algorithm developed in 2013 by Tomáš Mikolov and a team at Google to represent words in vector space, released under an open-source [Apache license](#). While much of the code will still be applicable across versions of both Python and Gensim, there may be some syntax adjustments necessary.

Corpus Size

Word embeddings require a lot of text in order to reasonably represent these relationships — you won't get meaningful output if you use only a couple of novels, or a handful of historical documents. The algorithm learns to predict the contexts in which words might appear based on the corpus it is trained on, so fewer words in the training corpus means less information from which to learn.

That said, there is no absolute minimum number of words required to train a word embedding model. Performance will vary depending on how the model is trained, what kinds of documents you are using, how many unique words appear in the corpus, and a variety of other factors. Although smaller corpora can produce more unstable vectors,³ a smaller corpus may make more sense for the kinds of questions you're interested in. If your purposes are exploratory, even a model trained on a fairly small corpus should still produce interesting results. However, if you find that the model doesn't seem to make sense, that might mean you need to add more texts to your input corpus, or adjust your settings for training the model.

Theory: Introducing Concepts

Word Embeddings

How do popular methods for cooking chicken change over time? How do associations of words like *grace* or *love* vary between prayers and romance novels? Natural language inquiries such as these can prove to be challenging to answer through traditional methods like close reading.

However, by using word embeddings, we can quickly identify relationships between words and begin to answer these types of questions. Word embeddings assign numerical values to words in a text based on their relation to other words. These numerical representations, or ‘word vectors’, allow us to measure the distance between words and gain insight into how they are used in similar ways or contexts. Scaled up to a whole corpus, word embeddings can uncover relationships between words or concepts within an entire time period, genre of writing, or author’s collected works.

Unlike [topic models](#), which rely on word frequency to better understand the general topic of a document, word embeddings are more concerned with how words are used across a whole corpus. This emphasis on relationships and contextual usage make word embeddings uniquely equipped to tackle many questions that humanists may have about a particular corpus of texts. For example, you can ask your word embedding model to identify the list of top ten words that are used in similar contexts as the word *grace*. You can also ask your model to produce that same list, this time removing the concept *holy*. You can even ask your model to show you the words in your corpus most similar to the combined concept of *grace* and *holy*. The ability to perform basic math with concepts (though much more complicated math is happening under the hood) in order to ask really complicated questions about a corpus is one of the key benefits of using word embeddings for textual analysis.

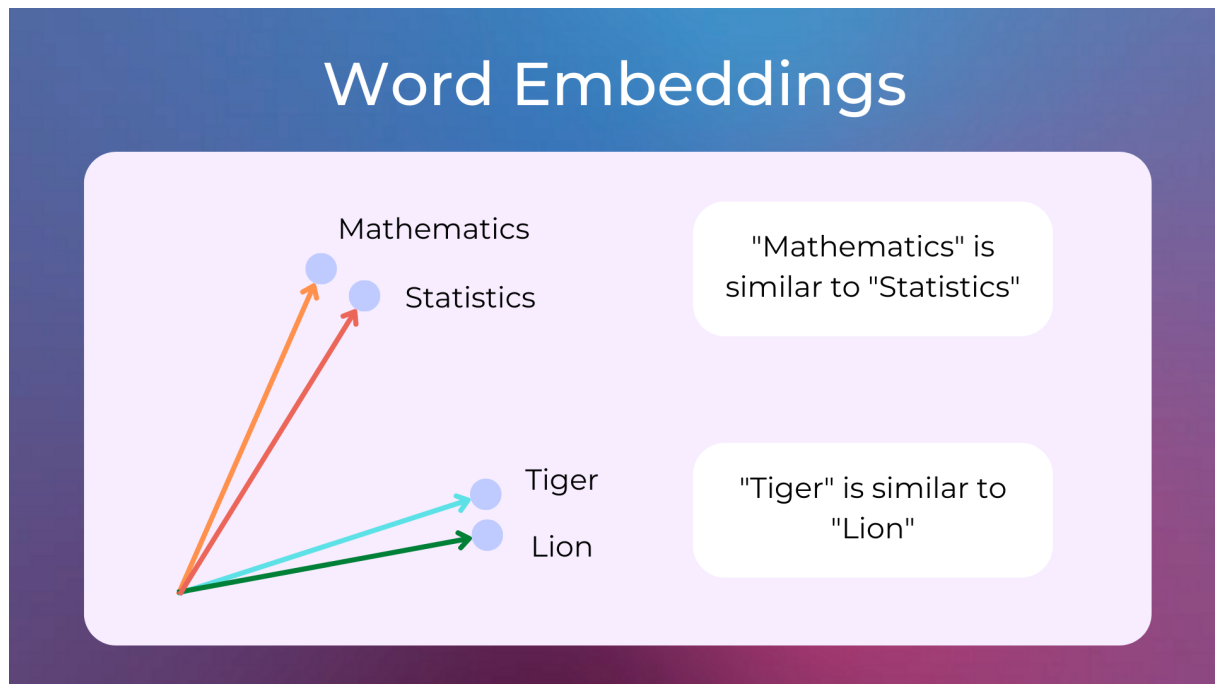
Word Vectors

Word embedding models represent words through a series of numbers referred to as a ‘word vector’. A word vector represents the positioning of a word in multi-dimensional space. Just like we could perform basic math on objects that we’ve mapped onto two-dimensional space (e.g. visualizations with an X and Y axis), we can perform slightly more complicated math on words mapped onto multi-dimensional space.

A ‘vector’ is a point in space that has both ‘magnitude’ (or ‘length’) and ‘direction.’ This means that vectors are less like isolated points, and more like lines that trace a path from an origin point to that vector’s designated position, in what is called a ‘vector space.’

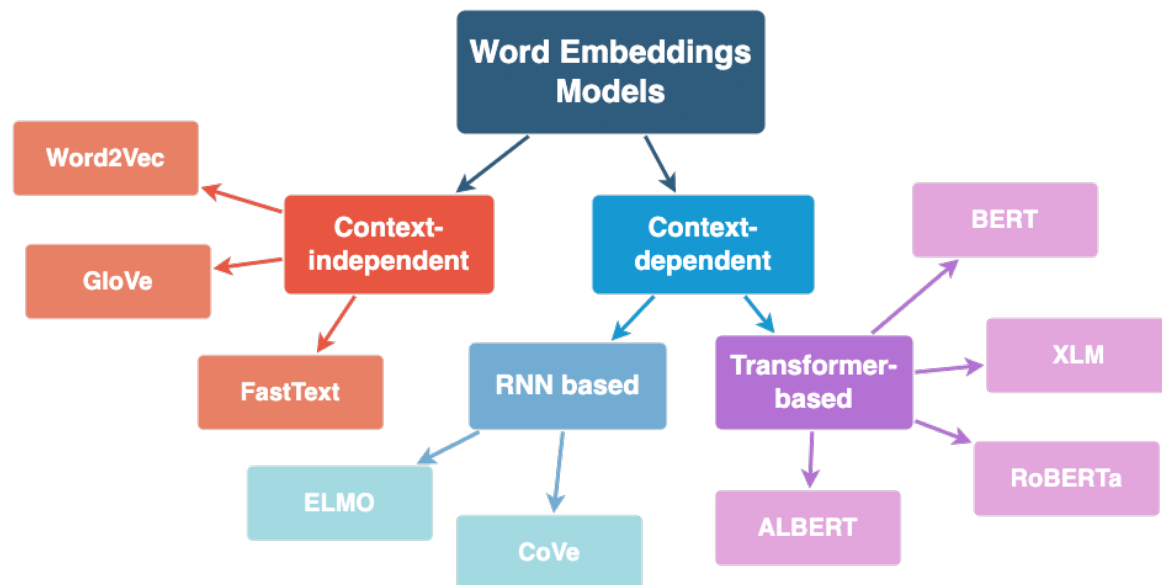
Models created with word vectors, called '**word embedding models**,' use word vectors to capture the relationships between words based on how close words are to one another in the vector space.

Word embeddings are simply a type of word representation (with numerical vectors) that allows words with similar meanings to have a similar representation:



These vectors can be learned by a variety of machine learning algorithms and large datasets of texts. One of the main roles of word embeddings is to provide input features for downstream tasks like text classification and information retrieval.

Several word embedding methods have been proposed in the past decade, here are some of them:



This may sound complicated and abstract, but let's start with a kind of word vector that is more straightforward: a [document-term matrix](#).

Document-Term Matrices

One way of representing a corpus of texts is a **document-term matrix**: a large table in which each row represents a word, and each column represents a text in the corpus. The cells are filled with the count of that particular word in that specific text. If you include every single word in every single text as part of this matrix (including things like names, typos, and obscure words), you'll have a table where most of the cells have a value of 0, because most of the words just don't occur in most of the texts. This setup is called a '**sparse vector representation**.' The matrix also becomes harder to work with as the number of words increases, filling the matrix with more and more 0s. This becomes problematic, because you need a large number of words to have enough data to meaningfully represent language.

The innovation of algorithms like **word2vec** is that they represent relationships between words in a 'dense' way. Different algorithms take different approaches, with consequences on the model's output, but all use a process called 'embedding' to make the vector smaller and much faster. Instead of a vector with tens of thousands of dimensions (including information about the relationship of every unique word with every other unique word), these word embedding models typically use only a few hundred abstracted dimensions, which nonetheless manage to capture the most essential information about relations between different groups of words.

word2vec

word2vec was the first algorithm invented for creating word embedding models, and it remains one of the most popular. It is a predictive model, meaning that it works out the likelihood that either 1) a word will occur in a particular context (using the **Continuous Bag of Words (CBOW)** method), or 2) the likelihood that a particular context will occur for a given word (using the **skip-gram** method).

For this lab, you don't need to worry about the differences between these methods. If you would like to learn more about how word embedding models are trained, there are many useful resources online, such as the [“Illustrated Word2vec”](#) guide by Jay Alammar. The two methods tend to perform equally well, but **skip-gram** often works better with smaller datasets and has better success representing less common words; by contrast, **CBOW** tends to perform better at representing more common words.

For instance, take this set of phrases with the word *milk* in the middle:

- *Pour the milk into the*
- *Add 1c milk slowly while*
- *Set the milk aside to*
- *Bring the milk to a*

word2vec samples a variety of contexts around each word throughout the corpus, but also collects examples of contexts that never occur around each word, known as ‘negative sampling.’ Negative sampling might generate examples like:

- *Colorless green milk sleeps furiously*
- *My horrible milk ate my*
- *The idealistic milk meowed all*

It then takes this data and uses it to train a model that can predict the words that are likely, or unlikely, to appear around the word *milk*. Because the sampling is random, you will likely end up with a small amount of variation in your results if you run **word2vec** on a corpus multiple times.

Tip: If you find that running **word2vec** multiple times gets you a large amount of variation, your corpus may be too small to meaningfully use word vectors.

The model learns a set of ‘weights’ (probabilities) which are constantly adjusted as the network is trained, in order to make the network more accurate in its predictions. At the end of training, the values of those weights become the dimensions of the word vectors which form the embedding model.

word2vec works particularly well for identifying synonyms, or words that could be substituted in a particular context. In this sense, *juice* will probably end up being closer to *milk* than *cow*, because it's more feasible to substitute *juice* than *cow* in a phrase like "Pour the [WORD] into the".

Distance in Vector Space

Vectors have both a direction (where is it going?) and a length (how far does it go in that direction?). Both their direction and length reflect word associations in the corpus. If two vectors are going in the same direction, and have a similar length, that means that they are very close to each other in vector space, and they have a similar set of word associations.

'Cosine similarity' is a common method of measuring 'closeness' between words (for more examples of measuring distance, see [this lesson](#) by *Programming Historian*). When you are comparing two vectors from the same corpus, you are comparing two lines that share an origin point. In order to figure out how similar those words are, all we need to do is to connect their designated position in vector space with an additional line, forming a triangle. The distance between the two vectors can then be calculated using the [cosine](#) of this new line. The larger this number, the closer those two vectors are in vector space. For example, two words that are far from each other (say, *email* and *yeoman*) might have a low cosine similarity of around 0.1, while two words that are near to each other (say *happy* and *joyful* or even *happy* and *sad*) might have a higher cosine similarity of around 0.8.

Words that are close in vector space are those that the model predicts are likely to be used in similar contexts. It is often tempting to think of these as synonyms, but that's not always the case. In fact, antonyms are often very close to each other in word2vec vector space. Words that are likely to be used in the same contexts might have some semantic relationship, but their relationship might also be structural or syntactic. For instance, if you have a collection of letters, you might find that *sincerely*, *yours*, *friend*, and *warmly* are close because they all tend to be used in the salutations of letters. This doesn't mean that *friend* is a synonym of *warmly*! Along the same lines, days of the week and months of the year will often be very close in vector space – *Friday* is not a synonym for *Monday* but the words tend to get used in the same contexts.

When you observe that words are close to each other in your models (high cosine similarity), you should return to your corpus to get a better understanding of how the use of language might be reinforcing this proximity.

Vector Math

Because word vectors represent natural language numerically, it becomes possible to perform mathematical equations with them. Let's say for example that you wanted to ask your corpus the following question: "How do people in nineteenth-century novels use the word *bread* when they aren't referring to food?" Using vector math allows us to present this very specific query to our model.

The equation you might use to ask your corpus that exact question might be: "bread - food = x".

To be even more precise, what if you wanted to ask: "How do people talk about bread in kitchens when they aren't referring to food?" That equation may look like: "bread + kitchen - food = x".

The more complex the math, the larger the corpus you'll likely need to get sensible results. While the concepts discussed thus far might seem pretty abstract, they are easier to understand once you start looking at specific examples. Let's now turn to a specific corpus and start running some code to train and query a **word2vec** model.

Practice: Exploring Nineteenth-Century American Recipes

The **corpus** we are using in this lab is built from nineteenth-century American recipes. Nineteenth-century people thought about food differently than we do today. Before modern technology like the refrigerator and the coal stove became widespread, cooks had to think about preserving ingredients or accommodating to the uneven temperatures of wood-burning stoves. Without the modern conveniences of instant cake mixes, microwaves, air fryers, and electric refrigerators, nineteenth-century kitchens were much less equipped to handle food that could quickly go bad. As a result, many nineteenth-century cookbooks prioritize thriftiness and low-waste methods, though sections dedicated to more fanciful or recreational cooking increase substantially by the turn of the century. Attitudes towards food were much more conservative and practical in the nineteenth century, compared to our forms of 'stress-baking,' or cooking for fun.

Word embedding models allow us to pursue questions such as: "What does American cooking look like if you remove the more fanciful dishes like 'cake,' or low-waste techniques like 'preserves'?" Our research question for this lab is: "How does the language of our corpus reflect attitudes towards recreational cooking and the limited lifespan of perishable ingredients (such as milk) in the nineteenth century?" Since we know milk was difficult to preserve, we can check what words are used in similar contexts to *milk* to see if that reveals potential substitutions. Using vector space model queries, we will be able to retrieve a list of words which do not include *milk* but do share its contexts, thus pointing us to possible substitutions. Similarly, by finding words which share contexts with dishes like *cake*, we can

see how authors talk about cake and find dishes that are talked about in a similar way. This will reveal the general attitude within our corpus towards more recreational dishes. Pursuing these questions will allow us to unpack some of the complicated shifts in language and instruction that occurred in nineteenth-century kitchens.

Retrieving the Code and Data

The first step in building the word embedding model is to identify the files you will be using as your corpus. We will be working with a corpus of 1,000 recipes sourced by Avery Blankenship from cookbooks on [Project Gutenberg](#).² The file names for each recipe are based on the Project Gutenberg ID for the cookbook from which the recipe is pulled, as well as an abbreviated title.

You can download this lesson's [Jupyter notebook](#) and the [corpus](#) to train a model on your own computer. (These files can also be found in Blackboard under Assessments / Lab04.)

We will start by importing all the Python libraries needed for this lab. It is a good practice to keep your import statements together at the top of your code. The code block below first imports each Python library we will be using.

```
import re                # for regular expressions
import os                # to look up operating system-based info
import string            # to do fancy things with strings
import glob              # to locate a specific file type
from pathlib import Path  # to access files in other directories
import gensim            # to access Word2Vec
from gensim.models import Word2Vec  # to access Gensim's flavor of Word2Vec
import pandas as pd      # to sort and organize data
```

You need to then edit the variable `dirpath` for the pathname to the folder where you have uploaded the corpus files.

```
dirpath = r'FILL IN YOUR FILEPATH HERE' # get file path (you can change this)
file_type = ".txt" # if your data is not in a plain text format, you can change this
filenames = []
data = []
```

The code then iterates through your defined directory to identify the text files that make up your corpus. It then reads these text files one by one, and appends them to the list variable `data`.

```
# this for loop will run through folders and subfolders looking for a specific file type
for root, dirs, files in os.walk(dirpath, topdown=False):
    # look through all the files in the given directory
    for name in files:
        if (root + os.sep + name).endswith(file_type):
            filenames.append(os.path.join(root, name))
    # look through all the directories
    for name in dirs:
        if (root + os.sep + name).endswith(file_type):
            filenames.append(os.path.join(root, name))

# this for loop then goes through the list of files, reads them, and then adds the text to a list
for filename in filenames:
    with open(filename) as afile:
        print(filename)
        data.append(afile.read()) # read the file and then add it to the list
        afile.close() # close the file when you're done
```

EXERCISE 4.1 – LOADING THE COOKBOOKS CORPUS (10 MARKS)

Test out the code above to load the Cookbooks Corpus into the `data` variable. **Print out the first file text** that gets loaded by the `read()` method to see what it looks like. Also print out **three further random file texts**. Make sure these texts are truly random so that your output is unlikely to match any other student's submission. **Discuss anything interesting** about the text that is output. Include all the code and output produced into your report.

Building your Model's Vocabulary

Using textual data to train a model builds what is called a '**vocabulary**.' The vocabulary is all of the words that the model has processed during training. This means that the model knows only the words it has been shown. If your data includes misspellings or inconsistencies in capitalization, the model won't understand that these are mistakes. Think of the model as having complete trust in you: the model will believe any misspelled words to be correct. Errors will make asking the model questions about its vocabulary difficult: the model has less data about how each spelling is used, and any query you make will only account for the unique spelling used in that query.

It might seem, then, that regularizing the text's misspellings is always helpful, but that's not necessarily the case. Decisions about regularization should take into account how spelling variations operate in the corpus, and should consider how original spellings and word usages could affect the model's interpretations of the corpus. For example, a collection might contain deliberate archaisms used for poetic voice, which would be flattened in the regularized text. In fact, some researchers advocate for more embracing of textual noise, and a project interested in spelling variations over time would certainly want to keep those!⁴

Nevertheless, regularization is worth considering, particularly for research projects exploring language usage over time: it might not be important whether the spelling is *queen*, *quean*,

or *queene* for a project studying discourse around queenship within a broad chronological frame. As with many aspects of working with word embeddings, the right approach is whatever best matches your corpus and research goals.

Regardless of your approach, it is generally useful to lowercase all of the words in the corpus and remove most punctuation. You can also make decisions about how to handle contractions (*can't*) and commonly occurring word-pairings (*olive oil*), which can be tokenized to be treated as either one or two objects.

Different tokenization modules will have different options for handling contractions, so you can choose a module that allows you to preprocess your texts to best match your corpus and research needs. We've already covered several of these in past lectures and labs. For more on tokenizing text with Python, see this *Programming Historian* [lesson](#).

Cleaning the Corpus

The code below is a reasonable general-purpose starting point for 'cleaning' English-language texts. The function `clean_text()` uses regular expressions to standardize the format of the text (lower-casing, for example) and any remove punctuation that may get in the way of the model's textual understanding. By default, this code will remove the punctuation `!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~` using Python 3's `string.punctuation`. If you wanted to retain certain punctuation marks, however, you could replace `string.punctuation` in the code with your customized string of punctuation.

This process helps the model understand, for example, that *apple* and *Apple* are the same word. It also removes numbers from our textual data, since we are only interested in words. We end the function by checking that our cleaned data hasn't lost any words that we actually need, by making sure that the set of un-cleaned text data is the same length as the cleaned version.

```
def clean_text(text):  
  
    # Cleans the given text using regular expressions to split and lower-cased versions to create  
    # a list of tokens for each text.  
    # The function accepts a list of texts and returns a list of lists of tokens  
  
    # lower case  
    tokens = text.split()  
    tokens = [t.lower() for t in tokens]  
  
    # remove punctuation using regular expressions  
    # this line of code locates the punctuation within the given text and compiles that punctuation  
    # into a single variable  
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))  
    # this line of code substitutes the punctuation we just compiled with nothing ''  
    tokens = [re_punc.sub('', token) for token in tokens]  
  
    # only include tokens that aren't numbers  
    tokens = [token for token in tokens if token.isalpha()]  
    return tokens  
  
# clean text from folder of text files, stored in the data variable  
data_clean = []  
for x in data:  
    data_clean.append(clean_text(x))
```

After this, the list variable `data_clean` will have the cleaned text stored in it (and `data` will still have the raw uncleaned text).

While testing our program, we should always use repeated testing to see if the program is working correctly throughout. The following code does this by comparing parts of the two variables `data` and `data_clean`:

```
# Check that the length of data and the length of data_clean are the same. Both numbers  
# printed should be the same.  
  
print(len(data))  
print(len(data_clean))  
  
# Check that the first item in data and the first item in data_clean are the same.  
# Both print statements should print the same word, with the data cleaning function  
# applied in the second one.  
  
print(data[0].split()[0])  
print(data_clean[0][0])  
  
# Check that the last item in data_clean and the last item in data are the same.  
# both print statements should print the same word, with the data cleaning function  
# applied in the second one  
  
print(data[0].split()[-1])  
print(data_clean[0][-1])
```

EXERCISE 4.2 – NO HARM IN TESTING (10 MARKS)

Execute the above code and **record the output** in your report. **Add further tests of your**

own devising to gain a better understanding of the difference between the two data variables. **Explain** what you have found out.

Creating your Model

To train a **word2vec** model, the code first extracts the corpus vocabulary and generates from it a random set of initial word vectors. Then, it improves their predictive power by changing their weights, based on sampling contexts (where the word exists) and negative contexts (where the word doesn't exist).

Parameters

In addition to the corpus selection and cleaning described above, at certain points in the process you can decide adjust what are known as configuration parameters. These are almost as essential as the texts you select for your corpus. You can think of the training process (where we take a corpus and create a model from it) as being sort of like an industrial operation:

- You take raw materials and feed them into a big machine which outputs a product on the other end.
- This hypothetical machine has a whole bunch of knobs and levers on it that you use to control the settings (the parameters).
- Depending on how you adjust the parameters, you get back differing products (differently trained models).

These parameters will have significant impact on the models you produce. They control things such as which specific algorithm to use in training, how to handle rare words in your corpus, and how many times the algorithm should pass through your corpus as it learns.

There is no 'one size fits all' configuration approach. The most effective parameters will depend on the length of your texts, the variety of the vocabulary within those texts, their languages and structures — and, of course, on the kinds of questions you want to investigate. Part of working with word vector models is turning the knobs on that metaphorical industrial machine, to test how different parameters impact your results. It is usually best to vary parameters one at a time, so you can observe how each one impacts the resulting model.

A particular challenge of working with word vectors is just how much the parameters impact your results. If you are sharing your research, you will need to be able to explain how you chose the parameters that you used. This is why testing different parameters and looking at

multiple models is so important. Below are some parameters which may be of particular interest:

Sentences:

The sentences parameter is where you tell **word2vec** what data to train the model with. In our case, we are going to set this attribute to our cleaned textual data.

Mincount (minimum count):

Mincount is how many times a word has to appear in the dictionary in order for it to count as a word in the model. The default value for **mincount** is 5. You may want to change this value depending on the size of your corpus, but in most cases, 5 is a reasonable minimum. Words that occur less frequently than that don't have enough data to get you sensible results.

Window:

This lets you set the size of the window that is sliding along the text when the model is trained. The default is 5, which means that the window will look at five words at a time: two words before the target word, the target word, and then two words after the target word. Both the words before and after the target words will form the context of the target word. The larger the window, the more words you are including in that calculation of context. As long as they are within the window, however, all words are treated indiscriminately in terms of how relevant they are to the calculated context.

Workers (optional):

The workers parameter represents how many 'threads' you want processing your text at a time. The default setting for this parameter is 3. Increasing this parameter means that your model will train faster, but will also take up more of your computer's processing power. If you are concerned about strain on your computer, leave this parameter at the default.

Epochs (optional):

The number of epochs signifies how many iterations over the text it will take to train the model. There is no rule for what number works best. Generally, the more epochs you have the better, but too many could actually decrease the quality of the model, due to 'overfitting' (your model learns the training data so well that it performs worse on any other data set). To determine what number of epochs will work best for your data, you may wish to experiment with a few settings (for example, 5, 10, 50, and 100).

Sg ('skip-gram'):

The **sg** parameter tells the computer what training algorithm to use. The options are **CBOW** (Continuous Bag of Words) or **skip-gram**. In order to select **CBOW**, you set **sg** to the value 0

and, in order to select **skip-gram**, you set the **sg** value to 1. The best choice of training algorithm really depends on what your data looks like.

Vector_size (optional):

The **vector_size** parameter controls the dimensionality of the trained model, with a default value of 100 dimensions. Higher numbers of dimensions can make your model more precise, but will also increase both training time and the possibility of random errors.

Because **word2vec** samples the data before training, you won't end up with the same result every time. It may be worthwhile to run a **word2vec** model a few times to make sure you don't get dramatically different results for the things you're interested in. If you're looking to make a fine point about shifts in language meaning or usage, you need to take special care to minimize random variation (for instance, by keeping random seeds the same and using the same **skip-gram** window).

The code below will actually train the model, using some of the parameters discussed above:

```
"""## Model Creation"""  
  
# train the model  
model = Word2Vec(sentences=data_clean, window=5, min_count=3, workers=4, epochs=5, sg=1)  
  
# save the model  
model.save("word2vec.model")
```

Interrogating the Model with Exploratory Queries

It's important to begin by checking that the word we want to examine is actually part of our model's vocabulary.

```
### Exploratory Queries  
"""  
  
# set the word that we are checking for  
word = "milk"  
  
# if that word is in our vocabulary  
if word in model.wv.key_to_index:  
    # print a statement to let us know  
    print("The word %s is in your model vocabulary" % word)  
  
# otherwise, let us know that it isn't  
else:  
    print("%s is not in your model vocabulary" % word)
```

Now, we can use **word2vec**'s powerful built-in functions to ask the model how it understands the provided text. Let's walk through each of these function calls below.

One important thing to remember is that the results you get from each of these function calls do not reflect words that have similar definitions, but rather words that are used in the same contexts. While some of the words you'll get in your results are likely to have similar meanings, your model may output a few strange or confusing words. This does not necessarily indicate that something is wrong with your model or corpus, but may reflect instead that these very different words are used in similar ways in your corpus. It always helps to go back to your corpus and get a better sense of how the language is actually used in your texts.

most_similar

This function allows you to retrieve words that are similar to your chosen word. In this case, we are asking for the top ten words in our corpus that are closest to the word *milk*. If you want a longer list, change the number assigned for **topn** to the desired number. The code below will return the ten words with the highest cosine similarities to the word *milk* (or whatever other query term you supply). The higher the cosine similarity, the 'closer' the words are to your query term in vector space (remember that closeness in vector space means that words are used in the same kinds of contexts).

```
# returns a list with the top ten words used in similar contexts to the word "milk"
model.wv.most_similar('milk', topn=10)
```

You can also provide the **most_similar** function with more specific information about your word(s) of interest. In the code block below, you'll notice that one word (*recipe*) is tied to the positive parameter and the other (*milk*) is associated with negative. This call to **most_similar** will return a list of words that are most contextually similar to *recipe*, but not the word *milk*.

```
# returns the top ten most similar words to "recipe" that are dissimilar from "milk"
model.wv.most_similar(positive = ["recipe"], negative=["milk"], topn=10)
```

You can also include more than one word in the positive parameter, as shown below:

```
# returns the top ten most similar words to both "recipe" and "milk"
model.wv.most_similar(positive = ["recipe", "milk"], topn=10)
```

Similarity

This function will return a cosine similarity score for the two words you provide it. The higher the cosine similarity, the more similar those words are.

```
# returns a cosine similarity score for the two words you provide
model.wv.similarity("milk", "cream")
```

predict_output_word

This function will predict the word most likely to appear next from a set of context words, depending on the choice of words provided. This function works by inferring the vector of an unseen word.

```
# returns a prediction for the other words in a set containing the words "flour," "eggs," and "cream"
model.predict_output_word([ "flour", "eggs", "cream"])
```

You can also easily print the number of words in your model's vocabulary:

```
# displays the number of words in your model's vocabulary
print(len(model.wv))
```

EXERCISE 4.3 – NO HARM IN MORE TESTING (5 MARKS)

Further test out how the above code works and what output is produced by changing the words and other parameters passed to the different methods. Include your testing code and output in your report.

EXERCISE 4.4 – NO HARM IN EVEN MORE TESTING (15 MARKS)

Find out if there are other parameters and methods you can play around with. (Hint: Use Python's in-built help facility – **include the output from this in your report**). **Try out several of these variations.** Again, include your further testing code and output in your report.

Validating the Model

Now that we have explored some of its functionalities, it is important to evaluate our working model. Does it respond to our queries the way we would expect? Is it making obvious mistakes?

Validation of word vector models is currently an unsolved problem. The test below provides a sample of one approach to testing a model, which involves seeing how well the model performs with word pairs that are likely to have high cosine similarities. This approach is just an example, and not a substitute for more rigorous testing processes. The word pairs will be very specific to the corpus being tested, and you would want to use many more pairs than are in this demonstration sample!

The code below evaluates the model first by opening the folder of models you provide, and identifying all files that are of type `.model`. Then, the code takes a list of test word pairs and calculates their cosine similarities. The word pairs are words that, as a human, you would expect to have high cosine similarity. Then, the code saves the cosine similarities for each word pair in each model in a `.csv` file for future reference.

If you were interested in adapting this code for your own models, you would want to select word pairs that make sense for the vocabulary of your corpus (for example, we've chosen recipe-related words we can reasonably expect to have high cosine similarities). Choosing to evaluate this particular model using words from a completely different field (e.g. looking for *boat* and *ship*) would clearly not be effective, because those terms would not appear in this corpus at all, let alone in high enough frequencies. Selecting your word pairs is a matter of judgement and knowing your own corpus.

If you test out several pairs of similar words, and you find that their vectors are not close to each other in the model, you should check your corpus using your own eyes. Search for the words: how many times do each of the words appear in the corpus? Are they appearing in similar contexts? Is one appearing in many more contexts (or many more times) than the other? The example corpus for this lab works, despite being quite small, because the texts belong to the fairly tightly-scoped domain of recipes from the same century. Nineteenth-century recipes from cookbooks are very different from twenty-first century recipes from the internet, so while using texts of both types in the same corpus would greatly expand the total vocabulary, you would also need to expand the corpus size in order to get enough examples of all the relevant words. If you test word pairs that should be similar and the cosine distance between them is high, you may need a larger corpus.

In this example, we find a set of models (filename ends in `.model`) in a specified directory, add them to a list, and then evaluate cosine distance for a set of test word pairs.

```
dirpath = Path(r".").glob('*.*model') #current directory plus only files that end in 'model'
files = dirpath
model_list = [] # a list to hold the actual models
model_filenames = [] # the filepath for the models so we know where they came from

#this for loop looks for files that end with ".model" loads them, and then adds those to a list
for filename in files:
    # turn the filename into a string and save it to "file_path"
    file_path = str(filename)
    print(file_path)
    # load the model with the file_path
    model = Word2Vec.load(file_path)
    # add the model to our model_list
    model_list.append(model)
    # add the filepath to the model_filenames list
    model_filenames.append(file_path)
```

The specific test word pairs being used to evaluate the models is defined in the code as follows:

```
#test word pairs that we are going to use to evaluate the models
test_words = [("stir", "whisk"),
              ("cream", "milk"),
              ("cake", "muffin"),
              ("jam", "jelly"),
              ("reserve", "save"),
              ("bake", "cook")]
```

We then add all the results to a Pandas **dataframe**. This is then printed out and also saved to a .CSV file which can then be examined separately:

```
# iterate though the model_list
for i in range(len(model_list)):

    # for each model in model_list, test the tuple pairs
    for x in range(len(test_words)):

        # calculate the similarity score for each tuple
        similarity_score = model_list[i].wv.similarity(*test_words[x])

        # create a temporary dataframe with the test results
        df = [model_filenames[i], test_words[x], similarity_score]

        # add the temporary dataframe to our final dataframe
        evaluation_results.loc[x] = df

print (evaluation_results)

# save the evaluation_results dataframe as a .csv called "word2vec_model_evaluation.csv" in our current directory
# if you want the .csv saved somewhere specific, include the filepath in the .to_csv() call
evaluation_results.to_csv('word2vec_model_evaluation.csv')
```

EXERCISE 4.5 – EVALUATING THE MODEL (10 MARKS)

Add at least four further word pairs to the set of test words. Check out the output that is produced when running the above code and include it in your report. **Discuss** what the output shows.

There are other methods for conducting a model evaluation. For example, a popular method for evaluating a **word2vec** model is using the built in **evaluate_word_analogies()** function to evaluate syntactic analogies. You can also evaluate word pairs using the built-in function **evaluate_word_pairs()** which comes with a default dataset of word pairs. You can read more about evaluating a model on Gensim's [documentation](#).

Application: Building a Corpus for your own Research

Now that you've had a chance to explore training and querying a model using a sample corpus, you might consider applying word embeddings to your own research.

Important Preliminary Considerations

When determining whether word vectors could be useful for your research, it's important to consider whether the kinds of questions you are trying to investigate can be answered by analyzing word usage patterns across a large corpus. This means considering the following issues:

- Is it possible to assemble a corpus that gives enough insight into the phenomenon you would like to investigate? For example, if you are studying how early modern British historians distinguished their work from that of their medieval predecessors, you might assemble two corpora: one of medieval histories, and another of early modern accounts.
- Are 'relationships between words' a useful heuristic for your research? Can you identify terms or groups of terms which represent the larger conceptual spaces you are studying? With our early modern history example, we might decide to see how closely words like *fiction*, *lie* or *falsehood* (suggesting untruthful accounts of the past) are associated to earlier histories (through terms such as *monk*, *medieval* or *chronicler*).

Another important consideration for building your corpus is the composition of the texts. You should think about questions like:

- Are the texts in your corpus in a single language, or more than one? If multiple languages, what is their distribution? Keep in mind that if you have multiple languages, there's no magical translation layer in the word vector creation: the information about the contexts of *gato* (in Spanish) won't merge with the contexts of *cat* (in English). Mixing multiple languages in a corpus might get you meaningful and interesting results if you're studying, for instance, novels from the US borderlands where code-switching between languages can be a significant literary device. However, throwing Spanish-only documents and English-only documents together in a single corpus just gives you two sets of words that at best don't co-occur with each other, and at worst give misleading results. For example, a model can't differentiate between *con* (with) as a conjunction in Spanish and *con* (convict) as a noun in English. If your research question involves analyzing English words related to crime, the vector for English *con* will be skewed by the identically-spelled Spanish word.
- Do your texts vary in other features, such as length, genre, or form? Be deliberate about what you're including in your corpus and why: if you want to work on the language of eighteenth-century poetry, and find that all your poems together don't have enough of a word count to get decent results, don't start adding eighteenth-century novels without adjusting your research questions accordingly. When big tech

companies create giant word embeddings to help with things like machine translation, they're working with unimaginably large corpora, at a scale where factors like genre and form have little impact. However, the smaller your data, the more careful you need to be — especially when trying to make scholarly claims about the output.

- What principles will you use to scope your corpus—date, publisher, publication location, author? You should make sure that your selection principles match with the kinds of questions you want to investigate. This applies even when you may be tempted to cast a wide net to get enough text.
- If you are not selecting all possible texts within your scope parameters — which you probably will not — how will you ensure that the texts you do select are broadly representative of the full set of potential texts? How can you at least make sure that there are no serious imbalances between the texts you include? Returning to our early modern history example, it would be problematic if a corpus of seventeenth-century histories consisted of 59 texts published in 1699 and one published in 1601.

Overall, you should aim toward a balance in the salient features of the texts that you select (publication date, genre, publication location, language) and a close alignment between the texts in your corpus and your research objective. If you are comparing corpora, make sure that the only difference between them is the feature that you are investigating. Remember that word vectors are going to give you information about the relationships between words — so the actual words that go into your corpora are crucial!

Preparing the Texts in your Corpus

When you are preparing your corpus, bear in mind that the model is trained on all the words in your corpus. Because the results depend so heavily on the input data, you should always include a data analysis phase early in your research to ensure you only include the words you really want. In fact, data preparation and analysis should be an iterative process: review the texts, identify where the data needs to be adjusted, make those changes, review the results, identify additional changes, and so on. This makes it important to keep track of all the changes you make to your texts.

If you're sourcing texts from Project Gutenberg, you will want to remove the project's own boilerplate description at the beginning and end of the texts. Consider removing as well: editorially-authored text (such as annotations or descriptions of images), page numbers, and labels. Removing these features is preferable because they are not likely to be of interest and they could skew the distances between related terms.

The goals of the project will impact which document features are best kept or removed. These include paratexts — such as indices, tables of contents, and advertisements — and

features like stage directions. Finally, you may choose to manipulate the language of your documents directly, such as by regularizing or modernizing the spelling, correcting errors, or lemmatizing text. Note that if you make changes to the language of your documents, you will also want to maintain an unmodified corpus, so that you can investigate the impacts of your data manipulations.

Once you have identified a corpus and prepared your texts, you can adapt the code above to train, query, and validate your model. Remember: this is an iterative process! You will likely need to make additional changes to your data and your parameters as you better understand what your model reveals about the texts in your corpus. The more you experiment with your data and your models, the better you will understand how these methods can help you answer new kinds of questions, and the better prepared you will be to learn even more advanced applications of word vector models!

Next Steps

Now that you've learned how to build and analyze word embeddings, you can see *Programming Historian's* related [Clustering and Visualizing Documents using Word Embeddings](#) lesson to learn more advanced methods of analysis with word vectors.

Here are some other resources if you would like to learn more about word vectors:

- The Women Writers Project provides a full set of tutorials for training word vector models in Python, which can be downloaded with sample data from the WWP's [Public Code Share GitHub repository](#).
- The [Women Writers Vector Toolkit](#) is a web interface for exploring word vectors, accompanied by glossaries, sources, case studies, and sample assignments. This toolkit includes links to a [GitHub repository with RMD walkthroughs](#) with code for training word2vec models in R, as well as [downloadable resources on preparing text corpora](#).
- The [Women Writers Project Resources page](#) has guides on searching your corpus; corpus analysis and preparation; model validation and assessment, and more.

To get a better sense of how word vectors might be used in research and the classroom, you can see this [series of blog posts](#); see also this [annotated list of readings](#).

Below are individual readings and research projects that help to illuminate the applications of word vector models in humanistic research:

- Ryan Heuser's [Word Vectors in the Eighteenth Century](#) walks through a research project using word vector models to understand eighteenth-century conceptions of originality.
- Michael Gavin, Collin Jennings, Lauren Kersey, and Brad Pasanek. [Spaces of Meaning: Vector Semantics, Conceptual History, and Close Reading](#) explores how to use word embedding models to study concepts and their history.
- Laura Nelson, [Leveraging the alignment between machine learning and intersectionality: Using word embeddings to measure intersectional experiences of the nineteenth century U.S. South](#) considers ways of using word embedding models in deliberately intersectional research applications, working through an example project on racialized and gendered identities in the nineteenth-century U.S. South.
- Anouk Lang, [Spatial Dialectics: Pursuing Geospatial Imaginaries with Word Embedding Models and Mapping](#) uses word embedding models to explore 'spatial imaginaries' in early twentieth-century Canadian periodicals, looking at gendered discourse and also at ways to combine results word embedding models with geospatial analysis.
- Siobhan Grayson, Maria Mulvany, Karen Wade, Gerardine Meaney, and Derek Greene, [Novel2Vec: Characterising 19th Century Fiction via Word Embeddings](#) explores the use of word embedding models to study narrative formations in 19th-century novels.
- Benjamin Schmidt's [Vector Space Models for the Digital Humanities](#) outlines some foundational concepts of word embeddings, walks through a sample project with periodical texts, and discusses the kinds of humanities research questions for which this method is most relevant.

Characterizing 19th Century Fiction via Word Embeddings

The following is a link to a paper that reports results on a study that used Word Embeddings to characterize 19th Century Fiction – specifically, novels written by Jane Austen, Charles Dickens and Arthur Conan Doyle.

https://ceur-ws.org/Vol-1751/AICS_2016_paper_48.pdf

Reference: Grayson, S., Mulvany, M., Wade, K., Meaney, G. and Greene, D., 2016. *Novel2vec: Characterising 19th century fiction via word embeddings*.

(The paper has also been saved to Blackboard under Assessments / Lab03 in the file Fiction_Paper.pdf).

EXERCISE 4.6 – EVALUATING 19TH CENTURY FICTION (10 MARKS)





In your own words, include a summary of the the main findings of the paper in your report.

EXERCISE 4.7 – REPEATING THE EVALUATION (40 MARKS)

Adapt the code from this lab (see above) to conduct your own investigation into the effectiveness of using word embeddings to characterize the fiction of one of the 19th century authors – either Jane Austen, Charles Dickens or Arthur Conan Doyle. You can easily download literature from Project Gutenberg or Kaggle (or any other online repository).

Include all of your code, output and **an analysis** in your report.

Endnotes

1. See, for example, work by [Benjamin Schmidt](#), [Ryan Heuser](#), and [Laura Nelson](#). 
2. Blankenship, Avery. “A Dataset of Nineteenth-Century American Recipes,” *Viral Texts: Mapping Networks of Reprinting in 19th-Century Newspapers and Magazines*. 2021. <https://github.com/ViralTexts/nineteenth-century-recipes/>.  
3. Many research questions in the humanities address bigger-picture concepts like gender, identity, or justice. A corpus the size of the one we are using here would be poorly suited to these kinds of research questions, because relevant terms are used in a diffuse set of contexts. As a general guideline, a million words is a minimum starting point for these kinds of queries. In our example, we are looking at a set of terms that appear with some frequency in a very consistent set of contexts, which makes it possible to produce reasonable results with a smaller corpus. Weavers and Koolen lay out a set of considerations around corpus size in greater detail, and the piece is worth consulting as you consider your own corpus. See Wevers, Melvin and Koolwen, Marijn. “Digital begriffsgeschichte: Tracing semantic change using word embeddings.” *Historical Methods: A Journal of Quantitative and Interdisciplinary History* 53, no. 4 (2020): 226-243. <https://doi.org/10.1080/01615440.2020.1760157>. 
4. For example, see Cordell, Ryan. “‘Q i-Jtb the Raven’: Taking Dirty OCR Seriously.” *Book History* 20, no. 1 (2017): 188–225. <https://doi.org/10.1353/bh.2017.0006> for a discussion of how OCR errors can provide useful information in research. See also Rawson, Katie, and Muñoz, Trevor. “Against Cleaning.” *Curating Menus*, July

2016.<http://www.curatingmenus.org/articles/against-cleaning/> for a discussion on the many and significant complexities that are often obscured under the concept of 'cleaning' data. 