CS 322: NATURAL LANGUAGE PROCESSING

Spring 2021 · Prof. James Ryan

Homework 5: Vectorial Semantics
Due: 21 May 2021

OVERVIEW

In 1957, the linguist J. R. Firth famously remarked, "You shall know a word by the company it keeps." This has become a mantra for the area of *distributional semantics*, whereby a word's distribution in the language—the contexts in which it tends to occur, i.e., the words that tend to surround it—is taken to be an actionable representation of its meaning. A few years later, the pioneers of *information retrieval* began to use computers to do *vectorial semantics*, which is an approach to NLP that takes distributional semantics to mathematical and computational extremes. Generally, in vectorial semantics, words are represented as vectors that characterize the contexts in which they occur in some large corpus of text. Likewise, chunks of such corpora, called *documents*, are likewise represented as vectors. This technology lies at the heart of search engines that accept textual queries and return textual results, such as Google and its competitors.

In this assignment, you'll be experimenting with an increasingly complex array of technologies in the area of vectorial semantics. This week's work is more about learning concepts than doing novel work, so I don't have a resume item for you this week. This document is long, but most of the work involves using code examples that I provide.

LOGISTICS

Before you get started, be sure that you're working in your group's Homework 5 repl. If someone doesn't have access, drop me a line right away. Like all assignments in this class, this one is to be completed by your homework group, and it's made up of a number of components. Each of these components will be graded on the `M-R-N` scale that is described in the syllabus. In turn, the grades attributed to each of your components will determine your group's grade on the entire assignment, again on the `M-R-N` scale. As the syllabus notes, you will be empowered in this class to continuously improve upon your work if it's missing the mark—this is called *repair*, and it will likely be a central part of your experience in this class. Do your best the first time around, but don't agonize over this assignment—if you complete it in good faith, you'll always have a chance to correct any errors or other issues! Please generate your homework submission PDF using our CS 322 Homework Template.

1. CORPUS EXTRACTION

In this assignment, you'll be conducting various experiments on a corpus of Wikipedia articles. For this component, you'll be constructing the corpus. Your goal here is for the corpus to comprise at least 1000 Wikipedia articles, where the articles pertain to one of three *topics*. You should have at least 250 documents for each topic. Make your topics distinct enough that a human could identify which articles pertain to which topic, and be sure to select at least one topic with which members of your group are deeply familiar (feel free to pick something niche).

To compile your Wikipedia articles, you'll be using the handy Python library `wikipedia`, which makes it super easy to extract the text of any Wikipedia article whose title is known. The title of a Wikipedia article generally comes at the end of the article's URL, with underscores replacing any whitespaces in the title. For instance, `Prince (musician)` is the title for Prince's Wikipedia article, which is hosted at the URL `https://en.wikipedia.org/wiki/Prince_(musician)`. Thus, if you have the URL for a Wikipedia article, you can easily extract the text of the article.[1] Here's a code example that extracts the content of article for Minneapolis:

```
import wikipedia
article_text = wikipedia.page("Minneapolis").content
```

The trickiest part of this component will be forming a list of all the titles for Wikipedia articles associated with a given topic. For this part, I recommend consulting the infamous Wikipedia article *List of lists of lists*. I'll leave it to you to decide what a good level of detail is for your topic, though you can feel free to reach out if you're unsure about a prospective one. You can expect to dive a few levels deep in this nested list structure to find an actual list of articles for a topic. Once you find a promising list, I recommend viewing the page source to search for the URL's for the actual Wikipedia articles included in the list, which will generally look something like this:

```
<a href="/wiki/University_of_Minnesota" title="University of
Minnesota">University of Minnesota Twin Cities</a>
```

A good approach would be to have a single text file associated with each topic, with one article title on each line. You might come up with some way of populating these files semi-programatically, for instance by using a regex over the page source. Even so, you should expect this to be a bit tedious. I'm not aware of any Python library that allows you to extract article titles or contents by topic, likely because Wikipedia doesn't really have any formal notion of what a topic is. Once you have these text files with one article title on each line, you can load them one at a time, do a for loop over the contents, and then use the code above to extract the contents of the article for the title at hand.

Your corpus itself should be structured as a directory with three subdirectories, one for each topic, and these subdirectories should each contain a number of files, each holding only the text of an individual Wikipedia article. Be sure to name each file as the article's title (it's fine to include

[1] Note: the module works better if you replace all underscores in the URL with whitespace characters.

underscores in place of whitespace characters). Place your corpus directory in your Homework 5 repl and name it `corpus`.

2. BAG OF WORDS

The simplest approach to vectorial semantics, popularized in the 1970s, is called the *bag-of-words*, or *BOW*, *model*. By this method, a corpus is processed to construct a *document–term matrix*, which is a simple table that records how many times each term (i.e., token) in the corpus occurred in each document. Here, a document is simply a chunk of the corpus—e.g., in a corpus of Wikipedia articles, each article would be treated as a document. In the document–term matrix, each row corresponds to a document, each column corresponds to a term, and each cell contains an integer marking how many times that term (the column of the cell) appeared in that document (the row of the cell). Here's a simple example for a toy corpus comprising the "documents" `The state of MN is the land of lakes` and `The city of MPLS is the city of lakes`:

	`the`	`state`	`of`	`MN`	`is`	`land`	`lakes`	`MPLS`	`city`
`Doc 1`	2	1	2	1	1	1	1	0	0
`Doc 2`	2	0	2	0	1	0	1	1	2

These two documents are quite related semantically, and the term–document matrix captures this by recording the degree to which the same words were used in each document. Critically, however, the matrix does not capture word order—this is why it's called a *bag of words* (imagine the words composing each document being thrown into a bag, thereby losing all the aspects of meaning that depend on the words being in a particular order).

The cleverness of the BOW model is in the treatment of the rows and columns in the matrix as *vectors*, which in this case simply means an array of numbers. In this conception, we have two kinds of vectors: the rows are *document vectors*, and the columns are *word vectors*. For instance, the word vector for `the` is `[2, 2]`, while the document vector for `Doc 2` is `[2, 0, 2, 0, 1, 0, 1, 1, 2]`. By treating these as vectors, we can take the *cosine* between them, which will provide a single real-valued number in the range [-1, 1] that captures the *similarity* of the two vectors. In this case, the vectors represent things—words or documents—and so cosine similarity can be treated as a measure of the relatedness of those things. If the documents describe particular concepts, like Wikipedia articles do, then the cosine similarity between two document vectors can be treated as a measure of the relatedness of those *concepts*.

In case it's not clear, it only makes sense to compare documents to other documents and words to other words. In this sense, the document–term matrix contains two *vector spaces*, one containing only documents and the other containing only words. In the document space, for instance, the document vectors can be thought of as coordinates in a high-dimensional space—

in this toy example, the document space is 9-dimensional, but in the case of an actual corpus, you'd have thousands and thousands of dimensions (one for every unique token in the corpus). In this high-dimensional space, related documents will be closer in proximity and less related documents will be farther away; likewise in the word space. This is called the *vector space model of semantics*.

Okay, with that background in place, it's time to start coding. Place your code for this component in `component2.py` of the Homework 5 repl. In this component, you'll be constructing your own BOW model on your corpus from the previous component. To complete this component, carry out the following steps:

a. First, you need to prepare the corpus that you constructed for the previous component. For our purposes here, you'll be representing each document (Wikipedia article) as a collection of tokens, all converted to lowercase. This means that you'll need to tokenize your text—feel free to use whichever tokenizer you like (we've already used a few in this class). Additionally, you need to assign unique identifiers to the tokens and to the documents. Each unique token should be given an integer identifier in the range [0, *number of unique tokens*); there should be no gaps in the range. Do the same with the documents, using the range [0, *number of documents*).

b. Now it's time to construct a document-term matrix for your corpus. In Python, it's best to represent such a matrix as a `numpy` array. Generally, you have to know the size of a `numpy` array before initializing it, so first count the number of unique tokens in your entire corpus. Next, initialize an array contains only zeros, with the dimensions *number of unique tokens* x *number of documents*. Assuming `numpy` is imported as `np` (this is a convention), you can do this using the `np.zeros()` function. For example, the snippet `np.zeros((2, 3))` constructs an array of zeros with two rows and three columns. Now, iterate over each token occurrence in each document, incrementing the count of the cell in the a) row of the token and b) column of the document. For instance, upon encountering an occurrence of the token with ID `353` in the document with ID `22`, you'd do something like: `matrix[353][22] += 1`. After doing this for every token occurrence, you will have a fully constructed document–term matrix that records how many times each unique token in the corpus occurred in each document.

c. Let's carry out a simple experiment to determine if your simple BOW model can distinguish between the three topics in your corpus. Here, we'll rely on cosine similarity as a measure the relatedness of a pair of documents. You can take the cosine similarity between any two `numpy` arrays (call them `a` and `b`) using the `spatial` submodule in `scipy` (`from scipy import spatial`): `relatedness = 1 - spatial.distance.cosine(a, b)`.[2] Next, produce the following table and include it in your homework submission. In the table, the corpus topics will be on the rows and on the columns. In each cell of this table, put the average cosine similarity for all the documents in the row topic relative to all the documents

[2] We subtract from 1 to convert cosine *distance* into cosine *similarity*.

in the column topic. If the values along the diagonal of this matrix are significantly higher than the values of the other cells, your BOW model is doing a good job at distinguishing the corpus topics. Post this table in the `#results` channel of our Slack workspace, along with a brief explanation of your corpus and its topics; include a screenshot of this post in your submission. Be sure to explain a bit about your process for compiling the corpus.

d. Create a wrapper around your BOW model that allows a user to submit the title of a Wikipedia article in the model and receive a ranked listing of the ten most related articles to one that they submitted. If there is no article in your model with that title, indicate as much. When displaying the ranking, include the cosine similarities for each of the top ten articles. Now take the topic with which your group is most familiar and submit a few article titles for concepts that you know well. How good are the results? Include in your submission a response to this question.

e. How could you use your BOW model to infer the topic of some new, unseen Wikipedia article (assuming it indeed pertains to one of your three topics)? If the article does not pertain to one of your topics, how could you use your BOW model to infer this? Include your response to these questions in your homework submission.

3. TF-IDF

Regardless of how well your BOW model did, you might have started thinking about more principled approaches. The classic improvement over the basic BOW model is the *term frequency–inverse document frequency*, or *tf–idf*, model. This refinement over bag of words exploits the fact that words that occur frequently in a corpus tell you less about the documents in which they occur. Extreme cases of this are function words like `the`, `is`, and `and`, whereas the occurrence of a rare word like `achnelith` likely indicates that the document pertains to geology in some way. In a information-theoretic sense, the latter term provides more information about the document in which it occurs. From the standpoint of information retrieval (e.g., a search engine), such terms would better serve as keywords in a search query.

In tf–idf, "term frequency" refers to the number of times that a given term occurred in a given document—each cell count in your document–term matrix from last component is a term frequency. Often, the raw term frequency is divided by the number of tokens in the document, as a way to normalize for document length. The "inverse document frequency" for a term is a measure of the proportion of corpus documents that contain the term. It's typically calculated as *log(N/D)*, where *N* is the number of documents in the corpus and *D* is the number of documents containing the term. By taking the product of these two values for a given term and document, one derives a measure of the term's importance in describing the document. For instance, if the term `achnelith` only appears in five documents, but appears in them quite frequently, it probably provides a lot of information about those five documents.

In this component, you'll be refining your bag-of-words model to convert it into a tf–idf model. You can start by just pasting your code from `component2.py` into `component3.py`. Next, carry out the following steps:

a. Before doing the tf–idf analysis, you'll want to do some preprocessing on your corpus. First, convert all the text to lowercase. Next, you'll be removing stopwords. While tf–idf penalizes terms that appear all over in a corpus, a conventional step is to outright remove all *stopwords* beforehand. These are generally functional words that occur often and provide little information about the contexts in which they occur. I've added a file to your Homework 5 repl called `stopwords.txt`, which you can use for this purpose. Now remove all punctuation from your corpus. Next, you should remove all terms that appear in only one document. These terms provide no information with which the document can be related to other documents, so they're typically discarded both to improve the model and to speed up the computation. Finally, you're going to *lemmatize* all the words in the corpus. If you recall from the midterm project, a word's lemma is its canonical, or dictionary, form. For instance, the lemma of `giraffes` is `giraffe`, while the lemma for `swims` is `swim`. In vectorial semantics, we generally don't care about the inflection of a word, but rather its general meaning. Rather than representing `giraffe` and `giraffes` as two separate terms, we'd like to combine them into the canonical term `giraffe`. I recommend using the NLTK lemmatizer for this. Here's a code example:

```
>>> from nltk.stem.wordnet import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize('giraffes')
'giraffe'
>>> lemmatizer.lemmatize('swims')
'swim'
```

b. Next, you're going to construct the tf–idf matrix for your (preprocessed) corpus. Again, this will be a matrix with documents on the rows and terms on the columns, and again you will represent this as a `numpy` array. Instead of placing raw term frequencies in the cells, this time you'll be placing tf–idf values in them. Given a term and document, a tf–idf value can be derived in the following manner. The tf component is calculated as the number of occurrences of that term in that document divided by the number of total term occurrences in the document (i.e., its length after preprocessing). The idf component is calculated as *log(N/D)*, where *N* is the number of documents in the corpus and *D* is the number of documents containing the term at hand. Finally, the tf–idf value is the product of these two values. Once you've placed tf–idf values in each cell, you will have constructed a full tf–idf matrix for your (preprocessed) corpus.

c. Save your tf–idf matrix (i.e., the `numpy` array), since you'll be using it in the next component, too. Here's some code showing how to save and load numpy arrays:

```
example_array = np.zeros((3, 2))
outfile = open('/path/to/save/array/to/filename.dat', 'wb')
np.save(outfile, example_array)
outfile.close()

infile = open('/path/to/save/array/to/filename.dat', 'rb')
loaded_array = np.load(infile, allow_pickle=True)
```

d. Save a mapping from each unique term appearing in your preprocessed corpus to the number of its column in your tf–idf matrix. Your numbering scheme must span the range [0, *number of unique terms in the preprocessed corpus*), with no gaps, and it must correspond directly to your matrix. Now create a dictionary whose keys are the unique IDs and whose values are the associated terms. Save this dictionary as a JSON file. Here's some code showing how to save and load these JSON files (here, the dictionary is called `id2word`):

```
import json
outfile = open('/path/to/save/array/to/id2word.json', 'w')
outfile.write(json.dumps(id2word))
outfile.close()

infile = open('/path/to/save/array/to/id2word.json', 'r')
loaded_id2word = json.loads(infile)
# Convert the keys back to ints (JSON only allows str keys)
loaded_id2word = {int(k): v for k, v in loaded_id2word.items()}
```

e. Now, carry out the same experiment from Component 2c, this time using your tf–idf matrix. Post this new table in the `#results` channel, along with some discussion of whether it improved upon the results of your bag-of-words model from the previous experiment. Include a screenshot of this post in your submission.

f. Finally, carry out the same experiment from Component 2d, this time using your tf–idf model. How did these results compare to the earlier model?

4. LATENT SEMANTIC ANALYSIS

While tf–idf greatly improves upon the bag-of-words model, it is still susceptible to a major issue: two documents in a tf–idf model will only be related if they share many of the exact same words, but that's not how language tends to work. It's possible to describe similar things (or even a single thing) in many different ways, and humans hearing such variant descriptions still manage to pick up on the similarities.

Along these lines, the most impactful breakthrough after tf–idf, which was introduced in the 1970s, came in the form of *latent semantic analysis*, or LSA, which was invented in the late 1980s. In LSA, the tf–idf matrix is *reduced* to a much smaller matrix by a technique from linear algebra called the *singular-value decomposition*, or SVD. This much smaller matrix will still have the documents on the rows, but the columns now pertain to abstract *dimensions* that are inferred by the SVD in an unsupervised manner. Specifically, the SVD is a deterministic procedure that produces a reduced matrix that best maintains the statistical relations among the rows in the full matrix. Prior to invoking the SVD, one must select the number of dimensions for the reduced matrix; decades of experiments have shown that 200 to 500 dimensions tend to work best. These abstract dimensions can be conceived as pertaining to concepts that are important to the corpus. The values in the cells of the reduced matrix are positive or negative real-valued numbers that capture the degree to which the dimension is active in the document in the corresponding row. Remarkably, the reduced matrix does a far better job at capturing relatedness among the documents than does the original matrix that contains far more data![3] Rumor has it that Google used LSA in their search engine early on, leading to its crushing of the competition. The vector space represented by the reduced matrix is called an *embedding space*, because it embeds the high-dimensional vectors of the original tf–idf matrix into a smaller-dimensional space. In the context of vectorial semantics, such a space is often called a *word embedding*.

For this component, you'll be using a cool Python library called `gensim` to conduct LSA on your corpus.[4] To complete it, carry out the following steps and place your code in `component4.py`:

a. Load your `id2word` dictionary that you saved in Component 3d. Again, the IDs here must correspond exactly to the columns of your tf–idf matrix. I will refer to this dictionary in the example code below using the variable `id2word`.

b. Load your tf–idf matrix that you saved in Component 3c and then convert it into the format expected by `gensim`, using the following code (where `saved` is your saved array). I will refer to the resulting array in the example code below using the variable `tf_idf`.

```
tf_idf = [[(i, x) for i, x in enumerate(e)] for e in saved]
```

c. Use `gensim` to train a 200-dimensional LSA model on your corpus. This likely won't work on replit, due to RAM usage, but place your code in `component4.py` of your Homework 5 in any event. Here's code that you can use for this part:

```
from gensim.models import LsiModel
print("Training model (this may take a while!)...")
model = LsiModel(tf_idf, id2word=id2word, num_topics=200)
```

[3] In this great paper, the creators of LSA argue that this might shed light on the nature of language acquisition.

[4] Note: LSA is sometimes called *latent semantic indexing*, or LSI, which is how `gensim` refers to it.

```
document_lsa_vectors = [model[doc] for doc in tf_idf]
# It’s best to just ignore the special first dimension,
# which means our LSA vectors will actually be 199D
document_lsa_vectors = [np.array([component[1] for component in
vector[1:]]) for vector in document_lsa_vectors]
```

d. Next, you’re going to want to save your document vectors, which you should do while the `document_lsa_vectors` variable above is still in memory. The serialization format will be a `numpy` array whose entry arrays correspond to documents and each contain: 1) the topic of a document, and 2) the document vector for that document. This format is required for the visualization code that you’ll be using in the last component.

```
model_export_data = []
for doc_id in sorted(document_ids):
    document_vector = document_lsa_vectors[doc_id]
    document_topic = topics[doc_id]  # TODO MAKE THIS WORK
    document_entry = (document_topic, document_vector)
    model_export_data.append(document_entry)
# Convert to numpy array and save to file
model_export_data_array = np.array(model_export_data)
outfile = open(‘/path/to/save/array/to/lsa.dat', ‘wb')
np.save(outfile, model_export_data_array)
outfile.close()
```

e. Unlike BOW and tf–idf models, LSA models are not particularly interpretable, but we can inspect the individual dimensions to get a sense of the concepts to which they pertain. In LSA, the first dimension is a special one that isn’t worth inspecting. To inspect the second dimension of your model, use the snippet `model.print_topics()[1]`. You can inspect the third dimension using `model.print_topics()[2]`, the fourth dimension with `model.print_topics()[3]`, and so forth. How do the dimensions seem to be represented? What does the second dimension seem to stand for? How about the next few dimensions? At what point do the dimensions become uninterpretable to you? Include in your submission your team’s responses to these questions.

f. Again carry out the same experiment from Component 2c, but this time use the document LSA vectors that you just derived. Post this new table in the `#results` channel, along with some discussion of whether it improved upon the results of your models from the previous experiments. Include a screenshot of this post in your submission.

g. Finally, carry out the same experiment from Component 2d, this time using your LSA model. How did these results compare to the earlier models?

5. DOC2VEC

While LSA was essentially the start of the art for 25 years, in 2014 a new technique called *word2vec* took the NLP world by storm. This technique trains a neural network to fit vectors to the terms in a corpus using one of two methods, both of which rely on the context of a word. In the first method, centering on *continuous bag of words*, the neural network attempts to predict a missing word given some number of words both preceding and succeeding the blank word (the order of these words isn't considered). The other method, which pertains to *continuous skip-grams*, the neural network attempts to predict the surrounding words to a word at hand. In each case, the neural network learns an embedding scheme for fitting vectors to the words, and as these vectors get better through training, it gets better at achieving its task.

While word2vec derives a vector space whose points represent terms, a variant called *doc2vec* uses a similar methodology to produce a vector space whose points represent documents—just like in the models that you created in the previous components.

For this component, you'll be trying out the doc2vec functionality that `gensim` also supports. To complete it, carry out the following steps and place your code in `component5.py`:

a. Load in your raw text corpus and preprocess it in the same manner as in Component 3a. Now construct a list of lists, where each sublist contains all the tokens in a given document (each token being a single string). I recommend ordering the documents in terms of their IDs, so that you can easily access a document's topic. In the code below, I refer to this list of lists as `corpus`.

b. Use the following code to train your doc2vec model:

```
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
documents = [TaggedDocument(doc, [i]) for i, doc in
             enumerate(corpus)]
model = Doc2Vec(documents, vector_size=100, window=5,
min_count=1, workers=4)
document_vectors = [model.dv[i] for i in range(len(documents))]
```

c. Next, you're going to want to save your document vectors, which you should do while the `document_vectors` variable above is still in memory. The serialization format will be a `numpy` array whose entry arrays correspond to documents and each contain: 1) the topic of a document, and 2) the document vector for that document. This format is required for the visualization code that you'll be using in the last component.

```
model_export_data = []
for doc_id in sorted(document_ids):
    document_vector = document_vectors[doc_id]
```

```
        document_topic = topics[doc_id]  # TODO MAKE THIS WORK
        document_entry = (document_topic, document_vector)
        model_export_data.append(document_entry)
    # Convert to numpy array and save to file
    model_export_data_array = np.array(model_export_data)
    outfile = open('/path/to/save/array/to/doc2vec.dat', 'wb')
    np.save(outfile, model_export_data_array)
    outfile.close()
```

d. Again carry out the same experiment from Component 2c, but this time use the document doc2vec vectors that you just derived. Post this new table in the `#results` channel, along with some discussion of whether it improved upon the results of your models from the previous experiments. Include a screenshot of this post in your submission.

e. Finally, carry out the same experiment from Component 2d, this time using your doc2vec model. How did these results compare to the earlier models?

f. All of the models that you've encountered in this assignment were developed in the context of *information retrieval*. The idea is that you can take a search query, fit a vector to it in the semantic space of the model, and then take the cosine between the query vector and document vectors to find the most related documents. For instance, if the query is for an internet search engine, this method could be used to retrieve the most relevant web pages, where each is treated as a document. Generally, this notion of injecting a query into a pre-trained vector space is called *folding in*. Let's try it out with your doc2vec model! To fold a new query/document into a `gensim` doc2vec model, you can use a snippet like this: `vector = model.infer_vector(["system", "response"])`. Here, the query is `["system", "response"]`. Note that you should first preprocess the query using the same method employed in part a of this component. Go find three new Wikipedia articles, such that each one relates somewhat to one of your three topics and none of the others (find one new article for each topic). For each new document: a) fold it in to your space to fit a vector to it, b) calculate cosine similarities between that vector, c) isolate the five documents that had the highest cosine similarity with the new article, and d) print out the tiles and topics of those five articles. Was your model able to produce good results for your queries? Also try manually typing in some queries, in the style of an internet search engine, that should each retrieve articles of a certain topic. How did that go? Include in your submission responses to these questions.

6. MULTIDIMENSIONAL SCALING

While the per-topic relatedness tables that you've produced for each of the above components can reveal how well your model is performing, it's a shame that the high-dimensional vector spaces cannot themselves be visualized. Enter *t-distributed stochastic neighbor embedding*, or t-

SNE (pronounced 'tee-snee'), which is a technique for visualizing high-dimensional spaces using only two or three dimensions. While techniques like the SVD (from Component 4) are meant to maintain the general structure of high-dimensional spaces when reducing them to smaller ones, t–SNE instead seeks to maintain only local structure. Specifically, it seeks to produce a low-dimensional space in which the structure of each point's local neighborhood is preserved, and it does this at the cost of not maintaining structure between the neighborhoods themselves. This ends up producing really nice visualizations that, pending the quality of the underlying high-dimensional model, show nice clustering of items.

Here's a decent blog post on the topic, if you'd like to learn more. You can also check out a tool that I made in grad school called *GameSpace*, which is a t–SNE 3D embedding of an LSA model trained on Wikipedia articles about videogames; you can fly around in the 3D space and click on the "stars," each of which represents an actual videogame.

In this component, you'll be running some code (that I've prepared for you) to produce 2D visualizations of your LSA and doc2Vec vector spaces. To complete, carry out the following steps and place your code in `component6.py`:

a. Download my `component6.py` file that I posted on Slack, in the `#announcements` channel. Place this file in your Homework 5 repl.

b. First, we'll visualize your LSA vector space in 2D. To do this, simply modify the file path at the top of `component6.py` (so that the variable `PATH_TO_MODEL_DATA` holds a path to the `numpy` array that you saved in Component 4d) and then run `component6.py`. Take a screenshot of the resulting plot.

c. Next, we'll visualize your doc2vec vector space in 2D. To do this, modify the file path at the top of `component6.py` (so that the variable `PATH_TO_MODEL_DATA` holds a path to the `numpy` array that you saved in Component 5c) and then run `component6.py` again. Take a screenshot of the resulting plot.

d. Post both of the screenshots on Slack (indicate which is which), along with some discussion of which model appears to have performed better. Include a screenshot of this post in your submission.