soc128d_notebook_5_document_similarity

July 20, 2021

Sociology 128D: Mining Culture Through Text Data: Introduction to Social Data Science

1 Notebook 5: Document Similarity

In this notebook, we're going to go a bit farther with vector semantics, which is one of the main approaches we'll use in this class and which has had an enormous influence in cultural sociology. Specifically, we are going to build on Notebook 3 by using document-term matrices and tf-idf weighting as a basis for directly measuring how similar or dissimilar documents are.

Regarding the exercises at the end, don't worry if you aren't a history buff. The exercises are just meant to reinforce how text data can be used for social inquiry.

Please download the State of the Union Corpus (1790-2018), which was posted to Kaggle by Rachael Tatman and Liling Tan.

[1]:
```
import copy
import matplotlib.pyplot as plt
import os
import pandas as pd
import numpy as np
import re
import seaborn as sns
import spacy
import time

from collections import Counter
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity, euclidean_distances

sns.set_theme(style="darkgrid")
```

1.1 Loading and Cleaning the Data

We're going to load the data a bit like we did in Notebook 3. First, we're going to create a `list` called text_files using the `filter()` method. `filter()` uses a lambda function to filter out unwanted elements of an iterable (such as a list). The function and the iterable are the two parts of the call to `filter()`.

[2]:
```
text_files = list(filter(lambda x: x.endswith(".txt"), os.listdir("sotu")))
```

[3]:
```
text_files[:10]
```

[3]:
```
['Adams_1797.txt',
 'Adams_1798.txt',
 'Adams_1799.txt',
 'Adams_1800.txt',
 'Adams_1825.txt',
 'Adams_1826.txt',
 'Adams_1827.txt',
 'Adams_1828.txt',
 'Arthur_1881.txt',
 'Arthur_1882.txt']
```

In this application, `lambda x: x.endswith(".txt")` returns True if something is a string and ends with ".txt" or False otherwise.

The second part, `os.listdir("sotu")`, uses the `os` module's `listdir()` method to return a list of filenames in the directory.

Our call to `filter()` checks whether each filename in that directory ends with ".txt" and returns only the files for which that is true. We then cast the result as a `list`.

The result, text_files, is a list of filenames in the directory "sotu" that end with ".txt"–but the file names don't include the full path from our working directory to the files. We need to add the directory "sotu" to the filenames to access the files.

We're going to use `os.path.join()` to create a list of file paths. We'll call this list address_paths because it's a list of the file paths for State of the Union Addresses (stored as files ending in .txt). We'll use a list comprehension, but this is the same as looping through text_files with a for loop, using `os.path.join()`, and appending the result to the list we are creating.

[4]:
```
address_paths = [os.path.join("sotu", f) for f in text_files]
```

[5]:
```
print(address_paths[:10])
```

```
['sotu\\Adams_1797.txt', 'sotu\\Adams_1798.txt', 'sotu\\Adams_1799.txt',
'sotu\\Adams_1800.txt', 'sotu\\Adams_1825.txt', 'sotu\\Adams_1826.txt',
'sotu\\Adams_1827.txt', 'sotu\\Adams_1828.txt', 'sotu\\Arthur_1881.txt',
'sotu\\Arthur_1882.txt']
```

Now we're going to use these file paths to create a data frame with the text of each State of the Union as well as the president and year. The function return_sotu_name_year_text() accepts just one argument, a file path.

The file path points us to the text, but the *filename* includes the president and the year joined by an underscore.

Adams_1797.txt

Since the filename is just a string, we can easily associate the text of each State of the Union with the corresponding president and year.

[6]:
```
def return_sotu_name_year_text(f: str):
    """
    Return the name, year, and text of a SOTU.
    """
    doc = open(f, "r").read().strip()
    f = os.path.split(f)[-1]
    f = f.replace(".txt", "")
    pres, year = f.split("_")

    return pres, year, doc
```

The line

```
doc = open(f, "r").read().strip()
```

opens the file path f, reads the full file into memory as a string, and strips whitespace like linebreaks from the ends.

```
f = os.path.split(f)[-1]
```

replaces the file path we've stored as the variable f with the last part of the file path, in this case the filename ending in .txt, by splitting the file path and taking the last element by using the index -1.

```
pres, year = f.split("_")
```

creates the variables pres and year by splitting the filename f on underscores.

Now we can load the data. First, we create a dataframe with only one column: the file path. Next, we apply our function return_sotu_name_year_text() to each row's file path, creating three new columns.

[7]:
```
df = pd.DataFrame(address_paths, columns = ["file_path"])
```

[8]:
```
df.head()
```

[8]:

	file_path
0	sotu\Adams_1797.txt
1	sotu\Adams_1798.txt
2	sotu\Adams_1799.txt
3	sotu\Adams_1800.txt
4	sotu\Adams_1825.txt

[9]:
```
df[["president", "year", "text"]] = df.file_path.apply(lambda x: pd.
 ↪Series(return_sotu_name_year_text(x)))
df.drop(columns = ["file_path"], inplace = True)
```

We also drop the original column containing the file path. Now we have a dataframe with just the president, year, and text for each State of the Union.

[10]:
```
df.head()
```

[10]:

	president	year	text
0	Adams	1797	Gentlemen of the Senate and Gentlemen of the H…
1	Adams	1798	Gentlemen of the Senate and Gentlemen of the H…
2	Adams	1799	Gentlemen of the Senate and Gentlemen of the H…
3	Adams	1800	Gentlemen of the Senate and Gentlemen of the H…
4	Adams	1825	Fellow Citizens of the Senate and of the House…

[11]:
```
df.shape
```

[11]:
```
(228, 3)
```

Next, we sort the dataframe by year, remove any rows without actual speeches (i.e., where the text column is an empty string), identify any missing years, and reset the index.

[12]:
```
df = df[df["text"] != ""]
df = df.astype({"year": int})
df.year.min(), df.year.max()
```

[12]:
```
(1791, 2018)
```

[13]:
```
[i for i in range(1791,2019) if i not in df.year.values]
```

[13]:
```
[1933]
```

[14]:
```
len(df.index)
```

[14]:
```
227
```

[15]:
```
df.shape
```

[15]:
```
(227, 3)
```

[16]:
```
df.sort_values(by="year", inplace=True)
df.reset_index(inplace=True, drop=True)
```

[17]:
```
df.head()
```

[17]:

	president	year	text
0	Washington	1791	Fellow-Citizens of the Senate and House of Rep…
1	Washington	1792	Fellow-Citizens of the Senate and House of Rep…
2	Washington	1793	Fellow-Citizens of the Senate and House of Rep…
3	Washington	1794	Fellow-Citizens of the Senate and House of Rep…
4	Washington	1795	Fellow-Citizens of the Senate and House of Rep…

Finally, we want to make sure that we're distinguishing between presidents with the same last names. There were two Harrisons, but William Henry Harrison didn't live long enough to give a State of the Union address. Neither did James A. Garfield. Since Grover Cleveland counts twice,

that gives us 42 unique presidents who had given a State of the Union address in the time period covered by the corpus, which stops in 2018.

[18]:
```
df.president = np.where(df.president.eq("Adams") & df["year"].gt(1800),␣
 ↪"Adams2", df.president)
df.president = np.where(df.president.eq("Bush") & df["year"].gt(2000), "Bush2",␣
 ↪df.president)
df.president = np.where(df.president.eq("Johnson") & df["year"].gt(1900),␣
 ↪"Johnson2", df.president)
df.president = np.where(df.president.eq("Roosevelt") & df["year"].gt(1930),␣
 ↪"Roosevelt2", df.president)
```

[19]:
```
len(df.president.unique())
```

[19]:
```
42
```

[20]:
```
df.president.unique()
```

[20]:
```
array(['Washington', 'Adams', 'Jefferson', 'Madison', 'Monroe', 'Adams2',
       'Jackson', 'Buren', 'Tyler', 'Polk', 'Taylor', 'Fillmore',
       'Pierce', 'Buchanan', 'Lincoln', 'Johnson', 'Grant', 'Hayes',
       'Arthur', 'Cleveland', 'Harrison', 'McKinley', 'Roosevelt', 'Taft',
       'Wilson', 'Harding', 'Coolidge', 'Hoover', 'Roosevelt2', 'Truman',
       'Eisenhower', 'Kennedy', 'Johnson2', 'Nixon', 'Ford', 'Carter',
       'Reagan', 'Bush', 'Clinton', 'Bush2', 'Obama', 'Trump'],
      dtype=object)
```

We can use a `dict` and the `.apply()` method to create a column for the party of the president who gave the speech.

[21]:
```
party_dict = {
    'Washington': "Unaffiliated",
    'Adams': "Federalist",
    'Jefferson': "Democratic-Republican",
    'Madison': "Democratic-Republican",
    'Monroe': "Democratic-Republican",
    'Adams2': "Democratic-Republican",
    'Jackson': "Democrat",
    'Buren': "Democrat",
    'Tyler': "Whig",
    'Polk': "Democrat",
    'Taylor': "Whig",
    'Fillmore': "Whig",
    'Pierce': "Democrat",
    'Buchanan': "Democrat",
    'Lincoln': "Republican",
    'Johnson': "Democrat",
    'Grant': "Republican",
```

```
    'Hayes': "Republican",
    'Arthur': "Republican",
    'Cleveland': "Democrat",
    'Harrison': "Republican",
    'McKinley': "Republican",
    'Roosevelt': "Republican",
    'Taft': "Republican",
    'Wilson': "Democrat",
    'Harding': "Republican",
    'Coolidge': "Republican",
    'Hoover': "Republican",
    'Roosevelt2': "Democrat",
    'Truman': "Democrat",
    'Eisenhower': "Republican",
    'Kennedy': "Democrat",
    'Johnson2': "Democrat",
    'Nixon': "Republican",
    'Ford': "Republican",
    'Carter': "Democrat",
    'Reagan': "Republican",
    'Bush': "Republican",
    'Clinton': "Democrat",
    'Bush2': "Republican",
    'Obama': "Democrat",
    'Trump': "Republican"
}

df["party"] = df.president.apply(lambda x: party_dict[x])
```

[22]:
```
df[["party", "year"]].groupby("party").count() # number of speeches by party in␣
↪the dataset
```

[22]:

party	year
Democrat	92
Democratic-Republican	28
Federalist	4
Republican	89
Unaffiliated	6
Whig	8

[23]:
```
df[["president", "year"]].groupby("president").count() # number of speeches by␣
↪each pres in the dataset
```

[23]:

president	year
Adams	4

Adams2	4
Arthur	4
Buchanan	4
Buren	4
Bush	4
Bush2	8
Carter	4
Cleveland	8
Clinton	8
Coolidge	6
Eisenhower	8
Fillmore	3
Ford	3
Grant	8
Harding	2
Harrison	4
Hayes	4
Hoover	4
Jackson	8
Jefferson	8
Johnson	4
Johnson2	6
Kennedy	2
Lincoln	4
Madison	8
McKinley	4
Monroe	8
Nixon	5
Obama	8
Pierce	4
Polk	4
Reagan	7
Roosevelt	8
Roosevelt2	12
Taft	4
Taylor	1
Truman	8
Trump	2
Tyler	4
Washington	6
Wilson	8

1.2 Preprocessing the Text

[24]:
```
def preprocess_post(post: str) -> str:
    """
    Tokenize, lemmatize, remove stop words,
    remove non-alphabetic characters.
    """
    post = " ".join([word.lemma_ for word in nlp(post) if not word.is_stop])
    post = re.sub("[^a-z]", " ", post.lower())

    return re.sub("\s+", " ", post).strip()


nlp = spacy.load("en_core_web_sm", disable=["ner"])
```

The object we call nlp is a language model from spaCy. It does part-of-speech tagging, named entity recognition, and more. `disable=["ner"]` tells it not to perform named entity recognition. Turning things off might speed it up!

The function preprocess_post() is equivalent to the following:

```
def preprocess_post(post: str) -> str:
    """
    Tokenizes and returns the lowercase lemmas of
    tokens that are not stop words, minus any
    non-alphabetic characters
    """
    words = []
    for word in nlp(post): # each "word" in nlp(post) has been part-of-speech tagged, etc.
        if not word.is_stop: # ".is_stop" checks whether spacy has determined it's a stop word
            words.append(word.lemma_) # adding the lemma of the word, not the word itself, to
    post = " ".join(words) # converting the list of words to a string variable separated by sp
    post = post.lower() # make everything lowercase
    post = re.sub("[^a-z]", " ", post) # now we replace non-alphabetic chars with spaces
    post = re.sub("\s+", " ", post) # now we replace long stretches of whitespace with a singl
    post = post.strip() # now we strip whitespace from the edges

    return post
```

[25]:
```
start_time = time.time()

df["preprocessed"] = list(map(preprocess_post, df["text"].tolist()))

print(f"Preprocessed corpus in {(time.time() - start_time)/60:.1f} minutes")
```

```
Preprocessed corpus in 3.4 minutes
```

[26]:
```
df.head()
```

[26]:
```
   president  year                                                text  \
0  Washington  1791  Fellow-Citizens of the Senate and House of Rep…
1  Washington  1792  Fellow-Citizens of the Senate and House of Rep…
2  Washington  1793  Fellow-Citizens of the Senate and House of Rep…
3  Washington  1794  Fellow-Citizens of the Senate and House of Rep…
4  Washington  1795  Fellow-Citizens of the Senate and House of Rep…

         party                                        preprocessed
0  Unaffiliated  fellow citizens senate house representatives v…
1  Unaffiliated  fellow citizens senate house representatives a…
2  Unaffiliated  fellow citizens senate house representatives c…
3  Unaffiliated  fellow citizens senate house representatives m…
4  Unaffiliated  fellow citizens senate house representatives t…
```

1.3 Creating a Document-Term Matrix

Converting a corpus of documents to a `document-term matrix` is a core step for many NLP tasks. We saw how to do that in Notebook 3, but we'll review it now before introducing a faster way to complete this step.

We'll start by getting the number of times each *type* (unique word) occurs in the entire corpus. We'll save this as a `dict` called term_frequencies, which we'll create using the `Counter()` method.

[27]:
```
term_frequencies = Counter(" ".join(df["preprocessed"]).split())
vocabulary = list(term_frequencies.keys())
print(f"There are {len(vocabulary):,} unique words in the corpus.")
```

```
There are 17,541 unique words in the corpus.
```

Let's break this down a bit:

```
" ".join(df["preprocessed"]).split()
```

joins each of the preprocessed speeches with a single space, creating one big document with all of the *tokens* in the entire corpus. This would be a single string. The `str.split()` method then splits that string on whitespace (like spaces), returning a `list` containing all the tokens.

We then use the `Counter()` method to count the number of times each type occurs in that list, saving it as term_frequencies.

```
vocabulary = list(term_frequencies.keys())
```

accesses the keys (types) and casts the result as a `list`, giving us a list of the unique words.

Let's take a look at some of the most frequent words. Here, we create a `list` called tups (short for tuples) by accessing the `.items()` from term_frequencies. Each "item" is a tuple like (key, value), where the key and value are the type and count. We sort this list using a lambda function that checks the value at index 1. The value at index 1 of each tuple is the number of times the word occurs in the corpus. This means that

```
key = lambda x: x[1]
```

says we want to sort by the frequencies. We also set `reverse=True` to get a list in order from most frequent to least frequent. We then display the first ten using `tups[:10]`.

[28]:
```
tups = sorted(list(term_frequencies.items()), key=lambda x: x[1], reverse=True)
tups[:10]
```

[28]:
```
[('government', 7909),
 ('year', 6293),
 ('states', 6092),
 ('congress', 4975),
 ('united', 4746),
 ('country', 4440),
 ('people', 4251),
 ('great', 4117),
 ('nation', 3664),
 ('law', 3396)]
```

Now we filter the vocabulary to exclude words that occur only once.

[29]:
```
vocabulary = list(filter(lambda x: term_frequencies[x] > 1, vocabulary))
print(f"{len(vocabulary):,} unique words occur more than once.")
```

```
12,079 unique words occur more than once.
```

Next we define a function that accepts a string as an argument and returns a version of that string without any duplicate words.

[30]:
```
def set_of_types(document: str) -> str:
    """
    Returns a string with all duplicates of words removed
    """

    return " ".join(set(document.split()))
```

The function set_of_types() uses the `str.split()` method to split the speech (a string) on whitespace, casts it as a `set()` (removing any duplicates), then uses the `.join()` method to join the tokens into a single string again using whitespace.

Now we create a column called types by applying this function to the preprocessed text, row by row. This creates a copy of each speech without any duplicate words.

[31]:
```
df["types"] = df.preprocessed.apply(set_of_types)
```

Why do we do that? It's just a convenient way to count the number of documents each word occurs in. Just like the code we used to create term_frequencies, the code to create document_frequencies first joins the types with a single space, creating one big document. It then uses `str.split()` to convert that giant document into a list of tokens, and then uses the `Counter()` method to count how many times each type occurs.

We can get rid of the types column. It has served its purpose: helping avoid going word by word through each document to get the number of documents in which each word occurs.

[32]:
```
document_frequencies = Counter(" ".join(df.types).split())
df.drop(columns=["types"], inplace=True)
```

Words that occur in only one document don't give us a lot of information, so we'll filter them out.

[33]:
```
vocabulary = list(filter(lambda x: document_frequencies[x] > 1, vocabulary))
print(f"The vocabulary now has {len(vocabulary):,} words.")
```

```
The vocabulary now has 11,528 words.
```

Now we're almost ready to create the `document-term matrix`. We'll create a copy of our dataframe and call it dtm (for "document-term matrix"). We're only going to keep the column with the preprocessed text. We're also going to convert this from a string for each speech to a list of tokens for each speech using `str.split()` and then rename the column "<preprocessed>" somewhat arbitrarily; we're about to create a column for each unique word, and while we might not expect "preprocessed" to be in the vocabulary, it's good practice.

[34]:
```
dtm = copy.copy(df)
dtm.preprocessed = dtm.preprocessed.apply(str.split)
dtm = dtm[["preprocessed"]]
dtm.rename(columns={"preprocessed": "<preprocessed>"}, inplace = True)
dtm.head()
```

[34]:

	<preprocessed>
0	[fellow, citizens, senate, house, representati…
1	[fellow, citizens, senate, house, representati…
2	[fellow, citizens, senate, house, representati…
3	[fellow, citizens, senate, house, representati…
4	[fellow, citizens, senate, house, representati…

Now we're going to create a column for each word in the vocabulary. Each row still corresponds to a State of the Union, and the value for each cell will be the number of times that word occurs in that (preprocessed) speech. Since we've converted the preprocessed text to a list, we can use the `.count()` method to get the number of times each word in the vocabulary occurs in each document.

The function term_frequency() uses a list comprehension and returns a list of word counts for each speech. We can then create the columns with the counts for each word. Next, we drop the "<preprocessed>" column because we no longer need the text anymore for the document-term matrix.

[35]:
```
def term_frequency(doc: list, vocab: list) -> list:
    """
    Returns counts of each term in a list
    """

    return [doc.count(term) for term in vocab]
```

[36]:
```
dtm[list(vocabulary)] = dtm["<preprocessed>"].apply(lambda x: pd.
↪Series(term_frequency(x, vocabulary)))
```

[37]:
```
dtm.drop(columns=["<preprocessed>"], inplace=True)
dtm.head()
```

[37]:

	fellow	citizens	senate	house	representatives	vain	expect	peace	\
0	1	1	3	3	3	1	2	4	
1	1	1	2	3	3	0	1	5	
2	2	1	2	3	3	0	1	6	
3	3	1	2	5	3	0	0	3	
4	2	1	4	4	4	0	1	4	

	indians	frontier	…	isil	bully	beijing	grieve	remake	austin	\
0	5	2	…	0	0	0	0	0	0	
1	1	3	…	0	0	0	0	0	0	
2	2	1	…	0	0	0	0	0	0	
3	2	0	…	0	0	0	0	0	0	
4	9	4	…	0	0	0	0	0	0	

	dreamer	isis	obamacare	mattis
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

[5 rows x 11528 columns]

1.4 A Faster Way: CountVectorizer

We can do this much more quickly with `scikit-learn's` `CountVectorizer()` method.

The argument `min_df=2` excludes words that occur in fewer than two documents.

Let's compare the results!

[38]:
```
count_vectorizer = CountVectorizer(min_df=2)
counts = count_vectorizer.fit_transform(df["preprocessed"])
counts.shape
```

[38]:
```
(227, 11503)
```

We'll use `pd.DataFrame.sparse.from_spmatrix()` to convert the result to a dataframe.

The vocabulary is in a different order, and it's 25 columns narrower. Let's see what's missing.

[39]:
```
new_df = pd.DataFrame.sparse.from_spmatrix(counts, columns=count_vectorizer.
↪get_feature_names())
```

[40]:
```
new_df.head()
```

[40]:

	aaron	abandon	abandonment	abate	abatement	abating	abdicate	\
0	0	0	0	0	0	0	0	
1	0	0	0	0	1	0	0	
2	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	
4	0	1	0	0	0	0	0	

	abdication	abet	abettor	…	zeal	zealand	zealous	zealously	zelaya	\
0	0	0	0	…	0	0	1	0	0	
1	0	0	0	…	1	0	1	0	0	
2	0	0	0	…	0	0	0	0	0	
3	0	1	0	…	1	0	0	0	0	
4	0	0	0	…	0	0	0	0	0	

	zero	zimbabwe	zinc	zone	zuloaga
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

[5 rows x 11503 columns]

[41]:
```
diffs = list(set(dtm.columns).difference(set(new_df.columns)))
```

[42]:
```
len(diffs)
```

[42]: 25

[43]:
```
print(sorted(diffs))
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

`CountVectorizer()` has removed single characters from the vocabulary.

1.5 Measuring Similarity

This brings us to a few of the key intuitions of text analysis. We now have **vectors** instead of strings as representations of the speeches, and these vectors can be compared quanitatively. More specifically, the documents can now be compared quantitatively as if they were points in a high-dimensional space.

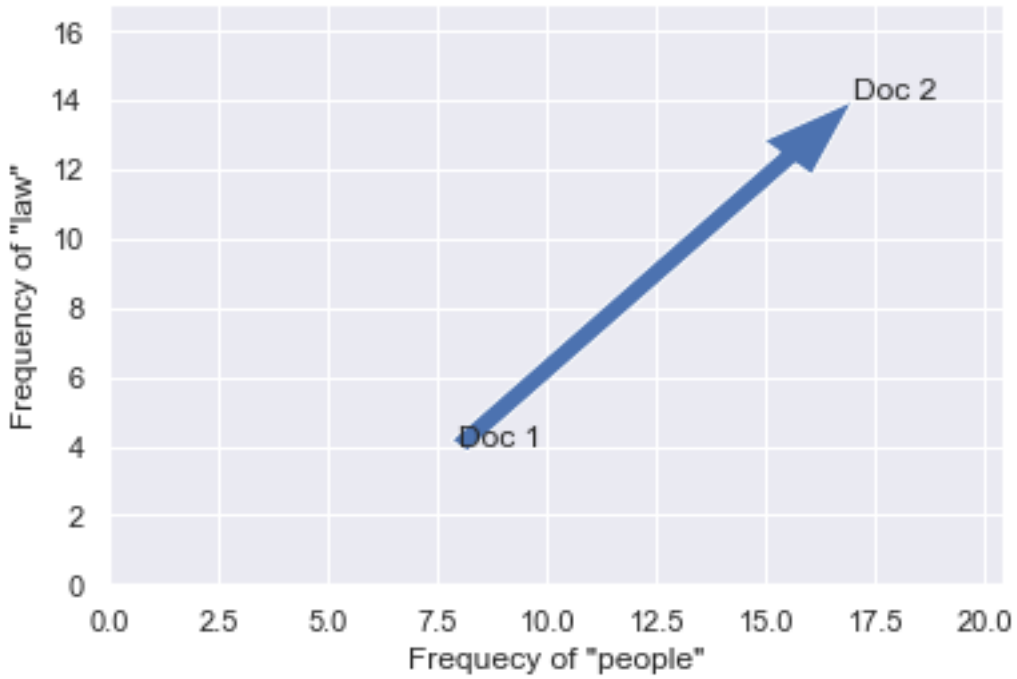
Let's start with a simple case of a vocabulary of two words. You can assign different words to keyword1 and keyword2 below, and documents are selected at random. We're going to show where documents are in a *two*-dimensional space using counts of these two words. The function plot_distances() handles that for us.

[44]:
```
def plot_distances(doc1_idx: int, doc2_idx: int, keyword1: str, keyword2: str,␣
↪extend=False, cosine=False):
    """
    Plots an arrow illustrating the distance between two
    2D "word vectors" based on term frequencies
    """
    x1 = dtm.loc[doc1_idx, keyword1]
    y1 = dtm.loc[doc1_idx, keyword2]
    x2 = dtm.loc[doc2_idx, keyword1]
    y2 = dtm.loc[doc2_idx, keyword2]

    doc1 = min([[x1, y1], [x2, y2]], key=lambda x: np.sqrt(x[0]**2 + x[1]**2))
    doc2 = max([[x1, y1], [x2, y2]], key=lambda x: np.sqrt(x[0]**2 + x[1]**2))

    if extend==True:
        doc3 = [doc2[0]*2, doc2[1]*2]
        plt.xlim(0, max(x1, x2, doc3[0])*1.2)
        plt.ylim(0, max(y1, y2, doc3[1])*1.2)
        plt.text(x = doc3[0], y = doc3[1], s = "Doc 3")
        plt.arrow(doc1[0], doc1[1], doc3[0]-doc1[0], doc3[1]-doc1[1], width=0.
↪5, length_includes_head=True)
    else:
        plt.xlim(0, max(x1, x2) * 1.2)
        plt.ylim(0, max(y1, y2) * 1.2)
    plt.arrow(doc1[0], doc1[1], doc2[0]-doc1[0], doc2[1]-doc1[1], width=0.5,␣
↪length_includes_head=True)
    plt.text(x = doc1[0], y = doc1[1], s = "Doc 1")
    plt.text(x = doc2[0], y = doc2[1], s = "Doc 2")
    plt.xlabel(f'Frequecy of "{keyword1}"')
    plt.ylabel(f'Frequency of "{keyword2}"')

    print(f"Document 1 features '{keyword1}' {doc1[0]} times and '{keyword2}'␣
↪{doc1[1]} times.")
    print(f"Document 2 features '{keyword1}' {doc2[0]} times and '{keyword2}'␣
↪{doc2[1]} times.")
    print(f"Documents 1 and 2 are {euclidean_distances([doc1], [doc2])[0][0]:.
↪1f} units apart in this 2D space.")

    if extend==True:
        print(f"\nDocument 3 features '{keyword1}' {doc3[0]} times and␣
↪'{keyword2}' {doc3[1]} times.")
        print(f"Documents 1 and 3 are {euclidean_distances([doc1],␣
↪[doc3])[0][0]:.1f} units apart in this 2D space.")
        if cosine==True:
            print(f"\nDocuments 1 and 2 have a cosine similarity of␣
↪{cosine_similarity([doc1], [doc2])[0][0]:.2f}.")
```

```
            print(f"Documents 1 and 3 have a cosine similarity of␣
↪{cosine_similarity([doc1], [doc3])[0][0]:.2f}.")
            print(f"Documents 2 and 3 have a cosine similarity of␣
↪{cosine_similarity([doc2], [doc3])[0][0]:.2f}.")
```

[92]:
```
doc1_idx = np.random.randint(0, dtm.shape[0])
doc2_idx = np.random.randint(0, dtm.shape[0])

keyword1 = "people"
keyword2 = "law"

plot_distances(doc1_idx, doc2_idx, keyword1, keyword2)

plt.show()
```

```
Document 1 features 'people' 8 times and 'law' 4 times.
Document 2 features 'people' 17 times and 'law' 14 times.
Documents 1 and 2 are 13.5 units apart in this 2D space.
```



This arrow represents the distance between the two documents in this two-dimensional space, and we can calculate the length of the arrow directly using the Pythagorean theorem. This gives us the Euclidean distance between the points.

If we make the assumption that the meaning of the documents is related to the distribution of words in them, then we can make the additional assumption that the distance between the points is related to how semantically similar the documents are. In other words, we might assume that

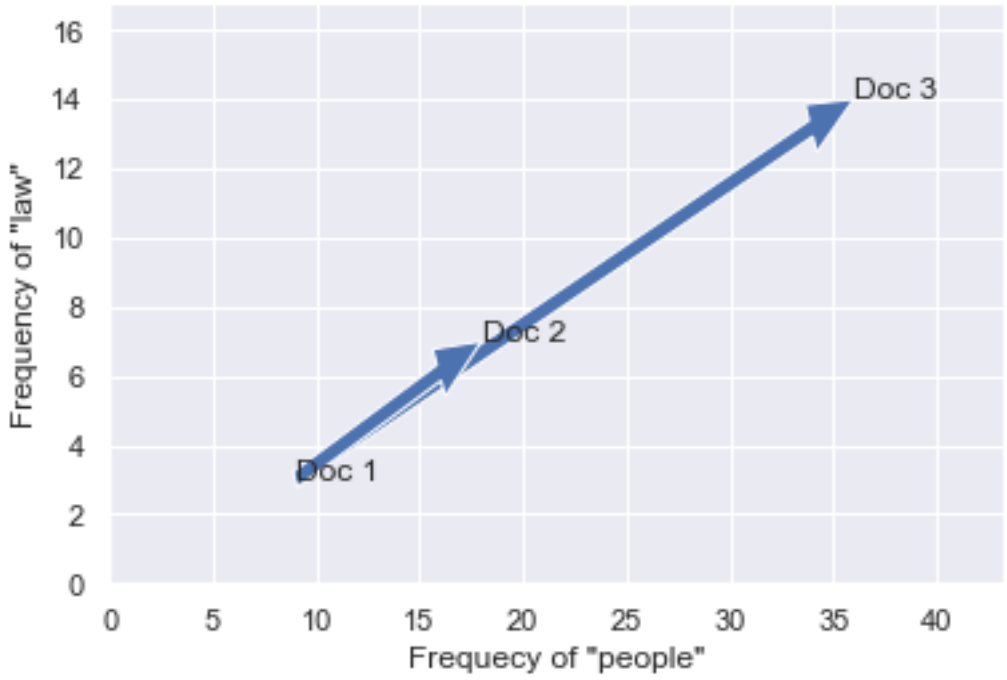
two documents close together mean something similar, whereas documents farther apart are less likely to mean (or be about) the same thing.

What about document length, though?

Consider the (contrived) example below. Document 3 is identical to Document 2, but twice as long. It's farther from Document 1, but it's in the exact same direction as Document 2 from the origin.

[104]:
```
doc1_idx = np.random.randint(0, dtm.shape[0])
doc2_idx = np.random.randint(0, dtm.shape[0])

keyword1 = "people"
keyword2 = "law"

plot_distances(doc1_idx, doc2_idx, keyword1, keyword2, extend=True)

plt.show()
```

```
Document 1 features 'people' 9 times and 'law' 3 times.
Document 2 features 'people' 18 times and 'law' 7 times.
Documents 1 and 2 are 9.8 units apart in this 2D space.

Document 3 features 'people' 36 times and 'law' 14 times.
Documents 1 and 3 are 29.2 units apart in this 2D space.
```



If we use Euclidean distance as our measure of similarity, this makes it look like Document 3 is much less similar to Document 1 than Document 2 is to Document 1, and like Documents 2 and 3

are not very similar. Euclidean distance is a problem in this case.

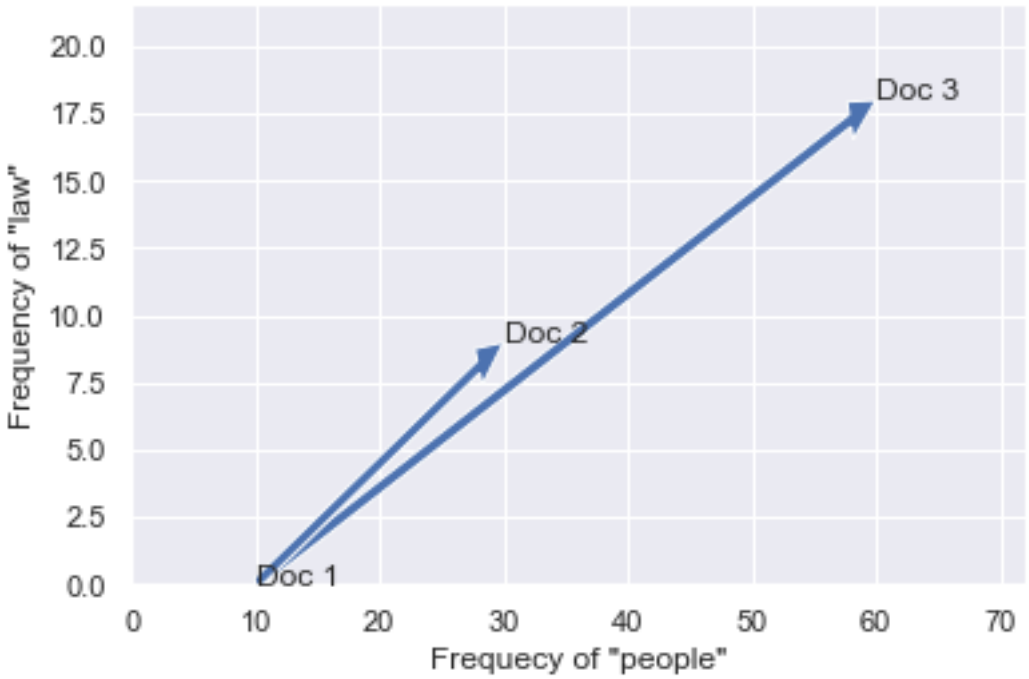
We can use cosine similarity instead. Whereas Euclidean distance measures the length of the arrow between the two points, cosine similarity takes into account the *angle*. Document 3 is longer than Document 2, but it has the same proportions of the keywords.

If we use cosine, two important points emerge: 1. The similarity of Document 2 and Document 3 will be 1.0 2. The similarity of Document 1 to Document 2 will be the same as the similarity of Document 1 to Document 3

These are both desirable because Document 2 and 3 only differ in length.

[106]:
```
doc1_idx = np.random.randint(0, dtm.shape[0])
doc2_idx = np.random.randint(0, dtm.shape[0])

keyword1 = "people"
keyword2 = "law"

plot_distances(doc1_idx, doc2_idx, keyword1, keyword2, extend=True, cosine=True)

plt.show()
```

```
Document 1 features 'people' 10 times and 'law' 0 times.
Document 2 features 'people' 30 times and 'law' 9 times.
Documents 1 and 2 are 21.9 units apart in this 2D space.

Document 3 features 'people' 60 times and 'law' 18 times.
Documents 1 and 3 are 53.1 units apart in this 2D space.

Documents 1 and 2 have a cosine similarity of 0.96.
Documents 1 and 3 have a cosine similarity of 0.96.
Documents 2 and 3 have a cosine similarity of 1.00.
```



We can also use these measures beyond this two-dimensional case.

1.6 Tf-idf Revisited

> In Section 6.2…we developed the notion of a document vector that captures the relative importance of the terms in a document. The representation of a set of documents as vectors in a common vector space is known as the vector space model and is fundamental to a host of information retrieval (IR) operations including scoring documents on a query, document classification, and document clustering. Manning, Raghavan, & Schutze (2008, p. 110)

In Notebook 3, we also encountered tf-idf weighting. The key idea is that having rare things in common is more informative than having common things in common. It is not terribly informative if two documents share really common words. Inverse document frequency (idf) weighting helps us assign less importance to words that appear in many documents. We also experimented a bit with weighting the frequencies of terms within documents by logging them. There are many ways to calculate term frequency and inverse document frequency.

`sklearn` does offer the most flexibility in terms of weighting systems, but it does make tf-idf weighting fast. If we want to calculate a document-term matrix and apply tf-idf weighting, we can use `sklearn's TfidfVectorizer()`.

[48]:
```
tfidf_vectorizer = TfidfVectorizer(min_df=2, sublinear_tf=True) # sublinear_tf␣
↪logs the term frequencies
tfidf = tfidf_vectorizer.fit_transform(df["preprocessed"])
tfidf.shape
```

[48]:
```
(227, 11503)
```

[49]:
```
tfidf_df = pd.DataFrame.sparse.from_spmatrix(tfidf, columns=tfidf_vectorizer.
↪get_feature_names())
tfidf_df.head()
```

[49]:

	aaron	abandon	abandonment	abate	abatement	abating	abdicate	\
0	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0	
1	0.0	0.000000	0.0	0.0	0.076562	0.0	0.0	
2	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0	
3	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0	
4	0.0	0.027238	0.0	0.0	0.000000	0.0	0.0	

	abdication	abet	abettor	…	zeal	zealand	zealous	zealously	\
0	0.0	0.000000	0.0	…	0.000000	0.0	0.052076	0.0	
1	0.0	0.000000	0.0	…	0.038764	0.0	0.057123	0.0	
2	0.0	0.000000	0.0	…	0.000000	0.0	0.000000	0.0	
3	0.0	0.060665	0.0	…	0.029356	0.0	0.000000	0.0	
4	0.0	0.000000	0.0	…	0.000000	0.0	0.000000	0.0	

	zelaya	zero	zimbabwe	zinc	zone	zuloaga
0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0

[5 rows x 11503 columns]

1.7 Finding Similar Documents

[50]:
```
def find_most_similar(query: str, num_matches: int=1, cosine: bool=True) ->␣
↪list:
    """
    Preprocess a query and find `num_matches` using tf-idf and either cosine or␣
↪Euclidean distance
    """
    query = preprocess_post(query)
    query = tfidf_vectorizer.transform([query])

    if cosine==True:
        matches = [(idx, cosine_similarity(query, np.array(post).
↪reshape(1,-1))[0][0]) for idx, post in tfidf_df.iterrows()]
    else:
        matches = [(idx, euclidean_distances(query, np.array(post).
↪reshape(1,-1))[0][0]) for idx, post in tfidf_df.iterrows()]
```

```
    matches = sorted(matches, key=lambda x: x[1], reverse=True)

    return matches[:num_matches]
```

Let's make sure everything is working. The most similar document to any given speech should be the speech itself!

Here's a random selection:

[51]:
```
query = df.sample(1)["text"].values[0]
```

[52]:
```
print(query[:500])
```

```
Members of the Congress:

In reporting to the Congress the state of the Union, I find it impossible
to characterize it other than one of general peace and prosperity. In some
quarters our diplomacy is vexed with difficult and as yet unsolved
problems, but nowhere are we met with armed conflict. If some occupations
and areas are not flourishing, in none does there remain any acute chronic
depression. What the country requires is not so much new policies as a
steady continuation of those which are
```

Now let's get the tf-idf-weighted version of it. Notably, *these weights are based on the document frequencies in the original corpus*. While the resulting vector represents the new query, the relative importance assigned to each word means we have already incorporated information from the other documents.

[53]:
```
query_tfidf = tfidf_vectorizer.transform([query])
```

[54]:
```
query_tfidf
```

[54]:
```
<1x11503 sparse matrix of type '<class 'numpy.float64'>'
        with 1469 stored elements in Compressed Sparse Row format>
```

Now let's find the most similar speech.

[55]:
```
find_most_similar(query, num_matches=1)
```

[55]:
```
[(135, 1.0)]
```

[56]:
```
print(df.loc[216].text[:500]) # the speech is selected at random, so you will␣
↪need to change the index
```

```
Madam Speaker, Vice President Cheney, Members of Congress, distinguished
guests, and fellow citizens:

Seven years have passed since I first stood before you at this rostrum. In
that time, our country has been tested in ways none of us could have imagined.
We have faced hard decisions about peace and war, rising competition in the
world economy, and the health and welfare of our citizens. These issues call
```

```
for vigorous debate, and I think its fair to say weve answered that call. Yet
history will
```

We can also compare the documents to new documents, for example to other figures who influenced social thought. This quote is an arbitrary example (but comes from Simmel's "Fashion" in *Georg Simmel on Individuality and Social Forms*.

[57]:
```
query = """The charm of imitation in the first place is to be found in the fact␣
 ↪that it makes possible an expedient
test of power, which, however, requires no great personal or creative␣
 ↪application, but is displayed easily and
smoothly, because its content is a given quantity. We might define it as the␣
 ↪child of thought and thoughtlessness.
It affords the pregnant possibility of continually extending the greatest␣
 ↪creations of the human spirit, without the
aid of the forces which were originally the very condition of their birth.␣
 ↪Imitation, furthermore, gives to the
individual the satisfaction of not standing alone in his actions. Whenever we␣
 ↪imitate, we transfer not only the
demand for creative activity, but also the responsibility for the action from␣
 ↪ourselves to another. Thus the
individual is freed from the worry of choosing and appears simply as a creature␣
 ↪of the group, as a vessel of the
social contents..."""
```

[58]:
```
find_most_similar(query, num_matches=1)
```

[58]:
```
[(179, 0.08459199532015019)]
```

[59]:
```
df.loc[179]
```

[59]:
```
president                                                    Nixon
year                                                          1971
text            Mr. Speaker, Mr. President, my colleagues in t…
party                                                   Republican
preprocessed    mr speaker mr president colleague congress dis…
Name: 179, dtype: object
```

1.8 Document Similarity as an Outcome Variable

The ability to position documents in a shared vector space also means we can treat document similarity as an outcome. We can compare many things (like an entire corpus!) to a single document, calculating the similarity (or distance) for each comparison. We can look at relationships between document similarity and other variables.

1.8.1 Example 1: Similarity to First Document

Let's pick an obvious starting point: the first document. We'll calculate the similarity between each subsequent document and the first document and then observe trends in that measure over

time.

[60]:
```
df.loc[0]
```

[60]:
```
president                                                Washington
year                                                           1791
text            Fellow-Citizens of the Senate and House of Rep…
party                                                  Unaffiliated
preprocessed    fellow citizens senate house representatives v…
Name: 0, dtype: object
```

[61]:
```
query = df.loc[0].text # the text of the first row in the dataframe
```

[62]:
```
print(query[:500])
```

```
Fellow-Citizens of the Senate and House of Representatives:

"In vain may we expect peace with the Indians on our frontiers so long as a
lawless set of unprincipled wretches can violate the rights of hospitality,
or infringe the most solemn treaties, without receiving the punishment they
so justly merit."

I meet you upon the present occasion with the feelings which are naturally
inspired by a strong impression of the prosperous situations of our common
country, and by a persuasion equally stron
```

We already have the vector. Let's assign it to a variable called query_tfidf and check the `shape`.

[63]:
```
query_tfidf = tfidf_df.loc[0]
query_tfidf.shape
```

[63]:
```
(11503,)
```

When we retrieve it using `.loc` and the index (row), it is giving us a data structure with a row for every word in the vocabulary. We want to make sure that stays as a single row with a column for every word. We can use `numpy's .reshape()` method. `.reshape(1, -1)` will do the trick.

[64]:
```
query_tfidf = np.array(query_tfidf).reshape(1, -1)
query_tfidf.shape
```

[64]:
```
(1, 11503)
```

Now let's create a new variable and call it sim_to_first. We'll initially save this as a list, and we'll create it using a list comprehension that compares the tf-idf-weighted vector representation of the first speech to the vector for each speech in order.

[65]:
```
sim_to_first = [cosine_similarity(query_tfidf, np.array(post).
 ↪reshape(1,-1))[0][0] for idx, post in tfidf_df.iterrows()]
```

We've kept everything in the same order, but let's put our minds at ease by first confirming there are the same number of speeches.

[66]:
```
len(sim_to_first) == df.shape[0] == dtm.shape[0] == tfidf_df.shape[0] == new_df.
↪shape[0]
```

[66]:
```
True
```

And to be extra safe, let's make sure the first result–which should be comparing the first speech to itself–has a similarity of 1.0. There will always be some noise, but this is effectively 1.0:

[67]:
```
print(sim_to_first[0])
```

```
1.0000000000000004
```

Looks good! Now we can add the new variable to our original dataframe, which has useful metadata, namely the president who gave the speech and the year it was given.

[68]:
```
df["sim_to_first"] = sim_to_first
```

[69]:
```
df.head()
```

[69]:

	president	year	text	party	preprocessed	sim_to_first
0	Washington	1791	Fellow-Citizens of the Senate and House of Rep…	Unaffiliated	fellow citizens senate house representatives v…	1.000000
1	Washington	1792	Fellow-Citizens of the Senate and House of Rep…	Unaffiliated	fellow citizens senate house representatives a…	0.359244
2	Washington	1793	Fellow-Citizens of the Senate and House of Rep…	Unaffiliated	fellow citizens senate house representatives c…	0.310133
3	Washington	1794	Fellow-Citizens of the Senate and House of Rep…	Unaffiliated	fellow citizens senate house representatives m…	0.273526
4	Washington	1795	Fellow-Citizens of the Senate and House of Rep…	Unaffiliated	fellow citizens senate house representatives t…	0.315451

Using seaborn, pandas, or pyplot itself with missing data may mean missing datapoints are interpolated. We can see this if we explicitly interperolate a value for the missing year, 1933.

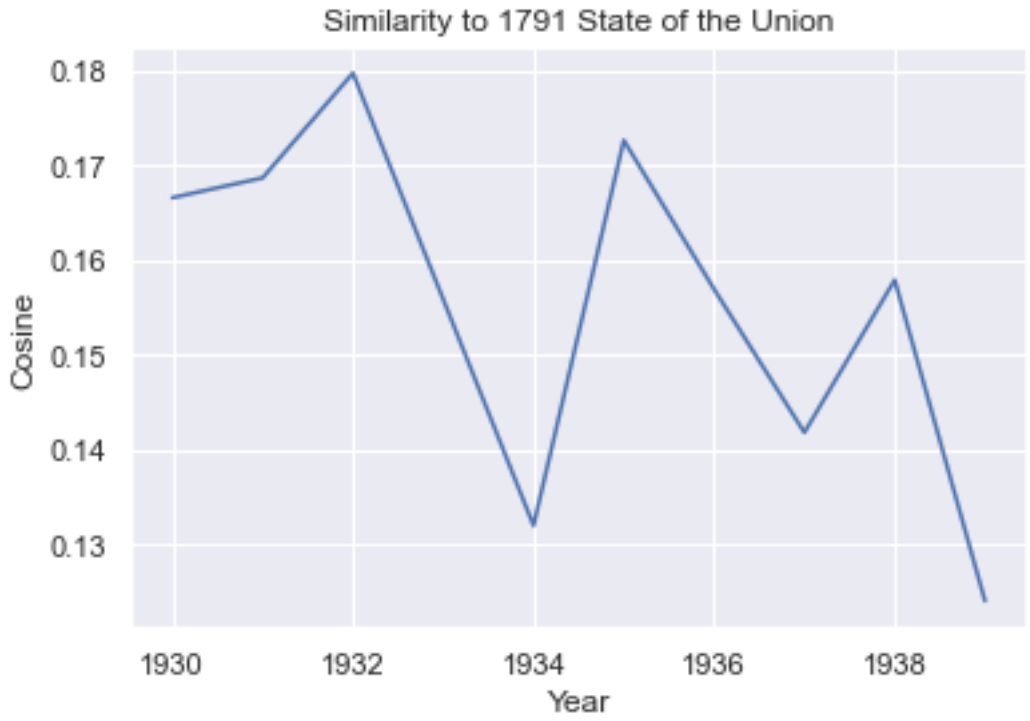
[70]:
```
[i for i in range(1791,2019) if i not in df.year.values]
```

[70]:
```
[1933]
```

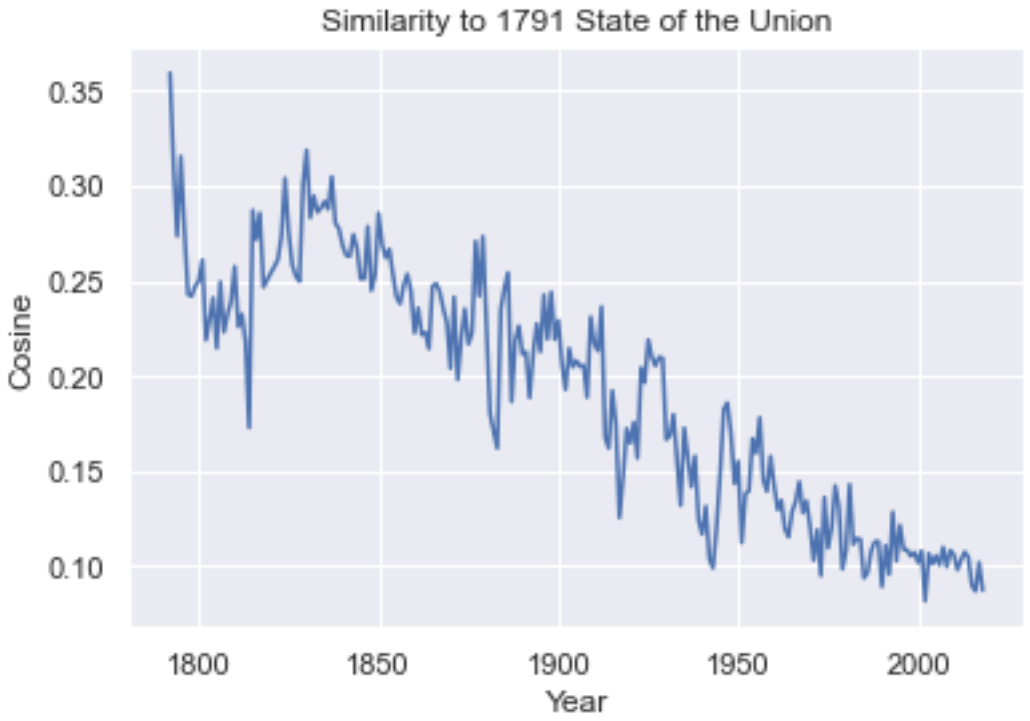
[71]:
```
tmp = copy.copy(df)
tmp.loc[len(tmp.index)] = ["Roosevelt2", 1933, np.nan, "Democrat", np.nan, np.
↪nan]
```

```
tmp = tmp.sort_values("year")
tmp = tmp.reset_index()
tmp.sim_to_first = tmp.sim_to_first.interpolate()

tmp = tmp[tmp.year.isin(range(1930,1940))]
display(tmp.head())

sns.lineplot(x="year", y="sim_to_first", data=tmp[tmp.index != 0])
plt.title("Similarity to 1791 State of the Union")
plt.xlabel("Year")
plt.ylabel("Cosine")
plt.show()
```

```
     index   president  year  \
139    139      Hoover  1930
140    140      Hoover  1931
141    141      Hoover  1932
142    227  Roosevelt2  1933
143    142  Roosevelt2  1934

                                                  text       party  \
139  To the Senate and House of Representatives:\n\...  Republican
140  To the Senate and House of Representatives:\n\...  Republican
141  To the Senate and House of Representatives:\n\...  Republican
142                                                NaN    Democrat
143  Mr. President, Mr. Speaker, Senators and Repre...    Democrat

                                        preprocessed  sim_to_first
139  senate house representatives honor comply requ...      0.166614
140  senate house representatives duty constitution...      0.168713
141  senate house representatives accord constituti...      0.179774
142                                                NaN      0.155896
143  mr president mr speaker senators representativ...      0.132017
```



[72]:
```
sns.lineplot(x="year", y="sim_to_first", data=df[df.index != 0])
plt.title("Similarity to 1791 State of the Union")
plt.xlabel("Year")
plt.ylabel("Cosine")
plt.show()
```



1.8.2 Example 2: Similarity to a President

Now let's try a less obvious starting point: the "average" speech for a particular president. I've put "average" in scare quotes because the idea of just averaging these representations may seem a little sketchy, but we are, in fact, going to average them. You can think of the average as the centroid (center) of the cluster of points belonging to a particular president's speeches in this vector space.

[73]:
```
df[df.president=="Polk"]
```

[73]:

	president	year	text	party	preprocessed	sim_to_first
54	Polk	1845	Fellow-Citizens of the Senate and of the House…	Democrat	fellow citizens senate house representatives s…	0.251119
55	Polk	1846	Fellow-Citizens of the Senate and of the House…	Democrat	fellow citizens senate house representatives r…	0.251108
56	Polk	1847	Fellow-Citizens of the Senate and of the House…	Democrat	fellow citizens senate house representatives a…	0.278595
57	Polk	1848	Fellow-Citizens of the Senate and of the House…	Democrat	fellow citizens senate house representatives b…	0.245021

James K. Polk's speeches have the indices 54, 55, 56, and 57.

We could have also used the following line:

[74]:
```
df[df.president=="Polk"].index
```

[74]:
```
Int64Index([54, 55, 56, 57], dtype='int64')
```

[75]:
```
polk_tfidf = tfidf_df[tfidf_df.index.isin([54, 55, 56, 57])]
```

[76]:
```
polk_tfidf
```

[76]:

	aaron	abandon	abandonment	abate	abatement	abating	abdicate	\
54	0.0	0.018011	0.018273	0.0	0.0	0.0	0.0	
55	0.0	0.027580	0.016970	0.0	0.0	0.0	0.0	
56	0.0	0.029731	0.000000	0.0	0.0	0.0	0.0	
57	0.0	0.015318	0.000000	0.0	0.0	0.0	0.0	

	abdication	abet	abettor	…	zeal	zealand	zealous	zealously	zelaya	\
54	0.0	0.0	0.0	…	0.0	0.0	0.0	0.0	0.0	
55	0.0	0.0	0.0	…	0.0	0.0	0.0	0.0	0.0	
56	0.0	0.0	0.0	…	0.0	0.0	0.0	0.0	0.0	
57	0.0	0.0	0.0	…	0.0	0.0	0.0	0.0	0.0	

	zero	zimbabwe	zinc	zone	zuloaga
54	0.0	0.0	0.0	0.0	0.0
55	0.0	0.0	0.0	0.0	0.0
56	0.0	0.0	0.0	0.0	0.0
57	0.0	0.0	0.0	0.0	0.0

[4 rows x 11503 columns]

Now we'll average them, and it's *really* important to keep checking the shape.

[77]:
```
polk_average = np.mean(polk_tfidf)
polk_average.shape
```

[77]:
```
(11503,)
```

[78]:
```
polk_average = np.array(polk_average).reshape(1, -1)
polk_average.shape
```

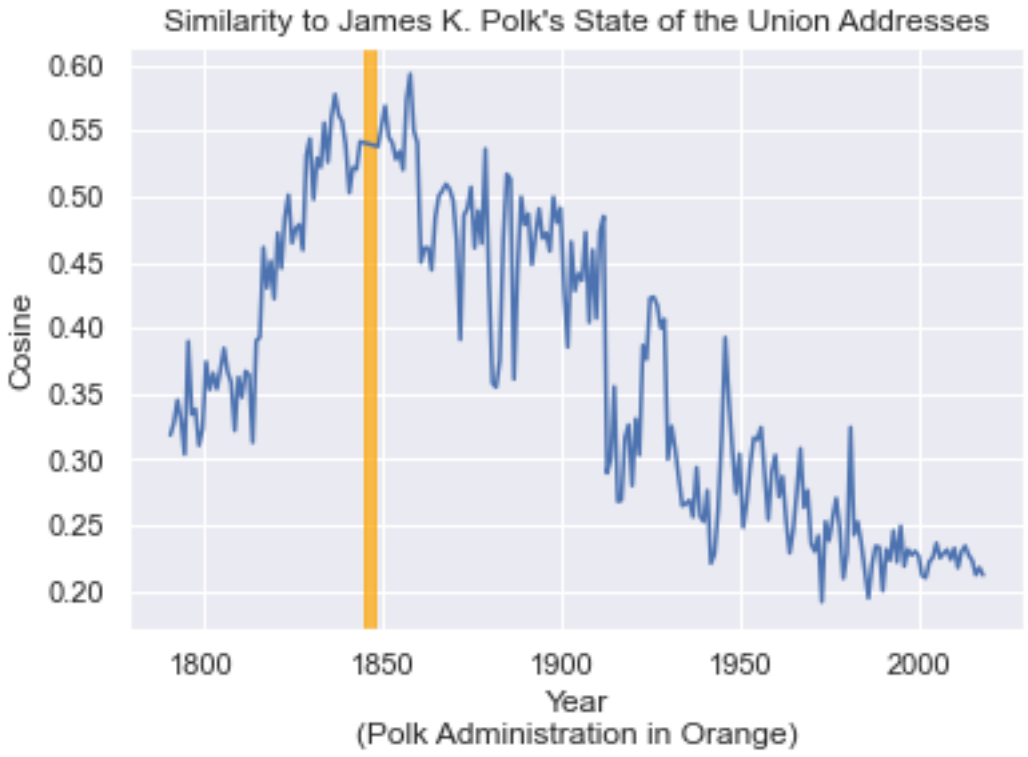
[78]:
```
(1, 11503)
```

Finally, we'll create a variable sim_to_polk in the same manner as the previous example.

[79]:
```
df["sim_to_polk"] = [cosine_similarity(polk_average, np.array(post).
 ↪reshape(1,-1))[0][0] for idx, post in tfidf_df.iterrows()]
```

[80]:
```
sns.lineplot(x="year", y="sim_to_polk", data=df[df.president != "Polk"])
plt.title("Similarity to James K. Polk's State of the Union Addresses")
plt.xlabel("Year\n(Polk Administration in Orange)")
plt.ylabel("Cosine")
```

```
plt.axvspan(1845, 1848, alpha = 0.7, color = "orange")
plt.show()
```



[81]:
```
df.sort_values("sim_to_polk", ascending=False) # most similar
```

[81]:

	president	year	text	\
56	Polk	1847	Fellow-Citizens of the Senate and of the House…	
55	Polk	1846	Fellow-Citizens of the Senate and of the House…	
54	Polk	1845	Fellow-Citizens of the Senate and of the House…	
57	Polk	1848	Fellow-Citizens of the Senate and of the House…	
67	Buchanan	1858	Fellow-Citizens of the Senate and House of Rep…	
..	…	…	…	
210	Bush2	2002	Thank you very much. Mr. Speaker, Vice Preside…	
187	Carter	1979	Tonight I want to examine in a broad sense the…	
198	Bush	1990	Tonight, I come not to speak about the "State …	
194	Reagan	1986	Mr. Speaker, Mr. President, distinguished Memb…	
181	Nixon	1973	To the Congress of the United States:\n\nThe t…	

	party	preprocessed	\
56	Democrat	fellow citizens senate house representatives a…	
55	Democrat	fellow citizens senate house representatives r…	
54	Democrat	fellow citizens senate house representatives s…	

```
57     Democrat  fellow citizens senate house representatives b…
67     Democrat  fellow citizens senate house representatives c…
..          …                                                 …
210  Republican  thank mr speaker vice president cheney members…
187    Democrat  tonight want examine broad sense state america…
198  Republican  tonight come speak state government detail new…
194  Republican  mr speaker mr president distinguished member c…
181  Republican  congress united states traditional form presid…

     sim_to_first  sim_to_polk
56       0.278595     0.817134
55       0.251108     0.816113
54       0.251119     0.800426
57       0.245021     0.786088
67       0.253581     0.593389
..            …            …
210      0.081755     0.210069
187      0.098512     0.209558
198      0.089294     0.200072
194      0.096690     0.194554
181      0.095044     0.191706

[227 rows x 7 columns]
```

[82]:
```
df[df.president != "Polk"].sort_values("sim_to_polk", ascending=False).head(10)␣
↪# most similar, excluding Polk himself
```

[82]:
```
   president  year                                               text  \
67  Buchanan  1858  Fellow-Citizens of the Senate and House of Rep…
46     Buren  1837  Fellow-Citizens of the Senate and House of Rep…
66  Buchanan  1857  Fellow-Citizens of the Senate and House of Rep…
60  Fillmore  1851  Fellow-Citizens of the Senate and of the House…
47     Buren  1838  Fellow-Citizens of the Senate and House of Rep…
45   Jackson  1836  Fellow Citizens of the Senate and of the House…
48     Buren  1839  Fellow-Citizens of the Senate and House of Rep…
43   Jackson  1834  Fellow Citizens of the Senate and of the House…
59  Fillmore  1850  Fellow-Citizens of the Senate and of the House…
68  Buchanan  1859  Fellow-Citizens of the Senate and House of Rep…

       party                                       preprocessed  sim_to_first  \
67  Democrat  fellow citizens senate house representatives c…      0.253581
46  Democrat  fellow citizens senate house representatives r…      0.304796
66  Democrat  fellow citizens senate house representatives o…      0.247766
60      Whig  fellow citizens senate house representatives c…      0.269272
47  Democrat  fellow citizens senate house representatives c…      0.280311
45  Democrat  fellow citizens senate house representatives a…      0.287691
48  Democrat  fellow citizens senate house representatives r…      0.277358
```

```
43  Democrat  fellow citizens senate house representatives p…      0.288335
59      Whig  fellow citizens senate house representatives s…      0.285617
68  Democrat  fellow citizens senate house representatives d…      0.244563

    sim_to_polk
67     0.593389
46     0.577841
66     0.574041
60     0.569029
47     0.562332
45     0.560665
48     0.557278
43     0.556208
59     0.554304
68     0.550442
```

1.8.3 Example 3: Similarity to the Whigs

We also know the party of the president who gave the speech (although this variable may not be meaningful for the first several presidents). We can calculate the centroid for a party and then calculate the similarity of every subsequent speech to that.

[83]:
```
df[df.party=="Whig"].index
```

[83]:
```
Int64Index([50, 51, 52, 53, 58, 59, 60, 61], dtype='int64')
```

[84]:
```
whig_indices = df[df.party=="Whig"].index
```

[85]:
```
whig_tfidf = tfidf_df[tfidf_df.index.isin(whig_indices)]
whig_tfidf
```

[85]:
```
    aaron   abandon  abandonment     abate  abatement  abating  abdicate  \
50    0.0  0.029733          0.0  0.000000   0.063384      0.0        0.0
51    0.0  0.000000          0.0  0.000000   0.000000      0.0        0.0
52    0.0  0.014576          0.0  0.000000   0.000000      0.0        0.0
53    0.0  0.000000          0.0  0.031865   0.000000      0.0        0.0
58    0.0  0.025215          0.0  0.000000   0.000000      0.0        0.0
59    0.0  0.000000          0.0  0.000000   0.000000      0.0        0.0
60    0.0  0.000000          0.0  0.000000   0.000000      0.0        0.0
61    0.0  0.012568          0.0  0.000000   0.000000      0.0        0.0

    abdication  abet  abettor  …      zeal  zealand  zealous  zealously  \
50         0.0   0.0      0.0  …  0.018954      0.0      0.0   0.000000
51         0.0   0.0      0.0  …  0.000000      0.0      0.0   0.033548
52         0.0   0.0      0.0  …  0.019500      0.0      0.0   0.000000
53         0.0   0.0      0.0  …  0.030268      0.0      0.0   0.000000
58         0.0   0.0      0.0  …  0.019923      0.0      0.0   0.000000
59         0.0   0.0      0.0  …  0.000000      0.0      0.0   0.000000
```

```
60         0.0   0.0      0.0  …  0.000000      0.0      0.0   0.000000
61         0.0   0.0      0.0  …  0.000000      0.0      0.0   0.000000

    zelaya  zero  zimbabwe  zinc  zone  zuloaga
50     0.0   0.0       0.0   0.0   0.0      0.0
51     0.0   0.0       0.0   0.0   0.0      0.0
52     0.0   0.0       0.0   0.0   0.0      0.0
53     0.0   0.0       0.0   0.0   0.0      0.0
58     0.0   0.0       0.0   0.0   0.0      0.0
59     0.0   0.0       0.0   0.0   0.0      0.0
60     0.0   0.0       0.0   0.0   0.0      0.0
61     0.0   0.0       0.0   0.0   0.0      0.0

[8 rows x 11503 columns]
```

[86]:
```
whig_average = np.mean(whig_tfidf)
whig_average.shape
```

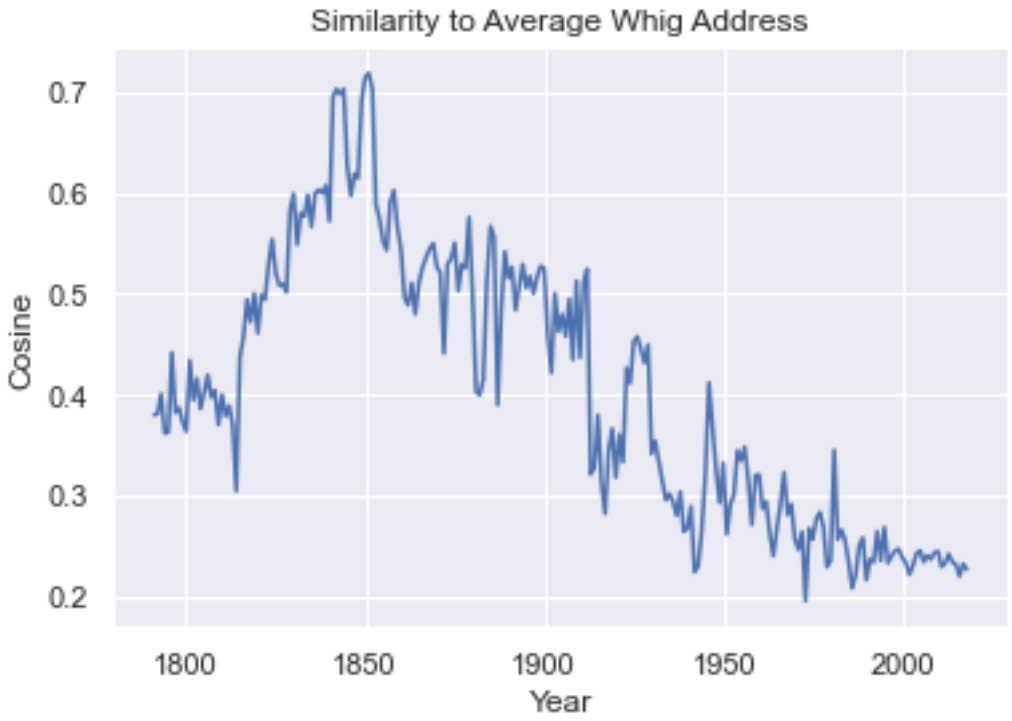
[86]:
```
(11503,)
```

[87]:
```
whig_average = np.array(whig_average).reshape(1, -1)
whig_average.shape
```

[87]:
```
(1, 11503)
```

[88]:
```
df["sim_to_whig"] = [cosine_similarity(whig_average, np.array(post).
 ↪reshape(1,-1))[0][0] for idx, post in tfidf_df.iterrows()]
```

[89]:
```
sns.lineplot(x="year", y="sim_to_whig", data=df)
plt.title("Similarity to Average Whig Address")
plt.xlabel("Year")
plt.ylabel("Cosine")
plt.show()
```



1.9 Exercises

Exercise 1

For this exercise, pick a year when you think the State of the Union Address may have referred
1.1 What sociologically significant events, institutions, or processes might make this year dis

Your text here

1.2 Identify the index

[91]:
```
df[df.year == PICK_A_YEAR].index # replace "PICK_A_YEAR" with a year
```

1.3 Get the tf-idf-weighted vector from the dataframe <tt>tfidf_df</tt>

[ ]:
```
chosen_year_vec = tfidf_df.loc[YOUR_INDEX] # replace "YOUR_INDEX" with the␣
↪index from the previous step
chosen_year_vec = np.array(chosen_year_vec).reshape(1,-1)
chosen_year_vec.shape
```

1.4 Run the cell below to calculate the cosine similarity of each speech to the speech from you

[ ]:
```
df["sim_to_chosen_year"] = [cosine_similarity(chosen_year_vec, np.array(post).
↪reshape(1,-1))[0][0] for idx, post in tfidf_df.iterrows()]
```

1.5 What trend do you expect in how similar other speeches are? For example, will earlier or la

Your answer here

1.6 Plot the trend as in Example 1 above

[ ]:
```
# YOUR CODE HERE
```

1.7 What do you notice about the trend? Does the plot reflect your expectations? What might exp

Your answer here

Exercise 2

Now, as in Example 2, you will examine trends in the similarity of State of the Union addresses
2.1 Pick a president other than James K. Polk. What sociologically significant events happened

Your answer here

2.2 Get the index or indices of that president's speeches. Save them to the variable <tt>pres_

[ ]:
```
pres_indices = df[df.president=="PRESIDENT"].index # replace "PRESIDENT" with␣
↪the name as it appears in the data
df.loc[pres_indices]
```

2.3 Get the subset of tf-idf-weighted vectors corresponding to those indices and save the resul

[ ]:
```
pres_tfidf = tfidf_df[tfidf_df.index.isin(pres_indices)]
pres_tfidf
```

2.4 Now calculate the centroid (average) of the vectors and display the shape using the <tt>.sh

[ ]:
```
pres_average = np.mean(pres_tfidf)
pres_average.shape
```

2.5 Reshape the vector (as in the examples above) using the <tt>.reshape</tt> method. The first

[ ]:
```
pres_average = np.array(pres_average).reshape(1, -1)
pres_average.shape
```

2.5 Now calculate the similarity of each speech to that average, just as we have done in the ex

[ ]:
```
# YOUR CODE HERE
```

2.6 What trends do you expect in the similarity of other documents to this average?

Your answer here

2.7 Plot the trend.

[ ]:
```
# YOUR CODE HERE
```

2.8 Does the plot match your expectations? What might explain any differences?

Your answer here

Exercise 3

Now pick another group of speeches that were given in years during which sociologically signifi
3.1 What group of years are you choosing, and what makes that period of time interesting?

Your answer here

3.2 What do you expect the plotted trends in similarity to look like? Why?

Your answer here

3.3 Following the steps in the previous exercise, calculate the centroid of the appropriate ve

[ ]:
```
# YOUR CODE HERE
```

3.4 Plot the trend.

[ ]:
```
# YOUR CODE HERE
```

3.5 Does the plot match your expectations? What might explain any differences? If you've offer

Your answer here