soc128d_notebook_2_manipulating_quantifying_visualizing_text_data

June 28, 2021

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

1 Notebook 2: Manipulating, Quantifying, and Visualizating Text Data

A quick note about this notebook versus subsequent notebooks As a reminder, this class is meant to be a bit like a series of workshops, many of which could be self-contained. If you feel a bit lost amid all the code, that's okay–and if this all strikes you as incredibly basic, that's okay, too! Subsequent notebooks will use various methods to analyze a corpus in a way that can produce sociologically-interesting knowledge. These notebooks will be accessible even if you don't feel confident in your ability to follow the code because the focus won't be on the coding. The notebooks should also be interesting even if the code seems pretty basic to you because we'll be looking at applications to social research.

A quick note about this notebook In this notebook, we'll begin to look at how to use libraries like `pandas` to manage and manipulate text. We'll also build toward different ways of quantifying information about documents and visualizing the results.

This is only the second notebook, and this is all preliminary material. From here, we'll move on to measures like TF-IDF that can be used for information retrieval (e.g., search engines) and measuring document similarity, which has all sorts of applications in the social sciences. TF-IDF is also a jumping off point for other measures we'll consider, including dense vector representations.

Let's have an initial go at using `pandas` with a dataset of tweets from January 6, 2021. You can download the dataset from Kaggle at this link. Just unzip/extract the archive and place the CSV in the same directory as this notebook (or edit the `f = "..."` line in the following cell to include the file path).

[ ]:
```
import pandas as pd

f = "tweets_2021-01-06.csv"
df = pd.read_csv(f)
```

`.head()` shows the first few rows. You can supply a number to see more or fewer.

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

`.columns` lists the column names.

```
[ ]: df.columns
```

`.shape` shows the "shape" of the dataframe, meaning the number of rows followed by the number of columns. The following line of code shows there are 82,309 tweets (rows) and 14 columns.

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

We can subset the dataset by column by supplying a list of the columns we want.

```
[ ]: cols_to_keep = ["tweet_id", "text", "query", "follower_count", "likes",␣
      ↪"retweets"]
```

```
[ ]: df = df[cols_to_keep]
```

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

We can also subset the **rows** using criteria like specific values for specific columns.

```
[ ]: df[df["tweet_id"]==1346863072435179520]
```

```
[ ]: df[df["likes"] > 100000]
```

```
[ ]: df[df["query"]=="mask"]
```

We can also combine criteria to select rows.

```
[ ]: df[(df["query"]=="mask") & (df["retweets"]==0)]
```

`pandas` has a lot of built-in functionality for manipulating, summarizing, and plotting data. Let's look at how we can calculate and plot the counts and means of different variables.

```
[ ]: df[["query", "tweet_id"]].groupby("query").count()
```

```
[ ]: df[["query", "tweet_id"]].groupby("query").count().plot.bar()
```

```
[ ]: df[["query", "retweets"]].groupby("query").mean()
```

```
[ ]: df[["query", "retweets"]].groupby("query").mean().plot.bar()
```

Although we can use methods like `str.contains()` or even `if` statements like `if "str" in string: ...` when we're operating on strings, like below, sometimes we need to find other ways to interact with strings in pandas dataframes.

In the following cell, there should be a `TypeError` when the we check whether the string "capitol" is in the "text" column of a row that apparently has a float instead. This is actually a missing value.

```
[ ]: COUNT = 0

     for idx, row in df.iterrows():
```

```
    if "capitol" in row["text"]:
        COUNT += 1

print(COUNT)
```

[ ]:
```
COUNT = 0

for idx, row in df.iterrows():
    try:
        if "capitol" in row["text"]:
            COUNT += 1
    except TypeError:
        print(f"Problematic type and value: {type(row['text'])}, {row['text']}")

print(COUNT)
```

We can drop the row with the missing value. The `.dropna()` method can be applied to the entire row, but we may not care if data are missing for a variable that is not important to our analysis. We can use the `subset` argument to specify the columns we want `.dropna()` to look at. The following line means that we will drop any rows with missing data for the "text" column. We lose only one row this way.

[ ]:
```
df = df.dropna(subset = ["text"])
```

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

Now we can count the number of tweets containing the string "capitol" at least once.

[ ]:
```
COUNT = 0

for idx, row in df.iterrows():
    if "capitol" in row["text"]:
        COUNT += 1

print(COUNT)
```

We'll need to take a quick detour to think about lambda functions, which are functions that we define in just one line of code. As we'll see in a moment, these are useful for operating on columns in pandas dataframes. Let's also look at list comprehensions, which allow us to execute a loop in just one line of code.

The for loop below appends the values 0-9 to a list called `integers`. In the next cell, we use a list comprehension to do the same thing, and both print statements match. List comprehensions can be faster and actually quite flexible, but they can also end up being difficult to read.

[ ]:
```
integers = []

for i in range(10):
    integers.append(i)
```

```
print(integers)
```

[ ]:
```
integers = [i for i in range(10)]

print(integers)
```

In the next two cells, we see a function defined in a relatively conventional way. This function takes in the variable `x`, which should be an integer, and multiplies it by 10, returning another integer.

The second cell defines an equivalent function in only one line. `lambda x:` means we're about to do something to `x`, for any value of `x` that is supplied. The part after the colon is what we do to `x`, which in this case is multiplying it by 10.

[ ]:
```
def multiply_by_ten(x: int) -> int:
    return x * 10

integers = [i for i in range(10)]
integers = [multiply_by_ten(x) for x in integers]

print(integers)
```

[ ]:
```
integers = [i for i in range(10)]

print(f"Original list: {integers}")

multiply_by_ten = lambda x: x*10

integers = [multiply_by_ten(x) for x in integers]

print(f"After applying lambda function: {integers}")
```

We'll use those in a moment. Next, let's look at how we can iterate through a pandas column. We'll look at the "text" column as a list, then use a for loop to calculate the wordcounts. Then we'll define a function in the traditional way and apply it to the "text" column in the pandas dataframe directly using the `.apply()` method. Then we'll do the same thing using a lambda function instead of a function we've defined the conventional way. Finally, we'll show that these provide the same results.

[ ]:
```
df["text"].tolist()
```

[ ]:
```
tweets = []

for tweet in df["text"].tolist():
    tweets.append(tweet)

for i in range(5):
    print(tweets[i])
```

```
[ ]: for i in range(5):
         print(tweets[i].split())
```

```
[ ]: wordcounts = []

     for tweet in df["text"].tolist():
         wordcount = len(tweet.split())
         wordcounts.append(wordcount)
```

pyplot and seaborn will be staples of data visualization in the course.

```
[ ]: import matplotlib.pyplot as plt
     import seaborn as sns

     sns.set_palette("flare")

     sns.kdeplot(wordcounts)
     plt.show()
```

```
[ ]: def wordcount(word: str) -> int:
         return len(word.split())


     df["wordcount"] = df["text"].apply(wordcount)

     sns.kdeplot("wordcount", data = df)
     plt.show()
```

```
[ ]: df["wordcount2"] = df["text"].apply(lambda x: len(x.split()))

     sns.kdeplot("wordcount2", data = df)
     plt.show()
```

Finally, we can superimpose these distributions to show that they are the same (though we can do this other ways). We should just see one distribution in this case because they are being plotted over one another. If these were different variables, we would see multiple distributions in this plot, as we'll see below.

```
[ ]: sns.kdeplot("wordcount", data = df)
     sns.kdeplot("wordcount2", data = df)
     sns.kdeplot(wordcounts)
     plt.show()
```

The `collections` module provides a lot useful methods for manipulating data. We will use `defaultdict()` a lot later on, for example. Right now, let's take a look at `Counter()` . Counter provides an object like a dictionary with the elements of an iterable (like a list) as keys and the frequencies as values. In the example below, we get a mapping of values from the "query" column (keys) to their frequencies in the dataframe (values), or the number of rows with those values.

```
[ ]: from collections import Counter

     c = Counter(df["query"])
     print(type(c))
     print(c.keys())
```

```
[ ]: c
```

Let's take a look at tweets from a subset of these queries. We'll subset the dataframe a bit like we did above, this time using the `.isin()` method.

The code in the second cell, `df[df["query"].isin(query_subset)]`, will return a dataframe matching that criterion. What's going on with this way of subsetting the data, though?

As we can see in the third cell, if we execute just the code used to subset the data, `df["query"].isin(query_subset)`, we get a pandas Series (like a list) of `True` and `False` of the same length as the number of rows in the dataframe. When we apply that condition to the dataframe to subset it in the second cell (`df[df["query"].isin(query_subset)]`), we are return the rows where that condition is `True`.

```
[ ]: query_subset = ["lockdown", "mask", "quarantine", "travel"]
```

```
[ ]: df[df["query"].isin(query_subset)]
```

```
[ ]: df["query"].isin(query_subset)
```

```
[ ]: type(df["query"].isin(query_subset))
```

```
[ ]: sns.kdeplot("wordcount", data = df[df["query"].isin(query_subset)], hue =␣
      ↪"query")
     plt.xlabel("Wordcount")
     plt.title("Distribution of Wordcounts in Tweets by Query")
     plt.show()
```

```
[ ]: sns.set_palette("Set2")

     sns.kdeplot("wordcount", data = df[df["query"].isin(query_subset)], hue =␣
      ↪"query")
     plt.xlabel("Wordcount")
     plt.title("Distribution of Wordcounts in Tweets by Query")
     plt.show()
```

```
[ ]: sns.scatterplot(x = "wordcount", y = "retweets", data = df)
     plt.show()
```

Now let's subset the data so we can look at tweets containing the string "capitol" at least once. We'll define a function, `contains_capitol`, that takes in a string and returns `True` if the string contains "capitol" and `False` if it doesn't.

```
[ ]: def contains_capitol(text: str) -> bool:
         if "capitol" in text:
             return True
         return False

     test_string1 = "this string contains capitol"
     test_string2 = "this string does not"

     print(contains_capitol(test_string1))
     print(contains_capitol(test_string2))
```

Now we'll create a variable using this function.

```
[ ]: df["contains_capitol"] = df["text"].apply(contains_capitol)
```

```
[ ]: df
```

```
[ ]: Counter(df["contains_capitol"])
```

Finally, let's subset the rows to keep only the tweets containing "capitol" (in the next cell) and then drop our new column and the duplicate wordcount column.

```
[ ]: df = df[df["contains_capitol"]] # we don't need to specify "== True" because␣
      ↪Python assumes this
```

```
[ ]: df = df.drop(["wordcount2", "contains_capitol"], axis = 1)
```

```
[ ]: df
```

Now let's take a quick look at some properties of the wordcounts of this subset of tweets. We can control the precision when we print floats, including when we use f-strings, like below. `.2f` restricts to the output to two digits after the decimal.

```
[ ]: print(f"Average wordcount: {df['wordcount'].mean()}")
```

```
[ ]: print(f"Average wordcount: {df['wordcount'].mean():.2f}")
     print(f"Minimum wordcount: {df['wordcount'].min()}")
     print(f"Maximum wordcount: {df['wordcount'].max()}")
     print(f"Standard deviation: {df['wordcount'].std()}")
```

A quick note on calculating variances and standard deviations in Python and other languages:

The `std()` method from pandas divides by N - 1. This is similar to the default for R and Stata.

The `std()` method from numpy divides by N. You can change this by using the `ddof` argument in either method (setting it to 0 or 1).

If you're curious, see here and here for more on why we often divide by N - 1 for the sample variance (and thus standard deviation), rather than by N (as we do for the population).

```
[ ]: integers_as_strings = [str(i) for i in range(10)]
     print(f"As a list: {integers_as_strings}")

     integers_as_strings = " ".join(integers_as_strings)
     print(f"As a single string: {integers_as_strings}")
```

Now let's take a look at term frequency and document frequency. We'll use these to build up to powerful ways to compare documents. One application is finding similar documents (e.g., in search engines). We'll see later in the quarter how these can also be used to compare things like online communities like subreddits.

To get the frequencies of all the words in the corpus, we can quickly combine all of the tweets (the "text" column) using the `.join()` method, implicitly treating the column as a list of tweets. If we look at a slice of the document (by characters), we can see that we've just merged all of the tweets together, joining them with a space.

```
[ ]: all_text = " ".join(df["text"])
```

```
[ ]: all_text[:1000]
```

If we `split` all of this on whitespace, we get the individual *tokens*, or instances of words. If we use `Counter`, this returns the number of times each `type` (*unique* token) occurs. See here for the type-token distinction. Each word is a token, but it is also an instance of a unique type.

```
[ ]: word_frequencies = dict(Counter(all_text.split()))

     types_and_counts = sorted(list(word_frequencies.items()), reverse = True, key =␣
      ↪lambda x: x[1])
     print(types_and_counts[:100])
```

Now let's examine the distribution of word frequencies. The code below "unzips" the types and frequencies into lists that stay in the same order. `types_` will be a list of the types in the original order, and `token_counts` will be a list of the frequencies in the same order. The first element in `types_` will be the most frequent, while the last will be the least frequent.

```
[ ]: types_, token_counts = zip(*types_and_counts)
```

```
[ ]: plt.bar(x = range(100), height = token_counts[:100])
     plt.show()
```

```
[ ]: plt.bar(x = types_[:20], height = token_counts[:20])
     plt.xticks(rotation = 90)
     plt.show()
```

Linguists will sometimes refer to Zipf's law when discussing the distribution of words. You'll typically see that a few words are extremely common and most are extremely rare. Put one way, there is a negative correlation between the rank of a word (where 1 = most frequent) and its frequency.

We'll talk about "stop words" more soon; these are extremely frequent words which are generally seen to have little semantic information (e.g., “the” or “to”). Looking at the distribution of words will also be relevant later when we start to look at tools like topic modeling, where some algorithms make assumptions about how words are distributed.

```
[ ]: import numpy as np

     log_rank = np.log(range(1, len(token_counts)+1))
     log_frequencies = np.log(token_counts)

     plt.plot(log_rank, log_frequencies)
     plt.ylabel("ln(word frequency)")
     plt.xlabel("ln(word rank)")
     plt.title("Word Rank versus Frequency (log-log)")
     plt.show()
```

```
[ ]: print(types_[:100])
```

```
[ ]: print(types_[-100:])
```

```
[ ]: print(len(types_))
```

```
[ ]: print(sum(token_counts))
```

Now we have a dictionary, `word_frequencies`, that maps each type (unique word) to its frequency in the corpus. Let’s make a dictionary mapping types to their *document frequency*, or the number of documents in which they occur.

```
[ ]: def set_of_types(document: str) -> str:
         return " ".join(list(set(document.split())))
```

```
[ ]: s = "this is a string that repeats some words, like string and words and some"

     Counter(s.split()) # three types occur twice
```

```
[ ]: s2 = set_of_types(s)

     Counter(s2.split()) # each type occurs only once
```

If we applly this function to each document, each row will have a string containing the types in the original tweet, but each will appear only once. If we count up the occurrences of each word in this column, it’s the same as going through each row and checking whether the type occurs at least once. In other words, this helps us count the number of documents in which each word occurs. There are other ways to do this, of course, and later we’ll see that we can use libraries like `scikit-learn` to do this more quickly.

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

```
[ ]: df
[ ]: document_frequencies = dict(Counter(" ".join(df["types"]).split()))
```

```
[ ]: document_frequencies["capitol"]
```

Why does the type "capitol" occur in only 1,736 documents, even though there are 1,750 documents we subset based on whether they contained "capitol" as part of the overall string? The answer is that we didn't look for the *type* "capitol" but instead for whether the string "capitol" occurred in the overall tweet. If we iterate through the words in each tweet, we can find out which strings *contain* "capitol" but aren't themselves instances of the *type* "capitol."

```
[ ]: s = set()

     for tweet in df["types"].tolist():
         for word in tweet.split():
             if "capitol" in word:
                 s.add(word)

     print(s)
```

```
[ ]: document_frequencies_list = sorted(list(document_frequencies.items()), reverse␣
      ↪= True, key = lambda x: x[1])
```

```
[ ]: print(document_frequencies_list[:100])
```

```
[ ]: print(document_frequencies_list[-100:])
```

At the corpus level, word frequency and document frequency are highly correlated. However, we will also look at the frequency of words *within* a document, which can help us identify similar documents. Documents that use the same words at approximately the same rates are similar in that important way. However, not all words are equally informative. As we saw above, some words are extremely common.

If two sentence uses the word "the" many times, that doesn't tell us a lot. If the two documents use the word "insurgents" many times, relative to the typical document, then they likely share information we are interested in. This is a step toward saying the documents are "about" the same thing–if not quite saying that they "mean" something similar. In practice, if we are comparing documents based on the frequencies of the words they use, we will normalize the word counts in some way by the document frequency (i.e., how many documents a type appears in).

```
[ ]: from scipy.stats import pearsonr, spearmanr

     vocabulary = sorted(list(word_frequencies.keys()))

     x = [word_frequencies[word] for word in vocabulary]
     y = [document_frequencies[word] for word in vocabulary]

     print("Correlation between each word's frequency in the overall corpus and its␣
      ↪document frequency:")
     print(f"Pearson's correlation coefficient: {pearsonr(x, y)[0]:.2f}")
print(f"Spearman's rank-order correlation: {spearmanr(x, y)[0]:.2f}")
```

[ ]:
```
len(vocabulary)
```

If we are interested in analyzing meaning from a corpus, in practice we will often remove words that appear only once or in only one document (which aren't the same thing!). We sometimes call these hapaxes. We can't say that two documents have a word in common if only one document in the entire corpus has the word!

[ ]:
```
words_to_keep = [word for word in vocabulary if document_frequencies[word] > 1]
print(len(words_to_keep))
```

We may often exclude words that appear in *every* document for similar reasons.

Let's remove hapaxes from our dictionaries of word and document frequencies.

[ ]:
```
word_frequencies = {key:value for key, value in word_frequencies.items() if key␣
 ↪in words_to_keep}
document_frequencies = {key:value for key, value in document_frequencies.
 ↪items() if key in words_to_keep}

vocabulary = words_to_keep
```

[ ]:
```
len(word_frequencies.keys()) == len(document_frequencies.keys()) ==␣
 ↪len(vocabulary)
```

[ ]:
```
print(len(vocabulary))
```

That's it for Notebook 2! Now we're ready to talk about TF-IDF and other ways of comparing documents, which will set us up for a shift to tasks like comparing (or unmasking!) authors, comparing the meaning of words, identifying latent themes in documents, and using these kinds of features–information we have mined by quantifying properites of the text–to answer all manners of social research questions. These tools and skills will also transfer to other areas, so don't worry if studying culture isn't 100% your bag!