learn-co-curriculum / dsc-word-vectorization-lab Public

View license

1 star 180 forks

Star Notifications

Code Issues Pull requests Actions Projects Security Insights

solution Go to file

This branch is 10 commits ahead, 12 commits behind master.

lindseyberlin Update lab ... on Sep 1, 2022 11

View code

README.md

Word Vectorization - Lab

Introduction

In this lab, you'll tokenize and vectorize text documents, create and use a bag of words, and identify words unique to individual documents using TF-IDF vectorization.

Objectives

In this lab you will:

- Implement tokenization and count vectorization from scratch
- Implement TF-IDF from scratch
- Use dimensionality reduction on vectorized text data to create and interpret visualizations

Let's get started!

Run the cell below to import everything necessary for this lab.

```
import pandas as pd
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.manifold import TSNE
from nltk.tokenize import word_tokenize
import nltk
nltk.download('punkt', quiet=True)
np.random.seed(0)
```

Our Corpus

In this lab, we'll be working with 20 different documents, each containing song lyrics from either Garth Brooks or Kendrick Lamar albums.

The songs are contained within the `data` subdirectory, contained within the same folder as this lab. Each song is stored in a single file, with files ranging from `song1.txt` to `song20.txt`.

To make it easy to read in all of the documents, use a list comprehension to create a list containing the name of every single song file in the cell below.

```
filenames = [f'song{str(i)}.txt' for i in range(1,21)]
filenames
```

```
['song1.txt',
 'song2.txt',
 'song3.txt',
 'song4.txt',
 'song5.txt',
 'song6.txt',
 'song7.txt',
 'song8.txt',
 'song9.txt',
 'song10.txt',
 'song11.txt',
 'song12.txt',
 'song13.txt',
 'song14.txt',
 'song15.txt',
 'song16.txt',
 'song17.txt',
 'song18.txt',
 'song19.txt',
 'song20.txt']
```

Next, let's import a single song to see what our text looks like so that we can make sure we clean and tokenize it correctly.

Use the code in the cell below to read in the lyrics from `song18.txt` as a list of lines, just using vanilla Python:

```
# Import and print song18.txt
with open('data/song18.txt') as f:
    test_song = f.readlines()

test_song
```

```
['[Kendrick Lamar:]\n',
 "Two wrongs don't make us right away\n",
 "Tell me something's wrong\n",
 'Party all of our lives away\n',
 'To take you on\n',
 '[Zacari:]\n',
 'Oh, baby I want you\n',
 'Baby I need you\n',
 'I wanna see you\n',
 'Baby I wanna go out yeah\n',
 'Baby I wanna go out yeah\n',
 'Baby I want you\n',
 'Baby I need you\n',
 'I wanna see you\n',
 'Baby I wanna go out yeah\n',
 'Baby I wanna go out yeah\n',
 'All night (all night, all night)\n',
 'All night\n',
 "Your body's on fire\n",
 'And your drinks on ice\n',
 'All night (all night, all night)\n',
 'All night\n',
 "Your body's on fire\n",
 'And your drinks on ice\n',
 '[Babes Wodumo:]\n',
 'Oh my word oh my gosh oh my word (Oh my gosh)\n',
        'Oh my word oh my gosh oh my word (Oh my gosh)\n',
        'Oh my word oh my gosh oh my word (Oh my gosh)\n',
        'Oh my word oh my gosh oh my word (Oh my gosh)\n',
        'Everybody say kikiritikiki (kikiritikiki)\n',
        'Everybody say kikiritikiki (kikiritikiki)\n',
        'Everybody say kikiritikiki (kikiritikiki)\n',
        'Everybody say kikiritikiki (kikiritikiki)\n',
        "Ung'bambe, ung'dedele. Ung'bhasobhe, ung'gudluke\n",
        "Ung'bambe, ung'dedele. Ung'bhasobhe, ung'gudluke\n",
        "Ung'bambe, ung'dedele. Ung'bhasobhe, ung'gudluke\n",
        "Ung'bambe, ung'dedele. Ung'bhasobhe, ung'gudluke\n",
        '[Zacari:]\n',
        'Baby I want you\n',
        'Baby I need you\n',
        'I wanna see you\n',
        'Baby I wanna go out yeah\n',
        'Baby I wanna go out yeah\n',
        'Baby I want you\n',
        'Baby I need you\n',
        'I wanna see you\n',
        'Baby I wanna go out yeah\n',
        'Baby I wanna go out yeah\n',
        'All night (all night all night)\n',
        'All night\n',
        "Your body's on fire\n",
        'And your drinks on ice\n',
        'All night (all night all night)\n',
        'All night\n',
        "Your body's on fire\n",
        'And your drinks on ice\n',
        '[Kendrick Lamar:]\n',
        '(We go)\n',
        'High up (High up)\n',
        'High up (High up)\n',
        'High up (High up)\n',
        'High up (High up)\n',
        'High up (High up)\n',
        'High up (High up)\n',
        'High up (High up)\n',
        'High up (High up)\n',
        '[?]\n',
        '[Zacari:]\n',
        'Baby I want you\n',
        'Baby I need you\n',
        'I wanna see you\n',
        'Baby I wanna go out yeah\n',
        'Baby I wanna go out yeah\n',
        'Baby I want you\n',
        'Baby I need you\n',
 'I wanna see you\n',
 'Baby I wanna go out yeah\n',
 'Baby I wanna go out yeah\n',
 'All night (all night all night)\n',
 'All night\n',
 "Your body's on fire\n",
 'And your drinks on ice\n',
 'All night (all night all night)\n',
 'All night\n',
 "Your body's on fire\n",
 'And your drinks on ice\n']
```

Tokenizing our Data

Before we can create a bag of words or vectorize each document, we need to clean it up and split each song into an array of individual words. Our goal is to tken

Consider the following sentences from the example above:

`"Two wrongs don't make us right away\n", "Tell me something's wrong\n"`

After tokenization, this should look like:

`['two', 'wrongs', 'dont', 'make', 'us', 'right', 'away', 'tell', 'me', 'somethings', 'wrong']`

Tokenization is pretty tedious if we handle it manually, and would probably make use of regular expressions, which is outside the scope of this lab. In order to keep this lab moving, we'll use a library function to clean and tokenize our data so that we can move onto vectorization.

Tokenization is a required task for just about any Natural Language Processing (NLP) task, so great industry-standard tools exist to tokenize things for us, so that we can spend our time on more important tasks without getting bogged down hunting every special symbol or punctuation in a massive dataset. For this lab, we'll make use of the tokenizer in the amazing `nltk` library, which is short for *Natural Language Tool Kit*.

NOTE: NLTK requires extra installation methods to be run the first time certain methods are used. If `nltk` throws you an error about needing to install additional packages, follow the instructions in the error message to install the dependencies, and then rerun the cell.

Before we tokenize, however, we need to do one more step! Computers are very particular about strings. If we tokenized our data in its current state, we would run into the following problems:

- Counting things that aren't actually words. In the example above, `"[Kendrick Lamar:]"` is a note specifying who is speaking, not a lyric contained in the actual song, so it should be removed.
- Punctuation and capitalization would mess up our word counts. To the Python interpreter, `all`, `All`, and `(all` are unique words, and would all be counted separately. We need to remove punctuation and capitalization, so that all words will be counted correctly.

Before we tokenize our songs, we'll do only a small manual bit of cleaning.

In the cell below, write a function to:

- remove lines that just contain `['artist names']`
- join the list of strings into one big string for the entire song
- remove newline characters `\n`
- remove the following punctuation marks: `",.'?!()"`
- make every word lowercase

Test the function on `test_song` to show that it has successfully removed `'[Kendrick Lamar:]'` and other instances of artist names from the song, and is returning the song as one string (NOT a list of strings) with newlines (`\n`) and punctuation removed and every word in lowercase.

```
def clean_song(song):
    clean_lines = [line for line in song if "[" not in line and "]" not in line]
    clean_song = " ".join(clean_lines)
    for symbol in ",.'?!()":
        clean_song = clean_song.replace(symbol, "")
    clean_song = clean_song.replace("\n", " ")
    return clean_song.lower()

clean_test_song = clean_song(test_song)
print(clean_test_song)
```

```
two wrongs dont make us right away  tell me somethings wrong  party all of our
lives away  to take you on  oh baby i want you  baby i need you  i wanna see you
baby i wanna go out yeah  baby i wanna go out yeah  baby i want you  baby i need
you  i wanna see you  baby i wanna go out yeah  baby i wanna go out yeah  all
night all night all night  all night  your bodys on fire  and your drinks on ice
all night all night all night  all night  your bodys on fire  and your drinks on
ice  oh my word oh my gosh oh my word oh my gosh  oh my word oh my gosh oh my
word oh my gosh  oh my word oh my gosh oh my word oh my gosh  oh my word oh my
gosh oh my word oh my gosh  everybody say kikiritikiki kikiritikiki  everybody
```

```
say kikiritikiki kikiritikiki  everybody say kikiritikiki kikiritikiki  everybody
say kikiritikiki kikiritikiki  ungbambe ungdedele ungbhasobhe unggudluke
ungbambe ungdedele ungbhasobhe unggudluke  ungbambe ungdedele ungbhasobhe
unggudluke  ungbambe ungdedele ungbhasobhe unggudluke  baby i want you  baby i
need you  i wanna see you  baby i wanna go out yeah  baby i wanna go out yeah
baby i want you  baby i need you  i wanna see you  baby i wanna go out yeah  baby
i wanna go out yeah  all night all night all night  all night  your bodys on fire
and your drinks on ice  all night all night all night  all night  your bodys on
fire  and your drinks on ice  we go  high up high up  high up high up  high up
high up  high up high up  high up high up  high up high up  high up high up  high
up high up  baby i want you  baby i need you  i wanna see you  baby i wanna go
out yeah  baby i wanna go out yeah  baby i want you  baby i need you  i wanna see
you  baby i wanna go out yeah  baby i wanna go out yeah  all night all night all
night  all night  your bodys on fire  and your drinks on ice  all night all night
all night  all night  your bodys on fire  and your drinks on ice
```

Great! Now, we can use `nltk` 's `word_tokenize()` function on the song string to get a fully tokenized version of the song. Test this function on `clean_test_song` to ensure that the function works.

```
tokenized_test_song = word_tokenize(clean_test_song)
tokenized_test_song[:10]
```

```
['two',
 'wrongs',
 'dont',
 'make',
 'us',
 'right',
 'away',
 'tell',
 'me',
 'somethings']
```

Great! Now that we can tokenize our songs, we can move onto vectorization.

Count Vectorization

Machine Learning algorithms don't understand strings. However, they do understand math, which means they understand vectors and matrices. By ***Vectorizing*** the text, we just convert the entire text into a vector, where each element in the vector represents a different word. The vector is the length of the entire vocabulary -- usually, every word that occurs in the English language, or at least every word that appears in our corpus. Any given sentence can then be represented as a vector where all the vector is 1 (or some other value) for each time that word appears in the sentence.

Consider the following example:

"I scream, you scream, we all scream for ice cream."

'aardvark'	'apple'	[...]	'I'	'you'	'scream'	'we'	'all'	'for'	'ice'
0	0	0	1	1	3	1	1	1	1

This is called a ***Sparse Representation***, since the strong majority of the columns will have a value of 0. Note that elements corresponding to words that do not occur in the sentence have a value of 0, while words that do appear in the sentence have a value of 1 (or 1 for each time it appears in the sentence).

Alternatively, we can represent this sentence as a plain old Python dictionary of word frequency counts:

```
BoW = {
    'I':1,
    'you':1,
    'scream':3,
    'we':1,
    'all':1,
    'for':1,
    'ice':1,
    'cream':1
}
```

Both of these are examples of ***Count Vectorization***. They allow us to represent a sentence as a vector, with each element in the vector corresponding to how many times that word is used.

Positional Information and Bag of Words

Notice that when we vectorize a sentence this way, we lose the order that the words were in. This is the ***Bag of Words*** approach mentioned earlier. Note that sentences that contain the same words will create the same vectors, even if they mean different things -- e.g. `'cats are scared of dogs'` and `'dogs are scared of cats'` would both produce the exact same vector, since they contain the same words.

In the cell below, create a function that takes in a tokenized, cleaned song and returns a count vectorized representation of it as a Python dictionary.

Hint: Consider using a `set()` since you'll need each unique word in the tokenized song!

```
def count_vectorize(tokenized_song):
    unique_words = set(tokenized_song)

    song_dict = {word:0 for word in unique_words}

    for word in tokenized_song:
        song_dict[word] += 1

    return song_dict

test_vectorized = count_vectorize(tokenized_test_song)
print(test_vectorized)
```

```
{'oh': 17, 'night': 24, 'high': 16, 'baby': 24, 'away': 2, 'need': 6, 'me': 1,
'i': 30, 'you': 19, 'our': 1, 'lives': 1, 'everybody': 4, 'your': 12, 'na': 18,
'on': 13, 'out': 12, 'somethings': 1, 'take': 1, 'kikiritikiki': 8, 'two': 1,
'to': 1, 'wrong': 1, 'dont': 1, 'party': 1, 'and': 6, 'yeah': 12, 'word': 8,
'bodys': 6, 'wrongs': 1, 'all': 25, 'drinks': 6, 'make': 1, 'of': 1, 'fire': 6,
'right': 1, 'us': 1, 'gosh': 8, 'up': 16, 'we': 1, 'say': 4, 'ungdedele': 4,
'see': 6, 'my': 16, 'ice': 6, 'unggudluke': 4, 'want': 6, 'tell': 1, 'wan': 18,
'ungbambe': 4, 'ungbhasobhe': 4, 'go': 13}
```

Great! You've just successfully vectorized your first text document! Now, let's look at a more advanced type of vectorization, TF-IDF!

TF-IDF Vectorization

TF-IDF stands for ***Term Frequency, Inverse Document Frequency***. This is a more advanced form of vectorization that weighs each term in a document by how unique it is to the given document it is contained in, which allows us to summarize the contents of a document using a few key words. If the word is used often in many other documents, it is not unique, and therefore probably not too useful if we wanted to figure out how this document is unique in relation to other documents. Conversely, if a word is used many times in a document, but rarely in all the other documents we are considering, then it is likely a good indicator for telling us that this word is important to the document in question.

The formula TF-IDF uses to determine the weights of each term in a document is ***Term Frequency*** multiplied by ***Inverse Document Frequency***. We just calculated our Term Frequency above with Count Vectorization!

Now that we have this, we can easily calculate *Inverse Document Frequency*. Note that this will need ALL of our documents (aka our songs), not just an individual document - so we'll put off testing this function for now.

In the cell below, complete a function that takes in a list of tokenized songs, with each item in the list being a clean, tokenized version of the song. The function should return a dictionary containing the inverse document frequency values for each word.

The formula for Inverse Document Frequency is:

$$\text{IDF}(t) = log_e(\frac{\text{Total Number of Documents}}{\text{Number of Documents with } t \text{ in it}})$$

```python
def inverse_document_frequency(list_of_token_songs):
    num_docs = len(list_of_token_songs)

    unique_words = set([item for sublist in list_of_token_songs for item in sublist]
    # Same as:
    # unique_words = set()
    # for song in list_of_dicts:
    #     for word in song.keys():
    #         unique_words.add(word)

    inv_doc_freq = {word:0 for word in unique_words}

    for word in unique_words:
        num_docs_with_word = 0
        for song_tokens in list_of_token_songs:
            if word in song_tokens:
                num_docs_with_word += 1
        inv_doc_freq[word] = np.log(num_docs / num_docs_with_word)

    return inv_doc_freq
```

Computing TF-IDF

Now that we can compute both Term Frequency and Inverse Document Frequency, computing an overall TF-IDF value is simple! All we need to do is multiply the two values.

In the cell below, complete the `tf_idf()` function. This function should take in a list of tokenized songs, just as the `inverse_document_frequency()` function did. This function returns a new list of dictionaries, with each dictionary containing the tf-idf vectorized representation of a corresponding song document. You'll need to calculate the term frequency for each song using the `count_vectorize()` function we defined above.

NOTE: Each document should contain the full vocabulary of the entire combined corpus! So, even if a song doesn't have the word "kikiritikiki" (a vocalization in our test song), it should have a dictionary entry with that word as the key and `0` as the value.

```
def tf_idf(list_of_token_songs):

    unique_words = set([item for sublist in list_of_token_songs for item in sublist]

    idf = inverse_document_frequency(list_of_token_songs)

    tf_idf_list_of_dicts = []
    for song_tokens in list_of_token_songs:
        song_tf = count_vectorize(song_tokens)
        doc_tf_idf = {word:0 for word in unique_words}
        for word in unique_words:
            if word in song_tokens:
                doc_tf_idf[word] = song_tf[word] * idf[word]
            else:
                doc_tf_idf[word] = 0
        tf_idf_list_of_dicts.append(doc_tf_idf)

    return tf_idf_list_of_dicts
```

Vectorizing All Documents

Now that we've created all the necessary helper functions, we can load in all of our documents and run each through the vectorization pipeline we've just created.

In the cell below, complete the `main()` function. This function should take in a list of file names (provided for you in the `filenames` list we created at the start), and then:

- Read in each document into a list of raw songs (where each song is a list of strings)
- Tokenize each document into a list of cleaned and tokenized songs
- Return a list of dictionaries vectorized using `tf-idf`, where each dictionary is a vectorized representation of each song

```
def main(filenames):

    all_songs = []
    for song in filenames:
        with open(f'data/{song}') as f:
            song_lyrics = f.readlines()
            all_songs.append(song_lyrics)

    all_song_tokens = []

    for song in all_songs:
        song_tokens = word_tokenize(clean_song(song))
        all_song_tokens.append(song_tokens)

    tf_idf_all_docs = tf_idf(all_song_tokens)
    return tf_idf_all_docs

tf_idf_all_docs = main(filenames)
```

Visualizing our Vectorizations

Now that we have a tf-idf representation of each document, we can move on to the fun part -- visualizing everything!

In the cell below, examine our dictionaries to figure out how many dimensions our dataset has.

HINT: Remember that every word is its own dimension!

```
vocab = list(tf_idf_all_docs[0].keys())
num_dims = len(vocab)
# Same as:
```

```
# num_dims = len(tf_idf_all_docs[0])
print(f"Number of Dimensions: {num_dims}")
```

```
Number of Dimensions: 1342
```

There are too many dimensions for us to visualize! In order to make it understandable to human eyes, we'll need to reduce it to 2 or 3 dimensions.

To do this, we'll use a technique called ***t-SNE*** (short for *t-Stochastic Neighbors Embedding*). This is too complex for us to code ourselves, so we'll make use of scikit-learn's implementation of it.

First, we need to pull the words out of the dictionaries stored in `tf_idf_all_docs` so that only the values remain, and store them in lists instead of dictionaries. This is because the t-SNE only works with array-like objects, not dictionaries.

In the cell below, create a list of lists that contains a list representation of the values of each of the dictionaries stored in `tf_idf_all_docs` . The same structure should remain -- e.g. the first list should contain only the values that were in the first dictionary in `tf_idf_all_docs` , and so on.

```
tf_idf_vals_list = []

for i in tf_idf_all_docs:
    tf_idf_vals_list.append(list(i.values()))

tf_idf_vals_list[0][:10]
```

```
[0, 0, 0, 2.302585092994046, 0, 0, 0, 0, 0, 0]
```

Now that we have only the values, we can use the `TSNE()` class from `sklearn` to transform our data appropriately. In the cell below, instantiate `TSNE()` with the following arguments:

- `n_components=3` (so we can compare 2 vs 3 components when graphing)
- `perplexity=19` (the highest number of neighbors explored given the size of our dataset)
- `learning_rate=200` (a higher learning rate than using 'auto', to avoid getting stuck in a local minimum)
- `init='random'` (so SKLearn will randomize the initialization)

- `random_state=13` (so that random initialization won't be TOO random)

Then, use the created object's `.fit_transform()` method to transform the data stored in `tf_idf_vals_list` into 3-dimensional data. Then, inspect the newly transformed data to confirm that it has the correct dimensionality.

```
t_sne_object_3d = TSNE(n_components = 3,
                       perplexity = 19,
                       learning_rate = 200,
                       init = 'random',
                       random_state = 13)

transformed_data_3d = t_sne_object_3d.fit_transform(np.array(tf_idf_vals_list))
transformed_data_3d
```

```
array([[  72.17791  ,   37.443684 , -116.68213  ],
       [ -67.891136 ,   81.81856  ,  243.60434  ],
       [ -24.182257 ,  109.413086 ,  440.08405  ],
       [ -64.313835 ,   25.211964 , -268.2567   ],
       [ 120.91158  ,  218.72166  ,  -17.992275 ],
       [ 300.82266  ,  -74.27421  ,   91.64757  ],
       [ 220.03706  , -117.81348  , -118.62039  ],
       [ 108.55898  , -148.69943  ,   71.698326 ],
       [ -73.65116  ,  231.3272   ,   43.416885 ],
       [  31.005625 , -176.20738  , -172.3928   ],
       [ 277.11734  ,   92.663025 ,  -58.398315 ],
       [-112.07886  , -182.77405  ,  -17.11454  ],
       [ 167.24355  ,   81.739235 ,  167.04172  ],
       [ -63.69105  , -185.92758  ,  182.46227  ],
       [-239.15219  ,   -2.7302566,  116.264694 ],
       [-145.02255  ,   24.366793 ,  -61.43305  ],
       [ 168.30348  ,   26.26588  ,  374.98557  ],
       [  -2.244803 ,   21.746862 ,   74.41611  ],
       [-163.42085  ,  202.15169  , -153.5312   ],
       [-213.13747  , -136.83485  , -195.99536  ]], dtype=float32)
```

We'll also want to check out how the visualization looks in 2d. Repeat the process above, but this time, instantiate `TSNE()` with 2 components instead of 3. Again, use `.fit_transform()` to transform the data and store it in the variable below, and then inspect it to confirm the transformed data has only 2 dimensions.

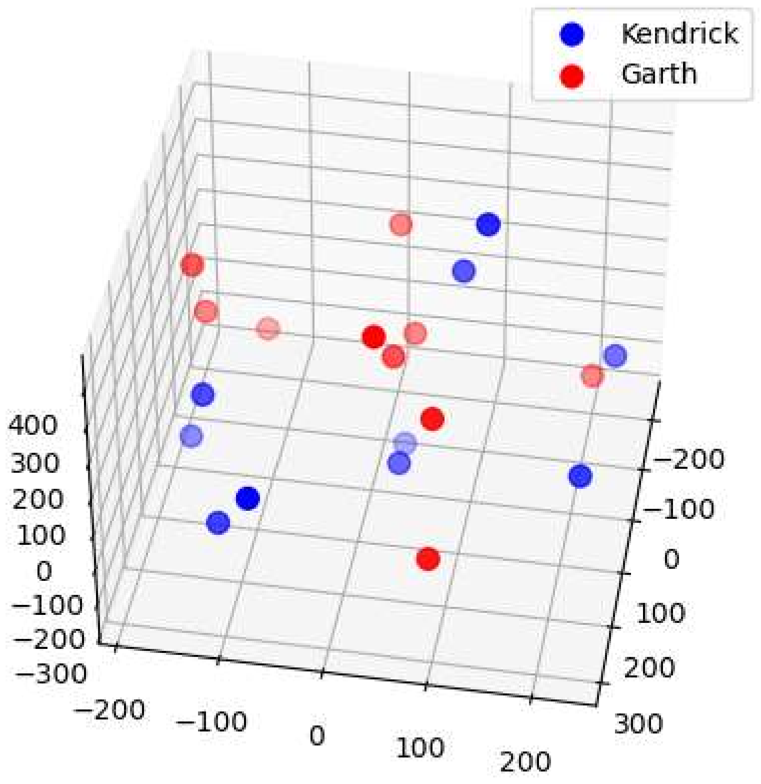
```
t_sne_object_2d = TSNE(n_components = 2,
                       perplexity = 19,
                       learning_rate = 200,
                       init = 'random',
                       random_state = 13)
transformed_data_2d = t_sne_object_2d.fit_transform(np.array(tf_idf_vals_list))
transformed_data_2d
```

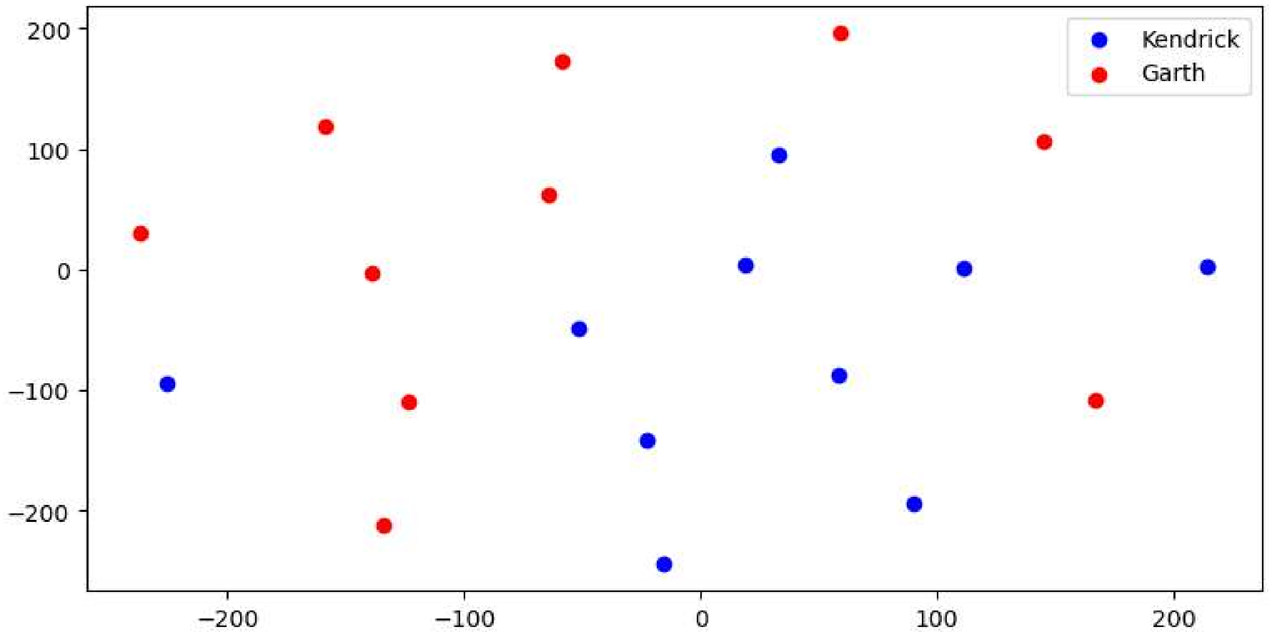
```
array([[  33.22776  ,   95.35896  ],
       [ -22.805809 , -141.09244  ],
       [ -51.266922 ,  -48.038868 ],
       [-225.76776  ,  -94.223045 ],
       [  58.735867 ,  -87.52201  ],
       [ -15.463868 , -244.34956  ],
       [  18.84531  ,    3.7228637],
       [ 214.41714  ,    2.8757088],
       [ 111.275665 ,    0.8636966],
       [  90.043365 , -193.43053  ],
       [ 144.8562   ,  106.657814 ],
       [ -64.06784  ,   61.506577 ],
       [  59.135918 ,  196.00252  ],
       [ -58.603584 ,  172.06688  ],
       [-236.7647   ,   30.09216  ],
       [-123.48959  , -109.93242  ],
       [-134.10022  , -212.06287  ],
       [ 167.00887  , -108.54331  ],
       [-138.71497  ,   -3.2438464],
       [-158.6191   ,  118.98085  ]], dtype=float32)
```

Now, let's visualize everything! Run the cell below to view both 3D and 2D visualizations of the songs.

```
kendrick_3d = transformed_data_3d[:10]
k3_x = [i[0] for i in kendrick_3d]
k3_y = [i[1] for i in kendrick_3d]
k3_z = [i[2] for i in kendrick_3d]

garth_3d = transformed_data_3d[10:]
g3_x = [i[0] for i in garth_3d]
g3_y = [i[1] for i in garth_3d]
g3_z = [i[2] for i in garth_3d]

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(k3_x, k3_y, k3_z, c='b', s=60, label='Kendrick')
ax.scatter(g3_x, g3_y, g3_z, c='red', s=60, label='Garth')
ax.view_init(40,10)
ax.legend()
plt.show()
```

```
kendrick_2d = transformed_data_2d[:10]
k2_x = [i[0] for i in kendrick_2d]
k2_y = [i[1] for i in kendrick_2d]

garth_2d = transformed_data_2d[10:]
g2_x = [i[0] for i in garth_2d]
g2_y = [i[1] for i in garth_2d]

fig = plt.figure(figsize=(20,10))
ax = fig.add_subplot(222)
ax.scatter(k2_x, k2_y, c='b', label='Kendrick')
ax.scatter(g2_x, g2_y, c='red', label='Garth')
ax.legend()
plt.show()
```





Interesting! Take a crack at interpreting these graphs by answering the following questions below:

What does each graph mean? Do you find one graph more informative than the other? Do you think that this method shows us discernable differences between Kendrick Lamar songs and Garth Brooks songs? Use the graphs and your understanding of TF-IDF to support your answer.

```
'''
WRITE YOUR ANSWER HERE

Both graphs show a basic trend among the red and blue dots, although the 3-dimensior
graph is more informative than the 2-dimensional graph. We see a separation between
two artists because they both have words that they use, but the other artist does nc

The words in each song that are common to both are reduced to very small numbers or
because of the log operation in the IDF function.  This means that the elements of e
song vector with the highest values will be the ones that have words that are unique
that specific document, or at least are rarely used in others.
'''
```

```
'\nWRITE YOUR ANSWER HERE\n\nBoth graphs show a basic trend among the red and
blue dots, although the 3-dimensional \ngraph is more informative than the 2-
dimensional graph. We see a separation between the \ntwo artists because they
both have words that they use, but the other artist does not. \n\nThe words in
each song that are common to both are reduced to very small numbers or to 0,
\nbecause of the log operation in the IDF function.  This means that the elements
```

```
of each \nsong vector with the highest values will be the ones that have words
that are unique to \nthat specific document, or at least are rarely used in
others.  \n'
```

Summary

In this lab, you learned how to:

- Tokenize a corpus of words and identify the different choices to be made while parsing them
- Use a count vectorization strategy to create a bag of words
- Use TF-IDF vectorization with multiple documents to identify words that are important/unique to certain documents
- Visualize and compare vectorized text documents

Releases

No releases published

Packages

No packages published

Contributors 6

Languages

● **Jupyter Notebook** 97.7% ● **Python** 2.3%