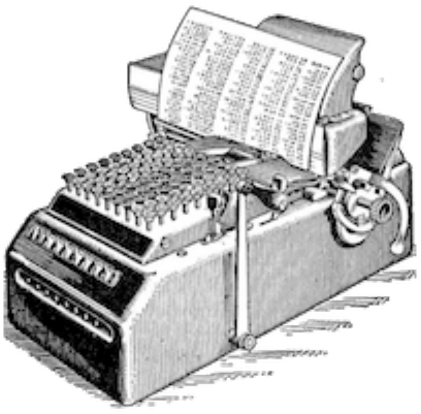
Analyzing Documents with TF-IDF

Matthew Lavin
Denison University

Recommended Citation
Lavin, M. (2019). Analyzing Documents with TF-IDF. Programming Historian, undefined(8).
http://dx.doi.org/10.46430/phen0082

Programming Historian



Analyzing Documents with TF-IDF (/en/lessons/analyzing-documents-with-tfidf)

Matthew J. Lavin

This lesson focuses on a foundational natural language processing and information retrieval method called Term Frequency - Inverse Document Frequency (tf-idf). This lesson explores the foundations of tf-idf, and will also introduce you to some of the questions and concepts of computationally oriented text analysis.

Peer-reviewed (https://github.com/programminghistorian/ph-submissions/issues/206)

Support PH (/en/individual)

edited by

- Zoe LeBlanc

reviewed by

- Quinn Dombrowski (https://orcid.org/0000-0001-5802-6623)
- Catherine Nygren

published | 2019-05-13 modified | 2021-11-10 difficulty | Medium https://doi.org/10.46430/phen0082

Contents

Overview

This lesson focuses on a core natural language processing and information retrieval method called Term Frequency - Inverse Document Frequency (**tf-idf**). You may have heard about **tf-idf** in the context of topic modeling, machine learning, or or other approaches to text analysis. **Tf-idf** comes up a lot in published work because it's both a corpus (https://en.wikipedia.org/wiki/Text_corpus) exploration method and a pre-processing step for many other text-mining measures and models.

Looking closely at tf-idf will leave you with an immediately applicable text analysis method. This lesson will also introduce you to some of the questions and concepts of computationally oriented text analysis. Namely, this lesson addresses how you can isolate a document's most important words from the kinds of words that tend to be highly frequent across a set of documents in that language. In addition to tf-idf, there are a number of computational methods for determining which words or phrases characterize a set of documents, and I highly recommend Ted Underwood's 2011 blog post as a supplement.[1]

Preparation

Suggested Prior Skills🔗

- Prior familiarity with Python or a similar programming language. Code for this lesson is written in Python 3.6, but you can run **tf-idf** in several different versions of Python, using one of several packages, or in various other programming languages. The precise level of code literacy or familiarity recommended is hard to estimate, but you will want to be comfortable with basic types and operations. To get the most out of this lesson, it is recommended that you work your way through something like Codeacademy's "Introduction to Python" course (https://www.codecademy.com/learn/learn-python), or that you complete some of the introductory Python lessons on the *Programming Historian* (/en/lessons/introduction-and-installation).
- In lieu of the above recommendation, you should review Python's basic types (https://www.learnpython.org/) (string, integer, float, list, tuple, dictionary), working with variables, writing loops in Python, and working with object classes/instances.
- Experience with Excel or an equivalent spreadsheet application if you wish to examine the linked spreadsheet files. You can also use the pandas library in python to view the CSVs.

Before You Begin🔗

- Install the Python 3 version of Anaconda. Installing Anaconda is covered in Text Mining in Python through the HTRC Feature Reader (/en/lessons/text-mining-with-extracted-features). This will install Python 3.6 (or higher), the Scikit-Learn library (https://scikit-learn.org/stable/install.html) (which we will use for **tf-idf**), and the dependencies needed to run a Jupyter Notebook (https://jupyter.org/).
- It is possible to install all these dependencies without Anaconda (or with a lightweight alternative like Miniconda (https://docs.conda.io/en/latest/miniconda.html)). For more information, see the section below titled "Alternatives to Anaconda"

Lesson Dataset🔗

Tf-idf, like many computational operations, is best understood by example. To this end, I've prepared a dataset of 366 *New York Times* historic obituaries (https://en.wikipedia.org/wiki/Obituary) scraped from https://archive.nytimes.com/www.nytimes.com/learning/general/onthisday/ (https://archive.nytimes.com/www.nytimes.com/learning/general/onthisday/). On each day of the year, *The New York Times* featured an obituary of someone born on that day.

Lesson files, including, this dataset, can be downloaded from lesson-files.zip (/assets/tf-idf/lesson-files.zip). The dataset is small enough that you should be able to open and read some if not all of the files. The original data is also available in the 'obituaries' folder, containing the '.html' files downloaded from the 2011 "On This Day" website and a folder of '.txt' files that represent the body of each obituary. These text files were generated using a Python library (https://docs.python.org/3/library/intro.html) called BeautifulSoup (https://www.crummy.com/software/BeautifulSoup/), which is covered in another *Programming Historian* lesson (see Intro to BeautifulSoup (/en/lessons/intro-to-beautiful-soup)).

This obituary corpus is an historical object in its own right. It represents, on some level, how the questions of inclusion and representation might affect both the decision to publish an obituary, and the decision to highlight a particular obituary many years later. The significance of such decisions has been further highlighted in recent months by *The New York Times* itself. In March 2018, the newspaper began publishing obituaries for "overlooked women".[2] In the words of Amisha Padnani and Jessica Bennett, "who gets remembered — and how — inherently involves judgment. To look back at the obituary archives can, therefore, be a stark lesson in how society valued various achievements and achievers." Viewed through this lens, the dataset provided here stands not as a representative sample of historic obituaries but, rather, a snapshot of who *The New York Times* in 2010-2011 considered worth highlighting. You'll notice that many of the historic figures are well known, which suggests a self-conscious effort to look back at the history of *The New York Times* and select obituaries based on some criteria.[3]

Tf-idf Definition and Background

Often inaccurately attributed to others, the procedure called Term Frequency - Inverse Document Frequency was introduced in a 1972 paper by Karen Spärck Jones under the name "term specificity."[4] Fittingly, Spärck Jones was the subject of an "Overlooked No More" obituary in January 2019.[5]

With **tf-idf**, instead of representing a term in a document by its raw frequency (number of occurrences) or its relative frequency (term count divided by document length), each term is weighted by dividing the term frequency by the number of documents in the corpus containing the word. The overall effect of this weighting scheme is to avoid a common problem when conducting text analysis: the most frequently used words in a document are often the most frequently used words in all of the documents. In contrast, terms with the highest **tf-idf** scores are the terms in a document that are *distinctively* frequent in a document, when that document is compared other documents.

If this explanation doesn't quite resonate, a brief analogy might help. Imagine that you are on vacation for a weekend in a new city, called Idf City. You're trying to choose a restaurant for dinner, and you'd like to balance two competing goals: first, you want to have a very good meal, and second, you want to choose a style of cuisine that's distinctively good in Idf City. That is, you don't want to have something you can get just anywhere. You can look up online reviews of restaurants all day, and that's just fine for your first goal, but what you need in order to satisfy the second goal is some way to tell the difference between good and distinctively good (or perhaps even uniquely good).

It's relatively easy, I think, to see that restaurant food could be:

1. both good and distinctive,
2. good but not distinctive,
3. distinctive but not good, or
4. neither good nor distinctive.

Term frequencies could have the same structure. A term might be:

1. Frequently used in a language like English, and especially frequent or infrequent in one document
2. Frequently used in a language like English, but used to a typical degree in one document
3. Infrequently used in a language like English, but distinctly frequent or infrequent in one document

4. Infrequently used in a language like English, and used at to a typical degree in one document

To understand how words can be frequent but not distinctive, or distinctive but not frequent, let's look at a text-based example. The following is a list of the top ten most frequent terms (and term counts) from one of the obituaries in our *New York Times* corpus.

Rank	Term	Count
1	the	21
2	of	16
3	her	15
4	in	14
5	and	13
6	she	10
7	at	8
8	cochrane	4
9	was	4
10	to	4

After looking at this list, imagine trying to discern information about the obituary that this table represents. We might infer from the presence of *her* and *cochrane* in the list that a woman named Cochrane is being discussed but, at the same time, this could easily be about a person from Cochrane, Wisconsin or someone associated with the Cochrane Collaboration (https://en.wikipedia.org/wiki/Cochrane_(organisation)), a non-profit, non-governmental organization. The problem with this list is that most of top terms would be top terms in any obituary and, indeed, any sufficiently large chunk of writing in most languages. This is because most languages are heavily dependent on function words like *the, as, of, to,* and *from* that serve primarily grammatical or structural purposes, and appear regardless of the text's subject matter. A list of an obituary's most frequent terms tell us little about the obituary or the person being memorialized. Now let's use **tf-idf** term weighting to compare the same obituary from the first example to the rest of our corpus of *New York Times* obituaries. The top ten term scores look like this:

Rank	Term	Count
1	cochrane	24.85
2	her	22.74
3	she	16.22
4	seaman	14.88
5	bly	12.42
6	nellie	9.92
7	mark	8.64
8	ironclad	6.21
9	plume	6.21
10	vexations	6.21

In this version of the list, *she* and *her* have both moved up. *cochrane* remains, but now we have at least two new name-like words: *nellie* and *bly.* Nellie Bly (https://en.wikipedia.org/wiki/Nellie_Bly) was a turn-of-the-century journalist best known today for her investigative journalism, perhaps most remarkably when she had herself committed to the New York City Lunatic Asylum for ten days in order to write an expose on the mistreatment of mental health patients. She was born Elizabeth Cochrane Seaman, and Bly was her pen name or *nom-de-plume*. With only a few details about Bly, we can account for seven of the top ten **tf-idf** terms: *cochrane, her, she, seaman, bly, nellie,* and *plume*. To understand *mark*, *ironclad*, and *vexations*, we can return to the original obituary and discover that Bly died at St. Mark's Hospital. Her husband (https://en.wikipedia.org/wiki/Robert_Seaman) was president of the Ironclad Manufacturing Company. Finally, "a series of forgeries by her employees, disputes of various sorts, bankruptcy and a mass of vexations and costly litigations swallowed up Nellie Bly's fortune."[6] Many of the terms on this list are mentioned as few as one, two, or three times; they are not frequent by any measure. Their presence in this one document, however, are all distinctive compared with the rest of the corpus.

Procedure

How the Algorithm Works

Tf-idf can be implemented in many flavors, some more complex than others. Before I begin discussing these complexities, however, I would like to trace the algorithmic operations of one particular version. To this end, we will go back to the Nellie Bly obituary and convert the top ten term counts into **tf-idf** scores using the same steps that were used to create the above **tf-idf** example. These steps parallel Scikit-Learn's **tf-idf** (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) implementation. Addition, multiplication, and division are the primary mathematical operations necessary to follow along. At one point, we must calculate the natural logarithm (https://en.wikipedia.org/wiki/Natural_logarithm) of a variable, but this can be done with most online calculators and calculator mobile apps. Below is a table with the raw term counts for the first thirty words, in alphabetical order, from Bly's obituary, but this version has a second column that represents the number of documents in which each term can be found. Document frequency (**df**) is a count of how many documents from the corpus each word appears in. (Document frequency for a particular word can be represented as $\mathbf{df_i}$.)

Index	Term	Count	Df
1	afternoon	1	66
2	against	1	189
3	age	1	224
4	ago	1	161
5	air	1	80
6	all	1	310
7	american	1	277
8	an	1	352
9	and	13	364
10	around	2	149
11	as	2	357

12	ascension	1	6
13	asylum	1	2
14	at	8	362
15	avenue	2	68
16	balloon	1	2
17	bankruptcy	1	8
18	barrel	1	7
19	baxter	1	4
20	be	1	332
21	beat	1	33
22	began	1	241
23	bell	1	24
24	bly	2	1
25	body	1	112
26	born	1	342
27	but	1	343
28	by	3	349
29	career	1	223
30	character	1	89

To calculate inverse document frequency for each term, the most direct formula would be **N/df$_i$**, where **N** represents the total number of documents in the corpus. However, many implementations normalize the results with additional operations. In TF-IDF, normalization is generally used in two ways: first, to prevent bias in term frequency from terms in shorter or longer documents; second, to calculate each term's idf value (inverse document frequency). For example, Scikit-Learn's implementation represents **N** as **N+1**, calculates the natural logarithm of **(N+1)/df$_i$**, and then adds 1 to the final result. This lesson will return to the topic of normalization in the section below titled "Scikit-Learn Settings".

To express Scikit-Learn's **idf** transformation[7], we can state the following equation:

$$idf_i = ln[\,(N+1)/\,df_i] + 1$$

Once **idf$_i$** is calculated, **tf-idf$_i$** is **tf$_i$** multiplied by **idf$_i$**.

$$tf\text{-}idf_i = tf_i \,\times\, idf_i$$

Mathematical equations like these can be a bit bewildering if you're not used to them. Once you've had some experience with them, they can provide a more lucid description of an algorithm's operations than any well written paragraph. (For more on this subject, Ben Schmidt's "Do Digital Humanists Need to Understand Algorithms?" is a good place to start.[8]) To make the **idf** and **tf-idf** equations more concrete, I've added two new columns to the terms frequency table from before. The first new column represents the derived **idf** score, and the second new column multiplies the Count and Idf columns to derive the final

tf-idf score. Notice that that **idf** score is higher if the term appears in fewer documents, but that the range of visible **idf** scores is between 1 and 6. Different normalization (https://en.wikipedia.org/wiki/Normalization_(statistics)) schemes would produce different scales.

Note also that the **tf-idf** column, according to this version of the algorithm, cannot be lower than the count. This effect is also the result of our normalization method; adding 1 to the final **idf** value ensures that we will never multiply our Count columns by a number smaller than one, which preserves the original distribution of the data.

Index	Term	Count	Df	Idf	Tf-idf
1	afternoon	1	66	2.70066923	2.70066923
2	against	1	189	1.65833778	1.65833778
3	age	1	224	1.48926145	1.48926145
4	ago	1	161	1.81776551	1.81776551
5	air	1	80	2.51091269	2.51091269
6	all	1	310	1.16556894	1.16556894
7	american	1	277	1.27774073	1.27774073
8	an	1	352	1.03889379	1.03889379
9	and	13	364	1.00546449	13.07103843
10	around	2	149	1.89472655	3.78945311
11	as	2	357	1.02482886	2.04965772
12	ascension	1	6	4.95945170	4.95945170
13	asylum	1	2	5.80674956	5.80674956
14	at	8	362	1.01095901	8.08767211
15	avenue	2	68	2.67125534	5.34251069
16	balloon	1	2	5.80674956	5.80674956
17	bankruptcy	1	8	4.70813727	4.70813727
18	barrel	1	7	4.82592031	4.82592031
19	baxter	1	4	5.29592394	5.29592394
20	be	1	332	1.09721936	1.09721936
21	beat	1	33	3.37900132	3.37900132
22	began	1	241	1.41642412	1.41642412
23	bell	1	24	3.68648602	3.68648602
24	bly	2	1	6.21221467	12.42442933
25	body	1	112	2.17797403	2.17797403
26	born	1	342	1.06763140	1.06763140
27	but	1	343	1.06472019	1.06472019
28	by	3	349	1.04742869	3.14228608
29	career	1	223	1.49371580	1.49371580
30	character	1	89	2.40555218	2.40555218

These tables collectively represent one particular version of the **tf-idf** transformation. Of course, **tf-idf** is generally calculated for all terms in all of the documents in your corpus so that you can see which terms in each document have the highest **tf-idf** scores. To get a better sense of the what your output might look like after executing such an operation, download and open the full Excel file for Bly's obituary by downloading the lesson files (/assets/tf-idf/lesson-files.zip), extracting the '.zip' archive, and opening 'bly_tfidf_all.xlsx'.

How to Run it in Python 3🔗

In this section of the lesson, I will walk through the steps I followed to calculate **tf-idf** scores for all terms in all documents in the lesson's obituary corpus. If you would like to follow along, you can download the lesson files, extract the '.zip' archive, and run the Jupyter Notebook inside of the 'lesson' folder. You can also create a new Jupyter Notebook in that location copy/paste blocks of code from this tutorial as you go. If you are using Anaconda, visit the Jupyter Notebook Documentation Page (https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/execute.html) for more information on changing the Jupyter Notebook startup location. As with any programming language, there's more than one way to do each of the steps I discuss below.

My first block of code is designed to retrieve all the filenames for '.txt' files in the 'txt' folder. The following lines of code import the `Path` class from the `pathlib` library and use the `Path().rglob()` method to generate a list of all the files in the 'txt' folder that end with '.txt'. `pathlib` will also join the `file.parent` folder location with each file name to provide full file paths for each file (on MacOS or Windows).

Using this method, I append each text file name to the list called `all_txt_files`. Finally, I return the length of `all_txt_files` to verify that I've found 366 file names. This loop-and-append approach is very common in Python.

```
from pathlib import Path

all_txt_files =[]
for file in Path("txt").rglob("*.txt"):
     all_txt_files.append(file.parent / file.name)
# counts the length of the list
n_files = len(all_txt_files)
print(n_files)
```

A quick note on variable names. The two most common variable naming patterns prioritize convenience and semantic meaning respectively. For convenience, one might name a variable **x** so it's easier and faster to type when referencing it. Semantic variables, meanwhile, attempt to describe function or purpose. By naming my list of all text files `all_txt_files` and the variable representing the number of files `n_files`, I'm prioritizing semantic meaning. Meanwhile, I'm using abbreviations like `txt` for text and `n` for number to save on typing, or using `all_txt_files` instead of `all_txt_file_names` because brevity is still a goal. Underscore and capitalization norms are specified in PEP-8, Python's official style guide, with which you should try to be generally familiar.[9]

For various resons, we want our files to count up by day and month since there's on file for every day and month of a year. We can use the `sort()` method to put the files in ascending numerical order and print the first file to make sure it's 'txt/0101.txt'.

```
all_txt_files.sort()
all_txt_files[0]
```

Next, we can use our list of file names to load each file and convert them to a format that Python can read and understand as text. In this block of code, I do another loop-and-append operation. This time, I loop my list of file names and open each file. I then use Python's `read()` method to convert each text file to a string (`str`), which is how Python knows to think of the data as text. I append each string, one by one, to a new list called `all_docs` . Crucially, the string objects in this list have the same order as the file names in the `all_txt_files` list.

```
all_docs = []
for txt_file in all_txt_files:
    with open(txt_file) as f:
        txt_file_as_string = f.read()
    all_docs.append(txt_file_as_string)
```

This is all the setup work we require. Text processing steps like tokenization (https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization) and removing punctuation will happen automatically when we use Scikit-Learn's `TfidfVectorizer` to convert documents from a list of strings to **tf-idf** scores. One could also supply a list of stopwords here (commonly used words that you want to ignore). To tokenize and remove stopwords in languages other than English, you may need to preprocess the text with another Python library or supply a custom tokenizer and stopword list when Scikit-Learn's `TfidfVectorizer` . The following block of code imports `TfidfVectorizer` from the Scikit-Learn library, which comes pre-installed with Anaconda. `TfidfVectorizer` is a class (written using object-oriented programming), so I instantiate it with specific parameters as a variable named `vectorizer` . (I'll say more about these settings in the section titled "Scikit-Learn Settings".) I then run the object's `fit_transform()` method on my list of strings (a variable called `all_docs`). The stored variable `X` is output of the `fit_transform()` method.

```
#import the TfidfVectorizer from Scikit-Learn.
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(max_df=.65, min_df=1, stop_words=None, use_idf=True, 
norm=None)
transformed_documents = vectorizer.fit_transform(all_docs)
```

The `fit_transform()` method above converts the list of strings to something called a sparse matrix (https://en.wikipedia.org/wiki/Sparse_matrix). In this case, the matrix represents **tf-idf** values for all texts. Sparse matrices save on memory by leaving out all zero values, but we want access to those, so the next block uses the `toarray()` method to convert the sparse matrices to a numpy array (https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html). We can print the length of the array to ensure that it's the same length as our list of documents.

```
transformed_documents_as_array = transformed_documents.toarray()
# use this line of code to verify that the numpy array represents the same number of
documents that we have in the file list
len(transformed_documents_as_array)
```

A numpy array is list-like but not exactly a list, and I could fill an entire tutorial discussing the differences, but there's only one aspect of numpy arrays we need to know right now: it converts the data stored in `transformed_documents` to a format where every **tf-idf** score for every term in every document is represented. Sparse matrices, in contrast, exclude zero-value term scores.

We want every term represented so that each document has the same number of values, one for each word in the corpus. Each item in `transformed_documents_as_array` is an array of its own representing one document from our corpus. As a result of all this, we essentially have a grid where each row is a document, and each column is a term. Imagine one table from a spreadsheet representing each document, like the tables above, but without column or row labels.

To merge the values with their labels, we need two pieces of information: the order of the documents, and the order in which term scores are listed. The order of these documents is easy because it's the same order as the variable `all_docs list`. The full term list is stored in our `vectorizer` variable, and it's in the same order that each item in `transformed_documents_as_array` stores values. We can use the `the TFIDFVectorizer` class's `get_feature_names()` method to get that list, and each row of data (one document's **tf-idf** scores) can be rejoined with the term list. (For more details on pandas dataframes, see the lesson "Visualizing Data with Bokeh and Pandas" (/en/lessons/visualizing-with-bokeh).)

```
import pandas as pd

# make the output folder if it doesn't already exist
Path("./tf_idf_output").mkdir(parents=True, exist_ok=True)

# construct a list of output file paths using the previous list of text files the
relative path for tf_idf_output
output_filenames = [str(txt_file).replace(".txt", ".csv").replace("txt/",
"tf_idf_output/") for txt_file in all_txt_files]

# loop each item in transformed_documents_as_array, using enumerate to keep track of
the current position
for counter, doc in enumerate(transformed_documents_as_array):
    # construct a dataframe
    tf_idf_tuples = list(zip(vectorizer.get_feature_names(), doc))
    one_doc_as_df = pd.DataFrame.from_records(tf_idf_tuples, columns=['term',
'score']).sort_values(by='score', ascending=False).reset_index(drop=True)

    # output to a csv using the enumerated value for the filename
    one_doc_as_df.to_csv(output_filenames[counter])
```

The above block of code has three parts:

1. After importing the pandas library, it checks for a folder called ‘tf_idf_output’ and creates it if it doesn’t exist.

2. It takes the list of ‘.txt’ files from my earlier block of code and use it to construct a counterpart ‘.csv’ file path for each ‘.txt’ file. The output_filenames variable will, for example, convert ‘txt/0101.txt’ (the path of the first ‘.txt’ file) to ‘tf_idf_output/0101.csv’, and on and on for each file.

3. Using a loop, it merges each vector of **tf-idf** scores with the feature names from vectorizer, converts each merged term/score pairs to a pandas dataframe, and saves each dataframe to its corresponding ‘.csv’ file.

Interpreting Word Lists: Best Practices and Cautionary Notes🔗

After you run the code excerpts above, you will end up with a folder called ‘tf_idf_output’ with 366 ‘.csv’ files in it. Each file corresponds to an obituary in the ‘txt’ folder, and each contains a list of terms with **tf-idf** scores for that document. As we saw with Nellie Bly’s obituary, these term lists can be very suggestive; however, it’s important to understand that over-interpreting your results can actually distort your understanding of an underlying text.

In general, it’s best to begin with the ideas that these term lists will be helpful for generating hypotheses or research questions. **Tf-idf** results but will not necessarily produce definitive claims. For example, I have assembled a quick list of obituaries for late 19th- and early 20th-century figures who all worked for newspapers and magazines and had some connection to social reform. My list includes Nellie Bly, Willa Cather (https://en.wikipedia.org/wiki/Willa_Cather), W.E.B. Du Bois (https://en.wikipedia.org/wiki/W._E._B._Du_Bois), Upton Sinclair (https://en.wikipedia.org/wiki/Upton_Sinclair), Ida Tarbell (https://en.wikipedia.org/wiki/Ida_Tarbell), but there may be other figures in the corpus who fit the same criteria.[10]

I originally expected to see many shared terms, but I was surprised. Each list is dominated by individualized words (proper names, geographic places, companies, etc.) but I could screen these out using my **tf-idf** settings, or just ignore them. Simultaneously, I can look for words overtly indicating each figure’s ties to the profession of authorship. (The section of this tutorial titled Scikit-Learn Settings says more about how you can treat a named entity or a phrase as a single token.) The following table shows the top 20 **tf-idf** terms by rank for each obituary:

Tf-idf Rank	Nellie Bly	Willa Cather	W.E.B. Du Bois	Upton Sinclair	Ida Tarbell
1	cochrane	cather	dubois	sinclair	tarbell
2	her	her	dr	socialist	she
3	she	she	negro	upton	her
4	seaman	nebraska	ghana	**books**	lincoln
5	bly	miss	peace	lanny	miss
6	nellie	forrester	**encyclopedia**	social	oil
7	mark	sibert	communist	budd	abraham
8	ironclad	twilights	barrington	jungle	mcclure

9	**plume**	willa	fisk	brass	easton
10	vexations	antonia	atlanta	california	**volumes**
11	phileas	mcclure	folk	**writer**	minerva
12	597	**novels**	booker	vanzetti	standard
13	elizabeth	pioneers	successively	macfadden	business
14	**nom**	cloud	souls	sacco	titusville
15	balloon	**book**	council	**wrote**	**articles**
16	forgeries	calif	party	meat	bridgeport
17	mcalpin	**novel**	disagreed	**pamphlets**	expose
18	asylum	southwest	harvard	my	trusts
19	fogg	**verse**	**arts**	industry	mme
20	verne	**wrote**	soviet	**novel**	**magazine**

I've used boldface to indicate terms that seem overtly related to authorship or writing. The list includes *articles*, *arts*, *book*, *book*, *books*, *encyclopedia*, *magazine*, *nom*, *novel*, *novels*, *pamphlets*, *plume*, *verse*, *volumes*, *writer*, and *wrote*, but it could be extended to include references to specific magazine or book titles. Setting aside momentarily such complexities, it is striking to me that Cather and Sinclair's lists have so many words for books and writing, whereas Bly, Du Bois and Tarbell's do not.

I could easily jump to conclusions. Cather's identity seems to be tied most to her gender, her sense of place, and her fiction and verse. Sinclair more so with his politics and his writings about meat, industry, and specifically the well known, controversial trial and execution of Nicola Sacco and Bartolomeo Vanzetti. Bly is tied to her pen name, her husband, and her writing about asylums. Du Bois is linked to race and his academic career. Tarbell is described by what she wrote about: namely business, the trusts, Standard Oil, and Abraham Lincoln. Going further, I could argue that gender seems more distinctive for women than it is for men; race is only a top term for the one African American in my set.

Each of these observations forms the basis for a deeper question, but these details aren't enough to make generalizations. Foremost, I need to consider whether my **tf-idf** settings are producing effects that would disappear under other conditions; robust results should be stable enough to appear with various settings. (Some of these settings are covered in the "Scikit-Learn Settings" section.) Next, I should read at least some of the underlying obituaries to make sure I'm not getting false signals from any terms. If I read Du Bois's obituary, for example, I may discover that mentions of his work "The Encyclopedia of the Negro," contribute at least partially to the overall score of the word *negro*.

Likewise, I can discover that Bly's obituary does include words like *journalism*, *journalistic*, *newspapers*, and *writing*, but the obituary is very short, meaning most words mentioned in it occur only once or twice, which means that words with very high **idf** scores are even more likely to top her list. I really want **tf** and **idf** to be balanced, so I could rule out words that appear in only a few documents, or I could ignore results for obituaries below a certain word count.

Finally, I can design tests to measure directly questions like: were obituaries of African Americans are more likely to mention race? I think the prediction that they did is a good hypothesis, but I should still subject my generalizations to scrutiny before I form conclusions.

Some Ways Tf-idf Can Be Used in Computational History🔗

As I have described, **tf-idf** has its origins in information retrieval, and the idea of weighting term frequencies against norms in a larger corpus continues to be used to power various aspects of everyday web applications, especially text-based search engines. However, in a cultural analytics or computational history context, **tf-idf** is suited for a particular set of tasks. These uses tend to fall into one of three groups.

1. As an Exploratory Tool or Visualization Technique🔗

As I've already demonstrated, terms lists with **tf-idf** scores for each document in a corpus can be a strong interpretive aid in themselves, they can help generate hypotheses or research questions. Word lists can also be the building bocks for more sophisticated browsing and visualization strategies. "A full-text visualization of the Iraq War Logs" (http://jonathanstray.com/a-full-text-visualization-of-the-iraq-war-logs), by Jonathan Stray and Julian Burgess, is a good example of this use case.[11] Using **tf-idf**-transformed features, Stray and Burgess build a network visualization that positions Iraq War logs in relation to their most distinctive keywords. This way of visualizing textual information led Stray to develop the Overview Project (https://www.overviewdocs.com), which provides a dashboard for users to visualize and search thousands of documents at a time. We could use this kind of approach to graph our obituaries corpus and see if there are keyword communities.

2. Textual Similarity and Feature Sets🔗

Since **tf-idf** will often produce lower scores for high frequency function words and increased scores for terms related to the topical signal of a text, it is well suited for tasks involving textual similarity. A search index will often perform **tf-idf** on a corpus and return ranked results to user searches by looking for documents with the highest cosine similarity to the user's search string. The same logic can be used to ask a question like "Which obituary in our corpus is most similar to Nellie Bly's obituary?"

Similarly, we could use **tf-idf** to discover the top terms related to a document or a group of documents. For example, I could gather together a selection of obituaries about journalists (Bly included) and combine them into one document before running **tf-idf**. The output for that document would now work as a heuristic for terms that are distinctive in my journalism obituaries in the corpus when compared with other obituaries in the corpus. I could use such a term list for a range of other computational tasks.

3. As a Pre-processing Step🔗

The above paragraphs gesture at why **tf-idf** pre-processing is so often used with machine learning. **Tf-idf**-transformed features tend to have more predictive value than raw term frequencies, especially when classifying a supervised machine learning model, in part because it tends to increase the weight of topic words and reduce the weight of high frequency function words. One notable exception to this generalization is authorship attribution, where high frequency function words are highly predictive. As I will show in the "Scikit-Learn Settings" section, **tf-idf** can also be used to cull machine learning feature lists and, often, building a model with fewer features is desirable.

Potential Variations of Tf-idf🔗

Scikit-Learn Settings🔗

The Scikit-Learn `TfidfVectorizer` has several internal settings that can be changed to affect the output. In general, these settings all have pros and cons; there's no singular, correct way to preset them and produce output. Instead, it's best to understand exactly what each setting does so that you can describe and defend the choices you've made. The full list of parameters is described in Scikit-Learn's documentation (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html), but here are some of the most important settings:

1. stopwords

In my code, I used `python stopwords=None` but `python stopwords='english'` is available. This setting will filter out words using a preselected list (https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/feature_extraction/_stop_words.py) of high frequency function words such as 'the', 'to', and 'of'. Depending on your settings, many of these terms will have low **tf-idf** scores regardless because they tend to be found in all documents. For a discussion of some publicly available stop word lists (including Scikit-Learn's), see "Stop Word Lists in Free Open-source Software Packages" (https://aclweb.org/anthology/W18-2502).

2. min_df, max_df

These settings control the minimum number of documents a term must be found in to be included and the maximum number of documents a term can be found in in order to be included. Either can be expressed as a decimal between 0 and 1 indicating the percent threshold, or as a whole number that represents a raw count. Setting max_df below .9 will typically remove most or all stopwords.

3. max_features

This parameter can be used to winnow out terms by frequency before running tf-idf. It can be especially useful in a machine learning context when you do not wish to exceed a maximum recommended number of term features.

4. norm, smooth_idf, and sublinear_tf

Each of these will affect the range of numerical scores that the **tf-idf** algorithm outputs. norm supports l1 and l2 normalization, which you can read about on machinelearningmastery.com (https://machinelearningmastery.com/vector-norms-machine-learning/). Smooth-idf adds one to each document frequency score, "as if an extra document was seen containing every term in the collection exactly once." Sublinear_tf applies another scaling transformation, replacing tf with log(tf). For more on **tf-idf** smoothing and normalization, see Manning, Raghavan, and Schütze.[12]

Beyond Term Features

Since the basic idea of **tf-idf** is to weight term counts against the number of documents in which terms appear, the same logic can be used on other text-based features. For example, it is relatively straightforward to combine **tf-idf** with stemming or lemmatization (https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html). Stemming and lemmatization are two common ways to group together different word forms/inflections; for example, the stem of both *happy* and *happiness* is *happi*, and the lemma of both is *happy*. After stemming or lemmatization, stem or lemma counts can be substituted for term counts, and the **(s/l)f-idf** transformation can be applied. Each stem or lemma will have a higher **df** score than each of the words it groups together, so lemmas or stems with many word variants will tend to have lower **tf-idf** scores.

Similarly, the **tf-idf** transformation can be applied to n-grams. A Fivethirtyeight.com post from March 2016 called "These Are The Phrases Each GOP Candidate Repeats Most" (https://fivethirtyeight.com/features/these-are-the-phrases-each-gop-candidate-repeats-most/) uses such an approach to perform the inverse-document frequency calculation on phrases rather than words.[13]

Tf-idf and Common Alternatives

Tf-idf can be compared with several other methods of isolating and/or ranking important term features in a document or collection of documents. This section provides a brief mention of four related but distinct measures that target similar but not identical aspects of textual information.

1. Keyness

Keyness is a catchall term for a constellation of statistical measures that attempt to indicate the numerical significance of a term to a document or set of documents, in direct comparison with a larger set of documents or corpus. Depending on how we set up our **tf-idf** transformation, it may isolate many of a document's most important features, but **tf-idf** is not as precise as the most commonly used measures of keyness. Rather than changing a document's term frequency scores, keyness testing produces a numerical indicator of how statistically typical or atypical the term's usage in a text is. With a Chi-square test (https://en.wikipedia.org/wiki/Chi-squared_test), for example, we can evaluate the relationship of a term frequency to an established norm, and derive a P-value (https://en.wikipedia.org/wiki/P-value) indicating the probability of encountering the observed difference in a random sample. For more information on keyness, see Bondi and Scott.[14]

2. Topic Models

Topic modeling and **tf-idf** are radically different techniques, but I find that newcomers to digital humanities often want to run topic modeling on a corpus as a first step and, in at least some of those cases, running **tf-idf** instead of generating topic models would be preferable.[15] **Tf-idf** is especially appropriate if you are looking for a way to get a bird's eye view of your corpus early in the exploratory phase of your research because the algorithm is transparent and the results are reproducible. As Ben Schmidt suggests, scholars using topic modeling need to know that "topics may not be as coherent as they assume."[16] This is one reason **tf-idf** is integrated into the Overview Project (https://www.overviewdocs.com). Topic models can also help scholars explore their corpora, and they have several advantages over other techniques, namely that they suggest broad categories or communities of texts, but this a general advantage of unsupervised clustering methods. Topic models are especially appealing because documents are assigned scores for how well they fit each topic, and because topics are represented as lists of co-occurring terms, which provides a strong sense of how terms relate to groupings. However, the probabilistic model behind topic models is sophisticated, and it's easy to warp your results if you don't understand what you're doing. The math behind **tf-idf** is lucid enough to depict in a spreadsheet.

3. Automatic Text Summarization

Text summarization is yet another way to explore a corpus. Rada Mihalcea and Paul Tarau, for example, have published on TextRank, "a graph-based ranking model for text processing" with promising applications for keyword and sentence extraction.[17] As with topic modeling, TextRank and **tf-idf** are altogether dissimilar in their approach to information retrieval, yet the goal of both algorithms has a great deal of overlap. It may be appropriate for your research, especially if your goal is to get a relatively quick a sense of your documents' contents before designing a larger research project.

References and Further Reading

- Beckman, Milo. "These Are The Phrases Each GOP Candidate Repeats Most," *FiveThirtyEight*, March 10, 2016. https://fivethirtyeight.com/features/these-are-the-phrases-each-gop-candidate-repeats-most/

- Bennett, Jessica, and Amisha Padnani. "Overlooked," March 8, 2018. https://www.nytimes.com/interactive/2018/obituaries/overlooked.html

- Blei, David M., Andrew Y. Ng, and Michael I. Jordan, "Latent Dirichlet Allocation" *Journal of Machine Learning Research* 3 (January 2003): 993-1022.

- Bondi, Marina, and Mike Scott, eds. *Keyness in Texts*. Philadelphia: John Benjamins, 2010.

- Bowles, Nellie. "Overlooked No More: Karen Sparck Jones, Who Established the Basis for Search Engines" *The New York Times*, January 2, 2019. https://www.nytimes.com/2019/01/02/obituaries/karen-sparck-jones-overlooked.html

- Documentation for TfidfVectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

- Grimmer, Justin and King, Gary, Quantitative Discovery from Qualitative Information: A General-Purpose Document Clustering Methodology (2009). APSA 2009 Toronto Meeting Paper. Available at SSRN: https://ssrn.com/abstract=1450070

- "Ida M. Tarbell, 86, Dies in Bridgeport" *The New York Times*, January 7, 1944, 17. https://www.nytimes.com

- Manning, C.D., P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge: Cambridge University Press, 2008.

- Mihalcea, Rada, and Paul Tarau. "Textrank: Bringing order into text." In Proceedings of the 2004 conference on empirical methods in natural language processing. 2004.

- "Nellie Bly, Journalist, Dies of Pneumonia" *The New York Times*, January 28, 1922, 11. https://www.nytimes.com

- Salton, G. and M.J. McGill, *Introduction to Modern Information Retrieval*. New York: McGraw-Hill, 1983.

- Schmidt, Ben. “Do Digital Humanists Need to Understand Algorithms?” *Debates in the Digital Humanities 2016*. Online edition. Minneapois: University of Minnesota Press. http://dhdebates.gc.cuny.edu/debates/text/99

- –. “Words Alone: Dismantling Topic Models in the Humanities,” *Journal of Digital Humanities*. Vol. 2, No. 1 (2012): n.p. http://journalofdigitalhumanities.org/2-1/words-alone-by-benjamin-m-schmidt/

- Spärck Jones, Karen. “A Statistical Interpretation of Term Specificity and Its Application in Retrieval.” Journal of Documentation 28, no. 1 (1972): 11–21.

- Stray, Jonathan, and Julian Burgess. “A Full-text Visualization of the Iraq War Logs,” December 10, 2010 (Update April 2012). http://jonathanstray.com/a-full-text-visualization-of-the-iraq-war-logs

- Underwood, Ted. “Identifying diction that characterizes an author or genre: why Dunning’s may not be the best method,” *The Stone and the Shell*, November 9, 2011. https://tedunderwood.com/2011/11/09/identifying-the-terms-that-characterize-an-author-or-genre-why-dunnings-may-not-be-the-best-method/

- –. “The Historical Significance of Textual Distances”, Preprint of LaTeCH-CLfL Workshop, COLING, Santa Fe, 2018. https://arxiv.org/abs/1807.00181

- van Rossum, Guido, Barry Warsaw, and Nick Coghlan. “PEP 8 – Style Guide for Python Code.” July 5, 2001. Updated July 2013. https://www.python.org/dev/peps/pep-0008/

- Whitman, Alden. “Upton Sinclair, Author, Dead; Crusader for Social Justice, 90” *The New York Times*, November 26, 1968, 1, 34. https://www.nytimes.com

- “W. E. B. DuBois Dies in Ghana; Negro Leader and Author, 95” *The New York Times*, August 28, 1963, 27. https://www.nytimes.com

- “Willa Cather Dies; Noted Novelist, 70” *The New York Times*, April 25, 1947, 21. https://www.nytimes.com

Alternatives to Anaconda

If you are not using Anaconda, you will need to cover the following dependencies:

1. Install Python 2 or 3 (preferably Python 3.6 or later)
2. Recommended: install and run a virtual environment
3. Install the Scikit-Learn library and its dependencies (see http://scikit-learn.org/stable/install.html (http://scikit-learn.org/stable/install.html)).
4. Install Jupyter Notebook and its dependencies

Endnotes

1. Underwood, Ted. “Identifying diction that characterizes an author or genre: why Dunning’s may not be the best method,” *The Stone and the Shell*, November 9, 2011. https://tedunderwood.com/2011/11/09/identifying-the-terms-that-characterize-an-author-or-genre-why-dunnings-may-not-be-the-best-method/ (https://tedunderwood.com/2011/11/09/identifying-the-terms-that-characterize-an-author-or-genre-why-dunnings-may-not-be-the-best-method/) ↩

2. Bennett, Jessica, and Amisha Padnani. “Overlooked,” March 8, 2018. https://www.nytimes.com/interactive/2018/obituaries/overlooked.html (https://www.nytimes.com/interactive/2018/obituaries/overlooked.html) ↩

3. This dataset is from a version of *The New York Times* “On This Day” website that hasn’t been updated since January 31, 2011, and it has been replaced by a newer, sleeker blog located at https://learning.blogs.nytimes.com/on-this-day/ (https://learning.blogs.nytimes.com/on-this-day/). What’s left on the older “On This Day” Website is a static .html file for each day of the year (0101.html, 0102.html, etc.), including a static page for February 29th (0229.html). Content appears to have been overwritten whenever it was last updated, so there are no archives of content by year. Presumably, the “On This Day” entries for January 1 - January 31 were last updated on their corresponding days in 2011. Meanwhile, February 1 - December 31 were probably last updated on their corresponding days in 2010. The page representing February 29 was probably last updated on February 29, 2008. ↩

4. Spärck Jones, Karen. “A Statistical Interpretation of Term Specificity and Its Application in Retrieval.” *Journal of Documentation* vol. 28, no. 1 (1972): 16. ↩

5. Bowles, Nellie. “Overlooked No More: Karen Spärck Jones, Who Established the Basis for Search Engines” *The New York Times*, January 2, 2019. https://www.nytimes.com/2019/01/02/obituaries/karen-sparck-jones-overlooked.html (https://www.nytimes.com/2019/01/02/obituaries/karen-sparck-jones-overlooked.html) ↩

6. “Nellie Bly, Journalist, Dies of Pneumonia” *The New York Times*, January 28, 1922, 11. https://www.nytimes.com (https://www.nytimes.com) ↩

7. Documentation for TfidfVectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) ↩

8. Schmidt, Ben. “Do Digital Humanists Need to Understand Algorithms?” *Debates in the Digital Humanities 2016*. Online edition. (Minneapois: University of Minnesota Press): n.p. http://dhdebates.gc.cuny.edu/debates/text/99 (http://dhdebates.gc.cuny.edu/debates/text/99) ↩

9. van Rossum, Guido, Barry Warsaw, and Nick Coghlan. “PEP 8 – Style Guide for Python Code.” July 5, 2001. Updated July 2013. https://www.python.org/dev/peps/pep-0008/ (https://www.python.org/dev/peps/pep-0008/) ↩

10. "Ida M. Tarbell, 86, Dies in Bridgeport" *The New York Times*, January 7, 1944, 17. https://www.nytimes.com (https://www.nytimes.com); "Nellie Bly, Journalist, Dies of Pneumonia" *The New York Times*, January 28, 1922, 11. https://www.nytimes.com (https://www.nytimes.com); "W. E. B. DuBois Dies in Ghana; Negro Leader and Author, 95" *The New York Times*, August 28, 1963, 27. https://www.nytimes.com (https://www.nytimes.com); Whitman, Alden. "Upton Sinclair, Author, Dead; Crusader for Social Justice, 90" *The New York Times*, November 26, 1968, 1, 34. https://www.nytimes.com (https://www.nytimes.com); "Willa Cather Dies; Noted Novelist, 70" *The New York Times*, April 25, 1947, 21. https://www.nytimes.com (https://www.nytimes.com) ↩

11. Stray, Jonathan, and Julian Burgess. "A Full-text Visualization of the Iraq War Logs," December 10, 2010 (Update April 2012). http://jonathanstray.com/a-full-text-visualization-of-the-iraq-war-logs (http://jonathanstray.com/a-full-text-visualization-of-the-iraq-war-logs) ↩

12. Manning, C.D., P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. (Cambridge: Cambridge University Press, 2008): 118-120. ↩

13. Beckman, Milo. "These Are The Phrases Each GOP Candidate Repeats Most," *FiveThirtyEight*, March 10, 2016. https://fivethirtyeight.com/features/these-are-the-phrases-each-gop-candidate-repeats-most/ (https://fivethirtyeight.com/features/these-are-the-phrases-each-gop-candidate-repeats-most/) ↩

14. Bondi, Marina, and Mike Scott, eds. *Keyness in Texts*. (Philadelphia: John Benjamins, 2010). ↩

15. **Tf-idf** is not typically a recommended pre-processing step when generating topic models. See https://datascience.stackexchange.com/questions/21950/why-we-should-not-feed-lda-with-tfidf (https://datascience.stackexchange.com/questions/21950/why-we-should-not-feed-lda-with-tfidf) ↩

16. Schmidt, Ben. "Words Alone: Dismantling Topic Models in the Humanities," *Journal of Digital Humanities*. Vol. 2, No. 1 (2012): n.p. http://journalofdigitalhumanities.org/2-1/words-alone-by-benjamin-m-schmidt/ (http://journalofdigitalhumanities.org/2-1/words-alone-by-benjamin-m-schmidt/) ↩

17. Mihalcea, Rada, and Paul Tarau. "Textrank: Bringing order into text." In *Proceedings of the 2004 conference on empirical methods in natural language processing*. 2004. ↩

ABOUT THE AUTHOR

Matthew J. Lavin is a Clinical Assistant Professor of English and Director of the Digital Media Lab at the University of Pittsburgh. His current scholarship and teaching focus on book history, cultural analytics, turn-of-the-twentieth-century U.S. literature and culture.

SUGGESTED CITATION

Matthew J. Lavin, "Analyzing Documents with TF-IDF," *Programming Historian* 8 (2019), https://doi.org/10.46430/phen0082.

ISSN 2397-2068 (English) (/)

ISSN 2517-5769 (Spanish) (/es)

ISSN 2631-9462 (French) (/fr)

ISSN 2753-9296 (Portuguese) (/pt)