

TF•IDF (part 2)

This lesson continues our learning of TF•IDF, specifically how to use the powerful scikit-learn library to perform a TF•IDF analysis. Not only will you learn how to use the library but also how it calculates TF and IDF. Our goal for this lesson is to build a tf•idf representation using scikit-learn. You will learn how it differs from your implementation in the previous lesson.

In order to get the most out of this lesson, you might want to make a copy of the tf•idf lesson so that you can make the requested edits without disturbing your original workbook.

Welcome to Scikit-learn



The Python library [scikit-learn](#) (pronounced s-eye (like 'science') kit) is a set of modules (think Python classes) that can be used to build data analysis and machine learning software. SciKit-learn uses Numpy, SciPy and Matplotlib -- all libraries you worked with in INFO 490. It's another mountain to climb, but we will explore it slowly and try to see as much of it as possible.

The Python package uses the prefix `sklearn`, so we may use either `sklearn`, `sci-kit`, or `scikit-learn` to reference the software.

Feature Vectors

This begins our journey into learning some of the vocabulary of machine learning. When you use the `collections.Counter` class to count occurrences of words, you are building a simple **feature vector**. A feature vector contains information describing an object's more important characteristics. The vector is usually used as input into a machine learning algorithm.

Another closely related term, **feature engineering** is the process of determining which features (attributes, 'columns', traits) to use that best represent the input.

Feature engineering may involve introducing new 'features' that are the result of human intervention, algorithm processing, or both. A feature vector is almost always numerical and there is a mapping between the original item and its numerical representation.

For now you can think of a feature vector as a set of columns (attributes) for a row (an instance/observation). We will be using feature vectors throughout this class. Luckily, scikit-learn has a sub-module that specializes in building feature vectors (a numerical representation) for text documents.

Feature-1	Feature-2	Feature-3	Feature-4	Feature-n	
x_1^1	x_2^1	x_3^1	x_4^1	x_n^1	Sample-1
x_1^2	x_2^2	x_3^2	x_4^2	x_n^2	Sample-2
x_1^3	x_2^3	x_3^3	x_4^3	x_n^3	Sample-3
...	
x_1^m	x_2^m	x_3^m	x_4^m	x_n^m	Sample-m

Vectorization

Scikit uses the word *vectorization* as the general process of turning a collection of text documents into numerical feature vectors. We will start with the `CountVectorizer` class that builds a fancy version of the `collections.Counter`.

The `CountVectorizer` Class



We will start our tour using a simple, reduced four chapter story. The function `get_corpus` returns this data .

```
def get_corpus():
    c1 = "Do you like Green eggs and ham"
    c2 = "I do not like them Sam I am I do not like Green eggs and ham"
    c3 = "Would you like them Here or there"
    c4 = "I would not like them Here or there I would not like them Anywhere"
    return [c1, c2, c3, c4]
```

Let's now see how we can use the CountVectorizer class with that corpus:

```
from sklearn.feature_extraction.text import CountVectorizer

def cv_demo1():

    corpus = get_corpus()

    # normalize all the words to lowercase
    cvec = CountVectorizer(lowercase=True)

    # convert the documents into a document-term matrix
    doc_term_matrix = cvec.fit_transform(corpus)

    # Let's now see how we can use the CountVectorizer class with that corpus:
    print(cvec.get_feature_names())

    # get the counts
    print(doc_term_matrix.toarray())

cv_demo1()
```

Your output should match the following:

```
['am', 'and', 'anywhere', 'do', 'eggs', 'green', 'ham', 'here', 'like', 'not', 'or',
'sam', 'them', 'there', 'would', 'you']
[[0 1 0 1 1 1 1 0 1 0 0 0 0 0 0 1]
 [1 1 0 2 1 1 1 0 2 2 0 1 1 0 0 0]
 [0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1]
 [0 0 1 0 0 0 0 1 2 2 1 0 2 1 2 0]]
```

The one issue is that the token 'I' is missing. The CountVectorizer ignores tokens less than 2 characters long by default. You can pass in your own tokenizer (among many parameters) to customize how it parses the text.

Update the code below (see the comments), and run `cv_demo2` (note that the function has been updated to return both the tokens and the document matrix). Note we are passing in the same tokenizer you wrote (that had a default of `min_length=0`). You can re-implement it or just copy&paste it from your previous `tf-idf` lesson.

```
# add in the function split_into_tokens with the following instructions
# parameters: data, normalize, min_length (in that order) # returns a list
of words/tokens
# normalize has a default value of True
# min_length has a default value of 0
# if normalize, make tokens lowercase
# only returns tokens longer than min_length
def cv_demo2():

    corpus = get_corpus()

    # pass in our own tokenizer
    cvec = CountVectorizer(tokenizer=split_into_tokens)

    # convert the documents into a document-term matrix
    doc_term_matrix = cvec.fit_transform(corpus)

    # get the terms found in the corpus
    tokens = cvec.get_feature_names()

    return doc_term_matrix, tokens

dtm, tokens = cv_demo2()
print(tokens)
print(dtm.toarray())
$_
```

After you re-run the code, you should get the following output.

```
['am', 'and', 'anywhere', 'do', 'eggs', 'green', 'ham', 'here', 'i', 'like', 'not',
'or', 'sam', 'them', 'there', 'would', 'you']
[[0 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 1]
 [1 1 0 2 1 1 1 0 3 2 2 0 1 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 1 0 1 0 1 1 1 1]
 [0 0 1 0 0 0 0 1 2 2 2 1 0 2 1 2 0]]
```

'Visualize' with Pandas

We can use Pandas to help make this a bit easier to look at. We are also going to reuse `cv_demo2` to allow us to focus on the new code.

```
import pandas as pd

def word_matrix_to_df(wm, feature_names):
    # create an index for each row
    doc_names = ['Doc{:d}'.format(idx+1) for idx, _ in enumerate(wm)]
    df = pd.DataFrame(data=wm.toarray(), index=doc_names, columns=feature_names)
    return df

def cv_demo3():

    doc_term_matrix, tokens = cv_demo2()
    df = word_matrix_to_df(doc_term_matrix, tokens)

    return df

df = cv_demo3()
print(df.head())
$_
```

You should see data that matches the table below:

	17 Terms	am	and	any where	do	eggs	green	ham	here	i	like	not	or	sam	them	there	would	you	TOTAL
doc1	7		1		1	1	1	1			1							1	7
doc2	11	1	1		2	1	1	1		3	2	2		1	1				16
doc3	7								1		1		1		1	1	1	1	7
doc4	9			1					1	2	2	2	1		2	1	2		14
TF (raw)		1	2	1	3	2	2	2	2	5	6	4	2	1	4	2	3	2	44

Be sure to take a pause to understand everything that has been done so far. Also take a look at sklearn's [documentation](#). The `CountVectorizer` allows you to customize many of the parsing routines:

```
cvec = CountVectorizer(tokenizer=split_into_tokens, # custom tokenizer
                       ngram_range=(1, 2),        # both single and bi-grams
                       stop_words=['and', 'but'])   # custom stop words
```

TfidfTransformer

Now that we have a vector of word counts, we want to transform those counts into a TF•IDF matrix (essentially the same process as the previous lesson). We will use sklearn's TfidfTransformer class. It is meant to work side-by-side with the CountVectorizer.

```
from sklearn.feature_extraction.text import TfidfTransformer
def cv_demo_idf():

    # get the data from the CountVectorizer
    doc_term_matrix, tokens = cv_demo2()

    # create the tf•idf transformer
    tfidf_transformer=TfidfTransformer()

    # transform the doc_term_matrix into TF•IDF
    tfidf_transformer.fit(doc_term_matrix)

    # make it a dataframe for easy viewing
    df = pd.DataFrame(tfidf_transformer.idf_,
                      index=tokens, columns=["idf_weights"])

    # sort descending
    df.sort_values(by=['idf_weights'], inplace=True, ascending=False)

    return df

df = cv_demo_idf()
print(df.head(20))
```

For easy comparison, here's the same IDF values from the previous lesson. What do you notice?

DOC CNT (N)	4	am	and	any where	do	eggs	green	ham	here	i	like	not	or	sam	them	there	would	you
DFC (n)		1	2	1	2	2	2	2	2	2	4	2	2	1	3	1	2	2
IDF: log(N/n)		1.386	0.693	1.386	0.693	0.693	0.693	0.693	0.693	0.693	0.000	0.693	0.693	1.386	0.288	1.386	0.693	0.693
IDF: sci-kit		1.916	1.511	1.916	1.511	1.511	1.511	1.511	1.511	1.511	1.000	1.511	1.511	1.916	1.223	1.916	1.511	1.511

The calculated values do not match (e.g. for the token 'i', we calculated 0.693; and sklearn's value is 1.511). However the relative magnitudes of the words seems about correct. In order to get the numbers to match, you will need to adjust *both* your TF and IDF calculations and adjust the calculations that TfidfTransformer uses.

One of the key points of this lesson is to understand and confirm the output of various algorithms. It's almost always necessary to confirm or validate your implementation (or your use of another library) with a reference. Let's go through all the adjustments.

IDF Formula

We used the standard formula for idf as

$$idf_j = \log\left(\frac{n}{df_j}\right)$$

Sklearn's default idf formula is **1 + math.log((N+1)/(n+1))** where N is total number of documents and n is the number of documents for the term. They add one to prevent division by zero and prevent taking the log of zero. When setting the named parameter `smooth_idf` to `False`, the formula becomes `1 + ln(N/n)`.

```
# will use 1 + ln(N/n)
tfidf_transformer=TfidfTransformer(smooth_idf=False)
```

If you go back to the previous lesson (and you should) and update the IDF formula to `1 + ln((N+1)/(n + 1))`, you should get 1.5108 for the word 'i'. Make sure you set '`min_length`' to 0 to capture 'i'.

[✓] Matching IDF

Let's now print out the TF•IDF values and see where we are:

```
def cv_demo_tf_idf():

    doc_term_matrix, tokens = cv_demo2()
    tfidf_transformer=TfidfTransformer(smooth_idf=True)

    # learn the IDF vector
    tfidf_transformer.fit(doc_term_matrix)
    idf = tfidf_transformer.idf_

    # transform the count matrix to tf-idf
    tf_idf_vector = tfidf_transformer.transform(doc_term_matrix)
    print(tf_idf_vector)

$ _
```

You should see the following, when you run the function (below is just a sample of the output):

```
(0, 3)    0.39411340505265774 (doc 0, token w/index 3 has tf•idf = 0.3941)
(0, 1)    0.39411340505265774
(1, 13)   0.1568972451918658
(1, 12)   0.24580985322181814 (doc 1, token w/index 12 has tf•idf = 0.2458)
```

This isn't too easy to read, we can use Pandas to help make this easy.

```
def cv_demo_pd_tf_idf():

    doc_term_matrix, tokens = cv_demo2()
    tfidf_transformer=TfidfTransformer(smooth_idf=True)

    # learn the IDF vector
    tfidf_transformer.fit(doc_term_matrix)
    idf = tfidf_transformer.idf_

    # transform the count matrix to tf-idf
    tf_idf_vector = tfidf_transformer.transform(doc_term_matrix)

    # print out the values
    # for the token 'i' in the second document
    token = 'i'
    doc = 1
    df_idf = pd.DataFrame(idf, index=tokens, columns=["idf_weights"])
    df_idf.sort_values(by=['idf_weights'], inplace=True, ascending=False)
    idf_token = df_idf.loc[token]['idf_weights']

    doc_vector = tf_idf_vector[doc]
    df_tfidf = pd.DataFrame(doc_vector.T.todense(), index=tokens, columns=["tfidf"])
    df_tfidf.sort_values(by=["tfidf"], ascending=False, inplace=True)
    tfidf_token = df_tfidf.loc[token]['tfidf']

    # tfidf = tf * idf
    tf_token = tfidf_token / idf_token
    print('TF      {s} {2.4f}'.format(token, tf_token))
    print('IDF      {s} {2.4f}'.format(token, idf_token))
    print('TFIDF {s} {2.4f}'.format(token, tfidf_token))

$ _
```

Once you run the code you should see the following for the second document, token 'i':

```
TF      i 0.3845
IDF      i 1.5108
TFIDF i 0.5814
```

However, in the previous lesson (with the updated IDF formula), you will see the following:

```
TF:      0.1875
IDF:      1.5108
TFIDF:    0.28328
```

[X] Matching TF (NO!!!)

Since the IDF values match and that we can only really see the end result (TF • IDF), we can infer that the issue (two actually) must be with the TF formula.

TF Formula

By default sklearn is using the raw term counts. You can adjust this slightly by setting the named parameter `sublinear_tf=True`. This changes the formula to $1 + \text{math.log}(tf)$ where tf

is the number of occurrences of the term in the document.

We used a normalized term count (which was divided by the length of the document). There's no option in sklearn for this option. So let's update our TF calculation to be the following:

`1 + math.log(tf)` to match sci-kit learn's.

Where **tf** is the count of how many times term appear in the current document

Normalization

By Default, the final TF•IDF calculations are normalized using the L2 norm (a.k.a. Euclidean).

The L2 norm is just the square root of the sum of the squared vector values:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}}$$

As an example, let's look at how L2 normalization is done using TF•IDF values for the word 'i':

```
L2_norm = sqrt(tfidf('am')2 + tfidf('and')2 + .. + tfidf('i')2 + .. +  
               tfidf('would')2 + tfidf('you')2)  
  
tfidf_norm('i') = tfidf('i')/L2_norm
```

The nice thing about applying L2 normalization is that it puts different features on the same scale.

Also, mathematically, the L2 Norm of the resulting normalized values is 1.0. That is if you took the square root of the sum of the squared tfidf norm values (e.g. `tfidf_norm('i')`), it would be 1.0.

The L1 norm uses the sum of the absolute values (a.k.a. Manhattan distance). We will see more examples in the future of using L2 normalization. A closely related topic, L2 regularization will be discussed later as well.

We will NOT do this to our code in the previous lesson -- it would require a lot of code refactoring. Instead, we will turn normalization off when we create a `TfidfTransformer`.

In order to turn off normalization (remember our goal is to get sklearn's output to match our output from the previous lesson), we can set the `norm` parameter to `None`:

```
TfidfTransformer(smooth_idf=True, sublinear_tf=True, norm=None)
```

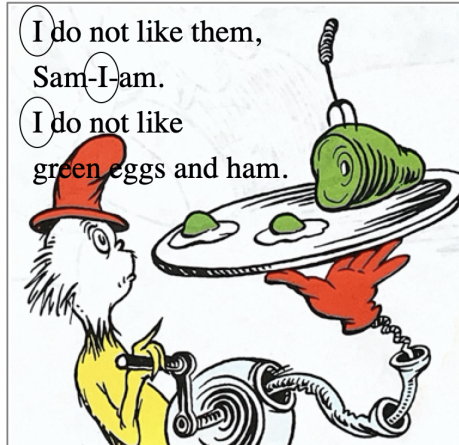
Now we should see the following after running `cv_demo_pd_tf_idf`. These numbers will match the output of your code in the previous lesson (if you made to two respective changes).

```
TF      i 2.0986
IDF      i 1.5108
TFIDF i 3.1706
```

[✓] Matching IDF

[✓] Matching TF

Here's a quick summary of what we calculated (using the token 'i' for the example):



Document: 16 words
Term: "I"
TermCount: 3 (n)
Total Documents: 4 (N)
Documents containing "I": 2

Traditional Calculation

TF: 0.1875 i.e. $3/16$
IDF: 0.693 i.e. $\ln(4/2)$
TF•IDF: 0.130

Sklearn Calculation
norm=None
smooth_idf=True
sublinear_tf=True

TF: 2.099 i.e. $1 + \ln(3)$
IDF: 1.511 i.e. $1 + \ln((4+1)/(2+1))$
TF•IDF: 3.171

Please for the love of your brain, take a break to ensure you understand the above image and its numbers. Its so easy to gloss over it and convince ourselves we 'get it'.

The TfidfVectorizer

Another class provided by scikit-learn is the `TfidfVectorizer` class. This class creates its own `CountVectorizer` to use. You use it essentially the same, but it's bit more compact:

```
cv = TfidfVectorizer(smooth_idf=True, use_idf=True, tokenizer=split_into_tokens,
norm=None)
tfidf = cv.fit_transform(corpus)
tokens = cv.get_feature_names()
idf = cv.idf_
values = tfidf.todense().tolist()
```

Fit vs Transform

As we have seen, there are both `fit` and `transform` methods on the `TfidfTransformer` (there's `fit_transform` too, but that's just the combination of the two). The method `fit` attempts to build (and train) a model based on the data given to it. In this specific case, we give it the documents, and 'fit' builds an idf vector. It essentially transforms the raw data into normalized data.

Usually for sklearn, `fit` is associated with the *training* phase in machine learning. Fitting is similar to finding the best parameters for a model. These parameters are then used to transform the data. As we march forward, this will become a bit clearer.

If 'fitting' attempts to train a model and/or parameters, the `transform` method then applies what was fitted to incoming data. For this specific situation, transform creates the TF•IDF vectors. However, for sklearn, transform is associated with the *testing* phase in machine learning. That is the part where we test the accuracy of the model.

This will become much clearer when we get to machine learning. TF•IDF is actually preparing our data to be used (possibly) with other machine learning algorithms. Since computers can't process text directly, changing them into numeric vectors is an essential preparation stage for almost all machine learning algorithms (that work on text).

“ ***Machine Learner's Log:*** When building a machine learning 'model', you will typically split your data into two parts: a **training set** and a **validation set**. The 'fit' method works on the training set. The 'transform' method works on the validation set. The testing set helps determine the accuracy of the model (or the fitting part).

Sklearn is trying to use the fit/transform vocabulary in a non machine learning situation. For us, it is building a vector space model (VSM) or a sparse feature set.

The Sparse Matrix

Perhaps you are wondering exactly what is the thing/object/data returned from either `transform` or `fit_transform`?

```
tf_idf_vector = tfidf_transformer.transform(doc_term_matrix)
```

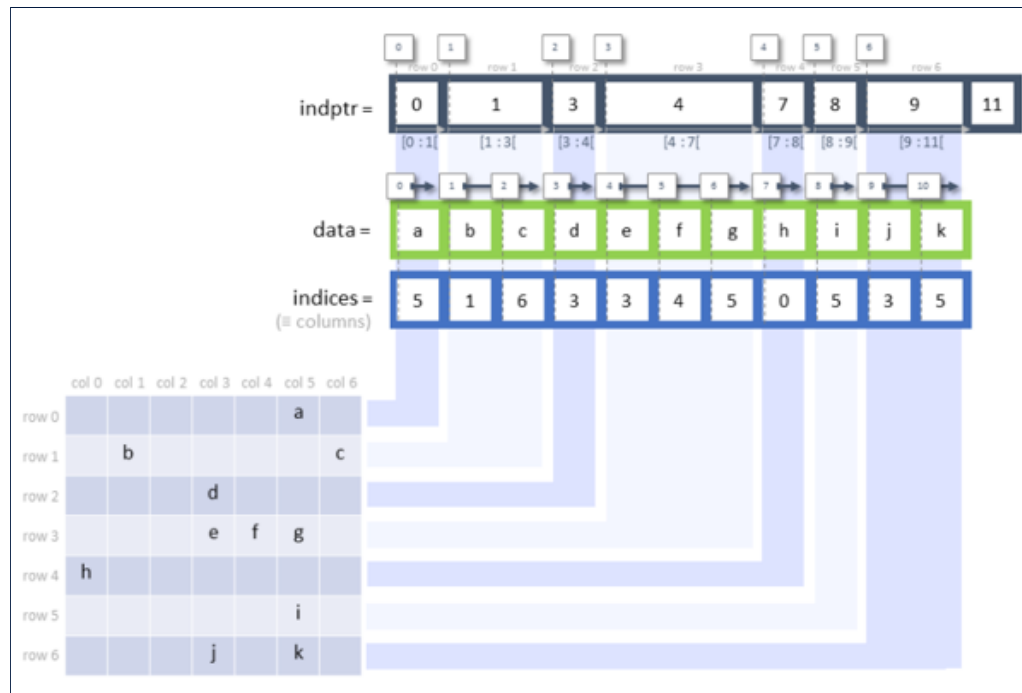
It is actually a *sparse* matrix ([see sklearn doc](#)). As you can imagine, as the set of documents become larger, there will a large set of words that only belong to a small set of documents;

id	mean	entered	bars	charlotte	mississ	maas	arwen	grris	whitman	reported	shar	hwy	august
avg0.txt	0.239441	0	0.133457	0.195263	0	0.237029	0	0.195263	0.237029	0.140004	0.195263	0.237029	0
avg1.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg2.txt	0	0.195263	0	0	0	0	0	0	0	0	0	0	0.177461
avg3.txt	0	0	0	0	0	0	0	0	0	0	0	0	0.140004
avg4.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg5.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg6.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg7.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg8.txt	0	0.133457	0	0	0	0	0	0	0	0	0	0	0
avg9.txt	0	0	0.195263	0	0	0	0	0	0.140004	0	0	0.237029	0
avg10.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg11.txt	0	0.239441	0	0	0	0	0	0	0	0	0	0	0
avg12.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg13.txt	0	0	0	0	0.133457	0	0.127589	0	0	0	0	0	0
avg14.txt	0	0	0	0	0	0	0	0	0.140004	0	0	0	0
avg15.txt	0	0	0	0	0	0	0.117966	0	0.117966	0	0	0	0
avg16.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg17.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
avg18.txt	0	0	0.239441	0	0	0	0.234761	0	0	0	0	0	0
avg19.txt	0	0	0	0	0.133457	0	0	0	0	0	0	0	0
avg20.txt	0	0	0	0	0	0	0.127589	0	0	0	0	0.140004	0
avg21.txt	0	0	0	0	0	0.133457	0	0	0	0	0	0	0
avg22.txt	0.177461	0	0	0	0	0	0	0	0	0	0	0	0
avg23.txt	0	0.239441	0	0	0	0	0.134701	0	0	0	0	0	0
avg24.txt	0	0	0	0	0	0	0	0	0	0	0	0	0

hence, the tf-idf matrix will contain mostly zeros.

The sklearn sparse matrix efficiently stores the necessary data. Many of sklearn's algorithms can work with a sparse matrix as well as regular Numpy vectors.

“ **Coder's Log:** As you become more familiar with using sparse matrices, you may come across references to `indptr`, `data`, `indices`. The image below helps depict sklearn's efficient (via indirection) implementation of sparse matrices:



'Visualizing' the data

It's easy to get all the non-zero values out of a sparse matrix. The **Coordinate** format allows you to get both the index (e.g row or row,column -- for 2D) and the values. The following section of code can help to understand and work with the sparse matrix data:

```
def dump_sparse_matrix():

    from sklearn.feature_extraction.text import TfidfVectorizer
    vec = TfidfVectorizer(use_idf=True)

    corpus = ["another day of rain; rain rain go away, comeback another day"]
    matrix = vec.fit_transform(corpus)
    print(matrix.shape)
    print(vec.idf_) # all 1's (there's only 1 document)

    coo_format = matrix.tocoo()
    print(coo_format.col)
    print(coo_format.data)
    tuples = zip(coo_format.col, coo_format.data)
    in_order = sorted(tuples, key=lambda x: (x[1], x[0]), reverse=True)
    features = vec.get_feature_names() # the unique words
    print(features)
    for score in in_order:
        idx = score[0]
        word = features[idx]
        print("{:10s} tfidf:".format(word), score)

dump_sparse_matrix()
$ _
```

Pandas to the rescue

When you use libraries like Pandas, you can easily convert these sparse matrices into regular Numpy arrays:

```
df = pd.DataFrame(data=matrix.toarray(), columns=vec.get_feature_names())
```

Gensim

Another popular library for working with text is [gensim](#). It also has TF-IDF implementations as well. Although we will not be using this library now, it's important to know that it is an option. We will use gensim very soon!

```
corpus = [tokenize(doc) for doc in corpus]
lexicon = gensim.corpora.Dictionary(corpus)
tfidf = gensim.models.TfidfModel(dictionary=lexicon, normalize=True)
vectors = [tfidf[lexicon.doc2bow(doc)] for doc in corpus]
```

Clustering and Distance Document Similarity

Now that we have a matrix where the rows are documents (document vectors) of normalized TF•IDF values we can "easily" compare two documents and see how similar (or different they are).

The distance between two document vectors is the essence behind using TF•IDF for document clustering (an unsupervised machine learning algorithm), as well as trying to retrieve relevant documents by search terms. Your search terms become a 'document' and gets converted to a vector. Distances are then calculated for each document to the query/search vector.

Much of this will be covered in a separate lesson on distance metrics. There's a lot more to cover.

Review

🚧 Before you go, you should **know**:

What does CountVectorizer do	⊕
What does TfidfTransformer do	⊕
The Meaning of sklearn's fit function	⊕
The Meaning of sklearn's transform function	⊕

Lesson Assignment

For this lesson, if you have been following along and implementing the requested functionality, you are done!

The End!