Feature Engineering for Text Representation

All the code examples showcased in this chapter are available on the book's official GitHub repository which you can access here: https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition.

Understanding Text Data

I'm sure all of you must be having a fair idea of what textual data comprises of by now considering we have covered three chapters on the same! Do remember you can always have text data in the form of structured data attributes, but usually those fall under the umbrella of structured, categorical data. In this scenario, we are talking about free flowing text in the form of words, phrases, sentences and entire documents. Essentially, we do have some syntactic structure like words make phrases, phrases make sentences which in turn make paragraphs. However, there is no inherent structure to text documents because you can have a wide variety of words which can vary across documents and each sentence will also be of variable length as compared to a fixed number of data dimensions in structured datasets. This chapter itself is a perfect example of text data! An important question might be that how can we represent text data then which can be easy for machines to comprehend and understand it easily?

A ***Vector Space Model*** is a concept and model which is very useful in case we are dealing with textual data and is very popular in information retrieval and document ranking. The Vector Space Model is also known as Term Vector Model and is defined as a mathematical and algebraic model for transforming and representing text documents as numeric vectors of specific terms which form the vector dimensions. Mathematically this can be defines as follows. Consider we have a document *D* in a document vector space *VS*. The number of dimensions or columns for each document will be the total number of distinct terms or words for all documents in the vector space. Hence the vector space can be denotes as

$$VS = \{W_1, W_2, \ldots, W_n\}$$

where there are *n* distinct words across all documents. Now we can represent document *D* in this vector space as

$$D = \{w_{D1}, w_{D2}, \ldots, w_{Dn}\}$$

where w_{Dn} denotes the weight for word n in document *D*. This weight is a numeric value and can be anything ranging from the frequency of that word in the document, the average frequency of occurrence, embedding weights or even the tf-idf weight which we will be discussing shortly.

An important thing to remember for feature extraction and engieering is that once we build a feature engineering model using some transformations and mathematical operations, we need to make sure we reuse the same process when extracting features from new documents to be predicted and not rebuild the whole algorithm again based on the new documents.

Building a Text Corpus

We need a text corpus to work on and demonstrate different feature engineering and representation methodologies. To keep things simple and easy to understand, we will build a simple text corpus in this section. To get started, load up the following dependencies in your Jupyter notebook.

```
import pandas as pd
import numpy as np
import re
import nltk
import matplotlib.pyplot as plt

pd.options.display.max_colwidth = 200
%matplotlib inline
```

Let's now build a sample corpus of documents on which we will run most of our analyses in this chapter. A corpus is typically a collection of text documents usually belonging to one or more subjects or topics. The following code helps us create this corpus.

```
# building a corpus of documents
corpus = ['The sky is blue and beautiful.',
          'Love this blue and beautiful sky!',
          'The quick brown fox jumps over the lazy dog.',
          "A king's breakfast has sausages, ham, bacon, eggs, toast and beans",
          'I love green eggs, ham, sausages and bacon!',
          'The brown fox is quick and the blue dog is lazy!',
          'The sky is very blue and the sky is very beautiful today',
          'The dog is lazy but the brown fox is quick!'
]
labels = ['weather', 'weather', 'animals', 'food', 'food', 'animals',
'weather', 'animals']

corpus = np.array(corpus)
corpus_df = pd.DataFrame({'Document': corpus,
                          'Category': labels})
corpus_df = corpus_df[['Document', 'Category']]
corpus_df
```

	Document	Category
0	The sky is blue and beautiful.	weather
1	Love this blue and beautiful sky!	weather
2	The quick brown fox jumps over the lazy dog.	animals
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food
4	I love green eggs, ham, sausages and bacon!	food
5	The brown fox is quick and the blue dog is lazy!	animals
6	The sky is very blue and the sky is very beautiful today	weather
7	The dog is lazy but the brown fox is quick!	animals

Figure 4-1. *Our sample text corpus*

You can see our sample text corpus depicted in the output shown by Figure 4-1. You can also observe that we have taken a few sample text documents belonging to different categories for our toy corpus. Before we talk about feature engineering, we need to do some data pre-processing and wrangling to remove unnecessary characters, symbols and tokens.

Pre-processing our Text Corpus

There can be multiple ways of cleaning and pre-processing textual data. In the following points, we highlight some of the most important ones which are used heavily in Natural Language Processing (NLP) pipelines. A lot of this would be a refresher for you if you have already read Chapter 3

- **Removing tags:** Our text often contains unnecessary content like HTML tags, which do not add much value when analyzing text. The BeautifulSoup library does an excellent job in providing necessary functions for this.
- **Removing accented characters:** In any text corpus, especially if you are dealing with the English language, often you might be dealing with accented characters\letters. Hence we need to make sure that these characters are converted and standardized into ASCII characters. A simple example would be converting **é** to **e**.

- **Expanding contractions:** In the English language, contractions are basically shortened versions of words or syllables. These shortened versions of existing words or phrases are created by removing specific letters and sounds. Examples would be, *do not* to *don't* and *I would* to *I'd*. Converting each contraction to its expanded, original form often helps with text standardization.
- **Removing special characters:** Special characters and symbols which are usually non alphanumeric characters often add to the extra noise in unstructured text. More than often, simple regular expressions (regexes) can be used to achieve this.
- **Stemming and lemmatization:** Word stems are usually the base form of possible words that can be created by attaching *affixes* like *prefixes* and*suffixes* to the stem to create new words. This is known as inflection. The reverse process of obtaining the base form of a word is known as *stemming*. A simple example are the words *WATCHES*, *WATCHING*, and *WATCHED*. They have the word root stem *WATCH* as the base form. *Lemmatization* is very similar to stemming, where we remove word affixes to get to the base form of a word. However the base form in this case is known as the root word but not the root stem. The difference being that the root word is always a lexicographically correct word (present in the dictionary) but the root stem may not be so.
- **Removing stopwords:** Words which have little or no significance especially when constructing meaningful features from text are known as stopwords or stop words. These are usually words that end up having the maximum frequency if you do a simple term or word frequency in a corpus. Words like *a*, *an*, *the*, *and* so on are considered to be stopwords. There is no universal stopword list but we use a standard English language stopwords list from nltk. You can also add your own domain specific stopwords as needed.

Besides this you can also do other standard operations like tokenization, removing extra whitespaces, text lower casing and more advanced operations like spelling corrections, grammatical error corrections, removing repeated characters and so on. If you are interested, check our the detailed section on text pre-processing and wrangling covered in Chapter 3.

Since the focus of this article is on feature engineering, we will build a simple text pre-processor which focuses on removing special characters, extra whitespaces, digits, stopwords and lower casing the text corpus.

```
wpt = nltk.WordPunctTokenizer()
stop_words = nltk.corpus.stopwords.words('english')

def normalize_document(doc):
    # lower case and remove special characters\whitespaces
```

```
    doc = re.sub(r'[^a-zA-Z\s]', '', doc, re.I|re.A)
    doc = doc.lower()
    doc = doc.strip()
    # tokenize document
    tokens = wpt.tokenize(doc)
    # filter stopwords out of document
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # re-create document from filtered tokens
    doc = ' '.join(filtered_tokens)
    return doc

normalize_corpus = np.vectorize(normalize_document)
```

Once we have our basic pre-processing pipeline ready, let's apply the same to our sample corpus so we can use it for feature engineering.

```
norm_corpus = normalize_corpus(corpus)
norm_corpus
```

```
array(['sky blue beautiful', 'love blue beautiful sky',
       'quick brown fox jumps lazy dog',
       'kings breakfast sausages ham bacon eggs toast beans',
       'love green eggs ham sausages bacon',
       'brown fox quick blue dog lazy', 'sky blue sky beautiful today',
       'dog lazy brown fox quick'], dtype='<U51')
```

The above output should give you a clear view of how each of our sample documents looks like after pre-processing. Let's explore various feature engineering techniques now!

Traditional Feature Engineering Models

Traditional (count-based) feature engineering strategies for textual data involve models belonging to a family of models popularly known as the Bag of Words model in general. This includes term frequencies, TF-IDF (term frequency-inverse document frequency), N-grams, Topic Models and so on. While they are effective methods for extracting features from text, due to the inherent nature of the model being just a bag of unstructured words, we lose additional information like the semantics, structure, sequence and context around nearby words in each text document. There are more advanced models which take care of these aspects and we will be covering them in a subsequent section in this chapter. The traditional feature engineering models are built using mathematical and statistical methodologies. We will take a look at some of these models and apply the same on our sample corpus.

Bag of Words Model

This is perhaps the most simple vector space representational model for unstructured text. A vector space model is simply a mathematical model to represent unstructured text (or any other data) as numeric vectors, such that each dimension of the vector is a specific feature\attribute. The bag of words

model represents each text document as a numeric vector where each dimension is a specific word from the corpus and the value could be its frequency in the document, occurrence (denoted by 1 or 0) or even weighted values. The model's name is such because each document is represented literally as a 'bag' of its own words, disregarding word orders, sequences and grammar.

```
from sklearn.feature_extraction.text import CountVectorizer
# get bag of words features in sparse format
cv = CountVectorizer(min_df=0., max_df=1.)
cv_matrix = cv.fit_transform(norm_corpus)
cv_matrix
```

```
<8x20 sparse matrix of type '<class 'numpy.int64'>'
       with 42 stored elements in Compressed Sparse Row format>
```

```
# view non-zero feature positions in the sparse matrix
print(cv_matrix)
```

```
  (0, 2)      1
  (0, 3)      1
  (0, 17)     1
  (1, 14)     1
  ...
  ...
  (6, 17)     2
  (7, 6)      1
  (7, 13)     1
  (7, 8)      1
  (7, 5)      1
  (7, 15)     1
```

The feature matrix is traditionally represented as a sparse matrix since the number of features increase phenomenally with each document considering each distinct word becomes a feature. The preceding output tells us for each ***(x, y)*** pair what is the total count. Here ***x*** represents a document and ***y*** represents a specific word\feature and the value is the number of times ***y*** occurs in ***x***. We can leverage the following code to view the output in a dense matrix representation.

```
# view dense representation
# warning might give a memory error if data is too big
cv_matrix = cv_matrix.toarray()
cv_matrix
```

```
array([[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0],
       [1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0],
       [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1],
```

```
       [0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0]],
dtype=int64)
```

Thus you can see that our documents have been converted into numeric vectors such that each document is represented by one vector (row) in the above feature matrix and each column represents a unique word as a feature. The following code will help represent this in a more easy to understand format.

```
# get all unique words in the corpus
vocab = cv.get_feature_names()
# show document feature vectors
pd.DataFrame(cv_matrix, columns=vocab)
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
2	0	0	0	0	0	1	1	0	1	0	0	1	0	1	0	1	0	0	0	0
3	1	1	0	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	1	0
4	1	0	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	0
5	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0
6	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	1
7	0	0	0	0	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0

Figure 4-2. *Our Bag of Words model based document feature vectors*

This should make things more clearer! You can clearly see that each column or dimension in the feature vectors represents a word from the corpus and each row represents one of our documents. The value in any cell, represents the number of times that word (represented by column) occurs in the specific document (represented by row). A simple example would be the first document has the words ***blue***, ***beautiful*** and ***sky*** occuring ***once each*** and hence the corresponding featues have a value of **1** for the first row in the preceding output. Hence if a corpus of documents consists of ***N*** unique words across all the documents, we would have an ***N-dimensional*** vector for each of the documents.

Bag of N-Grams Model

A word is just a single token, often known as a unigram or 1-gram. We already know that the Bag of Words model doesn't consider order of words. But what if we also wanted to take into account phrases or collection of words which occur in a sequence? N-grams help us achieve that. An N-gram is basically a collection of word tokens from a text document such that these tokens are contiguous and occur in a sequence. Bi-grams indicate n-grams of order 2 (two words), Tri-grams indicate n-grams of order 3 (three words), and so on. The Bag of N-Grams model is hence just an extension of the Bag of Words model so we can also leverage N-gram

based features. The following example depicts bi-gram based features in each document feature vector.

```
# you can set the n-gram range to 1,2 to get unigrams as well as bigrams
bv = CountVectorizer(ngram_range=(2,2))
bv_matrix = bv.fit_transform(norm_corpus)

bv_matrix = bv_matrix.toarray()
vocab = bv.get_feature_names()
pd.DataFrame(bv_matrix, columns=vocab)
```

	bacon eggs	beautiful sky	beautiful today	blue beautiful	blue dog	blue sky	breakfast sausages	brown fox	dog lazy	eggs ham	...	lazy dog	love blue	love green	quick blue	quick brown	sausages bacon	sausages ham	sky beautiful
0	0	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	...	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	...	1	0	0	0	1	0	0	0
3	1	0	0	0	0	0	1	0	0	0	...	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0	0	1	...	0	0	1	0	0	1	0	0
5	0	0	0	0	1	0	0	1	1	0	...	0	0	0	1	0	0	0	0
6	0	0	1	0	0	1	0	0	0	0	...	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	1	1	0	...	0	0	0	0	0	0	0	0

8 rows × 29 columns

Figure 4-3. *Bi-gram based feature vectors using the Bag of N-Grams Model*

This gives us feature vectors for our documents, where each feature consists of a bi-gram representing a sequence of two words and values represent how many times the bi-gram was present for our documents. We encourage you to play around with the ngram_range argument. Try out the above functions by setting ngram_range to (1, 3) and see the outputs!

TF-IDF model

There are some potential problems which might arise with the Bag of Words model when it is used on large corpora. Since the feature vectors are based on absolute term frequencies, there might be some terms which occur frequently across all documents and these may tend to overshadow other terms in the feature set. Especially, words which may not occur as frequently but might be more interesting and effective as features to identify specific categories for the documents. This is where TF-IDF comes into the picture. TF-IDF stands for Term Frequency-Inverse Document Frequency which is a combination of two metrics, ***term frequency (tf)*** and ***inverse document frequency (idf)***. This technique was originally developed as a metric for ranking functions for showing search engine results based on user queries and has come to be a part of information retrieval and text feature extraction now.

Let us formally define tf-idf now and look at the mathematical representations for the same before diving into its implementation. Mathematically, td-idf is the product of two metrics and can be represented as

$$tfidf = tf \times idf$$

where term frequency ***(tf)*** and inverse-document frequency ***(idf)*** represent the two metrics we just talked about. ***Term frequency*** denoted by ***tf*** is what we had computed in the bag of words model in the previous section. Term frequency in any document vector is denoted by the raw frequency value of that term in a particular document. Mathematically it can be represented as

$$tf(w, D) = f_{w_D}$$

where f_{w_D} denoted frequency for ***word w*** in ***document D*** which becomes the ***term frequency (tf)***. There are various other representations and computations for term frequency like converting frequency to a binary feature where 1 depicts the term has occurred in the document and 0 depicts the term has not occurred in the document. Sometimes you can also normalize the absolute raw frequency using logarithms or averaging the frequency. We will be using the raw frequency in our computations.

Inverse document frequency denoted by ***idf*** is the inverse of the document frequency for each term which is computed by dividing the total number of documents in our corpus by the document frequency for each term and then applying logarithmic scaling on the result. In our implementation we will be adding 1 to the document frequency for each term just to indicate that we also have one more document in our corpus which essentially has every term in the vocabulary. This is to prevent potential division by zero errors and smoothen the inverse document frequencies. We also add 1 to the result of our idf computation to avoid ignoring terms completely which might have zero idf. Mathematically, our implementation for ***idf*** can be represented by

$$idf(w, D) = 1 + log\frac{N}{1 + df(w)}$$

where $idf(w, D)$ represents the ***idf*** for the ***term\word w*** *in* ***document D*** and ***N*** represents the count of the total number of documents in our corpus and $df(t)$ represents the frequency of the number of documents in which the term ***w*** is present.

Thus the term frequency-inverse document frequency can be computed by multiplying the above two measures together. The final tf-idf metric which we will be using is a normalized version of the tfidf matrix we get from the product of tf and idf. We will normalize the ***tfidf*** matrix by dividing it with the L2 norm of the matrix also known as the Euclidean norm which is the square root of the sum of the square of each term's tfidf weight. Mathematically we can represent the final ***tfidf*** feature vector as

$$tfidf = \frac{tfidf}{\| tfidf \|}$$

where $\| tfidf \|$ represents the Euclidean L2 norm for the ***tfidf*** matrix. There are multiple variants of this model but they all end up giving quite similar results. Let's apply this on our corpus now!

Using TfidfTransformer

The following code shows us an implementation of getting the *tfidf* based feature vectors considering we already have our bag of words feature vectors which we had obtained in a previous section.

```
from sklearn.feature_extraction.text import TfidfTransformer

tt = TfidfTransformer(norm='l2', use_idf=True)
tt_matrix = tt.fit_transform(cv_matrix)

tt_matrix = tt_matrix.toarray()
vocab = cv.get_feature_names()
pd.DataFrame(np.round(tt_matrix, 2), columns=vocab)
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.00	0.00	0.60	0.53	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.60	0.00	0.0
1	0.00	0.00	0.49	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.57	0.00	0.00	0.49	0.00	0.0
2	0.00	0.00	0.00	0.00	0.00	0.38	0.38	0.00	0.38	0.00	0.00	0.53	0.00	0.38	0.00	0.38	0.00	0.00	0.00	0.0
3	0.32	0.38	0.00	0.00	0.38	0.00	0.00	0.32	0.00	0.00	0.32	0.00	0.38	0.00	0.00	0.00	0.32	0.00	0.38	0.0
4	0.39	0.00	0.00	0.00	0.00	0.00	0.00	0.39	0.00	0.47	0.39	0.00	0.00	0.00	0.39	0.00	0.39	0.00	0.00	0.0
5	0.00	0.00	0.00	0.37	0.00	0.42	0.42	0.00	0.42	0.00	0.00	0.00	0.00	0.42	0.00	0.42	0.00	0.00	0.00	0.0
6	0.00	0.00	0.36	0.32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.72	0.00	0.5
7	0.00	0.00	0.00	0.00	0.00	0.45	0.45	0.00	0.45	0.00	0.00	0.00	0.00	0.45	0.00	0.45	0.00	0.00	0.00	0.0

Figure 4-4. *Our TF-IDF model based document feature vectors using TfidfTransformer*

You can see that we have used the L2 norm option in the parameters and also made sure we smoothen the idfs to give weightages also to terms which may have zero idf so that we do not ignore them.

Using TfidfVectorizer

You don't always need to generate features beforehand using a Bag of Words - count based model before engineering TF-IDF features. The TfidfVectorizer by scikit-learn enables us to directly compute the *tfidf* vectors by taking the raw documents themselves as input and internally computing the term frequencies as well as the inverse document frequencies eliminating the need to use the CountVectorizer for computing the term frequencies based on the bag of words model. Support is also present for adding n-grams to the feature vectors. We can see the function in action in the following snippet.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tv = TfidfVectorizer(min_df=0., max_df=1., norm='l2',
                     use_idf=True, smooth_idf=True)
tv_matrix = tv.fit_transform(norm_corpus)
tv_matrix = tv_matrix.toarray()
```

```
vocab = tv.get_feature_names()
pd.DataFrame(np.round(tv_matrix, 2), columns=vocab)
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.00	0.00	0.60	0.53	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.60	0.00	0.0
1	0.00	0.00	0.49	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.57	0.00	0.00	0.49	0.00	0.0
2	0.00	0.00	0.00	0.00	0.00	0.38	0.38	0.00	0.38	0.00	0.00	0.53	0.00	0.38	0.00	0.38	0.00	0.00	0.00	0.0
3	0.32	0.38	0.00	0.00	0.38	0.00	0.00	0.32	0.00	0.00	0.32	0.00	0.38	0.00	0.00	0.00	0.32	0.00	0.38	0.0
4	0.39	0.00	0.00	0.00	0.00	0.00	0.00	0.39	0.00	0.47	0.39	0.00	0.00	0.00	0.39	0.00	0.39	0.00	0.00	0.0
5	0.00	0.00	0.00	0.37	0.00	0.42	0.42	0.00	0.42	0.00	0.00	0.00	0.00	0.42	0.00	0.42	0.00	0.00	0.00	0.0
6	0.00	0.00	0.36	0.32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.72	0.00	0.5
7	0.00	0.00	0.00	0.00	0.00	0.45	0.45	0.00	0.45	0.00	0.00	0.00	0.00	0.45	0.00	0.45	0.00	0.00	0.00	0.0

Figure 4-5. *Our TF-IDF model based document feature vectors using TfidfVectorizer*

You can see that just like before, we have used the L2 norm option in the parameters and also made sure we smoothen the idfs. You can see from the above outputs that the *tfidf* feature vectors match to the ones we had obtained previously.

Understanding the TF-IDF Model

This section is dedicated to machine learning experts and our curious readers who are often interested in how things work behind the scenes! We will start with loading necessary dependencies and computing the term frequencies (*TF*) for our sample corpus.

```
# get unique words as feature names
unique_words = list(set([word for doc in [doc.split() for doc in norm_corpus]
                            for word in doc]))
def_feature_dict = {w: 0 for w in unique_words}
print('Feature Names:', unique_words)
print('Default Feature Dict:', def_feature_dict)
```

```
Feature Names: ['lazy', 'fox', 'love', 'jumps', 'sausages', 'blue', 'ham',
'beautiful', 'brown', 'kings', 'eggs', 'quick', 'bacon', 'breakfast', 'toast',
'beans', 'green', 'today', 'dog', 'sky']
Default Feature Dict: {'lazy': 0, 'fox': 0, 'kings': 0, 'love': 0, 'jumps': 0,
'sausages': 0, 'breakfast': 0, 'today': 0, 'brown': 0, 'ham': 0, 'beautiful':
0, 'green': 0, 'eggs': 0, 'blue': 0, 'bacon': 0, 'toast': 0, 'beans': 0, 'dog':
0, 'sky': 0, 'quick': 0}
```

```
from collections import Counter
# build bag of words features for each document - term frequencies
bow_features = []
```

```
for doc in norm_corpus:
    bow_feature_doc = Counter(doc.split())
    all_features = Counter(def_feature_dict)
    bow_feature_doc.update(all_features)
    bow_features.append(bow_feature_doc)

bow_features = pd.DataFrame(bow_features)
bow_features
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
2	0	0	0	0	0	1	1	0	1	0	0	1	0	1	0	1	0	0	0	0
3	1	1	0	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	1	0
4	1	0	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	0
5	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0
6	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	1
7	0	0	0	0	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0

Figure 4-6. *Constructing count based Bag of Words features from scratch for our corpus*

We will now compute our document frequencies (*DF*) for each term based on the number of documents in which it occurs. The following snippet shows us how to obtain it from our bag of words features.

```
import scipy.sparse as sp
feature_names = list(bow_features.columns)

# build the document frequency matrix
df = np.diff(sp.csc_matrix(bow_features, copy=True).indptr)
df = 1 + df # adding 1 to smoothen idf later

# show smoothened document frequencies
pd.DataFrame([df], columns=feature_names)
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	3	2	4	5	2	4	4	3	4	2	3	2	2	4	3	4	3	4	2	2

Figure 4-7. *Document frequencies for each feature in our corpus*

This tells us the document frequency (*DF*) for each term and you can verify it with the documents in our sample corpus. Remember that we have added 1 to each frequency value to smoothen the *IDF* values later and prevent division by zero errors by assuming we have a document (imaginary) which has all the terms once. Thus if you check in the corpus, you will see that bacon occurs 2(+1) times, sky occurs 3(+1) times and so on considering (+1) for our smoothening.

Now that we have the document frequencies, we will compute the inverse document frequency (*IDF*) by using our formula which we had defined earlier. Remember to add 1 to the total count of documents in the corpus to add the document which we had assumed earlier to contain all the terms at least once for smoothening the idfs.

```
# compute inverse document frequencies
total_docs = 1 + len(norm_corpus)
idf = 1.0 + np.log(float(total_docs) / df)

# show smoothened idfs
pd.DataFrame([np.round(idf, 2)], columns=feature_names)
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	2.1	2.5	1.81	1.59	2.5	1.81	1.81	2.1	1.81	2.5	2.1	2.5	2.5	1.81	2.1	1.81	2.1	1.81	2.5	2.5

Figure 4-8. *Inverse Document frequencies for each feature in our corpus*

Thus, we can see that Figure 4-8 depicts the inverse document frequencies (smoothed) for each feature in our corpus. We will now convert it into a matrix as follows for easier operations when we compute the overall TF-IDF score later.

```
# compute idf diagonal matrix
total_features = bow_features.shape[1]
idf_diag = sp.spdiags(idf, diags=0, m=total_features, n=total_features)
idf_dense = idf_diag.todense()

# print the idf diagonal matrix
pd.DataFrame(np.round(idf_dense, 2))
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	2.1	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
1	0.0	2.5	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
2	0.0	0.0	1.81	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
3	0.0	0.0	0.00	1.59	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
4	0.0	0.0	0.00	0.00	2.5	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
5	0.0	0.0	0.00	0.00	0.0	1.81	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
6	0.0	0.0	0.00	0.00	0.0	0.00	1.81	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
7	0.0	0.0	0.00	0.00	0.0	0.00	0.00	2.1	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
8	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	1.81	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
9	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	2.5	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
10	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	2.1	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
11	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	2.5	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
12	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	2.5	0.00	0.0	0.00	0.0	0.00	0.0	0.0
13	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	1.81	0.0	0.00	0.0	0.00	0.0	0.0
14	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	2.1	0.00	0.0	0.00	0.0	0.0
15	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	1.81	0.0	0.00	0.0	0.0
16	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	2.1	0.00	0.0	0.0
17	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	1.81	0.0	0.0
18	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	2.5	0.0
19	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	2.5

Figure 4-9. *Constructing an nverse Document frequency diagonal matrix for each feature in our corpus*

You can now see the *idf* matrix which we created based on our mathematical equation and we also convert it to a diagonal matrix which will be helpful later on when we want to compute the product with term frequency. Now that we have our *TFs* and *IDFs*, we can compute the raw *TF-IDF* feature matrix using matrix multiplication as depicted in the following snippet.

```
# compute tfidf feature matrix
tf = np.array(bow_features, dtype='float64')
tfidf = tf * idf
# view raw tfidf feature matrix
pd.DataFrame(np.round(tfidf, 2), columns=feature_names)
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.0	0.0	1.81	1.59	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	1.81	0.0	0.0
1	0.0	0.0	1.81	1.59	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	2.1	0.00	0.0	1.81	0.0	0.0
2	0.0	0.0	0.00	0.00	0.0	1.81	1.81	0.0	1.81	0.0	0.0	2.5	0.0	1.81	0.0	1.81	0.0	0.00	0.0	0.0
3	2.1	2.5	0.00	0.00	2.5	0.00	0.00	2.1	0.00	0.0	2.1	0.0	2.5	0.00	0.0	0.00	2.1	0.00	2.5	0.0
4	2.1	0.0	0.00	0.00	0.0	0.00	0.00	2.1	0.00	2.5	2.1	0.0	0.0	0.00	2.1	0.00	2.1	0.00	0.0	0.0
5	0.0	0.0	0.00	1.59	0.0	1.81	1.81	0.0	1.81	0.0	0.0	0.0	0.0	1.81	0.0	1.81	0.0	0.00	0.0	0.0
6	0.0	0.0	1.81	1.59	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	3.62	0.0	2.5
7	0.0	0.0	0.00	0.00	0.0	1.81	1.81	0.0	1.81	0.0	0.0	0.0	0.0	1.81	0.0	1.81	0.0	0.00	0.0	0.0

Figure 4-10. *Constructing the raw TF-IDF matrix from the TF and IDF components*

We now have our *tfidf* feature matrix, but wait! It is not yet over since we have to divide it with the L2 norm if you remember from our equations depicted earlier. The following snippet computes the *tfidf* norms for each document and then divides the *tfidf* weights with the norm to give us the final desired *tfidf* matrix.

```
from numpy.linalg import norm
# compute L2 norms
norms = norm(tfidf, axis=1)

# print norms for each document
print (np.round(norms, 3))
```

```
[ 3.013  3.672  4.761  6.534  5.319  4.35   5.019  4.049]
```

```
# compute normalized tfidf
norm_tfidf = tfidf / norms[:, None]

# show final tfidf feature matrix
pd.DataFrame(np.round(norm_tfidf, 2), columns=feature_names)
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.00	0.00	0.60	0.53	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.60	0.00	0.0
1	0.00	0.00	0.49	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.57	0.00	0.00	0.49	0.00	0.0
2	0.00	0.00	0.00	0.00	0.00	0.38	0.38	0.00	0.38	0.00	0.00	0.53	0.00	0.38	0.00	0.38	0.00	0.00	0.00	0.0
3	0.32	0.38	0.00	0.00	0.38	0.00	0.00	0.32	0.00	0.00	0.32	0.00	0.38	0.00	0.00	0.00	0.32	0.00	0.38	0.0
4	0.39	0.00	0.00	0.00	0.00	0.00	0.00	0.39	0.00	0.47	0.39	0.00	0.00	0.00	0.39	0.00	0.39	0.00	0.00	0.0
5	0.00	0.00	0.00	0.37	0.00	0.42	0.42	0.00	0.42	0.00	0.00	0.00	0.00	0.42	0.00	0.42	0.00	0.00	0.00	0.0
6	0.00	0.00	0.36	0.32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.72	0.00	0.5
7	0.00	0.00	0.00	0.00	0.00	0.45	0.45	0.00	0.45	0.00	0.00	0.00	0.00	0.45	0.00	0.45	0.00	0.00	0.00	0.0

Figure 4-11. *Constructing the final normalized TF-IDF matrix*

Compare the above obtained tfidf feature matrix for the documents in our corpus to the feature matrix obtained using TfidfTransformer or TfidfVectorizer earlier. You will notice they are exactly the same thus verifying that our mathematical implementation was correct and in fact this very same implementation is adopted by scikit-learn behind the scenes using some more optimizations.

Extracting Features for New Documents

Suppose you have built a machine learning model to classify and categorize news articles and it in currently in production. How will you generate features for completely new documents so that we can feed it into our machine learning models for predictions? The scikit-learn API provides the transform(…) function for the vectorizers we discussed previously and we can leverage the same to get features for a completely new document which was not present in our corpus previously (when we trained our model in the past).

```
new_doc = 'the sky is green today'
pd.DataFrame(np.round(tv.transform([new_doc]).toarray(), 2),
             columns=tv.get_feature_names())
```

	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.63	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.46	0.0	0.63

Figure 4-12. *Generating the TF-IDF feature vector for a completely new document*

Thus always leverage the fit_transform(…) function to build a feature matrix on all documents in your corpus which typically becomes the training featureset on which you build and train your predictive or other machine learning models. Once ready, leverage the transform(…) function to generate feature vectors of new documents which can then be fed into your trained models to generate insights as needed.

Document Similarity

Document similarity is the process of using a distance or similarity based metric that can be used to identify how similar a text document is with any other document(s) based on features extracted from the documents like bag of words or tf-idf. Thus you can see that we can build on top of the tf-idf based features we engineered in the previous section and use them to generate new features which can be useful in domains like search engines, document clustering and information retrieval by leveraging these similarity based features.

Pairwise document similarity in a corpus involves computing document similarity for each pair of documents in a corpus. Thus if you have *C* documents in a corpus, you would end up with a *C* **x** *C* matrix such that each row and column represents the similarity score for a pair of documents, which represent the

indices at the row and column, respectively. There are several similarity and distance metrics that are used to compute document similarity. These include cosine distance/similarity, euclidean distance, manhattan distance, BM25 similarity, jaccard distance and so on. In our analysis, we will be using perhaps the most popular and widely used similarity metric, cosine similarity and compare pairwise document similarity based on their TF-IDF feature vectors.

```
from sklearn.metrics.pairwise import cosine_similarity

similarity_matrix = cosine_similarity(tv_matrix)
similarity_df = pd.DataFrame(similarity_matrix)
similarity_df
```

	0	1	2	3	4	5	6	7
0	1.000000	0.820599	0.000000	0.000000	0.000000	0.192353	0.817246	0.000000
1	0.820599	1.000000	0.000000	0.000000	0.225489	0.157845	0.670631	0.000000
2	0.000000	0.000000	1.000000	0.000000	0.000000	0.791821	0.000000	0.850516
3	0.000000	0.000000	0.000000	1.000000	0.506866	0.000000	0.000000	0.000000
4	0.000000	0.225489	0.000000	0.506866	1.000000	0.000000	0.000000	0.000000
5	0.192353	0.157845	0.791821	0.000000	0.000000	1.000000	0.115488	0.930989
6	0.817246	0.670631	0.000000	0.000000	0.000000	0.115488	1.000000	0.000000
7	0.000000	0.000000	0.850516	0.000000	0.000000	0.930989	0.000000	1.000000

Figure 4-13. *Pairwise document similarity matrix (cosine similarity)*

Cosine similarity basically gives us a metric representing the cosine of the angle between the feature vector representations of two text documents. Lower the angle between the documents, the closer and more similar they are as depicted with the scores in Figure 4-13 and with some sample document vectors depicted in Figure 4-14.

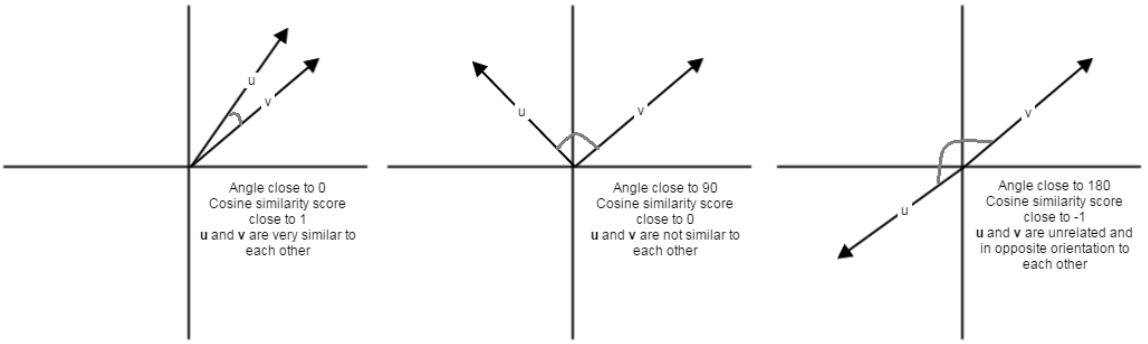

Figure 4-14. *Cosine similarity depictions for text document feature vectors*

Looking closely at the similarity matrix in Figure 4-13, you can clearly see that documents (0, 1 and 6), (2, 5 and 7) are very similar to one another and documents 3 and 4 are slightly similar to each other but the magnitude is not very strong, however still stronger than the other documents. This must indicate these similar documents have some similar features. This is a perfect example of grouping or clustering that can be solved by unsupervised learning especially when you are dealing with huge corpora of millions of text documents.

Topic Models

While we will be covering topic modeling in detail in a separate chapter in this book, a discussion about feature engineering is not complete without talking about topic models. We can use some summarization techniques to extract topic or concept based features from text documents. The idea of topic models revolves around the process of extracting key themes or concepts from a corpus of documents which are represented as topics. Each topic can be represented as a bag or collection of words/terms from the document corpus. Together, these terms signify a specific topic, theme or a concept and each topic can be easily distinguished from other topics by virtue of the semantic meaning conveyed by these terms. However often you do end up with overlapping topics based on the data. These concepts can range from simple facts and statements to opinions and outlook. Topic models are extremely useful in summarizing large corpus of text documents to extract and depict key concepts. They are also useful in extracting features from text data that capture latent patterns in the data.

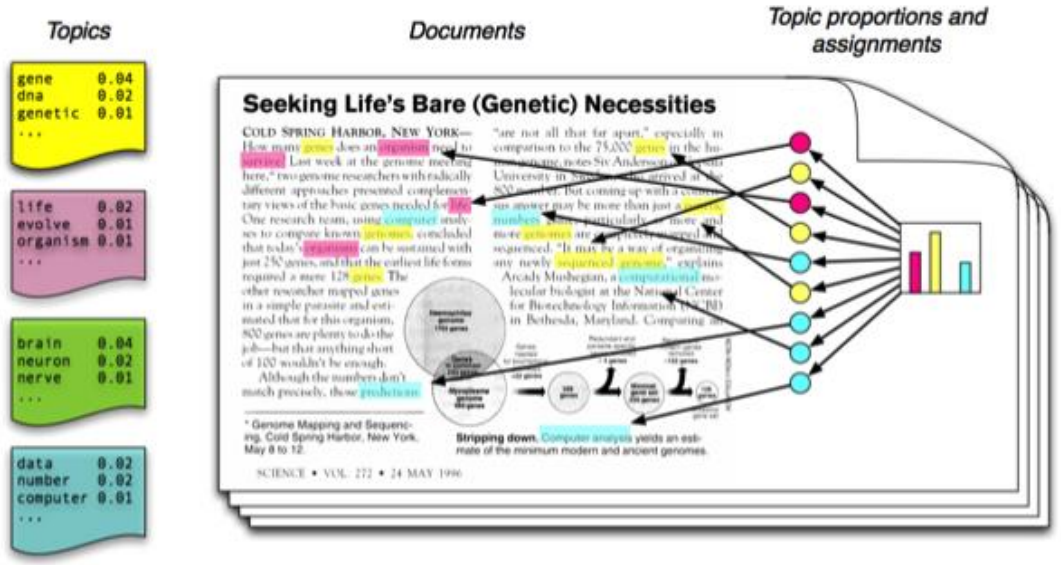

Figure source: Blei, D. M. (2012). Probabilistic topic models. *Communications of the ACM*, *55*(4), 77-84.

Frameworks like gensim or scikit-learn enable us to leverage the LDA (Latent Dirichlet Allocation) model for generating topics. For the purpose of feature engineering which is the intent of this article, you need to remember that when LDA is applied on a document-term matrix (TF-IDF or Bag of Words feature matrix), it gets decomposed into two main components.

- A document-topic matrix, which would be the feature matrix we are looking for.
- A topic-term matrix, which helps us in looking at potential topics in the corpus.

Let's now leverage scikit-learn to get the document-topic matrix as follows, which can be used as features for any subsequent modeling requirements.

```
from sklearn.decomposition import LatentDirichletAllocation

lda = LatentDirichletAllocation(n_topics=3, max_iter=10000, random_state=0)
dt_matrix = lda.fit_transform(cv_matrix)
features = pd.DataFrame(dt_matrix, columns=['T1', 'T2', 'T3'])
features
```

	T1	T2	T3
0	0.832191	0.083480	0.084329
1	0.863554	0.069100	0.067346
2	0.047794	0.047776	0.904430
3	0.037243	0.925559	0.037198
4	0.049121	0.903076	0.047802
5	0.054901	0.047778	0.897321
6	0.888287	0.055697	0.056016
7	0.055704	0.055689	0.888607

Figure 4-22. *Document-Topic Feature Matrix from our LDA Model*

You can clearly see which documents contribute the most to which of the three topics in the above output. You can view the topics and their main constituents as follows.

```
tt_matrix = lda.components_
for topic_weights in tt_matrix:
    topic = [(token, weight) for token, weight in zip(vocab, topic_weights)]
    topic = sorted(topic, key=lambda x: -x[1])
    topic = [item for item in topic if item[1] > 0.6]
    print(topic)
    print()
```

```
[('sky', 4.3324395825632624), ('blue', 3.373753174831771), ('beautiful',
3.3323652405224857), ('today', 1.3325579841038182), ('love',
1.3304224288080069)]

[('bacon', 2.332695948479998), ('eggs', 2.332695948479998), ('ham',
2.332695948479998), ('sausages', 2.332695948479998), ('love',
1.335454457601996), ('beans', 1.332773525378464), ('breakfast',
1.332773525378464), ('kings', 1.332773525378464), ('toast', 1.332773525378464),
('green', 1.3325433207547732)]

[('brown', 3.3323474595768783), ('dog', 3.3323474595768783), ('fox',
3.3323474595768783), ('lazy', 3.3323474595768783), ('quick',
3.3323474595768783), ('jumps', 1.3324193736202712), ('blue',
1.2919635624485213)]
```

Thus you can clearly see the three topics are quite distinguishable from each other based on their constituent terms, first one talking about weather, second one about food and the last one about animals. Choosing the number of topics for topic modeling is an entire topic on its own (pun not intended!) and

is an art as well as a science. There are various methods and heuristics to get the optimal number of topics but due to the detailed nature of these techniques, we don’t discuss them here.