CHAPTER 4

Text Vectorization and Transformation Pipelines

Machine learning algorithms operate on a numeric feature space, expecting input as a two-dimensional array where rows are instances and columns are features. In order to perform machine learning on text, we need to transform our documents into vector representations such that we can apply numeric machine learning. This process is called *feature extraction* or more simply, *vectorization*, and is an essential first step toward language-aware analysis.

Representing documents numerically gives us the ability to perform meaningful analytics and also creates the *instances* on which machine learning algorithms operate. In text analysis, instances are entire documents or utterances, which can vary in length from quotes or tweets to entire books, but whose vectors are always of a uniform length. Each property of the vector representation is a *feature*. For text, features represent attributes and properties of documents—including its content as well as meta attributes, such as document length, author, source, and publication date. When considered together, the features of a document describe a multidimensional feature space on which machine learning methods can be applied.

For this reason, we must now make a critical shift in how we think about language—from a sequence of words to points that occupy a high-dimensional semantic *space*. Points in space can be close together or far apart, tightly clustered or evenly distributed. Semantic space is therefore mapped in such a way where documents with similar meanings are closer together and those that are different are farther apart. By encoding similarity as distance, we can begin to derive the primary components of documents and draw decision boundaries in our semantic space.

The simplest encoding of semantic space is the *bag-of-words* model, whose primary insight is that meaning and similarity are encoded in vocabulary. For example, the

Wikipedia articles about baseball and Babe Ruth are probably very similar. Not only will many of the same words appear in both, they will not share many words in common with articles about casseroles or quantitative easing. This model, while simple, is extremely effective and forms the starting point for the more complex models we will explore.

In this chapter, we will demonstrate how to use the vectorization process to combine linguistic techniques from NLTK with machine learning techniques in Scikit-Learn and Gensim, creating custom *transformers* that can be used inside repeatable and reusable *pipelines*. By the end of this chapter, we will be ready to engage our preprocessed corpus, transforming documents to model space so that we can begin making predictions.

Words in Space

To vectorize a corpus with a bag-of-words (BOW) approach, we represent every document from the corpus as a vector whose length is equal to the vocabulary of the corpus. We can simplify the computation by sorting token positions of the vector into alphabetical order, as shown in Figure 4-1. Alternatively, we can keep a dictionary that maps tokens to vector positions. Either way, we arrive at a vector mapping of the corpus that enables us to uniquely represent every document.

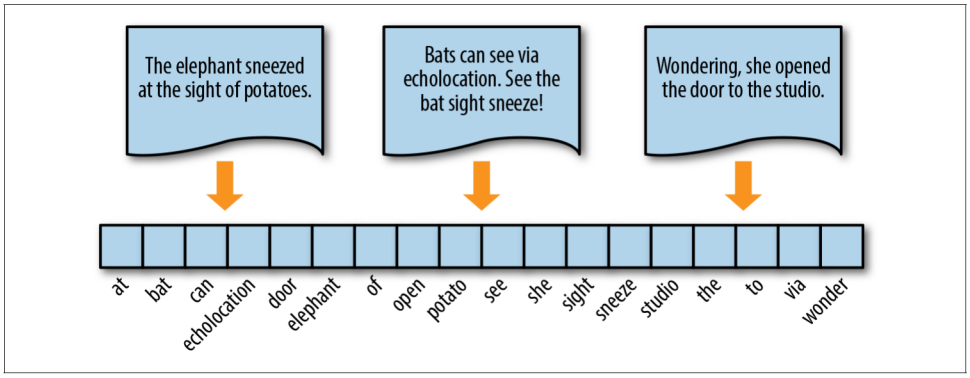

Figure 4-1. Encoding documents as vectors

What should each element in the document vector be? In the next few sections, we will explore several choices, each of which extends or modifies the base bag-of-words model to describe semantic space. We will look at four types of vector encoding—frequency, one-hot, TF–IDF, and distributed representations—and discuss their implementations in Scikit-Learn, Gensim, and NLTK. We'll operate on a small corpus of the three sentences in the example figures.

To set this up, let's create a list of our documents and tokenize them for the proceeding vectorization examples. The `tokenize` method performs some lightweight nor-

malization, stripping punctuation using the `string.punctuation` character set and setting the text to lowercase. This function also performs some feature reduction using the `SnowballStemmer` to remove affixes such as plurality ("bats" and "bat" are the same token). The examples in the next section will utilize this example corpus and some will use the tokenization method.

```
import nltk
import string

def tokenize(text):
    stem = nltk.stem.SnowballStemmer('english')
    text = text.lower()

    for token in nltk.word_tokenize(text):
        if token in string.punctuation: continue
        yield stem.stem(token)

corpus = [
    "The elephant sneezed at the sight of potatoes.",
    "Bats can see via echolocation. See the bat sight sneeze!",
    "Wondering, she opened the door to the studio.",
]
```

The choice of a specific vectorization technique will be largely driven by the problem space. Similarly, our choice of implementation—whether NLTK, Scikit-Learn, or Gensim—should be dictated by the requirements of the application. For instance, NLTK offers many methods that are especially well-suited to text data, but is a big dependency. Scikit-Learn was not designed with text in mind, but does offer a robust API and many other conveniences (which we'll explore later in this chapter) particularly useful in an applied context. Gensim can serialize dictionaries and references in matrix market format, making it more flexible for multiple platforms. However, unlike Scikit-Learn, Gensim doesn't do any work on behalf of your documents for tokenization or stemming.

For this reason, as we walk through each of the four approaches to encoding, we'll show a few options for implementation—"With NLTK," "In Scikit-Learn," and "The Gensim Way."

Frequency Vectors

The simplest vector encoding model is to simply fill in the vector with the frequency of each word as it appears in the document. In this encoding scheme, each document is represented as the multiset of the tokens that compose it and the value for each word position in the vector is its count. This representation can either be a straight count (integer) encoding as shown in Figure 4-2 or a normalized encoding where each word is weighted by the total number of words in the document.

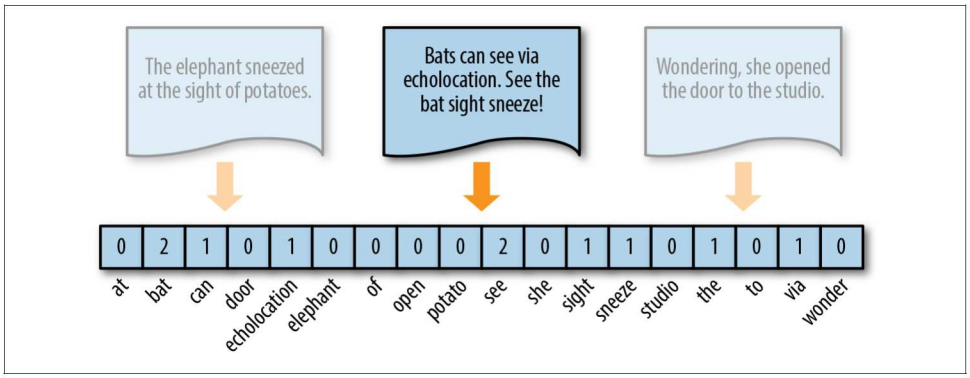

Figure 4-2. Token frequency as vector encoding

With NLTK

NLTK expects features as a `dict` object whose keys are the names of the features and whose values are boolean or numeric. To encode our documents in this way, we'll create a `vectorize` function that creates a dictionary whose keys are the tokens in the document and whose values are the number of times that token appears in the document.

The `defaultdict` object allows us to specify what the dictionary will return for a key that hasn't been assigned to it yet. By setting `defaultdict(int)` we are specifying that a `0` should be returned, thus creating a simple counting dictionary. We can `map` this function to every item in the corpus using the last line of code, creating an iterable of vectorized documents.

```
from collections import defaultdict

def vectorize(doc):
    features = defaultdict(int)
    for token in tokenize(doc):
        features[token] += 1
    return features

vectors = map(vectorize, corpus)
```

In Scikit-Learn

The `CountVectorizer` transformer from the `sklearn.feature_extraction` model has its own internal tokenization and normalization methods. The `fit` method of the vectorizer expects an iterable or list of strings or file objects, and creates a dictionary of the vocabulary on the corpus. When `transform` is called, each individual document is transformed into a sparse array whose index tuple is the row (the document ID) and the token ID from the dictionary, and whose value is the count:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
vectors = vectorizer.fit_transform(corpus)
```

Vectors can become extremely sparse, particularly as vocabularies get larger, which can have a significant impact on the speed and performance of machine learning models. For very large corpora, it is recommended to use the Scikit-Learn `HashingVectorizer`, which uses a hashing trick to find the token string name to feature index mapping. This means it uses very low memory and scales to large datasets as it does not need to store the entire vocabulary and it is faster to pickle and fit since there is no state. However, there is no inverse transform (from vector to text), there can be collisions, and there is no inverse document frequency weighting.

The Gensim way

Gensim's frequency encoder is called `doc2bow`. To use `doc2bow`, we first create a Gensim `Dictionary` that maps tokens to indices based on observed order (eliminating the overhead of lexicographic sorting). The dictionary object can be loaded or saved to disk, and implements a `doc2bow` library that accepts a *pretokenized document* and returns a sparse matrix of `(id, count)` tuples where the `id` is the token's id in the dictionary. Because the `doc2bow` method only takes a single document instance, we use the list comprehension to restore the entire corpus, loading the tokenized documents into memory so we don't exhaust our generator:

```
import gensim

corpus  = [tokenize(doc) for doc in corpus]
id2word = gensim.corpora.Dictionary(corpus)
vectors = [
    id2word.doc2bow(doc) for doc in corpus
]
```

One-Hot Encoding

Because they disregard grammar and the relative position of words in documents, frequency-based encoding methods suffer from the *long tail*, or Zipfian distribution, that characterizes natural language. As a result, tokens that occur very frequently are orders of magnitude more "significant" than other, less frequent ones. This can have a significant impact on some models (e.g., generalized linear models) that expect normally distributed features.

A solution to this problem is *one-hot encoding*, a boolean vector encoding method that marks a particular vector index with a value of true (1) if the token exists in the document and false (0) if it does not. In other words, each element of a one-hot enco-

ded vector reflects either the presence or absence of the token in the described text as shown in Figure 4-3.

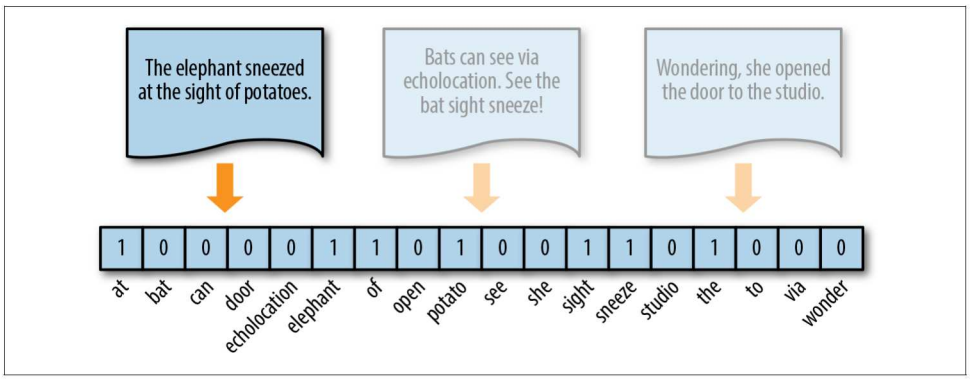

Figure 4-3. One-hot encoding

One-hot encoding reduces the imbalance issue of the distribution of tokens, simplifying a document to its constituent components. This reduction is most effective for very small documents (sentences, tweets) that don't contain very many repeated elements, and is usually applied to models that have very good smoothing properties. One-hot encoding is also commonly used in artificial neural networks, whose activation functions require input to be in the discrete range of `[0,1]` or `[-1,1]`.

With NLTK

The NLTK implementation of one-hot encoding is a dictionary whose keys are tokens and whose value is `True`:

```
def vectorize(doc):
    return {
        token: True
        for token in doc
    }

vectors = map(vectorize, corpus)
```

Dictionaries act as simple sparse matrices in the NLTK case because it is not necessary to mark every absent word `False`. In addition to the boolean dictionary values, it is also acceptable to use an integer value; 1 for present and 0 for absent.

In Scikit-Learn

In Scikit-Learn, one-hot encoding is implemented with the `Binarizer` transformer in the `preprocessing` module. The `Binarizer` takes only numeric data, so the text data must be transformed into a numeric space using the `CountVectorizer` ahead of one-hot encoding. The `Binarizer` class uses a threshold value (0 by default) such that all

values of the vector that are less than or equal to the threshold are set to zero, while those that are greater than the threshold are set to 1. Therefore, by default, the `Binarizer` converts all frequency values to 1 while maintaining the zero-valued frequencies.

```
from sklearn.preprocessing import Binarizer

freq   = CountVectorizer()
corpus = freq.fit_transform(corpus)

onehot = Binarizer()
corpus = onehot.fit_transform(corpus.toarray())
```

The `corpus.toarray()` method is optional; it converts the sparse matrix representation to a dense one. In corpora with large vocabularies, the sparse matrix representation is much better. Note that we could also use `CountVectorizer(binary=True)` to achieve one-hot encoding in the above, obviating the `Binarizer`.

In spite of its name, the `OneHotEncoder` transformer in the `sklearn.preprocessing` module is not exactly the right fit for this task. The `OneHotEncoder` treats each vector component (column) as an independent categorical variable, expanding the dimensionality of the vector for each observed value in each column. In this case, the component `(sight, 0)` and `(sight, 1)` would be treated as two categorical dimensions rather than as a single binary encoded vector component.

The Gensim way

While Gensim does not have a specific one-hot encoder, its `doc2bow` method returns a list of tuples that we can manage on the fly. Extending the code from the Gensim frequency vectorization example in the previous section, we can one-hot encode our vectors with our `id2word` dictionary. To get our `vectors`, an inner list comprehension converts the list of tuples returned from the `doc2bow` method into a list of `(token_id, 1)` tuples and the outer comprehension applies that converter to all documents in the corpus:

```
corpus  = [tokenize(doc) for doc in corpus]
id2word = gensim.corpora.Dictionary(corpus)
vectors = [
    [(token[0], 1) for token in id2word.doc2bow(doc)]
    for doc in corpus
]
```

One-hot encoding represents similarity and difference at the *document* level, but because all words are rendered equidistant, it is not able to encode per-word similarity. Moreover, because all words are equally distant, *word form* becomes incredibly

important; the tokens "trying" and "try" will be equally distant from unrelated tokens like "red" or "bicycle"! Normalizing tokens to a single word class, either through *stemming* or *lemmatization*, which we'll explore later in this chapter, ensures that different forms of tokens that embed plurality, case, gender, cardinality, tense, etc., are treated as single vector components, reducing the feature space and making models more performant.

Term Frequency–Inverse Document Frequency

The bag-of-words representations that we have explored so far only describe a document in a standalone fashion, not taking into account the context of the corpus. A better approach would be to consider the relative frequency or rareness of tokens in the document against their frequency in other documents. The central insight is that meaning is most likely encoded in the more rare terms from a document. For example, in a corpus of sports text, tokens such as "umpire," "base," and "dugout" appear more frequently in documents that discuss baseball, while other tokens that appear frequently throughout the corpus, like "run," "score," and "play," are less important.

TF–IDF, *term frequency–inverse document frequency*, encoding normalizes the frequency of tokens in a document with respect to the rest of the corpus. This encoding approach accentuates terms that are very relevant to a specific instance, as shown in Figure 4-4, where the token `studio` has a higher relevance to this document since it only appears there.

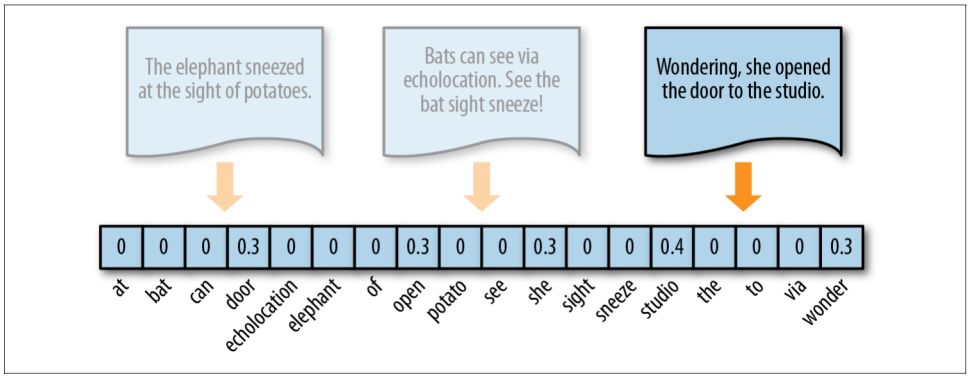

Figure 4-4. TF–IDF encoding

TF–IDF is computed on a per-term basis, such that the relevance of a token to a document is measured by the scaled frequency of the appearance of the term in the document, normalized by the inverse of the scaled frequency of the term in the entire corpus.

Computing TF–IDF

The term frequency of a term given a document, $tf(t, d)$, can be the boolean frequency (as in one-hot encoding, 1 if t occurs in d 0 otherwise), or the count. However, generally both the term frequency and inverse document frequency are scaled logarithmically to prevent bias of longer documents or terms that appear much more frequently relative to other terms: $tf(t, d) = 1 + \log f_{t,d}$.

Similarly, the inverse document frequency of a term given the set of documents can be logarithmically scaled as follows: $idf(t, D) = \log 1 + \frac{N}{n_t}$, where N is the number of documents and n_t is the number of occurrences of the term t in all documents. TF–IDF is then computed completely as $tfidf(t, d, D) = tf(t, d) \cdot idf(t, D)$.

Because the ratio of the id f log function is greater or equal to 1, the TF–IDF score is always greater than or equal to zero. We interpret the score to mean that the closer the TF–IDF score of a term is to 1, the more informative that term is to that document. The closer the score is to zero, the less informative that term is.

With NLTK

To vectorize text in this way with NLTK, we use the `TextCollection` class, a wrapper for a list of texts or a corpus consisting of one or more texts. This class provides support for counting, concordancing, collocation discovery, and more importantly, computing `tf_idf`.

Because TF–IDF requires the entire corpus, our new version of `vectorize` does not accept a single document, but rather all documents. After applying our tokenization function and creating the text collection, the function goes through each document in the corpus and yields a dictionary whose keys are the terms and whose values are the TF–IDF score for the term in that particular document.

```
from nltk.text import TextCollection

def vectorize(corpus):
    corpus = [tokenize(doc) for doc in corpus]
    texts  = TextCollection(corpus)

    for doc in corpus:
        yield {
            term: texts.tf_idf(term, doc)
            for term in doc
        }
```

In Scikit-Learn

Scikit-Learn provides a transformer called the `TfidfVectorizer` in the module called `feature_extraction.text` for vectorizing documents with TF–IDF scores. Under the hood, the `TfidfVectorizer` uses the `CountVectorizer` estimator we used to produce the bag-of-words encoding to count occurrences of tokens, followed by a `TfidfTransformer`, which normalizes these occurrence counts by the inverse document frequency.

The input for a `TfidfVectorizer` is expected to be a sequence of filenames, file-like objects, or strings that contain a collection of raw documents, similar to that of the `CountVectorizer`. As a result, a default tokenization and preprocessing method is applied unless other functions are specified. The vectorizer returns a sparse matrix representation in the form of `((doc, term), tfidf)` where each key is a document and term pair and the value is the TF–IDF score.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf  = TfidfVectorizer()
corpus = tfidf.fit_transform(corpus)
```

The Gensim way

In Gensim, the `TfidfModel` data structure is similar to the `Dictionary` object in that it stores a mapping of terms and their vector positions in the order they are observed, but additionally stores the corpus frequency of those terms so it can vectorize documents on demand. As before, Gensim allows us to apply our own tokenization method, expecting a corpus that is a list of lists of tokens. We first construct the lexicon and use it to instantiate the `TfidfModel`, which computes the normalized inverse document frequency. We can then fetch the TF–IDF representation for each vector using a `getitem`, dictionary-like syntax, after applying the `doc2bow` method to each document using the lexicon.

```
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]
```

Gensim provides helper functionality to write dictionaries and models to disk in a compact format, meaning you can conveniently save both the TF–IDF model and the lexicon to disk in order to load them later to vectorize new documents. It is possible (though slightly more work) to achieve the same result by using the `pickle` module in combination with Scikit-Learn. To save a Gensim model to disk:

```
lexicon.save_as_text('lexicon.txt', sort_by_word=True)
tfidf.save('tfidf.pkl')
```

This will save the lexicon as a text-delimited text file, sorted lexicographically, and the TF–IDF model as a pickled sparse matrix. Note that the `Dictionary` object can also be saved more compactly in a binary format using its `save` method, but `save_as_text` allows easy inspection of the dictionary for later work. To load the models from disk:

```
lexicon = gensim.corpora.Dictionary.load_from_text('lexicon.txt')
tfidf = gensim.models.TfidfModel.load('tfidf.pkl')
```

One benefit of TF–IDF is that it naturally addresses the problem of *stopwords*, those words most likely to appear in all documents in the corpus (e.g., "a," "the," "of", etc.), and thus will accrue very small weights under this encoding scheme. This biases the TF–IDF model toward moderately rare words. As a result TF–IDF is widely used for bag-of-words models, and is an excellent starting point for most text analytics.

Distributed Representation

While frequency, one-hot, and TF–IDF encoding enable us to put documents into vector space, it is often useful to also encode the similarities between documents in the context of that same vector space. Unfortunately, these vectorization methods produce document vectors with non-negative elements, which means we won't be able to compare documents that don't share terms (because two vectors with a cosine distance of 1 will be considered far apart, even if they are semantically similar).

When document similarity is important in the context of an application, we instead encode text along a continuous scale with a distributed representation, as shown in Figure 4-5. This means that the resulting document vector is not a simple mapping from token position to token score. Instead, the document is represented in a feature space that has been embedded to represent word similarity. The complexity of this space (and the resulting vector length) is the product of how the mapping to that representation is learned. The complexity of this space (and the resulting vector length) is the product of how that representation is trained and not directly tied to the document itself.

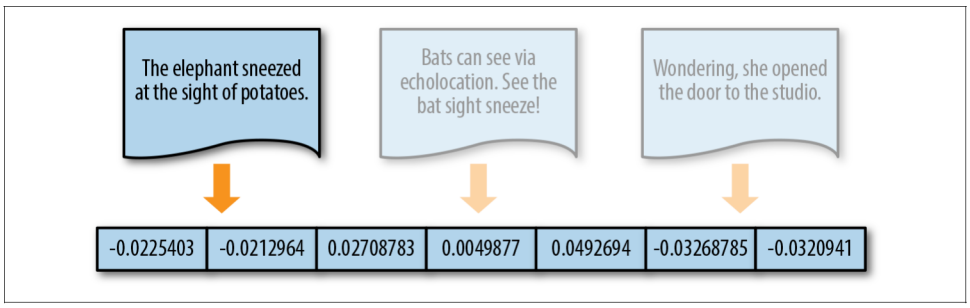

Figure 4-5. Distributed representation

Word2vec, created by a team of researchers at Google led by Tomáš Mikolov, implements a word embedding model that enables us to create these kinds of distributed representations. The word2vec algorithm trains word representations based on either a continuous bag-of-words (CBOW) or skip-gram model, such that words are embedded in space along with similar words based on their context. For example, Gensim's implementation uses a feedforward network.

The *doc2vec*[1] algorithm is an extension of *word2vec*. It proposes a *paragraph vector*—an unsupervised algorithm that learns fixed-length feature representations from variable length documents. This representation attempts to inherit the semantic properties of words such that "red" and "colorful" are more similar to each other than they are to "river" or "governance." Moreover, the paragraph vector takes into consideration the ordering of words within a narrow context, similar to an *n*-gram model. The combined result is much more effective than a bag-of-words or bag-of-*n*-grams model because it generalizes better and has a lower dimensionality but still is of a fixed length so it can be used in common machine learning algorithms.

The Gensim way

Neither NLTK nor Scikit-Learn provide implementations of these kinds of word embeddings. Gensim's implementation allows users to train both word2vec and doc2vec models on custom corpora and also conveniently comes with a model that is pretrained on the Google news corpus.

To use Gensim's pretrained models, you'll need to download the model bin file, which clocks in at 1.5 GB. For applications that require extremely lightweight dependencies (e.g., if they have to run on an AWS lambda instance), this may not be practicable.

We can train our own model as follows. First, we use a list comprehension to load our corpus into memory. (Gensim supports streaming, but this will enable us to avoid exhausting the generator.) Next, we create a list of `TaggedDocument` objects, which extend the `LabeledSentence`, and in turn the distributed representation of word2vec. `TaggedDocument` objects consist of *words* and *tags*. We can instantiate the tagged document with the list of tokens along with a single tag, one that uniquely identifies the instance. In this example, we've labeled each document as `"d{}".format(idx)`, e.g. `d0`, `d1`, `d2` and so forth.

Once we have a list of tagged documents, we instantiate the `Doc2Vec` model and specify the size of the vector as well as the minimum count, which ignores all tokens that

1 Quoc V. Le and Tomas Mikolov, *Distributed Representations of Sentences and Documents*, (2014) *http://bit.ly/2GJBHjZ*

have a frequency less than that number. The `size` parameter is usually not as low a dimensionality as 5; we selected such a small number for demonstration purposes only. We also set the `min_count` parameter to zero to ensure we consider all tokens, but generally this is set between 3 and 5, depending on how much information the model needs to capture. Once instantiated, an unsupervised neural network is trained to learn the vector representations, which can then be accessed via the `docvecs` property.

```
from gensim.models.doc2vec import TaggedDocument, Doc2Vec

corpus = [list(tokenize(doc)) for doc in corpus]
corpus = [
    TaggedDocument(words, ['d{}'.format(idx)])
    for idx, words in enumerate(corpus)
]

model = Doc2Vec(corpus, size=5, min_count=0)
print(model.docvecs[0])
# [ 0.01797447 -0.01509272  0.0731937   0.06814702 -0.0846546 ]
```

Distributed representations will dramatically improve results over TF–IDF models when used correctly. The model itself can be saved to disk and retrained in an active fashion, making it extremely flexible for a variety of use cases. However, on larger corpora, training can be slow and memory intensive, and it might not be as good as a TF–IDF model with Principal Component Analysis (PCA) or Singular Value Decomposition (SVD) applied to reduce the feature space. In the end, however, this representation is breakthrough work that has led to a dramatic improvement in text processing capabilities of data products in recent years.

Again, the choice of vectorization technique (as well as the library implementation) tend to be use case- and application-specific, as summarized in Table 4-1.

Table 4-1. Overview of text vectorization methods

Vectorization Method	Function	Good For	Considerations
Frequency	Counts term frequencies	Bayesian models	Most frequent words not always most informative
One-Hot Encoding	Binarizes term occurrence (0, 1)	Neural networks	All words equidistant, so normalization extra important
TF–IDF	Normalizes term frequencies across documents	General purpose	Moderately frequent terms may not be representative of document topics
Distributed Representations	Context-based, continuous term similarity encoding	Modeling more complex relationships	Performance intensive; difficult to scale without additional tools (e.g., Tensorflow)