3
Clustering – Finding Related Posts

In the previous chapter, you learned how to find the classes or categories of individual datapoints. With a handful of training data items that were paired with their respective classes, you learned a model, which we can now use to classify future data items. We called this supervised learning because the learning was guided by a teacher; in our case, the teacher had the form of correct classifications.

Let's now imagine that we do not possess those labels by which we can learn the classification model. This could be, for example, because they were too expensive to collect. Just imagine the cost if the only way to obtain millions of labels will be to ask humans to classify those manually. What could we have done in that case?

Well, of course, we will not be able to learn a classification model. Still, we could find some pattern within the data itself. That is, let the data describe itself. This is what we will do in this chapter, where we consider the challenge of a question and answer website. When a user is browsing our site, perhaps because they were searching for particular information, the search engine will most likely point them to a specific answer. If the presented answers are not what they were looking for, the website should present (at least) the related answers so that they can quickly see what other answers are available and hopefully stay on our site.

The naïve approach will be to simply take the post, calculate its similarity to all other posts and display the top *n* most similar posts as links on the page. Quickly, this will become very costly. Instead, we need a method that quickly finds all the related posts.

We will achieve this goal in this chapter using clustering. This is a method of arranging items so that similar items are in one cluster and dissimilar items are in distinct ones. The tricky thing that we have to tackle first is how to turn text into something on which we can calculate similarity. With such a similarity measurement, we will then proceed to investigate how we can leverage that to quickly arrive at a cluster that contains similar posts. Once there, we will only have to check out those documents that also belong to that cluster. To achieve this, we will introduce you to the marvelous SciKit library, which comes with diverse machine learning methods that we will also use in the following chapters.

Measuring the relatedness of posts

From the machine learning point of view, raw text is useless. Only if we manage to transform it into meaningful numbers, can we then feed it into our machine learning algorithms, such as clustering. This is true for more mundane operations on text such as similarity measurement.

How not to do it

One text similarity measure is the Levenshtein distance, which also goes by the name Edit Distance. Let's say we have two words, "machine" and "mchiene". The similarity between them can be expressed as the minimum set of edits that are necessary to turn one word into the other. In this case, the edit distance will be 2, as we have to add an "a" after the "m" and delete the first "e". This algorithm is, however, quite costly as it is bound by the length of the first word times the length of the second word.

Looking at our posts, we could cheat by treating whole words as characters and performing the edit distance calculation on the word level. Let's say we have two posts (let's concentrate on the following title, for simplicity's sake) called "How to format my hard disk" and "Hard disk format problems", we will need an edit distance of 5 because of removing "how", "to", "format", "my" and then adding "format" and "problems" in the end. Thus, one could express the difference between two posts as the number of words that have to be added or deleted so that one text morphs into the other. Although we could speed up the overall approach quite a bit, the time complexity remains the same.

But even if it would have been fast enough, there is another problem. In the earlier post, the word "format" accounts for an edit distance of 2, due to deleting it first, then adding it. So, our distance seems to be not robust enough to take word reordering into account.

How to do it

More robust than edit distance is the so-called **bag of word** approach. It totally ignores the order of words and simply uses word counts as their basis. For each word in the post, its occurrence is counted and noted in a vector. Not surprisingly, this step is also called vectorization. The vector is typically huge as it contains as many elements as words occur in the whole dataset. Take, for instance, two example posts with the following word counts:

Word	Occurrences in post 1	Occurrences in post 2
disk	1	1
format	1	1
how	1	0
hard	1	1
my	1	0
problems	0	1
to	1	0

The columns Occurrences in post 1 and Occurrences in post 2 can now be treated as simple vectors. We can simply calculate the Euclidean distance between the vectors of all posts and take the nearest one (too slow, as we have found out earlier). And as such, we can use them later as our feature vectors in the clustering steps according to the following procedure:

1. Extract salient features from each post and store it as a vector per post.
2. Then compute clustering on the vectors.
3. Determine the cluster for the post in question.
4. From this cluster, fetch a handful of posts having a different similarity to the post in question. This will increase diversity.

But there is some more work to be done before we get there. Before we can do that work, we need some data to work on.

Preprocessing – similarity measured as a similar number of common words

As we have seen earlier, the bag of word approach is both fast and robust. It is, though, not without challenges. Let's dive directly into them.

Converting raw text into a bag of words

We do not have to write custom code for counting words and representing those counts as a vector. SciKit's `CountVectorizer` method does the job not only efficiently but also has a very convenient interface. SciKit's functions and classes are imported via the `sklearn` package:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(min_df=1)
```

The `min_df` parameter determines how `CountVectorizer` treats seldom words (minimum document frequency). If it is set to an integer, all words occurring less than that value will be dropped. If it is a fraction, all words that occur in less than that fraction of the overall dataset will be dropped. The `max_df` parameter works in a similar manner. If we print the instance, we see what other parameters SciKit provides together with their default values:

```
>>> print(vectorizer)
CountVectorizer(analyzer='word', binary=False, charset=None,
        charset_error=None, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8',
input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

We see that, as expected, the counting is done at word level (`analyzer=word`) and words are determined by the regular expression pattern `token_pattern`. It will, for example, tokenize "cross-validated" into "cross" and "validated". Let's ignore the other parameters for now and consider the following two example subject lines:

```
>>> content = ["How to format my hard disk", " Hard disk format
problems "]
```

We can now put this list of subject lines into the `fit_transform()` function of our vectorizer, which does all the hard vectorization work.

```
>>> X = vectorizer.fit_transform(content)
>>> vectorizer.get_feature_names()
[u'disk', u'format', u'hard', u'how', u'my', u'problems', u'to']
```

The vectorizer has detected seven words for which we can fetch the counts individually:

```
>>> print(X.toarray().transpose())
[[1 1]
 [1 1]
 [1 1]
 [1 0]
 [1 0]
 [0 1]
 [1 0]]
```

This means that the first sentence contains all the words except "problems", while the second contains all but "how", "my", and "to". In fact, these are exactly the same columns as we have seen in the preceding table. From `X`, we can extract a feature vector that we will use to compare two documents with each other.

We will start with a naïve approach first, to point out some preprocessing peculiarities we have to account for. So let's pick a random post, for which we then create the count vector. We will then compare its distance to all the count vectors and fetch the post with the smallest one.

Counting words

Let's play with the toy dataset consisting of the following posts:

Post filename	Post content
`01.txt`	This is a toy post about machine learning. Actually, it contains not much interesting stuff.
`02.txt`	Imaging databases can get huge.
`03.txt`	Most imaging databases save images permanently.
`04.txt`	Imaging databases store images.
`05.txt`	Imaging databases store images. Imaging databases store images. Imaging databases store images.

In this post dataset, we want to find the most similar post for the short post "imaging databases".

Assuming that the posts are located in the directory `DIR`, we can feed `CountVectorizer` with it:

```
>>> posts = [open(os.path.join(DIR, f)).read() for f in
os.listdir(DIR)]
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(min_df=1)
```

We have to notify the vectorizer about the full dataset so that it knows upfront what words are to be expected:

```
>>> X_train = vectorizer.fit_transform(posts)
>>> num_samples, num_features = X_train.shape
>>> print("#samples: %d, #features: %d" % (num_samples,
num_features))
#samples: 5, #features: 25
```

Unsurprisingly, we have five posts with a total of 25 different words. The following words that have been tokenized will be counted:

```
>>> print(vectorizer.get_feature_names())
[u'about', u'actually', u'capabilities', u'contains', u'data',
u'databases', u'images', u'imaging', u'interesting', u'is', u'it',
u'learning', u'machine', u'most', u'much', u'not', u'permanently',
u'post', u'provide', u'save', u'storage', u'store', u'stuff',
u'this', u'toy']
```

Now we can vectorize our new post.

```
>>> new_post = "imaging databases"
>>> new_post_vec = vectorizer.transform([new_post])
```

Note that the count vectors returned by the `transform` method are sparse. That is, each vector does not store one count value for each word, as most of those counts will be zero (the post does not contain the word). Instead, it uses the more memory-efficient implementation `coo_matrix` (for "COOrdinate"). Our new post, for instance, actually contains only two elements:

```
>>> print(new_post_vec)
  (0, 7)  1
  (0, 5)  1
```

Via its `toarray()` member, we can once again access the full `ndarray`:

```
>>> print(new_post_vec.toarray())
[[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

We need to use the full array, if we want to use it as a vector for similarity calculations. For the similarity measurement (the naïve one), we calculate the Euclidean distance between the count vectors of the new post and all the old posts:

```
>>> import scipy as sp
>>> def dist_raw(v1, v2):
...     delta = v1-v2
...     return sp.linalg.norm(delta.toarray())
```

The `norm()` function calculates the Euclidean norm (shortest distance). This is just one obvious first pick and there are many more interesting ways to calculate the distance. Just take a look at the paper *Distance Coefficients between Two Lists or Sets* in The Python Papers Source Codes, in which Maurice Ling nicely presents 35 different ones.

With `dist_raw`, we just need to iterate over all the posts and remember the nearest one:

```
>>> import sys
>>> best_doc = None
>>> best_dist = sys.maxint
>>> best_i = None
>>> for i, post in enumerate(num_samples):
...     if post == new_post:
...         continue
...     post_vec = X_train.getrow(i)
...     d = dist_raw(post_vec, new_post_vec)
...     print("=== Post %i with dist=%.2f: %s"%(i, d, post))
...     if d<best_dist:
...         best_dist = d
...         best_i = i
>>> print("Best post is %i with dist=%.2f"%(best_i, best_dist))


=== Post 0 with dist=4.00: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.
=== Post 1 with dist=1.73: Imaging databases provide storage
capabilities.
```

```
=== Post 2 with dist=2.00: Most imaging databases save images
permanently.
=== Post 3 with dist=1.41: Imaging databases store data.
=== Post 4 with dist=5.10: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=1.41
```

Congratulations, we have our first similarity measurement. Post 0 is most dissimilar from our new post. Quite understandably, it does not have a single word in common with the new post. We can also understand that Post 1 is very similar to the new post, but not the winner, as it contains one word more than Post 3, which is not contained in the new post.

Looking at Post 3 and Post 4, however, the picture is not so clear any more. Post 4 is the same as Post 3 duplicated three times. So, it should also be of the same similarity to the new post as Post 3.

Printing the corresponding feature vectors explains why:

```
>>> print(X_train.getrow(3).toarray())
[[0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]]
>>> print(X_train.getrow(4).toarray())
[[0 0 0 0 3 3 0 3 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0]]
```

Obviously, using only the counts of the raw words is too simple. We will have to normalize them to get vectors of unit length.

Normalizing word count vectors

We will have to extend `dist_raw` to calculate the vector distance not on the raw vectors but on the normalized instead:

```
>>> def dist_norm(v1, v2):
...     v1_normalized = v1/sp.linalg.norm(v1.toarray())
...     v2_normalized = v2/sp.linalg.norm(v2.toarray())
...     delta = v1_normalized - v2_normalized
...     return sp.linalg.norm(delta.toarray())
```

This leads to the following similarity measurement:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.
=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.
```

```
=== Post 2 with dist=0.92: Most imaging databases save images
permanently.
=== Post 3 with dist=0.77: Imaging databases store data.
=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=0.77
```

This looks a bit better now. Post 3 and Post 4 are calculated as being equally similar. One could argue whether that much repetition would be a delight to the reader, but from the point of counting the words in the posts this seems to be right.

Removing less important words

Let's have another look at Post 2. Of its words that are not in the new post, we have "most", "save", "images", and "permanently". They are actually quite different in the overall importance to the post. Words such as "most" appear very often in all sorts of different contexts and are called stop words. They do not carry as much information and thus should not be weighed as much as words such as "images", which doesn't occur often in different contexts. The best option would be to remove all the words that are so frequent that they do not help to distinguish between different texts. These words are called stop words.

As this is such a common step in text processing, there is a simple parameter in `CountVectorizer` to achieve that:

```
>>> vectorizer = CountVectorizer(min_df=1, stop_words='english')
```

If you have a clear picture of what kind of stop words you would want to remove, you can also pass a list of them. Setting `stop_words` to `english` will use a set of 318 English stop words. To find out which ones, you can use `get_stop_words()`:

```
>>> sorted(vectorizer.get_stop_words())[0:20]
['a', 'about', 'above', 'across', 'after', 'afterwards', 'again',
'against', 'all', 'almost', 'alone', 'along', 'already', 'also',
'although', 'always', 'am', 'among', 'amongst', 'amoungst']
```

The new word list is seven words lighter:

```
[u'actually', u'capabilities', u'contains', u'data', u'databases',
u'images', u'imaging', u'interesting', u'learning', u'machine',
u'permanently', u'post', u'provide', u'save', u'storage', u'store',
u'stuff', u'toy']
```

Without stop words, we arrive at the following similarity measurement:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.
=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.
=== Post 2 with dist=0.86: Most imaging databases save images
permanently.
=== Post 3 with dist=0.77: Imaging databases store data.
=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 3 with dist=0.77
```

Post 2 is now on par with Post 1. It has, however, changed not much overall since our posts are kept short for demonstration purposes. It will become vital when we look at real-world data.

Stemming

One thing is still missing. We count similar words in different variants as different words. Post 2, for instance, contains "imaging" and "images". It will make sense to count them together. After all, it is the same concept they are referring to.

We need a function that reduces words to their specific word stem. SciKit does not contain a stemmer by default. With the **Natural Language Toolkit** (**NLTK**), we can download a free software toolkit, which provides a stemmer that we can easily plug into `CountVectorizer`.

Installing and using NLTK

How to install NLTK on your operating system is described in detail at `http://nltk.org/install.html`. Unfortunately, it is not yet officially supported for Python 3, which means that also pip install will not work. We can, however, download the package from `http://www.nltk.org/nltk3-alpha/` and install it manually after uncompressing using Python's `setup.py` install.

To check whether your installation was successful, open a Python interpreter and type:

```
>>> import nltk
```

You will find a very nice tutorial to NLTK in the book *Python 3 Text Processing with NLTK 3 Cookbook, Jacob Perkins, Packt Publishing*. To play a little bit with a stemmer, you can visit the web page `http://text-processing.com/demo/stem/`.

NLTK comes with different stemmers. This is necessary, because every language has a different set of rules for stemming. For English, we can take `SnowballStemmer`.

```
>>> import nltk.stem
>>> s = nltk.stem.SnowballStemmer('english')
>>> s.stem("graphics")
u'graphic'
>>> s.stem("imaging")
u'imag'
>>> s.stem("image")
u'imag'
>>> s.stem("imagination")
u'imagin'
>>> s.stem("imagine")
u'imagin'
```

Note that stemming does not necessarily have to result in valid English words.

It also works with verbs:

```
>>> s.stem("buys")
u'buy'
>>> s.stem("buying")
u'buy'
```

This means, it works most of the time:

```
>>> s.stem("bought")
u'bought'
```

Extending the vectorizer with NLTK's stemmer

We need to stem the posts before we feed them into `CountVectorizer`. The class provides several hooks with which we can customize the stage's preprocessing and tokenization. The preprocessor and tokenizer can be set as parameters in the constructor. We do not want to place the stemmer into any of them, because we will then have to do the tokenization and normalization by ourselves. Instead, we overwrite the `build_analyzer` method:

```
>>> import nltk.stem
>>> english_stemmer = nltk.stem.SnowballStemmer('english'))
>>> class StemmedCountVectorizer(CountVectorizer):
...     def build_analyzer(self):
...         analyzer = super(StemmedCountVectorizer,
self).build_analyzer()
...         return lambda doc: (english_stemmer.stem(w) for w in
analyzer(doc))
>>> vectorizer = StemmedCountVectorizer(min_df=1,
stop_words='english')
```

This will do the following process for each post:

1. The first step is lower casing the raw post in the preprocessing step (done in the parent class).
2. Extracting all individual words in the tokenization step (done in the parent class).
3. This concludes with converting each word into its stemmed version.

As a result, we now have one feature less, because "images" and "imaging" collapsed to one. Now, the set of feature names is as follows:

```
[u'actual', u'capabl', u'contain', u'data', u'databas', u'imag',
u'interest', u'learn', u'machin', u'perman', u'post', u'provid',
u'save', u'storag', u'store', u'stuff', u'toy']
```

Running our new stemmed vectorizer over our posts, we see that collapsing "imaging" and "images", revealed that actually Post 2 is the most similar post to our new post, as it contains the concept "imag" twice:

```
=== Post 0 with dist=1.41: This is a toy post about machine learning.
Actually, it contains not much interesting stuff.
=== Post 1 with dist=0.86: Imaging databases provide storage
capabilities.
```

```
=== Post 2 with dist=0.63: Most imaging databases save images
permanently.
=== Post 3 with dist=0.77: Imaging databases store data.
=== Post 4 with dist=0.77: Imaging databases store data. Imaging
databases store data. Imaging databases store data.
Best post is 2 with dist=0.63
```

Stop words on steroids

Now that we have a reasonable way to extract a compact vector from a noisy textual post, let's step back for a while to think about what the feature values actually mean.

The feature values simply count occurrences of terms in a post. We silently assumed that higher values for a term also mean that the term is of greater importance to the given post. But what about, for instance, the word "subject", which naturally occurs in each and every single post? Alright, we can tell `CountVectorizer` to remove it as well by means of its `max_df` parameter. We can, for instance, set it to `0.9` so that all words that occur in more than 90 percent of all posts will always be ignored. But, what about words that appear in 89 percent of all posts? How low will we be willing to set `max_df`? The problem is that however we set it, there will always be the problem that some terms are just more discriminative than others.

This can only be solved by counting term frequencies for every post and in addition discount those that appear in many posts. In other words, we want a high value for a given term in a given value, if that term occurs often in that particular post and very seldom anywhere else.

This is exactly what **term frequency – inverse document frequency** (**TF-IDF**) does. TF stands for the counting part, while IDF factors in the discounting. A naïve implementation will look like this:

```
>>> import scipy as sp
>>> def tfidf(term, doc, corpus):
...     tf = doc.count(term) / len(doc)
...     num_docs_with_term = len([d for d in corpus if term in d])
...     idf = sp.log(len(corpus) / num_docs_with_term)
...     return tf * idf
```

You see that we did not simply count the terms, but also normalize the counts by the document length. This way, longer documents do not have an unfair advantage over shorter ones.

For the following documents, `D`, consisting of three already tokenized documents, we can see how the terms are treated differently, although all appear equally often per document:

```
>>> a, abb, abc = ["a"], ["a", "b", "b"], ["a", "b", "c"]
>>> D = [a, abb, abc]
>>> print(tfidf("a", a, D))
0.0
>>> print(tfidf("a", abb, D))
0.0
>>> print(tfidf("a", abc, D))
0.0
>>> print(tfidf("b", abb, D))
0.270310072072
>>> print(tfidf("a", abc, D))
0.0
>>> print(tfidf("b", abc, D))
0.135155036036
>>> print(tfidf("c", abc, D))
0.366204096223
```

We see that `a` carries no meaning for any document since it is contained everywhere. The `b` term is more important for the document `abb` than for `abc` as it occurs there twice.

In reality, there are more corner cases to handle than the preceding example does. Thanks to SciKit, we don't have to think of them as they are already nicely packaged in `TfidfVectorizer`, which is inherited from `CountVectorizer`. Sure enough, we don't want to miss our stemmer:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> class StemmedTfidfVectorizer(TfidfVectorizer):
...     def build_analyzer(self):
...         analyzer = super(TfidfVectorizer,
                             self).build_analyzer()
...         return lambda doc: (
                english_stemmer.stem(w) for w in analyzer(doc))
>>> vectorizer = StemmedTfidfVectorizer(min_df=1,
                  stop_words='english', decode_error='ignore')
```

The resulting document vectors will not contain counts any more. Instead they will contain the individual TF-IDF values per term.

Our achievements and goals

Our current text pre-processing phase includes the following steps:

1. Firstly, tokenizing the text.
2. This is followed by throwing away words that occur way too often to be of any help in detecting relevant posts.
3. Throwing away words that occur way so seldom so that there is only little chance that they occur in future posts.
4. Counting the remaining words.
5. Finally, calculating TF-IDF values from the counts, considering the whole text corpus.

Again, we can congratulate ourselves. With this process, we are able to convert a bunch of noisy text into a concise representation of feature values.

But, as simple and powerful the bag of words approach with its extensions is, it has some drawbacks, which we should be aware of:

- **It does not cover word relations**: With the aforementioned vectorization approach, the text "Car hits wall" and "Wall hits car" will both have the same feature vector.
- **It does not capture negations correctly**: For instance, the text "I will eat ice cream" and "I will not eat ice cream" will look very similar by means of their feature vectors although they contain quite the opposite meaning. This problem, however, can be easily changed by not only counting individual words, also called "unigrams", but instead also considering bigrams (pairs of words) or trigrams (three words in a row).
- **It totally fails with misspelled words**: Although it is clear to the human beings among us readers that "database" and "databas" convey the same meaning, our approach will treat them as totally different words.

For brevity's sake, let's nevertheless stick with the current approach, which we can now use to efficiently build clusters from.

Clustering

Finally, we have our vectors, which we believe capture the posts to a sufficient degree. Not surprisingly, there are many ways to group them together. Most clustering algorithms fall into one of the two methods: flat and hierarchical clustering.

Flat clustering divides the posts into a set of clusters without relating the clusters to each other. The goal is simply to come up with a partitioning such that all posts in one cluster are most similar to each other while being dissimilar from the posts in all other clusters. Many flat clustering algorithms require the number of clusters to be specified up front.

In hierarchical clustering, the number of clusters does not have to be specified. Instead, hierarchical clustering creates a hierarchy of clusters. While similar posts are grouped into one cluster, similar clusters are again grouped into one *uber-cluster*. This is done recursively, until only one cluster is left that contains everything. In this hierarchy, one can then choose the desired number of clusters after the fact. However, this comes at the cost of lower efficiency.

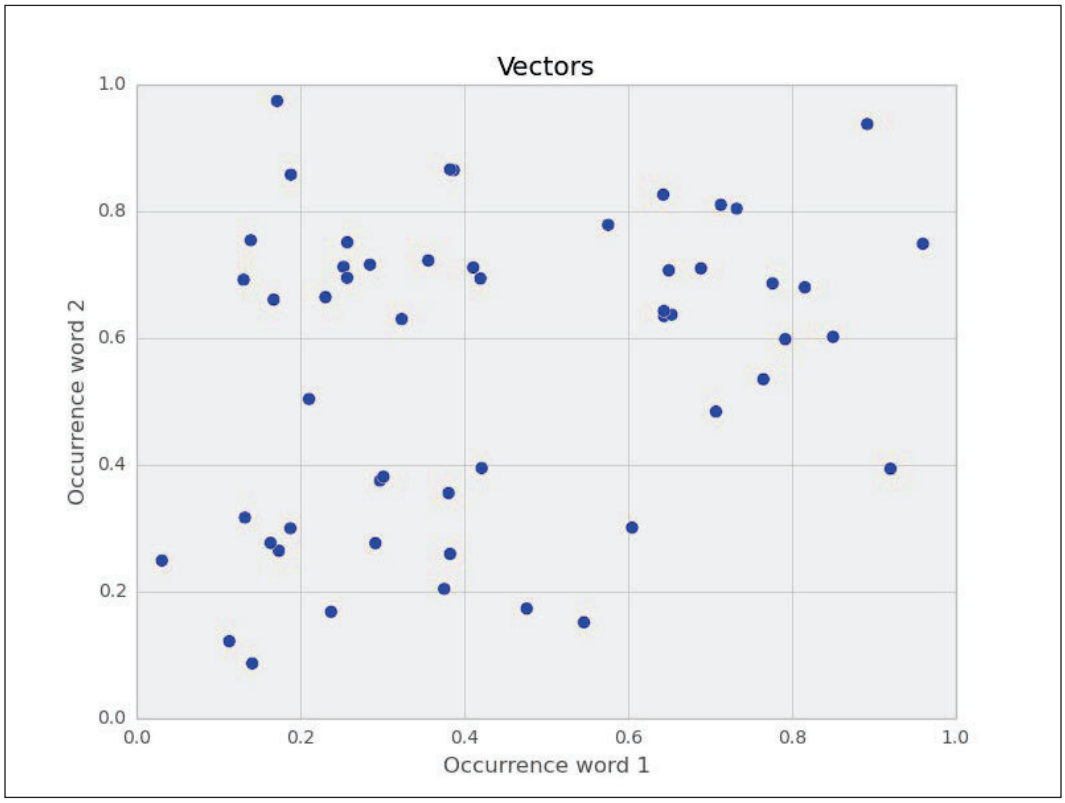
SciKit provides a wide range of clustering approaches in the `sklearn.cluster` package. You can get a quick overview of advantages and drawbacks of each of them at `http://scikit-learn.org/dev/modules/clustering.html`.

In the following sections, we will use the flat clustering method K-means and play a bit with the desired number of clusters.

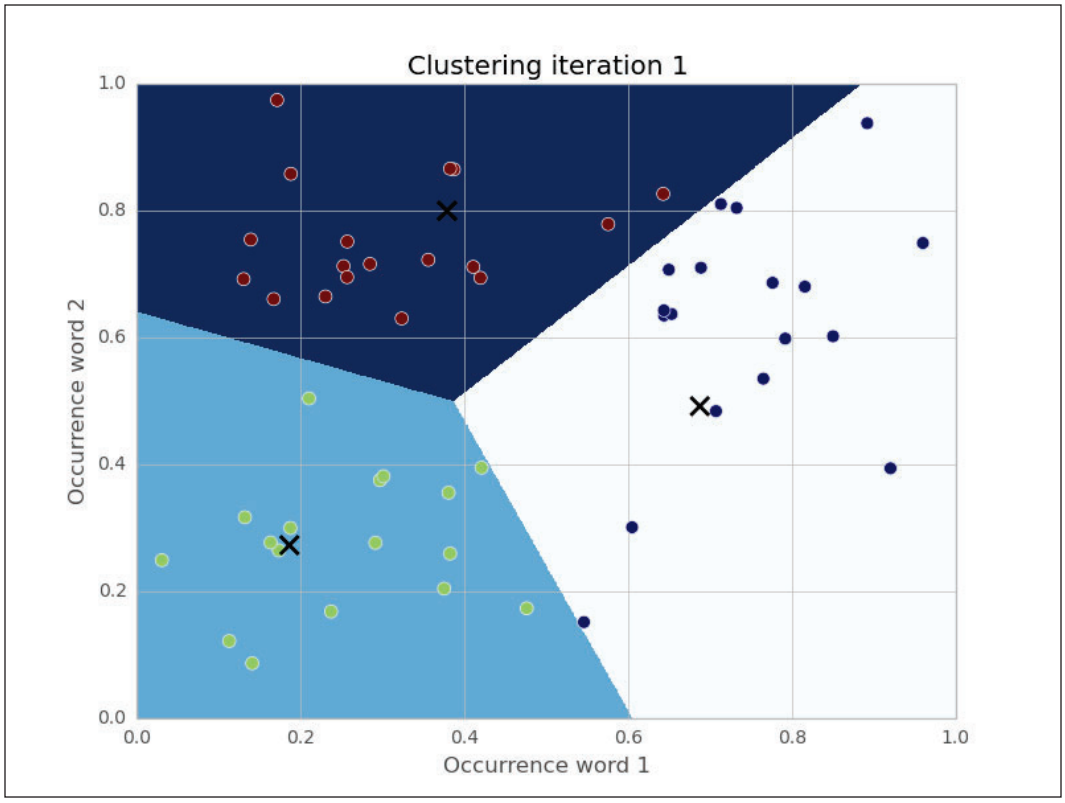
K-means

k-means is the most widely used flat clustering algorithm. After initializing it with the desired number of clusters, `num_clusters`, it maintains that number of so-called cluster centroids. Initially, it will pick any `num_clusters` posts and set the centroids to their feature vector. Then it will go through all other posts and assign them the nearest centroid as their current cluster. Following this, it will move each centroid into the middle of all the vectors of that particular class. This changes, of course, the cluster assignment. Some posts are now nearer to another cluster. So it will update the assignments for those changed posts. This is done as long as the centroids move considerably. After some iterations, the movements will fall below a threshold and we consider clustering to be converged.

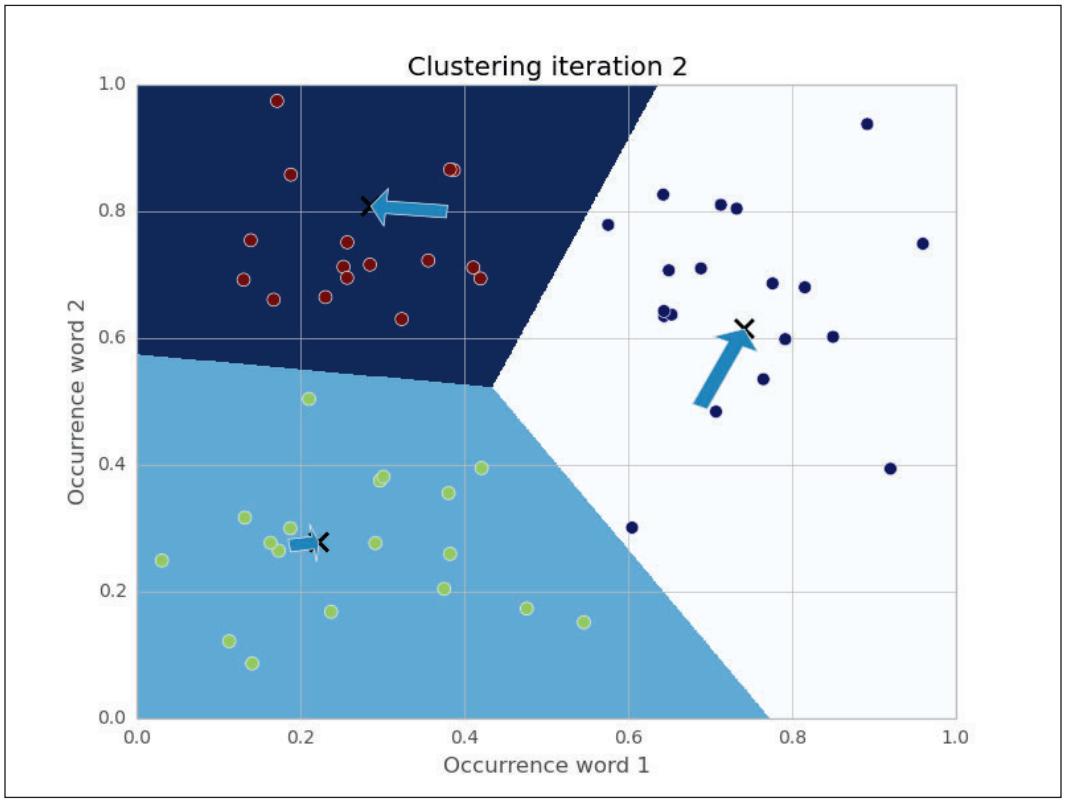
Let's play this through with a toy example of posts containing only two words. Each point in the following chart represents one document:



After running one iteration of K-means, that is, taking any two vectors as starting points, assigning the labels to the rest and updating the cluster centers to now be the center point of all points in that cluster, we get the following clustering:



Because the cluster centers moved, we have to reassign the cluster labels and recalculate the cluster centers. After iteration 2, we get the following clustering:



The arrows show the movements of the cluster centers. After five iterations in this example, the cluster centers don't move noticeably any more (SciKit's tolerance threshold is 0.0001 by default).

After the clustering has settled, we just need to note down the cluster centers and their identity. Each new document that comes in, we then have to vectorize and compare against all cluster centers. The cluster center with the smallest distance to our new post vector belongs to the cluster we will assign to the new post.

Getting test data to evaluate our ideas on

In order to test clustering, let's move away from the toy text examples and find a dataset that resembles the data we are expecting in the future so that we can test our approach. For our purpose, we need documents about technical topics that are already grouped together so that we can check whether our algorithm works as expected when we apply it later to the posts we hope to receive.

One standard dataset in machine learning is the `20newsgroup` dataset, which contains 18,826 posts from 20 different newsgroups. Among the groups' topics are technical ones such as `comp.sys.mac.hardware` or `sci.crypt`, as well as more politics- and religion-related ones such as `talk.politics.guns` or `soc.religion.christian`. We will restrict ourselves to the technical groups. If we assume each newsgroup as one cluster, we can nicely test whether our approach of finding related posts works.

The dataset can be downloaded from `http://people.csail.mit.edu/jrennie/20Newsgroups`. Much more comfortable, however, is to download it from MLComp at `http://mlcomp.org/datasets/379` (free registration required). SciKit already contains custom loaders for that dataset and rewards you with very convenient data loading options.

The dataset comes in the form of a ZIP file `dataset-379-20news-18828_WJQIG.zip`, which we have to unzip to get the directory `379`, which contains the datasets. We also have to notify SciKit about the path containing that data directory. It contains a metadata file and three directories `test`, `train`, and `raw`. The `test` and `train` directories split the whole dataset into 60 percent of training and 40 percent of testing posts. If you go this route, then you either need to set the environment variable `MLCOMP_DATASETS_HOME` or you specify the path directly with the `mlcomp_root` parameter when loading the dataset.

`http://mlcomp.org` is a website for comparing machine learning programs on diverse datasets. It serves two purposes: finding the right dataset to tune your machine learning program, and exploring how other people use a particular dataset. For instance, you can see how well other people's algorithms performed on particular datasets and compare against them.

For convenience, the `sklearn.datasets` module also contains the `fetch_20newsgroups` function, which automatically downloads the data behind the scenes:

```
>>> import sklearn.datasets
>>> all_data = sklearn.datasets.fetch_20newsgroups(subset='all')
>>> print(len(all_data.filenames))
18846
>>> print(all_data.target_names)
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',
'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles',
'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt',
'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.christian',
'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.misc',
'talk.religion.misc']
```

We can choose between training and test sets:

```
>>> train_data = sklearn.datasets.fetch_20newsgroups(subset='train',
categories=groups)
>>> print(len(train_data.filenames))
11314
>>> test_data = sklearn.datasets.fetch_20newsgroups(subset='test')
>>> print(len(test_data.filenames))
7532
```

For simplicity's sake, we will restrict ourselves to only some newsgroups so that the overall experimentation cycle is shorter. We can achieve this with the `categories` parameter:

```
>>> groups = ['comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',
'comp.windows.x', 'sci.space']
>>> train_data = sklearn.datasets.fetch_20newsgroups(subset='train',
categories=groups)
>>> print(len(train_data.filenames))
3529


>>> test_data = sklearn.datasets.fetch_20newsgroups(subset='test',
categories=groups)
>>> print(len(test_data.filenames))
2349
```

Clustering posts

You would have already noticed one thing—real data is noisy. The newsgroup dataset is no exception. It even contains invalid characters that will result in `UnicodeDecodeError`.

We have to tell the vectorizer to ignore them:

```
>>> vectorizer = StemmedTfidfVectorizer(min_df=10, max_df=0.5,
...              stop_words='english', decode_error='ignore')
>>> vectorized = vectorizer.fit_transform(train_data.data)
>>> num_samples, num_features = vectorized.shape
>>> print("#samples: %d, #features: %d" % (num_samples,
num_features))
#samples: 3529, #features: 4712
```

We now have a pool of 3,529 posts and extracted for each of them a feature vector of 4,712 dimensions. That is what K-means takes as input. We will fix the cluster size to 50 for this chapter and hope you are curious enough to try out different values as an exercise.

```
>>> num_clusters = 50
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=num_clusters, init='random', n_init=1,
verbose=1, random_state=3)
>>> km.fit(vectorized)
```

That's it. We provided a random state just so that you can get the same results. In real-world applications, you will not do this. After fitting, we can get the clustering information out of members of `km`. For every vectorized post that has been fit, there is a corresponding integer label in `km.labels_`:

```
>>> print(km.labels_)
[48 23 31 ...,  6  2 22]
>>> print(km.labels_.shape)
3529
```

The cluster centers can be accessed via `km.cluster_centers_`.

In the next section, we will see how we can assign a cluster to a newly arriving post using `km.predict`.

Solving our initial challenge

We will now put everything together and demonstrate our system for the following new post that we assign to the `new_post` variable:

> *"Disk drive problems. Hi, I have a problem with my hard disk.*
>
> *After 1 year it is working only sporadically now.*
>
> *I tried to format it, but now it doesn't boot any more.*
>
> *Any ideas? Thanks."*

As you learned earlier, you will first have to vectorize this post before you predict its label:

```
>>> new_post_vec = vectorizer.transform([new_post])
>>> new_post_label = km.predict(new_post_vec)[0]
```

Now that we have the clustering, we do not need to compare `new_post_vec` to all post vectors. Instead, we can focus only on the posts of the same cluster. Let's fetch their indices in the original data set:

```
>>> similar_indices = (km.labels_==new_post_label).nonzero()[0]
```

The comparison in the bracket results in a Boolean array, and `nonzero` converts that array into a smaller array containing the indices of the `True` elements.

Using `similar_indices`, we then simply have to build a list of posts together with their similarity scores:

```
>>> similar = []
>>> for i in similar_indices:
...    dist = sp.linalg.norm((new_post_vec -
vectorized[i]).toarray())
...    similar.append((dist, dataset.data[i]))
>>> similar = sorted(similar)
>>> print(len(similar))
131
```

We found 131 posts in the cluster of our post. To give the user a quick idea of what kind of similar posts are available, we can now present the most similar post (`show_at_1`), and two less similar but still related ones – all from the same cluster.

```
>>> show_at_1 = similar[0]
>>> show_at_2 = similar[int(len(similar)/10)]
>>> show_at_3 = similar[int(len(similar)/2)]
```

The following table shows the posts together with their similarity values:

Position	Similarity	Excerpt from post
1	1.038	BOOT PROBLEM with IDE controller Hi, I've got a Multi I/O card (IDE controller + serial/parallel interface) and two floppy drives (5 1/4, 3 1/2) and a Quantum ProDrive 80AT connected to it. I was able to format the hard disk, but I could not boot from it. I can boot from drive A: (which disk drive does not matter) but if I remove the disk from drive A and press the reset switch, the LED of drive A: continues to glow, and the hard disk is not accessed at all. I guess this must be a problem of either the Multi I/o card or floppy disk drive settings (jumper configuration?) Does someone have any hint what could be the reason for it. […]
2	1.150	Booting from B drive I have a 5 1/4" drive as drive A. How can I make the system boot from my 3 1/2" B drive? (Optimally, the computer would be able to boot: from either A or B, checking them in order for a bootable disk. But: if I have to switch cables around and simply switch the drives so that: it can't boot 5 1/4" disks, that's OK. Also, boot_b won't do the trick for me. […] […]
3	1.280	IBM PS/1 vs TEAC FD Hello, I already tried our national news group without success. I tried to replace a friend s original IBM floppy disk in his PS/1-PC with a normal TEAC drive. I already identified the power supply on pins 3 (5V) and 6 (12V), shorted pin 6 (5.25"/3.5" switch) and inserted pullup resistors (2K2) on pins 8, 26, 28, 30, and 34. The computer doesn't complain about a missing FD, but the FD s light stays on all the time. The drive spins up o.k. when I insert a disk, but I can't access it. The TEAC works fine in a normal PC. Are there any points I missed? […] […]

It is interesting how the posts reflect the similarity measurement score. The first post contains all the salient words from our new post. The second also revolves around booting problems, but is about floppy disks and not hard disks. Finally, the third is neither about hard disks, nor about booting problems. Still, of all the posts, we would say that they belong to the same domain as the new post.

Another look at noise

We should not expect a perfect clustering in the sense that posts from the same newsgroup (for example, `comp.graphics`) are also clustered together. An example will give us a quick impression of the noise that we have to expect. For the sake of simplicity, we will focus on one of the shorter posts:

```
>>> post_group = zip(train_data.data, train_data.target)
>>> all = [(len(post[0]), post[0], train_data.target_names[post[1]])
for post in post_group]
>>> graphics = sorted([post for post in all if
post[2]=='comp.graphics'])
>>> print(graphics[5])
(245, 'From: SITUNAYA@IBM3090.BHAM.AC.UK\nSubject:
test....(sorry)\nOrganization: The University of Birmingham, United
Kingdom\nLines: 1\nNNTP-Posting-Host: ibm3090.bham.ac.uk<…snip…>',
'comp.graphics')
```

For this post, there is no real indication that it belongs to `comp.graphics` considering only the wording that is left after the preprocessing step:

```
>>> noise_post = graphics[5][1]
>>> analyzer = vectorizer.build_analyzer()
>>> print(list(analyzer(noise_post)))
['situnaya', 'ibm3090', 'bham', 'ac', 'uk', 'subject', 'test',
'sorri', 'organ', 'univers', 'birmingham', 'unit', 'kingdom', 'line',
'nntp', 'post', 'host', 'ibm3090', 'bham', 'ac', 'uk']
```

This is only after tokenization, lowercasing, and stop word removal. If we also subtract those words that will be later filtered out via `min_df` and `max_df`, which will be done later in `fit_transform`, it gets even worse:

```
>>> useful = set(analyzer(noise_post)).intersection
(vectorizer.get_feature_names())
>>> print(sorted(useful))
['ac', 'birmingham', 'host', 'kingdom', 'nntp', 'sorri', 'test',
'uk', 'unit', 'univers']
```

Even more, most of the words occur frequently in other posts as well, as we can check with the IDF scores. Remember that the higher TF-IDF, the more discriminative a term is for a given post. As IDF is a multiplicative factor here, a low value of it signals that it is not of great value in general.

```
>>> for term in sorted(useful):
...     print('IDF(%s)=%.2f'%(term,
vectorizer._tfidf.idf_[vectorizer.vocabulary_[term]]))
IDF(ac)=3.51
IDF(birmingham)=6.77
IDF(host)=1.74
IDF(kingdom)=6.68
IDF(nntp)=1.77
IDF(sorri)=4.14
IDF(test)=3.83
IDF(uk)=3.70
IDF(unit)=4.42
IDF(univers)=1.91
```

So, the terms with the highest discriminative power, `birmingham` and `kingdom`, clearly are not that computer graphics related, the same is the case with the terms with lower IDF scores. Understandably, posts from different newsgroups will be clustered together.

For our goal, however, this is no big deal, as we are only interested in cutting down the number of posts that we have to compare a new post to. After all, the particular newsgroup from where our training data came from is of no special interest.

Tweaking the parameters

So what about all the other parameters? Can we tweak them to get better results?

Sure. We can, of course, tweak the number of clusters, or play with the vectorizer's `max_features` parameter (you should try that!). Also, we can play with different cluster center initializations. Then there are more exciting alternatives to K-means itself. There are, for example, clustering approaches that let you even use different similarity measurements, such as Cosine similarity, Pearson, or Jaccard. An exciting field for you to play.

But before you go there, you will have to define what you actually mean by "better". SciKit has a complete package dedicated only to this definition. The package is called `sklearn.metrics` and also contains a full range of different metrics to measure clustering quality. Maybe that should be the first place to go now. Right into the sources of the metrics package.

Summary

That was a tough ride from pre-processing over clustering to a solution that can convert noisy text into a meaningful concise vector representation, which we can cluster. If we look at the efforts we had to do to finally being able to cluster, it was more than half of the overall task. But on the way, we learned quite a bit on text processing and how simple counting can get you very far in the noisy real-world data.

The ride has been made much smoother, though, because of SciKit and its powerful packages. And there is more to explore. In this chapter, we were scratching the surface of its capabilities. In the next chapters, we will see more of its power.