6 Clustering text

In this chapter

- Basic concepts behind common text clustering algorithms
- Examples of how clustering can help improve text applications
- How to cluster words to identify topics of interest
- Clustering whole document collections using Apache Mahout and clustering search results using Carrot^2

How often have you browsed through content online and clicked through on an article that had an interesting title, but the underlying story was basically the same as the one you just finished? Or perhaps you're tasked with briefing your boss on the day's news but don't have the time to wade through all the content involved when all you need is a summary and a few key points. Alternatively, maybe your users routinely enter ambiguous or generic query terms or your data covers a lot of different topics and you want to group search results in order to save users from wading through unrelated results. Having a text processing tool that can automatically group similar items and present the results with summarizing labels is a good way to wade through large amounts of text or search results without having to read all, or even most, of the content.

In this chapter, we'll take a closer look at how to solve problems like these using a machine learning approach called *clustering*. Clustering is an unsupervised task (no human intervention, such as annotating training text, required) that can automatically put related content into buckets, helping you better organize your content or reduce the amount of content that you must manually process. In some cases, it also can assign labels to these buckets and even give summaries of what's in each bucket.

After looking at the concepts of clustering in the first section, we'll delve into how to cluster search results using a project called Carrot^2. Next up, we'll look at how Apache Mahout can be used to cluster large collections of documents into buckets. In fact, both Carrot^2 and Mahout come with several different approaches to clustering, each with their own merits and demerits. We'll also look at how clustering can be applied at the word level to identify topics in documents (sometimes called *topic modeling*) by using a technique called *Latent Dirichlet Allocation*, which also happens to be in Apache Mahout. Throughout the examples, we'll show how this can all be built on the work you did earlier with Apache Solr, making it easy to access cluster information alongside all of your other access pathways, enabling richer access to information. Finally, we'll finish the chapter with a section on performance, with an eye toward both quantity (how fast?) and quality (how good?). First, let's look at an example application that many of you are probably already familiar with but may not have known that it was an implementation of clustering: Google News.

6.1 Google News document clustering

In the age of the 24-hour news cycle with countless news outlets hawking their version of events, Google News enables readers to quickly see most of the stories published in a particular time period on a topic by grouping similar articles together. For instance, in figure 6.1, the headline "Vikings Begin Favre era on the road in Cleveland" shows there are 2,181 other similar stories to the main story. Though it's not clear what clustering algorithms Google is using to implement this feature, Google's documentation clearly states they're using clustering (Google News 2011):

> Our grouping technology takes into account many factors, such as titles, text, and publication time. We then use various clustering algorithms to identify stories we think are closely related. These stories displayed on Google News present news articles, videos, images and other information.

The power of being able to do this kind of grouping at a large scale should be obvious to anyone with an internet connection. And though there's more to the problem of grouping news content on a near-real-time basis than running a clustering algorithm over the content, having clustering implementations designed to scale like those in Apache Mahout are vital to getting off the ground.

In a task like news clustering, an application needs to be able to quickly cluster large numbers of documents, determine representative documents or labels for display, and deal with new, incoming documents. There's more to the issue than just having a good clustering algorithm, but for our purposes we'll focus on how clustering can help solve these and other unsupervised tasks that aide in the discovery and processing of information.

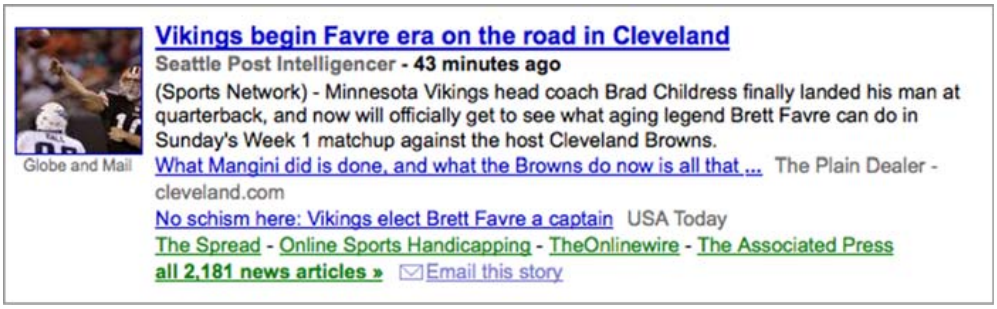

Figure 6.1 Example of clustering news documents on Google News. Captured on 09/13/2009.

6.2 Clustering foundations

Clustering boils down to grouping similar unlabeled documents together based on some similarity measure. The goal is to divide all documents in the collection that are similar into the same cluster as each other while ensuring that dissimilar documents are in different clusters. Before we begin looking into the foundations of clustering, it's important to set expectations about clustering in general. Though clustering is often useful, it's not a cure-all. The quality of the clustering experience often comes down to setting expectations for your users. If your users expect perfection, they'll be disappointed. If they expect something that will, for the most part, help them wade through large volumes of data quickly while still dealing with false positives, they'll likely be happier. From an application designer standpoint, a fair amount of testing may be required to find the right settings for striking a balance between speed of execution and quality of results. Ultimately, remember your goal is to encourage discovery and serendipitous interaction with your content, not necessarily perfectly similar items.

With the basic definition and admonitions out of the way, the remaining foundational sections below will examine

- Different types of text clustering can be applied to
- How to choose a clustering algorithm
- Ways to determine similarity
- Approaches to identifying labels
- How to evaluate clustering results

6.2.1 Three types of text to cluster

Clustering can be applied to many different aspects of text, including the words in a document, the documents themselves, or the results from doing searches. Clustering is also useful for many other things besides text, like grouping users or data from a series of sensors, but those are outside the scope of this book. For now, we'll focus on three types of clustering: document, search result, and word/topic.

In document clustering, the focus is on grouping documents as a whole together, as in the Google News example given earlier. Document clustering is typically done as

an offline batch processing job and the output is typically a list of documents and a centroid vector. Since document clustering is usually a batch processing task, it's often worthwhile to spend the extra time (within reason) to get better results. Descriptions of the clusters are often generated by looking at the most important terms (determined by some weighting mechanism such as TF-IDF) in documents closest to the centroid. Some preprocessing is usually required to remove stopwords and to stem words, but neither of these are necessarily a given for all algorithms. Other common text techniques such as identifying phrases or using *n*-grams may also be worth experimenting with when testing approaches. To read more on document clustering, see *An Introduction to Information Retrieval* (Manning 2008) for starters.

For search result clustering, the clustering task, given a user query, is to do a search and group the results of the search into clusters. Search result clustering can be quite effective when users enter generic or ambiguous terms (such as *apple*) or when the dataset contains a disparate set of categories. Search result clustering is often characterized by several factors:

- Clustering on short snippets of text (title, maybe a small section of the body where the query terms matched).
- Algorithms designed to work on small sets of results and to return as fast as possible.
- Labels take on more significance, since users will likely treat them like facets to make decisions about how to further navigate the result set.

Preprocessing is often done just as in document clustering, but since labels are usually more significant, it may make sense to spend the extra time to identify frequently occurring phrases. For an overview of search result clustering, see "A Survey of Web Clustering Engines" (Carpineto 2009).

Clustering words into topics, otherwise called *topic modeling*, is an effective way to quickly find the topics that are covered in a large set of documents. The approach is based on the assumption that documents often cover several different topics and that words related to a given topic are often found near each other. By clustering the words, you can quickly see which words appear near each other and then also what documents are attached to those words. (In a sense, the approach also does document clustering.) For instance, the output from running a topic modeling algorithm (see section 6.6) yields the clustered words in table 6.1 (formatted for display from the original).

Table 6.1 Example topics and words

Topic 0	Topic 1
win saturday time game know nation u more after two take over back has from texa first day man offici 2 high one sinc some sunday	yesterday game work new last over more most year than two from state after been would us polic peopl team run were open five american

In this example, the first thing to notice is that the topics themselves lack names. Naming the topic is the job of the person generating the topics. Next, you don't even know what documents contain which topic. So why bother? Generating the topics for a collection is one more way to aid users in browsing a collection and discovering interesting information about the collection without having to read the whole collection. Additionally, there has been some more recent work on better characterizing the topics through phrases (see Blei [2009]).

To learn more on topic modeling, start with the references at the end of this chapter and also http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation, which will lead you to the primary academic papers written on the topic.

Now that you have some groundwork for the types of text you want to cluster, let's take a look at the factors that play into the selection of a clustering algorithm.

6.2.2 Choosing a clustering algorithm

Many different algorithms are available for clustering, and covering them all is beyond the scope of this book. For instance, as of this writing, Apache Mahout contains implementations of K-Means (demonstrated later), Fuzzy K-Means, Mean-Shift, Dirichlet, Canopy, Spectral, and Latent Dirichlet Allocation, and there will no doubt be more by the time this book is published. Instead of digging too much into how each is implemented, let's look at some of the more common features of clustering algorithms in order to better understand what criteria are helpful in picking a clustering algorithm.

In discussing clustering algorithms, there are many aspects to examine to determine what's going to work best for an application. Traditionally, one of the main deciding factors has been whether the algorithm is hierarchical in nature or flat. As the name implies, *hierarchical* approaches work either top-down or bottom-up, building a hierarchy of related documents that can be broken down into smaller and smaller sets. Flat approaches are usually much faster since they don't have to relate the clusters to other clusters. Also keep in mind that some flat algorithms can be modified to be hierarchical.

Moving beyond hierarchical versus flat, table 6.2 provides details on many other factors that play out when choosing a clustering approach.

Table 6.2 Clustering algorithm choices

Characteristic	Description
Cluster membership (soft/hard)	Hard—Documents belong to one and only one cluster. Soft—Documents can be in more than one cluster and often have an associated probability of membership.
Updateable	Can the clusters be updated when new documents are added or does the entire calculation need to be re-executed?
Probabilistic approach	Understanding the underpinnings of an approach will help you know the benefits and failings of such an approach.

Table 6.2 Clustering algorithm choices *(continued)*

Characteristic	Description
Speed	The runtime of most flat clustering approaches is linear in the number of documents, whereas many hierarchical approaches are nonlinear.
Quality	Hierarchical approaches are often more accurate than flat ones, at the cost of execution time. More on evaluation in section 6.2.5.
Handles feedback	Can the algorithm adjust/improve based on user feedback? For instance, if a user marked a document as not appropriate for a cluster, can the algorithm exclude that document? Does it change other clusters?
Number of clusters	Some algorithms require an application to decide on the number of clusters up front; others pick the appropriate number as part of the algorithm. If the algorithm requires this to be specified, expect to experiment with this value to obtain good results.

Individual algorithms also have their own quirks that need to be considered when evaluating what approach to take, but table 6.2 should provide some overall guidance on choosing an algorithm. From here, expect to spend some time evaluating the various approaches to determine which works best for your data.

6.2.3 Determining similarity

Many clustering algorithms contain a notion of similarity that's used to determine whether a document belongs to a cluster. Similarity, in many clustering algorithms, is implemented as a measure of the distance between two documents. In order for these distance measures to work, most systems represent documents as vectors (almost always sparse—meaning most entries are zero) where each cell in the vector is the weight of that particular term for that particular document. The weight can be any value the application wants, but is typically some variation on TF-IDF. If this all sounds vaguely familiar to you, it should, as the approaches for weighting documents for clustering are similar to those used for searching. To remind yourself of these concepts, see section 3.2.3.

In practice, document vectors are almost always normalized first using a p-norm ($p >= 0$) so that really short and really long documents don't affect the results in a negative way. Normalizing by a p-norm just means dividing each vector by its length, thereby scaling all the vectors onto the unit shape (for example, the 2-norm's is a unit circle). The most common norms used, and the ones most readers will be familiar with, are the 1-norm (Manhattan distance) and 2-norm (Euclidean distance). You'll notice that the examples later in the chapter use the Euclidean distance for normalizing our vectors. For more information on p-norms, see http://en.wikipedia.org/wiki/Norm_(mathematics).

After the vectors are created, it's then reasonable to measure the distance between two documents as the distance between two vectors. There are many different distance measures available, so we'll focus on the few most common ones:

- *Euclidean distance*—The tried and true "as the crow flies" distance between any two points. Variations include the squared Euclidean distance (saving a square root calculation) and one that can weight parts of the vector.
- *Manhattan distance*—Otherwise known as the *taxicab distance*, as it represents the distance traveled if one were driving a taxicab in a city laid out on a grid, as in Manhattan in New York City. Sometimes the parts of the calculation may be weighted.
- *Cosine distance*—Takes the cosine of the angle formed by putting the tails of two vectors together; so two similar documents (angle == 0) have a cosine equal to 1. See section 3.2.3.

As you'll see in the Apache Mahout section later, the distance measure is often a parameter that can be passed in, allowing experimentation with different measures. As to which distance measure to use, you should use one that corresponds to the normalization applied to the vector. For instance, if you used the 2-norm, then the Euclidean or Cosine distances would be most appropriate. That being said, though theoretically incorrect, some approaches will work without such an alignment.

For probabilistic approaches, the question of similarity is really a question of the probability that a given document is in a cluster. They often have a more sophisticated model of how documents are related based on statistical distributions and other properties. Also, some of the distance-based approaches (K-Means) can be shown to be probabilistic.

6.2.4 Labeling the results

Because clustering is often used in discovery tools with real users, picking good labels and/or good representative documents is often as important to a clustering-based application as determining the clusters themselves. Without good labels and representative documents, users will be discouraged from interacting with the clusters to find and discover useful documents.

Picking representative documents from a cluster can be done in several ways. At the most basic, documents can be selected randomly, giving users a wider mix of results, and potentially lead to new discoveries, but also, if the documents are far from the center, failing to capture what the cluster is about. To remedy this, documents can be picked based on their proximity to the cluster's centroid or their likelihood of membership. In this way, documents are likely good indicators of what the cluster is about, but may lose some of the serendipity associated with random selection. This leads to a dual approach where some documents are picked randomly and some are picked based on proximity/probability.

Picking good labels, or topics, is more difficult than picking representative documents, and there are many approaches, each with their pros and cons. In some applications, simple techniques akin to faceting (see chapter 3) can be used to effectively showcase the frequency of tags common to a cluster, but most applications are better served by finding and displaying important terms and phrases in the cluster. Of course, what's deemed important is a subject of ongoing research. One simple approach is to leverage the weights in the vector (say, using our friend TF-IDF; see section 3.2.3) and return a list of the terms sorted by weight. Using *n*-grams, this can be extended to return a list of phrases (technically they're phrases, but they may not be of high quality) based on their weights in the collection/cluster. Another common approach is to do some conceptual/topic modeling through techniques that utilize singular value decomposition, such as latent semantic analysis (see Deerwester [1990]) or Latent Dirichlet Allocation (see Blei [2003], demonstrated later in the chapter). Another useful approach is to use the log-likelihood ratio (LLR; see Dunning [1993]) of terms that are in the cluster versus those outside of the cluster. The underlying math for the approaches other than TF-IDF (which we've already discussed) is beyond the scope of this book. But in using this chapter's tools (Carrot[2] and Apache Mahout), you'll see demonstrations of all of these approaches either implicitly as part of the algorithm itself or explicitly via a specific tool. Regardless of how you get labels, the results will be useful in understanding the quality of your clusters, which is the subject of the next section.

6.2.5 How to evaluate clustering results

As with any text processing tool, experimenting and evaluating clustering results should be as much a part of building the application as designing the architecture or figuring out how to deploy it. Just as in search, named entity recognition, and the other concepts in this book, clustering can be evaluated in a number of ways.

The first approach most people use is the *laugh test*, otherwise known as the *smell test*. Do these clusters look reasonable when viewed by a person with some knowledge of what a good result should be? Though you should never read too much into the smell test, it's nevertheless an invaluable part of the process and one that usually catches "dumb" mistakes like bad or missing input parameters. The downsides are that it's impossible to replicate a person's reaction on demand and in a repeatable manner, not to mention it's only one person's opinion. It's also, for better or worse, dependent on the label generation process, which may not capture the clusters accurately.

Taking a few people and having them rate the results is often the next step up testing-wise. Whether it's a quality assurance team or a group of target users, correlating the reactions of a small group of people can provide valuable feedback. The cost is the time and expense involved in arranging these tests, not to mention the human error and lack of on-demand repeatability. But if done several times, a gold standard can be derived, which is the next approach.

A gold standard is a set of clusters created by one or more people that are interpreted as the ideal set of results for the clustering task. Once constructed, this set can then be compared against clustering results from various experiments. Creating a gold standard is often impractical, brittle (dealing with updates, new documents, and so on), or prohibitively expensive for large datasets. If you know your collection isn't going to change much and if you have the time, creating a gold standard, perhaps on a subset of the total, may be worthwhile. One semi-automated approach is to run one or more of the clustering algorithms and then have one or more people adjust the clusters manually to arrive at the final result. When the judgments are in place, a number of formulas (purity, normalized mutual information, Rand index, and F-measure) can be used to sum up the results into a single metric that indicates the quality of the clustering. Rather than lay out their formulas here, we'll refer the interested reader to section 16.3 of *An Introduction to Information Retrieval* (Manning 2008), where proper treatment is given these measures.

Finally, some mathematical tools are available that can help evaluate clusters. These tools are all heuristics for evaluating clustering and don't require human input. They shouldn't be used in isolation, but instead as helpful indicators of clustering quality. The first measure is calculated by randomly removing some subset of the input data and then running the clustering. After the clustering is complete, calculate the percentage of documents in each cluster out of the total number of points and set it aside. Next, add back in the random data, rerun the clustering, and recalculate the percentage. Given that the held back data was randomly distributed, you'd expect it to roughly conform to the distribution from the first set. If cluster A had 50% of the documents in the first set, it's reasonable to expect it (but not guaranteed) to still hold 50% of the documents in the larger set.

From information theory (see http://en.wikipedia.org/wiki/Information_theory for starters) come several other useful measures that may help assess clustering quality. The first is the notion of *entropy*. Entropy is a measure of the uncertainty of a random variable. In practical terms, it's a measure of the information contained in a cluster. For text-based clustering, you can build on the entropy and calculate the perplexity, which measures how well cluster membership predicts what words are used.

There's plenty more to learn about clustering. For those interested in learning more about the concepts behind clustering, a good starting point is *An Introduction to Information Retrieval* (Manning 2008). In particular, chapters 16 and 17 focus at a deeper level on the concepts discussed here. Cutting et al. also provide good information on how to utilize clustering in discovery in their paper "Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections" (Cutting 1992). For now, we'll continue on and look at how to run several clustering implementations, including one for search results and another for document collections.

6.3 Setting up a simple clustering application

For the discussions in the following sections on clustering, we'll demonstrate the concepts using the content from several news websites via their RSS/Atom feeds. To that end, we've set up a simple Solr Home (schema, config, and so forth), located under the solr-clustering directory, that ingests the feeds from several newspapers and news organizations. This instance will rely on Solr's Data Import Handler to automatically ingest and index the feeds into the schema. From these feeds, we can then demonstrate the various clustering libraries discussed in the following sections.

Building on the search knowledge gained in chapter 3, the three primary Solr pieces of interest for our new clustering application are schema.xml, rss-data-config.xml, and the addition of the Data Import Handler to solrconfig.xml. For the schema and the RSS configuration, we examined the content from the various feeds and mapped that into a few common fields, which were then indexed. We also stored term vectors, for reasons shown later in section 6.5.1.

The details of the Data Import Handler (DIH) configuration can be found on Solr's wiki at http://wiki.apache.org/solr/DataImportHandler. To run Solr with the clustering setup from the *Taming Text* source, execute the following commands in the source distribution root directory:

- `cd apache-solr/example`
- `./bin/start-solr.sh solr-clustering`
- Invoke the Data Import Handler import command: http://localhost:8983/solr/dataimport?command=full-import
- Check the status of the import: http://localhost:8983/solr/dataimport?command=status

With this basic setup, we can now begin to demonstrate clustering in action using the data indexed from the feeds we just described. We'll start with Carrot^2 for search results and then look into document collection clustering using Apache Mahout.

6.4 Clustering search results using Carrot^2

Carrot^2 is an open source search results clustering library released under a BSD-like license and found at http://project.carrot2.org/. It's specifically designed for delivering high-performance results on typical search results (say, a title and small snippet of text). The library comes with support for working with a number of different search APIs, including Google, Yahoo!, Lucene, and Solr (as a client) as well as the ability to cluster documents in XML or those created programmatically. Additionally, the Solr project has integrated Carrot^2 into the server side, which we'll demonstrate later.

Carrot^2 comes with two clustering implementations: STC (suffix tree clustering) and Lingo.

STC was first introduced for web search result clustering by Zamir and Etzioni in "Web document clustering: a feasibility demonstration" (Zamir 1998). The algorithm is based on the suffix tree data structure, which can be used to efficiently (linear time)

identify common substrings. Efficiently finding common substrings is one of the keys to quickly finding labels for clusters. To read more on suffix trees, start with http://en.wikipedia.org/wiki/Suffix_tree.

The Lingo algorithm was created by Stanisław Osiński and Dawid Weiss (the creators of the Carrot^2 project). At a high level, Lingo uses singular value decomposition (SVD; see http://en.wikipedia.org/wiki/Singular_value_decomposition to learn more) to find good clusters and phrase discovery to identify good labels for those clusters.

Carrot^2 also comes with a user interface that can be used for experimenting with your own data, and a server implementation supporting REST that makes it easy to interact with Carrot^2 via other programming languages. Finally, if so inclined, an application may add its own clustering algorithm into the framework via a well-defined API. To explore Carrot^2 in greater depth, refer to the manual at http://download.carrot2.org/head/manual/.

For the remainder of this section, we'll focus on showing how to use the API to cluster a data source and then look at how Carrot^2 is integrated into Solr. We'll finish the section with a look at performance both in terms of quality and speed for both of the algorithms.

6.4.1 Using the Carrot^2 API

Carrot^2 architecture is implemented as a pipeline. Content is ingested from a document source and then handed off to one or more components that modify and cluster the sources, outputting the clusters at the other end. In terms of actual classes, at its most basic, the pipeline consists of one or more `IProcessingComponents` that are controlled by the `IController` implementation. The controller handles initializing the components and invoking the components in the correct order and with the appropriate inputs. Examples of `IProcessingComponent` implementations include the various document sources (`GoogleDocumentSource`, `YahooDocumentSource`, `LuceneDocumentSource`) as well as the clustering implementations themselves: `STCClusteringAlgorithm` and `LingoClusteringAlgorithm`.

Naturally, a bunch of other pieces get used by the implementation to do things like tokenize and stem the text. As for the controller, there are two implementations: `SimpleController` and `CachingController`. The `SimpleController` is designed for easy setup and one-time use, whereas the `CachingController` is designed for use in production environments where it can take advantage of the fact that queries are often repeated and therefore cache the results.

To see Carrot^2 in action, let's look at some sample code that clusters some simple documents. The first step is to create some documents. For Carrot^2, documents contain three elements: a title, a summary/snippet, and a URL. Given a set of documents with these characteristics, it's straightforward to cluster them, as is demonstrated in the next listing.

Listing 6.1 Simple Carrot² example

```
//... setup some documents elsewhere
final Controller controller =
        ControllerFactory.createSimple();                    <-- Create IController.
documents = new ArrayList<Document>();
for (int i = 0; i < titles.length; i++) {
  Document doc = new Document(titles[i], snippets[i],
          "file://foo_" + i + ".txt");
  documents.add(doc);
}
final ProcessingResult result = controller.process(documents,
        "red fox",
        LingoClusteringAlgorithm.class);                     <-- Cluster documents.
displayResults(result);                                      <-- Print out clusters.
```

Running listing 6.1 yields the following results:

```
Cluster: Lamb
    Mary Loses Little Lamb.  Wolf At Large.
    March Comes in like a Lamb
Cluster: Lazy Brown Dogs
    Red Fox jumps over Lazy Brown Dogs
    Lazy Brown Dogs Promise Revenge on Red Fox
```

Though the documents of the example are obviously made up (see Carrot2ExampleTest.java in the source for their construction), the code effectively demonstrates the simplicity of using the Carrot² APIs. Moving beyond the simple case, many applications will want to use the `CachingController` for performance reasons. As the name implies, the `CachingController` caches as much of the results as possible in order to improve performance. Applications may also want to use other data sources (such as Google or Yahoo!) or implement their own `IDataSource` to represent their content. Additionally, many of the components come with a variety of attributes that can be set to tune/alter both the speed and quality of results, which we'll discuss in section 6.7.2.

Now that you have an idea of how to implement some of the basics of clustering with Carrot², we can take a look at how it integrates with Solr.

6.4.2 Clustering Solr search results using Carrot²

As of version 1.4, Apache Solr adds full support for search result clustering using Carrot², including the ability to configure, via the solrconfig.xml file, all of the component attributes and the algorithms used for clustering. Naturally, Carrot² uses the Solr search results to cluster on, allowing the application to define which fields are used to represent the title, snippet, and URL. In fact, this has already been set up and configured in the *Taming Text* source distribution under the solr-clustering directory.

There are three parts to configuring Solr to use the Carrot² clustering component. First, the component is implemented as a `SearchComponent`, which means it can be plugged into a Solr `RequestHandler`. The XML to configure this component looks like this:

```
<searchComponent
  class="org.apache.solr.handler.clustering.ClusteringComponent"
  name="cluster">
    <lst name="engine">
      <str name="name">default</str>
      <str name="carrot.algorithm"><lineArrow/>
      org.carrot2.clustering.lingo.LingoClusteringAlgorithm</str>
    </lst>
    <lst name="engine">
      <str name="name">stc</str>
      <str name="carrot.algorithm"><lineArrow/>
      org.carrot2.clustering.stc.STCClusteringAlgorithm</str>
    </lst>
  </searchComponent>
```

In the `<searchComponent>` declaration, you set up the `ClusteringComponent` and then tell it which Carrot^2 clustering algorithms to use. In this case, we set up both the Lingo and the STC clustering algorithms. The next step is to hook the `SearchComponent` into a `RequestHandler`, like this:

```
<requestHandler name="standard"
  class="solr.StandardRequestHandler" default="true">
    <!-- default values for query parameters -->
    <!-- ... -->
    <arr name="last-components">
      <str>cluster</str>
    </arr>
  </requestHandler>
```

Finally, it's often useful in Solr to set up some intelligent defaults so that all of the various parameters need not be passed in on the command line. In our example, we used this:

```
<requestHandler name="standard"
  class="solr.StandardRequestHandler" default="true">
    <!-- default values for query parameters -->
    <lst name="defaults">
    <!-- ... -->
      <!-- Clustering -->
      <!--<bool name="clustering">true</bool>-->
       <str name="clustering.engine">default</str>
       <bool name="clustering.results">true</bool>
       <!-- The title field -->
       <str name="carrot.title">title</str>
       <!-- The field to cluster on -->
       <str name="carrot.snippet">desc</str>
       <str name="carrot.url">link</str>
       <!-- produce summaries -->
       <bool name="carrot.produceSummary">false</bool>
       <!-- produce sub clusters -->
       <bool name="carrot.outputSubClusters">false</bool>
    </lst>
  </requestHandler>
```

In the configuration of the default parameters, we declared that Solr should use the default clustering engine (Lingo) and that Carrot^2 should use the Solr title field as the

Carrot^2 title, the Solr description field as the Carrot^2 snippet field, and the Solr link field as the Carrot^2 URL field. Lastly, we told Carrot^2 to produce summaries but to skip outputting subclusters. (For the record, subclusters are created by clustering within a single cluster.)

That's all the setup needed! Assuming Solr was started as outlined in listing 6.3, asking Solr for search result clusters is as simple as adding the `&clustering=true` parameter to the URL, as in http://localhost:8983/solr/select/?q=*:*&clustering=true&rows=100. Executing that command will result in Solr retrieving 100 documents from the index and clustering them. A screenshot of some of the results of running this clustering query appears in figure 6.2.

At the bottom of figure 6.2, we purposefully left in a junk result of *R Reuters sportsNews 4* to demonstrate the need for proper tuning of Carrot^2 via the various

Figure 6.2 **A screenshot of running a Solr clustering command**

attributes available, which will be discussed in section 6.7.2. See http://wiki.apache.org/solr/ClusteringComponent for a full accounting of the options available for tuning the clustering component in Solr.

Now that you have an understanding of how to cluster search results, let's move on and take a look at how to cluster whole document collections using Apache Mahout. We'll revisit Carrot[2] later in the chapter when we look at performance.

6.5 Clustering document collections with Apache Mahout

Apache Mahout is an Apache Software Foundation project with the goal of developing a suite of machine learning libraries designed from the ground up to be scalable to large numbers of input items. As of this writing, it contains algorithms for classification, clustering, collaborative filtering, evolutionary programming, and more, as well as useful utilities for solving machine learning problems such as manipulating matrices and storing Java primitives (Maps, Lists, and Sets for storing ints, doubles, and so on). In many cases, Mahout relies on the Apache Hadoop (http://hadoop.apache.org) framework (via the MapReduce programming model and a distributed filesystem called *HDFS*) for developing algorithms designed to scale. And though much of this chapter focuses on clustering with Mahout, chapter 7 covers classification with Mahout. The other parts of Mahout can be discovered on its website at http://mahout.apache.org/ and in *Mahout in Action* (see http://manning.com/owen/). To get started for this section, you'll need to download Mahout 0.6 from http://archive.apache.org/dist/mahout/0.6/mahout-distribution-0.6.tar.gz and unpack it into a directory, which we'll call $MAHOUT_HOME from here on out. After you download it and unpack it, change into the $MAHOUT_HOME directory and run `mvn install -DskipTests` (you can run the tests, but they take a long time!).

Without further ado, the next three sections examine how to prepare your data and then cluster it using Apache Mahout's implementation of the K-Means algorithm.

Apache Hadoop—The yellow elephant with big computing power

Hadoop is an implementation of ideas put forth by Google (see Dean [2004]), first implemented in the Lucene project Nutch, and since spun out to be its own project at the Apache Software Foundation. The basic idea is to pair a distributed filesystem (called *GFS* by Google and *HDFS* by Hadoop) with a programming model (MapReduce) that makes it easy for engineers with little-to-no background in parallel and distributed systems to write programs that are both scalable and fault tolerant to run on very large clusters of computers.

Though not all applications can be written in the MapReduce model, many text-based applications are well suited for the approach.

For more information on Apache Hadoop, see *Hadoop: The Definitive Guide* (http://oreilly.com/catalog/9780596521981) by Tom White or *Hadoop in Action* (http://manning.com/lam/) by Chuck Lam.

6.5.1 Preparing the data for clustering

For clustering, Mahout relies on data to be in an `org.apache.mahout.matrix.Vector` format. A `Vector` in Mahout is simply a tuple of floats, as in <0.5, 1.9, 100.5>. More generally speaking, a vector, often called a *feature vector*, is a common data structure used in machine learning to represent the properties of a document or other piece of data to the system. Depending on the data, vectors are often either densely populated or sparse. For text applications, vectors are often sparse due to the large number of terms in the overall collection, but the relatively few terms in any particular document. Thankfully, sparseness often has its advantages when computing common machine learning tasks. Naturally, Mahout comes with several implementations that extend `Vector` in order to represent both sparse and dense vectors. These implementations are named `org.apache.mahout.matrix.SparseVector` and `org.apache.mahout.matrix.DenseVector`. When running your application, you should sample your data to determine whether it's spare or dense and then choose the appropriate representation. You can always try both on subsets of your data to determine which performs best.

Mahout comes with several different ways to create `Vectors` for clustering:

- *Programmatic*—Erite code that instantiates the `Vector` and then saves it to an appropriate place.
- *Apache Lucene index*—Transforms an Apache Lucene index into a set of `Vectors`.
- *Weka's ARFF format*—Weka is a machine learning project from the University of Waikato (New Zealand) that defines the ARFF format. See http://cwiki.apache.org/MAHOUT/creating-vectors-from-wekas-arff-format.html for more information. For more information on Weka, see *Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)* (http://www.cs.waikato.ac.nz/~ml/weka/book.html) by Witten and Frank.

Since we're not using Weka in this book, we'll forgo coverage of the ARFF format here and focus on the first two means of producing `Vectors` for Mahout.

PROGRAMMATIC VECTOR CREATION

Creating `Vectors` programmatically is straightforward and best shown by a simple example, as shown here.

Listing 6.2 Vector creation using Mahout

```
double[] vals = new double[]{0.3, 1.8, 200.228};
Vector dense = new DenseVector(vals);
assertTrue(dense.size() == 3);
Vector sparseSame = new SequentialAccessSparseVector(3);
Vector sparse = new SequentialAccessSparseVector(3000);
for (int i = 0; i < vals.length; i++) {
  sparseSame.set(i, vals[i]);
```

Create DenseVector with label of my-dense and 3 values. The cardinality of this vector is 3.

Create SparseVector with a label of my-sparse-same that has cardinality of 3.

Create SparseVector with a label of my-sparse and a cardinality of 3000.

Set values to first 3 items in sparse vectors.

```
    sparse.set(i, vals[i]);
  }
  assertFalse(dense.equals(sparse));
  assertEquals(dense, sparseSame);
  assertFalse(sparse.equals(sparseSame));
```

The dense and sparse Vectors aren't equal because they have different cardinality.

The dense and sparse Vectors are equal because they have the same values and cardinality.

Vectors are often created programmatically when reading data from a database or some other source that's not supported by Mahout. When a `Vector` is constructed, it needs to be written to a format that Mahout understands. All of the clustering algorithms in Mahout expect one or more files in Hadoop's SequenceFile format. Mahout provides the `org.apache.mahout.utils.vectors.io.SequenceFileVectorWriter` to assist in serializing `Vectors` to the proper format. This is demonstrated in the following listing.

Listing 6.3 Serializing vectors to a `SequenceFile`

```
File tmpDir = new File(System.getProperty("java.io.tmpdir"));
File tmpLoc = new File(tmpDir, "sfvwt");
tmpLoc.mkdirs();
File tmpFile = File.createTempFile("sfvwt", ".dat", tmpLoc);

Path path = new Path(tmpFile.getAbsolutePath());
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
SequenceFile.Writer seqWriter = SequenceFile.createWriter(fs, conf,
        path, LongWritable.class, VectorWritable.class);
VectorWriter vecWriter = new SequenceFileVectorWriter(seqWriter);
List<Vector> vectors = new ArrayList<Vector>();
vectors.add(sparse);
vectors.add(sparseSame);
vecWriter.write(vectors);
vecWriter.close();
```

Create Hadoop SequenceFile.Writer to handle the job of physically writing out the vectors to a file in HDFS.

Create Configuration for Hadoop.

A VectorWriter processes the Vectors and invokes the underlying write methods on SequenceFile.Writer.

Do work of writing out files.

Mahout can also write out `Vectors` to JSON, but doing so is purely for human-readability needs as they're slower to serialize and deserialize at runtime, and slow down the clustering algorithms significantly. Since we're using Solr, which uses Apache Lucene under the hood, the next section on creating vectors from a Lucene index is much more interesting.

CREATING VECTORS FROM AN APACHE LUCENE INDEX

One or more Lucene indexes are a great source for creating `Vectors`, assuming the field to be used for `Vector` creation was created with the `termVector="true"` option set in the schema, as in this code:

```
<field name="description" type="text"
            indexed="true" stored="true"
             termVector="true"/>
```

Given an index, we can use Mahout's Lucene utilities to convert the index to a `SequenceFile` containing `Vectors`. This conversion can be handled on the command

line by running the `org.apache.mahout.utils.vector.lucene.Driver` program. Though the `Driver` program has many options, table 6.3 outlines the more commonly used ones.

Table 6.3 Lucene index conversion options

Argument	Description	Required
`--dir <Path>`	Specifies the location of the Lucene index.	Yes
`--output <Path>`	The path to output the `SequenceFile` to on the filesystem.	Yes
`--field <String>`	The name of the Lucene `Field` to use as the source.	Yes
`--idField <String>`	The name of the Lucene `Field` containing the unique ID of the document. Can be used to label the vector.	No
`--weight [tf|tfidf]`	The type of weight to be used for representing the terms in the field. TF is term frequency only; TF-IDF uses both term frequency and inverse document frequency.	No
`--dictOut <Path>`	The location to output the mapping between terms and their position in the vector.	Yes
`--norm [INF|-1|A double >= 0]`	Indicates how to normalize the vector. See http://en.wikipedia.org/wiki/Lp_norm.	No

To put this in action in the context of our Solr instance, we can point the driver at the directory containing the Lucene index and specify the appropriate input parameters, and the driver will do the rest of the work. For demonstration purposes, we'll assume Solr's index is stored in <Solr Home>/data/index and that it has been created as shown earlier in the chapter. You might generate your `Vectors` by running the driver as in the next listing.

Listing 6.4 Sample `Vector` creation from a Lucene index

```
<MAHOUT_HOME>/bin/mahout lucene.vector

    --dir <PATH>/solr-clustering/data/index
    --output /tmp/solr-clust-n2/part-out.vec --field description
    --idField id --dictOut /tmp/solr-clust-n2/dictionary.txt --norm 2
```

In the example in listing 6.4, the driver program ingests the Lucene index, grabs the necessary document information from the index, and writes it out to the part-out.dat (the *part* is important for Mahout/Hadoop) file. The dictionary.txt file that's also created will contain a mapping between the terms in the index and the position in the vectors created. This is important for re-creating the vectors later for display purposes.

Finally, we chose the 2-norm here, so that we can cluster using the `CosineDistanceMeasure` included in Mahout. Now that we have some vectors, let's do some clustering using Mahout's K-Means implementation.

6.5.2 K-Means clustering

There are many different approaches to clustering, both in the broader machine learning community and within Mahout. For instance, Mahout alone, as of this writing, has clustering implementations called

- Canopy
- Mean-Shift
- Dirichlet
- Spectral
- K-Means and Fuzzy K-Means

Of these choices, K-Means is easily the most widely known. K-Means is a simple and straightforward approach to clustering that often yields good results relatively quickly. It operates by iteratively adding documents to one of k clusters based on the distance, as determined by a user-supplied distance measure, between the document and the centroid of that cluster. At the end of each iteration, the centroid may be recalculated. The process stops after there's little-to-no change in the centroids or some maximum number of iterations have passed, since otherwise K-Means isn't guaranteed to converge. The algorithm is kicked off by either seeding it with some initial centroids or by randomly choosing centroids from the set of vectors in the input dataset. K-Means does have some downsides. First and foremost, you must pick k and naturally you'll get different results for different values of k. Furthermore, the initial choice for the centroids can greatly affect the outcome, so you should be sure to try different values as part of several runs. In the end, as with most techniques, it's wise to run several iterations with various parameters to determine what works best for your data.

Running the K-Means clustering algorithm in Mahout is as simple as executing the `org.apache.mahout.clustering.kmeans.KMeansDriver` class with the appropriate input parameters. Thanks to the power of Hadoop, you can execute this in either standalone mode or distributed mode (on a Hadoop cluster). For the purposes of this book, we'll use standalone mode, but there isn't much difference for distributed mode.

Instead of looking at the options that `KMeansDriver` takes first, let's go straight to an example using the `Vector` dump we created earlier. The next listing shows an example command line for running the `KMeansDriver`.

Listing 6.5 Example of using the `KMeansDriver` command-line utility

```
<$MAHOUT_HOME>/bin/mahout kmeans \
      --input /tmp/solr-clust-n2/part-out.vec \
      --clusters /tmp/solr-clust-n2/out/clusters -k 10 \
      --output /tmp/solr-clust-n2/out/ --distanceMeasure \
```

```
org.apache.mahout.common.distance.CosineDistanceMeasure \
--convergenceDelta 0.001 --overwrite --maxIter 50 --clustering
```

Most of the parameters should be self-explanatory, so we'll focus on the six main inputs that drive K-Means:

- `--k`—The *k* in K-Means. Specifies the number of clusters to be returned.
- `--distanceMeasure`—Specifies the distance measure to be used for comparing documents to the centroid. In this case, we used the Cosine distance measure (similar to how Lucene/Solr works, if you recall). Mahout comes with several that are located in the `org.apache.mahout.common.distance` package.
- `--convergenceDelta`—Defines the threshold below which clusters are considered to be converged and the algorithm can exit. Default is 0.5. Our choice of 0.001 was purely arbitrary. Users should experiment with this value to determine the appropriate time-quality trade-offs.
- `--clusters`—The path containing the "seed" centroids to cluster around. If `--k` isn't explicitly specified, this path must contain a file with *k* `Vectors` (serialized as described in listing 6.3). If `--k` is specified, then *k* random vectors will be chosen from the input.
- `--maxIter`—Specifies the maximum number of iterations to run if the algorithm doesn't converge before then.
- `--clustering`—Take the extra time to output the members of each cluster. If left off, only the centroids of the clusters are determined.

When running the command in listing 6.5, you should see a bunch of logging messages go by and (hopefully) no errors or exceptions. Upon completion, the output directory should contain several subdirectories containing the output from each iteration (named clusters-*X*, where *X* is the iteration number) as well as the input clusters (in our case, they were randomly generated) and the points that map to the final iteration's cluster output.

Since Hadoop sequence files themselves are the output, they're not human-readable in their raw form. But Mahout comes with a few utilities for viewing the results from a clustering run. The most useful of these tools is the `org.apache.mahout.utils.clustering.ClusterDumper`, but the `org.apache.mahout.utils.ClusterLabels`, `org.apache.mahout.utils.SequenceFileDumper`, and `org.apache.mahout.utils.vectors.VectorDumper` can also be useful. We'll focus on the `ClusterDumper` here. As you can probably guess from the name, the `ClusterDumper` is designed to dump out the clusters created to the console window or a file in a human-readable format. For example, to view the results of running the `KMeansDriver` command given earlier, try this:

```
<MAHOUT_HOME>/bin/mahout clusterdump \
 --seqFileDir /tmp/solr-clust-n2/out/clusters-2 \
 --dictionary /tmp/solr-clust-n2/dictionary.txt --substring 100 \
 --pointsDir /tmp/solr-clust-n2/out/points/
```

In this representative example, we told the program where the directory containing the clusters (`--seqFileDir`), the dictionary (`--dictionary`), and the original points (`--pointsDir`) were. We also told it to truncate the printing of the cluster vector center to 100 characters (`--substring`) so that the result is more legible. The output from running this on an index created based on July 5, 2010, news yields is shown in the following code:

```
:C-129069: [0:0.002, 00:0.000, 000:0.002, 001:0.000, 0011:0.000, \
  002:0.000, 0022:0.000, 003:0.000, 00
  Top Terms:
          time                                    =>0.027667414950403202
          a                                       => 0.02749764550779345
          second                                  => 0.01952658941437323
          cup                                     =>0.018764212101531803
          world                                   =>0.018431212697043415
          won                                     =>0.017260178342226474
          his                                     => 0.01582891691616071
          team                                    =>0.015548434499094444
          first                                   =>0.014986381107308856
          final                                   =>0.014441638909228182
:C-129183: [0:0.001, 00:0.000, 000:0.003, 00000000235:0.000, \
 001:0.000, 002:0.000, 01:0.000, 010:0.00
  Top Terms:
          a                                       => 0.05480601091954865
          year                                    =>0.029166628670521253
          after                                   =>0.027443270009727756
          his                                     =>0.027223628226736487
          polic                                   => 0.02445617250281346
          he                                      =>0.023918227316575336
          old                                     => 0.02345876269515748
          yearold                                 =>0.020744182153039508
          man                                     =>0.018830109266458044
          said                                    =>0.018101838778995336
...
```

In this example output, the `ClusterDumper` outputs the ID of the cluster's centroid vector along with some of the common terms in the cluster based on term frequency. Close examination of the top terms reveals that though there are many good terms, there are also some bad ones, such as a few stopwords (a, his, said, and so on). We'll gloss over this for now and revisit it later in section 6.7.

Though simply dumping out the clusters is often useful, many applications need succinct labels that summarize the contents of the clusters, as discussed earlier in section 6.2.4. Mahout's `ClusterLabels` class is a tool for generating labels from a Lucene (Solr) index and can be used to provide a list of words that best describe the clusters. To run the `ClusterLabels` program on the output from our earlier clustering run, execute the following on the command line in the same directory the other commands were run:

```
<MAHOUT_HOME>/bin/mahout \
org.apache.mahout.utils.vectors.lucene.ClusterLabels \
--dir /Volumes/Content/grantingersoll/data/solr-clustering/data/index/\
```

```
--field desc-clustering --idField id \
--seqFileDir /tmp/solr-clust-n2/out/clusters-2  \
--pointsDir /tmp/solr-clust-n2/out/clusteredPoints/ \
--minClusterSize 5 --maxLabels 10
```

In this example, we told the program many of the same things we did to extract the content from the index, such as the location of the index and the fields to use. We also added information about where the clusters and points live. The `minClusterSize` parameter sets a threshold for how many documents must be in a cluster in order to calculate the labels. This will come in handy for clustering really large collections with large clusters, as the application may want to ignore smaller clusters by treating them as outliers. The `maxLabels` parameter indicates the maximum number of labels to get for the cluster. Running this on our sample of data created earlier in the chapter yields (shortened for brevity) this:

```
Top labels for Cluster 129069 containing 15306 vectors
Term             LLR             In-ClusterDF           Out-ClusterDF
team            8060.366745727311               3611            2768
cup             6755.711004478377               2193            645
world           4056.4488459853746              2323            2553
reuter          3615.368447394372               1589            1058
season          3225.423844734556               2112            2768
olymp           2999.597569386533               1382            1004
championship    1953.5632186210423              963             781
player          1881.6121935029223              1289            1735
coach           1868.9364836380992              1441            2238
u               1545.0658127206843              35              7101

Top labels for Cluster 129183 containing 12789 vectors
Term             LLR             In-ClusterDF           Out-ClusterDF
polic           13440.84178933248               3379            550
yearold         9383.680822917435               2435            427
old             8992.130047334154               2798            1145
man             6717.213290851054               2315            1251
kill            5406.968016825078               1921            1098
year            4424.897345832258               4020            10379
charg           3423.4684087312926              1479            1289
arrest          2924.1845144664694              1015            512
murder          2706.5352747719735              735             138
death           2507.451017449319               1016            755
...
```

In the output, the columns are

- *Term*—The label.
- *LLR (log-likelihood ratio)*—The LLR is used to score how good the term is based on various statistics in the Lucene index. For more on LLR, see http://en.wikipedia.org/wiki/Likelihood-ratio_test.
- *In-ClusterDF*—The number of documents the term occurs in that are in the cluster. Both this and the Out-ClusterDF are used in calculating the LLR.
- *Out-ClusterDF*—The number of documents the term occurs in that are not in the cluster.

As in the case of the `ClusterDumper` top terms, closer inspection reveals some good terms (ignoring the fact that they're stemmed) and some terms of little use. It should be noted that most of the terms do a good job of painting a picture of what the overall set of documents in the cluster are about. As mentioned earlier, we'll examine how to improve things in section 6.7. For now, let's look at how we can use some of Mahout's clustering capabilities to identify topics based on clustering the words in the documents.

6.6 Topic modeling using Apache Mahout

Just as Mahout has tools for clustering documents, it also has an implementation for topic modeling, which can be thought of, when applied to text, as clustering at the word level. Mahout's only topic modeling implementation is of the Latent Dirichlet Allocation (LDA) algorithm. LDA (Deerwester 1990) is a

> ...generative probabilistic model for collections of discrete data such as text corpora. LDA is a three-level hierarchical Bayesian model, in which each item of a collection is modeled as a finite mixture over an underlying set of topics. Each topic is, in turn, modeled as an infinite mixture over an underlying set of topic probabilities.

In laymen's terms, LDA is an algorithm that converts clusters of words into topics based on the assumption that the underlying documents are on a number of different topics, but you're not sure which document is about which topic, nor are you sure what the topic is actually labeled. Though on the surface this may sound less than useful, there's value in having the topic words associated with your collection. For instance, they could be used in conjunction with Solr to build more discovery capabilities into the search application. Alternatively, they can be used to succinctly summarize a large document collection. The topic terms may also be used for other tasks like classification and collaborative filtering (see Deerwester [1990] for more on these applications). For now, let's take a look at how to run Mahout's LDA implementation.

To get started using LDA in Mahout, you need some vectors. As outlined earlier after listing 6.3, you can create vectors from a Lucene index. But for LDA you need to make one minor change to use just term frequencies (TF) for weight instead of the default TF-IDF, due to the way the algorithm calculates its internal statistics. This might look like this:

```
<MAHOUT_HOME>/bin/mahout lucene.vector \
--dir <PATH TO INDEX>/solr-clustering/data/index/ \
--output /tmp/lda-solr-clust/part-out.vec \
--field desc-clustering --idField id \
--dictOut /tmp/lda-solr-clust/dictionary.txt \
--norm 2 --weight TF
```

This example is nearly identical to the one generated earlier, with the exception of the different output paths and the use of the TF value for the `--weight` input parameter.

With vectors in hand, the next step is to run the LDA algorithm, like this:

```
<MAHOUT_HOME>/bin/mahout lda --input /tmp/lda-solr-clust/part-out.vec \
--output /tmp/lda-solr-clust/output --numTopics 30 --numWords 61812
```

Though most of the parameters should be obvious, a few things are worth calling out. First, we gave the application some extra memory. LDA is a fairly memory-intensive application, so you may need to give it more memory. Next, we asked the LDADriver to identify 30 topics (`--numTopics`) in the vectors. Similar to K-Means, LDA requires you to specify how many items you want created. For better or worse, this means you'll need to do some trial and error to determine the appropriate number of topics for your application. We chose 30 rather unscientifically after looking at the results for 10 and 20. Last, the `--numWords` parameter is the number of words in all the vectors. When using the vector creation methods outlined here, the number of words can easily be retrieved from the first line of the dictionary.txt file. After running LDA, the output directory will contain a bunch of directories named *state-**, as in state-1, state-2, and so on. The number of directories will depend on the input and the other parameters. The highest-numbered directory represents the final result.

Naturally, after running LDA, you'll want to see the results. LDA doesn't, by default, print out these results. But Mahout comes with a handy tool for printing out the topics, appropriately named `LDAPrintTopics`. It takes three required input parameters and one optional parameter:

- `--input`—The state directory containing the output from the LDA run. This can be any state directory, not necessarily the last one created. Required.
- `--output`—The directory to write the results. Required.
- `--dict`—The dictionary of terms used in creating the vectors. Required.
- `--words`—The number of words to print per topic. Optional.

For the example run of LDA shown earlier, `LDAPrintTopics` was run as

```
java -cp "*" \
        org.apache.mahout.clustering.lda.LDAPrintTopics \
        --input ./lda-solr-clust/output/state-118/  \
        --output lda-solr-clust/topics \
        --dict lda-solr-clust/dictionary.txt --words 20
```

In this case, we wanted the top 20 words in the state-118 directory (which happens to be the final one). Running this command fills the topics output directory with 30 files, one for each topic. Topic 22, for example, looks like this:

```
Topic 22
===========
yearold
old
cowboy
texa
14
second
year
manag
```

```
3414
quarter
opera
girl
philadelphia
eagl
arlington
which
dalla
34
counti
five
differ
1996
tri
wide
toni
regul
straight
stadium
romo
twitter
```

In looking at the top words in this category, the topic is likely about the fact that the Dallas Cowboys beat the Philadelphia Eagles in the NFL playoffs on the day before we ran the example. And, though some of the words seem like outliers (opera, girl), for the most part you get the idea of the topic. Searching in the index for some of these terms reveals there are in fact articles about just this event, including one about the use of Twitter by Eagles receiver DeSean Jackson to predict the Eagles would beat the Cowboys (don't these athletes ever learn?) That's all you need to know about running LDA in Apache Mahout. Next up, let's take a look at clustering performance across Carrot[2] and Mahout.

6.7 Examining clustering performance

As with any real-world application, when the programmer has a basic understanding of how to run the application, their mind quickly turns to how to use the application in production. To answer that question, we need to look at both qualitative and quantitative measures of performance. We'll start by looking at feature selection and reduction to improve quality and then look at algorithm selection and input parameters for both Carrot[2] and Apache Mahout. We'll finish by doing some benchmarking on Amazon's (http://aws.amazon.com) compute-on-demand capability called *EC2*. For the Amazon benchmarking, we've enlisted the help of two contributors, Timothy Potter and Szymon Chojnacki, and set out to process a fairly large collection of email content and see how Mahout performs across a number of machines.

6.7.1 Feature selection and reduction

Feature selection and feature reduction are techniques designed to either improve the quality of results or reduce the amount of content to be processed. Feature selection focuses on choosing good features up front, either as part of preprocessing or

algorithm input, whereas feature reduction focuses on removing features that contribute little value as part of an automated process. Both techniques are often beneficial for a number of reasons, including those listed here:

- Reducing the size of the problem to something more tractable both in terms of computation and storage
- Improving quality by reducing the amount of noise, such as stopwords, in the data
- Visualization and post-processing—too many features can clog up user interfaces and downstream processing techniques

In many respects, you're already familiar with feature reduction thanks to our work in chapter 3. In that chapter, we employed several analysis techniques such as stopword removal and stemming to reduce the number of terms to be searched. These techniques are also helpful for improving results for clustering. Moreover, in clustering, it's often beneficial to be even more aggressive in feature selection, since you're often processing very large sets of documents, and reductions up front can make for large savings.

For instance, for the examples in this chapter, we used a different stopword file than the one used in the chapter on search, with the difference being that the clustering stopwords (see stopwords-clustering.txt in the source) are a superset of the original. To build the stopwords-clustering.txt, we examined the list of the most frequently occurring terms in the index and also ran several iterations of clustering to determine what words we thought were worth removing.

Unfortunately, this approach is ad hoc in nature and can require a fair amount of work to produce. It also isn't portable across languages or necessarily even different corpora. To make it more portable, applications generally try to focus on removing terms based on term weights (using TF-IDF or other approaches) and then iteratively looking at some measure to determine whether the clusters improved. For example, see the References section (Dash [2000], Dash [2002], and Liu [2003]) for a variety of approaches and discussions. You can also use an approach based on singular value decomposition (SVD), which is integrated into Mahout, to significantly reduce the size of the input. Note, also, that Carrot^2's Lingo algorithm is built on SVD out of the box, so there's nothing you need to do for Carrot^2.

Singular value decomposition is a general feature reduction technique (meaning it isn't limited to just clustering) designed to reduce the dimensionality (typically in text clustering, each unique word represents a cell in an *n*-dimensional vector) of the original dataset by keeping the "important" features (words) and throwing out the unimportant features. This reduction is a lossy process, so it's not without risk, but generally speaking it can make for a significant savings in storage and CPU costs. As for the notion of importance, the algorithm is often likened to extracting the concepts in a corpus, but that isn't guaranteed. For those mathematically inclined, SVD is the factorization of the matrix (our documents, for clustering, are represented as a matrix) into its eigenvectors and other components. We'll leave the details of the

math to others; readers can refer to https://cwiki.apache.org/confluence/display/MAHOUT/Dimensional+Reduction for more details on Mahout's implementation as well as links to several tutorials and explanations of SVD.

To get started with Mahout's singular value decomposition, we can again rely on the `bin/mahout` command-line utility to run the algorithm. Running SVD in Mahout is a two-step process. The first step decomposes the matrix and the second step does some cleanup calculations. The first step of running SVD on our clustering matrix (created earlier) might look like this:

```
<MAHOUT_HOME>/bin/mahout svd --input /tmp/solr-clust-n2/part-out.vec \
--tempDir /tmp/solr-clust-n2/svdTemp \
--output /tmp/solr-clust-n2/svdOut \
--rank 200 --numCols 65458 --numRows  130103
```

In this example, we have the usual kinds of inputs, like the location of the input vectors (`--Dmapred.input.dir`) and a temporary location to be used by the system (`--tempDir`), as well as some SVD-specific items:

- `--rank`—Specifies the rank of the output matrix.
- `--numCols`—This is the total number of columns in the vector. In this case, it's the number of unique terms in the corpus, which can be found at the top of the /tmp/solr-clust-n2/dictionary.txt file.
- `--numRows`—The total number of vectors in the file. This is used for sizing data structures appropriately.

Of these options, rank is the one that determines what the outcome will look like, and is also the hardest to pick a good value for. Generally speaking, some trial and error is needed, starting with a small number (say, 50) and then increasing. According to Jake Mannix (Mannix 2010, July), Mahout committer and the original author of Mahout's SVD code, a rank in the range of 200-400 is good for text problems. Obviously, several trial runs will need to be run to determine which rank value yields the best results.

After the main SVD algorithm has been run, a single cleanup task must also be run to produce the final output, as shown here:

```
<MAHOUT_HOME>/bin/mahout cleansvd \
--eigenInput /tmp/solr-clust-n2/svdOut \
--corpusInput /tmp/solr-clust-n2/part-out.vec \
--output /tmp/solr-clust-n2/svdFinal --maxError 0.1 \
--minEigenvalue 10.0
```

The keys to this step are picking the error threshold (`--maxError`) and the minimum eigenvalue (`--minEigenValue`) items. For the minimum eigenvalue, it's always safe to choose 0, but you may wish to choose a higher value. For the maximum error value, trial and error along with use of the output in the clustering algorithm will lead to insight into how well it performs (see Mannix [2010, August] for more details).

As you can see, there are many ways to select features or reduce the size of the problem. As with most things of this nature, experimentation is required to determine

what will work best in your situation. Finally, if you want to use the results of the SVD in clustering (that's the whole point, right?), there's one final step. You need to do a multiplication of the transpose of the original matrix with the transpose of the SVD output. These can all be done using the `bin/mahout` command, making sure to get the input arguments correct. We'll leave it as an exercise to the reader to verify this. For now, we'll move on and look at some of the quantitative performance aspects of both Carrot^2 and Apache Mahout.

6.7.2 Carrot^2 performance and quality

When it comes to performance and quality of results, Carrot^2 provides a myriad of tuning options, not to mention several considerations for which algorithm to pick in the first place. We'll take a brief look at algorithm performance here, but leave an in-depth discussion of all the parameter options to the Carrot^2 manual.

Picking a Carrot^2 algorithm

First and foremost, both the STC and the Lingo algorithms have one thing in common: documents may belong to more than one cluster. Beyond that, the two algorithms take different approaches under the hood in order to arrive at their results. Generally speaking, Lingo produces better labels than STC, but at the cost of being much slower, as can be seen in figure 6.3.

As you can see in the figure, Lingo is a lot slower than STC, but for smaller result sizes, the quality of the labels may be worth the longer running time. Also, keep in

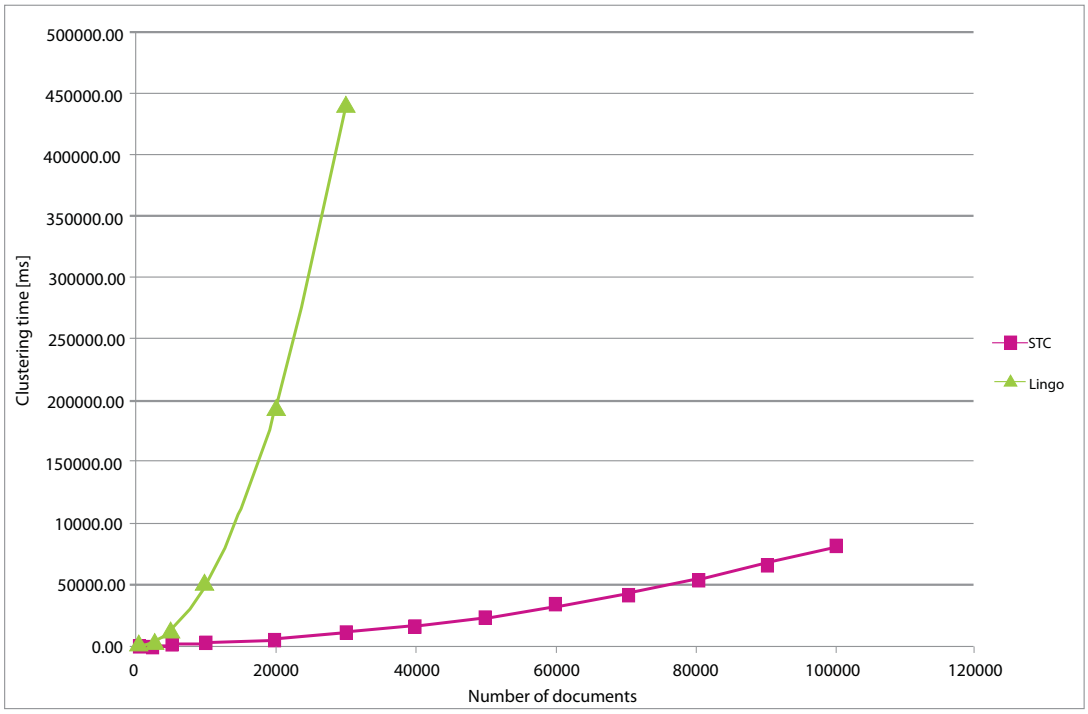

Figure 6.3 A comparison of STC versus Lingo over a variety of document sizes run on the Open Directory Project Data (http://www.dmoz.org)

mind Carrot^2 can link with some native code matrix libraries to help speed up the Lingo matrix decomposition. For applications where performance is more important, we recommend starting with STC. If quality is more important, then start with Lingo. In either case, take the time to flesh out which attributes help the most on your data. See http://download.carrot2.org/head/manual/index.html#chapter.components for a full accounting of the Carrot^2 attributes.

6.7.3 Mahout clustering benchmarks

One of Mahout's strongest attributes is the ability to distribute its computation across a grid of computers thanks to its use of Apache Hadoop. To demonstrate this, we ran K-Means and other clustering algorithms on Amazon's Elastic MapReduce (http://aws.amazon.com/elasticmapreduce/) and EC2 instances using an increasing number of instances (machines) to benchmark Mahout's scalability.

PREPARATION

As discussed in section 6.5.1, the mail archives must be transformed into Mahout vectors. The preparation steps can be done on your local workstation and don't require a Hadoop cluster. The prep_asf_mail_archives.sh script (in the utils/bin directory) in the Mahout distribution does the following:

- Download the files from s3://asf-mail-archives/ and extract using `tar`.
- Convert extracted directories containing gzipped mail archives into Hadoop SequenceFiles using a custom utility based on Mahout's `seqdirectory` utility. (See `org.apache.mahout.text.SequenceFilesFromMailArchives` in the Mahout source.) Each file contains multiple mail messages; we split the messages and extract the subject and body text using regular expressions. All other mail headers are skipped, as they provide little value for clustering. Each message is appended to a block-compressed SequenceFile, resulting in 6,094,444 key-value pairs in 283 files taking around 5.7 GB of disk.

HADOOP SETUP We performed all benchmarking work described in this section with Mahout 0.4 on Hadoop 0.20.2 using Amazon EC2. Specifically, we used EC2 `xlarge` instances deployed using the contrib/ec2 scripts provided in the Hadoop distribution. We allocated three reducers per node (mapred.reduce.tasks = n*3) with 4 GB max. heap per child process (mapred.child.java.opts = -Xmx4096M). The Hadoop contrib/ec2 scripts allocate an extra node for the NameNode, which we don't include in our cluster sizes—a 4-node cluster actually has five running EC2 instances. Detailed instructions on how to set up a Hadoop cluster to run Mahout are available on the Mahout Wiki at https://cwiki.apache.org/confluence/display/MAHOUT/Use+an+Existing+Hadoop+AMI.

VECTORIZING CONTENT

The SequenceFiles need to be converted into sparse vectors using Mahout's `seq2sparse` MapReduce job. We chose sparse vectors because most mail messages are short and we have many unique terms across all messages. Using the default

`seq2sparse` configuration produces vectors with several million dimensions, as each unique term in the corpus represents a cell in an *n*-dimensional vector. Clustering vectors of this magnitude isn't feasible and is unlikely to produce useful results given the long tail of unique terms within the mail archives.

To reduce the number of unique terms, we developed a custom Lucene analyzer that's more aggressive than the default `StandardAnalyzer`. Specifically, the `MailArchivesClusteringAnalyzer` uses a broader set of stopwords, excludes non-alphanumeric tokens, and applies porter stemming. We also leveraged several feature reduction options provided by `seq2sparse`. The following command shows how we launched the vectorization job:

```
bin/mahout seq2sparse \ --input
s3n://ACCESS_KEY:SECRET_KEY@asf-mail-archives/mahout-0.4/sequence-files

/ \
--output /asf-mail-archives/mahout-0.4/vectors/ \
--weight tfidf \ --minSupport 500 \ --maxDFPercent 70 \
--norm 2 \ --numReducers 12 \ --maxNGramSize 1 \
--analyzerName org.apache.mahout.text.MailArchivesClusteringAnalyzer
```

For input, we used Hadoop's S3 Native protocol (s3n) to read the SequenceFiles directly from S3. Note that you must include your Amazon Access and Secret Key values in the URI so that Hadoop can access the asf-mail-archives bucket. If you're unsure about these values, please see the EC2 page on the Mahout Wiki.

> **AUTHOR'S NOTE** The asf-mail-archives bucket no longer exists due to potential abuse by malicious users. We're keeping the commands here for historical accuracy given that's what was used to generate the performance metrics based on Mahout 0.4, and due to the fact that we're out of credits on Amazon and these benchmarks are an expensive undertaking! In order for you to produce similar results, you can use Amazon's public dataset containing a newer version of the ASF public mail Archives. These archives are located at http://aws.amazon.com/datasets/7791434387204566.

Most of the parameters have already been discussed, so we'll concentrate on the ones that we found to be important for clustering:

- `--minSupport 500`—Excludes terms that don't occur at least 500 times across all documents. For smaller corpora, 500 may be too high and may exclude important terms.
- `--maxDFPercent 70`—Excludes terms that occur in 70% or more documents, which helps remove any mail-related terms that were missed during text analysis.
- `--norm 2`—The vectors are normalized using the 2-norm, as we'll be using Cosine distance as our similarity measure during clustering.
- `--maxNGramSize 1`—Only consider single terms.

With these parameters, `seq2sparse` created 6,077,604 vectors with 20,444 dimensions in about 40 minutes on a 4-node cluster. The number of vectors differs from the

number of input documents because empty vectors are excluded from the `seq2sparse` output. After running the job, the resulting vectors and dictionary files are copied to a public S3 bucket so that we don't need to re-create them each time we run the clustering job.

We also experimented with generating bigrams (`--maxNGramSize=2`). Unfortunately, this made the vectors too large with roughly ~380K dimensions. In addition, creating collocations between terms greatly impacts the performance of the `seq2sparse` job; the job takes roughly 2 hours and 10 minutes to create bigrams, with at least half the time spent calculating collocations.

K-MEANS CLUSTERING BENCHMARKS

To begin clustering, we need to copy the vectors from S3 into HDFS using Hadoop's `distcp` job; as before, we use Hadoop's S3 Native protocol (s3n) to read from S3:

```
hadoop distcp -Dmapred.task.timeout=1800000 \
          s3n://ACCESS_KEY:SECRET_KEY@BUCKET/asf-mail-archives/mahout-0

 .4/sparse-1-gram-stem/tfidf-vectors \
            /asf-mail-archives/mahout-0.4/tfidf-vectors
```

This should only take a few minutes depending on the size of your cluster and doesn't incur data transfer fees if you launched your EC2 cluster in the default us-east-1 region. After the data is copied to HDFS, launch Mahout's K-Means job using the following command:

```
bin/mahout kmeans \ -i /asf-mail-archives/mahout-0.4/tfidf-vectors/ \
            -c /asf-mail-archives/mahout-0.4/initial-clusters/ \
            -o /asf-mail-archives/mahout-0.4/kmeans-clusters \
            --numClusters 60 --maxIter 10 \
            --distanceMeasure org.apache.mahout.common.distance.CosineD

 istanceMeasure \
            --convergenceDelta 0.01
```

This job begins by creating 60 random centroids using Mahout's `RandomSeedGenerator`, which takes about 9 minutes to run on the master server only (it's not a distributed MapReduce job). After the job completes, we copy the initial clusters to S3 to avoid having to re-create them for each run of the benchmarking job, which works as long as *k* stays the same. Our selection of 0.01 for the `convergenceDelta` was chosen to ensure the K-Means job completes at least 10 iterations for benchmarking purposes; at the end of 10 iterations, 59 of 60 clusters converged. As was discussed in section 6.5.2, we use Mahout's `clusterdump` utility to see the top terms in each cluster.

To determine the scalability of Mahout's K-Means MapReduce implementation, we ran the K-Means clustering job in clusters of 2, 4, 8, and 16 nodes, with three reducers per node. During execution, the load average stays healthy (< 4) and the nodes don't swap. The graph in figure 6.4 demonstrates that the results are nearly linear, as we hoped they'd be.

Each time we double the number of the nodes, we can see an almost two-fold reduction in the processing time. But the curve flattens slightly as the number of nodes

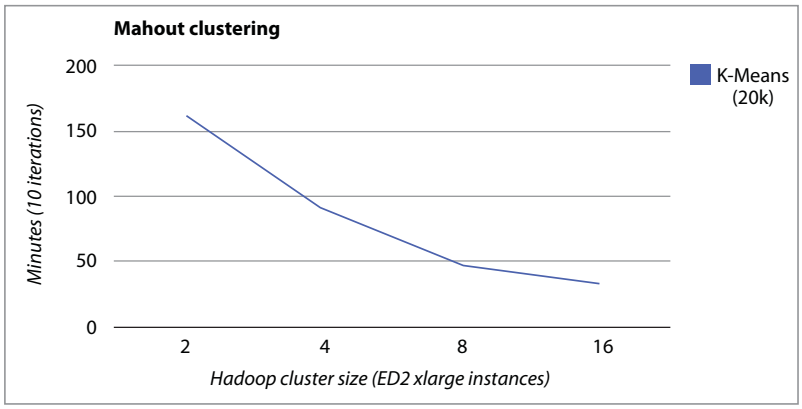

Figure 6.4 A graph of Mahout's K-Means performance on Amazon EC2 over 2 to 16 nodes

increases. This convex shape occurs because some nodes receive more demanding samples of data and others have to wait for them to finish. Hence, some resources are underutilized and the more nodes we have, the more likely this is to occur. Moreover, the differences among samples of documents are envisioned when two conditions are met. First, the vectors are represented with sparse structures. Second, the dataset possesses the long tail feature, which leads to an appearance of computationally demanding large vectors. Both conditions are fulfilled in our setting. We also attempted the same job on a 4-node cluster of EC2 large instances with two reducers per node. With this configuration, we expected the job to finish in about 120 minutes, but it took 137 minutes and the system load average was consistently over 3.

Benchmarking Mahout's other clustering algorithms

We also experimented with Mahout's other clustering algorithms, including Fuzzy K-Means, Canopy, and Dirichlet. Overall, we were unable to produce any conclusive results with the current dataset. For example, one iteration of the Fuzzy K-Means algorithm runs on average 10 times slower than one iteration of K-Means. But Fuzzy K-Means is believed to converge faster than K-Means, in which case you may require fewer iterations. A comparison of running times of Fuzzy K-Means and two variants of K-Means is depicted in figure 6.5.

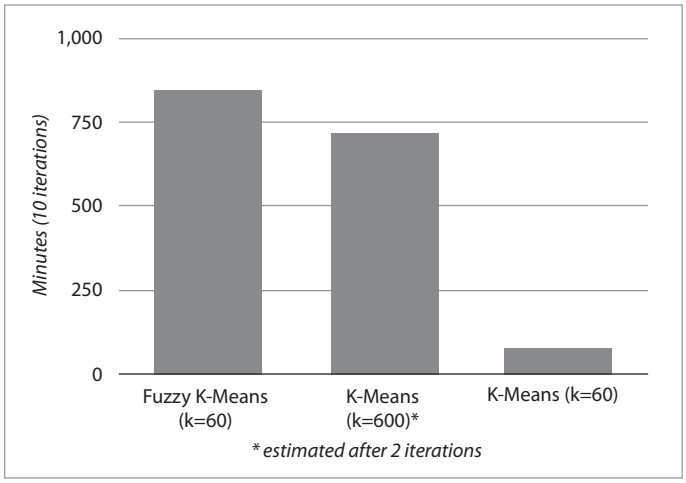

Figure 6.5 Comparison of running times with different clustering algorithms

We used four extra-large instances during experiments. It took over 14 hours (848 minutes) to complete all 10 iterations of Fuzzy K-Means with 60 clusters and the smoothing parameter m set to 3. It's approximately 10 times longer than with the K-Means algorithm, which took only 91 minutes. We've observed that the first iteration of both clustering algorithms is always much faster than the subsequent ones. Moreover, the second, third, and later iterations require comparable amounts of time. Therefore, we utilized the feedback from the second iteration to estimate an overall 10-iterations time consumption for various levels of k. When the number of clusters k is increased 10 times, we can expect proportional slowdown. Precisely, we've estimated the running time with k=600 to be 725 minutes. It's a bit below 10 x 91 because increasing k lets us better utilize an overhead of the fixed-cost processing. The difference between the first and the second iteration can be attributed to the fact that in the first iteration, random vectors are used as centroids. In the following iterations, centroids are much denser and require longer computations.

Mahout's Canopy algorithm is potentially useful as a preprocessing step to identify the number of clusters in a large dataset. With Canopy, a user has to define only two thresholds, which impact the distances between created clusters. We've found that Canopy outputs a nontrivial set of clusters when T1=0.15 and T2=0.9. But the time required to find these values doesn't seem to pay back in speeding up other algorithms. Keep in mind, also, that Mahout is still in a pre-1.0 release at the time of this writing, so speedups are likely to occur as more people use the code.

We also encountered problems with Dirichlet, but with some assistance from the Mahout community, we were able to complete a single iteration using alpha0 = 50 and modelDist = L1ModelDistribution. Using a larger value for alpha0 helps increase the probability of choosing a new cluster during the first iteration, which helps distribute the workload in subsequent iterations. Unfortunately, subsequent iterations still failed to complete in a timely fashion because too many data points were assigned to a small set of clusters in previous iterations.

BENCHMARKING SUMMARY AND NEXT STEPS

At the outset of this benchmarking process, we hoped to compare the performance of Mahout's various clustering algorithms on a large document set and to produce a recipe for large-scale clustering using Amazon EC2. We found that Mahout's K-Means implementation scales linearly for clustering millions of documents with roughly 20,000 features. For other types of clustering, we were unable to produce comparable results, and our only conclusion is that more experimentation is needed, as well as more tuning of the Mahout code. That said, we documented our EC2 and Elastic MapReduce setup notes in the Mahout wiki so that others can build upon our work.

6.8 Acknowledgments

The authors wish to acknowledge the valuable input of Ted Dunning, Jake Mannix, Stanisław Osiński, and Dawid Weiss in the writing of this chapter. The benchmarking of Mahout on Amazon Elastic MapReduce and EC2 was possible thanks to credits from the Amazon Web Services Apache Projects Testing Program.

6.9 Summary

Whether it's reducing the amount of news you have to wade through, quickly summarizing ambiguous search terms, or identifying topics in large collections, clustering can be an effective way to provide valuable discovery capabilities to your application. In this chapter, we discussed many of the concepts behind clustering, including some of the factors that go into choosing and evaluating a clustering approach. We then focused in on real-world examples by demonstrating how to use Carrot^2 and Apache Mahout to cluster search results, documents, and words into topics. We finished off the chapter by looking at techniques for improving performance, including using Mahout's singular value decomposition code.

6.10 References

Blei, David; Lafferty, John. 2009. "Visualizing Topics with Multi-Word Expressions." http://arxiv.org/abs/0907.1013v1.

Blei, David; Ng, Andrew; Jordan, Michael. 2003. "Latent Dirichlet allocation." *Journal of Machine Learning Research*, 3:993–1022, January.

Carpineto, Claudio; Osiński, Stanisław; Romano, Giovanni; Weiss, Dawid. 2009. "A Survey of Web Clustering Engines." *ACM Computing Surveys*.

Crabtree, Daniel; Gao, Xiaoying; Andreae, Peter. 2005. "Standardized Evaluation Method for Web Clustering Results." The 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI'05).

Cutting, Douglass; Karger, David; Pedersen, Jan; Tukey, John W. 1992. "Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections." Proceedings of the 15th Annual International ACM/SIGIR Conference.

Dash, Manoranjan; Choi, Kiseok; Scheuermann, Peter; Liu, Huan. 2002. "Feature Selection for Clustering - a filter solution." Second IEEE International Conference on Data Mining (ICDM'02).

Dash, Manoranjan, and Liu, Huan. 2000. "Feature Selection for Clustering." Proceedings of Fourth Pacific-Asia Conference on Knowledge Discovery and Data Mining.

Dean, Jeffrey; Ghemawat, Sanjay. 2004. "MapReduce: Simplified Data Processing on Large Clusters." OSDI'04: 6th Symposium on Operating Systems Design and Implementation. http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf.

Deerwester, Scott; Dumais, Susan; Landauer, Thomas; Furnas, George; Harshman, Richard. 1990. "Indexing by latent semantic analysis." *Journal of the American Society of Information Science*, 41(6):391–407.

Dunning, Ted. 1993. "Accurate methods for the statistics of surprise and coincidence." *Computational Linguistics*, 19(1).

Google News. 2011. http://www.google.com/support/news/bin/answer.py?answer=40235&topic=8851.

Liu, Tao; Liu, Shengping; Chen, Zheng; Ma, Wei-Ying. 2003. "An evaluation on feature selection for text clustering." Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003).

Manning, Christopher; Raghavan, Prabhakar; Schütze, Hinrich. 2008. *An Introduction to Information Retrieval*. Cambridge University Press.

Mannix, Jake. 2010, July. "SVD Memory Reqs." http://mail-archives.apache.org/mod_mbox/mahout-user/201007.mbox/%3CAANLkTik-uHrN2d838dHfY-wOhxHDQ3bhHkvCQvEIQCLT@mail.gmail.com%3E.

Mannix, Jake. 2010, August. "Understanding SVD CLI inputs." http://mail-archives.apache.org/mod_mbox/mahout-user/201008.mbox/%3CAANLkTi=ErpLuaWK7Z-2an786v5AsX3u5=adU2WJM5Ex7@mail.gmail.com%3E.

Steyvers, Mark, and Griffiths, Tom. 2007. "Probabilistic Topic Models." *Handbook of Latent Semantic Analysis*. http://citeseerx.ist.psu.edu/viewdoc/download?doi -0.=10.1.1.80.9625&rep=rep1&type=pdf.

Zamir, Oren, and Etzioni, Oren. 1998. "Web document clustering: a feasibility demonstration." Association of Computing Machinery Special Interest Group in Information Retrieval (SIGIR).