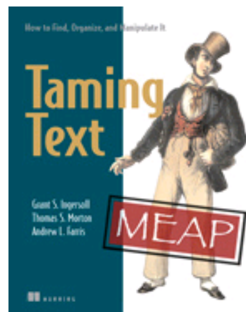


Taming Text
By Grant S. Ingersoll, Thomas S. Morton, and Andrew L. Farris

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. This article, based on chapter 6 of Taming Text, looks at how Apache Mahout can be used to cluster large collections of documents into buckets.

You may also be interested in…

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. To get started, for this article 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!).

We'll 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.

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 whichperforms best.

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

- *Programmatic*—Write 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 `Vector`s.
- *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 here, we'll forgo coverage of the ARFF format and focus on the first two means of producing Vector s for Mahout.

Programmatic vector creation

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

Listing 1 Vector creation using Mahout

```
double[] vals = new double[]{0.3, 1.8, 200.228};
Vector dense = new DenseVector(vals);        #A
assertTrue(dense.size() == 3);
Vector sparseSame = new SequentialAccessSparseVector(3);      #B
Vector sparse = new SequentialAccessSparseVector(3000);       #C
for (int i = 0; i < vals.length; i++) {      #D
  sparseSame.set(i, vals[i]);
  sparse.set(i, vals[i]);
}
assertFalse(dense.equals(sparse));   #E
assertEquals(dense, sparseSame);    #F
assertFalse(sparse.equals(sparseSame));
```

#A Create DenseVector with label of my-dense and 3 values. The cardinality of this vector is 3.
#B Create SparseVector with a label of mysparse-same that has cardinality of 3.
#C Create SparseVector with a label of my-sparse and a cardinality of 3000.
#D Set values to first 3 items in sparse vectors.
#E The dense and sparse Vectors aren't equal because they have different cardinality.
#F 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 `Vector`s to the proper format. This is demonstrated in the following listing.

Listing 2 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();    #A
FileSystem fs = FileSystem.get(conf);
SequenceFile.Writer seqWriter = SequenceFile.createWriter(fs, conf,
        path, LongWritable.class, VectorWritable.class);     #B
VectorWriter vecWriter = new SequenceFileVectorWriter(seqWriter);   #C
List<Vector> vectors = new ArrayList<Vector>();
vectors.add(sparse);
vectors.add(sparseSame);
vecWriter.write(vectors);   #D
vecWriter.close();
```

#A Create Configuration for Hadoop.
#B Create Hadoop SequenceFile. Writer to handle the job of physically writing out the vectors to a file in HDFS.
#C A VectorWriter processes the Vectors and invokes the underlying write methods on SequenceFile.Writer.
#D Do work of writing out files.

Mahout can also write out `Vector` s 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`s, 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 1 outlines the more commonly used ones.

Table 1 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. You might generate your `Vector`s by running the driver as in the next listing.

Listing 3 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 3, 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 `CosineDistance-Measure` included in Mahout. Now that we have some vectors, let's do some clustering using Mahout's K-Means implementation.

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 article, 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 4 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 three 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*`Vector`s. 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 4, 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 `Cluster-Dumper` 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 `KMeans-Driver` 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 the example output from this listing, 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 stop words (a, his, said, and so on).

Though simply dumping out the clusters is often useful, many applications need succinct labels that summarize the contents of the clusters. 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 thecontent 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 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.
- *Log-likelihood ratio (LLR)*—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.

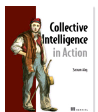
As in the case of the `ClusterDumper` top terms, closer inspection reveals some good terms (ignoring the fact that they're stemmed) and some terms of little use. It should be noted that most of the terms do a good job of painting a picture of what the overall set of documents in the cluster are about.

Summary

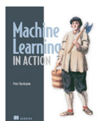
Whether it's reducing the amount of news you have to wade through, quickly summarizing ambiguous search terms, or identifying topics in large collections, clustering can be an effective way to provide valuable discovery capabilities to your application. In this article, we discussed how to use Apache Mahout to cluster search results, documents, and words into topics.

Here are some other Manning titles you might be interested in:

[Mahout in Action](#)
Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman



[Collective Intelligence in Action](#)
Satnam Alag



[Machine Learning in Action](#)
Peter Harrington

Last updated: November 1, 2012