Two years is a seeming eternity in the software world. In the past two years, we've seen the meteoric rise of social media, the commoditization of large-scale clustered computing (thanks to players like Amazon and RackSpace), and massive growth in data and our ability to make sense of it. And it's been two years since "Introducing Apache Mahout" was first published on developerWorks. Since then, the Mahout community — and the project's code base and capabilities — have grown significantly. Mahout has also seen significant uptake by companies large and small across the globe.

In my previous article on Mahout, I introduced many of the concepts of machine learning and the basics of using Mahout's suite of algorithms. The concepts I presented are still valid, but the algorithm suite has changed fairly significantly. Instead of going over the basics again, this article focuses on Mahout's current status and on how to scale Mahout across a compute cluster using Amazon's EC2 service and a data set comprising 7 million email documents. For a refresher on the basics, check out the Related topics, in particular the Mahout in Action book. Also, I'm going to assume a basic knowledge of Apache Hadoop and the Map-Reduce paradigm. (See Related topics for more information on Hadoop.)

Podcast: Grant Ingersoll on Apache Mahout machine learning

In this podcast, Apache Mahout committer and co-founder Grant Ingersoll introduces machine learning, the concepts involved, and explains how it applies to real-world applications.
Mahout's status

Mahout has come a long way in a short amount of time. Although the project's focus is still on what I like to call the "three Cs" — collaborative filtering (recommenders), clustering, and classification — the project has also added other capabilities. I'll highlight a few key expansions and improvements in two areas: core algorithms (implementations) for machine learning, and supporting infrastructure including input/output tools, integration points with other libraries, and more examples for reference. Do note, however, that this status is not complete. Furthermore, the limited space of this article means I can only offer a few sentences on each of the improvements. I encourage readers to find more information by reading the News section of the Mahout website and the release notes for each of Mahout's releases.

Algorithms, algorithms, algorithms

After trying to solve machine-learning problems for a while, one quickly realizes that no one algorithm is right for every situation. To that end, Mahout has added a number of new implementations. Table 1 contains my take on the most significant new algorithmic implementations in Mahout as well as some example use cases. I'll put some of these algorithms to work later in the article.

Table 1. New algorithms in Mahout

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Brief description</th>
<th>Use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression, solved by SGD</td>
<td>Blazing fast, simple, sequential classifier capable of online learning in demanding environments</td>
<td>Recommend ads to users, classify text into categories</td>
</tr>
<tr>
<td>Hidden Markov Models (HMM)</td>
<td>Sequential and parallel implementations of the classic classification algorithm designed to model real-world processes when the underlying generation process is unknown</td>
<td>Part-of-speech tagging of text; speech recognition</td>
</tr>
<tr>
<td>Singular Value Decomposition (SVD)</td>
<td>Designed to reduce noise in large matrices, thereby making them smaller and easier to work on</td>
<td>As a precursor to clustering, recommenders, and classification to do feature selection automatically</td>
</tr>
<tr>
<td>Dirichlet Clustering</td>
<td>Model-based approach to clustering that determines membership based on whether the data fits into the underlying model</td>
<td>Useful when the data has overlap or hierarchy</td>
</tr>
<tr>
<td>Spectral Clustering</td>
<td>Family of similar approaches that use a graph-based approach to determine cluster membership</td>
<td>Like all clustering algorithms, useful for exploring large, unseen data sets</td>
</tr>
<tr>
<td>Minhash Clustering</td>
<td>Uses a hashing strategy to group similar items</td>
<td>Same as other clustering approaches</td>
</tr>
</tbody>
</table>
Numerous recommender improvements

<table>
<thead>
<tr>
<th>Collocations</th>
<th>Map-Reduce enabled collocation implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed co-occurrence, SVD, Alternating Least-Squares</td>
<td>Finding statistically interesting phrases in text</td>
</tr>
</tbody>
</table>

Mahout has also added a number of low-level math algorithms (see the math package) that users may find useful. Many of these are used by the algorithms described in Table 1, but they are designed to be general-purpose and thus may fit your needs.

**Improving and expanding Mahout's infrastructure**

**Mahout's command line**

A while back, Mahout published a shell script that makes running Mahout programs (those that have a main()) easier by taking care of classpaths, environment variables, and other setup items. The script — named mahout — is in the $MAHOUT_HOME/bin directory.

As more people use an open source project and work to make the project's code work with their code, the more the infrastructure gets filled in. For Mahout, this evolution has led to a number of improvements. The most notable one is a much improved and consistent command-line interface, which makes it easier to submit and run tasks locally and on Apache Hadoop. This new script is located in the bin directory inside the Mahout top-level directory (which I'll refer to as $MAHOUT_HOME from here on). (See the Mahout's command line sidebar.)

Two key components of any machine-learning library are a reliable math library and an efficient collections package. The math library (located in the math module under $MAHOUT_HOME) provides a wide variety of functionality — ranging from data structures representing vectors, matrices, and related operators for manipulating them to tools for generating random numbers and useful statistics like the log likelihood (see Resources). Mahout's collections library consists of data structures similar to those provided by Java collections (Map, List, and so on) except that they natively support Java primitives such as int, float, and double instead of their Object counterparts of Integer, Float, and Double. This is important because every bit (pun intended) counts when you are dealing with data sets that can have millions of features. Furthermore, the cost of boxing between the primitives and their Object counterparts is prohibitive at large scale.

Mahout has also introduced a new Integration module containing code that's designed to complement or extend Mahout's core capabilities but is not required by everyone in all situations. For instance, the recommender (collaborative filtering) code now has support for storing its model in a database (via JDBC), MongoDB, or Apache Cassandra (see Related topics). The Integration module also contains a number of mechanisms for getting data into Mahout's formats as well as evaluating the results coming out. For example, it includes tools that can convert directories full of text files into Mahout's vector format (see the org.apache.mahout.text package in the Integration module).

Finally, Mahout has a number of new examples, ranging from calculating recommendations with the Netflix data set to clustering Last.fm music and many others. Additionally, the example
I developed for this article has also been added to Mahout's code base. I encourage you to take some time to explore the examples module (located in $MAHOUT_HOME/examples) in more detail.

Now that you're caught up on the state of Mahout, it's time to delve into the main event: how to scale out Mahout.

**Scaling Mahout in the cloud**

Getting Mahout to scale effectively isn't as straightforward as simply adding more nodes to a Hadoop cluster. Factors such as algorithm choice, number of nodes, feature selection, and sparseness of data — as well as the usual suspects of memory, bandwidth, and processor speed — all play a role in determining how effectively Mahout can scale. To motivate the discussion, I'll work through an example of running some of Mahout's algorithms on a publicly available data set of mail archives from the Apache Software Foundation (ASF) using Amazon's EC2 computing infrastructure and Hadoop, where appropriate (see Related topics).

Each of the subsections after the Setup takes a look at some of the key issues in scaling out Mahout and explores the syntax of running the example on EC2.

**Setup**

The setup for the examples involves two parts: a local setup and an EC2 (cloud) setup. To run the examples, you need:

1. **Apache Maven** 3.0.2 or higher.
2. **Git** version-control system (you may also wish to have a Github account).
3. A *NIX-based operating system such as Linux or Apple OS X. Cygwin may work for Windows®, but I haven't tested it.

To get set up locally, run the following on the command line:

```
1. mkdir -p scaling_mahout/data/sample
2. git clone git://github.com/lucidimagination/mahout.git mahout-trunk
3. cd mahout-trunk
4. mvn install (add a -DskipTests if you wish to skip Mahout's tests, which can take a while to run)
5. cd bin
6. /mahout (you should see a listing of items you can run, such as kmeans)
```

This should get all the code you need compiled and properly installed. Separately, download the sample data, save it in the scaling_mahout/data/sample directory, and unpack it (tar -xf scaling_mahout.tar.gz). For testing purposes, this is a small subset of the data you'll use on EC2.

To get set up on Amazon, you need an Amazon Web Services (AWS) account (noting your secret key, access key, and account ID) and a basic understanding of how Amazon's EC2 and Elastic Block Store (EBS) services work. Follow the documentation on the Amazon website to obtain the necessary access.
With the prerequisites out of the way, it's time to launch a cluster. It is probably best to start with a single node and then add nodes as necessary. And do note, of course, that running on EC2 costs money. Therefore, make sure you shut down your nodes when you are done running.

To bootstrap a cluster for use with the examples in the article, follow these steps:

1. **Download** Hadoop 0.20.203.0 from an ASF mirror and unpack it locally.
2. `cd hadoop-0.20.203.0/src/contrib/ec2/bin`
3. Open hadoop-ec2-env.sh in an editor and:
   a. Fill in your `AWS_ACCOUNT_ID`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `EC2_KEYDIR`, `KEY_NAME`, and `PRIVATE_KEY_PATH`. See the Mahout Wiki's "Use an Existing Hadoop AMI" page for more information (see **Related topics**).
   b. Set the `HADOOP_VERSION` to 0.20.203.0.
   c. Set `S3_BUCKET` to 499429964467.
   d. Set `ENABLE_WEB_PORTS=true`.
   e. Set `INSTANCE_TYPE` to `m1.xlarge` at a minimum.
4. Open hadoop-ec2-init-remote.sh in an editor and:
   a. In the section that creates hadoop-site.xml, add the following property:
   ```xml
   <property>
   <name>mapred.child.java.opts</name>
   <value>-Xmx8096m</value>
   </property>
   ```
   Note: If you want to run classification, you need to use a larger instance and more memory. I used double X-Large instances and 12GB of heap.
   b. Change `mapred.output.compress` to `false`.
5. Launch your cluster:
   ```bash
   ./hadoop-ec2 launch-cluster mahout-clustering X
   ```
   X is the number of nodes you wish to launch (for example, 2 or 10). I suggest starting with a small value and then adding nodes as your comfort level grows. This will help control your costs.
6. Create an EBS volume for the ASF Public Data Set (Snapshot: snap--17f7f476) and attach it to your master node instance (this is the instance in the mahout-clustering-master security group) on /dev/sdh. (See **Related topics** for links to detailed instructions in the EC2 online documentation.)
   a. If using the EC2 command line APIs (see **Related topics**), you can do:
      i. `ec2-create-volume --snapshot snap-17f7f476 --z ZONE`
      ii. `ec2-attach-volume $VOLUME_NUMBER -i $INSTANCE_ID -d /dev/sdh`, where $VOLUME_NUMBER is output by the create-volume step and the $INSTANCE_ID is the ID of the master node that was launched by the launch-cluster command
   b. Otherwise, you can do this via the AWS web console.
7. Upload the setup-asf-ec2.sh script (see **Download**) to the master instance:
   ```bash
   ./hadoop-ec2 push mahout-clustering $PATH/setup-asf-ec2.sh
   ```
8. Log in to your cluster:
   ```bash
   ./hadoop-ec2 login mahout-clustering
   ```
9. Execute the shell script to update your system, install Git and Mahout, and clean up some of the archives to make it easier to run:

```
./setup-asf-ec2.sh
```

With the setup details out of the way, the next step is to see what it means to put some of Mahout's more popular algorithms into production and scale them up. I'll focus primarily on the actual tasks of scaling up, but along the way I'll cover some questions about feature selection and why I made certain choices.

**Recommendations**

Collaborative filtering is one of Mahout's most popular and easy-to-use capabilities, so it's a logical starting place for a discussion of how to scale out Mahout. Recall that we are working with mail archives from the ASF. In the case of a recommendation task, one interesting possibility is to build a system that recommends potentially interesting mail threads to a user based on the threads that other users have read. To set this up as a collaborative-filtering problem, I'll define the item the system is recommending as the mail thread, as determined by the Message-ID and References in the mail header. The user will be defined by the From address in the mail message. In other words, I care about who has initiated or replied to a mail message. As for the value of the preference itself, I am simply going to treat the interaction with the mail thread as a Boolean preference: on if user X interacted with thread Y, and off if not. The one downstream effect of this choice is that we must use a similarity metric that works with Boolean preferences, such as the Tanimoto or log-likelihood similarities. This usually makes for faster calculations, and it likely reduces the amount of noise in the system, but your mileage may vary and you may wish to experiment with different weights.

**Threads, Message-IDs and the enemy of the good**

Note that my approach to handling message threads isn't perfect, because of the somewhat common practice of thread hijacking on mailing lists. Thread hijacking happens when someone starts a new message (that is, one with a new subject/topic) on the list by replying to an existing message, thereby passing along the original message reference. Instead of trying to work on this problem here, I've simply chosen to ignore it, but a real solution would need to address the problem head-on. Thus, I'm choosing "good enough" in lieu of perfection.

Because feature selection is straightforward when it comes to collaborative filtering (user, item, optional preference), we can fast-forward to look at the steps to take our content from raw mail archives to running locally and then to running in the cloud. Note that in many circumstances, the last step is often not necessary, because it is possible to get results fast enough on a single machine without adding the complexity of Hadoop to the equation. As a rough estimate, Mahout community benchmarks suggest one can reasonably provide recommendations of up to 100 million users on a single node. In the case of the email data, there aren't quite that many items (roughly 7 million messages), but I'm going to forge ahead and run it on Hadoop anyway.

To see the code in action, I've packaged up the necessary steps into a shell script located in the `$MAHOUT_HOME/examples/bin/build-asf-email.sh` file. Execute the shell script, passing in the location of your input data and where you would like the resulting output, as in:

```
./build-asf-email.sh ./scaling_mahout/data/sample/content ./scaling_mahout/output/
```
When prompted, choose recommender (option 1) and sit back and enjoy the verbosity of Mahout and Hadoop's logging output. When it is done, you'll see something resembling Listing 1:

**Listing 1. Sample output from running the recommender code**

```
11/09/08 09:57:37 INFO mapred.JobClient: Reduce output records=2072
11/09/08 09:57:37 INFO mapred.JobClient: Map output bytes=10210854
11/09/08 09:57:37 INFO mapred.JobClient: Combine input records=0
11/09/08 09:57:37 INFO mapred.JobClient: Map output records=24302
11/09/08 09:57:37 INFO mapred.JobClient: Reduce input records=24302
11/09/08 09:57:37 INFO driver.MahoutDriver: Program took 74847 ms
```

The results of this job will be all of the recommendations for all users in the input data. The results are stored in a subdirectory of the output directory named prefs/recommendations and contain one or more text files whose names start with part-r-. (This is how Hadoop outputs files.) Examining one of these files reveals, with one caveat, the recommendations formatted as:

```
user_id [item_id:score, item_id:score, ...]
```

For example, user ID 25 has recommendations for email IDs 26295 and 35548. The caveat is simply that `user_id` and `item_id` are not the original IDs, but mappings from the originals into integers. To help you understand why this is done, it's time to explain what actually happens when the shell script is executed.

Three steps are involved in producing the recommendation results:

1. Convert the raw mbox files into Hadoop's `SequenceFile` format using Mahout's `SequenceFilesFromMailArchives`.
2. Extract the message ID and From signature from the messages and output the results in a format Mahout can understand.
3. Run Mahout's `RecommenderJob` class.

I won't cover Step 1 beyond simply suggesting that interested readers refer to the code.

For Step 2, a bit more work was involved to extract the pertinent pieces of information from the files (message IDs, reply references, and the From addresses) and then store them as triples (From ID, Message-ID, preference) for the `RecommenderJob` to consume. The process for this is driven by the `MailToPrefsDriver`, which consists of three Map-Reduce jobs:

1. Create a dictionary mapping the string-based Message-ID to a unique `long`.
2. Create a dictionary mapping the string-based From email address to a unique `long`.
3. Extract the Message-ID, References, and From; map them to `longs` using the dictionaries from Steps 1 and 2; and output the triples to a text file.

After all that, it's time to generate some recommendations. To create recommendations, the `RecommenderJob` does the steps illustrated in Figure 1:
The main step doing the heavy lifting in the workflow is the "calculate co-occurrences" step. This step is responsible for doing pairwise comparisons across the entire matrix, looking for commonalities. As an aside, this step (powered by Mahout's RowSimilarityJob) is generally useful for doing pairwise computations between any rows in a matrix (not just ratings/reviews).

RecommenderJob is invoked in the shell script with the command:

```bash
bin/mahout recommenditembased --input $PREFS_REC_INPUT --output $RECS_OUT --tempDir $PREFS_TMP --similarityClassname SIMILARITY_LOGLIKELIHOOD
```

The first argument tells Mahout which command to run (RecommenderJob); many of the others (input/output/tempDir) are self-explanatory. The similarityClassname tells Mahout how to calculate the similarity between items when calculating co-occurrences. And I've chosen to use log likelihood for its simplicity, speed, and quality.

Once results are obtained, it's time to evaluate them. Mahout comes with an evaluation package (org.apache.mahout.cf.taste.eval) with useful tools that let you examine the results' quality. Unfortunately, they don't work with the Hadoop-based algorithms, but they can be useful in other cases. These tools hold out a percentage of the data as test data and then compare it against what the system produced, to judge the quality.
That really is all there is to generating recommendations — and the beautiful part of it is that this can then be run directly on the cluster. To do that, log into the EC2 cluster you set up earlier and run the same shell script (it's in /mnt/asf-email/mahout-trunk/examples/bin) as before. As you add nodes to your cluster, you should see a reduction in the overall time it takes to run the steps. As an example, running the full data set on a local machine took over three days to complete. Running on a 10-node cluster on EC2 took roughly 60 minutes for the main recommendation task plus the preparatory work of converting the email to a usable format.

The last piece, which I've left as an exercise for the reader, is to consume the recommendations as part of an application. Typically, once a significant number of items and users are in the system, recommendations are generated on a periodic basis — usually somewhere between hourly and daily, depending on business needs. After all, once a system reaches a certain amount of users and recommendations, changes to the recommendations produced will be much more subtle.

Next, let's take a look at classifying email messages, which in some cases can be thought of as a contextual recommendation system.

### Classification

Mahout has several classification algorithms, most of which (with one notable exception, stochastic gradient descent) are written to run on Hadoop. For this article's purposes, I'll use the naive bayes classifier, which many people start with and which often produces reasonable results while scaling effectively. For more details on the other classifiers, see the appropriate chapters in *Mahout in Action* or the Algorithms section of Mahout's wiki (see Related topics).

The email documents are broken down by Apache projects (Lucene, Mahout, Tomcat, and so on), and each project typically has two or more mailing lists (user, development, and so on). Given that the ASF email data set is partitioned by project, a logical classification problem is to try to predict the project a new incoming message should be delivered to. For example, does a new message belong to the Lucene mailing list or the Tomcat mailing list?

For Mahout's classification algorithms to work, a model must be trained to represent the patterns to be identified, and then tested against a subset of the data. In most classification problems, one or more persons must go through and manually annotate a subset of the data to be used in training. Thankfully, however, in this case the data set is already separated by project, so there is no need for hand annotation — although I'm counting on the fact that people generally pick the correct list when sending email, which you and I both know is not always the case.

Just as in the recommender case, the necessary steps are prepackaged into the build-asf-email.sh script and are executed when selecting option 3 (and then option 1 at the second prompt for standard naive bayes) from the menu. Similarly to recommendations, part of the work in scaling out the code is in the preparation of the data to be consumed. For classification of text, this primarily means encoding the features and then creating vectors out of the features, but it also includes setting up training and test sets. The complete set of steps taken are:

1. Convert the raw mbox files into Hadoop's SequenceFile format using Mahout's SequenceFilesFromMailArchives. (Note that the runtime options are slightly different here.)
bin/mahout org.apache.mahout.text.SequenceFilesFromMailArchives --charset "UTF-8" --body --subject --input $ASF_ARCHIVES --output $MAIL_OUT

2. Convert the SequenceFile entries into sparse vectors and modify the labels:
   a. bin/mahout seq2sparse --input $MAIL_OUT --output $SEQ2SP --norm 2 --weight TFIDF --namedVector --maxDFPercent 90 --minSupport 2 --analyzerName org.apache.mahout.text.MailArchivesClusteringAnalyzer
   b. bin/mahout org.apache.mahout.classifier.email.PrepEmailDriver --input $SEQ2SP --output $SEQ2SPLABEL --maxItemsPerLabel 1000

3. Split the input into training and test sets:
   bin/mahout split --input $SEQ2SPLABEL --trainingOutput $TRAIN --testOutput $TEST --randomSelectionPct 20 --overwrite --sequenceFiles

4. Run the naïve bayes classifier to train and test:
   a. bin/mahout trainnb -i $TRAIN -o $MODEL -extractLabels --labelIndex $LABEL
   b. bin/mahout testnb -i $TEST -m $MODEL --labelIndex $LABEL

The two main steps worth noting are Step 2 and Step 4. Step 2a is the primary feature-selection and encoding step, and a number of the input parameters control how the input text will be represented as weights in the vectors. Table 2 breaks down the feature-selection-related options of Step 2:

**Table 2. Feature-selection options for vector creation**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Examples and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>--norm</td>
<td>The norm modifies all vectors by a function that calculates its length (norm)</td>
<td>1 norm = Manhattan distance, 2 norm = Euclidean distance</td>
</tr>
<tr>
<td>--weight</td>
<td>Calculate the weight of any given feature as either TF-IDF (term frequency, inverse document frequency), or just Term Frequency</td>
<td>TF-IDF is a common weighting scheme in search and machine learning for representing text as vectors.</td>
</tr>
<tr>
<td>--maxDFPercent, --minSupport</td>
<td>Both of these options drop terms that are either too frequent (max) or not frequent enough across the collection of documents</td>
<td>Useful in automatically dropping common or very infrequent terms that add little value to the calculation</td>
</tr>
<tr>
<td>--analyzerName</td>
<td>An Apache Lucene analyzer class that can be used to tokenize, stem, remove, or otherwise change the words in the document</td>
<td>See Related topics to learn more about Lucene</td>
</tr>
</tbody>
</table>

The analysis process in Step 2a is worth diving into a bit more, given that it is doing much of the heavy lifting needed for feature selection. A Lucene Analyzer is made up of a Tokenizer class and zero or more TokenFilter classes. The Tokenizer is responsible for breaking up the original input into zero or more tokens (such as words). TokenFilter instances are chained together to then modify the tokens produced by the Tokenizer. For example, the Analyzer used in the example:

1. Tokenizes on whitespace, plus a few edge cases for punctuation.
2. Lowercases all tokens.
3. Converts non-ASCII characters to ASCII, where possible by converting diacritics and so on.
4. Throws away tokens with more than 40 characters.
5. Removes stop words (see the code for the list, which is too long to display here).
6. Stems the tokens using the Porter stemmer (see Related topics.)

The end result of this analysis is a significantly smaller vector for each document, as well as one that has removed common "noise" words (the, a, an, and the like) that will confuse the classifier. This Analyzer was developed iteratively by looking at examples in the email and then processing it through the Analyzer and examining the output, making judgment calls about how best to proceed. The process is as much intuition (experience) as it is science, unfortunately. The process and the result are far from perfect, but they are likely good enough.

Step 2b does some minor conversions of the data for processing as well as discards some content so that the various labels are evenly represented in the training data. This is an important point, because my first experiments with the data led to the all-too-common problem, in machine learning, of overfitting for those labels with significantly more training examples. In fact, when running on the cluster on the complete set of data, setting the --maxItemsPerLabel down to 1000 still isn't good enough to create great results, because some of the mailing lists have fewer than 1000 posts. This is possibly due to a bug in Mahout that the community is still investigating.

Step 4 is where the actual work is done both to build a model and then to test whether it is valid or not. In Step 4a, the --extractLabels option simply tells Mahout to figure out the training labels from the input. (The alternative is to pass them in.) The output from this step is a file that can be read via the org.apache.mahout.classifier.naivebayes.NaiveBayesModel class. Step 4b takes in the model as well as the test data and checks to see how good of a job the training did. The output is a confusion matrix as described in "Introducing Apache Mahout." For the sample data, the output is in Listing 2:

**Listing 2. Sample output from running the classifier code**

```
Correctly Classified Instances : 41523 61.9219%
Incorrectly Classified Instances : 25534 38.0781%
Total Classified Instances : 67057

Confusion Matrix
--------------------------------------
a b c d e f ><-Classified as
190440 12 1069 0 0 | 20125 a= cocoon_apache_org_dev
2066 0 1 477 0 0 | 2544 b= cocoon_apache_org_docs
165480 2370 704 0 0 | 19622 c= cocoon_apache_org_users
58 0 0 201090 0 | 20167 d= commons_apache_org_dev
147 0 1 4451 0 0 | 4599 e= commons_apache_org_user
```

You should notice that this is actually a fairly poor showing for a classifier (albeit better than guessing). The likely reason for this poor showing is that the user and development mailing lists for a given Apache project are so closely related in their vocabulary that it is simply too hard to distinguish. This is supported by the fact that 16,548 cocoon_user messages were incorrectly classified as cocoon_dev. In fact, rerunning the task using just the project name without distinguishing between user and dev lists in the sample data yields the results in Listing 3:
Listing 3. Sample output from rerunning the classifier code with project name only

Correctly Classified Instances : 38944 96.8949%
Incorrectly Classified Instances : 1248 3.1051%
Total Classified Instances : 40192

Confusion Matrix

a b c
---------------------------------------
a 18733 1241 0 | 19974 a = cocoon.apache.org
b 7 20211 0 | 20218 b = commons.apache.org

I think you will agree that 96 percent accuracy is a tad better than 61 percent! In fact, it is likely too good to be true. The score is likely due to the nature of this particular small data set or perhaps a deeper issue that needs investigating. In fact, a score like this should warrant one to investigate further by adding data and reviewing the code to generate it. For now, I'm happy to live with it as an example of what the results would look like. However, we could try other techniques or better feature selection, or perhaps more training examples, in order to raise the accuracy. It is also common to do cross-fold validation of the results. Cross-fold validation involves repeatedly taking parts of the data out of the training sample and either putting it into the test sample or setting it aside. The system is then judged on the quality of all the runs, not just one.

Taking this to the cloud is just as straightforward as it is with the recommenders. The entire script should run in your cluster simply by passing in the appropriate paths. Running this on EC2 on a 10-node cluster took mere minutes for the training and test, alongside the usual preparatory work. Unfortunately, however, when you run this the quality of running against the full data set in the cloud has suffers because some mailing lists have very few data points. These should likely be removed from consideration.

The next steps to production involve making the model available as part of your runtime system as well as setting up a workflow for making sure the model is updated as feedback is obtained from the system. From here, I'll take a look at clustering.

Clustering

As with classification, Mahout has numerous clustering algorithms, each with different characteristics. For instance, K-Means scales nicely but requires you to specify the number of clusters you want up front, whereas Dirchlet clustering requires you to pick a model distribution as well as the number of clusters you want. Clustering also has a fair amount in common with classification, and it is possible, in places, for them to work together by using clusters as part of classification. Moreover, much of the data-preparation work for classification is the same for clustering — such as converting the raw content into sequence files and then into sparse vectors — so you can refer to the Classification section for that information.

For clustering, the primary question to be answered is: can we logically group all of the messages based on content similarity, regardless of project? For instance, perhaps messages on the Apache Solr mailing list about using Apache Tomcat as a web container will be closer to messages for the Tomcat project than to the originating project.
For this example, the first steps are much like classification, diverging after the completion of the conversion to sparse vectors. The specific steps are:

1. The same steps as Steps 1 and 2 from classification.
2. $ bin/mahout kmeans --input "$SEQ2SP/tfidf-vectors" --output $CLUST_OUT -k 50 --maxIter 20 --distanceMeasure org.apache.mahout.common.distance.CosineDistanceMeasure --clustering --method mapreduce --clusters "$CLUST_OUT/clusters"

In this case, K-Means is run to do the clustering, but the shell script supports running Dirichlet clustering as well. (When executing the script, you’re prompted to choose the algorithm you wish to run.) In the previous example, the parameters worth delving into are:

1. -k: The number of clusters to create. I picked 50 out of a hat, but it certainly could be other values.
2. --maxIter: K-Means is an iterative algorithm whereby the center of the cluster is updated as part of each iteration. In some cases, it is not guaranteed to complete on its own, so this parameter can make sure the algorithm completes.
3. --distanceMeasure: The distance measure determines the similarity between the current centroid and the point being examined. In this case, I chose the Cosine Distance Measure, which is often a good choice for text data. Mahout has many other implementations, and you may find it worthwhile to try them out with your data.
4. --clustering: Tell Mahout to output which points belong to which centroid. By default, Mahout just computes the centroids, because this is often all that is needed.

Once the run is done, you can dump out the cluster centroids (and the associated points) by using Mahout’s ClusterDump program. The final results will be in the subdirectory under the kmeans directory starting with the name clusters- and ending with -final. The exact value will depend on how many iterations it took to run the task; for instance, clusters-2-final is the output from the third iteration. As an example, this command dumps out the clusters from running the small sample of data:

```
bin/mahout clusterdump --seqFileDir ../output/ibm/clustering/kmeans/clusters-2-final --pointsDir ../output/ibm/clustering/kmeans/clusteredPoints/
```

The --seqFileDir points at the centroids created, and the --pointsDir is the directory of clustered points. A small sampling of the results is in Listing 4:
Listing 4. Sample output from running the ClusterDumper

:VL-337881[n=4420 c=[
  Top Terms:
  user                                      => 0.060885823267350335
  mailscan                                  => 0.05059369006868677
  cocoon                                    => 0.048781178576134204
  virus                                     => 0.04285897589148712
  derek                                     => 0.04084340722527813
  legal                                     => 0.040052624979813184
  scan                                      => 0.03861016730680097
  danger                                    => 0.03848600584647758
  csir                                      => 0.03712359352614157
  transtec                                  => 0.03388019099942435

  Weight: [props - optional]: Point:
  1.0 : [distance=0.888270593967813]:
  /cocoon.apache.org/dev/200004.gz/NBBBIDJGOCOKIOHGDPBKEPPEEA.AXXXX =
  [ag:0.165, briefli:0.250, chang:0.075, close:0.137, cocoon:0.060, cocoon1:0.226, cocoon2:0.218, concept:0.277, develop:0.101, differ:0.144, explain:0.154, greet:0.197, klingenderstrass:0.223, langham:0.223, look:0.105, mailserv:0.293, matthew:0.277, mlangham:0.240, paderborn:0.215, processor:0.231, produc:0.202, put:0.170, scan:0.180, tel:0.163, understand:0.127, virus:0.194]

In Listing 4, notice that the output includes a list of terms the algorithm has determined are most representative of the cluster. This can be useful for generating labels for use in production, as well as for tuning feature selection in the preparatory steps — because it's often the case that stop words show up (in this case, for example, user likely is one) in the list in the first few experiments with running the data.

As you've likely come to expect, running this on your cluster is as simple as running it locally — and as simple as the other two examples. Besides the time spent converting the content (approximately 150 minutes), the actual clustering job took about 40 minutes on 10 nodes in my tests.

Unfortunately, with clustering, evaluating the results often comes down to the "smell test," although Mahout does have some tools for evaluation (see CDbwEvaluator and the ClusterDumper options for outputting top terms). For the smell test, visualizing the clusters is often the most beneficial, but unfortunately many graph-visualization toolkits choke on large datasets, so you may be left to your own devices to visualize.

As with recommendations and classification, the steps to production involve deciding on the workflow for getting data in as well as how often to do the processing and, of course, making use of it in your business environment. You will also likely need to work through the various algorithms to see which ones work best for your data.

What's coming next in Mahout?

Apache Mahout continues to move forward in a number of ways. The community's primary focus at the moment is on pushing toward a 1.0 release by doing performance testing, documentation, API improvement, and the addition of new algorithms. The next release, 0.6, is likely to happen towards the end of 2011, or soon thereafter. At a deeper level, the community is also starting to look at distributed, in-memory approaches to solving machine-learning problems. In many cases, machine-learning problems are too big for a single machine, but Hadoop induces too much
overhead that's due to disk I/O. Regardless of the approach, Mahout is well positioned to help solve today's most pressing big-data problems by focusing in on scalability and making it easier to consume complicated machine-learning algorithms.

Acknowledgments

Special thank you to Timothy Potter for assistance in AMI packaging and fellow Mahout committers Sebastian Schelter, Jake Mannix, and Sean Owen for technical review. Part of this work was supported by the Amazon Apache Testing Program.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shell script</td>
<td>j-mahout-scaling.zip</td>
<td>2KB</td>
</tr>
</tbody>
</table>
Related topics

- "Introducing Apache Mahout" (Grant Ingersoll, developerWorks, September 2009): Mahout co-founder Ingersoll introduces the basic concepts of machine learning and demonstrates how to use Mahout to cluster documents, make recommendations, and organize content.
- **Apache Mahout**: Keep an eye on Apache Mahout's home page for all the latest news as well as documentation and new downloads.
- **Powered By Mahout**: Mahout's powered by page lists companies willing to declare their usage of Mahout's algorithms.
- **Taming Text** (Grant S. Ingersoll, Thomas S. Morton, and Andrew L. Farris, Manning Publications, 2011): This book covers Mahout and related open source technologies for building text-based applications.
- **Mahout in Action** (Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman, Manning Publications, 2011): This book is the definitive guide to understanding Mahout.
- **Apache Hadoop**: Learn about Hadoop.
- **Algorithms**: Read more about the algorithms that Mahout implements, as well as areas of interest to the community on the Mahout Wiki.
- **Surprise and Coincidence**: Read more about the usefulness of the log-likelihood measure at Mahout committer Ted Dunning's blog.
- **MongoDB**: Discover more about MongoDB on its website.
- **Apache Cassandra**: Get more information on Apache Cassandra by browsing to its website.
- **Amazon Web Services including EC2**: Learn more about Amazon’s EC2 infrastructure.
- **Use an Existing Hadoop AMI**: This page in the Mahout wiki details the process for launching Hadoop clusters in EC2.
- **Amazon EC2 User Guide**: Consult the EC2 User Guide for this article’s Setup steps:
  - **Using Public Data Sets**: How to use Amazon EC2 public data sets.
  - **Apache Software Foundation's public mail archives**: Use a snapshot of the Apache Software Foundation's public mail archives for testing Mahout.
  - **Making an Amazon EBS Volume Available for Use**: How to make an Amazon EBS volume available to your operating system.
  - **Getting Started with the Command Line Tools**: How to set up your Amazon EC2 environment for use with the Amazon EC2 command line tools (also called the API tools or the CLI tools).
- **Message-ID**: Learn more about mail headers and the Message-ID.
- **Apache Lucene**: Find out more about Apache Lucene via its website.
- **Dr. Martin Porter's Snowball stemmers**: These are often useful for working with text in search and machine-learning projects.
- Download **IBM product evaluation versions** and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

© Copyright IBM Corporation 2011
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)