Introducing Apache Mahout

Scalable, commercial-friendly machine learning for building intelligent applications

Skill Level: Intermediate

Grant Ingersoll (grant@lucidimagination.com)
Member, Technical Staff
Lucid Imagination

08 Sep 2009

Once the exclusive domain of academics and corporations with large research budgets, intelligent applications that learn from data and user input are becoming more common. The need for machine-learning techniques like clustering, collaborative filtering, and categorization has never been greater, be it for finding commonalities among large groups of people or automatically tagging large volumes of Web content. The Apache Mahout project aims to make building intelligent applications easier and faster. Mahout co-founder Grant Ingersoll introduces the basic concepts of machine learning and then demonstrates how to use Mahout to cluster documents, make recommendations, and organize content.

Increasingly, the success of companies and individuals in the information age depends on how quickly and efficiently they turn vast amounts of data into actionable information. Whether it's for processing hundreds or thousands of personal e-mail messages a day or divining user intent from petabytes of weblogs, the need for tools that can organize and enhance data has never been greater. Therein lies the premise and the promise of the field of *machine learning* and the project this article introduces: Apache Mahout (see Resources).

Machine learning is a subfield of artificial intelligence concerned with techniques that allow computers to improve their outputs based on previous experiences. The field is closely related to data mining and often uses techniques from statistics, probability theory, pattern recognition, and a host of other areas. Although machine learning is not a new field, it is definitely growing. Many large companies, including IBM®,

Google, Amazon, Yahoo!, and Facebook, have implemented machine-learning algorithms in their applications. Many, many more companies would benefit from leveraging machine learning in their applications to learn from users and past situations.

After giving a brief overview of machine-learning concepts, I'll introduce you to the Apache Mahout project's features, history, and goals. Then I'll show you how to use Mahout to do some interesting machine-learning tasks using the freely available Wikipedia data set.

Machine learning 101

Machine learning uses run the gamut from game playing to fraud detection to stock-market analysis. It's used to build systems like those at Netflix and Amazon that recommend products to users based on past purchases, or systems that find all of the similar news articles on a given day. It can also be used to categorize Web pages automatically according to genre (sports, economy, war, and so on) or to mark e-mail messages as spam. The uses of machine learning are more numerous than I can cover in this article. If you're interested in exploring the field in more depth, I encourage you to refer to the Resources.

Several approaches to machine learning are used to solve problems. I'll focus on the two most commonly used ones — *supervised* and *unsupervised* learning — because they are the main ones supported by Mahout.

Supervised learning is tasked with learning a function from labeled training data in order to predict the value of any valid input. Common examples of supervised learning include classifying e-mail messages as spam, labeling Web pages according to their genre, and recognizing handwriting. Many algorithms are used to create supervised learners, the most common being neural networks, Support Vector Machines (SVMs), and Naive Bayes classifiers.

Unsupervised learning, as you might guess, is tasked with making sense of data without any examples of what is correct or incorrect. It is most commonly used for clustering similar input into logical groups. It also can be used to reduce the number of dimensions in a data set in order to focus on only the most useful attributes, or to detect trends. Common approaches to unsupervised learning include k-Means, hierarchical clustering, and self-organizing maps.

For this article, I'll focus on three specific machine-learning tasks that Mahout currently implements. They also happen to be three areas that are quite commonly used in real applications:

- Collaborative filtering

- Clustering
- Categorization

I'll take a deeper look at each of these tasks at the conceptual level before exploring their implementations in Mahout.

Collaborative filtering

Collaborative filtering (CF) is a technique, popularized by Amazon and others, that uses user information such as ratings, clicks, and purchases to provide recommendations to other site users. CF is often used to recommend consumer items such as books, music, and movies, but it is also used in other applications where multiple actors need to collaborate to narrow down data. Chances are you've seen CF in action on Amazon, as shown in Figure 1:

Figure 1. Example of collaborative filter on Amazon

Given a set of users and items, CF applications provide recommendations to the current user of the system. Four ways of generating recommendations are typical:

- **User-based**: Recommend items by finding similar users. This is often harder to scale because of the dynamic nature of users.
- **Item-based**: Calculate similarity between items and make recommendations. Items usually don't change much, so this often can be computed offline.
- **Slope-One**: A very fast and simple item-based recommendation approach applicable when users have given ratings (and not just boolean preferences).

- **Model-based**: Provide recommendations based on developing a model of users and their ratings.

All CF approaches end up calculating a notion of similarity between users and their rated items. There are many ways to compute similarity, and most CF systems allow you to plug in different measures so that you can determine which one works best for your data.

Clustering

Clustering helps in identifying abnormal behaviour or noise.

Given large data sets, whether they are text or numeric, it is often useful to group together, or *cluster*, similar items automatically. For instance, given all of the news for the day from all of the newspapers in the United States, you might want to group all of the articles about the same story together automatically; you can then choose to focus on specific clusters and stories without needing to wade through a lot of unrelated ones. Another example: Given the output from sensors on a machine over time, you could cluster the outputs to determine normal versus problematic operation, because normal operations would all cluster together and abnormal operations would be in outlying clusters.

Like CF, clustering calculates the similarity between items in the collection, but its only job is to group together similar items. In many implementations of clustering, items in the collection are represented as vectors in an *n*-dimensional space. Given the vectors, one can calculate the distance between two items using measures such as the Manhattan Distance, Euclidean distance, or cosine similarity. Then, the actual clusters can be calculated by grouping together the items that are close in distance.

There are many approaches to calculating the clusters, each with its own trade-offs. Some approaches work from the bottom up, building up larger clusters from smaller ones, whereas others break a single large cluster into smaller and smaller clusters. Both have criteria for exiting the process at some point before they break down into a trivial cluster representation (all items in one cluster or all items in their own cluster). Popular approaches include k-Means and hierarchical clustering. As I'll show later, Mahout comes with several different clustering approaches.

Categorization

The goal of *categorization* (often also called *classification*) is to label unseen documents, thus grouping them together. Many classification approaches in machine learning calculate a variety of statistics that associate the features of a document with the specified label, thus creating a model that can be used later to classify unseen documents. For example, a simple approach to classification might keep track of the words associated with a label, as well as the number of times those words are seen for a given label. Then, when a new document is classified, the words in the document are looked up in the model, probabilities are calculated, and the best result is output, usually along with a score indicating the confidence the

result is correct.

Features for classification might include words, weights for those words (based on frequency, for instance), parts of speech, and so on. Of course, features really can be anything that helps associate a document with a label and can be incorporated into the algorithm.

The field of machine learning is large and robust. Instead of focusing further on the theoretical, which is impossible to do proper justice to here, I'll move on and dive into Mahout and its usage.

Introducing Mahout

Apache Mahout is a new open source project by the Apache Software Foundation (ASF) with the primary goal of creating scalable machine-learning algorithms that are free to use under the Apache license. The project is entering its second year, with one public release under its belt. Mahout contains implementations for clustering, categorization, CF, and evolutionary programming. Furthermore, where prudent, it uses the Apache Hadoop library to enable Mahout to scale effectively in the cloud (see Resources).

Mahout history

What's in a name?
A *mahout* is a person who keeps and drives an elephant. The name Mahout comes from the project's (sometime) use of Apache Hadoop — which has a yellow elephant as its logo — for scalability and fault tolerance.

The Mahout project was started by several people involved in the Apache Lucene (open source search) community with an active interest in machine learning and a desire for robust, well-documented, scalable implementations of common machine-learning algorithms for clustering and categorization. The community was initially driven by Ng et al.'s paper "Map-Reduce for Machine Learning on Multicore" (see Resources) but has since evolved to cover much broader machine-learning approaches. Mahout also aims to:

- Build and support a community of users and contributors such that the code outlives any particular contributor's involvement or any particular company or university's funding.
- Focus on real-world, practical use cases as opposed to bleeding-edge research or unproven techniques.
- Provide quality documentation and examples.

Features

Although relatively young in open source terms, Mahout already has a large amount of functionality, especially in relation to clustering and CF. Mahout's primary features are:

> **A few words on Map-Reduce**
> Map-Reduce is a distributed programming API pioneered by Google and implemented in the Apache Hadoop project. Combined with a distributed file system, it often makes parallelizing problems easier by giving programmers a well-defined API for describing parallel computation tasks. (See Resources for more information.)

- Taste CF. Taste is an open source project for CF started by Sean Owen on SourceForge and donated to Mahout in 2008.
- Several Map-Reduce enabled clustering implementations, including k-Means, fuzzy k-Means, Canopy, Dirichlet, and Mean-Shift.
- Distributed Naive Bayes and Complementary Naive Bayes classification implementations.
- Distributed fitness function capabilities for evolutionary programming.
- Matrix and vector libraries.
- Examples of all of the above algorithms.

Getting started with Mahout

Getting up and running with Mahout is relatively straightforward. To start, you need to install the following prerequisites:

- JDK 1.6 or higher
- Ant 1.7 or higher
- If you want to build the Mahout source, Maven 2.0.9 or 2.0.10

You also need this article's sample code (see Download), which includes a copy of Mahout and its dependencies. Follow these steps to install the sample code:

1. `unzip sample.zip`
2. `cd apache-mahout-examples`
3. `ant install`

Step 3 downloads the necessary Wikipedia files and compiles the code. The Wikipedia file used is approximately 2.5 gigabytes, so download times will depend on your bandwidth.

Building a recommendation engine

Mahout currently provides tools for building a recommendation engine through the Taste library — a fast and flexible engine for CF. Taste supports both user-based and item-based recommendations and comes with many choices for making recommendations, as well as interfaces for you to define your own. Taste consists of five primary components that work with `Users`, `Items` and `Preferences`:

- `DataModel`: Storage for `Users`, `Items`, and `Preferences`
- `UserSimilarity`: Interface defining the similarity between two users
- `ItemSimilarity`: Interface defining the similarity between two items
- `Recommender`: Interface for providing recommendations
- `UserNeighborhood`: Interface for computing a neighborhood of similar users that can then be used by the `Recommenders`

These components and their implementations make it possible to build out complex recommendation systems for either real-time-based recommendations or offline recommendations. Real-time-based recommendations often can handle only a few thousand users, whereas offline recommendations can scale much higher. Taste even comes with tools for leveraging Hadoop to calculate recommendations offline. In many cases, this is a reasonable approach that allows you to meet the demands of a large system with a lot of users, items, and preferences.

To demonstrate building a simple recommendation system, I need some users, items, and ratings. For this purpose, I randomly generated a large set of `Users` and `Preferences` for the Wikipedia documents (`Items` in Taste-speak) using the code in `cf.wikipedia.GenerateRatings` (included in the source with the sample code) and then supplemented this with a set of hand-crafted ratings around a specific topic (Abraham Lincoln) to create the final recommendations.txt file included in the sample. The idea behind this approach is to show how CF can guide fans of a specific topic to other documents of interest within the topic. In the example data are 990 (labeled 0 to 989) random users who have randomly assigned ratings to all the articles in the collection, and 10 users (labeled 990 to 999) who have rated one or more of the 17 articles in the collection containing the phrase *Abraham Lincoln*.

Beware made-up data!
The example presented here contains purely made-up data. I did all of the ratings myself, simulating 10 actual users who like information about Abraham Lincoln. While I believe the concept behind the data

is interesting, the data itself and the values used are not. If you want real data, I suggest checking out the GroupLens project at the University of Minnesota and the Taste documentation (see Resources). I chose to make up the data because I wanted to use a single data set across all of the examples.

To start, I'll demonstrate how to create recommendations for a user given the set of ratings in recommendations.txt. As is the case with most uses of Taste, the first step is to load the data containing the recommendations and store it in a `DataModel`. Taste comes with several different implementations of `DataModel` for working with files and databases. For this example, I'll keep things simple and use the `FileDataModel` class, which expects each line to be of the form: user ID, item ID, preference — where both the user ID and the item ID are strings, while the preference can be a double. Given a model, I then need to tell Taste how it should compare users by declaring a `UserSimilarity` implementation. Depending on the `UserSimilarity` implementation used, you might also need to tell Taste how to infer preferences in the absence of an explicit setting for the user. Listing 1 puts all of these words into code. (`cf.wikipedia.WikipediaTasteUserDemo` in the sample code contains the full listing.)

Listing 1. Creating the model and defining user similarity

```
//create the data model
FileDataModel dataModel = new FileDataModel(new File(recsFile));
UserSimilarity userSimilarity = new PearsonCorrelationSimilarity(dataModel);
// Optional:
userSimilarity.setPreferenceInferrer(new AveragingPreferenceInferrer(dataModel));
```

In Listing 1, I use the `PearsonCorrelationSimilarity`, which measures the correlation between two variables, but other `UserSimilarity` measures are available. Choice of a similarity measure depends on the type of data present and your testing. For this data, I found this combination to work best while still demonstrating the issues. You'll find more information on choosing a similarity measure at the Mahout Web site (see Resources).

To complete the example, I construct a `UserNeighborhood` and a `Recommender`. The `UserNeighborhood` identifies users similar to my user and is handed off to the `Recommender`, which then does the work of creating a ranked list of recommended items. Listing 2 captures these ideas in code:

Listing 2. Generating recommendations

```
//Get a neighborhood of users
UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(neighborhoodSize, userSimilarity, dataModel);
//Create the recommender
Recommender recommender =
        new GenericUserBasedRecommender(dataModel, neighborhood, userSimilarity);
User user = dataModel.getUser(userId);
System.out.println("-----");
```

```
System.out.println("User: " + user);
//Print out the users own preferences first
TasteUtils.printPreferences(user, handler.map);
//Get the top 5 recommendations
List<RecommendedItem> recommendations =
        recommender.recommend(userId, 5);
TasteUtils.printRecs(recommendations, handler.map);
```

You can run the full example on the command line by executing `ant user-demo` in the directory containing the sample. Running this command prints the preferences and recommendations for the mythical user 995, who just happens to be a fan of Lincoln. Listing 3 shows the output from running `ant user-demo`:

Listing 3. Output from user recommendation

```
 [echo] Getting similar items for user: 995 with a neighborhood of 5
     [java] 09/08/20 08:13:51 INFO file.FileDataModel: Creating FileDataModel
            for file src/main/resources/recommendations.txt
     [java] 09/08/20 08:13:51 INFO file.FileDataModel: Reading file info...
     [java] 09/08/20 08:13:51 INFO file.FileDataModel: Processed 100000 lines
     [java] 09/08/20 08:13:51 INFO file.FileDataModel: Read lines: 111901
     [java] Data Model: Users: 1000 Items: 2284
     [java] -----
     [java] User: 995
     [java] Title: August 21 Rating: 3.930000066757202
     [java] Title: April Rating: 2.203000068664551
     [java] Title: April 11 Rating: 4.230000019073486
     [java] Title: Battle of Gettysburg Rating: 5.0
     [java] Title: Abraham Lincoln Rating: 4.739999771118164
     [java] Title: History of The Church of Jesus Christ of Latter-day Saints
              Rating: 3.430000066757202
     [java] Title: Boston Corbett Rating: 2.009999990463257
     [java] Title: Atlanta, Georgia Rating: 4.429999828338623
     [java] Recommendations:
     [java] Doc Id: 50575 Title: April 10 Score: 4.98
     [java] Doc Id: 134101348 Title: April 26 Score: 4.860541
     [java] Doc Id: 133445748 Title: Folklore of the United States Score: 4.4308662
     [java] Doc Id: 1193764 Title: Brigham Young Score: 4.404066
     [java] Doc Id: 2417937 Title: Andrew Johnson Score: 4.24178
```

From the results in Listing 3, you can see that the system recommended several articles with various levels of confidence. In fact, each of these items was rated by other Lincoln fans, but not by user 995. If you want to see the results for other users, just pass in the `-Duser.id=`*`USER-ID`* parameter on the command line, where *`USER-ID`* is a number between `0` and `999`. You can also change the size of the neighborhood by passing in `-Dneighbor.size=`*`X`*, where *`X`* is an integer greater than 0. In fact, changing the neighborhood size to `10` yields very different results, which are influenced by the fact that one of the random users is in the neighborhood. To see the neighborhood of users and the items in common, add `-Dcommon=true` to the command line.

Now, if you happened to enter a number not in the range of users, you might have noticed that the example spits out a `NoSuchUserException`. Indeed, your application would need to handle what to do when a new user enters the system. For instance, you might just show the 10 most popular articles, a random selection

of articles, or a selection of "dissimilar" articles — or, for that matter, nothing at all.

As I mentioned earlier, the user-based approach often does not scale. In this case, it is better to use an item-item based approach. Thankfully, Taste makes using an item-item approach just as straightforward. The basic code to get up and running with item-item similarity isn't much different, as you can see in Listing 4:

Listing 4. Example of item-item similarity (from cf.wikipedia.WikipediaTasteItemItemDemo)

```
//create the data model
FileDataModel dataModel = new FileDataModel(new File(recsFile));
//Create an ItemSimilarity
ItemSimilarity itemSimilarity = new LogLikelihoodSimilarity(dataModel);
//Create an Item Based Recommender
ItemBasedRecommender recommender =
        new GenericItemBasedRecommender(dataModel, itemSimilarity);
//Get the recommendations
List<RecommendedItem> recommendations =
        recommender.recommend(userId, 5);
TasteUtils.printRecs(recommendations, handler.map);
```

Just as in Listing 1, I create a `DataModel` from the recommendations file, but this time, instead of instantiating a `UserSimilarity` instance, I create an `ItemSimilarity` using the `LogLikelihoodSimilarity`, which helps handle rare events. After that, I feed the `ItemSimilarity` to an `ItemBasedRecommender` and then ask for the recommendations. That's it! You can run this in the sample code via the `ant item-demo` command. From here, of course, you'd want to set your system up to do these calculations offline, and you can also explore other `ItemSimilarity` measures. Note that, because of the randomness of the data in this example, the recommendations may not be as expected. In fact, it is important to make sure you evaluate your results during testing and try different similarity metrics, as many of the common metrics have certain edge cases that break down when insufficient data is available to give proper recommendations.

Revisiting the new-user example, the problem of what to do in the absence of user preferences becomes a lot easier to address once the user navigates to an item. In that case, you can take advantage of the item-item calculations and ask the `ItemBasedRecommender` for the items that are most similar to the current item. Listing 5 demonstrates this in code:

Listing 5. Similar items demo (from cf.wikipedia.WikipediaTasteItemRecDemo)

```
//create the data model
FileDataModel dataModel = new FileDataModel(new File(recsFile));
//Create an ItemSimilarity
ItemSimilarity itemSimilarity = new LogLikelihoodSimilarity(dataModel);
//Create an Item Based Recommender
ItemBasedRecommender recommender =
        new GenericItemBasedRecommender(dataModel, itemSimilarity);
//Get the recommendations for the Item
```

```
List<RecommendedItem> simItems
        = recommender.mostSimilarItems(itemId, numRecs);
TasteUtils.printRecs(simItems, handler.map);
```

You can run Listing 5 from the command line by executing `ant sim-item-demo`. The only real difference from Listing 4 is that Listing 5, instead of asking for recommendations, asks for the most similar items to the input item.

From here, you should have enough to dig in with Taste. To learn more, refer to the Taste documentation and the mahout-user@lucene.apache.org mailing list (see Resources). Next up, I'll take a look at how to find similar articles by leveraging some of Mahout's clustering capabilities.

Clustering with Mahout

Mahout supports several clustering-algorithm implementations, all written in Map-Reduce, each with its own set of goals and criteria:

- **Canopy**: A fast clustering algorithm often used to create initial seeds for other clustering algorithms.
- **k-Means** (and **fuzzy k-Means**): Clusters items into k clusters based on the distance the items are from the centroid, or center, of the previous iteration.
- **Mean-Shift**: Algorithm that does not require any *a priori* knowledge about the number of clusters and can produce arbitrarily shaped clusters.
- **Dirichlet**: Clusters based on the mixing of many probabilistic models giving it the advantage that it doesn't need to commit to a particular view of the clusters prematurely.

From a practical standpoint, the names and implementations aren't as important as the results they produce. With that in mind, I'll show how k-Means works and leave the others for you to explore. Keep in mind that each algorithm has its own needs for making it run efficiently.

In outline (with more details to follow), the steps involved in clustering data using Mahout are:

1. Prepare the input. If clustering text, you need to convert the text to a numeric representation.
2. Run the clustering algorithm of choice using one of the many Hadoop-ready driver programs available in Mahout.

3. Evaluate the results.

4. Iterate if necessary.

First and foremost, clustering algorithms require data that is in a format suitable for processing. In machine learning, the data is often represented as a *vector*, sometimes called a *feature vector*. In clustering, a vector is an array of weights that represent the data. I'll demonstrate clustering using vectors produced from Wikipedia documents, but the vectors can come from other areas, such as sensor data or user profiles. Mahout comes with two `Vector` representations: `DenseVector` and `SparseVector`. Depending on your data, you will need to choose an appropriate implementation in order to gain good performance. Generally speaking, text-based problems are sparse, making `SparseVector` the correct choice for them. On the other hand, if most values for most vectors are non-zero, then a `DenseVector` is more appropriate. If you are unsure, try both and see which one works faster on a subset of your data.

To produce vectors from the Wikipedia content (which I have done for you):

1. Index the content into Lucene, being sure to store term vectors for the field you want to generate vectors from. I won't cover the details of this — it's outside the article's scope — but I'll provide some brief hints along with some references on Lucene. Lucene comes with a class called the `EnWikiDocMaker` (in Lucene's `contrib/benchmark` package) that can read in a Wikipedia file dump and produce documents for indexing in Lucene.

2. Create vectors from the Lucene index using the `org.apache.mahout.utils.vectors.lucene.Driver` class located in Mahout's `utils` module. This driver program comes with a lot of options for creating vectors. The Mahout wiki page entitled Creating Vectors from Text has more information (see Resources).

The results of running these two steps is a file like the n2.tar.gz file you downloaded in the Getting started with Mahout section. For completeness, the n2.tar.gz file consists of vectors created from the indexing of all the documents in the Wikipedia "chunks" file that was automatically downloaded by the `ant install` method earlier. The vectors were normalized using the Euclidean norm (or L^2 norm; see Resources). In your use of Mahout, you will likely want to try creating vectors in a variety of ways to see which yields the best results.

Evaluating your results

There are many approaches to evaluating your cluster results. Many people start simply by using manual inspection and ad-hoc testing. However, to be truly satisfied, it is often necessary to use more

in-depth evaluation techniques such as developing a gold standard using several judges. To learn more about evaluating your results, see Resources. For my examples, I used manual inspection to see if some of the results that were clustered together actually made sense. If I were to put this in production, I would use a much more rigorous process.

Given a set of vectors, the next step is to run the k-Means clustering algorithm. Mahout provides driver programs for all of the clustering algorithms, including the k-Means algorithm, aptly named the `KMeansDriver`. The driver is straightforward to use as a stand-alone program without Hadoop, as demonstrated by running `ant k-means`. Feel free to examine the Ant k-means target in the build.xml for more information on the arguments `KMeansDriver` accepts. After the process completes, you can print out the results using the `ant dump` command.

After you've successfully run in stand-alone mode, you can proceed to run in distributed mode on Hadoop. To do so, you need the Mahout Job JAR, which is located in the hadoop directory in the sample code. A Job JAR packages up all of the code and dependencies into a single JAR file for easy loading into Hadoop. You will also need to download Hadoop 0.20 and follow the directions on the Hadoop tutorial for running first in pseudo-distributed mode (that is, a cluster of one) and then fully distributed. For more information, see the Hadoop Web site and resources, as well as the IBM cloud computing resources (see Resources).

Categorizing content with Mahout

Mahout currently supports two related approaches to categorizing/classifying content based on bayesian statistics. The first approach is a simple Map-Reduce-enabled Naive Bayes classifier. Naive Bayes classifiers are known to be fast and fairly accurate, despite their very simple (and often incorrect) assumptions about the data being completely independent. Naive Bayes classifiers often break down when the size of the training examples per class are not balanced or when the data is not independent enough. The second approach, called Complementary Naive Bayes, tries to correct some of the problems with the Naive Bayes approach while still maintaining its simplicity and speed. However, for this article, I'll show only the Naive Bayes approach, because it demonstrates the overall problem and inputs in Mahout.

In a nutshell, a Naive Bayes classifier is a two-part process that involves keeping track of the features (words) associated with a particular document and category and then using that information to predict the category of new, unseen content. The first step, called *training*, creates a model by looking at examples of already classified content and then keeps track of the probabilities that each word is associated with a particular content. The second step, called *classification*, uses the model created during training and the content of a new, unseen document, along with the Bayes Theorem, to predict the category of the passed-in document. Thus, to run Mahout's

classifier, you need to first train the model and then use that model to classify new content. The next section will demonstrate how to do this using the Wikipedia data set.

Running the Naive Bayes classifier

Before you can run the trainer and classifier, you need to do just a little prep work to set up a set of documents for training and a set of documents for testing. You can prepare the Wikipedia files (from those you downloaded via the `install` target) by running `ant prepare-docs`. This splits up the Wikipedia input files using the `WikipediaDatasetCreatorDriver` class included in the Mahout examples. Documents are split based on whether the document has a category that matches one of the categories of interest. The categories of interest can be any valid Wikipedia category (or even any substring of a Wikipedia category). For instance, in this example, I've included two categories: Science and History. Thus, any Wikipedia category that has a category containing the word *science* or *history* (it doesn't have to be an exact match) will be put into a bucket with other documents for that category. Also, each document is tokenized and normalized to remove punctuation, Wikipedia markup, and other features that are not needed for this task. The final results are stored in a single file labeled with the category name, one document per line, which is the input format that Mahout expects. Likewise, running `ant prepare-test-docs` does the same work for the test documents. It is important that the test and training documents do not overlap, which could skew the results. In theory, using the training documents for testing should result in perfect results, but even this is not likely in practice.

After the training and test sets are set up, it's time to run the `TrainClassifier` class via the `ant train` target. This should yield a large amount of logging from both Mahout and Hadoop. Once completed, `ant test` takes the sample test documents and tries to classify them using the model that was built during training. The output from such a test in Mahout is a data structure called a *confusion matrix*. A confusion matrix describes how many results were correctly classified and how many were incorrectly classified for each of the categories.

In summary, you run the following steps to produce classification results:

1. `ant prepare-docs`
2. `ant prepare-test-docs`
3. `ant train`
4. `ant test`

Running all of these (the Ant target `classifier-example` captures all of them in one call), yielding the summary and confusion matrix shown in Listing 6:

Listing 6. Results from running Bayes classifier for history and science

```
[java] 09/07/22 18:10:45 INFO bayes.TestClassifier: history
                                95.458984375    3910/4096.0
[java] 09/07/22 18:10:46 INFO bayes.TestClassifier: science
                                15.554072096128172      233/1498.0
[java] 09/07/22 18:10:46 INFO bayes.TestClassifier: ==================
[java] Summary
[java] -------------------------------------------------------
[java] Correctly Classified Instances          :       4143
                                                     74.0615%
[java] Incorrectly Classified Instances        :       1451
                                                     25.9385%
[java] Total Classified Instances              :       5594
[java]
[java] =======================================================
[java] Confusion Matrix
[java] -------------------------------------------------------
[java] a           b       <--Classified as
[java] 3910        186      |  4096        a     = history
[java] 1265        233      |  1498        b     = science
[java] Default Category: unknown: 2
```

The results of the intermediate processes are stored in the directory named wikipedia in the base directory.

With a results set in hand, the obvious question is: "How did I do?" The summary states that I got roughly 75 percent correct and 25 percent incorrect. At first glance this seems pretty reasonable, especially because it means I did better than randomly guessing. Closer examination shows, however, that I did really well at predicting history (approximately 95 percent correctly) and really poorly at predicting science (approximately 15 percent). In looking for reasons why, a quick look at the input files for training shows that I have a lot more examples of history than science (the file size is nearly double), which is likely one potential problem.

For the test, you can add the `-Dverbose=true` option to `ant test`, which spits out information about each test input and whether it was correctly labeled or not. Digging into this output, you can look up document and examine it for clues as to why it might have been incorrectly classified. I might also try different input parameters and also more science data and retrain the model to see if I can improve the results.

It is also important to think about feature selection for training the model. For these examples, I used the `WikipediaTokenizer` from Apache Lucene to tokenize the original documents, but I did not make much effort to remove common terms or junk terms that might have been incorrectly tokenized. If I were looking to put this classifier in production, I would make a much deeper examination of the inputs and other settings, trying to eke out every last bit of performance.

Just to see if the Science results were a fluke, I tried a different set of categories: Republicans and Democrats. In this case, I want to predict whether a new document is about Republicans or Democrats. To let you try this on your own, I created the

repubs-dems.txt in src/test/resources. I then ran the classification steps via:

```
ant classifier-example -Dcategories.file=./src/test/resources/repubs-dems.txt -Dcat.dir=rd
```

The two `-D` values simply point to the category file and the name of the directory to put the intermediate results in under the wikipedia directory. The summary and confusion matrix from this run looks like Listing 7:

Listing 7. Results from running Bayes classifier for Republicans and Democrats

```
[java] 09/07/23 17:06:38 INFO bayes.TestClassifier: --------------
[java] 09/07/23 17:06:38 INFO bayes.TestClassifier: Testing:
                            wikipedia/rd/prepared-test/democrats.txt
[java] 09/07/23 17:06:38 INFO bayes.TestClassifier: democrats      70.0
                                                                   21/30.0
[java] 09/07/23 17:06:38 INFO bayes.TestClassifier: --------------
[java] 09/07/23 17:06:38 INFO bayes.TestClassifier: Testing:
                          wikipedia/rd/prepared-test/republicans.txt
[java] 09/07/23 17:06:38 INFO bayes.TestClassifier: republicans    81.3953488372093
                                                                   35/43.0
[java] 09/07/23 17:06:38 INFO bayes.TestClassifier:
[java] Summary
[java] -------------------------------------------------------
[java] Correctly Classified Instances          :         56           76.7123%
[java] Incorrectly Classified Instances        :         17           23.2877%
[java] Total Classified Instances              :         73
[java]
[java] =======================================================
[java] Confusion Matrix
[java] -------------------------------------------------------
[java] a           b       <--Classified as
[java] 21          9         |  30          a     = democrats
[java] 8           35        |  43          b     = republicans
[java] Default Category: unknown: 2
```

Although the end result is about the same in terms of correctness, you can see that I did a better job of deciding between the two categories. A quick examination of the wikipedia/rd/prepared directory containing the input documents shows that the two training files were much more balanced in terms of training examples. The examination also shows I have a lot fewer examples overall in comparison to the history/science run, because each file is much smaller than either the history or science training set. Overall, the results at least seem a lot more balanced. Bigger training sets would likely balance out the differences between Republicans and Democrats, although if it didn't, that might imply that one group is better at sticking to its message on Wikipedia — but I'll leave that to the political pundits to decide.

Now that I've shown how to run classification in stand-alone mode, the next steps are to take the code to the cloud and run on a Hadoop cluster. Just as with the clustering code, you will need the Mahout Job JAR. Beyond that, all the algorithms I mentioned earlier are Map-Reduce-ready and should just work when running under the Job submission process outlined in the Hadoop tutorial.

What's next for Mahout?

Apache Mahout has come a long way in just over a year, with significant capabilities for clustering, categorization, and CF, but it also has plenty of room for growth. On the immediate horizon are Map-Reduce implementations of random decision forests for classification, association rules, Latent Dirichlet Allocation for identifying topics in documents, and more categorization options using HBase and other backing storage options. Beyond these new implementations, look for more demos, increased documentation, and bug fixes.

Finally, just as a real mahout leverages the strength and capabilities of the elephant, so too can Apache Mahout help you leverage the strength and capabilities of the yellow elephant that is Apache Hadoop. The next time you have a need to cluster, categorize, or recommend content, especially at large scale, give Apache Mahout a look.

Acknowledgments

Special thanks to fellow Mahout committers Ted Dunning and Sean Owen for their review and insights on this article.

Resources

Learn

- **Machine learning**
  - Machine Learning: Wikipedia's page contains some useful starting information as well as many good references to learn more about machine learning, including approaches such as supervised learning.
  - *Programming Collective Intelligence* (Toby Segaran, O'Reilly, 2007): This book is an excellent starting point for many machine-learning tasks.
  - Artificial Intelligence | Machine Learning: Take Andrew Ng's class at Stanford.
  - Evaluation of clustering: Learn more about evaluating clustering. Also see the discussion on the Mahout mailing list.
  - Bayes Theorem: Read up on how the Bayes Theorem works.
  - L^p space: Understand L^p norms.
- **Apache Mahout and Apache Lucene**
  - Mahout project home page: Discover all that is Mahout.
  - "Map-Reduce for Machine Learning on Multicore": Read the paper that helped launch Mahout.
  - "MapReduce: Simplified Data Processing on Large Clusters" (Google Research Publications): Read the original paper on Map-Reduce.
  - Taste: Explore the Taste documentation.
  - Apache Lucene: Learn more about Lucene.
  - Apache Lucene on developerWorks: Explore Lucene in these articles.
  - Creating Vectors from Text: Read this entry in the Mahout Wiki to learn more on converting your data to Mahout's `Vector` class.
  - Cluster Your Data: Check out this Mahout Wiki page to find out more about how to cluster your data.
- **Apache Hadoop:**
  - Apache Hadoop: Find out more about Hadoop.
  - Hadoop Quick Start Tutorial: Learn how to run a Hadoop Job.
  - HBase: Understand the Hadoop database.

- Browse the technology bookstore for books on these and other technical topics.
- Cloud Computing: Visit the developerWorks Cloud Computing space.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download Hadoop 0.20.0.
- Download a subset of Wikipedia.
- Download a subset of Wikipedia as vectors.
- Get real movie-rating data from the GroupLens project.

Discuss

- Participate in the Mahout community at mahout-user@lucene.apache.org.
- Get involved in the My developerWorks community.

About the author

Grant Ingersoll
Grant Ingersoll is a founder and member of the technical staff at Lucid Imagination. Grant's programming interests include information retrieval, machine learning, text categorization, and extraction. Grant is the co-founder of the Apache Mahout machine-learning project, as well as a committer and speaker on both the Apache Lucene and Apache Solr projects. He is also the co-author of *Taming Text* (Manning, forthcoming) covering open source tools for natural-language processing.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.