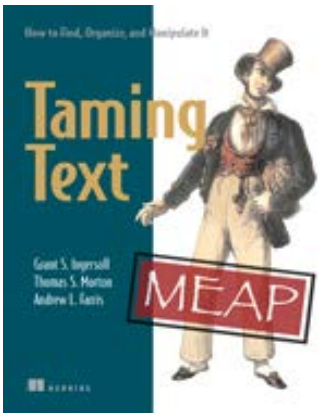
MANNING PUBLICATIONS



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

Mahout also includes a number of classification algorithms that can be used to assign category labels to text documents. One algorithm that Mahout provides is the Naive Bayes algorithm. This article, based on chapter 4 of Taming Text, shows how to use the Mahout implementation of the Naive Bayes algorithm to build a document categorizer.

Training a Naive Bayes Classifier using Apache Mahout

Apache Mahout could be used to group documents into clusters of similar subject area. Mahout also includes a number of classification algorithms that can be used to assign category labels to text documents. One algorithm that Mahout provides is the Naive Bayes algorithm.

This algorithm is used for a wide variety of classification problems and is as an excellent introduction into probabilistic classification. In order to perform class assignments, the algorithms that employ probabilistic classification techniques build a model based on the probability with which document features appear for a given class.

In this article, we will use the Mahout implementation of the Naive Bayes algorithm to build a document categorizer. In this example will develop our own test corpus from data we collected from the Internet and use it to train the classifier. From there, we will demonstrate how training a classifier is an iterative process and present strategies for re-organizing training data in order to improve categorization accuracy. Finally, I will demonstrate how the document categorizer can be integrated into Solr so that documents are automatically assigned to categories when they are indexed using the based on the trained category model. Let us begin by discussing the theoretical underpinnings of the Naive Bayes classification algorithm.

Categorizing text using Naive Bayes Classification

The Naive Bayes algorithm is a probabilistic classification algorithm. It makes its decisions about which class to assign to an input document using probabilities derived from training data. The training process analyzes the relationship between words in the training documents and categories, and then categories and the entire training set. The available facts are collected using calculations based on Bayes' Theorem to produce the probability that a collection of words (a document) belongs in a certain class.

What makes the algorithm naive is the assumption it makes about the independence of words appearing in a given class. Intuitively we know that words do not occur independently in text documents within a given subject area. Words like 'fish' are more likely to occur in documents containing the word *water* than the word *space*. As a result, the probabilities that are produced by the Naive Bayes algorithm aren't true probabilities. They are, nevertheless, useful as relative measures. These probabilities may not predict the absolute probability that a document belongs to a certain class, but they certainly can be used to determine that a document containing *fish* is

more likely about *oceanography* than *space travel* by comparing the probabilities assigned to the term *fish* for each category.

When training, the Naive Bayes algorithm counts the number of times each word appears in a document in the class and divides that by the number of words appearing in that class. This is referred to as a conditional probability, in this case, the probability that a word will appear in a particular category. This is commonly written as P(Word | Category). Imagine you have a small training set that contains three documents for the category *geometry*, and the word *angle* appears in one of the documents; there is a probability of 0.33 or 33% that any document labeled *geometry* would contain the word *angle*.

We can take individual word probabilities and multiply them together to determine the probability of a document given a class. This is not strictly useful on its own but Bayes' Theorem provides a way for us to turn these calculations around to provide the probability of a category given a document—the essence of the classification problem.

Simply put, Bayes' Theorem states that the probability of a category given a document is equal to the Probability of a document given a category multiplied by the probability of the category divided by the probability of a document. This can be expressed as:

```
P(Category | Document) = P(Document | Category) x P(Category) / P(Document)
```

We have shown how to calculate the probability of a document given a category. The probability of a category is simply the number of training documents for a category divided by the total number of training documents. The probability of a document is not needed in this case because it simply serves as a scaling factor. If we set `P(Document) = 1`, the results produced by the function above will be comparable across different categories. We can determine which category a document most likely belongs to by performing this calculation for each class we are attempting to assign to a document. The relationship between these results will have the same relative ranking as long as `P(Document)` is larger than zero for each calculation.

This explanation may be a useful starting point but it only provides part of the picture. The Mahout implementation of the Naive Bayes classification algorithm includes numerous enhancements to account for some of the unique cases where this algorithm falls down in the face of text data such as the problem described above with dependent terms. A description of these enhancements can be found on the Mahout wiki and in the paper "Tackling the Poor Assumptions of Naive Bayes Text Classifiers" by Jason D. M. Rennie and others.

Preparing the training data

A classifier performs only as well as its input. The amount of training data, the way it is organized, and the features chosen as input to the training process all play a vital role in the classifier's ability to accurately categorize new documents.

This section describes how training data must be prepared for use with the Mahout Bayes classifier. I'll demonstrate the process of extracting data from a Lucene index and present the process of bootstrapping to produce a training set using attributes of the existing data. By the end of this example, you will understand how different bootstrapping approaches have an effect on the overall quality of the classifier.

Say we have a simple clustering application using Solr. The application imports content from a number of RSS feeds and stores them in a Lucene index. We would use the data from this index to build a training set. You would run the Data Import Hander a number of times over a period of a few days to build up a reasonable corpus of training documents. Once you have collected some data, we can inspect the index to determine what can be used for training.

Now that you have some data in a Lucene index, we need to review what's there and determine how it can be used to train a classifier. There are a number of ways to view data stored in a Lucene index, but by far the easiest to use is Luke. We'll take a look at the data to determine which fields in the documents can be used as a source of categories on which we will base our categorization scheme. We will determine a set of categories whose contents we will use for training and then extract documents and write them in the training data format used as input to Mahout. The Bayes classifier training process will analyze the words that appear in documents assigned to a particular category and will produce the model that will use to determine the most likely category for a document based upon the words it contains.

You can download the latest release of Luke from http://code.google.com/p/luke/, the file lukeall-version.jar, where version is the current version of Luke, is the one you want. Once you have downloaded the jar file, running the command `java -jar lukeall-version.jar` will start Luke.

Upon startup, you will be greeted with a dialog window that will allow you to browse your file system in order to select the index you wish to open. Choose the directory containing your Lucene index and press the `OK` button to open the index. (The other default options should be fine.)

As you browse through the index with Luke, you will notice that many sources provide categories for their documents. These categories vary from highly general labels such as `Sports`, to the more specific `Baseball`, or even `New York Yankees`.

We will use these entries as a basis for organizing the training data. The goal here is to build a list of terms that we can use to group articles into coarse-grained categories that we'll use to train our classifier. The following list displays the top 12 categories found in the field named `categoryFacet` of the index I put together, each accompanied by the number of documents found in that category.

```
2081 Nation & World
923 Sports
398 Politics
356 Entertainment
295 sportsNews
158 MLB
128 Baseball
127 NFL
115 Movies
94 Sounders FC Blog
84 Medicine and Health
84 Golf
```

You will notice right away that 2,081 appear in the `Nation & World` category and that the number of documents per category drops off pretty quickly, with only 84 articles appearing in `Golf` the 10th ranked category. You will also notice overlapping subject areas like `Sports`, `Baseball`, and `MLB` and or different representations of the same subject such as `Sports` and `sportsNews`. It is our job to clean up this data in such a way that we can use it effectively for training. It is important to take care in the preparation of training data because it can have a significant effect on the classifier's accuracy. To demonstrate this, we will begin with a simple strategy for identifying training documents, follow up with a more complex strategy, and observe the difference in results.

From the list of categories found in the index you see that we have some useful terms appearing at the top of the list. I'll add some other interesting categories from the list of categories found in the index by exploring it with Luke.

```
Nation
Sports
Politics
Entertainment
Movies
Internet
Music
Television
Arts
Business
Computer
Technology
```

Enter this into your favorite text editor and save it to a file named `training-categories.txt`. Now that we have a list of categories we're interested in, run the `ExtractTrainingData` class using the category list and Lucene index as input.

```
$TT_HOME/bin/tt extractTrainingData \
--dir ./index \
--categories ./training-categories.txt \
--output ./category-bayes-data \
--category-fields categoryFacet,source \
--text-fields title,description \
--tv
```

This command will read documents from the Lucene index and search for matching categories in the category and source fields. When one of the categories listed in training-categories.txt is found in one of these documents, the

terms will be extracted from term vectors stored in the title and description fields. These terms will be written to a file in the category-bayes-data directory. There will be a single file for each category. Each is a plain text file that can be viewed with any text editor or display utility.

If you choose to inspect these files, you may notice that each line corresponds to a single document in the Lucene index. Each line has two columns delimited by a tab character. The category name appears in the first column, while each of the terms that appear in the document is contained in the second column. The Mahout Bayes classifiers expect the input fields to be stemmed, so you will see this reflected in the test data. The `--tv` argument to the `extractTraining` data command causes the stemmed terms from each document's term vector to be used.

```
arts 6 a across design feast nut store world a browser can chosen [...]
arts choic dealer it master old a a art auction current dealer find [...]
arts alan career comic dig his lay moor rest unearth up a a arena [...]

business app bank citigroup data i iphon phone say store account [...]
business 1 1500 500 cut job more plan tech unit 1 1500 2011 500 [...]
business caus glee home new newhom sale up a against analyst caution [...]

computer bug market sale what access address almost ani bug call card [...]
computer end forget mean web age crisi digit eras existenti face first [...]
computer mean medium onlin platon what 20 ad attract billion classifi [...]
```

When the `ExtractTrainingData` class has completed its run it will output a count of documents found in each category, similar to the following list.

```
5417 sports
2162 nation
1777 politics
1735 technology
778 entertainment
611 business
241 arts
147 music
115 movies
80 computer
60 television
32 internet
```

Notice that more documents appear in some categories than others. This may affect the accuracy of the classifier. Some classification algorithms like Naïve Bayes tend to be sensitive to unbalanced training data because the probabilities on the features in the categories with a larger number of examples will be more accurate that those on categories with few training documents.

Bootstrapping

This process of assembling a set of training documents using simple rules is known as bootstrapping. In this example we are bootstrapping our classifier using keywords to match existing category names assigned to the documents. Bootstrapping is often required because properly labeled data is often difficult to obtain. In many cases, there simply isn't enough data to train an accurate classifier. In other cases the data comes from a number of different sources with inconsistent categorization schemes. This keyword bootstrapping approach allows us to group documents based upon the presence of common words in the description of the document.

Not all documents in a given category may conform to this particular rule but it allows us to generate a sufficient number of examples to train a classifier properly. Countless bootstrapping techniques exist. Some involve producing short documents as category seeds or using output from other algorithms such as the clustering algorithms or even other types of classifiers. Bootstrapping techniques can be combined to further enhance training sets with additional data.

Withholding test data

Now we need to reserve some of the training data we have produced for testing. Once we have trained the classifier, we will use the model to classify the test data and verify that the categories produced by the classifier are identical to those that are already known for the document. In the code accompanying this book, we include a

utility for executing a simple split called `SplitBayesInput`. We'll point `SplitBayesInput` at the directory the extraction task wrote to and `SplitBayes` will produce two additional directories one containing the training date and the other containing the test data. SplitBayes can be run using the following command-line.

```
$TT_HOME/bin/tt splitBayesInput \
  -i category-bayes-data \
 -tr category-training-data \
 -te category-test-data \
 -sp 10 -c UTF-8
```

In this case we are taking 10% of the documents in each category and writing them to the test directory, the rest of the documents are written to the training data directory. The `SplitBayesInput` class offers a number of different methods for selecting a variety of training/test splits.

Training the classifier

Once the training data has been prepped using `SplitBayesInput`, it is time to roll up our sleeves and train our first classifier. If you are running on a Hadoop cluster, copy the training and test data up to the Hadoop distributed file system and execute the following command to build the classifier model. If you are not running on a Hadoop cluster, data will be read from your current working directory despite the `-source hdfs` argument.

```
$MAHOUT_HOME/bin/mahout trainclassifier \
  -i category-training-data \
  -o category-bayes-model \
  -type bayes -ng 1 -source hdfs
```

Training time will vary depending on the amount of data you are training upon and whether you're executing the training process locally or in distributed mode on a Hadoop cluster.

Once training has completed successfully, a model is written to the output directory specified on the command-line. The model directory contains a number of files in Hadoop Sequence File Format. Hadoop Sequence files contain key, value pairs and are usually the output of a process run using Hadoop's MapReduce framework. The keys and values may be primitive types or Java objects serialized by Hadoop. Apache Mahout ships with utilities that can be used to inspect the contents of these files.

```
$MAHOUT_HOME/bin/mahout seqdumper \
  -s category-bayes-model/trainer-tfIdf/trainer-tfIdf/part-00000 | less
```

The files in the trainer-tfIdf directory contain a list of all of the features the Naive Bayes algorithm will use to perform classification. When dumped, they will produce output like the following:

```
no HADOOP_CONF_DIR or HADOOP_HOME set, running locally
Input Path: category-bayes-model/trainer-tfIdf/trainer-tfIdf/part-00000
Key class: class org.apache.mahout.common.StringTuple
Value Class: class org.apache.hadoop.io.DoubleWritable
Key: [__WT, arts, 000]: Value: 0.9278920383255315
Key: [__WT, arts, 1]: Value: 2.4908377174081773
[..]
Key: [__WT, arts, 97]: Value: 0.8524586871132804
Key: [__WT, arts, a]: Value: 9.251850977219403
Key: [__WT, arts, about]: Value: 4.324291341340667
[..]
Key: [__WT, business, beef]: Value: 0.5541230386115379
Key: [__WT, business, been]: Value: 7.833436391647611
Key: [__WT, business, beer]: Value: 0.6470763007419856
[..]
Key: [__WT, computer, design]: Value: 0.9422458820512981
Key: [__WT, computer, desktop]: Value: 1.1081452859525993
Key: [__WT, computer, destruct]: Value: 0.48045301391820133
Key: [__WT, computer, develop]: Value: 1.1518455320100698
[..]
```

It is often useful to inspect this file to determine that the features you are training upon do indeed relate to those being extracted. Inspecting this output may inform you that you are not properly filtering out stopwords, there is something wrong with your stemmer or you are not producing n-grams when you expect to. It is also useful to inspect the number of features you are training upon as the size of the feature set will impact the Mahout Bayes classifier in terms of memory usage.

Testing the classifier

Once the classifier has been trained, you can evaluate its performance using the test data that we held back earlier. The following command will load the model produced by the training phase into memory and classify each of the documents in the test set. The label assigned to each document by the classifier will be compared to the label assigned to the document manually and results for all of the documents will be tallied.

```
$MAHOUT_HOME/bin/mahout testclassifier \
  -d category-test-data \
  -m category-bayes-model \
  -type bayes -source hdfs -ng 1 -method sequential
```

When the testing process is complete, you will be presented with two results, the percentage of correctly and incorrectly classified instances and the confusion matrix.

```
=======================================================Summary
Correctly Classified Instances     :       906            73.6585%
Incorrectly Classified Instances   :       324            26.3415%
Total Classified Instances         :       1230
=======================================================
Confusion Matrix
-------------------------------------------------------
a    b   c   d   e   f   g   h   I   j   k   l   <--Classified as
0    0   0   0   5   0   0   0   1   0   3   2   |   11  a   = movies
0    0   0   0   0   0   0   0   1   0   1   4   |   6   b   = computer
0    0   0   0   0   0   0   0   0   0   1   2   |   3   c   = internet
0    0   0   4   0   0   0   5   4   0   4   42  |   59  d   = business
0    0   0   1   26  0   0   6   10  0   18  10  |   71  e   = entertainment
0    0   0   0   2   0   0   0   1   0   3   0   |   6   f   = television
0    0   0   0   7   0   1   0   0   2   4   0   |   14  g   = music
0    0   0   0   0   0   0   103 43  0   10  10  |   166 h   = politics
0    0   0   0   1   0   0   25  145 0   16  10  |   197 i   = nation
0    0   0   0   8   0   0   3   7   1   3   1   |   23  j   = arts
1    0   0   0   1   0   0   1   7   0   493 4   |   507 k   = sports
0    0   0   0   0   0   0   15  12  0   7   433 |   167 l   = technology
Default Category: unknown: 12
```

In this case, we can use the confusion matrix to tune our bootstrapping process. The matrix shows us that the classifier did an excellent job classifying sports documents as belonging to the sports category. A total of 493 of 507 instances of sports-related documents were assigned to this class. Technology did well also with 133 of 167 documents assigned to this class. Movies did not do very well; out of 11 documents labeled with the movies class, none of them were properly assigned.

The largest single number of movie-labeled documents was assigned to the entertainment category. This makes sense, considering that movies are a form of entertainment and we had significantly more entertainment documents (713) than movie documents (115) in the training set. This demonstrates the effects of an unbalanced training set and overlapping examples. Entertainment clearly overpowers movies due to the significantly larger number of training documents available for that class, while we also see misclassification of entertainment content as related to the nation, sports and technology as a result of the larger amount of training data in those categories. This instance suggests that we can obtain better accuracy with better subject separation and a more balanced training set.

Improving the bootstrapping process

In the previous example, we used a single term to define each class of documents. `ExtractTrainingData` built groups of documents for each class by finding all documents that contained that class's term in their source or category fields. This produced a classifier that confused several classes due to topic similarity and imbalance in training sets side assigned to each category. To address this issue, we will use a group of related terms to define each class of documents. This allows us to collapse all of the sports-related categories into a single sports category and all of the entertainment related categories into another. In addition to combining similar categories, this approach also allows us to reach further into the pool of documents in our Lucene index and retrieve additional training samples.

Create a file named training-categories-mult.txt containing the labels below.

```
Sports MLB NFL MBA Golf Football Basketball Baseball
Politics
```

```
Entertainment Movies Music Television
Arts Theater Books
Business
Technology Internet Computer Science
Health
Travel
```

In this file, the first word on each line becomes the name of the category. Each word on a line is used when searching for documents. If any word on the line matches a term in the document's category or source field, that document is written to the training data file for that category. For example, any document containing the string MLB in its category field will be considered part of the Sports category, documents possessing a category field containing the term music will be a part of the Entertainment category, while those with category fields containing Computer will be part of the Technology category.

Rerun ExtractTrainingData using the following command-line:

```
$TT_HOME/bin/tt extractTrainingData \
  --dir index \
  --categories training-categories-mult.txt \
  --output category-mult-bayes-data \
  --category-fields categoryFacet,source \
  --text-fields title,description
```

Output will be written to the categories-mult-bayes-data directory and the following document counts will be displayed in your terminal:

```
Category document counts:
5139 sports
1757 technology
1676 politics
988 entertainment
591 business
300 arts
173 health
12 travel
```

It is pretty likely that we will be unable to train the classifier to accurately assign documents to the travel category based on the number of training examples so we might consider collecting additional training documents or discarding the travel category entirely at this point, but I'll leave it there for now to demonstrate the outcome.

Once again, perform the split, training and testing steps:

```
$TT_HOME/bin/tt splitBayesInput \
  -i category-mult-bayes-data \
 -tr category-mult-training-data \
 -te category-mult-test-data \
 -sp 10 -c UTF-8

$MAHOUT_HOME/bin/mahout trainclassifier \
  -i category-mult-training-data \
  -o category-mult-bayes-model \
  -type bayes -source hdfs -ng 1

$MAHOUT_HOME/bin/mahout testclassifier \
  -d category-mult-test-data \
  -m category-mult-bayes-model \
  -type bayes -source hdfs -ng 1 \
  -method sequential
```

The output of the testing phase shows that we've produced an improved classifier that can correctly assign categories 79.5% of the time:

```
Summary
Correctly Classified Instances     :       846     79.5113%
Incorrectly Classified Instances   :       218     20.4887%
Total Classified Instances :       1064
=======================================================
Confusion Matrix
a    b   c   d   e   f   g   h   <--Classified as
0    0   0   0   0   0   1   0   |   1   a   = travel
0    3   0   0   8   0   5   43  |   59  b   = business
0    0   2   1   7   1   2   4   |   17  c   = health
0    1   0   57  12  1   19  9   |   99  d   = entertainment
0    0   0   0   142 0   14  12  |   168 e   = politics
```

```
0    0   0   17  3   3   4   3   |   30  f   = arts
0    1   0   3   9   0   495 6   |   514 g   = sports
0    1   0   1   23  0   7   144 |   176 h   = technology
Default Category: unknown: 8
```

From the output, you'll see that our classifier output has improved 6%, a reasonable amount. Although we're heading in the right direction, from the confusion matrix it is clear that there are other issues that need to be addressed.

We are fortunate that for the goals of this example we have a significant amount of flexibility in terms of obtaining training data and choosing a categorization scheme. First and foremost, it is clear that we don't have enough training data for the travel category because the majority of the document we do have aren't being categorized at all. The health and arts categories suffer from the same problems with the majority of their documents being miscategorized. The fact that the majority of arts documents were assigned the entertainment category suggests that it may be worth combining these classes.

Integrating the Mahout Bayes Classifier with Solr

Once a classifier has been trained, it must be deployed into production. This section will demonstrate how the Mahout Bayes classifier can be integrated into the Solr search engine indexing process as a document categorizer. As Solr loads data into a Lucene index, we also run it through our document categorizer and produce a value for a category field that can be used as an additional search term or for faceted display of results.

This is done by creating a custom `Solr UpdateProcessor` that is called when an index update is received. When it is initialized, this update processor will load the model we have trained for Mahout's Bayes classifier and, as each document is processed, its content will be analyzed and classified. The UpdateProcessor adds the category label as a `Field` in the `SolrDocument` that gets added to the Lucene index.

We begin by adding a custom update request processor chain (see `org.apache.solr.update.processor.UpdateRequestProcessorChain`) to Solr by defining one in solrconfig.xml. This chain defines a number of factories that are used to create the object that will be used to process updates. The `BayesUpdateRequestProcessFactory` will produce the class that is used to process each update and assign a category while the `RunUpdateProcessorFactory` will process the update and add it to the Lucene index built by Solr, the `LogUpdateProcessorFactory` tracks update statistics and writes them to the Solr logs.

Listing 1 Update request processor chain configuration in solrconfig.xml

```
<updateRequestProcessorChain key="mahout" default="true">
 <processor
     class="com.tamingtext.classifier.BayesUpdateRequestProcessorFactory">
   <str name="inputField">details</str>
   <str name="outputField">subject</str>
   <str name="model">src/test/resources/classifier/bayes-model</str>
 </processor>
 <processor class="solr.RunUpdateProcessorFactory"/>
 <processor class="solr.LogUpdateProcessorFactory"/>
</updateRequestProcessorChain>
```

We configure the `BayesUpdateRequestProcessorFactory` using the `inputField` parameter to provide the name of a field that contains the text to classify, the `outputField` parameter to provide a name of the field to write the class label and the model parameter to define the path to the model to use for classification. The `defaultCategory` parameter is optional and, if specified, defines the category that will be added to a document in the event the classifier is unable to make a determination of the correct category label. This commonly happens when the input document contains no features that are present in the model. The factory is created when solr starts up and initializes its plugins. At this time, the parameters validated and the model is loaded through the initialization of a Mahout `Datastore` object. The classification algorithm is created and a `ClassifierContext` is initialized using each of these elements.

Listing 2 Setting up the Mahout `ClassifierContext`

```
Parameters p = new Parameters();
      p.set("basePath", modelDir.getCanonicalPath());
```

```
Datastore ds = new InMemoryBayesDatastore(p);
Algorithm a = new BayesAlgorithm();
ClassifierContext ctx = new ClassifierContext(a,ds);
ctx.initialize();
```

In this listing, we show how the classifier model is loaded into an `InMemoryBayesDatastore`. This is fine for smaller models trained with a modest amount of features but is not practical for models that can not fit into memory. Mahout provides an alternative datastore that pulls data from HBase. It is straightforward enough to implement alternative datastores as well.

Once the `ClassifierContext` is initialized, it is stored as a member variable in `BayesUpdateRequestProcessorFactory` and injected into each new `BayesUpdateRequestProcessor` instantiated when an update request is received by Solr. Each update request arrives in the form of one or more `SolrInputDocuments`. The Solr API makes it trivial to extract a field from a document, and from there it is easy to preprocess and classify the document using the classifier context we initialized above. Listing 3 shows how Preprocessing is achieved using the Solr analyzer, which performs the appropriate preprocessing steps based on the input field's configuration in the solr schema, and the result is written into a `String[]` array, which Mahout's classifier context accepts as input.

The Solr analyzer follows the Lucene `Analyzer` API, so the tokenization code presented here can be used in any context that makes use of the Lucene analyzers.

Listing 3 Tokenizing a `SolrInputDocument` using the Solr analyzer

```
String input = (String) field.getValue();

    ArrayList<String> tokenList = new ArrayList<String>(256);
    TokenStream ts = analyzer.tokenStream(inputField, new StringReader(input));
    while (ts.incrementToken()) {
      tokenList.add(ts.getAttribute(CharTermAttribute.class).toString());
    }
    String[] tokens = tokenList.toArray(new String[tokenList.size()]);
```

Once we have tokens on which the classifier can operate, obtaining the result is as simple as calling the `classifyDocument` method on the Mahout `ClassifierContext`. Listing 4 shows how this operation returns a `ClassifierResult` object containing the label of the document's assigned class. The `classify` method also takes a default category that is assigned in the event that no category can be defined, such as the case where the input document and the model have no words in common. Once the label is obtained, it is assigned to the `SolrInputDocument` as a new field as long as the `ClassifierResult` is not null or equal to the default value for the `defaultCategory` parameter, represented by the constant `NO_LABEL`.

Listing 7.11 Classifying a `SolrInputDocument` using the ClassifierContext

```
SolrInputField field = doc.getField(inputField);
      String[] tokens = tokenizeField(inputField, field);
      ClassifierResult result = ctx.classifyDocument(tokens, defaultCategory);
      if (result != null && result.getLabel() != NO_LABEL) {
        doc.addField(outputField, result.getLabel());
      }
```

A drawback to this approach, depending upon the type of documents being indexed, is the need to hold the results of tokenization in memory in order to provide them to the classifier. In the future perhaps Mahout will be extended to take a token stream as input directly. A second drawback to this approach is the need to effectively tokenize a document's field twice at indexing time.

Tokenization is performed once in the process as classification time and a second time, later on in the processing stream, in order to add the tokens to the Lucene index.

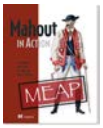
These issues aside, this is an effective mechanism for classifying documents as they are added to a Solr index and demonstrates how to use the Mahout Bayes classifier's API to classify documents programmatically. Both of these mechanisms can be used in your own projects as a way of tagging documents as they are indexed or automatically classifying documents using the Mahout classifier.

Summary

In this article, we explored the Naive Bayes algorithm, a statistical classification algorithm that determines the probability of a set of words given a category based on observations taken from a set of training data. The algorithm then uses Bayes' rule to invert this conditional probability in order to determine the probability of a category, given a set of words from a document.

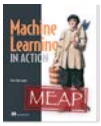
We also investigated techniques for making use of training data collected from the web. We examined the process of bootstrapping, experimented with different approaches towards grouping training documents and demonstrated how the amount of available training data has an impact on classifier accuracy.

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: August 27, 2011