Bruce Miller
12.10.03
Project 3 – Data Mining

Part One – Getting the Software

Done.

Part Two – Pre-Process the Data

A good pre-processing filter would do something about the large number of missing attributes and the data that generates the large number of outliers when visualizing any particular attribute against another. Looking at the flat file, it's obvious that there are numerous entries that have large numbers of missing attributes. One could eliminate all of these by applying something such as a RemoveUseless filter. After trying that out, I noticed it was just filling in the first attribute value in the attribute declaration, which would just result in corrupted results. Upon further investigation, I decided to apply an instance filter, the reasoning for which being a desire to retain only the instances that demonstrated a solid attribute relationship with other instances, simply dropping any instances with missing values, and remove any instances that would generate useless outliers upon later classification. The applied filter and parameters were:

RemoveMisclassified –W "weka.classifiers.rules.OneR –B 6" –C –1 –F 0 –T 0.1 –I 0

OneR is a classifier for building and using a 1-R classifier and predicts the mean (for a numeric class) or the mode (for a nominal class). The RemoveMisclassified threshold does not apply in this case, as the attributes are all of nominal type. Varying the fold modifier had no effect apart from increasing the runtime.

I thought perhaps using a DecisionTable classifier would prove more useful, but it would only remove a few of the missing instances. Varying the subset (-S flag) and the cross validation (-X flag) did not provide any significant improvements.

Applying RemoveMisclassified with OneR reduces the missing percentage to 0 in most attributes and reduces our data from 683 instances to 279.

Part Three – Feature Selection

After some experimentation, I selected the PCA evaluator and Ranker search to eliminate attributes, as it chooses enough eigenvectors to account for a chosen variance. As the varianceCovered is increased, the number of attributes that must be retained increases at an increasing rate. For example, at 0.50, it needs only 6 attributes from the original 36, 14 at 0.75, and 28 at 0.95. 28 is a little much still, so I reduced the variance to 0.80 and was given 17 attributes to keep, thereby reducing the attribute set by over half.

For the search method, PCA requires Ranker, sending it the attributes' column number and corresponding variance explanation percentage. I presume Ranker simply lists them by the latter.

The Attribute Evaluator called was:

PrincipleComponents –R 0.8

and the Search Method was:

Ranker –T –1.7976931348623157E308 –N -1

Part Four – Classification

bayes.NaïveBayes and variations – fast

Correctly Classified Instances	279	100	%
Incorrectly Classified Instances	0	0	%

lazy.kstar.Kstar – somewhat fast

Correctly Classified Instances	279	100	%
Incorrectly Classified Instances	0	0	%

functions.neural.NeuralNetwork – extremely slow

Correctly Classified Instances	279	100	%
Incorrectly Classified Instances	0	0	%

meta.Vote – fast

Correctly Classified Instances	91	32.6165	%
Incorrectly Classified Instances	188	67.3835	%

trees.adtree.DecisionStump - fast

Correctly Classified Instances	221	79.2115	%
Incorrectly Classified Instances	58	20.7885	%

None of these or others I tried resulted in significant improvements to the correctly classified instances. Though NeuralNetwork was the best classifier, its runtime was nearly 30 minutes on a very fast machine. Since NaïveBayes (which completes in under a second) already gives us 100% correctly classified instances, I've opted for NaïveBayes simply for the time considerations.

Part Five – Associative Rule Learning

Running Apriori for the 10 best rules results in:

Best rules found:

1. mycelium=absent 279 ==> int-discolor=none sclerotia=absent 279 conf:(1)
2. int-discolor=none 279 ==> mycelium=absent sclerotia=absent 279 conf:(1)

```
 3. sclerotia=absent 279 ==> mycelium=absent int-discolor=none 279
conf:(1)
 4. mycelium=absent int-discolor=none 279 ==> sclerotia=absent 279
conf:(1)
 5. mycelium=absent sclerotia=absent 279 ==> int-discolor=none 279
conf:(1)
 6. int-discolor=none sclerotia=absent 279 ==> mycelium=absent 279
conf:(1)
 7. int-discolor=none 279 ==> sclerotia=absent 279    conf:(1)
 8. sclerotia=absent 279 ==> int-discolor=none 279    conf:(1)
 9. mycelium=absent 279 ==> sclerotia=absent 279    conf:(1)
10. sclerotia=absent 279 ==> mycelium=absent 279    conf:(1)
```

All of these confidence(1) results are mostly due to the fact that we chose OneR earlier as our preprocess filter, giving us a nice, highly associative dataset.

With 10 conf(1) rules, varying minMetric will have no effect, since we're sorting by confidence metricType.

Part Six – Clustering

Using Expectation Maximization and letting Weka select the number of clusters on its own gives us:

```
Cluster 0 <-- frog-eye-leaf-spot
Cluster 1 <-- anthracnose
Cluster 2 <-- phytophthora-rot
Cluster 3 <-- alternarialeaf-spot

Incorrectly clustered instances :   94.0   33.6918 %
```

Changing the seed to 100 and trying for the same 4 clusters (which it does find) using the FarthestFirst algorithm gives us:

```
Incorrectly clustered instances :   61.0   21.8638 %
```

SimpleKMeans again with seed at 100 and clusters at 4 gives us:

```
Incorrectly clustered instances :   91.0   32.6165 %
```

Obviously, FarthestFirst is the better algorithm. Setting the seed too high causes an increase in error after a certain point, but incrementing the clusters to 5 takes care of a few of the mis-clustered items. By running the following Clusterer call, we get about as good of a result as we can:

```
FarthestFirst –N 5 –S 100

Cluster 0 <-- alternarialeaf-spot
Cluster 1 <-- anthracnose
Cluster 2 <-- phytophthora-rot
Cluster 3 <-- frog-eye-leaf-spot
Cluster 4 <-- diaporthe-stem-canker

Incorrectly clustered instances :   42.0   15.0538 %
```

Part Seven – Report Results

Summary:

In final analysis, it can be said that the choice of pre-processing filter should be carefully considered. Selecting a useless filter, one that would arbitrarily assign attribute values, or one that would throw away perfectly useful data can greatly vary the results being sought from what they actually would be otherwise. For example, having previously chosen ZeroR as the pre-processing filter, and not initially realizing that it was throwing out most of the useful entries along with the useless ones caused rather unimpressive classification rates (best was 47% with NeuralNetwork). OneR seems good, but it's probably not the optimum filter for this dataset, as it probably discards too much as well. However, it did do a good job of eliminating outliers.

With feature selection, PCA + Ranker is certainly an excellent choice, as it is quite simple to select the variance of eigenvectors desired and have PCA return the attributes you should keep. Since capturing the variance in the dataset is the desired goal of feature selection, PCA is optimal, especially on normalized data.

For classification, since NaïveBayes returned 100% correctly classified instances, it was unnecessary to use more computationally expensive methods, like NeuralNetwork. On previous data, filtered through a different pre-processing algorithm, NeuralNetwork would only return a marginal increase in correct classification (around 5%) at a runtime that was increased many thousand-fold. Therefore, both in this well-pre-processed dataset and in most real world situations, NaïveBayes would most likely be preferable.

In associative rule learning, our OneR pre-processed data had already been filtered in regard to the nominal norms, so it is no wonder that the results included large quantities of conf(1) associations. If one were looking for strong links between attributes, this method would prove useful as it returns many strong relationships.

The chosen clustering algorithm, FarthestFirst, finds the most consistently useful relationships between the chosen clusters and the pre-assigned class in the data. If an even smaller clustering error percentage were desired, one could increase the number of clusters to find, but as was shown in the example, the usefulness of further clusters decreases at an increasing rate.

Model Comparison:

In selecting the best model on the grounds that it would produce the best results, I choose Classification using NaïveBayes. Comparison with other classification algorithms only yields the lowest absolute error rates (including NeuralNetwork).

Mean absolute error	0.0001
Root mean squared error	0.0033
Relative absolute error	0.1142 %
Root relative squared error	1.6291 %

Between other analysis types, both clustering and attribute selection yield error rates (though low error rates), and association only returns a selected number of pair relationships, unlike a Bayes algorithm, with would generate a network or relationships.

Summary of NaïveBayes from an accredited source:

Naïve Bayesian classifiers operate under the assumption that attributes within a dataset are relationally independent with respect to their class. An interesting thing about this classification algorithm is that this assumption is false in most real world cases, yet the algorithm performs well.

In a general implementation of this algorithm, an input dataset is broken up into attribute components, the probabilities of such being taken with respect to all components.

Classification takes place on components with respect to a test document or dataset and preference is given to the component's class that exhibits the highest probability. Bayes' Rule is used to come to this conclusion.

Reference:

A. McCallum and K. Nigam. *Comparison of Event Models for Naïve Bayes Text Classification.* 1998.