Pronunciation Modeling for LVCSR **Summer 2000**

Practical Session III: Decision Trees

TeSTIA: The 8TH ELSNET European Summer School *Instructor: Eric Fosler-Lussier*

1 Starting with WEKA

In this session, we'll experiment with building decision trees, mostly to discover patterns in pronunciation data automatically. Before we get to actual pronunciation data, let's get familiar with the WEKA machine learning toolkit.

The WEKA toolkit is actually a large group of JAVA classes that are bundled together. There is a large support structure that I won't have time to go into here, and many types of machine learning algorithms other than decision trees are available within the toolkit. For the moment, though, we'll restrict ourselves to working with the d-tree (decision tree) algorithms.

First, let's look at a sample data file for the "weather" problem. The main idea of this data set is to determine whether we go outside and play dependent on the weather. The file "weather.arff" is the training data file; the particular format of the file is called ARFF[1]. Here's the contents of the file:

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature real
@attribute humidity real
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
...
```

The `@relation` tag gives a name to the data set. The next lines in the file define the possible values for each attribute, using the `@attribute` tag. There are two basic types of variables: nominal variables, such at the `outlook` attribute, take one-of-n values; one can also have real-valued attributes, such as the `temperature` attribute.

The final attribute by default[2] has a special status; it's called the *class attribute*. This is the value that you would like the classifier to predict.

Finally, the data are announced in the file via the `@data` tag. Each line that follows is one data sample, and the values for each attribute are listed.

[1] I have no idea why.

[2] One can change the index of the class attribute using the `-c` option in WEKA.

The decision tree algorithm in WEKA emulates the C4.5 algorithm by Ross Quinlan, a relatively famous decision tree algorithm. It has some more bells and whistles than the bare-bones algorithm I discussed in lecture, but is mostly the same. The WEKA implementation is called J4.8, and can be called like this:

```
prompt> java weka.classifiers.j48.J48 -t weather.arff
```

You'll get a lot of output, which should look like this:

```
J48 pruned tree
------------------
outlook = sunny
|   humidity <= 75: yes (2.0)
|   humidity > 75: no (3.0)
outlook = overcast: yes (4.0)
outlook = rainy
|   windy = TRUE: no (2.0)
|   windy = FALSE: yes (3.0)

Number of Leaves  :     5
Size of the tree :      8

=== Error on training data ===
Correctly Classified Instances          14               100      %
Incorrectly Classified Instances         0                 0      %
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Total Number of Instances               14

=== Confusion Matrix ===
 a b   <-- classified as
 9 0 | a = yes
 0 5 | b = no

=== Stratified cross-validation ===
Correctly Classified Instances           9               64.2857 %
Incorrectly Classified Instances         5               35.7143 %
Mean absolute error                      0.3036
Root mean squared error                  0.4813
Relative absolute error                 63.75   %
Root relative squared error             97.5542 %
Total Number of Instances               14

=== Confusion Matrix ===
 a b   <-- classified as
 7 2 | a = yes
 3 2 | b = no
```

The first part shows the tree that was learned from the training data. Here we can see that the `outlook`

variable was chosen as the first question; when the outlook was sunny or rainy, additional questions were posed to further subdivide the data.

The classification of all of the training data was perfect, as indicated by the first set of statistics. This is not necessarily good, however; it may be an indication that we've over-fitted the training data. The stratified cross-validation section gives a slightly more honest estimate of the error. The training data is divided into 10 sections; the d-tree is trained on 9/10 of the data and tested on the remaining 1/10th. This is repeated 9 more times, changing sections in the training data and testing data, so that all of the data is tested once.

There are various ways to control the algorithm, such as how many examples must be in each leaf (`-M`), or requiring that splits on nominal attributes must be binary (`-B`) or subsets (`-D`). A full list of options is available by typing:

```
prompt> java weka.classifiers.j48.J48 -h
```

To do: Try building d-trees with the different options listed above. How does modifying the options change the tree?

To do: Can you try to predict the outlook given the other variables?

To do: Set up a dataset that corresponds to a problem that you know the answer to, and see if the d-tree software will learn the rule. For example: all men over 40 and all women over 70 are worried about becoming bald. Generate some data points (10-20) that conform to your rule, and train a tree to learn the rule.

2 Training up pronunciation d-trees

I've constructed a number of different ARFF data files that we can use to train d-trees, based on a collection of connected numbers that we have at ICSI (the OGI Numbers corpus). In particular, I'd like to focus on the phone *t* because it's one of the most variable in English. The *t* phone in this corpus is used in words like *twenty, thirty, fifteen,* and *eight* — the way that *t* can be realized is very dependent on the context.

The master ARFF file is called `mastertrain-t.arff`; in it, I have placed a large number of attributes about the context in which each *t* occurs. One can select particular attributes from this corpus by using:

```
prompt> subset-arff.pl 1,2,3,10 mastertrain-t.arff > subsetx-t.arff
```

where 1,2,3,10 correspond to the attribute numbers. By sub-selecting features, one can compare the effects of including or leaving out different attributes. I will also provide a list of different "interesting" subsets.

I have also provided a set of test examples that are separate from the training corpus, in `mastertest-t.arff`. You should use this file (appropriately sub-selected) to test your trees, via the `-T` option to J48. This will give you an honest estimate of the generalization capability of the tree you've constructed.

To do: try constructing trees with different sets of features. Which features are useful, and which are not? Examine the trees you build — do the rules that they come up with make linguistic sense?

3 Converting trees to FSMs *(optional)*

You can see what the generated pronunciations look like in practice by running the d-trees with some test data. I've compiled a complete set of d-trees for all phones; the list of trees is given in `allphones.config`.

An example data file of test data is given for each word (e.g. `twenty.data`). To convert the decisions of the trees into FSMs, I've provided another java program, Classifier2FSM. You can run it like this:

```
prompt> java icsi.Classifier2FSM -a -f allphones.config -i twenty.data
```

This will probably provide a very bushy pronunciation graph. You can prune pronunciations at each d-tree leaf by passing the `-c` option; for example, `-c 0.1` will remove all phone pronunciations with a probability of less than 0.1.

If you compile this into a binary AT&T FSM, you can also use `fsmbestpath` to print out the n best pronunciations. I can provide more information about this if you're interested.

To do: Try constructing data files by concatenating strings of numbers (like "hundred twenty three") and build FSMs for them. (Concatenate `hundred.data`, `twenty.data`, and `three.data`, making sure to change the context). Do you get different results?

To do: You can replace the t tree currently in `allphones.config` with your own tree (if you haven't sub-selected features). To create a classifier file, use the `-d` option to J48. Make a copy of `allphones.config`, put a pointer to your file for the t phone, and try it out!