CS 539 Machine Learning
Project 1

Chris Winsor

Contents:

- Data Preprocessing (2)
  - Discretization (Weka) (2)
  - Discretization (Matlab) (4)
  - Missing Values (Weka) (12)
  - Missing Values (Matlab) (14)
- Attribute Selection (15)
  - Attribute Selection using Correlation, Covariance (Matlab + Custom Cost Function) (15)
  - Attribute Selection using Correlation (Weka) (18)
  - Comparison of Matlab + Custom Cost Function vs Weka CfsSubsetEval (19)
  - Feature selection (Matlab Functions) (20)
- Model Construction (21)
  - Splitting the data into Training and Test datasets (21)
  - Creating a Zero-R model using the Training data set (21)
  - Run the ZeroR model against the Testing data set (22)
  - Run the OneR model against the Spambase data with Training/Testing at 25%/75% (23)
  - Experiment - Varying the number of training samples (24)
  - Experiment - N-fold cross-validation (25)

Data Preprocessing

Discretization (Weka)

weka.filters.supervised.attribute.Discretize

Weka's supervised.attribute.Discretize filter transforms the representation of attributes from numeric into discrete for a supervised data set.

CLI command
java weka.filters.supervised.attribute.Discretize -h

CLI Option Switches

-R col1,col2-col4,...	specify the list of columns (attributes) which are to be discretized.
-V	invert the matching sense of column indices
-D	Discretize to binary values
-Y	Use bin numbers rather than ranges for discretized values
-E	Use better encoding of split point than MDL (minimum description length)
-K	Use Kononenko's MDL method, in place of the default Fayyad & Irani's MDL method

Java Source and Documentation

- Weka-3-7\weka-src\src\main\java\weka\filters\supervised\attribute\Discretize.java
- http://weka.sourceforge.net/doc/weka/filters/supervised/attribute/Discretize.html

The class extends Filters and implements SupervisedFilter. It adds the following methods that describe the functionality unique to this (supervised) Discretize:

setAttributeIndices(String)	set attributes to be discretized
setAttributeIndicesArray(int[])	set attributes to be discretized
getAttributeIndices()	get attributes to be discretized
setMakeBinary(boolean)	set conversion to discrete values
getMakeBinary()	get conversion to discrete values
setUseBetterEncoding(boolean)	set use better encoding than MDL
setUseKononenko(boolean)	set Kononenko MDL encoding
useFilter()	runs the filter (inherited from Filter class)
getCutPoints(int)	once the filter has been run, returns a list of cut points

weka.filters.unsupervised.attribute.Discretize

Weka's unsupervised.attribute.Discretize filter transforms the representation of attributes from numeric into discrete for an unsupervised data set.

CLI command
java weka.filters.unsupervised.attribute.Discretize -h

CLI Option Switches
-unset-class-temporarily Perform the transformation on a supervised data set while treating the class attribute as a normal attribute.

-B	Specifies the maximum number of bins to be created for any attribute
-M	Specifies the desired weight of instances per bin for equal-frequency binning
-F	Use equal-frequency instead of equal-width discretization
-O	Optimizes the number of bins using leave-one-out estimate of estimated entropy
-R	<same as supervised>
-V	<same as supervised>
-D	<same as supervised>
-Y	<same as supervised>

Java Source and Documentation
- Weka-3-7\weka-src\src\main\java\weka\filters\unsupervised\attribute\Discretize
- http://weka.sourceforge.net/doc/weka/filters/unsupervised/attribute/Discretize.html

The class extends from Filters and implements UnsupervisedFilter. It adds the following methods that provide unique functionality.

setAttributeIndices(String)	set attributes to be discretized
setAttributeIndicesArray(int[])	set attributes to be discretized
getAttributeIndices()	get attributes to be discretized
setBins(int)	set the number of bins for the attribute
getCutPoints(int)	get the cut points for the attribute (after run)
setInvertSelection(boolean)	sets invert selection switch
getInvertSelection()	gets invert selection switch

Discretization (Matlab)

hist() and histc() are functions to bin numeric attributes to nominal ones in Matlab.

hist()

hist() can be used in a couple ways

```
n = hist(y)
```

takes Y as a vector or matrix, automatically creates 10 bins. If y is a matrix the data is assumed to be in columns.

In the following example we assume 5 attributes and 100 samples. We bin just the first attribute. This results in 10 bins for the one attribute.

```
xmatrix = rand(100,5);  % 5 attributes, 100 samples
xvector = xmatrix(:,1);  % look at attribute #1
bins = hist(xvector)

bins =
    12     5     7    13     9    14     6    10    12    12
```

Now we take the same data and bin all the attributes. This results in 10 bins for each of the 5 attributes.

```
bins = hist(xmatrix)
bins =
    12    13    12    11     6
     5    10    11     7    14
     7     7     9    11     7
    13     4     9    12     9
     9     4     7     5     6
    14    15    10    11    14
     8     8     7     7     7
     8     8     5    15    11
    15    17    21    11    14
     9    14     9    10    12
```

Notice the bin counts for the first attribute (column 1) are NOT the same as the bin counts when that attributed is binned by itself. This makes me suspect hist() may use the same cutpoints for all the attributes.

The following experiment is performed to confirm the cutpoints are shared for all attributes. Here the first two attributes have values between 0 and 1, the second three attributes have values between 0 and 80.

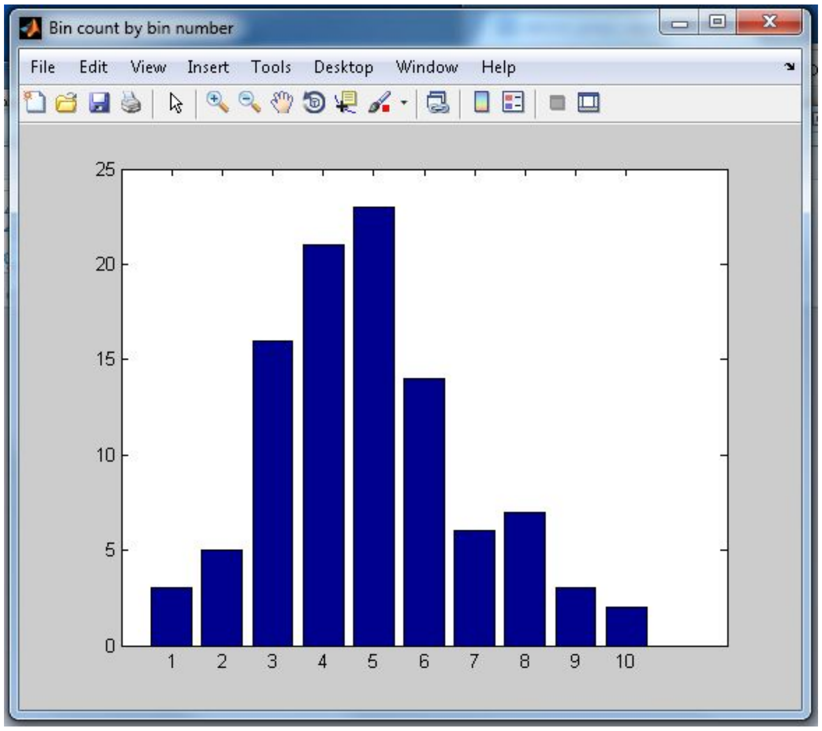
```
xmatrix = [ rand(100,2), (rand(100,3) * 80) ]
bins = hist(xmatrix)

bins =
   100   100     8    15     3
     0     0    17    14    11
     0     0     8     9     7
     0     0    12     8    12
     0     0     8     5    10
     0     0     5    10    13
     0     0    10     7    12
     0     0    16    10     8
     0     0     4    10    10
     0     0    12    12    14
```

We observe the first two attributes ended up entirely in the first bin. This confirms the same cutpoints are used for all attributes.   This is something to watch out for when using hist on an array!

Plotting and hist()

Plotting is the natural output of hist. Here we create random data that has a normal distribution, discretize it, and plot (by bin number)

```
xmatrix = randn(100,5);
xvector = xmatrix(:,1);
bins = hist(xvector);
f1 = figure('Name','Bin count by bin number','NumberTitle','off');
figure(f1);
bar(bins);
```



histc()

histc() is the Matlab function which transforms an attribute from numeric to discrete using user-specified cutpoints.

The first vector to histc(x,y) are the data points, the second vector are the cut points.

There is a challenge in identifying the cut points. This is best illustrated by the following example. Here we create 4 datapoints and want to cut the data at what seems to be 3 natural points, but the result is unexpected:

```
>> histc( [3,5,7,9], [4,6,8] )
ans =     1     1     0
```

This is unexpected because a) the total of 2 does not represent the number of samples in our data which was 4 and b) the number of bins is three but we had 3 cuts so we would expect either 2 or 5 bins depending if matlab creates bins outside our cuts.

The second try then attempts to understand how data is handled below the low cutpoint by adding a -inf to the cutpoint list.

```
>> histc( [3,5,7,9], [-inf,4,6,8] )
ans =     1     1     1     0
```

This is satisfying because it illustrates we have captured the datapoint "3".

But we are still missing one datapoint. So our third try attempts to understand how data is handled above the high cutpoint by adding +inf to the cutpoint list.

```
>> histc( [3,5,7,9], [-inf,4,6,8,inf] )
results in
ans =     1     1     1     1     0
```

The total of 4 confirms we have indeed captured the final datapoint "9" in our [8 to inf] bin. But another bin has been added (equal to or greater than inf). This is confirmed by the documentation for histc which describes behavior as:

> " The last bin counts any values of x that match edges(end)"

The conclusion is that we want to be careful in designing cutpoints for use with histc() so as to avoid systematically creating cuts that are on data points. This would occur if our algorithm sets low and high cutpoints to be min or max data values. For example the following code is a BAD implementation:

```
attribute_values = rand(100,1);  % create random attribute data
cuts = min(attribute_values) : (max(attribute_values)-min(attribute_values)) / 3 :
(max(attribute_values));   % create cuts using min and max values
histc(attribute_values,cuts)
```

```
ans =
   31
   35
   33
```

1

This is problematic because it will ALWAYS result in the high cutpoint being equal to the max(attribute_value) and this will result in top bin containing just this single data element, as confirmed by the result:

To resolve this - a slight tweak to the above algorithm makes the high cutpoint slightly greater than max(attribute_values) so that max(attribute_value) is not == the cutpoint and thus will fall into the next-to-highest bin, as follows:

```
high_cutpoint = max(attribute_values) * 1.01  % high cutpoint just above max data
low_cutpoint  = min(attribute_values) * 0.99  % low cutpoint just below min data
```

or more accurately (to support cases where data values are negative)

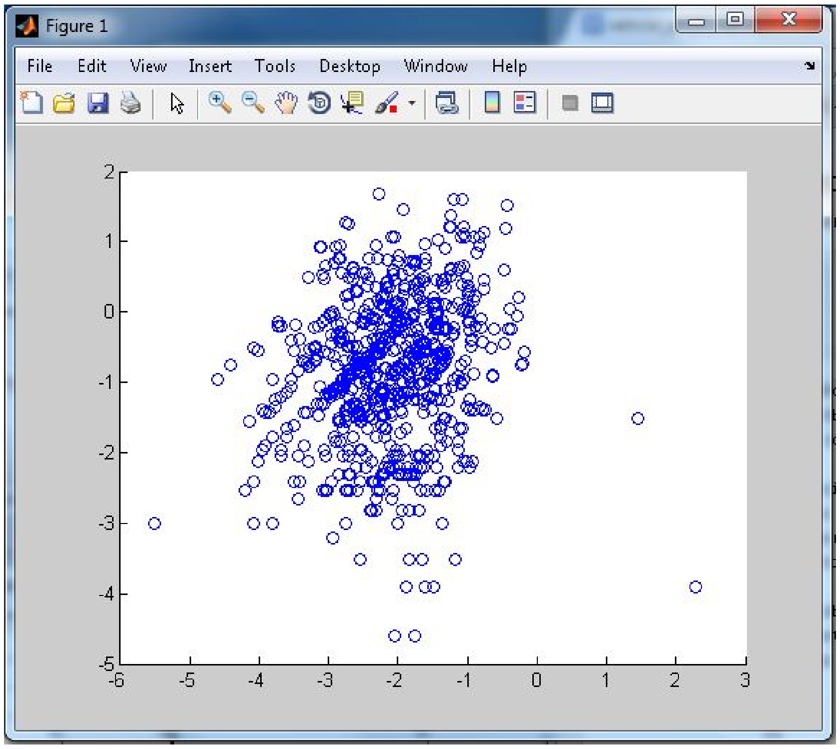
```
high_cutpoint = max(attribute_values) + ((abs(max(attribute_values))* 0.01);
low_cutpoint  = min(attribute_values) - ((abs(min(attribute_values))* 0.01);
```

scatter() and scatter3()

Where the data values are numeric and we want to visualize relationships between them, the Matlab scatter() and scatter3() are helpful visualization tools.

Here we analyze spambase attributes 11 and 16 to look for correlation. Because they are highly distributed around 0 I have taken the log of each.

```
scatter( log(a(:,11)), log(a(:,16)) );
```



We see they appear to be not correlated.

hist() and bar()

Where the class variable is discrete (as it is in our spambase data), and we want to analyze an attribute with respect to the class variable, it is helpful to view the data as a bar chart with the attribute values binned but separated for each value of the class variable. This is exactly what Weka shows in its Explorer window. But we think using Matlab will give us control over the process and allow more flexibility in our analysis.

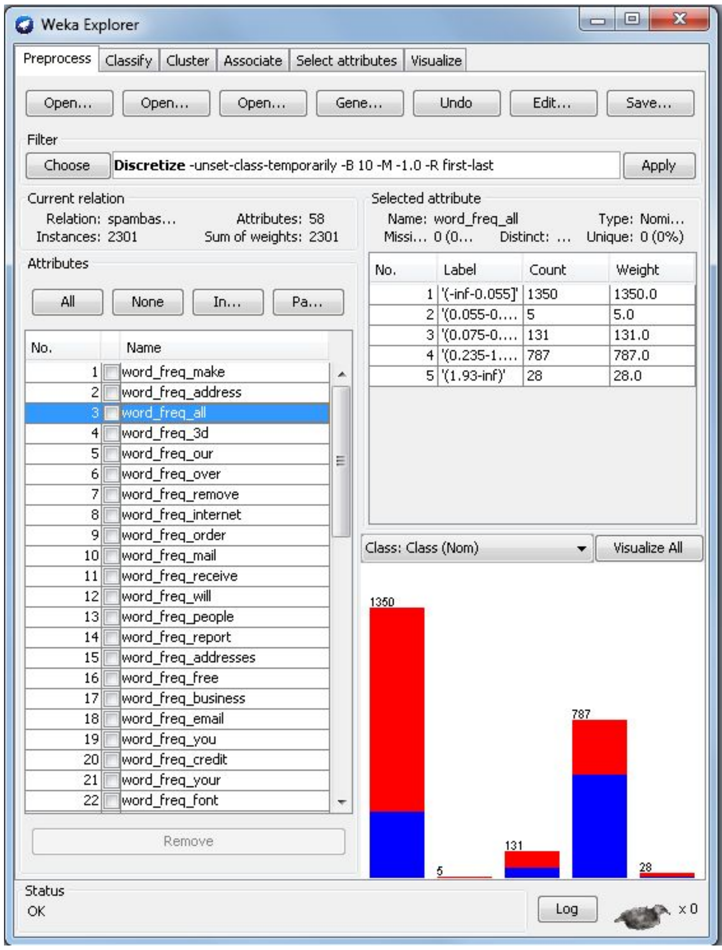
To do this

1. Split the database into subset-databases based on class attribute values. If there are N values for the class variable there will be N subset-databases.
2. Choose the attribute you wish to analyze.
3. Do a hist() on the original database to get bin centerpoints for the attribute over the entire database.
4. Using those centerpoints do a hist(), specifying centerpoints, on each subset-database. Each will result give a bin count. Do this for each subset-database (each value of the class variable).
5. Use bar() to display the bin counts.

Example - analyze spambase attribute 1 against the class variable.

```
%%%%% <import spambase data (use the import wizard, from .csv file) into "untitled">
a = untitled;
[nr,nc] = size(a);
%%%% Split the database by values of the class variable
instances_w_class_val_0 = [];
instances_w_class_val_1 = [];
for instance_num = 1:nr
  if (a(instance_num,nc)==1)
    instances_w_class_val_0 = [instances_w_class_val_0; a(instance_num,:)];
  end
  if (a(instance_num,nc)==0)
    instances_w_class_val_1 = [instances_w_class_val_1; a(instance_num,:)];
  end
end
%%%%% thus we have split the database into two subsets
size(instances_w_class_val_1)
size(instances_w_class_val_0)
%%%%% choose attribute to analyze
att_num = 3;
%%%% specify the number of bins then
%%%%% get centerpoints for that attribute on entire database
num_bins = 50;
[n,bin_centers] = hist(a(:,att_num), num_bins);
%%%%% for each of the class variable values, get a histogram for the attribute
attribute_bins_cv0 = hist(instances_w_class_val_0(:,att_num), bin_centers);
attribute_bins_cv1 = hist(instances_w_class_val_1(:,att_num), bin_centers);
%%%%% draw a bar chart X val are the bin centers, Y is the two histograms side-by-side
bar( bin_centers, [ attribute_bins_cv0; attribute_bins_cv1 ]', 1);
%%%%% for this one attribute, I will draw another figure focusing on just the first 15 bins
nbins = 15;
bar( bin_centers(1:nbins), [ attribute_bins_cv0(1:nbins); attribute_bins_cv1(1:nbins) ]', 1);
```

Here are the results, with Wika's cutpoints for attribute 1 on the left, and our Matlab analysis on the right for the same attribute on the right.



Filter

Choose **Discretize** -unset-class-temporarily -B 10 -M -1.0 -R first-last Apply

Current relation

Relation: spambas... Attributes: 58
Instances: 2301 Sum of weights: 2301

Selected attribute

Name: word_freq_all Type: Nomi...
Missi... 0 (0... Distinct: ... Unique: 0 (0%)

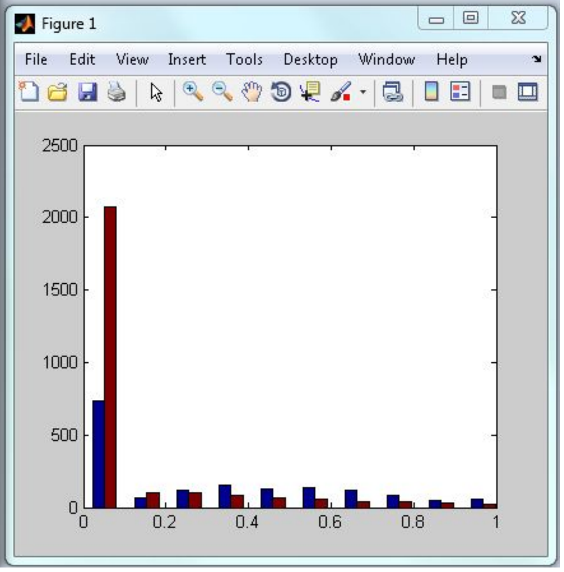
No.	Label	Count	Weight
1	'(-inf-0.055]'	1350	1350.0
2	'(0.055-0....	5	5.0
3	'(0.075-0....	131	131.0
4	'(0.235-1....	787	787.0
5	'(1.93-inf)'	28	28.0

Class: Class (Nom)

Attributes

No.	Name
1	word_freq_make
2	word_freq_address
3	word_freq_all
4	word_freq_3d
5	word_freq_our
6	word_freq_over
7	word_freq_remove
8	word_freq_internet
9	word_freq_order
10	word_freq_mail
11	word_freq_receive
12	word_freq_will
13	word_freq_people
14	word_freq_report
15	word_freq_addresses
16	word_freq_free
17	word_freq_business
18	word_freq_email
19	word_freq_you
20	word_freq_credit
21	word_freq_your
22	word_freq_font

Status
OK



From Matlab (Figure 1) we can see there should be a cutpoint around 0.2 where the blue (left) bars begins to exceed the brown (right) bars. In Weka we see they did choose a cutpoint at 0.235. But Weka also identified cutpoints at 0.055 and 0.075 which are at values below the resolution of our Matlab bins. Weka also identified a cutpoint at 1.93 which is too far to the right in our Matlab Figure 1. In Matlab we appear to have found one good cutpoint but missed 3 that Weka thought were important.

We conclude that:
1) Mablab hist() chooses centerpoints which are equally spaced without regard to the class variable. Weka Discretize considers the class variable value when choosing cutpoints with the intent of bringing out the relationship between the attribute and the class variable.

2) In this case Weka established 5 bins. This is more than most other word attributes many of which were given just 2 or 3. Weka apparently felt there was a complex relationship between this variable and the class variable which justified the additional bins. It is clear Weka is performing a fairly sophisticated analysis to make the determination of number, and placement of bins.

3) The analysis we did using Matlab is flexible but time consuming. We found one good cutpoint but we would need to spend a lot more time, or use a more sophisticated technique to find the other cutpoints that Weka found.

Missing Values (Weka)

Creating Missing Values (Weka)

To create missing values we start with the discretized data with better encoding.

```
cp 01_discretized_encoding_better.arff  02_before.arff
```

We then split the database into two subsets. We do this by running Resample specifying seed without replacement for 5% of the data. Then re-run inverting the selection for 95% of the data. The two comprise the entire database.

```
java weka.filters.unsupervised.instance.Resample -S 1 -Z 5 -no-replacement -i
proj1\02_before.arff   -o proj1\02b_resample_05pct.arff
java weka.filters.unsupervised.instance.Resample -S 1 -Z 5 -V -no-replacement -i
proj1\02_before.arff  -o proj1\02b_resample_95pct.arff
```

We then zero the attribute 55 in the 5% subset.

```
java weka.filters.unsupervised.attribute.Remove -R 55  -i
proj1\02b_resample_05pct.arff  -o proj1\02b_resample_05pct_zeroed.arff
java weka.filters.unsupervised.attribute.Add  -N capital_run_length_average -C 55 -i
proj1\02b_resample_05pct_zeroed.arff   -o proj1\02b_resample_05pct_zeroed2.arff
```

We next remove the header from the zeroed subset to prepare it for merging, and merge with the rest of the data.
emacs 02b_resample_05pct_zeroed2.arff
cat 02b_resample_95pct.arff 02b_resample_05pct_zeroed2.arff > 02_with_zeros.arff

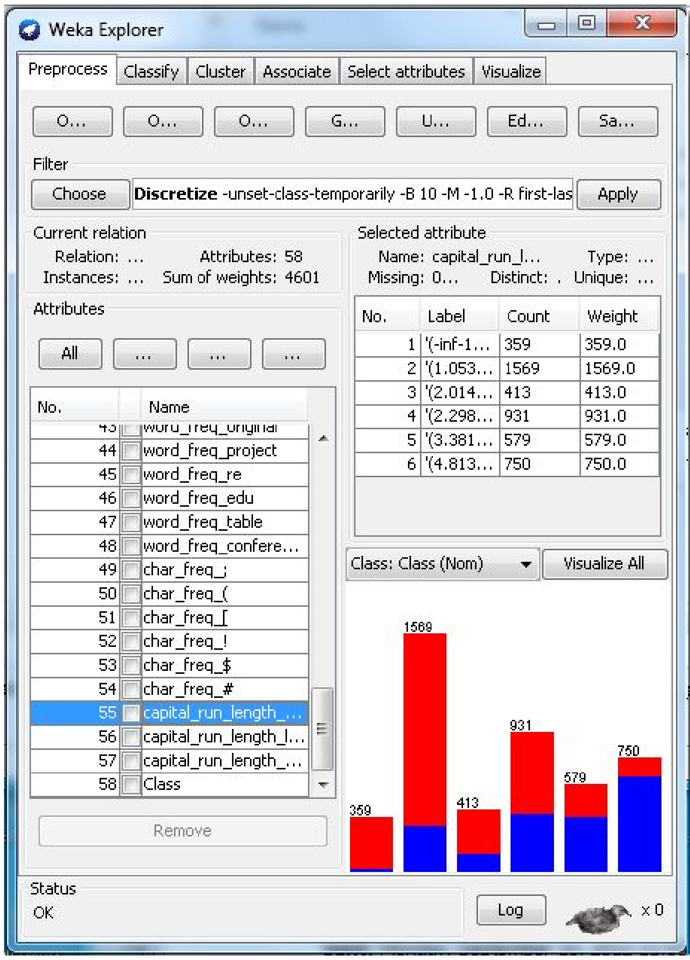
Replacing missing values (Weka)

The filter to replace missing samples is weka.filters.unsupervised.attribute.ReplaceMissingValues. For Nominal attributes the mode is used, for numeric values the mean. Although this is an unsupervised filter there is an ignoreClass switch. For our supervised data the -ignoreClass switch wants to be clear (the default).

```
java weka.filters.unsupervised.attribute.ReplaceMissingValues  -i
proj1\02_with_zeros.arff  -o proj1\02_after.arff
```

Results after Replacement (Weka)

The figure below shows Attribute 55 in the original dataset, and then after having been removed and replaced by the Weka ReplaceMissingValues(). The database is observed to have the same total instances (4601). All bins have decreased sample counts except for bin 2 which has increased and coincidentally had the most entries to start with. This is consistent with the Weka's ReaplaceMissingValues() policy which, in the case of nominal attribute, is to replace unknown values with the mode.



Filter

Choose **Discretize** -unset-class-temporarily -B 10 -M -1.0 -R first-las Apply

Current relation

Relation: ... Attributes: 58
Instances: ... Sum of weights: 4601

Selected attribute

Name: capital_run_l... Type: ...
Missing: 0... Distinct: . Unique: ...

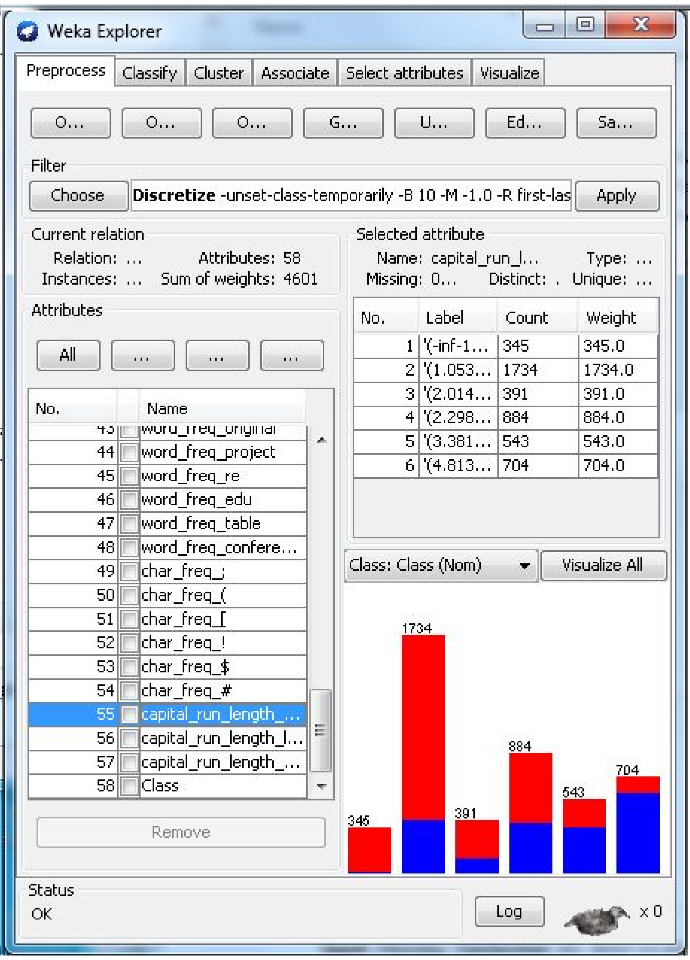
No.	Label	Count	Weight
1	'(-inf-1...	359	359.0
2	'(1.053...	1569	1569.0
3	'(2.014...	413	413.0
4	'(2.298...	931	931.0
5	'(3.381...	579	579.0
6	'(4.813...	750	750.0

Attributes

No.	Name
43	word_freq_original
44	word_freq_project
45	word_freq_re
46	word_freq_edu
47	word_freq_table
48	word_freq_confere...
49	char_freq_;
50	char_freq_(
51	char_freq_[
52	char_freq_!
53	char_freq_$
54	char_freq_#
55	capital_run_length_...
56	capital_run_length_l...
57	capital_run_length_...
58	Class

Class: Class (Nom)

Status
OK



Filter

Choose **Discretize** -unset-class-temporarily -B 10 -M -1.0 -R first-las Apply

Current relation

Relation: ... Attributes: 58
Instances: ... Sum of weights: 4601

Selected attribute

Name: capital_run_l... Type: ...
Missing: 0... Distinct: . Unique: ...

No.	Label	Count	Weight
1	'(-inf-1...	345	345.0
2	'(1.053...	1734	1734.0
3	'(2.014...	391	391.0
4	'(2.298...	884	884.0
5	'(3.381...	543	543.0
6	'(4.813...	704	704.0

Attributes

No.	Name
43	word_freq_original
44	word_freq_project
45	word_freq_re
46	word_freq_edu
47	word_freq_table
48	word_freq_confere...
49	char_freq_;
50	char_freq_(
51	char_freq_[
52	char_freq_!
53	char_freq_$
54	char_freq_#
55	capital_run_length_...
56	capital_run_length_l...
57	capital_run_length_...
58	Class

Class: Class (Nom)

Status
OK

Missing Values (Matlab)

Matlab functions to replace missing (NaN) include find(), Logical Indexing, and hist().
The source of this material is a good article on matrix indexing in Matlab
http://www.mathworks.com/company/newsletters/articles/Matrix-Indexing-in-MATLAB/matrix.html

Replace NaN values with Zero in Matlab using find()

It is possible to replace NaN values in a matrix using "find"

```
nan_locations = find(isnan(A));
A(nan_locations) = 0;
```

Replace NaN values with a default value

Using logical indexing it is possible to directly replace values

```
B(isnan(B)) = 5
```

Replace NaN values with attribute's non-NAN average

```
% for attribute #2
num_nan_in_column = sum(isnan(A(:,2)))
A(nan_locations) = 0;
attribute_non_nan_mean =  sum(A(:,2)) / (num(A(:,2) - num_nan_in_column;
```

Replace NaN values with an explicit "not present" value

```
the_bins = hist(A(:,2),5)  % convert the second attribute into 5 bins
the_bins = [the_bins, sum(isnan(A(:,2))) ]  % add a bin for the NaN values, and give it the count of NaNs for that attribute
```

Attribute Selection

Attribute Selection using Correlation, Covariance (Matlab + Custom Cost Function)

To choose attributes for removal using Matlab we will identify those attributes with the overall highest level of correlation to other attributes. To make this more specific, we define "*overall correlation*" using a cost function. This cost function is defined as the sum of the squares of that attribute's correlation to other attributes. For example if we have N attributes and we want to establish the overall correlation for attribute M, we would establish the correlation between attribute M and all the others, square those, and sum. Attributes with a high degree of overall correlation to all the others will be candidates for removal.

overall_correlation_for_attributeM = sum_over_attributes_1_to_N (corr(Am,An) ^2)

The motivation behind this cost function is that the square penalizes higher correlation more aggressively. The square also eliminates the sign where correlation is negative (we want only magnitude).

<u>Small Database</u>
A small example tests the algorithm. Create random data:

```
q = rand(3,5)

q =
    0.2575    0.8143    0.3500    0.6160    0.8308
    0.8407    0.2435    0.1966    0.4733    0.5853
    0.2543    0.9293    0.2511    0.3517    0.5497
```

Compute the correlation matrix, then the square of the correlations.

```
cc = corr(q)
cc_squared = cc .* cc

cc_squared =
    1.0000    0.9770    0.5909    0.0017    0.1534
    0.9770    1.0000    0.4391    0.0123    0.0612
    0.5909    0.4391    1.0000    0.4500    0.7914
    0.0017    0.0123    0.4500    1.0000    0.8751
    0.1534    0.0612    0.7914    0.8751    1.0000
```

Sum each column to find the cost function for each attribute.

```
cost_function = sum(cc_squared)

cost_function =
    2.7230    2.4896    3.2714    2.3392    2.8812
```

Identify the column with the maximum value (highest overall correlation).

```
[max_cost_function_val, column_with_max_cost_function_val] = max(cost_function)

max_cost_function_val =
    3.2714
column_with_max_cost_function_val =
     3
```

So based on this measure it is attribute 3 that has the highest overall correlation to other attributes and is the best candidate for removal.

To find the next candidate we zero the first attribute's column in the cost_function matrix to reveal the attribute with the next highest overall correlation.

```
cost_function(column_with_max_cost_function_val) = 0;
[max_cost_function_val, column_with_max_cost_function_val] = max(cost_function)

>> cost_function(column_with_max_cost_function_val) = 0
cost_function =
    2.7230    2.4896         0    2.3392    2.8812
>> [max_cost_function_val, column_with_max_cost_function_val] =
max(cost_function)
max_cost_function_val =
    2.8812
column_with_max_cost_function_val =
     5
```

An additional piece of analysis is simply to sort the cost_function matrix. This ranks the attributes in decreasing order of importance (lowest value = lowest correlation is first).

```
[cost_function_vals, cost_function_index] = sort(cost_function)
cost_function_vals =
    2.3392    2.4896    2.7230    2.8812    3.2714
cost_function_index =
     4     2     1     5     3
```

So the most important attributes are 4 and 2, the least important are 5 and 3.

spambase Database

We now perform this analysis on the spambase dataset

```
%Import the .csv file to 'untitled' using the import wizard ignoring the
header.
q = untitled;  % put that data into q
cc = corr(q);
cc_squared = cc .* cc;
cost_function = sum(cc_squared);
[max_cost_function_val, column_with_max_cost_function_val] =
max(cost_function);
[cost_function_vals, cost_function_index] = sort(cost_function)

cost_function_index =
  Columns 1 through 16
     4    47     2    48    44    38    51    54    14    33    45    12    41
10    13    39
  Columns 17 through 32
    20    27    49    16     8    46    52    22    24     5    18     1     3
6    42    43
  Columns 33 through 48
    55     7    37    17    11     9    15    53    19    57    23    21    50
56    26    25
  Columns 49 through 58
    29    58    35    28    30    31    36    40    34    32
```

Attributes 32 and 34 are the first two recommended for elimination when the algorithm is used with correlation.

Using Covariance

Covariance is similar to correlation (correlation = covariance * the ratio of the standard deviations). We will use the same algorithm as above for covariance.

```
q = untitled;
cc = cov(q);
cc_squared = cc .* cc;
cost_function = sum(cc_squared);
[max_cost_function_val, column_with_max_cost_function_val] = max(cost_function);
[cost_function_vals, cost_function_index] = sort(cost_function)

cost_function_index =
  Columns 1 through 16
    47    51    38    48    37    33    16     7    43    41    40     5    49    34
19    32
  Columns 17 through 32
     8    12    31    36    54    39     6    11    35    44    18     1     2    30
17     4
  Columns 33 through 48
    13    29    52    24    28     3    20    15    46    42    50    53    26    14
23    10
  Columns 49 through 58
```

```
    21      9    25    45    22    58    27    55    56    57
```
Attributes 57 and 56 are recommended for elimination when the algorithm is used with covariance.

Attribute Selection using Correlation (Weka)

CfsSubsetEval

When CfsSubsetEval is run with default switches it chooses to keep 15 attributes.

```
=== Run information ===

Evaluator:    weka.attributeSelection.CfsSubsetEval
Search:       weka.attributeSelection.BestFirst -D 1 -N 5
Relation:     spambase
Instances:    4601
Attributes:   58
...

=== Attribute Selection on all input data ===

Search Method:
	Best first.
	Start set: no attributes
	Search direction: forward
	Stale search after 5 node expansions
	Total number of subsets evaluated: 754
	Merit of best subset found:    0.456

Attribute Subset Evaluator (supervised, Class (nominal): 58 Class):
	CFS Subset Evaluator
	Including locally predictive attributes

Selected attributes: 4,5,7,16,21,23,24,25,27,42,44,46,52,53,55 : 15
                     word_freq_3d
                     word_freq_our
                     word_freq_remove
                     word_freq_free
                     word_freq_your
                     word_freq_000
                     word_freq_money
                     word_freq_hp
                     word_freq_george
                     word_freq_meeting
                     word_freq_project
                     word_freq_edu
                     char_freq_!
                     char_freq_$
                     capital_run_length_average
```

Comparison of Matlab + Custom Cost Function vs Weka CfsSubsetEval

The attributes that our Matlab cost_function chose for removal (32,34,56,57) were also recommended for removal by CfsSubsetEval.

Looking deeper to the top 10 attributes recommended for removal by the cost function
```
       corr -  29    58    35    28    30    31    36    40    34    32
       cov -   21     9    25    45    22    58    27    55    56    57
```
only four were chosen to by CfsSubsetEval as keepers (21, 25, 27, 55).

The conclusion is that the Matlab cost_function we defined did a pretty good job of choosing attributes for removal.

Feature selection (Matlab Functions)

Matlab defines the process of selecting features as "sequential feature selection". There are two pieces to sequential feature selection - the objective function and the sequential search algorithm. The objective function is that which the method tries to minimize through its choice of attributes. The sequential search algorithm adds or removes attributes in its attempt to maximize the objective function.

Sequential forward selection (SFS) - attributes are added until additional features provide no further benefit to the objective function.
Sequential backward selection (SBS) - attributes are removed (from the complete list) until the removal begins to deteriorate the objective function.

relieff() and rrelieff()
These Matlab functions ranks attributes as to their ability to predict the class variable. They return a vector with relative weights as to ability to predict. relieff() is used for classification using discrete attributes. rrelieff() is used for k-nearest-neighbors regression with normal variables.

relieff() run against spambase:

```
>> a = untitled;
>> size(a)
ans =
        4601          58

>> relieff(a(:,1:57),a(:,58),5)

ans =
  Columns 1 through 16
    19    21    18     7    52    16    53    24     9     8     1     5    11    10
50    17
  Columns 17 through 32
    57     6    23     2    20    13    56    12    38    40    55    45    22    49
46    54
  Columns 33 through 48
     3    14     4    28    51    47    34    27    32    33    31    41    35    36
48    44
  Columns 49 through 57
    15    39    29    42    37    43    30    25    26
```

The attributes chosen by Matlab's relieff() does not seem related to those chosen by Weka CfsSubsetEval.

Model Construction

Splitting the data into Training and Test datasets

Here we create datasets for training and testing starting from the original spambase which was discretized using "better" encoding. We specify 4 folds with the first fold as training, the remaining folds as testing.

```
java weka.filters.supervised.instance.StratifiedRemoveFolds -S 0 -N 4 -F 1  -c "last"
-i proj1\01_discretized_encoding_better.arff  -o proj1\03a_training.arff
java weka.filters.supervised.instance.StratifiedRemoveFolds -S 0 -N 4 -F 1 -V  -c
"last"  -i proj1\01_discretized_encoding_better.arff  -o proj1\03a_testing.arff
```

Creating a Zero-R model using the Training data set

Here we create a model using ZeroR using the training data set. ZeroR can automatically segment the data into training and testing create folds and perform cross-validation but here we disable that as we want to save the model and use it later against the testing data. The -i and -k switches give more verbose output on the model. The file for the output model is specified by the -d switch.

```
 java weka.classifiers.rules.ZeroR -no-cv  -i -k  -t proj1\03a_training.arff    -d
proj1\03b_model.xxx

+================= output ===========
ZeroR predicts class value: 0

=== Error on training data ===
Correctly Classified Instances         697               60.556  %
Incorrectly Classified Instances       454               39.444  %
Kappa statistic                          0
K&B Relative Info Score                  0      %
K&B Information Score                    0      bits      0      bits/instance
Class complexity | order 0            1113.7134 bits      0.9676 bits/instance
Class complexity | scheme             1113.7134 bits      0.9676 bits/instance
Complexity improvement     (Sf)          0      bits      0      bits/instance
Mean absolute error                      0.4778
Root mean squared error                  0.4887
Relative absolute error                100      %
Root relative squared error            100      %
Coverage of cases (0.95 level)         100      %
Mean rel. region size (0.95 level)     100      %
Total Number of Instances             1151
-===============================
```

The output shows the ZeroR achieved 60.56% correct classification on the training set. There are about 1150 instances in the data. ZeroR is based solely on the class attribute and this case always guesses "0".

Run the ZeroR model against the Testing data set

Here we test the model's ability to predict the Testing data set. The testing data should be statistically close to the training data so presumably the model should achieve a similar accuracy. The -T switch specifies the testing data set and the -l switch specifies the file with the model.

```
java weka.classifiers.rules.ZeroR -no-cv  -i  -T proj1\03a_testing.arff    -l
proj1\03b_model.xxx

+================ output ===========
ZeroR predicts class value: 0

=== Error on test data ===
Correctly Classified Instances        2091               60.6087 %
Incorrectly Classified Instances      1359               39.3913 %
Kappa statistic                          0
Mean absolute error                      0.4776
Root mean squared error                  0.4886
Coverage of cases (0.95 level)         100      %
Total Number of Instances             3450
-===============================
```

It is shown that the ZeroR model predicted the target with 60.61% accuracy when run using the training data.   There are about 3400 instances in the data. The accuracy is almost exactly the same as its performance on the training set which is what we expected.

Run the OneR model against the Spambase data with Training/Testing at 25%/75%

Here we run OneR against the same spambase dataset. In this case we allow the tool to split the data, train, and test using those splits. We use the same split ratio of 25%/75% for testing and training as used in the ZeroR test.

```
 java weka.classifiers.rules.OneR  -t proj1\01_discretized_encoding_better.arff
-split-percentage 25 -i -k

+================ output ===========
=== Error on training split ===
Correctly Classified Instances         920               80      %
Incorrectly Classified Instances       230               20      %
Kappa statistic                          0.5887
K&B Relative Info Score              65174.7184 %
K&B Information Score                  626.0684 bits      0.5444 bits/instance
Class complexity | order 0            1104.531  bits      0.9605 bits/instance
Class complexity | scheme           247020      bits    214.8    bits/instance
Complexity improvement     (Sf)    -245915.469  bits   -213.8395 bits/instance
Mean absolute error                      0.2
Root mean squared error                  0.4472
Relative absolute error                 42.2929 %
Root relative squared error             91.9751 %
Coverage of cases (0.95 level)          80      %
Mean rel. region size (0.95 level)      50      %
Total Number of Instances             1150

=== Confusion Matrix ===
   a   b   <-- classified as
 359  82 |   a = 1
 148 561 |   b = 0

=== Error on test split ===

Correctly Classified Instances        2733               79.1944 %
Incorrectly Classified Instances       718               20.8056 %
Kappa statistic                          0.5697
K&B Relative Info Score             193021.083  %
K&B Information Score                 1854.1606 bits      0.5373 bits/instance
Class complexity | order 0            3347.7961 bits      0.9701 bits/instance
Class complexity | scheme           771132      bits    223.4518 bits/instance
Complexity improvement     (Sf)    -767784.2039 bits   -222.4817 bits/instance
Mean absolute error                      0.2081
Root mean squared error                  0.4561
Relative absolute error                 43.6936 %
Root relative squared error             93.1656 %
Coverage of cases (0.95 level)          79.1944 %
Mean rel. region size (0.95 level)      50      %
```

```
Total Number of Instances             3451

=== Confusion Matrix ===
    a    b   <-- classified as
 1051  321 |    a = 1
  397 1682 |    b = 0
-=================================
```

We observe OneR has a 79.2% accuracy in its prediction which is significantly higher than that of ZeroR.

Experiment - Varying the number of training samples

Here we experiment with the number of training samples. The experiment investigates what happens when training is a very low percent of the original data. We will use OneR for this experiment.

Hypothesis

Where the percent of samples for training becomes small we would expect the accuracy during model generation would be close to 100% (it is easy to create a model to predict the small number of training samples). But when the model is run against the testing data it would have low accuracy since the training data would not fully represent the original dataset.

As the number of training samples increases we expect the accuracy during model generation would flatten out (the model reaches its theoretical limit based on its capabilities and the complexity of the data). We would also expect the model's ability to predict the testing data to flatten out and furthermore would approach the accuracy seen during model generation. This would indicate a) the model has reached the limits of its capabilities and b) the training and test data are consistent with one-another.

Procedure

The data for this test is the spambase dataset which has been discretized with the "better encoding" switch.

The OneR classifier is repeatedly run while varying the size of the training dataset. This is varied using the -split-percentage switch, starting with 0.05% and increasing to 50%. An example of the command line is:

```
 java weka.classifiers.rules.OneR  -t proj1\01_discretized_encoding_better.arff -split-percentage 0.05 -i -k
```

Results:

The results of the experiment are shown in the table:

Training dataset size (percent of original dataset)	Instances in training dataset	Model accuracy during training	Instances in testing data	Model accuracy against testing data
0.05%	2	100	4599	29.4
0.1%	5	100	4596	70.6
0.2%	9	100	4592	76.7
0.5%	23	87	4578	74.1
1.0%	40	87	4555	75.7
2.0%	92	80.4	4509	75.4
5.0%	230	78.7	4371	76.1
10%	460	80.4	4141	79.3
20%	920	80.1	3681	79.2
50%	1847	80.3	1806	78.5

Conclusions

The OneR model achieve 75% accuracy against the testing data using just 0.2% of the training data. After that there is limited change in performance.   We suspect this is because the OneR model uses a single attribute in its assessment and even a small sample is enough to choose that attribute and, essentially, establish its behavior.

Experiment - N-fold cross-validation

Here we explore n-fold cross-validation implementation in Weka. In Weka n-fold cross-validation can be run as part of model generation.

Cross-validation as part of model generation works as follows: take (for example) weka.classifiers.rules.OneR -x 10 -d outmodel.xxx This does TWO things - first it creates a model based on the full dataset. This is the model that is written to outmodel.xxx. This model is NOT used as part of cross-validation. THEN cross-validation is run. cross-validation involves creating (in this case) 10 new models, using 9 folds for training and 1 fold for testing, iterating through the 1 testing fold. Statistical performance is captured for the 10 tests. The important point is the models used in cross-validation are temporary and only used to generate statistics about the process. They are not equivalent to, or used for, the model that is given to the user

If this is true, then the model generated is independent of the "n" in n-fold cross validation. An experiment will confirm this.

Hypothesis
The model generated by OneR is independent of the "n" in n-fold cross validation.

Procedure
Create training and testing data sets, starting from the original spambase which was discretized using "better" encoding. Here we specify 2 folds for training, one for testing.

```
java weka.filters.supervised.instance.StratifiedRemoveFolds -S 0 -N 3 -F 1  -c "last"  -i
proj1\01_discretized_encoding_better.arff  -o proj1\03b_training.arff
java weka.filters.supervised.instance.StratifiedRemoveFolds -S 0 -N 3 -F 1 -V  -c "last"  -i
proj1\01_discretized_encoding_better.arff  -o proj1\03b_testing.arff
```

Using the training data generate two OneR models with cross-validation enabled. The first has cross-validation n=2 and the other has n=20.

```
 java weka.classifiers.rules.OneR -x 2 -i -k -t proj1\03b_training.arff  -d
proj1\03b_OneR_model_w_crossvalidate_2.xxx
 java weka.classifiers.rules.OneR -x 20 -i -k -t proj1\03b_training.arff  -d
proj1\03b_OneR_model_w_crossvalidate_20.xxx
```

The run with cross-validation n=2 outputs the following:

```
+=================output======================

=== Stratified cross-validation ===
Correctly Classified Instances        1194               77.8357 %
Incorrectly Classified Instances       340               22.1643 %
Kappa statistic                          0.5339
K&B Relative Info Score              80839.2032 %
K&B Information Score                  781.993  bits      0.5098 bits/instance
Class complexity | order 0            1484.2686 bits      0.9676 bits/instance
Class complexity | scheme          365160      bits    238.0443 bits/instance
Complexity improvement     (Sf)    -363675.7314 bits   -237.0767 bits/instance
Mean absolute error                 0.2216
Root mean squared error                0.4708
Relative absolute error             46.3927 %
Root relative squared error           96.3308 %
Coverage of cases (0.95 level)         77.8357 %
Mean rel. region size (0.95 level)      50      %
Total Number of Instances            1534
-=============================================
```

The run with cross-validation n=20 outputs the following:

```
+=================output======================

=== Stratified cross-validation ===

Correctly Classified Instances        1172              76.4016 %
Incorrectly Classified Instances       362              23.5984 %
Kappa statistic                       0.5005
K&B Relative Info Score             76112.8878 %
K&B Information Score                736.5101 bits      0.4801 bits/instance
Class complexity | order 0          1484.2762 bits      0.9676 bits/instance
Class complexity | scheme          388788      bits    253.4472 bits/instance
Complexity improvement     (Sf)    -387303.7238 bits   -252.4796 bits/instance
Mean absolute error                 0.236
Root mean squared error                0.4858
Relative absolute error             49.3972 %
Root relative squared error           99.3982 %
Coverage of cases (0.95 level)         76.4016 %
Mean rel. region size (0.95 level)      50      %
Total Number of Instances            1534
-=============================================
```

It is observed the cross-validation is different when it is run with N=2 and N=20. This is expected because in the first case 2 models are generated using data split into 50%/50% for training and testing. In the second case 20 models are generated using 95%5% for training/testing.

Next we test the two models that have generated using the testing data to compare their performance. The -l switch specifies the model to be used (from above either that generated with the n=2 cross-validation or the n=20 cross-validation). The -T switch specifies the testing data.

```
java weka.classifiers.rules.OneR -no-cv -l proj1\03b_OneR_model_w_crossvalidate_2.xxx  -T proj1\03b_testing.arff
 java weka.classifiers.rules.OneR -no-cv -l proj1\03b_OneR_model_w_crossvalidate_20.xxx  -T proj1\03b_testing.arff
```

Results:

The first model (generated with N=2 cross-validation) results in:

```
+=================output======================
=== Error on test data ===
Correctly Classified Instances       2418              78.8393 %
Incorrectly Classified Instances      649              21.1607 %
Kappa statistic                     0.5377
Mean absolute error                    0.2116
Root mean squared error                  0.46
Coverage of cases (0.95 level)          78.8393 %
Total Number of Instances            3067
-===============================================
```

The second model (generated with N=20 cross validation) results in:

```
+=================output=======================
=== Error on test data ===
Correctly Classified Instances       2418              78.8393 %
Incorrectly Classified Instances      649              21.1607 %
Kappa statistic                     0.5377
Mean absolute error                    0.2116
Root mean squared error                  0.46
Coverage of cases (0.95 level)          78.8393 %
Total Number of Instances            3067
-===============================================
```

As can be seen above the results are identical. The results when using the model generated when N=2 is used for cross-validation are identical to those using the model generated when N=20 is used for cross-validation.

Conclusion

The results are consistent with our prediction. We conclude that cross-validation does not affect generation of the model, rather is a separate mechanism to evaluate the likelihood the model that has been generated is a reasonable representation of the data.