****

Computer Science Faculty

**Final Project**

**In**

**Knowledge Discovery**

****

**Professor: Dr. Eric Harley**

By: S.M.Reza Dibaj

In this project, I am going to use different aspects of WEKA by working on some sample .arff files. I hope the result can be a simple Teach Yourself document, in order to get the most out of WEKA in a visual format and using the least possible time.

Let’s first know a little bit more about WEKA and its file format.

 **Building the dataset for WEKA**

To load data into WEKA, we have to put it into a format that will be understood. WEKA's preferred method for loading data is in the Attribute-Relation File Format (ARFF), where you can define the type of data being loaded, then supply the data itself. In the file, you define each column and what each column contains. In the case of the regression model, you are limited to a NUMERIC or a DATE column. Finally, you supply each row of data in a comma-delimited format. The ARFF file we'll be using with WEKA appears below. Notice in the rows of data that we've left out my house. Since we are creating the model, we cannot input my house into it since the selling price is unknown.

 **WEKA file format**

@RELATION house

@ATTRIBUTE houseSize NUMERIC

@ATTRIBUTE lotSize NUMERIC

@ATTRIBUTE bedrooms NUMERIC

@ATTRIBUTE granite NUMERIC

@ATTRIBUTE bathroom NUMERIC

@ATTRIBUTE sellingPrice NUMERIC

@DATA

3529,9191,6,0,0,205000

3247,10061,5,1,1,224900

4032,10150,5,0,1,197900

2397,14156,4,1,0,189900

2200,9600,4,0,1,195000

3536,19994,6,1,1,325000

2983,9365,5,0,1,230000

In the first step, I am going to open a car.arff file.

At the outset, let me explain a little about this specific dataset:

Car Evaluation Database was derived from a simple hierarchical decision model originally developed for the demonstration of DEX (M. Bohanec, V. Rajkovic: Expert system for decision making. Sistemica 1(1), pp. 145-157, 1990.). The model evaluates cars according to the following concept structure:

CAR car acceptability

. PRICE overall price

. . buying buying price

. . maint price of the maintenance

. TECH technical characteristics

. . COMFORT comfort

. . . doors number of doors

. . . persons capacity in terms of persons to carry

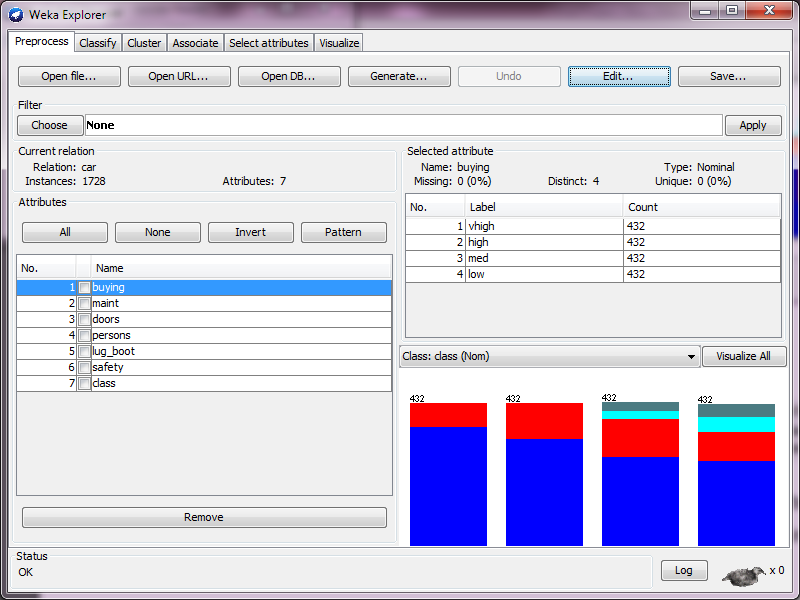
. . . lug\_boot the size of luggage boot

. . safety estimated safety of the car

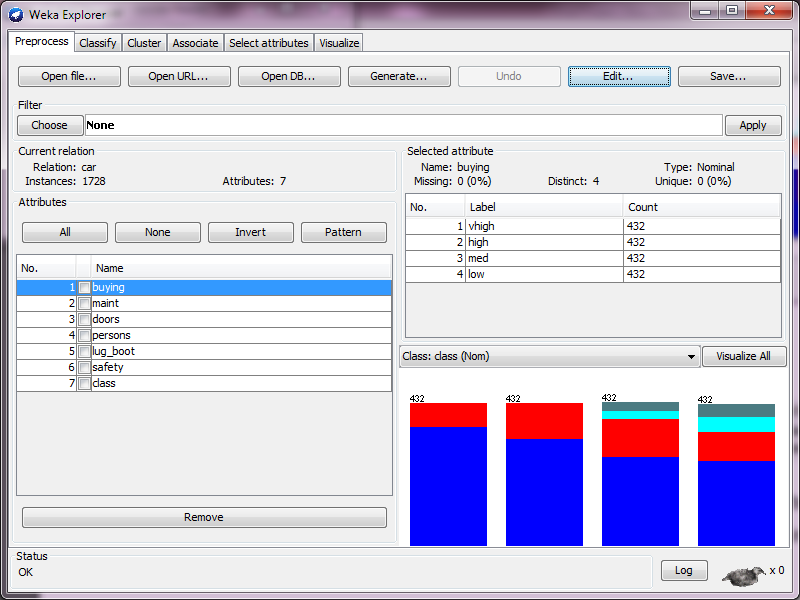
Input attributes are printed in lowercase. Besides the target concept (CAR), the model includes three intermediate concepts:

PRICE, TECH, COMFORT. Every concept is in the original model related to its lower level descendants by a set of examples. The Car Evaluation Database contains examples with the structural information removed, i.e., directly relates CAR to the six input attributes: buying, maint, doors, persons, lug\_boot, safety.

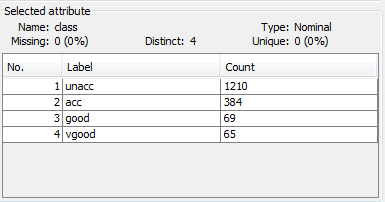
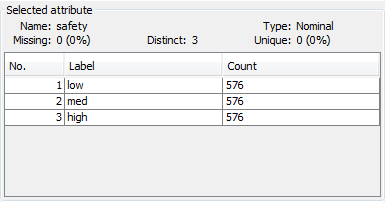
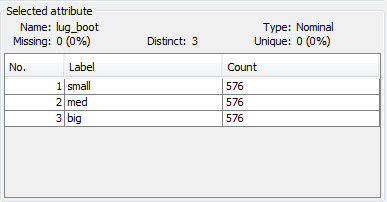
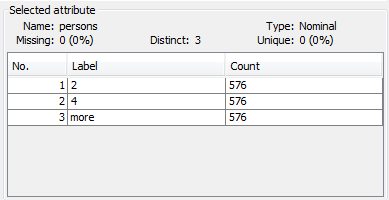
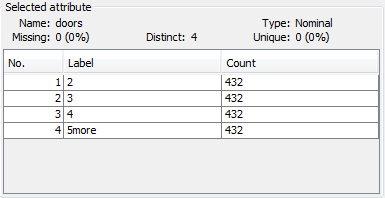
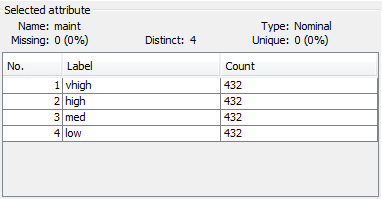
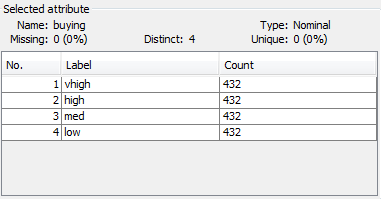
Because of known underlying concept structure, this database may be particularly useful for testing constructive induction and structure discovery methods, and this is the main reason that I used this dataset as an initiation for my project.



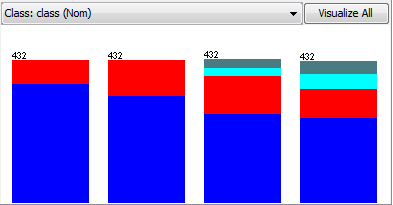
First, let’s load this dataset:

In the Current relation pane, we can see that there are 1728 instances in our dataset, which include 7 different attributes.

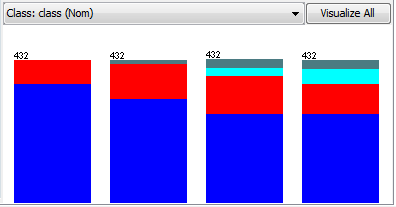
By selecting each attribute in the Attributes pane, we may see the relevant information regarding to that in the Selected attributes pane:

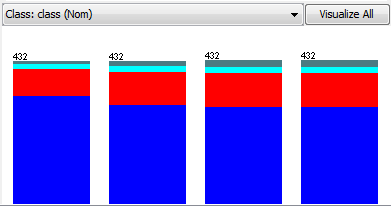


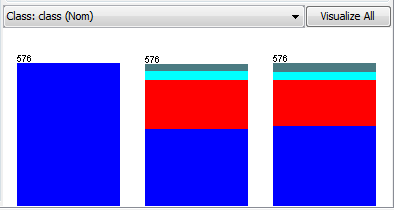
And in the lower-right pane we may see the distribution of a target attribute for the value of the selected attribute.



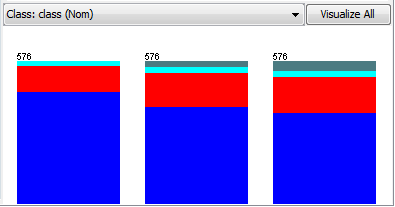
This is for buying attribute:

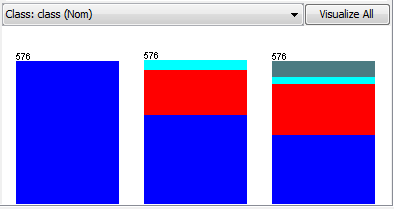
This is for maint attribute:

This is for doors attribute:

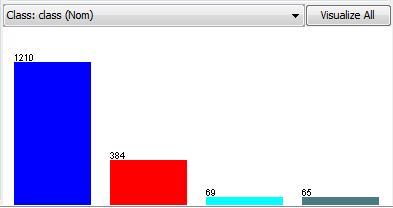


This is for persons attribute:

This is for lug\_boot attribute:

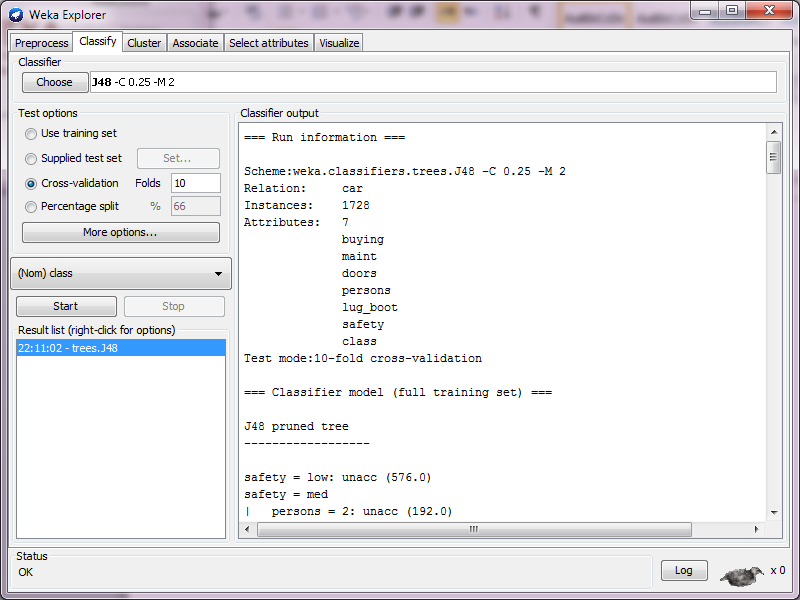
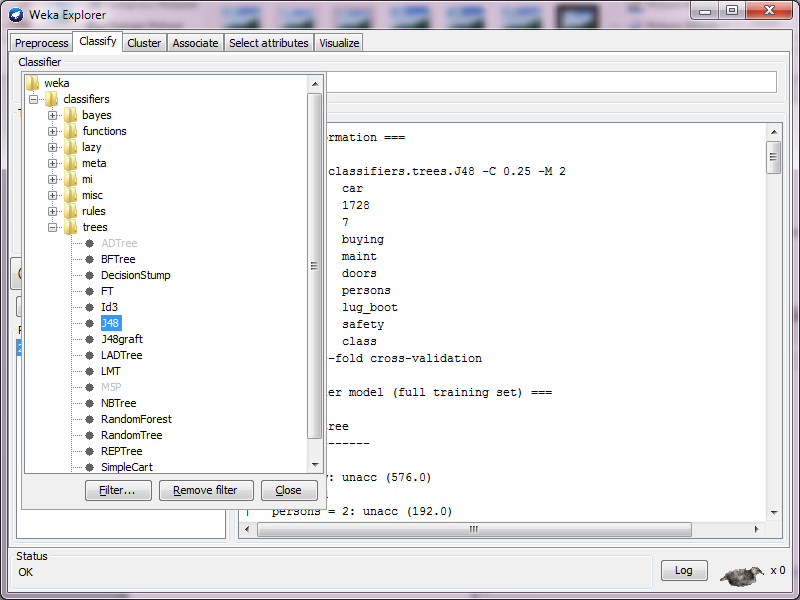


This is for safety attribute:



This is for class attribute:

In this step, let’s go to the Classify tab. Then in that screen we choose the J48 classifier which can be found under the tree group.



After pressing the Start button, by clicking in the right panel and selecting the result, we may copy and paste them as follows (with a gray background):

=== Run information ===

Scheme:weka.classifiers.trees.J48 -C 0.25 -M 2

Relation: car

Instances: 1728

Attributes: 7

buying

maint

doors

persons

lug\_boot

safety

class

Test mode:10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree

------------------

safety = low: unacc (576.0)

safety = med

| persons = 2: unacc (192.0)

| persons = 4

| | buying = vhigh

| | | maint = vhigh: unacc (12.0)

| | | maint = high: unacc (12.0)

| | | maint = med

.

.

.

.

| | | | lug\_boot = small: good (4.0/1.0)

| | | | lug\_boot = med: vgood (4.0/1.0)

| | | | lug\_boot = big: vgood (4.0)

Number of Leaves : 131

Size of the tree : 182

Time taken to build model: 0.11 seconds

=== Stratified cross-validation ===

=== Summary ===

Correctly Classified Instances 1596 92.3611 %

Incorrectly Classified Instances 132 7.6389 %

Kappa statistic 0.8343

Mean absolute error 0.0421

Root mean squared error 0.1718

Relative absolute error 18.3833 %

Root relative squared error 50.8176 %

Total Number of Instances 1728

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure ROC Area Class

0.962 0.064 0.972 0.962 0.967 0.983 unacc

0.867 0.047 0.841 0.867 0.854 0.962 acc

0.609 0.011 0.689 0.609 0.646 0.918 good

0.877 0.01 0.77 0.877 0.82 0.995 vgood

Weighted Avg. 0.924 0.056 0.924 0.924 0.924 0.976

=== Confusion Matrix ===

a b c d <-- classified as

1164 43 3 0 | a = unacc

33 333 11 7 | b = acc

0 17 42 10 | c = good

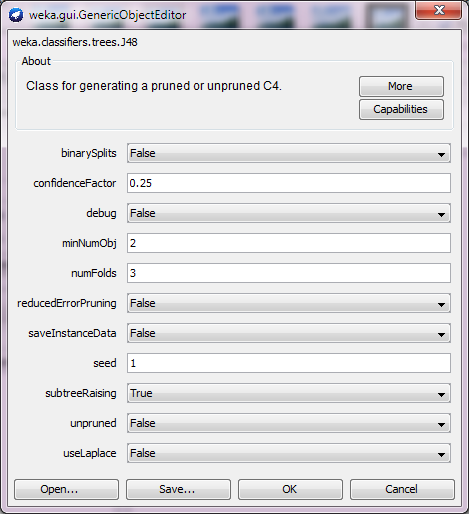
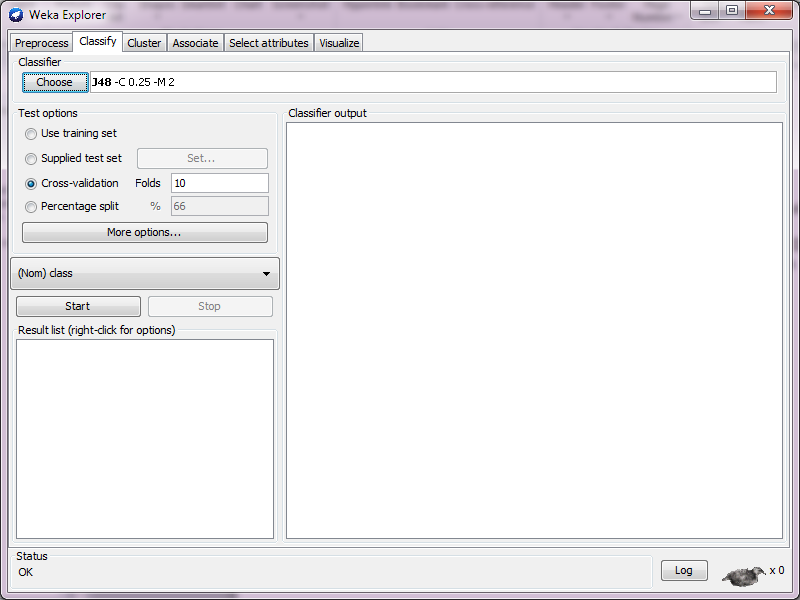
0 3 5 57 | d = vgood

We may see the overall accuracy from the following line:

Correctly Classified Instances 1596 92.3611 %

By selecting J48, in the Classifier output we can see a representation of a tree. We also can see the number of leaves as well as nodes there.

Furthermore, down in the bottom we may see the confusion matrix. Diagonal numbers are the one which correctly classified. Everything off this diagonal can be countered as mis-classification.



By clicking in the J48 box we can open a parameter window. For instance, one of the parameters is **unpruned tree** which is false by default. If we just change it to True and re-run the experiment we can see a higher accuracy, but we sacrifice generality by having this “**over-fitting**” setting here.

Also there is a parameter called: “minNumObj” which stands for the minimum number of instances per leaf. Let’s change that value from 2 to 20 and re-run the experiment.

The number of leaves and size of the tree change accordingly:

Number of Leaves : 15

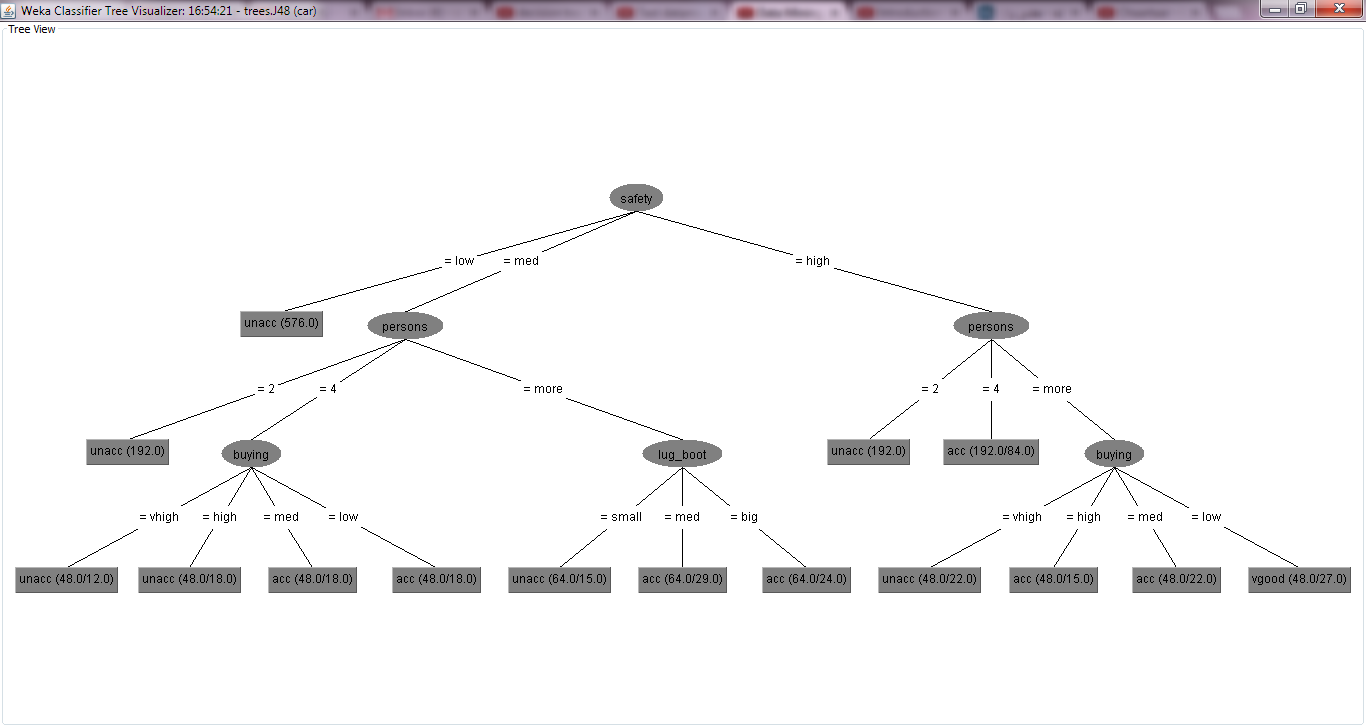
Size of the tree : 21

Before it was as follows:

Number of Leaves : 131

Size of the tree : 182

We will get to this option later in more details, but just for now let’s right click on the Result list and choose the Visualize tree. Then after the tree view appears, let’s right click in the open area and choose fit to screen. The result is something like the following picture:

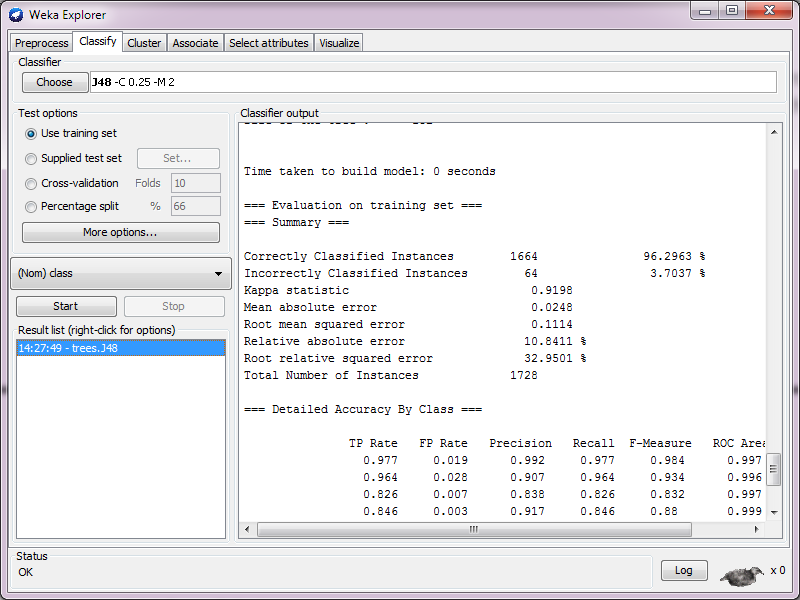


For example this shows we have 48 instances in here and 22 of them are not correctly classified

Also there is a more button in the **Classifier Option Editor** which gives you more info about that classifier and its options.

This name J48 has come from a famous system which was C4.8 which was invented in 1996. The developers of WEKA changed that C4.8 compatible with their needs and as they have implemented it with Java, they changed the name into J48.

 Note that in the default form, under the Test options pane, Cross-validation is selected.

With this option, the instances are going to split into 10 folds. Then the first 9 groups will be examined as a training set and the 10th group will be tested by the pattern which was gained through the first 9 groups. Then this routine will be done 9 more times and each time we choose 9 groups and a different group for the testing phase.

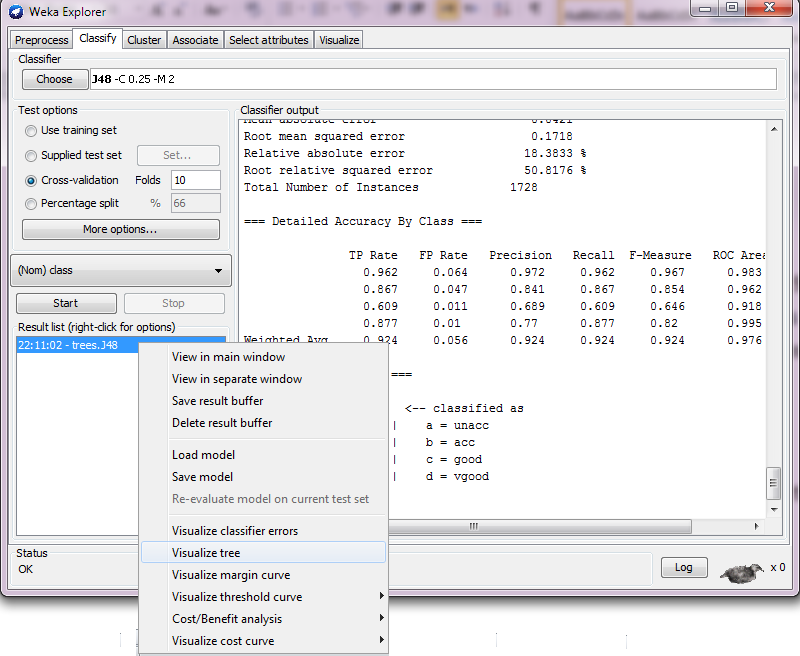
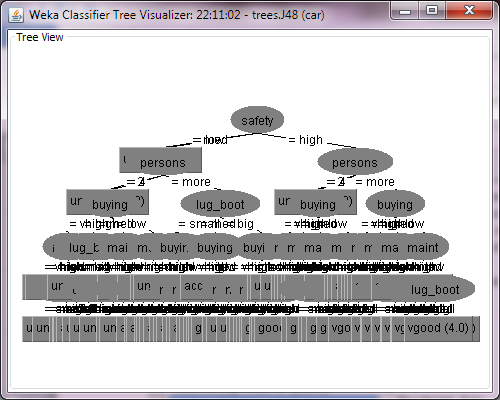
But if we change the Test option into Use training set it means that we use all our data to come up with a model and for sure that model has less deviation, but there is a doubt for next experiments to get the same accuracy. The result for that Use training set is as follows:

The accuracy is improved as you may see:

Correctly Classified Instances 1664 96.2963 %

But that happened with the expense of losing the generality. We call this form “Overfitting”.

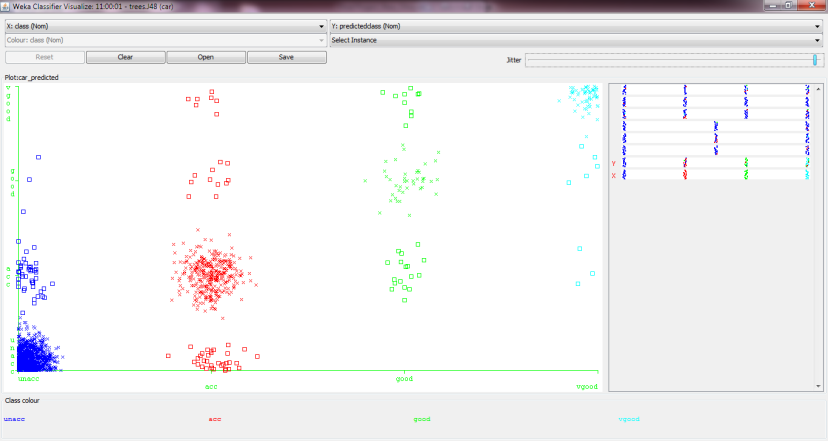
 Visualize tree



Now we are going to see the Visualize tree by right-clicking on the entry in the Result list.

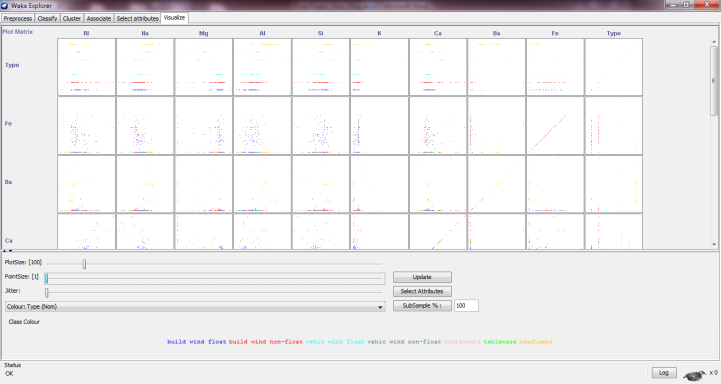
Normally we may have a better understanding of the pruning three by looking at the Tree Viualizer, but as our instances are so many in this very example, we may just see the higher level clearly and the buttomn of the graph is not very vivid.

We can see that there is a number shown next to each leaf of the tree. This is the number of instances which reached that leaf (for example you may see (4.0) beside vgood in the most right leaf in this chart.) Furthermore, it shows that they are all correctly classified, otherwise we may see something like vgood(4.0/1.0) which shows we have four instances in this leaf and one of them is not correctly classified.

 Visualize classified errors

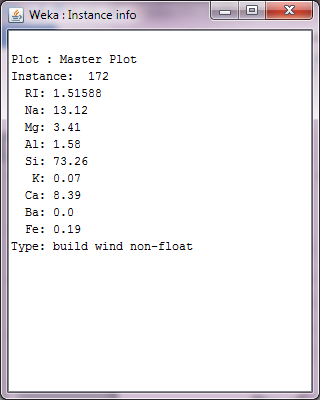
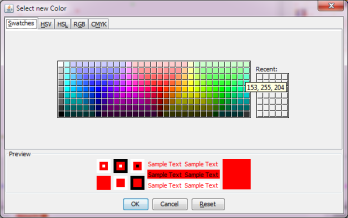
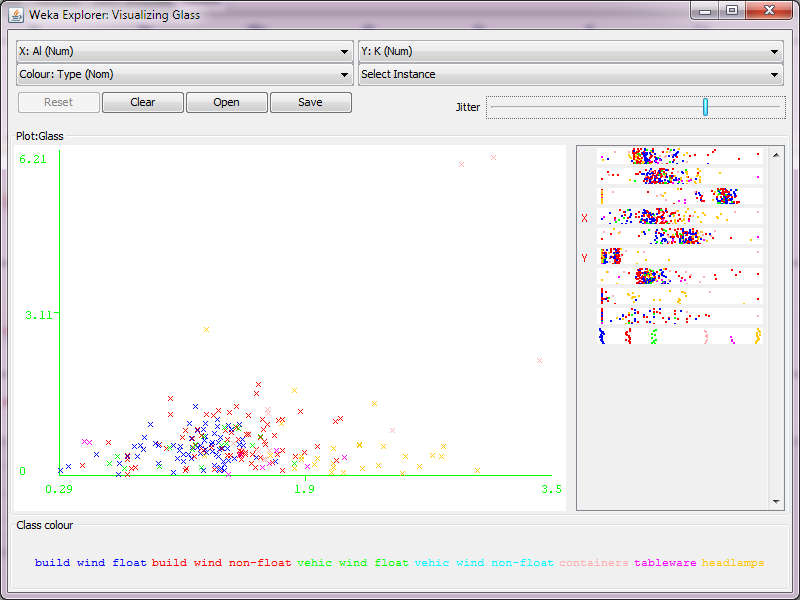
Now if we want to find a way of plotting one attribute against another and seeing the distribution of the target class, we can right-click on the result in the result list and choose Visualize classified errors.

The instances which are classified correctly are shown as X while those that are errors are plotted as boxes. Playing with jitter slider-bar can give us a better picture to clarify and distinguish the groups as well as the errors in our graph. We may click on each square and see what is its predicted class and what is its actual class which its classified in that by mistake.

 Changing the dataset to examine another aspect of WEKA

Now I change the dataset to glass.arff. After opening the file and using the J48 classifier, we may go to Visualize tab to find out the relation between each pair of attributes with the class.

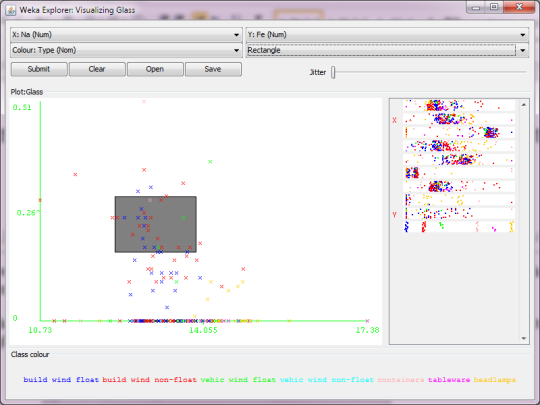
And if we click on any of these square frames, we may see a figure like this to see how each pair of attributes are related together to classify our classes. Furthermore, we can play with the jitter to make the classification more vivid.

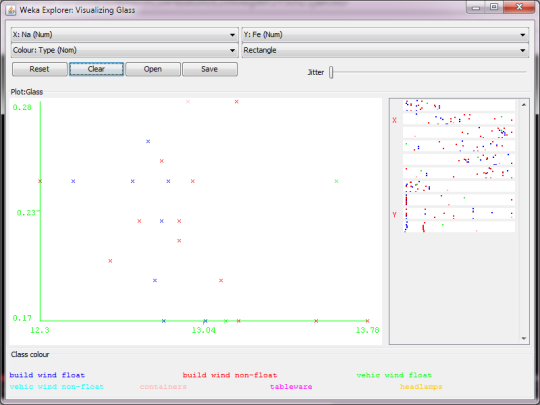


By clicking on each attribute name in the lower part of the screen, we can change its color accordingly.

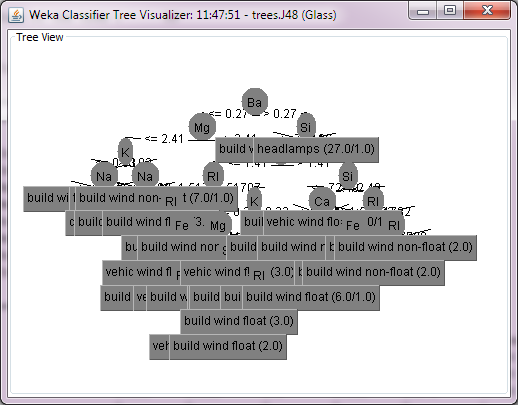
Also by clicking on any X on the screen we can see the relevant info about that in a window like this:

We can change X and Y axes on the top of the window, or we can change the X by clicking in the right frame or we can change the Y by right-click in the same place.

Also we can choose from the top of our window instead of Select Instance🡪 Rectangle and then we can draw a rectangle to include just those points. Then if I submit this rectangle, all these points are going to be included and the rest of the points are going to be excluded. The result is as follows after submit (also we can reset the result whenever we want.)



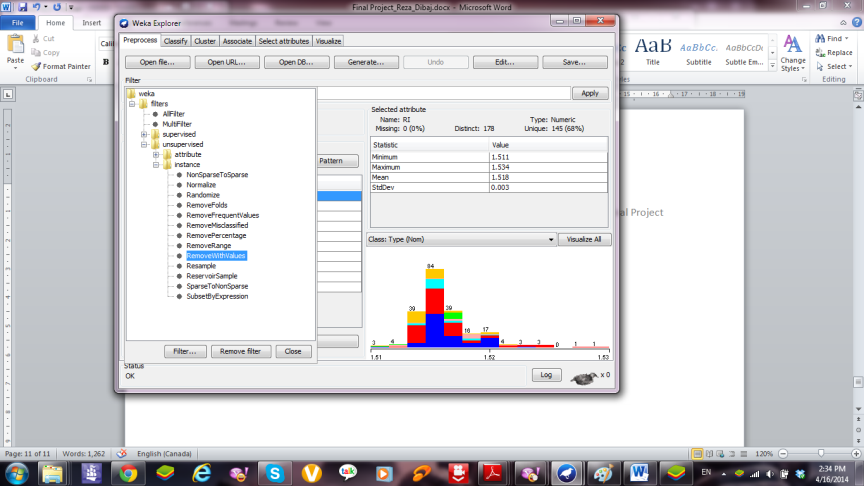
And here I show you one more Tree Visualizer graph to show how errors can be detected through the branches or leave signs.



27 instances that contains 1 error

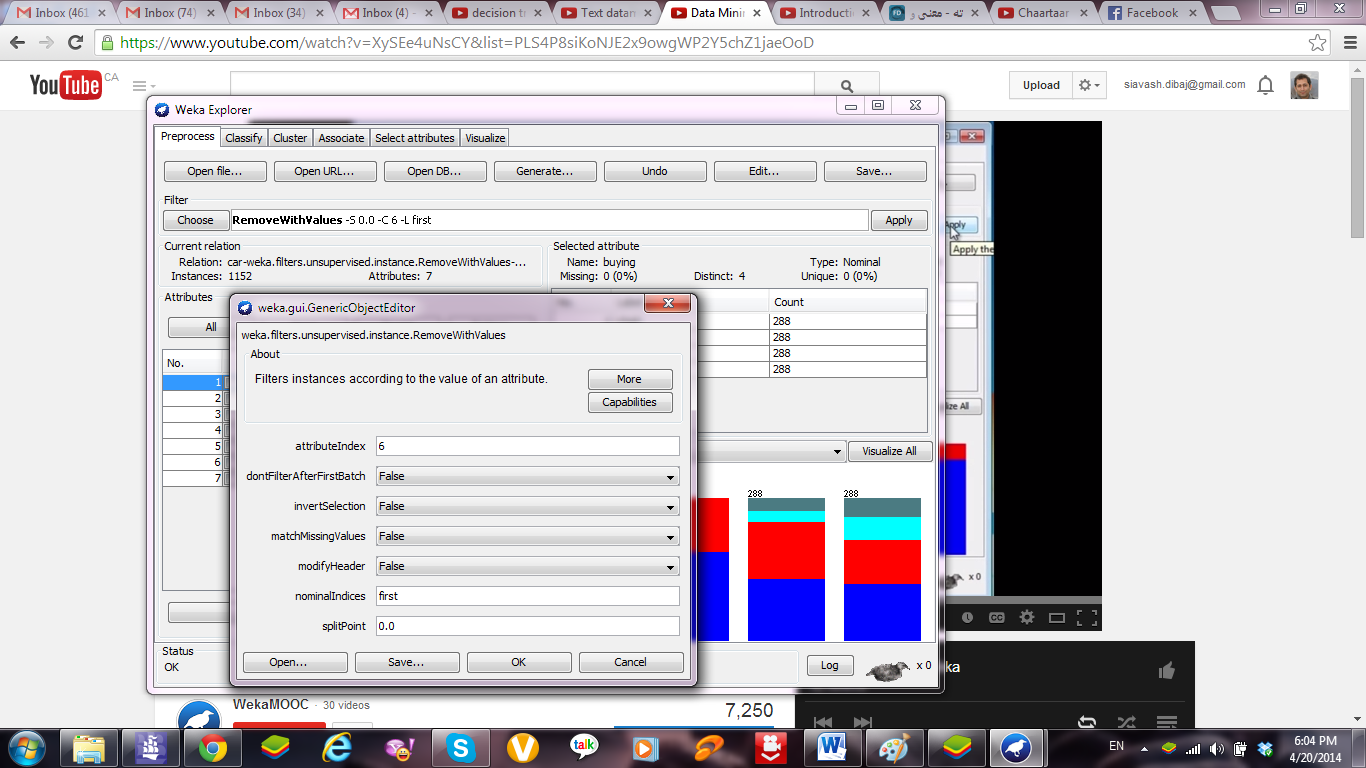
6 instances that contains 1 error

6 instances that contains 1 error

 Now let’s play with the filtering feature of the WEKA environment.

In the **Preprocess** tab, if we click on **Choose** under the **Filter** section. We may see two major categories: supervised and unsupervised. Supervised are the ones that that class-values for their operations. The unsupervised are more common. In this unsupervised filter section, we have attribute and instance categories. In the attribute section, there are so many filters that we can find and work with. Once we try “Remove” in the list. Suppose we want to remove the persons which is the fourth attribute in the list. Then we have to click in the box that we see “Remove” in it. Then in front of attributeIndices we simply type: 4. Then we press OK and after that we should press “Apply”. Then the relevant attribute (in here it is persons) will gone for the rest of experiment. We may press Undo button to bring it back if we want. We can continue our job without having persons attribute in our dataset. The easier way for this specific job is simply selecting that attribute and pressing the **Remove** button.

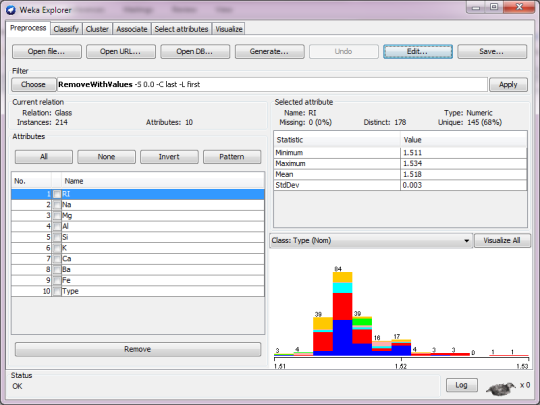
But what about the time that we want to remove part of our dataset which contains a specific value

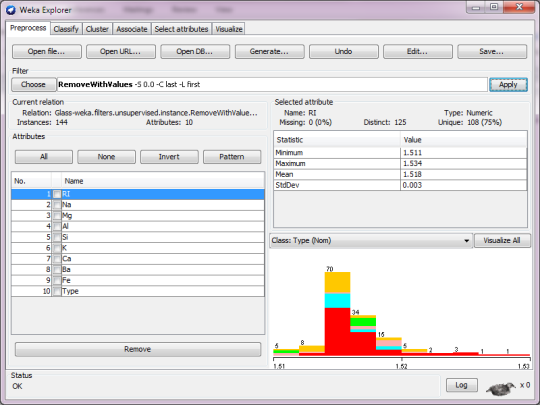
In the Preprocess tab, we should click on Choose. On the screen we may see a hierarchical filter selection. As it is one of the instance filters, we choose the filterweka.unsupervised.instance.RemoveWithValues. After clicking on that filter, on the main Preprocess panel we click on “RemoveWithValues” to get a panel where we can set the attribute and values to remove. Suppose in the car.arff sample we want to remove instances which their safety (fifth attribute) is low (first row sample.)

The window for this aim is as follows:

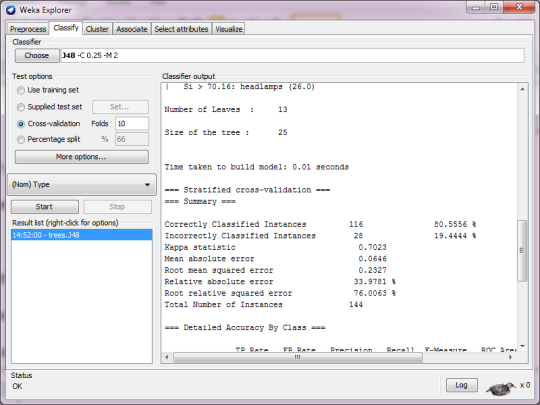
The number of instances will reduce from 1728 to 1152. By pressing the save button we replace the current dataset instead of the previous one.

 Now I am going to do another experiment on glass.arff that you can see in the following pictures.

 Before applying the filtering

 After applying the filtering

We can see by applying the filtering the number of our instances reduced from 214 to 144.



Then if we run the J48 classifier the result will be changed accordingly as follows:

It is obvious that the accuracy has changed a lot:

Correctly Classified Instances 116 80.5556 %

Incorrectly Classified Instances 28 19.4444 %

(Before its accuracy was as follow:

Correctly Classified Instances 143 66.8224 %)

 Working on different Classifiers

 Using J48 using Cross-validation Test-options

Now we are going to load car.arff file one more time. Then we go to a Classify tab. After that we click on Choose and select J48 from the trees section.

Then we click on the Start button with its default setting. It is Cross-validation with 10 folds. The accuracy of the result is:

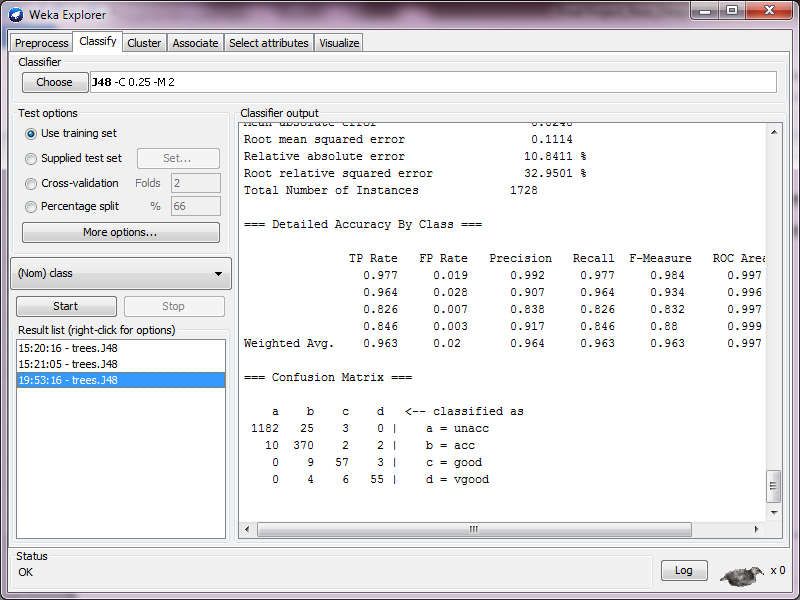
Correctly Classified Instances 1596 92.3611 %

Then we change the fold number into 5. In this condition the accuracy is:

Correctly Classified Instances 1582 91.5509 %

And if we test it with 2 folds the accuracy will be:

Correctly Classified Instances 1502 86.9213 %

There is a story behind 10 default folds that was mentioned in one of our lab instruction forms: “Cross-validation with n folds means that the training data is divided in to ‘n’ portions, and the algorithm is trained with n-1 parts as the training data, while one part is used to test the resulting model. This process is done in the n ways possible and the results are averaged. Theory and practice suggest that n = 10 folds is best in general.”

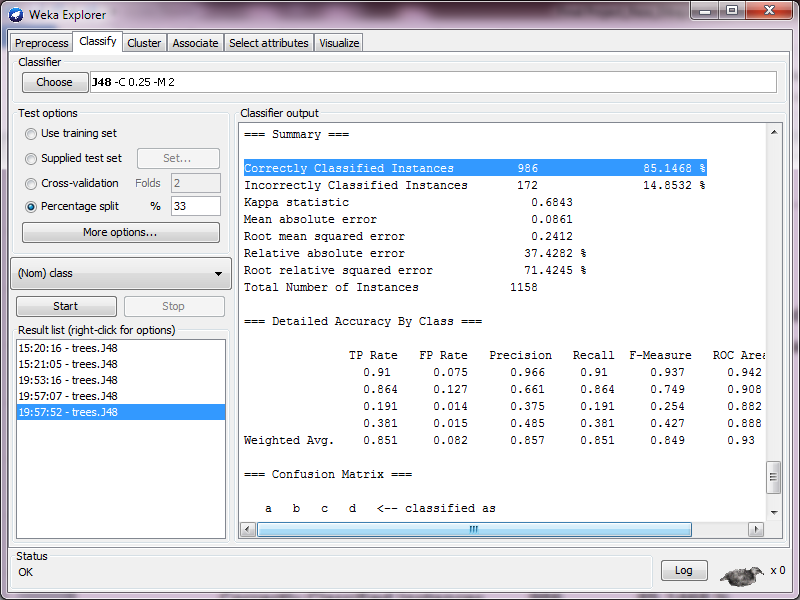
 Using J48 Using the Training Set Test-options

As we know testing by Using the training set prepares us the most optimistic accuracy evaluation. The reason is that we use the same data for training as well as testing.

The accuracy of Using the training set is as follows:

Correctly Classified Instances 1664 96.2963 %

 Using J48 Using the Percentage Split

Percentage split 66% means that two thirds of the data are used for training, and one third is held back for testing. Now let’s see the result of Percentage Split for 66% and 33% respectively:

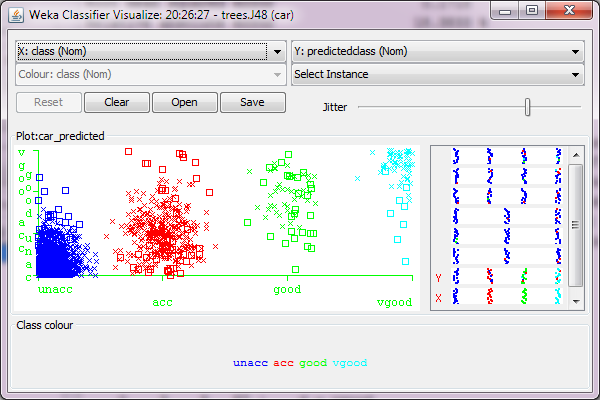
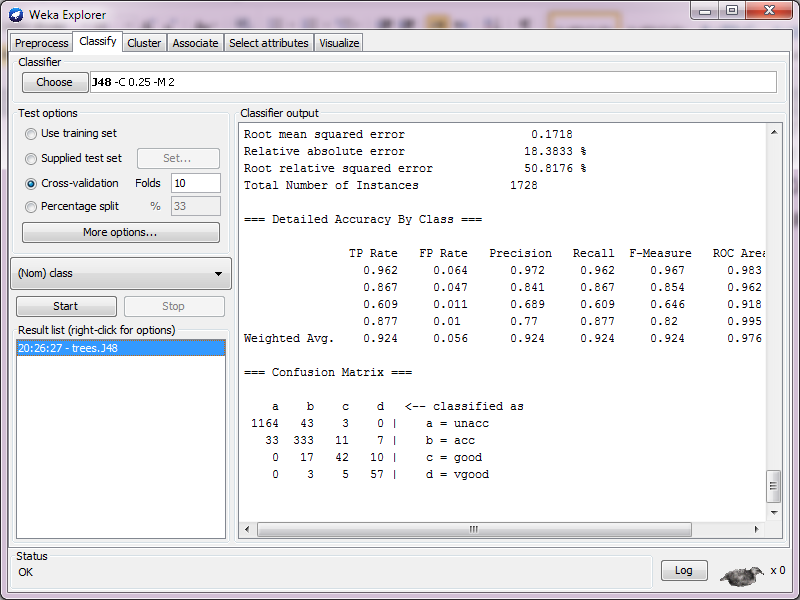
For 66% it is:

Correctly Classified Instances 535 90.9864 %

And for the 33% it is:

Correctly Classified Instances 986 85.1468 %

Long story short, we may perceive from the results that, the accuracy of “Using training set” is higher than the other models. But that accuracy is partly because of “Overfitting” and for any new data we probably do not hit the same accuracy. For the new data our expectation from Cross-validation model (Folds=10) is more realistic than the others. Several experiments and theory has proven that fact.



 Right click on the Result list and check the Visualize classifier errors

In this graph we can see Xs for the correctly classified instances, while we can see the squares for the errors.

 Naïve Bayes (NB)

Now let’s load car.arff in WEKA. Then we go to classify tab. Then we go to Classifier and choose the “Naïve Bayes”.

The number of instances is: 1728

One of the important part of the results is this:

=== Classifier model (full training set) ===

Naive Bayes Classifier

Class

Attribute unacc acc good vgood

(0.7) (0.22) (0.04) (0.04)

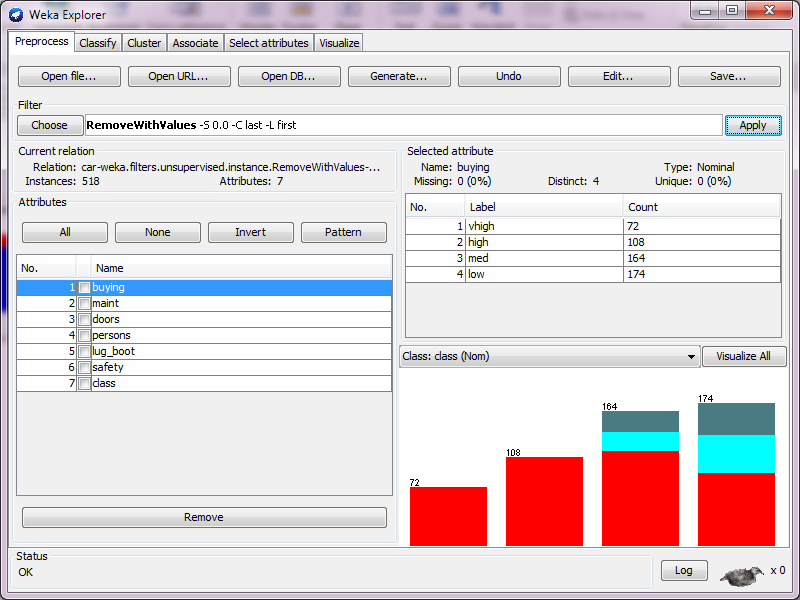
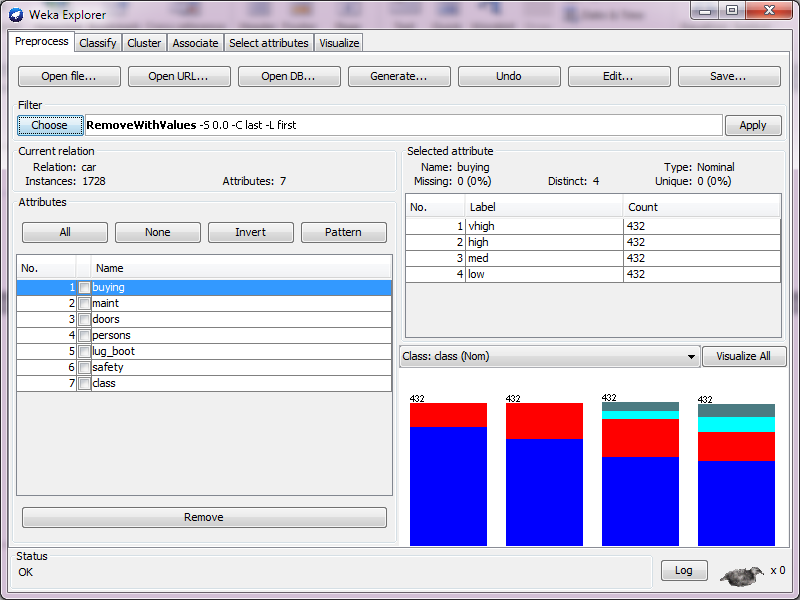
It is important to mention a point here. If there is any attribute with a zero value, in Naïve Bayes we add 1 to every single attribute in order to prevent from divided by zero error. We call this method: Laplace estimator.

 **Assigning a filter**

Now we get back to the Preprocess tab. Then we click on the Filter Choose. AS we may see the filter section is classified well. We can understand that the supervised part depends on the class value, while the unsupervised part is independent from the class value.

At this moment let’s try unsupervised🡪instance 🡪RemoveWithValues

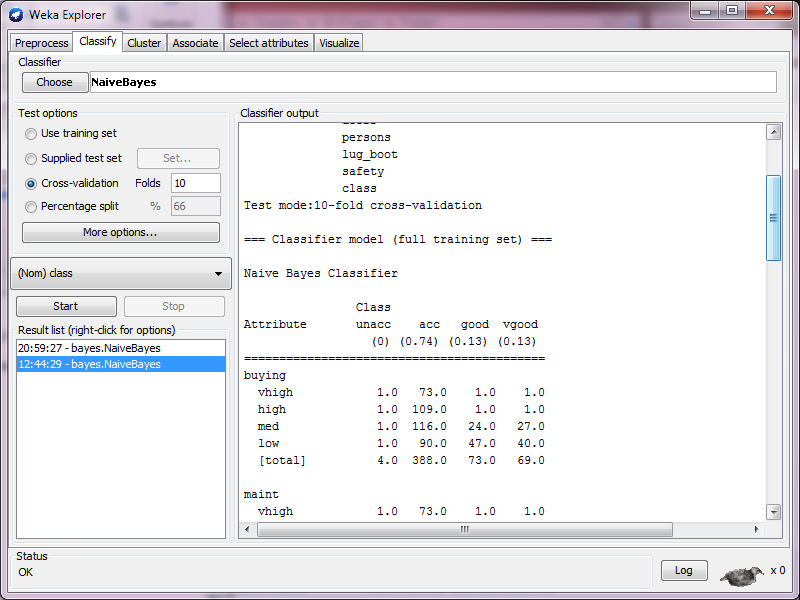
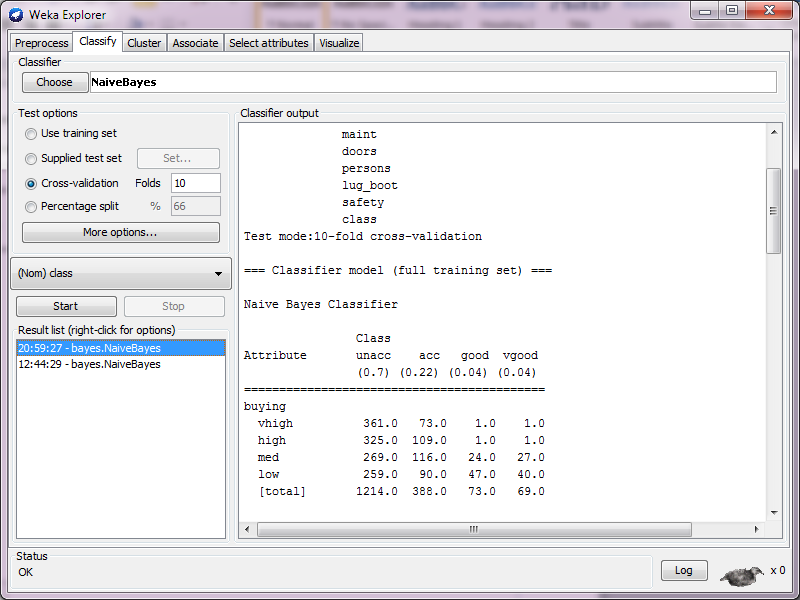
Then we click on the RemoveWithValues box, and use the attributeIndex and also nominalIndices to remove a part of our dataset. Then we should press Apply to make the change.



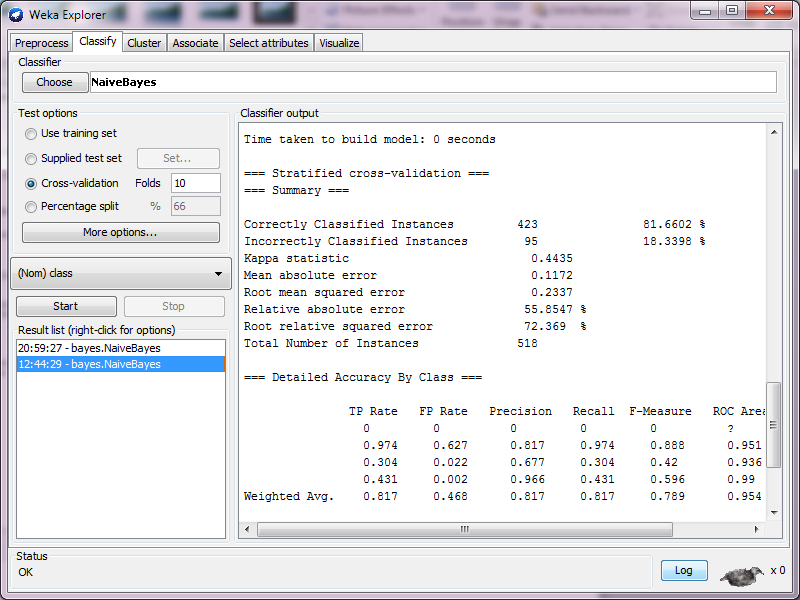
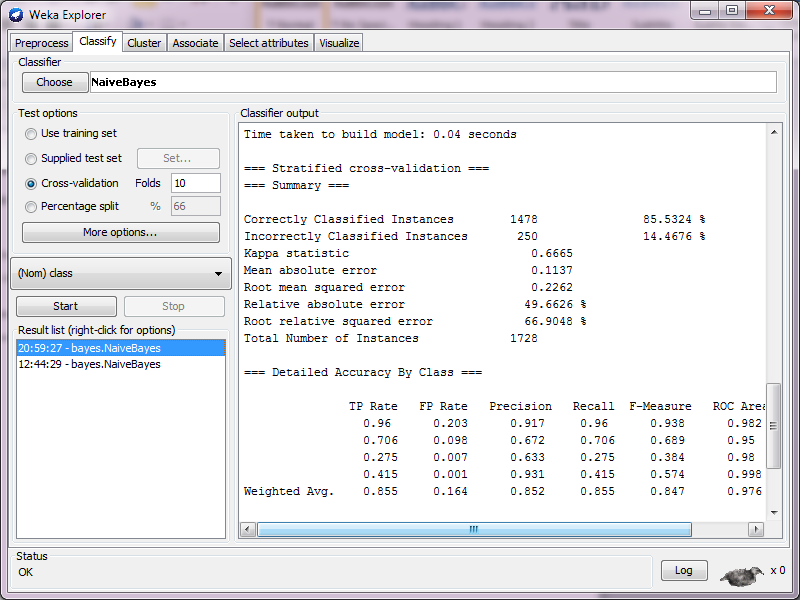
You can see by applying this filter, the number of instances reduced from 1728 to 518.

After that we are going to run Naïve Bayes classifier on this version of the data.

We can see the differences between two results. The left part belongs before applying the flter while the right one shows the result after applying the filter:



Here we may see the differences between two accuracies. The left part belongs before applying the flter while the right one shows the result after applying the filter:



 **Testing different classifiers (ZeroR, OneR, J48, and NB)**

Now we reload the car.arff file in order to have some new experiments on it. Then we test our data with all of these classifiers respectively (ZeroR, OneR, J48, and NB.) After finishing all the experiments we are going to compare our results and come up with a conclusion.

For ZeroR we have:

Correctly Classified Instances 1210 70.0231 %

For OneR we have:

Correctly Classified Instances 1210 70.0231 %

For J48 we have:

Correctly Classified Instances 1596 92.3611 %

For NB we have:

Correctly Classified Instances 1478 85.5324 %

Based on our results, we may see the accuracy of ZeroR and OneR are the same, but we know that in general OneR should bring us a better result. ZeroR just looks at the majority, while OneR deals with one attribute which can lead us better. That attribute should have the least number of errors.

J48 is a prune tree algorithm which most of the time works better among these methods, and in this very example it brought us a better result.

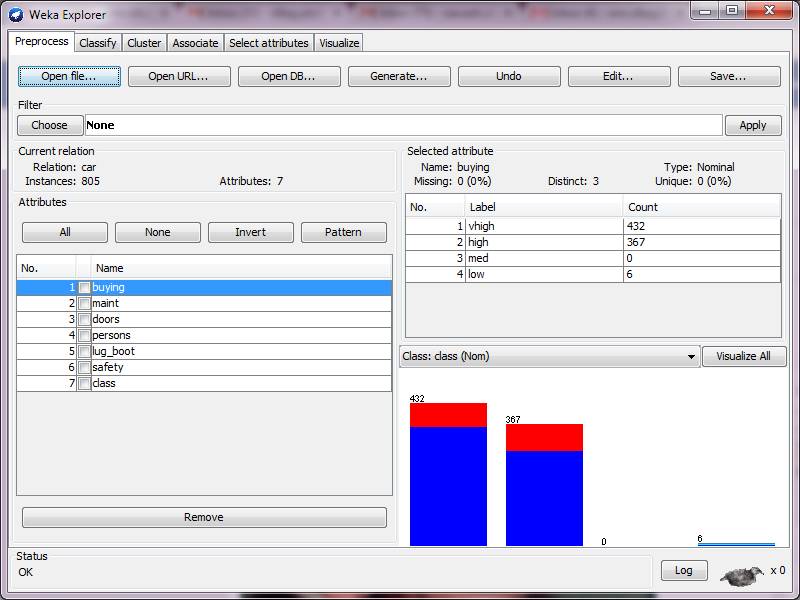
In Naïve Bayes we use conditional probability which brings us the second accurate result among all these four methods.

 **One Important Rule:**

When we are talking about Stratified cross-validation in our results, it means we propagate the results evenly among our folds. Suppose we have 70 NOs and 30 YESs in our class column. If we divide our samples in 10 folds, each of them should have 7 NOs and 3 YESs. In another words the result should be propagated evenly among the folds. When we are looking for the success rate, we should go to the lines after the following title:

=== Stratified cross-validation ===

=== Summary ===

 **Manipulating the dataset and rerunning the previous test, using different classifiers (ZeroR, OneR, J48, and NB)**

I deleted the rows after row-number 805, so we have 805 instances in our dataset

For ZeroR we have:

Correctly Classified Instances 648 80.4969 %

For OneR we have:

Correctly Classified Instances 646 80.2484 %

For J48 we have:

Correctly Classified Instances 763 94.7826 %

For NB we have:

Correctly Classified Instances 730 90.6832 %

 **Using AdaBoostM1 classifier**

Now we reload a fresh dataset by closing a current window and opening car.arff again. Then we go to classify tab. Then we choose AdaBoostM1 under Meta section. Based on our fourth lab instruction we know: “***AdaBoostM1 runs the simple Decision Stump learning method ten times. In each iteration Decision Stump produces a rule which is then applied to the instances in the training set. If the rule fails to classify an instance correctly, the weight of that instance is increased for the next iteration. Thus, in each subsequent iteration where a new Decision Stump model is made, more weight is given to previously misclassified instances. The resulting DS model for that iteration will work better for the previously misclassified instances. After the ten iterations, we have ten different DS models. When we want to make a prediction of the class for a new instance, we let the ten DS models vote. However, we do not give the votes all equal weight, but rather we give higher weight to models that had better success rate. The weights of the classifiers predicting a certain class are added, and the class with the highest total from these weighted votes is the winner.”***

Just to show a sample result, the first two classifiers are as follow:

Decision Stump

Classifications

persons = 2 : unacc

persons != 2 : unacc

persons is missing : unacc

Class distributions

persons = 2

unacc acc good vgood

1.0 0.0 0.0 0.0

persons != 2

unacc acc good vgood

0.5503472222222222 0.3333333333333333 0.059895833333333336 0.05642361111111111

persons is missing

unacc acc good vgood

0.7002314814814815 0.2222222222222222 0.03993055555555555 0.03761574074074074

**Weight: 0.85**

Decision Stump

Classifications

safety = low : unacc

safety != low : acc

safety is missing : unacc

Class distributions

safety = low

unacc acc good vgood

1.0 0.0 0.0 0.0

safety != low

unacc acc good vgood

0.34381778741865615 0.48643623480933545 0.08740651094230284 0.08233946682970561

safety is missing

unacc acc good vgood

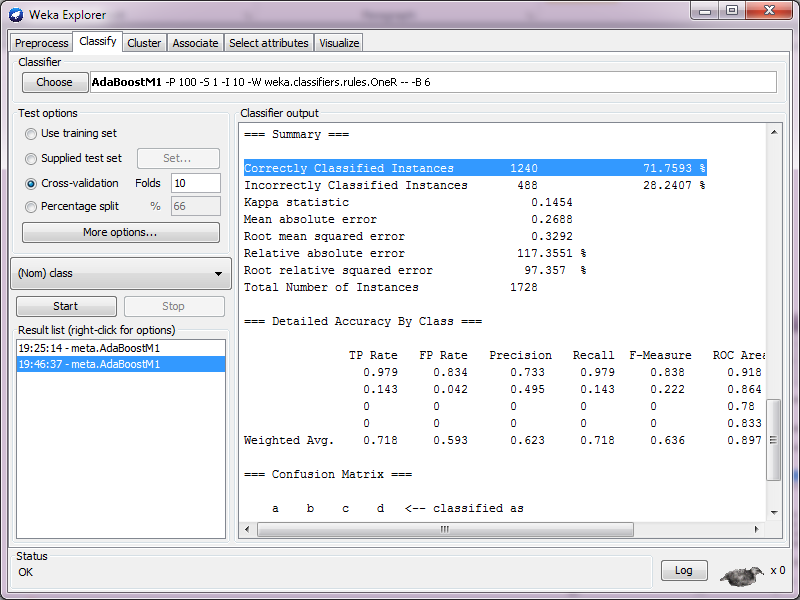
0.5000000000000011 0.37065637065636936 0.06660231660231665 0.06274131274131281

**Weight: 0.44**

And the overall success is:

Correctly Classified Instances 1210 70.0231 %

** Rerun the recent job but using AdaBoostM1 classifier boosting OneR**

Just let’s see the overall success in this case:

Correctly Classified Instances 1240 71.7593 %

We may perceive from these recent experiments the following points:

As we may see, in ZeroR because we have no rule, we just depend on the majority of the instances and we can see just by eliminating part of the dataset we come up with the different accuracy.

In the rest of results we may see there are lots of similarities between Decision Stump and OneR. These two are pretty close in the rule, but in this very example the Decision Stump model works better than OneR. (This is exactly opposite of what we had in our Lab 4 experiment)

Naïve Bayes works fine with those instances which have independent attributes, and based on our results in here, it really works fine and it shows the attributes are pretty independent from each other.

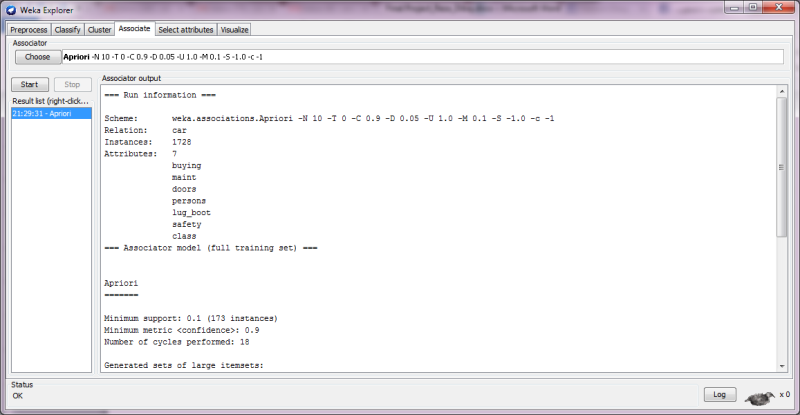
The AdaBoostM1 works pretty close to Decision Stump with its initiate setting and it works with high accuracy, but if we change its setting the way we did (OneR), we gain higher accuracy instead. (But this may reduce our accuracy, like what we experienced in Lab 4)

** Association rules:**

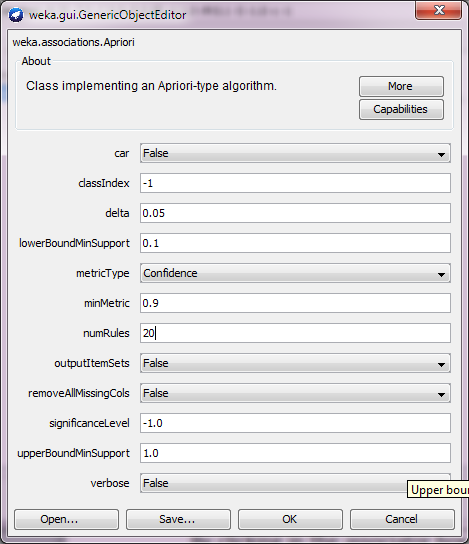
Before going to **Association** tab, let’s know a little more about **Association** **rules** and its usage. Association rule mining finds interesting association or correlation relationships among a large set of data items. With massive amounts of data continuously being collected and stored in databases, many industries are becoming interested in mining association rules from their databases. For example, the discovery of interesting association relationships among huge amounts of business transaction records can help catalog design, cross marketing, loss-leader analysis, and other business decision making processes.

A typical example of association rule mining is market basket analysis. This process analyzes customer-buying habits by finding associations between the different items that customer’s place in their “shopping baskets". The discovery of such associations can help retailers develop marketing strategies by gaining insight into which items are frequently purchased together by customers. For instance, if customers are buying milk, how likely are they to also buy bread (and what kind of bread) on the same trip to the supermarket? Such information can lead to increased sales.

The WEKA software efficiently produces association rules for the given data set. The Apriori algorithm is used as the foundation of the package. It gives all the item-sets and the subsequent frequent sets for the specified minimal support and confidence.

** Working with Association tab**

Let’s reload a fresh dataset by opening the car.arff file again. Then in Classifier I choose Naïve Bayes and run it. Then I go to **Association** tab. After pressing the **Start** button, we can see the top 10 association rules.



Best rules found:

1. persons=2 576 ==> class=unacc 576 conf:(1)

2. safety=low 576 ==> class=unacc 576 conf:(1)

3. persons=2 lug\_boot=small 192 ==> class=unacc 192 conf:(1)

4. persons=2 lug\_boot=med 192 ==> class=unacc 192 conf:(1)

5. persons=2 lug\_boot=big 192 ==> class=unacc 192 conf:(1)

6. persons=2 safety=low 192 ==> class=unacc 192 conf:(1)

7. persons=2 safety=med 192 ==> class=unacc 192 conf:(1)

8. persons=2 safety=high 192 ==> class=unacc 192 conf:(1)

9. persons=4 safety=low 192 ==> class=unacc 192 conf:(1)

10. persons=more safety=low 192 ==> class=unacc 192 conf:(1)

By clicking in the associator box, we may change the parameters. For instance, we can change the number of rules from 10 into 20.

By pressing Start, we receive different result in our Associator output window.

** Playing with the numerical attributes and class in WEKA**

Let’s reload a fresh dataset by opening the tae.arff file. The attributes of this dataset a numeric and the class is numeric as well. Before going any further step, let’s know a little bit more about this dataset. It can give us a better perspective to understand the rest of our analytical works better. The tile of this dataset is: Teaching Assistant Evaluation. The data consist of evaluations of teaching performance over three regular semesters and two summer semesters of 151 teaching assistant (TA) assignments at the Statistics Department of the University of Wisconsin-Madison. The scores were divided into 3 roughly equal-sized categories ("low", "medium", and "high") to form the class variable. The number of Instances is 151 and the number of Attributes is 6 (including the class attribute.)

Here is relevant info about the attributes:

1. Whether or not the TA is a native English speaker (binary)

1=English speaker, 2=non-English speaker

2. Course instructor (categorical, 25 categories)

3. Course (categorical, 26 categories)

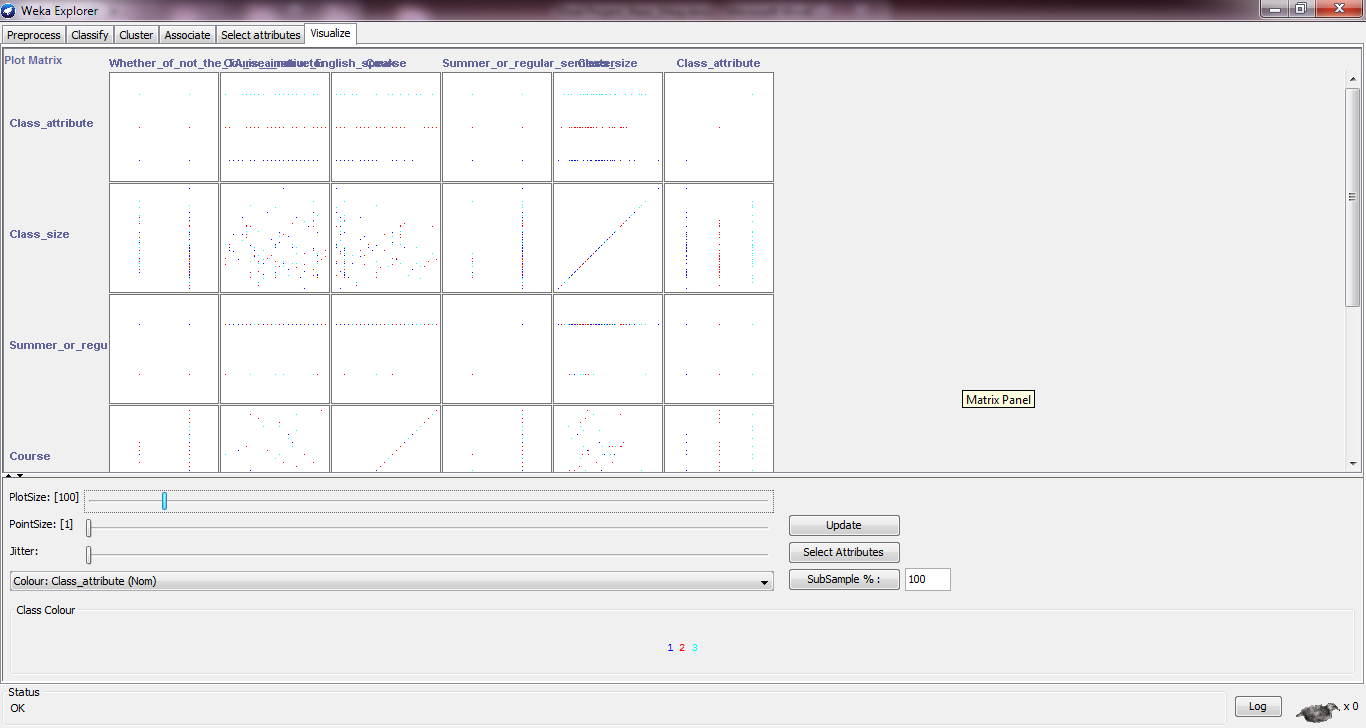
4. Summer or regular semester (binary) 1=Summer, 2=Regular

5. Class size (numerical)

6. Class attribute (categorical) 1=Low, 2=Medium, 3=High

As we can see the class-type is nominal which is quiet convenient for our experiments and the class-index is the last one.

First we load tae.arff file. Then we go to **Visualize** tab. And apparently those which are closer to an imaginary diagonal line are better fit linear fit.



** Playing with the simple Linear Regression method in WEKA**

In the following link I found a neat and clear explanation about Regression and there is a sample dataset that I enjoyed very much to use it as well.

<http://www.ibm.com/developerworks/library/os-weka1/>

** Regression**

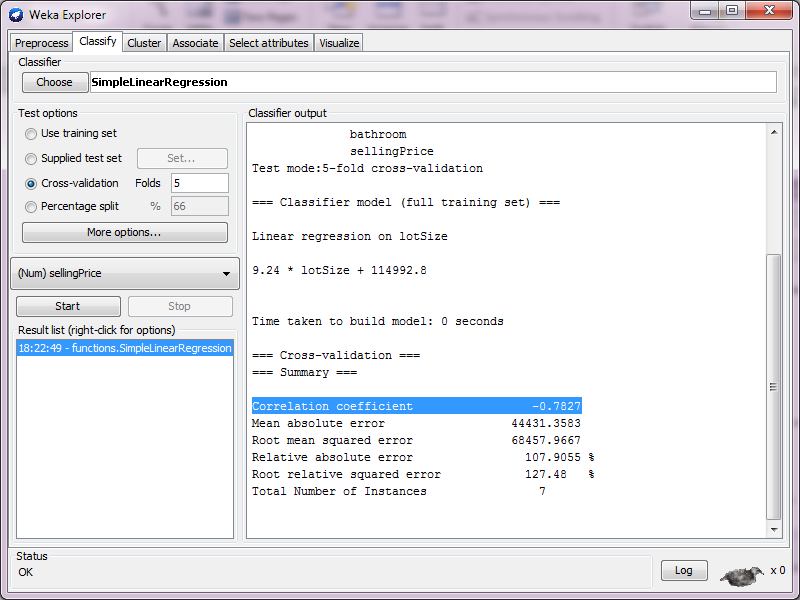
Regression is one of the most common technique and easiest way to use, but its functionality is not that much accurate and strong. We can simplify our model to have just one input variable and one output variable.Of course, it can get more complex than that, including dozens of input variables. In effect, regression models all fit the same general pattern. There are a number of independent variables, which, when taken together, produce a result — a dependent variable. The regression model is used to predict the result of an unknown dependent variable, given the values of the independent variables. We all have probably used or seen a regression model before, maybe even mentally creating a regression model. The example that immediately comes to mind is pricing a house. The price of the house (the dependent variable) is the result of many independent variables — the square footage of the house, the size of the lot, whether granite is in the kitchen, bathrooms are upgraded, etc. So, if you've ever bought a house or sold a house, you've likely created a regression model to price the house. You created the model based on other comparable houses in the neighborhood and what they sold for (the model,) then put the values of your own house into this model to produce an expected price.

Let's continue this example of a house price-based regression model, and create some real data to examine.

**House values for regression model**

| **House size (square feet)** | **Lot size** | **Bedrooms** | **Granite** | **Upgraded bathroom?** | **Selling price** |
| --- | --- | --- | --- | --- | --- |
| 3529 | 9191 | 6 | 0 | 0 | $205,000 |
| 3247 | 10061 | 5 | 1 | 1 | $224,900 |
| 4032 | 10150 | 5 | 0 | 1 | $197,900 |
| 2397 | 14156 | 4 | 1 | 0 | $189,900 |
| 2200 | 9600 | 4 | 0 | 1` | $195,000 |
| 3536 | 19994 | 6 | 1 | 1 | $325,000 |
| 2983 | 9365 | 5 | 0 | 1 | $230,000 |
|  |  |  |  |  |  |
| 3198 | 9669 | 5 | 1 | 1 | ???? |

The good news (or bad news, depending on your point of view) is that this little introduction to regression barely scratches the surface, and that scratch is really even barely noticeable. This example gets us be acquainted with the concept and suffice for our WEKA tests in this article.

** Loading houses.arff in WEKA**

Let’s reload a fresh dataset by opening the houses.arff file. Then we go to classify and choose SimpleLinearRegression.

The classifier model is:

Linear regression on lotSize

9.24 \* lotSize + 114992.8

Also we have the following info:

Correlation coefficient -0.7827

Mean absolute error 44431.3583

Root mean squared error 68457.9667

Relative absolute error 107.9055 %

Root relative squared error 127.48 %

Total Number of Instances 7

As we see the linear model was based on lotSize. Now we go to Preprocess tab and delete that attribute.

After doing so, we have in our the classifier model the following data:

Linear regression on bedrooms

36275 \* bedrooms + 42582.14

We can see instead of lotSize we have bedrooms. And also the summary is changed as follows:

Correlation coefficient -0.1595

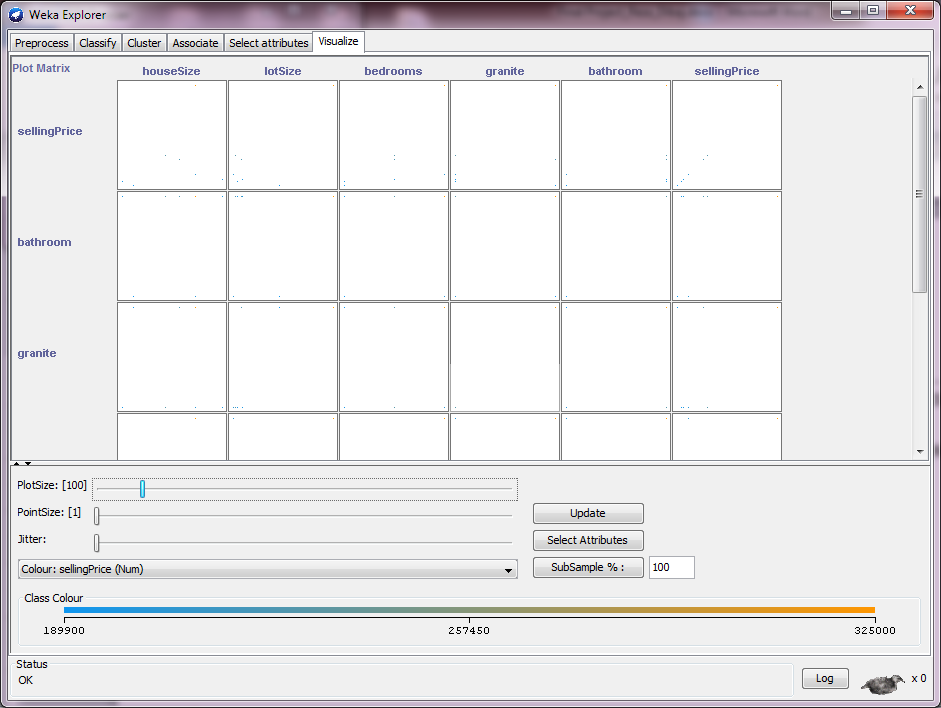
Mean absolute error 47786.1345

Root mean squared error 65412.3788

Relative absolute error 116.0528 %

Root relative squared error 121.8086 %

Total Number of Instances 7

If we repeat the same procedure the next important attribute would be granite. And after that bathroom, and then houseSize and finally sellingPrice.

Long story short, the attributes based on their importance are as follows respectively:

lotSize

bedrooms

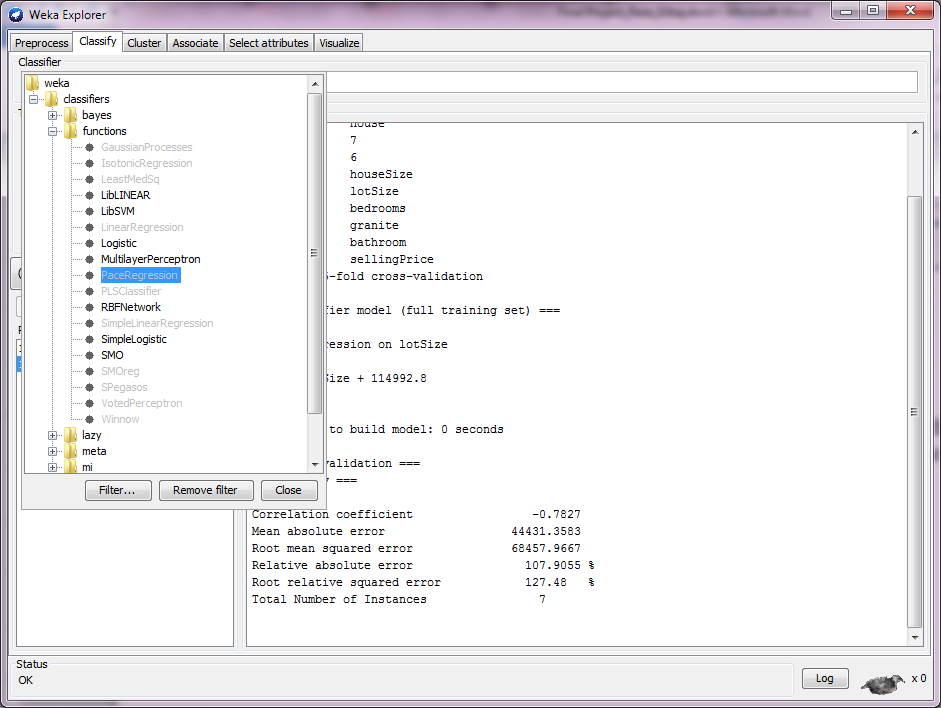
granite

bathroom

houseSize

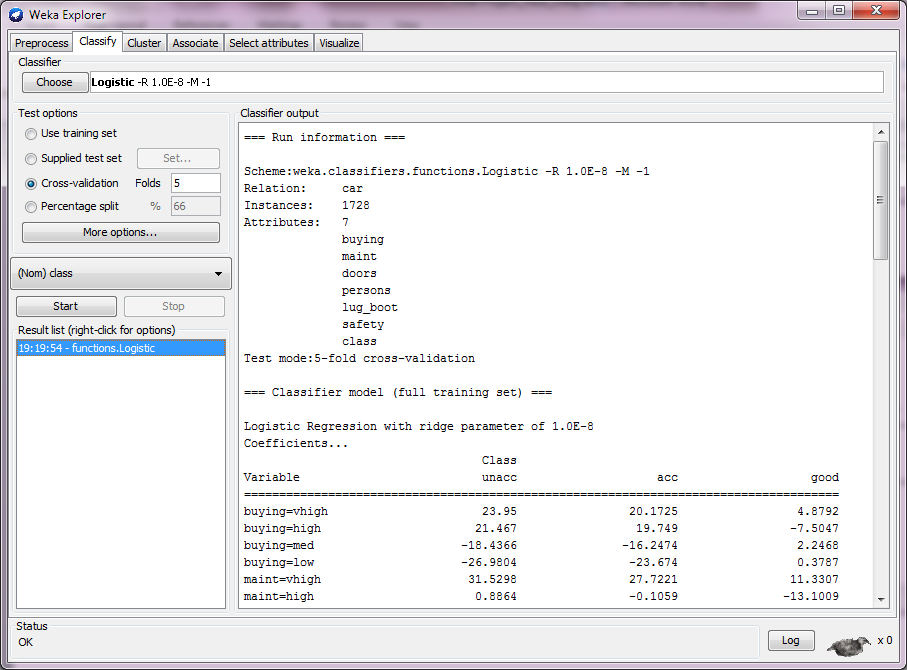
sellingPrice

If we go to the Visualize tab, we can see the graphs which prove this sequence that we have found in the classify tab.



** Loading car.arff in WEKA**

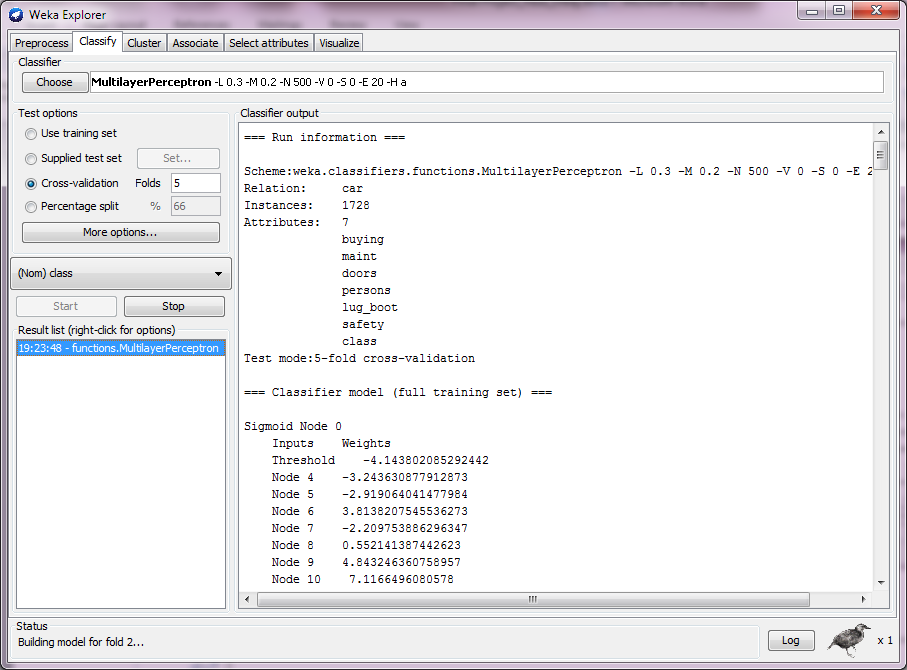
As it is clear the Simple Linear Regression is grayed out. The reason is the class in a kind of label (Nominal) and not a numeric type, but logistic regression works fine with these types of classes.



After choosing Logistic the result is as you may see in this picture:

The percent of correctly classified instances is:

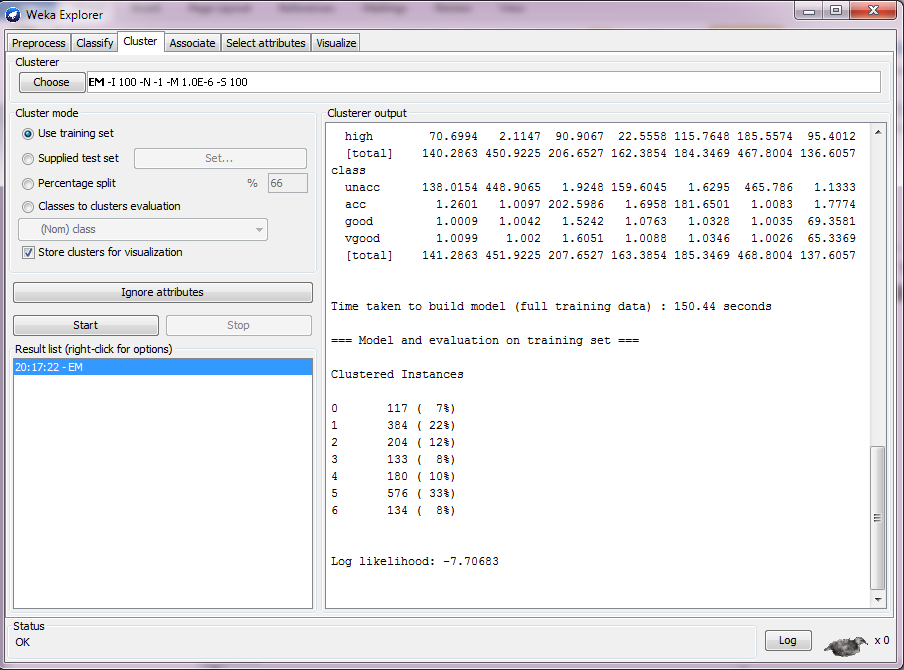
Correctly Classified Instances 1612 93.287 %

As it is visible in the previous picture, the **VotedPerceptron** is grayed out. The reason is that car.arff data is not linearly separable.

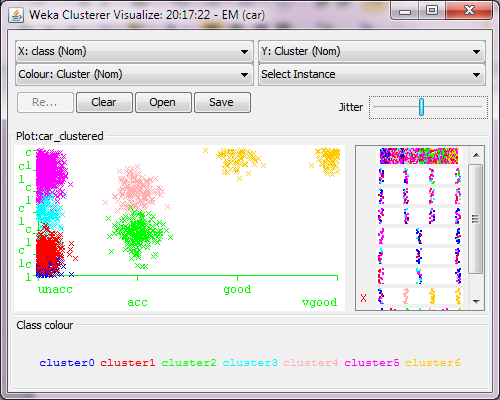
We may see that the car.arff works fine with Multilayer as this classifier is align with Non-linearly separable data like what we have in car.arff.

The percent of correctly classified instances is:

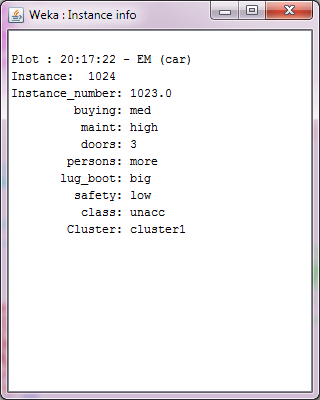
Correctly Classified Instances 1709 98.9005 %

** Clustering tab in WEKA**

Let’s load a fresh car.arff file into WEKA environment. Then we choose the **Cluster** tab, which allows running clustering algorithm on the data. Then we run the algorithm which is given by default, EM. Then we right-click on the result in the Result list, and choose **Visualize cluster assignments**.

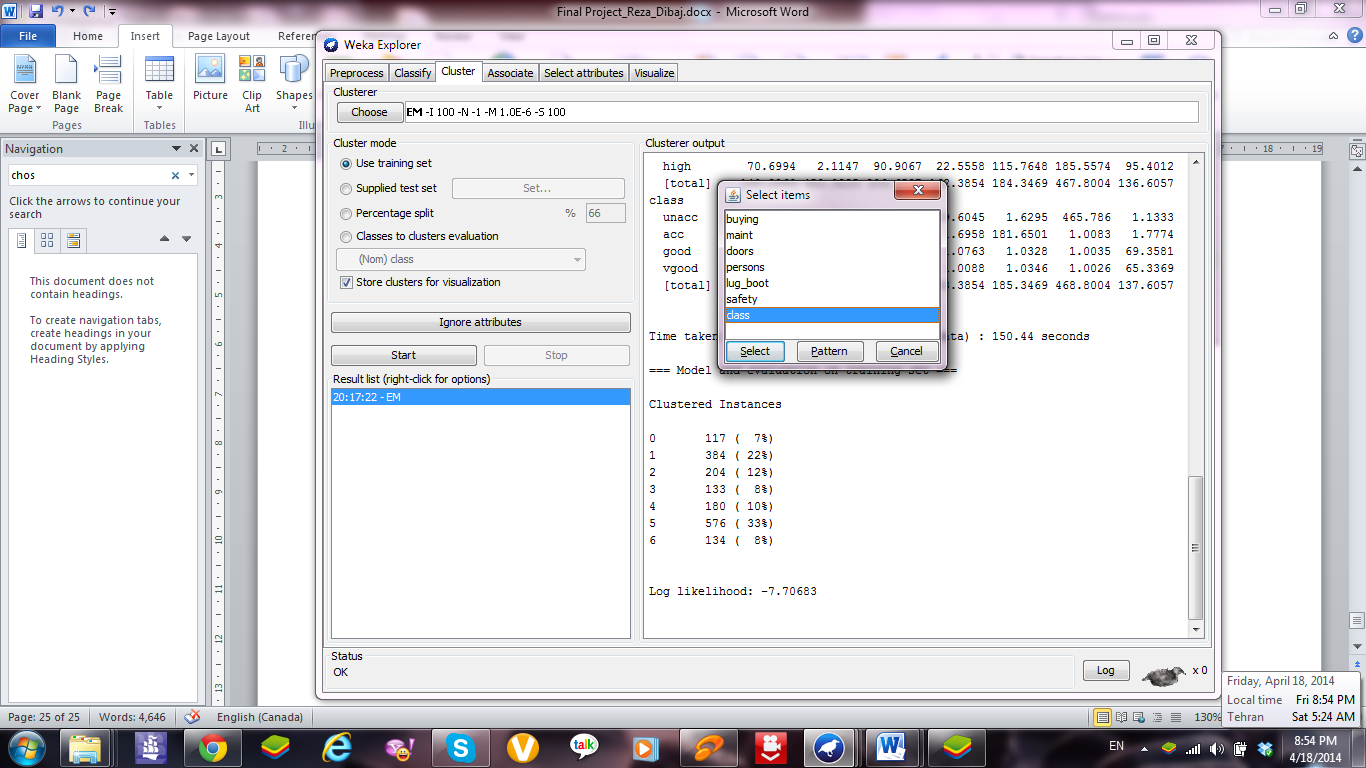


Then we make **x-axis** be class, and the y-axis be **Cluster**. We increase the jitter so the points do not overlap. The result can be seen in the picture that we can see it the right side.



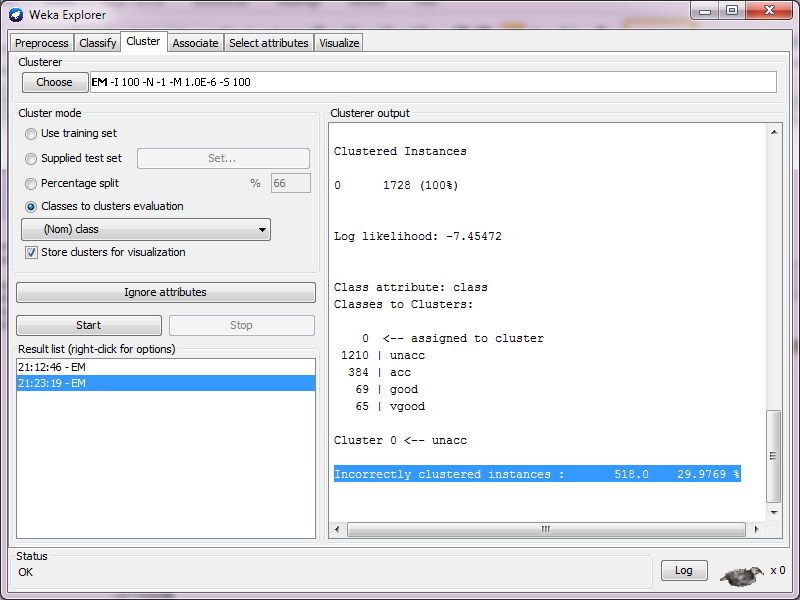
If we click on any X that is not classified correctly we can see a picture like this which tells us about that spot.

****



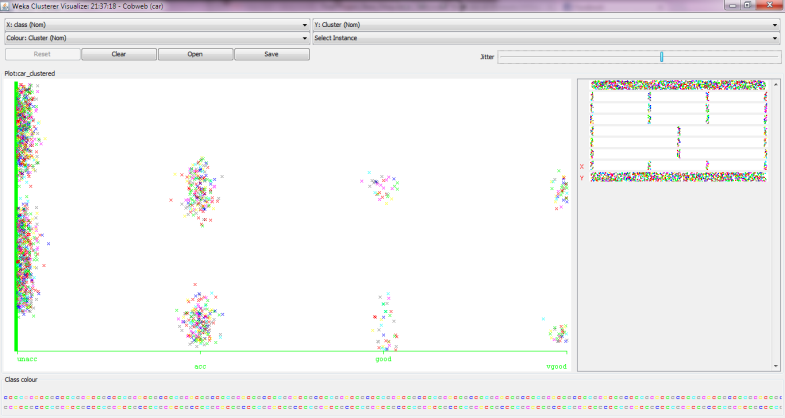
Now we click on **Ignore attributes** and then we choose **class** to be ignored.

Then we run the algorithm again. The result will be completely changed. Even the graph will be changed as well.



Then we choose Classes to clusters evaluation and accept the default class to be the class attribute. The incorrectly clustered instances are as follow:

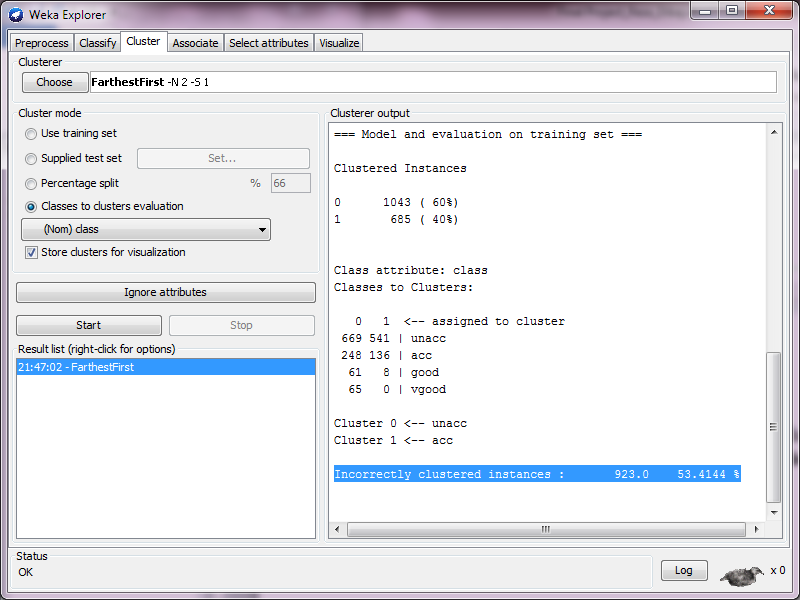
Incorrectly clustered instances : 518.0 29.9769 %



****

Let’s reload the car.arff and then go to the **Cluster** tab, and choose **Cobweb**, with **Class to clusters evaluation**. Then we run the results and finally we visualise the result as we did before.

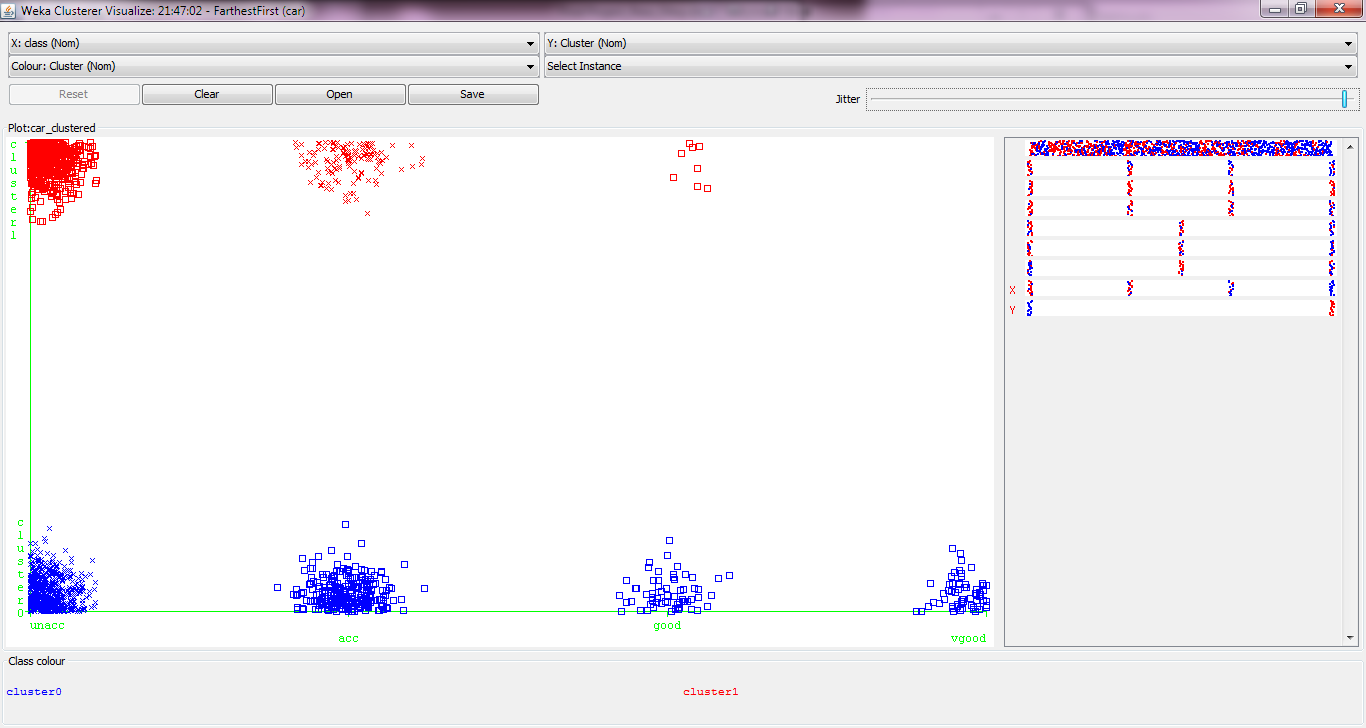
The result is in a picture that we may see in the right side.



****

Again we are going to reload the car.arff and then go to the **Cluster** tab, and this time choose **FarthestFirst**, with **Class to clusters evaluation**. The incorrectly clustered instances are as follow:

Incorrectly clustered instances : 923.0 53.4144 %

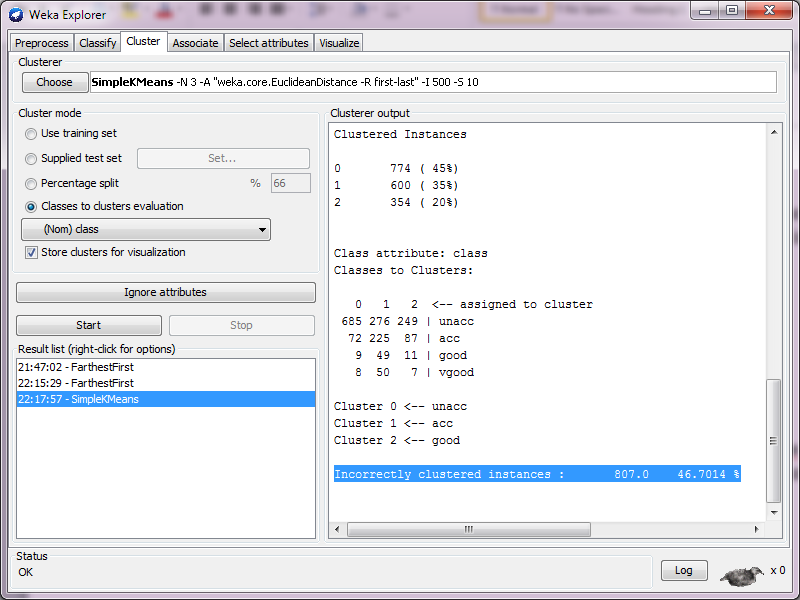


And if we finally visualise the result as we did before, the result can be seen in a picture that we have in the right side.

****

Now we click on **FarthestFirst** to the right of Choose, and then we set the number of clusters **numClusters** to 3 and re-run it again. In this condition the number of incorrectly clustered instances is as follows:

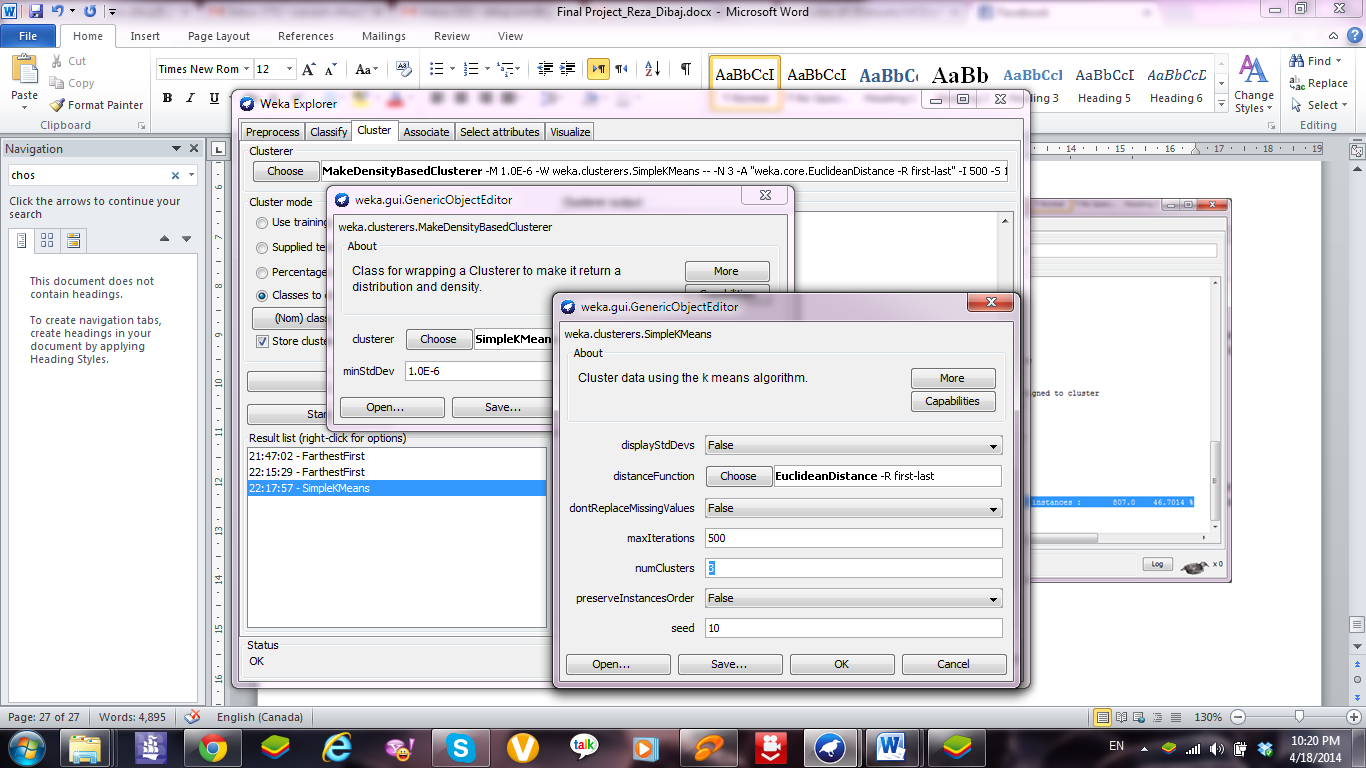
Incorrectly clustered instances : 1021.0 59.0856 %



****

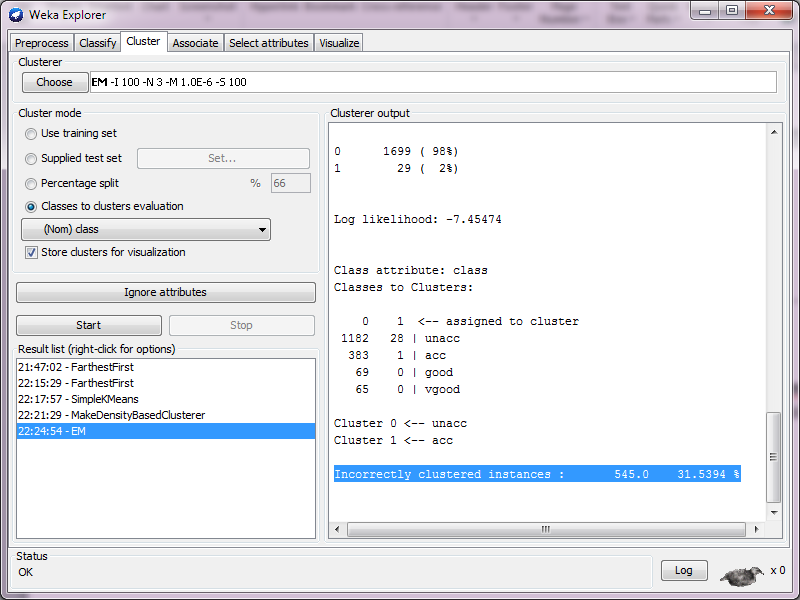
Again we choose another **Cluster**. This time we choose **SimpleKMeans and** set the number of clusters **numClusters** to 3 and re-run it again. In this condition the number of incorrectly clustered instances is as follows:

Incorrectly clustered instances : 807.0 46.7014 %

****

This time we choose another **Cluster**. This time we choose **MakeDensityBasedCluster** with **SimpleKMeans** and we set the number of clusters **numClusters** to 3 and re-run it again. In this condition the number of incorrectly clustered instances is as follows:

Incorrectly clustered instances : 771.0 44.6181 %

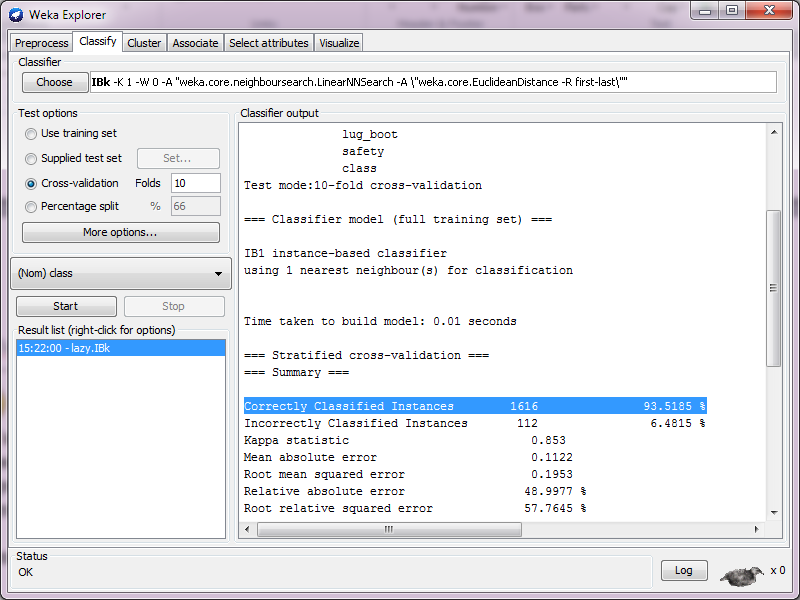
****

At this time we return to EM and set the number of clusters **numClusters** to 3 and re-run it again. In this condition the number of incorrectly clustered instances is as follows:

Incorrectly clustered instances : 545.0 31.5394 %

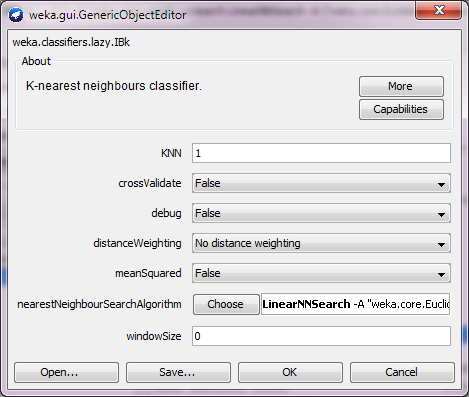
For conclusion we can say, the classification is the classes which we already have, but if we cluster our data we try to put them in some classes, and the more our clustering algorithm are accurate, the classification may see better result and less overlap.

Generally we may see by increasing the number of clusters, our accuracy may decrease and that is serious in some cases. Also sometimes we may see different clustering algorithms may work similarly on our data.

** Experiments with Nearest-Neighbour**

Let’s load car.arff file one more time. After that, I go to the **Classify** tab and then I go to **Classifier** section. Then I choose IBK from the **lazy** algorithm section. Then I run the algorithm by pressing the **Start** button. The result shows the following accuracy:

Correctly Classified Instances 1616 93.5185 %



Then we click on the textbox next to the right of the **Choose** button to see and change the classifier options. The first option is **KNN**, which determines how many of the nearest neighbours to consider when guessing the class of the test instance. The default value of KNN is 1.

Now we increase KNN from 1 to 5 and also 10, and run the algorithm again for KNN=1, 2, 3, 4, 5.

The accuracy for all KNNs are as follow:

For KNN=1:

Correctly Classified Instances 1616 93.5185 %

For KNN=2:

Correctly Classified Instances 1616 93.5185 %

For KNN=3:

Correctly Classified Instances 1616 93.5185 %

For KNN=4:

Correctly Classified Instances 1616 93.5185 %

For KNN=5:

Correctly Classified Instances 1616 93.5185 %

For KNN=10:

Correctly Classified Instances 1616 93.5185 %

In this specific dataset we find no difference at all. But in some examples we may see there could be a kind of optimum point between these KNN, like KNN=3 in glass.arff.

If we change KNN into 50 the accuracy will be:

Correctly Classified Instances 1327 76.794 %

In order to examine the role of each attribute, let’s remove each of them and check the accuracy accordingly.

Without buying:

Correctly Classified Instances 1224 70.8333 %

Without maint:

Correctly Classified Instances 1321 76.4468 %

Without doors:

Correctly Classified Instances 1628 94.213 %

Without persons:

Correctly Classified Instances 1011 58.5069 %

Without lug\_boot:

Correctly Classified Instances 1406 81.3657 %

Without safety:

Correctly Classified Instances 994 57.5231 %

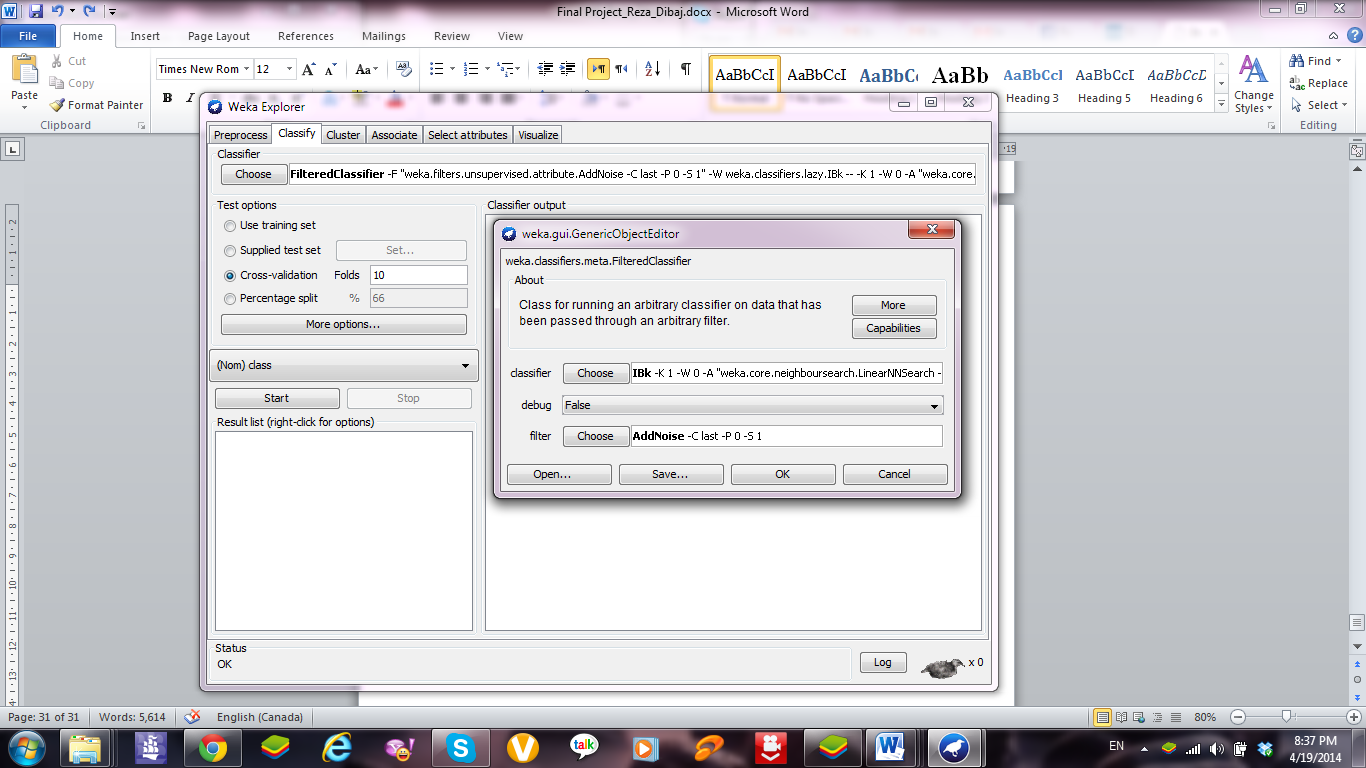
So, we can perceive that elimination of doors is the best choice. In next step we delete doors and continue the previous process from the beginning.

|  |  |  |
| --- | --- | --- |
| Subset Size (# of attributes) | Attributes in “Best” Subset | Classification Accuracy |
| 6 | {buying, maint, doors, persons, lug\_boot, safety} | 93.5185 |
| 5 | {buying, maint, persons, lug\_boot, safety} | 94.213 |
| 4 | {buying, maint, persons, safety} | 85.0116 |
| 3 | {buying, persons, safety} | 81.0185 |
| 2 | {persons, safety} | 76.5625 |
| 1 | {persons} | 70.0231 |
| 0 | { } | 70.0231 |

As it is clear we may have the best possible accuracy in the second row attribute-set, and the accuracy of that dataset is:

94.213 %

We should be aware of one important note that we should not rely on our test data to gain an unbiased estimate of accuracy for the future data. These results have come from our current data and they may change for any other dataset.

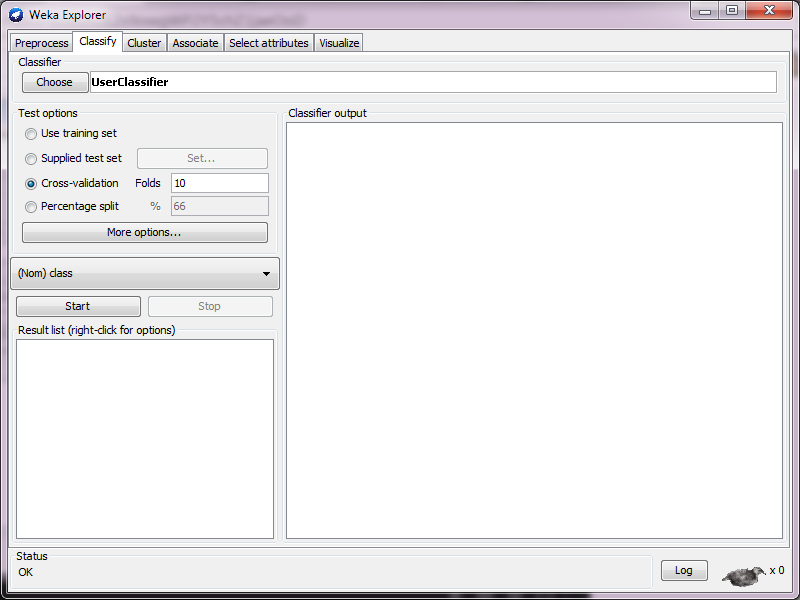
** Experiments Class Noise and Nearest-Neighbour Learning**

As we now from our lab experiences, if the training data contains noise, then the machine learning is going to be worse. In the following experiment, we add varying amounts of class noise that is wrong random values for the class attribute to see the effect on nearest-neighbour learning. To add noise to just the training data, we use a meta-learner.

Let’s load car.arff file one more time. After that, I go to **Classify**, then I choose **FilterClassifier** from the meta section. Then I modify the **FilterClassifier** so that it uses IBK as the learner and the unsupervised attribute filter AddNoise as the filter.

|  |  |  |  |
| --- | --- | --- | --- |
| Percentage Noise | K=1 | K=2 | K=3 |
| 0% | 93.5185 | 93.5185 | 93.5185 |
| 10% | 91.0301 | 91.0301 | 91.0301 |
| 20% | 88.1366 | 88.1366 | 88.1366 |
| 30% | 85.2431 | 85.2431 | 85.2431 |
| 40% | 80.9606 | 80.9606 | 80.9606 |
| 50% | 73.1481 | 73.1481 | 73.1481 |
| 60% | 60.8218 | 60.8218 | 60.8218 |
| 70% | 40.7986 | 40.7986 | 40.7986 |
| 80% | 19.9074 | 19.9074 | 19.9074 |
| 90% | 6.9444 | 6.9444 | 6.9444 |
| 100% | 1.4468 | 1.4468 | 1.4468 |

With increasing the effect of class noise, it drastically reduces the accuracy in our results.

** Using User Classifier**

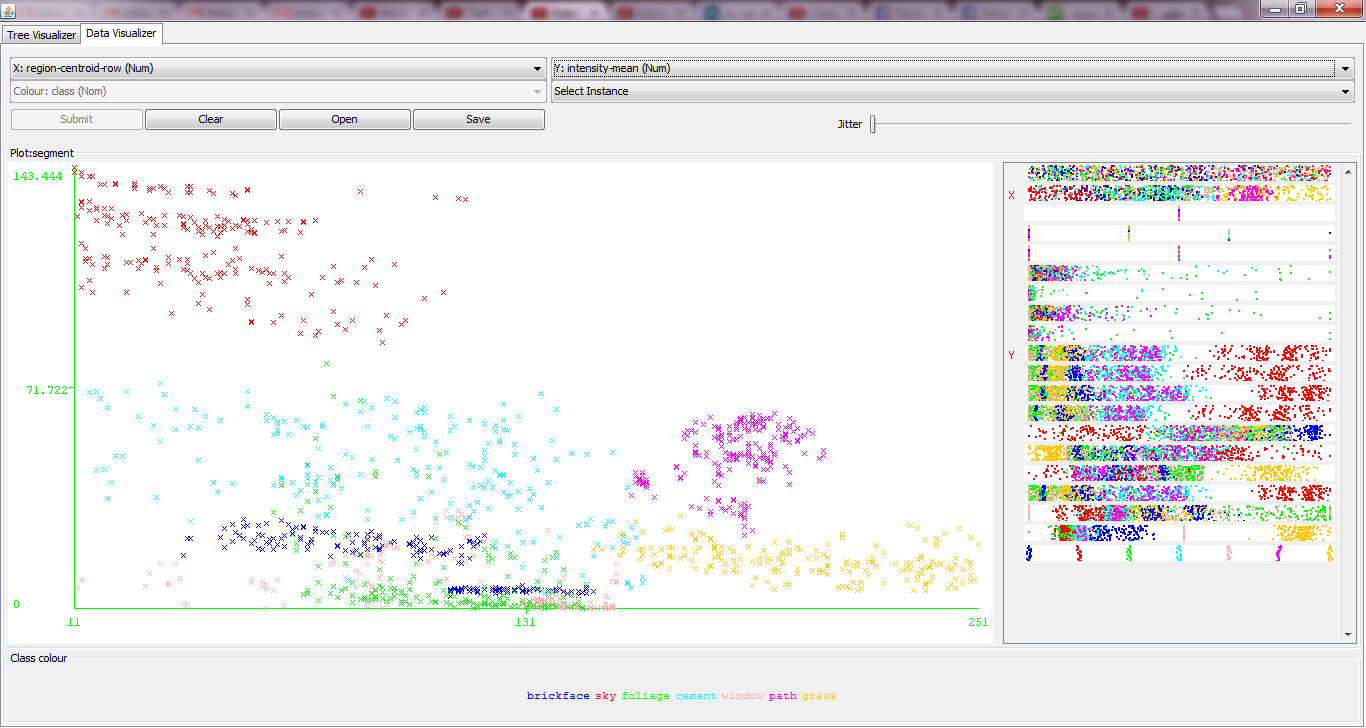
After loading a fresh segment-challenge.arff dataset, we go to the **Classifier** tab. Then we choose **UserClassifier** from the **Classifier** section.

Then we select Supplied test set, and then click on **Set…** and we open segment-test.arff.

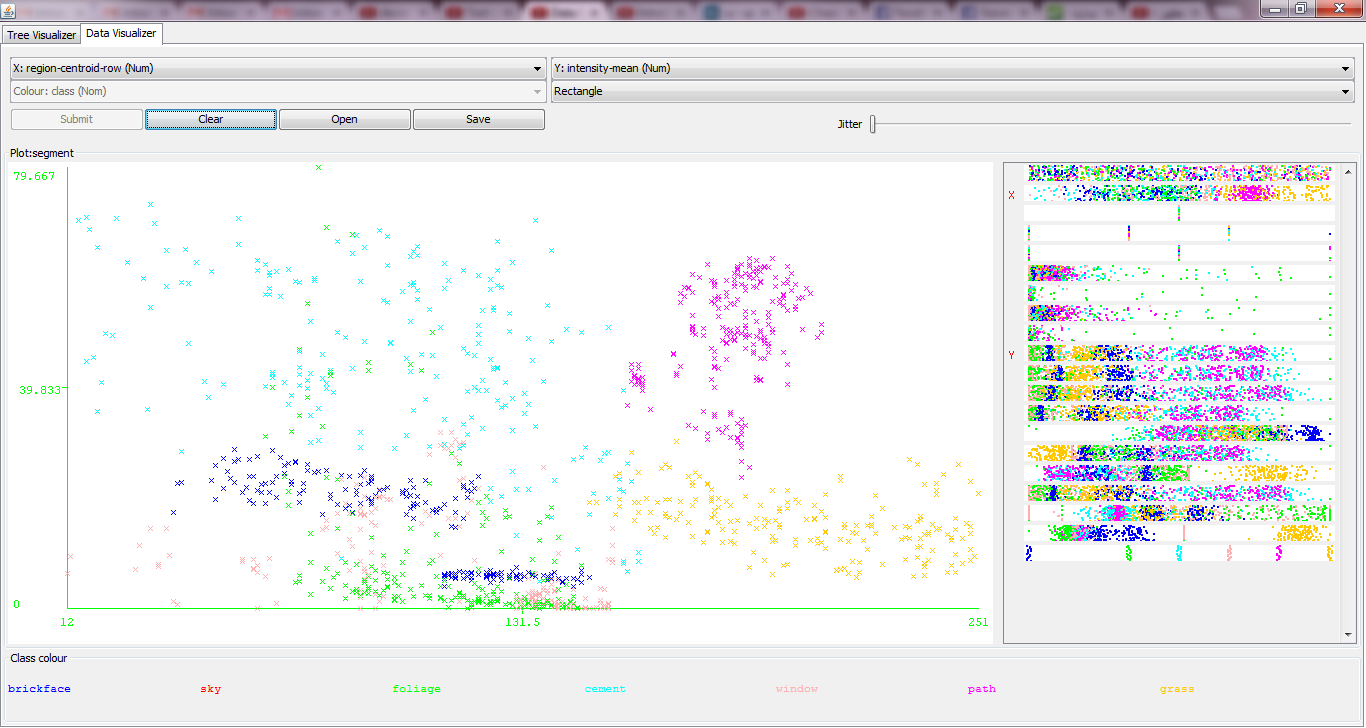
After that we press **Start** button. In the window that we can see, there are two tabs. One of them is Tree Visualizer and the other is Data Visualizer.

Let’s go to Data Visualizer tab.

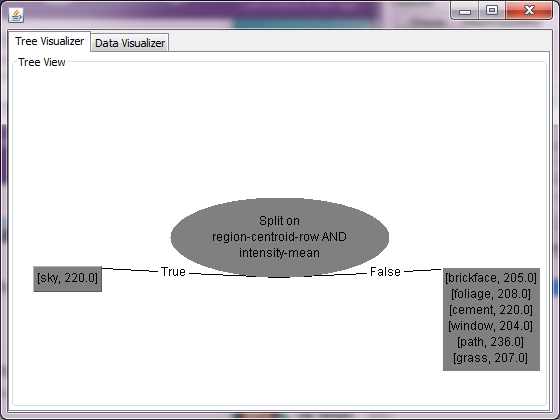
We change the X to region-centroid-row and we change the Y into intensity-mean.



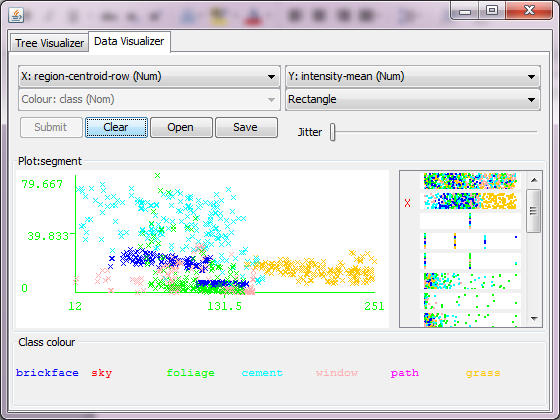
The result is like the right-side window:

****

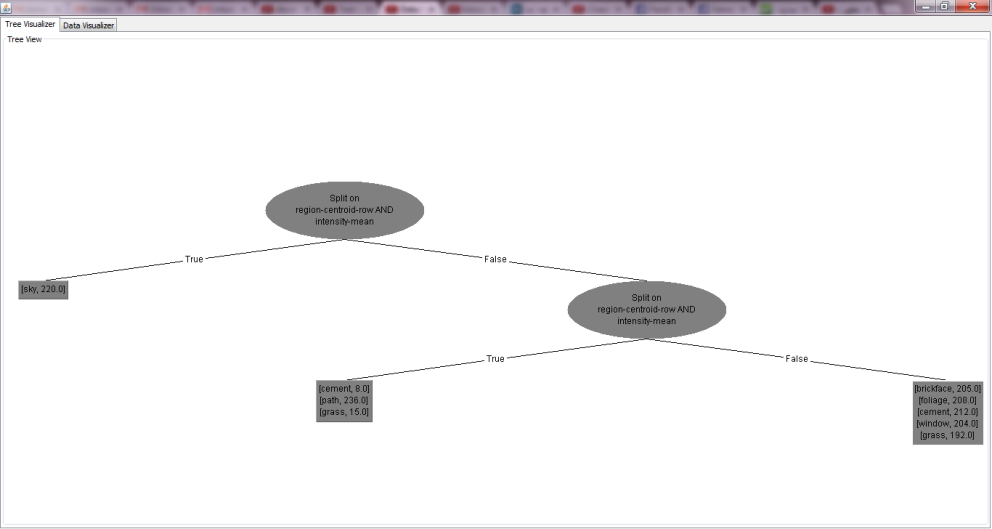
Then instead of Select Instance, we choose a rectangle, and after that we just select the red section of the frame, and submit. That selected area will be gone and the result is going to be something like this:

****

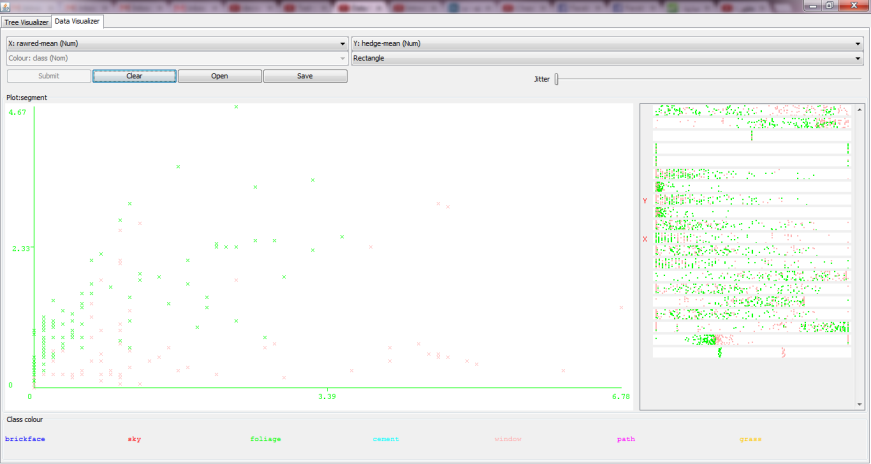
In the Tree Visualizer tab we may see a window like this:



Now if we select that purple section of the Data Visualizer graph, and submit it, the result could be like this:

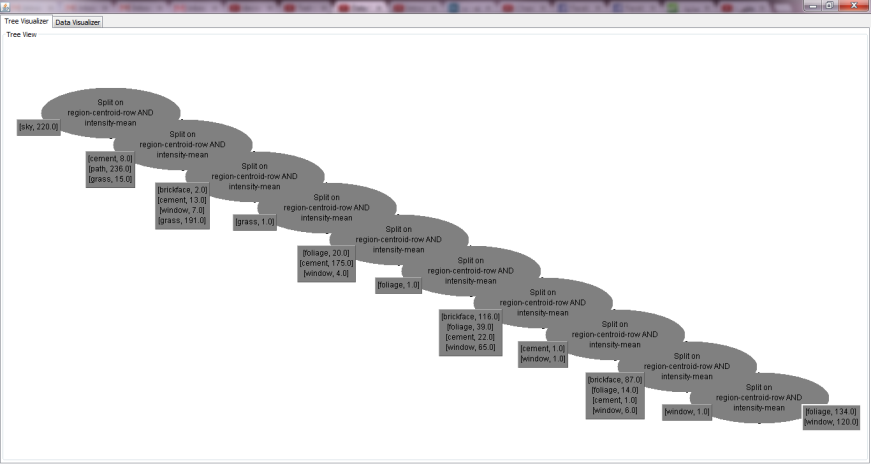
****

And accordingly the Tree Visualizer would change like this:

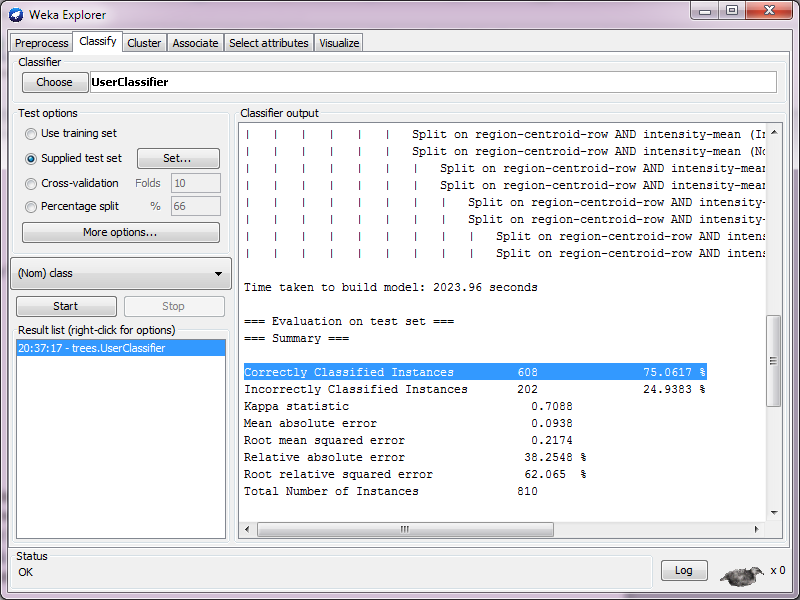
****Now let’s take more portions from the Data Visualizer:

Also we change the X and Y on top of the window or by clicking in the right frame.

The result is like this:



And if we go to the Tree Visualize tab, the result is like this:



****It is good to know that instead of choosing rectangle, we could choose polygon or polyline as well. In this window, if we right-click, then we can choose Accept The Tree, and we may not come back, so we must be sure of our selection and confirmation.

Now we can come back to **Classify** tab and check the accuracy:

Correctly Classified Instances 608 75.0617 %