

Prepared for:

Dr. Jiming Peng
Department of Computing and Software
McMaster University Hamilton,
Ontario L8S 4L8

TA: Mr.
Peng Du

Data Mining Course Project

Prepared by:

Dean Abdou	9806323
Anthony Helou	0058392
Asifa Hussain	9815943
Anish Passi	9942211

Course:

Introduction to Data Mining and Machine Learning
Soft. Eng. And Comp. Sci. 4TF3

Date: 10
December 2002

Table of Contents

1	INTRODUCTION.....	4
1.1	DEFINITION OF TASK.....	4
1.2	STRUCTURE OF DATA SET.....	4
1.3	COMPLEXITY OF THE TASK	5
2	PREPROCESSING THE DATA.....	7
2.1	ATTRIBUTE SELECTION.....	7
2.2	MISSING VALUES.....	8
2.3	OUTLIERS.....	8
2.4	OBSERVATIONS FROM THE CLEANUP	9
2.5	OTHER CONSIDERATIONS	10
	2.5.1 Clustering	10
	2.5.2 Linear Regression and Outliers.....	12
	2.5.3 Filtering.....	13
3	ALGORITHMS	14
3.1	PRUNING.....	14
3.2	DECISION TREE ALGORITHM.....	16
3.3	BUILDING THE TREE.....	18
4	EVALUATION	19
4.1	PREDICTING PERFORMANCE.....	19
5	WEKA RESULTS.....	21
5.1	DATA PREPARATION.....	21
5.2	WEKA ERROR ANALYSIS	21
6	ANALYSIS OF OUR CODE.....	24
6.1	TESTING.....	24
6.2	ERROR RATE.....	26
6.3	COMPARING ALTERNATIVES: WEKA VS. OURS	28
	CONCLUSIONS.....	29
8	REFERENCES.....	30
9	APPENDICES.....	31

APPENDIX A: 1R RESULTS	31
APPENDIX B.....	32
APPENDIX C: MISSING01.M.....	34
APPENDIX D: REPLACE_MISSING02.M	35
APPENDIX E: NOMINALIZE03.M.....	36
APPENDIX F: CATEGORIZE04.M	37
APPENDIX G: OUTLIER05.M.....	38
APPENDIX H: SPLIT.M.....	39
APPENDIX I: INFO.M	41
APPENDIX J: TREE.M.....	42
APPENDIX K: LINEAR REGRESSION RESULTS	43
APPENDIX L: ATTRIBUTE SELECTION.....	45
APPENDIX M: ZERO R	46
APPENDIX N: SMO.....	47

1 INTRODUCTION

1.1 DEFINITION OF TASK

The original aim of this project was to be able to predict a numeric house price given eight other covariates. List of covariates (attributes) are found in the next section.

Later, in the project the task was modified to predict if a house is cheap or expensive based on the 8 attributes listed below. The evolution of the task will be exemplified in the following sections.

1.2 STRUCTURE OF DATA SET

The data contains 20,640 instances. The data set can be found at:
<http://lib.stat.cmu.edu/datasets/> under the subtitle: houses. Zip [1].

The information on the data was collected using all the block groups in California from the 1990 Census [2]. It is therefore intuitive and logical that the results can only be applied to Californian Houses, possibly US west coast houses, and error rates will be noted.

The data sets include 9 attributes:

1. Median house price,
2. Median income of individuals living in block. Number merely correlates to income yet has no dollar meaning,
3. Housing median age i.e. median house age,
4. Total rooms in block,
5. Total bedrooms in block,
6. Population in block,
7. Number of households in block,
8. Latitude,
9. Longitude.

The 'creators' of this data were contacted via email to glean some information on what each attribute represents. The response was brief, and it said that if that specific attribute values had been modified slightly (i.e. divided by 10000) in order to avoid number explosion when they run their specific learning scheme. This applies to income for instance, which has no dollar meaning. Having said that, after the slight modification the

'relationship' between instances has to be intact i.e. a *high* dollar value received a *high* new value.

Sample data:

1. 4.5260000000000000e+005
2. 8.3252000000000000e+000
3. 4.1000000000000000e+001
4. 8.8000000000000000e+002
5. 1.2900000000000000e+002
6. 3.2200000000000000e+002
7. 1.2600000000000000e+002
8. 3.7880000000000003e+001
9. -1.2223000000000000e+002

All attributes are numeric.

The instances of the dataset represent values for the attributes listed above. California's residential areas were subdivided into 20640 zones or blocks. Information on the blocks is represented in the data [2].

Since the houses in a zone are located in pretty much the same geographical location; the latitude and longitude are very similar. This did lead us at the start of the project to disregard these attributes. The house price is averaged for all the houses within that block. The rest of the attributes are summations. For example, 322 people live in 126 houses in the example presented above.

One worry would be that the price is already averaged. However, when viewing each block as a unit itself, and not each house as a unit, the worry is eliminated. The worry would be that the averaging process removes some of the relevant information. However we are mining to see if there is a relationship between the attributes and a block, and maybe extending the conclusion to single houses (this is unlikely however).

Due to the modification of the task as stated in section 1.1- the data set was modified accordingly. The house prices were categorized into two groups: either cheap or expensive. The way in which this modification took place will be discussed in section 2.4.

Therefore, the final dataset consisted of 1 nominal class and 8 numeric attributes.

1.3 COMPLEXITY OF THE TASK

This project and report has been subdivided into two main parts. One part is our team's effort at implementing a learning algorithm. *The algorithm picked is the combination of C4.5 and linear regression.* The exact algorithms will be discussed in section 3.

The second part of the report deals with our analysis of the data set using Weka. This took on two forms.

- Analyzing the data using C4.5, 1R, Zero R, Support vector machine and Naive Bayes. Then comparing the results with our own coded C4.5.
- Analyzing differences between different learning schemes using our data set.

The objectives outlined above have proven to be quite difficult to fulfill especially with the time constraints. Not all the objectives might have been fulfilled. However; the greater goal that we set out for ourselves was to learn the most we can about the subject given the amount of time we were allotted!

2 PREPROCESSING THE DATA

2.1 ATTRIBUTE SELECTION

Before dealing with the actual values in the dataset, one has to evaluate the relevance of the attributes in the dataset. Throughout the course we have learned that "...adding irrelevant or distracting data attributes to a data set often confuses a machine learning system." [3]

Witten/Frank describe a significant disruption in results generated from the C4.5 algorithm, when a random attribute is used. This is only intuitive because as the tree grows deeper, it will get to a point where the irrelevant attribute will seem as a good choice. And picking it is obviously a bad idea.

Benefits from making the set smaller by eliminating an attribute(s):

- Target function would be more compact.
- Speedup in running actual learning algorithm.
- Due to the result being more compact, it will be more comprehensible.

Our book goes into several techniques of attribute selection. One of them would be to do a manual selection after attaining a good understanding of the dataset. And this is what we did. Some other techniques would be wrapper techniques where the learning scheme is run on the data first to glean information on the relevant attributes.

Making use of our understanding of the attributes we applied the first technique. There are not many spurious attributes at face value. The only attributes that were in question were latitude and number of bedrooms.

- Latitude

The dataset used was for houses that were within California, whose latitude is relatively the same for the entire state. Furthermore, its value is negative which complicated certain part of the learning algorithm. By it being negative, the learning algorithm ran much slower because it had many extra calculations in order to deal with absolute values. Much longer meaning a difference between 4 hours and up to a really long time (something we did not want to find out).

- Bedrooms

It was our understanding that the number of rooms should be enough to indicate size. This number of *bedrooms* attribute would just make run times longer and the decision tree more complex.

However, after running Weka and our algorithm on the reduced attribute data set, we got large errors in our results. Because of this we chose to reincorporate both attributes. To make computations simpler we multiplied latitude by negative one.

A lesson learned after 'great pain': The dataset at hand has been already carefully chosen to run learning algorithms. This careful choosing process was performed by the creators of this dataset.

One final note on the subject of attribute selection: Weka does have functions that perform attribute selection. This was learned only after noticing high error rates after attributes were removed by the crude method described above. Nevertheless, attribute selection was run using Weka. It was reassuring to know that even Weka preferred not to remove any attributes. The results are found in Appendix K.

2.2 MISSING VALUES

The data set is complete. However, the data set can be corrupted by deleting values. *The purpose of this is to prove the ability of the code to handle such difficulties.* Adding missing values is done randomly. The file **missing01.m** takes care of this. It can be noted the number of missing values can be hard coded into this code. Initially 2000 missing values were infused.

We had the following choices in order to deal with missing values:

- Ignoring instance completely. Too draconian.
- Reconsidering missing values as a separate value on its own. In this case a deeper understanding of the dataset reveals that missing values have no special significance.
- Replacing missing value with average of that attribute.
- Replacing missing value with average of same class within attribute.
- From nominal values replacing missing values with the most prevalent value for that attribute.
- Implementing a more sophisticated scheme that uses weights and traverses the tree.

The choices we made will be noted in section 2.4.

2.3 OUTLIERS

The outlier analysis proved to be very tricky. At first we incorporate the draconian approach of eliminating all instances which had one value of an attribute lying outside 2 standard deviations.

However, after consultation with Dr. Peng, it became apparent to us that the approach taken might cut out important clusters of data, if they existed.
Please refer section 2.5 for some other ideas we came up with for outlier analysis.

2.4 OBSERVATIONS FROM THE CLEANUP

With the initial run of the cleanup process and learning algorithm (including a Weka simulation) the results yielded 35% - 40% error rate. Much of the fault in incurring such a high error is due to an implementation of the cleanup phase where too many missing values were removed and later replaced with averages.

It was later decided that we should use no missing values. We have the code and the means to deal with missing values, but due to the extremely high error rate, we decided not to implement it. It had been simulated, and it worked very well. We decided that it will not be used because it increases the error in an apparently error prone data set.

Had we implemented the code for missing values, the clean-up phase would follow the following order:

- **missing01.m**: infuses a fixed number of missing values into data set.
- **Replace_missing02.m**: Replaces missing values in attributes 2 through 9, with the average of that attribute.
- **nominalize03.m**: Replaces our class which is numerical, with nominal values. The values at this point are -110 for less than 180000, -111 for greater than 180000.
These two nominal values represent cheap and expensive houses respectively. This nominalization process is to initialize the data set with a nominal value, and to make the class nominal which makes things easier when it comes to making the decision tree. Furthermore, since our implementation of C4.5 will be in Matlab it is more convenient and faster to use number rather than strings.
- **categorize04.m**: Replaces missing values in nominal attribute with most prevalent value for that attribute.
- **outlier05.m**: Removes values that lie beyond a fixed value of standard deviation away from mean.

For numeric values we replaced simulated missing values with the average for that attribute.

As for nominal values we replaced simulated missing values with the most prevalent value for that nominal attribute.

After implementing *missing values* and *outliers analysis* on the dataset it was found that the errors in our final decision tree were a lot higher than if we dealt with the original data without taking out the outliers and not dealing with missing values.

Thus it was decided that due to increased error rate and given the fact that our data did not contain these missing values in the first place anyway we shall actually not implement these segments to the original data.

Error rate with missing values infused and outlier analysis performed:

=== Stratified cross-validation ===

Correctly Classified Instances	7049	65.2806 %
Incorrectly Classified Instances	3749	34.7194 %

Error rate with no missing value and no outlier analysis:

=== Stratified cross-validation ===

Correctly Classified Instances	18196	88.1589 %
Incorrectly Classified Instances	2444	11.8411 %

Please note that J4.8 was used for this analysis.

Other outlier analyses techniques could have been used, however, at this point we had spent too much time on clean up. Other techniques we considered will be discussed below. None were implemented however.

2.5 OTHER CONSIDERATIONS

We would have liked to implement other variation of clean up, but we were not able to do so due to time constraints. We found that the following techniques would have been beneficial to our analysis.

2.5.1 CLUSTERING

Cluster analysis is an exploratory data analysis tool for solving classification problems. Its object is to sort cases into groups, or clusters, so that the degree of association is strong between members of the same cluster and weak between members of different clusters. Each cluster thus describes, in terms of the data collected, the class to which its members belong; and this description may be abstracted through use from the particular to the general class or type [4].

Clustering is applied when there is no class to be predicted but rather when the instances are to be divided into natural groups. Clustering requires different techniques to the

classification and association learning methods. An instance can either be exclusively within a group, or it can overlap between several groups. An instance can also be probabilistic, where the instance belongs to each group with a certain probability [3].

Three basic methods for clustering: k-Means algorithm, incremental clustering, and statistical clustering.

k-Means (also called Iterative Distance-Based Clustering) [4]

- forms clusters in numeric domain
- like nearest-neighbor scheme of learning
- first specify in advance how many clusters are being sought: this is the parameter k .
- k points chosen at random as cluster centers
- the mean is then calculated
- these centroids are taken to be the new centers
- the whole process is repeated with the new cluster centers
- iteration continues until the same points are assigned to each cluster in consecutive rounds, at which point the cluster centers have stabilized and will remain the same thereafter.
- the centers do not represent a global minimum but only a local one

Incremental Clustering [4]

- works instance by instance
- at any stage the clustering forms a tree with instances at the leaves and a root node that represents the entire dataset
- instances are added one by one, and the tree is updated appropriately at each stage
- an update may consist of either as simple as finding the appropriate node to place a leaf to represent a new instance, or it may be complicated where restructuring of part of the tree
- deciding how and where to update is called *category utility*, which measures the overall quality of a partition of instances into clusters
- in the beginning, each new instance form their own sub-cluster und the overall toplevel cluster
- each new instance is evaluated for each leaf, to see if it is a good "host" for the new instances.
- two types of action occur: merging, and splitting
 - Merging: replacing several children with one node
 - Splitting: taking the node and replacing it with its children
- splitting and Merging provide an incremental way of restructuring the tree to compensate for incorrect choices caused by infelicitous ordering of examples

Statistical Clustering [4]

- the goal of this type of clustering is to find the most likely set of clusters given the data and the prior expectations

- because no finite amount of evidence is enough to make a completely firm decision on the matter, instances--even training instances--should not be placed categorically in one cluster or the other: instead they have a certain probability of belonging to each cluster
- this helps to eliminate the rigidity that is often associated with schemes that make hard and fast judgments
- the foundation for statistical clustering is a statistical model called *finite mixtures*
- a *mixture* is a set of k probability distributions, representing k clusters, that govern the attribute values for members of that cluster
- each cluster has a different distribution
- a particular instance only belongs to one cluster, but it is not known which one
- finally, the clusters are not equally likely: there is some probability distribution that reflects their relative populations

It is truly a pity after learning all the above we did not have time to use clustering. However, had we had time we would have used clustering for cleaning up the data rather than it being the actual learning algorithm used to find target function. k-means would have been probably used because of the numeric nature of the dataset. Outliers would be any clusters with less than a specific number of instances.

The reason why our outlier analysis might not have worked so well is because it might have been cutting out significant clusters of the data. For example, it very probably cut out houses in the Beverly Hills area since the house price there is expected to be much higher than the average of the whole group.

2.5.2 LINEAR REGRESSION AND OUTLIERS

Since it was intended to represent the instances in the leaves of our decision using linear function, this topic in Witten/Frank seemed especially interesting.

Statisticians solved problems caused by noisy data in linear regression by checking data for outliers and manually removing them. In case of linear regression, they can be identified visually. Outliers have a dramatic effect on the usual least square regression because the squared distance measure accentuates the influence of points far way from the regression line. Statistical methods that address the problem of outliers are called robust. Regression can be made robust in the following ways [3]:

1. Use absolute value distance measure instead of the usual squared one. This weakens the effect of outliers.
2. Identify outliers automatically and remove them from considerations. Example: Form a regression line and remove from consideration those 10% of the points that lie farthest from the line.

3. Minimize the median (instead of the mean) of the squares of divergences from the regression line. This method copes with outliers in both the X-direction and the Ydirection. There is a serious disadvantage to this technique: high computational cost, which can make them infeasible for practicality.

In our case unfortunately we had no time to implement linear regression on the leaves. Later, on in this document we explain that we nominalized the numeric attributes. One would ask: *How then do you use linear regression on this nominalized data?*

The trick is to number the attributes. This is done by adding another column onto the dataset. This column is not modified in the learning scheme. It just sticks till the end. In that way, we can tell which nominal instance corresponds to which numeric one. The numeric can be substituted back in, and linear regression can be used to describe the route.

2.5.3 FILTERING

A problem with any form of automatic detection of incorrect data is that there is no real way of telling (except asking an expert) whether a particular instance really is an error, or whether it just does not fit the type of model that is being applied.

In statistical regression, it will usually be visually apparent, if the wrong kind of curve is being fitted. However, most classification problems cannot be easily visualized. Although good results are obtained on most standard datasets by discarding instances that do not fit a decision tree model, this not very helpful when dealing with a particularly new dataset.

One solution is, using several different learning schemes, like decision trees, nearest-neighbor learner, to filter the data. The approach to this is to ask that if all schemes fail to classify an instance correctly before it is labeled erroneous and removed from the data. Using this filtered data as input to a final learning scheme gives better performance than using the three learning schemes and letting them vote on the outcome. Training all the three schemes on the filtered data and letting them vote can yield even better results. However, there is a danger to voting techniques: some learning algorithms are better suited to certain types of data than others, and the most appropriate scheme can get outvoted. One possible danger of the filtering approach is that they might be sacrificing instances of a particular class (or group of classes) in order to improve accuracy on the remaining classes. However, it has not been found a problem in practice. Automatic filtering is a poor substitute for getting the right data initially. If it is too time consuming to implement this in our case.

3 ALGORITHMS

In our preliminary analysis of the data it was our understanding that because of its very nature of being numeric, it dictates that an algorithm that deals with numeric attributes be chosen.

Two natural choices are:

- Linear models
- Instance based learning

The preference at that point is linear models. The vision of the final product was an interface where relevant data is plugged in, and then a house value is outputted by the code. Possibly an interval based on a confidence level can also be outputted. For this, a quick analysis is required. In linear models, the training phase will be done only once. The products of the training phase are weights for the equation [5]:

$$\text{House value} = w_0 + w_1(a_1) + \dots + w_8(a_8)$$

As for instance based learning, each time a new instance house value is to be calculated, comparison against the training set is to be performed - each time. This takes a long time and it would seem inconsistent with our practicality vision for the program.

Upon consultation with Dr. Peng it became apparent that mathematical models obtained as described above would lead to a linear model that is not very stable. That is because the analysis is done on the whole dataset all at once. All differentiating characteristics would be ironed out to give an unstable model.

As suggested to us, a better learning algorithm would be to initially build a decision tree using the data followed by using linear regression to find a model that describes the leaves of the tree.

Vector machines were briefly considered, however they deal with multiple classes, which we do not have.

Nonlinear models would have also been considered had we had more time.

3.1 PRUNING

We initially came up with some research that we needed in order to decide what kind of pruning techniques to use. We found the following:

- Two types of pruning:
 - ✎ Prepruning (sometimes called backward pruning)

- ✂ Deciding during the building of the tree, when to stop
- ✎ Postpruning (sometimes called forward pruning)
 - ✂ After building tree, see what is necessary, and throw away the rest
- Postpruning is usually more advantageous
- Two operations for postpruning:
 - ✎ *Subtree replacement*
 - ✎ *Subtree raising*
- These operations can be applied to each node separately.
- Subtree replacement
 - ✎ The idea is to select some subtrees and replace them by single leaves, but accuracy usually decreases
- Subtree Raising
 - ✎ Lower node is raised to replace a higher node. The leaves of the raised node are noted with a ' so that they are not confused with the original values
 - ✎ Time consuming operation

The choice we have made for tree pruning (inside our learning algorithm) is Prepruning. This is a simpler method that suffices for our problem. The maximum depth of our tree is 8.

There were two prepruning methods considered:

- Limiting the number of instances in split nodes.
- When information gained has reached a predefined limit.

At the beginning of trials and while we were still implementing the algorithm, the information gained varied very little. The numbers coming in at each split were in the range of 0.001 to 0.002. It did not make much sense at that point to design for pruning using information gain. We instead opted for limiting how few the instances could be in each node.

The code for our prepruning proved to be especially simple:

```

if ( sizeL(1) >= 50 ) %Pruning on number of instances in left node
treemodified3(leftD, A,V); else
    %print out results for that node
end

```

The complete code for our tree generating program can be found in the appendices.

It is worthy to note the accuracy of the learning algorithm suffered in Weka when no pruning was performed (no subtree raising). The default J4.8 performs pruning to a prespecified confidence threshold. Pruning can be turned off by using the U option.

Note:

Some postpruning was performed on the results of our code. This tool is the form of subtree replacement. This was done manually and will be explained in section 6.1.

3.2 DECISION TREE ALGORITHM

The decision tree algorithm used implements C4.5 (loosely). It deals with numeric attributes, pruning and missing values. Missing values have been discussed in section 3. Pruning and numeric attributes will be described in the following sections.

Complexity:



For the purpose of demonstrating the complexity of decision trees, the training data contains 20640 instances and 8 attributes. We need to make some assumptions about the size of the tree, and we assume that the depth is on the order of $\log 8$, that is $O(\log 8)$. This is the standard rate of growth of a tree.

The computational cost of building the tree is $O(20640 * 8 \log 8)$.

Numeric Attributes

To start, it is worth noting that the main feature that distinguishes our learning form ID3 is that it can deal with numeric attributes.

Substituting nominal for numeric can be divided into two broad techniques.

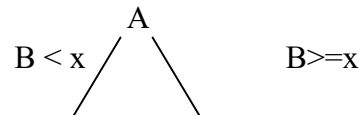
- Global
 -  Which prepares the data set for the learning scheme before the analysis begins. This is done only once at the beginning. It saves time and is very convenient.
- Local
 -  At each node in building the tree, the 'reanalysis' to nominalize is performed. This technique does seem more accurate, because it is for a more tailored context. However, looking at it from another angle, one could say that less data is analyzed at each lower node.

Another way to subdivide (descretization) techniques is by deeming them supervised or unsupervised. The supervised option is where the class is taken into account in the subdivision. An unsupervised technique would be to subdivide the attribute on the basis that a certain number of instances in each 'bin'. A supervised scheme is descretization based on frequency. A more recursive technique would be to use entropy.

Repeated splitting can be performed to get more accurate binning. We do not go into this in our algorithm. In the analysis section of this report, we see the indication that a degree of over-fitting is evident in Weka results. So we did not invest more effort into multiple splits.

The purpose of this section is to lay some light on the method we (C4.5) used to deal with numeric attributes in our dataset. The goal is to gain useful information from the attributes in such a way, which assists the divide and conquer approach of C4.5.

Eight of the nine attributes in our dataset are numerical. Thus, it is important to deal with these attributes separately and extend the algorithm to incorporate this. For simplicity purposes, the possibilities shall be restricted to binary splits. i.e., of the form:



Where A is the first attribute to split on, B is a numerical attribute to split on next, and x being a constant where the cut shall be made. It is easy just to choose a number x lying halfway between the values of an attribute for the cut, but it is not the best solution. A more complex, yet better, method mentioned in the textbook [3] is used for our purposes. The idea is that after sorting the data with the attribute to cut on, a cutting point is chosen from all the rows where the value of the class changes. Info and gain ratio formulas given in the textbook are used to come up with the best point to cut on, in other words, to come up with the 'x'.

For example:

Class :	Yes	No	No	No	No	Yes	Yes	Yes	No	No
Attribute:	64	65	68	69	70	71	72	75	80	83

The bars denote the points where the splits can be made.

Using: $\text{info}([\text{yes_on_left}, \text{no_on_left}], [\text{yes_on_right}, \text{no_on_right}])$ (of each bar) we find which point to cut on. Basically, it is the point with the least information (or most gain) which is chosen.

It is worthy to note 2 things at this point:

- More than one split can be made and that is what the book recommends [3]. However, applying this would make our algorithms too complicated and too long. We feel it is sufficient to note that we realize that this is possible and we chose not to use it.
- Gain ratio could have been used. This would have a simple modification to the code. However, after running test runs on information gain alone, the test runs proved to be especially time consuming. We found that it took two hours for just the information gain alone and this was just to attain values in order to nominalize the numeric attributes.

Having said that, it is encouraging and justifying in pointing out that our results for splits did correspond closely to some of that of Weka.

split.m deals with splitting of numeric attributes. It is included in the Appendix G.

3.3 BUILDING THE TREE

In summary, we would like to point out that only binary splits are allowed on each node. Furthermore, attributes shall be split on only once. This is unlike Weka, which resplits on an attribute as it sees fit. First we used **split.m** to get the values where the numeric attributes discretized (or made normal) then **treemodified03.m** is used to build the tree. A useful function called **info.m** (Appendix I) is used to calculate the information gain at a split. The output of **split.m** is:

V =

1.0e+003 *

0.0041 0.0250 2.1640 0.4020 1.3470 0.5900 0.0380 0.1220

A sample output for **tree.m** can be found in section 6 (the full output is several pages long, thus it will not be included in this document).

Finally, as a final note, linear regression was not used to describe each leaf rather a more crude method was used. The method is assigning a nominal value of cheap or expensive to each leaf depending on which class is more prevalent in that leaf. The reason for this approximation

1. the output function (the tree) becomes more comprehensible.
2. the learning system as a whole becomes simpler
3. testing the learning scheme is easier as is shown in the following sections.

4 EVALUATION

To start out the discussion of evaluation one must characterize the amount of data that is available, since this would have a significant impact of choice of training and testing sets.

The data is abundant, with more than 20000 instances available.

So the action that will be followed is to remove or holdout one third of the data from the original set and test the results with them. Removing the instances is a good idea since these instances will have nothing to do with formulating the rules. This avoids what would be re-substitution error. Note that this might not have been possible had we had a small data set.

There are three possible sets really: training set, validation set and testing set. At this point no validation will be used. It is our belief that this would be superfluous.

4.1 PREDICTING PERFORMANCE

The test cases will be used to test the classifier obtained. Successes will be recorded. The total amount of test cases will be 3000. Now the performance shall not be displayed as only the ratio of successes/total test cases. Rather, a confidence interval will be used.

Expected success rate $f = S/N = p = \text{mean}$

Variance = $p(1 - p) / N$

The above have been obtained since the model for our events is taken to be a Bernoulli process.

For a large distribution this process can be modeled as a normal distribution. Our scenario is that of one that can be modeled as a normal distribution, since using the rule of thumb that if number of trials is greater than 30 one can go with the approximation.

Standardizing and using equation found on page 124 of Witten and Frank [3], we can find the confidence interval.

The confidence level chosen will be 90%. This is the most common value used when dealing with confidence levels.

Even though holdout is used in testing, this does not mean that full-fledged cross validation will be used (only holding out a third of the data set). Cross validation is a technique used usually when the data set is limited, which is not the case in our situation. Due to the large size of the testing set, one has not to worry about the set not being representative. It is sufficiently large to assume that it is representative, therefore, stratification, cross validation, leave one out and bootstrap does not have to be used.

For further discussion on results please refer to the next two sections.

5 WEKA RESULTS

5.1 DATA PREPARATION

In order for Weka to access the data, the data must be converted to ARFF format. This is done with Excel. The following is performed:

- Text file is opened and saved on a comma separated format.
- The beginning signifier for the ARFF file is inserted.
- The file is saved as an ARFF extension.

5.2 WEKA ERROR ANALYSIS

Of the many schemes Weka provides for error analysis, three schemes were used.

- Re-substitution error
 - ✎ Error rate obtained using the training data. This method is not a good indicator of how the learned rules will perform for new instances.
- Subdividing the data into training and testing sets.
 - ✎ The recommended partition is 1/3 for testing and 2/3 for training. Note that this method of error analysis is viable in the first place because the data instances are abundant. The size not only improves the error analysis but also the classifier obtained.
- Stratified ten fold cross validation.
 - ✎ In this final scheme the data is subdivided into ten parts. One part in ten fold cross validation is held out, and the rest is used for training. The held set is then used for testing. The testing scheme is repeated ten times and the ten error yields are averaged out.

Later option 2 was dropped, because it was real overkill of the error analysis and besides by default Weka runs analysis 1 and 3. Not much discrepancy was evident between option 2 and 3 in the early tryouts. However, option 2 did yield slightly higher error rates than options 1 and 3. This is only expected since for option 2 there are no instances used in the training set *and* testing set.

The learning schemes used in Weka are:

- 1R
- 0R
- Support Vector machines
- C4.5

- Linear Regression
- Naive Bayes

Some notes on these schemes after analyzing the results:

- 1R
 - ✍ Excellent results for complexity of algorithm. It even beat Naive Bayes!
- Naive Bayes
 - ✍ Relatively simple. A reduction in accuracy if attributes are dependent. Some of the attributes are obviously dependent e.g. Number of bedrooms and number of rooms. Another fact that might reduce accuracy for Na ve Bayes is that is the data does not fit a normal distribution. (Assuming Weka uses Normal distribution for Na ve Bayes)
- Support Vector machines
 - ✍ Significant improvement over linear regression. This possibly implies that data set is not very linear, or at least has nonlinearities .

The results are summarized below. It is worthy to note here that the original set (with the class numeric) had to be re-substituted as the training set.

Scheme	Correctly Classified Instances	Percentages
ZeroR	10338	50.0872%
OneR	16053	77.7762%
Naive Bayes	15045	72.8924 %
C 4.5	19252	93.2752 %
Support Vector Machines	17194	83.3043 %

Scheme	Correlation coefficient
Linear Regression	0.7982

One of the most noteworthy learning schemes analyzed was the simple 1R method. Regardless of its simplicity, it is a good method to establish a baseline for performance. Furthermore, as Mitchell as reiterated several times, try the simplest first!

In general the 1R results make sense.

b2:

< 2.4175 -> cheap

>= 4.94745 -> expensive

Groups with low income have cheap houses, while high income groups have expensive houses. The omitted results between these two extremes show a slight degree of overfitting.

As a matter of curiosity 0R was implemented. The results make sense since we have a class of 2 nominal values. And with no relation (basically a guess) one has a 50% chance of getting the class right with the classes equally divided (which is what we did in our case to nominalize the class). Results for 0R can be found in Appendix L.

From the 1R results and results from analyzing results from C4.5 one notices that the attribute Median income of individuals living in block is the most important attribute. From C4.5 we see it is the most important because it is at the root node. This result is reasonable, since the richer the people living in a block, the more likely they will have an expensive house.

The second learning algorithm used was j4.8. This is the scheme we were hoping to compare with our learning scheme, which basically implements the same algorithm. Some of the default setting of Weka for this algorithm should be noted.

- Confidence threshold for pruning is 0.25.
- No reduced error pruning.
- Minimum number of instances in a leaf is 2.

Please refer to Appendix B for an excerpt of Weka results.

As with 1R learning scheme, the j4.8 does suffer from over fitting. An example would be the following.

b2 <= 2.3725: -1.1100000000000000e+002 (42.0/3.0)

In this case, a block with less median income has more expensive housing!

The over-fitting is not so detrimental, however. Most of what would seem logical is mirrored in the tree obtained (e.g. high incomes \leadsto expensive houses).

To obtain a simpler decision tree, we use J4.8 with different values for the parameters. Reduced-error pruning is used with two as the number of folds. The minimum number of instances in a leaf is set to 300. This does not increase error much; however, the tree does become much simpler.

6 **ANALYSIS OF OUR CODE**

6.1 TESTING

For testing we came up with a pseudo Matlab code which replicated the decision tree determined by our code. A sample of this code is presented below. This code is basically a set of rules and for each rule it specifies the class of houses it might fall under, cheap or expensive, that is.

Thus given the values for eight attributes, the code shall (with an error factor) give out the class (cheap or expensive) it belongs to.

The tree, while correct, is not very presentable to the user. A small chunk of it has been added here for inspection. Some explanation has appended to each line in order to clarify the output. Please note that the following output is of tree.m and not the error analysis code.

nodenum = 1

The node number in the tree. Due to recursion, all left nodes at each split are numbered before the right ones.

remainingattributes =

2 3 4 5 6 7 8 9

chosenatt =

2

The attribute chosen to split on next.

sizeR =

1410 9

Size of the right child node after split.

sizeL =

3590 9

Size of the left child node after split.

nodenum = 2

remainingattributes = 3 4 5 6 7 8 9

chosenatt = 8

sizeR = 671 9

sizeL = 2919 9

nodenum = 3

remainingattributes = 3 4 5 6 7 9

chosenatt = 7

sizeR = 720 9

sizeL = 2199 9

This tree gives a list of all the leaf nodes where the actual decision is made along with giving all these other nodes with the respective splits.

The same tree is mirrored by rules in our test code, a small portion of which is attached here:

```
function test(attribute)
```

```
a2=attribute(1,1) a3=
attribute(2,1)
a4=attribute(3,1) a5=
attribute(4,1)
a6=attribute(5,1) a7=
attribute(6,1)
a8=attribute(7,1) a9=
attribute(8,1) if(a2<
4.1 & a8<38 &
a7<590 & a3<25 &
a9<122 & a4<2164
& a5<402 &
a6<1347) disp '
THE HOUSE WILL
BE CHEAP '
```

```
elseif (a2< 4.1 & a8<38 & a7<590 & a3<25 & a9<122 & a4<2164 & a5<402 &
a6>=1347)
```



```

disp ' THE HOUSE WILL BE CHEAP '

elseif(a2< 4.1 & a8<38 & a7<590 & a3<25 & a9<122 & a4<2164 & a5>=402 &
a6<1347)
disp ' THE HOUSE WILL BE CHEAP '

elseif(a2< 4.1 & a8<38 & a7<590 & a3<25 & a9<122 & a4<2164 & a5>=402 &
a6>=1347)
disp ' THE HOUSE WILL BE CHEAP '

elseif(a2< 4.1 & a8<38 & a7<590 & a3<25 & a9<122 & a4>=2164 & a6<1347)
disp ' THE HOUSE WILL BE CHEAP '

```

For each traversal to a leaf in the tree there is a rule in the code.

For implementing this tree, some pre-pruning was used which stopped splitting if there were less than 50 instances in the node. There, however, was no criteria for the case where upon a split all instances from the parent node end up in one of the children nodes and the other being empty.

e.g.

```
nodenum = 16
```

```
remainingattributes = 5
```

```
chosenatt = 5
```

```
sizeR = 50 9
```

```
sizeL = 0 0
```

All instances are split into one child node thus making our learning algorithm no better. This small drawback is more or less covered in the test code with some manual postpruning.

A message like:

```
elseif(a2< 4.1 & a8<38 & a7<590 & a3<25 & a9<122 & a4>=2164 & a6>1347 a5<402)
disp ' No instances found with this path traversal to make a judgment'
```

6.2 ERROR RATE

Finding the error rate of our learning algorithm was subdivided into two parts. First, was the error ratio using instances used in the training set. Second was using a separate set for testing. It is interesting that the error rate using instances used in training can be found by counting the number of instances that have been wrongly grouped into a leaf during the phase that approximates the class of a leaf by its majority class.

To find the error rate using instances not used in training we **had** to use the code that was described in section 6.1.

Upon request decision tree rules, code used for testing (which is basically hundreds of nearly identical lines of code specifying rules that correspond to decision tree) and the complete list of nodes of the decision tree could be provided. They are not included here, because they constitute tens of pages of results.

The success rate of our learning algorithm is quite high.

Before stating success rates we shall discuss the method of coming up with these rates (probabilities) with a 90% confidence interval.

The success rate shall be denoted by f .

f is actually calculated by dividing the total number of successes by the total number of instances. Thus for S successes in N trials $f = S/N$

The variance of the trial is calculated using $f(1-f)$.

The interval for f with confidence 90% is calculated by:

$$P = (f + (z^2)/2N \pm z \cdot (f/N - f^2/N + z^2/4N^2)^{1/2}) / (1 + z^2/N)$$

Where z for 90% confidence is: 1.65.

For testing with untrained instances $P = [0.9338, 0.9394]$ with $f = 0.9366$.

For testing with trained instances $P = [0.9837, 0.9875]$ with $f = 0.9856$.

6.3 COMPARING ALTERNATIVES: WEKA VS. OURS

It is not enough for us to pick the training scheme that simply has the lowest error rate. Even though the data set is large, it would be more reliable and indeed necessary to compare confidence intervals with a certain confidence limit.

The two schemes that will be compared are our C4.5 code vs. Weka C4.5 (i.e. J4.8).

The test instances will be those not used in training. For each scheme a set of samples was obtained using successive tenfold cross-validations. Ten of such tests were conducted to find the average error rate for each scheme.

Since the training sets for Weka were picked by Weka and our training sets picked by us manually, then an *unpaired* t-test is necessary to compare these two schemes. The difference between unpaired and paired is in the calculation of the variance of the difference.

From our calculations the difference of the means comes out to be 0.87 and the estimate of the variance of the difference comes out to be 0.089. Therefore using the formula for t found on page 131 of Witten/Frank the t value is 29.29. For a confidence level of 10% and 9 degrees of freedom we can conclude that Weka outperforms our code (i.e. we reject the null hypothesis, since the value for t is larger than that of z -1.83 for two tailed test- for the chosen confidence level).

7 CONCLUSIONS

We coded C4.5 and were satisfied with the results obtained. For the amount of simplification it faired very well against the Weka counterpart. Furthermore, various techniques for improving the performance of the algorithms were also discussed and utilized.

For Weka, various data mining algorithms was used and analyzed. After testing the various learning algorithms, including our own code, one can confidently say that data mining can be used to successfully predict house values in California from the nine attributes described in the beginning of our paper.

One final note: In order to live in an expensive house, ask for more money from your employer or institution that employs you (e.g. McMaster University)! Our results reaffirm the age old notion that the attribute that defines most the price of your house, is the salary you get.

8 REFERENCES

1. Author Unknown, *StatLib---Datasets Archive* [Online], Available: < <http://lib.stat.cmu.edu/datasets/>> [Accessed 1 November 2002]
2. Pace R. K. and Ronald B. *Sparse Spatial Auto-regressions* , 33 (1997) 291-297.
3. Witten I. H. and Frank I. *Data Mining* , Morgan Kaufman Publishers, San Francisco, 2000.
4. Author Unknown, *What is Cluster Analysis?* [Online], Available: < http://www.clustan.com/what_is_cluster_analysis.html> [Accessed 24 November 2002].
5. Peng, Jiming, *Introduction to Machine Learning and Data Mining, SFWR 4TF3 McMaster course website* [Online], Available: <<http://www.cas.mcmaster.ca/~cs4tf3>> [Accessed 6 September 2002].

9 APPENDICES

APPENDIX A: 1R RESULTS

b2:

```
< 2.4175      -> cheap
< 2.419850000000003      -> expensive
< 2.5782499999999997      -> cheap
< 2.58005      -> expensive
< 2.6191500000000003      -> cheap
< 2.62235      -> expensive
< 2.70335      -> cheap
< 2.70525      -> expensive
```

....

....(If a detailed output is requested one can be provided)

....

```

< 4.66445    -> expensive
< 4.67065    -> cheap
< 4.7949     -> expensive
< 4.7993000000000001    -> cheap
< 4.8134999999999994    -> expensive
< 4.8218999999999999    -> cheap
< 4.8264999999999999    -> expensive
< 4.82995     -> cheap
< 4.85175     -> expensive
< 4.8548      -> cheap
< 4.9430499999999995    -> expensive
< 4.94745     -> cheap
>= 4.94745    -> expensive

```

APPENDIX B

(Note: -110 represents cheap, while -111 represents expensive)

```

> java weka.classifiers.j48.J48 -t
z:\cs4tf~34\weka\original.arff -c 1

```

J48 pruned tree

```

b2 <= 4.119
| h8 <= 37.92
| | i9 <= -121.87
| | | b2 <= 2.6673
| | | | i9 <= -122.37
| | | | | b2 <= 2.1513
| | | | | c3 <= 25
| | | | | c3 <= 15: -1.1100000000000000e+002 (2.0)
| | | | | c3 > 15: -1.1000000000000000e+002 (5.0)

```

(this is only an excerpt for the Weka output when C4.5 is used).

```

| | | | | | | | | | b2 <= 5.5798: -1.1000000000000000e+002 (3.0)
| | | | | | | | | | b2 > 5.5798: -1.1100000000000000e+002 (3.0/1.0)

```

```

| | | | | | | | f6 > 1470: -1.1000000000000000e+002 (3.0)
| | | | | | | | b2 > 5.7251: -1.1100000000000000e+002 (21.0/1.0)
| | | | | e5 > 627: -1.1100000000000000e+002 (17.0/1.0)
| | | | b2 > 6.2685
| | | | i9 <= -121.7
| | | | | i9 <= -122: -1.1100000000000000e+002 (3.0)
| | | | | i9 > -122: -1.1000000000000000e+002 (2.0)
| | | | | i9 > -121.7: -1.1100000000000000e+002

```

=== Confusion Matrix ===

```

      a      b  <-- classified as
9526  776 |      a = -1.1100000000000000e+002
612 9726 |      b = -1.1000000000000000e+002 ===
Stratified cross-validation ===

```

Correctly Classified Instances	18189	88.125 %
Incorrectly Classified Instances	2451	
11.875 %		
Kappa statistic	0.7625	
Mean absolute error	0.153	
Root mean squared error	0.3152	
Relative absolute error	30.6047 %	
Root relative squared error	63.0497 %	
Total Number of Instances	20640	

=== Confusion Matrix ===

```

      a      b  <-- classified
as
9012 1290 |      a = -1.1100000000000000e+002
1161 9177 |      b = -1.1000000000000000e+002

```

APPENDIX

C: MISSING01.M

```
%removing some element to simulate %the
effect of missing values.

clear
load originaldata.txt    %loads data.txt this is the original data file

data = originaldata;    % later modification

NumOfRands=rand(1); %assigns the number of missing variables in the data set
NumOfRands=ceil(NumOfRands*50); %assigns the maximum number of missing values to
    be up to 2000 values (or up to 10% of our data set)

%randomly assigns a '-999' to missing values
for i=1:NumOfRands    num1= rand(1);
    num1 = num1*20640;    num1 = ceil(num1);
    num2= rand(1);    num2 = num2*5;
        num2 = ceil(num2);

        data(num1, num2) = -999;
end

%saves the data to a file called 'data1.txt' save
'data1.txt' data -ASCII -double
```

D: REPLACE_MISSING02.M

```
%This function deals with Missing values, and substitutes the appropriate value to deal with
these missing values
clear load
data1.txt

SizeData=size(data1);
ArrayCol=SizeData(1,1);
ArrayRow=SizeData(1,2);
ColTotal=0; ColAvg=0;
NumOfRands=0;
ColTotal=zeros(ArrayRow,1);
```


APPENDIX

```
%goes through the entire array and calculates the totals of each column (but skips the
missing values)
for i=2:ArrayRow    for
j=1:ArrayCol        if
data1(j,i) ~= -999;
    ColTotal(i)=ColTotal(i) + data1(j,i);
end
end end

%the average is calculated by using the totals that were calculated dividing by the number of
valid entries in each column [x,y]=find(data1==-999); a=[x,y];
SizeOfA=size(a);
NumOfRands=SizeOfA(1,1);
ColAvg=zeros(ArrayRow,1); for
k=2:ArrayRow
    ColAvg(k)=ColTotal(k)/(ArrayCol-NumOfRands); end

%Replace the missing values with the column averages

for k=1:NumOfRands
i=a(k,1);    j=a(k,2);
if j ~= 1
    data1(i,j) = ColAvg(j);
end end
%saves the file as 'data2.txt' save
'data2.txt' data1 -ASCII -double

DELETE ('data1.txt');
%data1 = [];
%save 'data1.txt' data1 -ASCII -double
```

E: NOMINALIZE03.M

```
clear;

load data2.txt; %counter
values
m=0; a=0; b=0; c=0;

% -660, -661, -662 are categories in the missing values file
for (i = 1:20640)    if (data2(i,1)== -999)        m=m+1;
%    elseif data2(i,1) < 140000
%        data2(i,1) = -660; %
a=a+1;
%    elseif data2(i,1) < 230000
%        data2(i,1) = -661;
```

APPENDIX

```
%      b=b+1;
%      else
%      data2(i,1)= -662; %      c=c+1;
elseif data2(i,1) < 180000 % NEW
data2(i,1) = -110;% cheap      a=a+1;
      else
          data2(i,1)= -111;%expensive
b=b+1;      end %if          %
NEW
end %for
m %1844 a
%6184  b
%6346 %c
%6266
```

```
save 'data3.txt' data2 -ASCII -double
```

```
DELETE ('data2.txt');
%data2 = [];
%save 'data2.txt' data2 -ASCII -double
```

F: CATEGORIZE04.M

%this program takes the frequency of each of the nominal values
%-660, -661 and replaces the missing values (-999) with the
%nominal value of the maximum frequency %in
this case, it turns out to be -661.

```
clear; load
data3.txt;
```

```
%counters for nominal values -660, -661, -662 respectively a=0;
b=0;
```

```
% -999 is missing value
%counting frequency of nominal values -660, -661, -662
for (i = 1:20640) if data3(i,1) == -110      a=a+1;
elseif data3(i,1) == -111      b=b+1;      end; %if end; a
b
```

```
%finding the maximum nominal value
if      (a>b)
rep=-110;
else    rep = -
111;
end %if
```

APPENDIX

```
%replace -999 with max nominal value
for (i = 1:20640)    if data3(i,1)== -999
data3(i,1)=rep;
    end; %if
end; %for
```

```
save 'data4.txt' data3 -ASCII -double
```

```
delete('data3.txt');
```

G: OUTLIER05.M

```
clear; load
data4.txt;
```

```
m = mean ( data4 );
s = std( data4 );
```

```
sdb = 2* s ;
```

```
upp = m + sdb; low
= m - sdb;
count = 0;
```

```
for ( j = 2:8)
```

```
    count = 0;    sz =
size ( data4);    szz
= sz(1);    u= upp(j);
    l=low(j);
    j
    for ( i = 1: szz )
```

```
        if ( data4(i - count,j) > u ) % if value is more than 2 std. away
data4(i-count,:) = [] ;% delete row        count = count +1;
```

```
        elseif ( data4(i - count,j) < l )
data4(i-count,:) = [] ;        count =
count +1;        end
```

```
end
```

APPENDIX

end

save 'data5.txt' data4 -ASCII -double

H: SPLIT.M

```
clear
load 'original_nominalized.txt'

data5=original_nominalized;

sz=size(data5);
sz=sz(1);           %get the number of rows

for(big=2:9)         %running through all columns except the first one (being a class) big
    sortedData=sortrows(data5,big);           %sort according to attribute (column) big
    totalCheapNum=0;
    totalExpensiveNum=0;
    j=1;

    for(i=1:sz(1)-1)
        if(sortedData(i,1)==-110) %for total cheap houses
            totalCheapNum=totalCheapNum+1;
        else %for total expensive houses
            totalExpensiveNum=totalExpensiveNum+1;
        end
    end

    if(sortedData(i+1,1)==-110) %for the last instance
        totalCheapNum=totalCheapNum+1;
    else
        totalExpensiveNum=totalExpensiveNum+1;
    end

    for(i=1:sz(1)-1) %find how many possible split points and the respective
        number

        if(sortedData(i,1)~=sortedData(i+1,1))
            splitPoints(1,j)=i;
        j=j+1;
    end
```

APPENDIX

```
        end

        cnt=1;

for(i=1:j-1)          %for the number of split
points

    cheapNumL(1,cnt)=0;
    expensiveNumL(1,cnt)=0;

    for(k=1:splitPoints(1,i))
        %for each split point, the number of instances before the point

        if(sortedData(k,1)==-110)
            cheapNumL(1,cnt)=cheapNumL(1,cnt)+1;
        else
```

```

        expensiveNumL(1,cnt)=expensiveNumL(1,cnt)+1;
    end    end

    cheapNumR(1,cnt)=totalCheapNum-cheapNumL(1,cnt);

    expensiveNumR(1,cnt)=totalExpensiveNum-expensiveNumL(1,cnt);
    cnt=cnt+1;
end

%Call info here in a loop with cheap, expensive L and
R as parameters

for(i=1:cnt-1)    %for each split point get the info from the info function
    inf(1,i) = info(cheapNumL(1,i),
        expensiveNumL(1,i), cheapNumR(1,i), expensiveNumR(1,i)); end

    min=1000; index=0;
    for(i=1:i-1)    %calculate the minimum

        if(min>inf(1,i))
            min=inf(1,i);    index=i;
        end
    end

    %with splitting value being < and > (sortedData(splitPoints(1,index),1)
    +
    sortedData(splitPoints(1,index)+1,1))/2

    value(big)=sortedData(splitPoints(1,index),big)+sortedData(splitPoints(1,index)+1,big);
    value(big)=value(big)/2;    %value
    contains the avg. at all 8 splitting points

end

```

APPENDIX I: INFO.M

```

function BITS = info ( a,b,c,d)
% calculates information for a split of two
% 1- assuming no more than a split of 2
% 2- at this point and to increase speed it is assumed that none of the
% variables are equal to 0
% 3- at this point the gain and gain ratio will not be used.
% And that is to increase speed. If splits are too biased towards the ends %
then gain ratio will be incorporated.
% Note: Nov 22, 2002 --- Splits do not produce single element trees

```

```

Sum = a+b+c+d;
RightSum = c+d;
LeftSum = a+b;

if ( a*b*c*d ~= 0 )
    RightInfo= (-c/RightSum)*log2(c/RightSum) - (d/RightSum)*log2(d/RightSum);
    LeftInfo = (-a/LeftSum) *log2(a/LeftSum) - (b/LeftSum)*log2(b/LeftSum);
    BITS = (LeftSum/Sum) *LeftInfo + (RightSum/Sum) * RightInfo; else BITS = 1; %
    So that program doesn't crash, if any arguments are 0          % then maximum
    info (which will not make it a favorable choice)
        % is allotted
        % Note: Nov 22, 2002 --- No empty (smallest has 100 instances) trees are created
        % If statement may be removed end

```

APPENDIX J: TREE.M

```

function BITS = info ( a,b,c,d)

% calculates info for the split of two

% 1- assuming no more than a split of 2

Sum = a+b+c+d;
RightSum = c+d;
LeftSum = a+b;

if ( a*b*c*d ~= 0 )
    RightInfo= (-c/RightSum)*log2(c/RightSum) - (d/RightSum)*log2(d/RightSum);
    LeftInfo = (-a/LeftSum) *log2(a/LeftSum) - (b/LeftSum)*log2(b/LeftSum);

    BITS = (LeftSum/Sum) *LeftInfo + (RightSum/Sum) *
    RightInfo;

else BITS = 1;

end

```

APPENDIX K: LINEAR REGRESSION RESULTS

```
> java weka.classifiers.LinearRegression -t
z:\cs4tf~34\cleanup\weka_first_data.arff -c 1
```

Linear Regression Model

a1

=

```
40248.5143 * b2 +
1156.3039 * c3 +
-8.1816 * d4 +
113.4107 * e5 +
-38.5351 * f6 +
48.3083 * g7 +
-42576.7216 * h8 +
-42823.7435 * i9 +
```

-3594022.9115

Time taken to build model: 2.66 seconds

Time taken to test model on training data: 7.78 seconds

=== Error on training data ===

Correlation coefficient	0.7982
Mean absolute error	50765.5451
Root mean squared error	69513.4603
Relative absolute error	55.682 %
Root relative squared error	60.2407 %
Total Number of Instances	20640

=== Cross-validation ===

Correlation coefficient	0.7975
Mean absolute error	50795.481
Root mean squared error	69614.4715
Relative absolute error	55.7134 %
Root relative squared error	60.3265 %
Total Number of Instances	20640

APPENDIX L: ATTRIBUTE SELECTION

```
> java weka.attributeSelection.GainRatioAttributeEval -I
z:\cs4tf~34\weka\original.arff -C 1
```

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

Search Method:

Attribute ranking.

Attribute Evaluator (supervised, Class (nominal): 1 value):

Gain Ratio feature evaluator

Ranked attributes:

0.0881	2	b2
0.04626	9	i9
0.04171	8	h8
0.01142	4	d4
0.00554	7	g7
0.00505	3	c3
0.00474	6	f6
0.00427	5	e5

Selected attributes: 2,9,8,4,7,3,6,5 : 8

APPENDIX M: ZERO R

```
> java weka.classifiers.ZeroR -t z:\cs4tf~34\weka\original.arff -c 1
```

ZeroR predicts class value: -1.1000000000000000e+002

Time taken to build model: 0.04 seconds

Time taken to test model on training data: 0.16 seconds

```

=== Error on training data ===
Correctly Classified Instances      10338      50.0872 %
Incorrectly Classified Instances    10302      49.9128 %

Kappa statistic                     0
Mean absolute error                 0.5
Root mean squared error             0.5

Relative absolute error             100      %
Root relative squared error         100      %

Total Number of Instances          20640

=== Confusion Matrix ===          a
b  <-- classified as

    0 10302 |      a = -1.1100000000000000e+002
    0 10338 |      b = -1.1000000000000000e+002

=== Stratified cross-validation ===
Correctly Classified Instances      10338      50.0872 %
Incorrectly Classified Instances    10302      49.9128 %

Kappa statistic                     0
Mean absolute error                 0.5
Root mean squared error             0.5

Relative absolute error             100      %
Root relative squared error         100      %

Total Number of Instances          20640

=== Confusion Matrix ===          a
b  <-- classified as

    0 10302 |      a = -1.1100000000000000e+002
    0 10338 |      b = -1.1000000000000000e+002

```

APPENDIX N: SMO

```
> java weka.classifiers.SMO -t z:\cs4tf~34\weka\original.arff -c 1
```

An object is already running, use "break" to interrupt it.

Machine linear: showing attribute weights, not support vectors.

```

-11.809135573489048 * b2
+ -0.9623555831020517 * c3

```

```

+ 4.083380972597465 * d4

+ -8.511076243196946 * e5

+ 16.72025372141259 * f6

+ -5.601426887217788 * g7
+ 11.065738077480063 * h8
+ 11.782807099720333 * i9

- 5.762529355827576

```

Number of support vectors: 9234

Number of kernel evaluations: 10490458

Time taken to build model: 30.67 seconds

Time taken to test model on training data: 0.35 seconds

=== Error on training data ===

Correctly Classified Instances	17194	83.3043 %
Incorrectly Classified Instances	3446	16.6957 %

Kappa statistic	0.6661
Mean absolute error	0.281
Root mean squared error	0.3509

Relative absolute error	56.1907 %
Root relative squared error	70.1753 %

Total Number of Instances	20640
---------------------------	-------

```

=== Confusion Matrix ===      a
b  <-- classified as

```

```

8551 1751 |      a = -1.1100000000000000e+002

```

```

1695 8643 |      b = -1.1000000000000000e+002

```

=== Stratified cross-validation ===

Correctly Classified Instances	17191	83.2897 %
Incorrectly Classified Instances	3449	16.7103 %

Kappa statistic	0.6658
Mean absolute error	0.2821

Root mean squared error	0.3515
Relative absolute error	56.4113 %
Root relative squared error	70.2936 %
Total Number of Instances	20640

=== Confusion Matrix === a

b <-- classified as

8553 1749	a = -1.1100000000000000e+002
1700 8638	b = -1.1000000000000000e+002