CSC 272 - Final Project

Predicting a Medical Diagnosis of Chronic Kidney Disease Among Patients

Augustus Hayfron, Brooks Musangu, Haena Chon
Dr. Kevin Treu | CSC-272 | April 22, 2016

1 Introduction

Chronic kidney disease (CKD) is a condition characterized by a gradual loss of kidney function over time. Chronic kidney disease means that your kidneys are damaged and can't filter blood as they should. This damage can cause wastes to build up in your body. It can also cause other problems that can harm your health. If kidney disease gets worse, wastes can build to high levels in your blood and make you feel sick. You may develop complications like high blood pressure, anemia (low blood count), weak bones, poor nutritional health and nerve damage. Also, kidney disease increases your risk of having heart and blood vessel disease. These problems may happen slowly over a long period of time.

Most often than not, chronic kidney disease is caused by life patterns and eating habits. These life habits of ours tend to affect other parts and chemical compositions of the body such as serum creatinine, potassium, sodium, just to mention a few. This project takes a dataset that was collected over a period of two months at a hospital on over 400 patients and attempts to predict the underlying factors that can classify a patient as to whether he/she has chronic kidney disease or not. We attempted to predict which patient has chronic kidney disease and which patient doesn't using several variations of our dataset.

2 Data Description

The dataset we analysed came from a medical diagnosis of over 400 patients that was collected at a hospital for over a period of two months. We obtained our data from the UCI Machine Learning Repository website at http://archive.ics.uci.edu/ml/datasets/Chronic_Kidney_Disease#. The dataset was provided by Dr.P.Soundarapandian who is a senior consultant Nephrologist at the Apollo hospital in India. The dataset was compressed in a zip file so we had it unzipped using an application called WinRar. Our dataset contained 400 unique instances and 25 attributes (11 of them being numeric and 14 nominal). 250 of the 400 (62.5%) instances were classified as Chronic kidney disease and 150 (37.5%) were classified as not chronic disease. There were quite a number of missing values in each attribute except for 4 attributes which had only one missing value and represented 0% of the instances in those attributes. Below is the list of the key attributes contained in our dataset along with the data type and description:

Attribute	Type	Description
age	numeric	age of patients

bp	numeric	blood pressure
sg	nominal	specific gravity
al	nominal	albumin
su	nominal	sugar
rbc	nominal	red blood cells
pc	nominal	pus cell
pcc	nominal	pus cell clumps
ba	nominal	bacteria
bgr	numeric	blood glucose random
bu	numeric	blood urea
sc	numeric	serum creatinine
sod	numeric	sodium
pot	numeric	potassium
hemo	numeric	hemoglobin
pcv	numeric	packed cell volume
wc	numeric	white blood cell count
rbcc	numeric	red blood cell count
htn	nominal	hypertension
dm	nominal	diabetes mellitus
cad	nominal	coronary artery disease
appet	nominal	appetite
pe	nominal	pedal edema
ane	nominal	anemia
class	nominal	class

3 Data Preparation

We took several steps to prepare the data for mining and analysis. Since our dataset contained a mix of numeric and nominal attributes we thought it wise to discretize our dataset, changing all the numeric values in the dataset to nominal values. We used Weka to perform an equal-length discretization of all the attributes. To test numerous hypotheses, we produced two versions of our dataset; We created a first data set from the discretized data where we kept the data with missing values as well as the outliers and extreme values, as missing values could still reveal interesting connections in the data. We created a second set from the discretized data where we used Weka's '*ReplaceMissingValues*' filter to replace all missing values in the dataset based on the modes and mean of the entire dataset. For the second dataset, we wanted to make sure there were no outliers and extreme values so in getting rid of these, we deployed Weka's '*InterQuartileRange*' filter which after applying to our dataset, created two new attributes in our dataset: one being the outlier attribute and the other being the extreme values attribute. The outlier and extreme value attribute both had instances of yes and no. The 'yes' instance denoted an instance in the dataset as being an outlier or an extreme value. The 'no' instance meant that those particular instances were not outliers or extreme values. For the outlier attribute that was generated, there were 57 'yes' instances while the extreme value attributes showed 17 extreme values. Now that we discovered the outliers and extreme values in the dataset, we had to get rid of them. To remove the outliers and extreme values, we deployed Weka's unsupervised 'RemoveWithValues' filter. The 'RemoveWithValues' filter removes instances according to the value of an attribute so in dealing with the outliers, we set the *attributeIndex* parameter to 26, which was the number assigned to the Outlier attribute in the dataset. We then set the *NominalIndex* parameter to 'last' so the filter could deal with/remove all the values of the instances that was last in the outlier attribute which in this case was the 'yes' instance. We applied the attribute to the dataset and this got rid of all the outliers generated by the *InterQuartileRange* filter. We repeated the same process for the ExtremeValues attribute and successfully got rid of all the extreme values in the dataset. Now that all the extreme values and outliers had been dealt with, we didn't need the outlier and extremeValue attribute sitting in our dataset so we use Weka's unsupervised 'Remove' filter to get rid of these attributes. After all the outliers and extreme values had been removed our the number of unique instances in our dataset came down to 326 out of 400 which represented 81.5% of our original dataset.

4 Data Analysis

Because our dataset had only 25 attributes and 400 instances, we thought it wise to use cross validation on our algorithms. Cross validation partitions a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set). It also helps reduce variability, by performing multiple rounds of cross-validation using different partitions, and then averaging the validation results. Though we knew cross validation was the way to go considering the size of our dataset, we went ahead to also split the dataset into a 70% training set and a 30% test set. The reason for this was that we wanted to see how much of a difference in model accuracy that was going to be generated between cross validation and train/test split for the size of the dataset that we had. In splitting our dataset into a training and test set, we used Weka's '*Resample*' filter to create a subsample of our dataset setting the '*noReplacement*' parameter to 'true', InvertSelection to false and the sampleSizePercent parameter to 70% representing the training set. The noReplacement parameter being set to true ensure that no instances were replaced in the subsample of the dataset produced. We applied the filter and this generated and this cut down the number of instances in dataset 1 to 280 representing 70% of the original number of instances in the actual dataset. We saved this dataset and named it dataset1train. To get our remaining 30% of the data which we were going to use for testing, we changed the *sampleSizePercent* parameter to 30% and kept the noReplacement parameter at true. This time we changed the InvertSelection parameter to true so the filter could create a subsample that did not have any of the instances included in the 70% subsample already created. We applied the filter once again and this generated 120 instances representing 30% of the number of instances in dataset 1. Now our next challenge was to go back and look at the training and test sets created to make sure that both instances of the class attribute (ckd, 'notckd') had been fairly represented in both subsamples but we quickly realized that this wasn't a challenge anymore because it seemed as though the 'Resample' filter in weka used for splitting the dataset stratified the subsamples by default. To make sure of this we looked at the total number of each instance of 'ckd' and 'notckd' in the original dataset and the subsamples created and compared the ratios. The original dataset had 400 instances with 250 of those being 'ckd' and 150 being 'notckd' representing a ratio of 1:1.6. After using the *Resample* filter to create the training set, we had a total of 280 instances, 173 being 'ckd' and 107 being 'notckd' which also represented a ration of 1:1.6. The test data subsample which generated 123 instances of 'ckd' and 73 instances of 'notckd', also represented a ratio of 1:1.6. With this result, we knew that the training and test set subsamples generated using the Resample filter was stratified. We repeated the same approach on our dataset2 which had missing values replaced, outliers and extreme values taken out and we had a ratio of 1:1.2. Once again, we obtained an almost similar ratio of 1:1.3 for the training set of dataset2 which had 183 instances of 'ckd' and 150 instances of 'notckd' and after creating a 70% subsample of it for training. The test data which had 94 instances of 'ckd' and 70 instances 'notckd' also had a ratio of 1:1.3 which is almost the same as the ratio generated for the actual data for dataset2. After our data split and resampling, we obtained the following dataset which was what we worked with:

Dataset1:	discretized but still with missing values, outliers and extreme values
Dataset1train:	70% training set of dataset1
Dataset1test:	30% test of dataset1
Dataset2:	discretized, missing values replaced, outliers and extreme values taken out
Dataset2train:	70% training set of dataset2
Dataset2test:	30% test set of dataset2

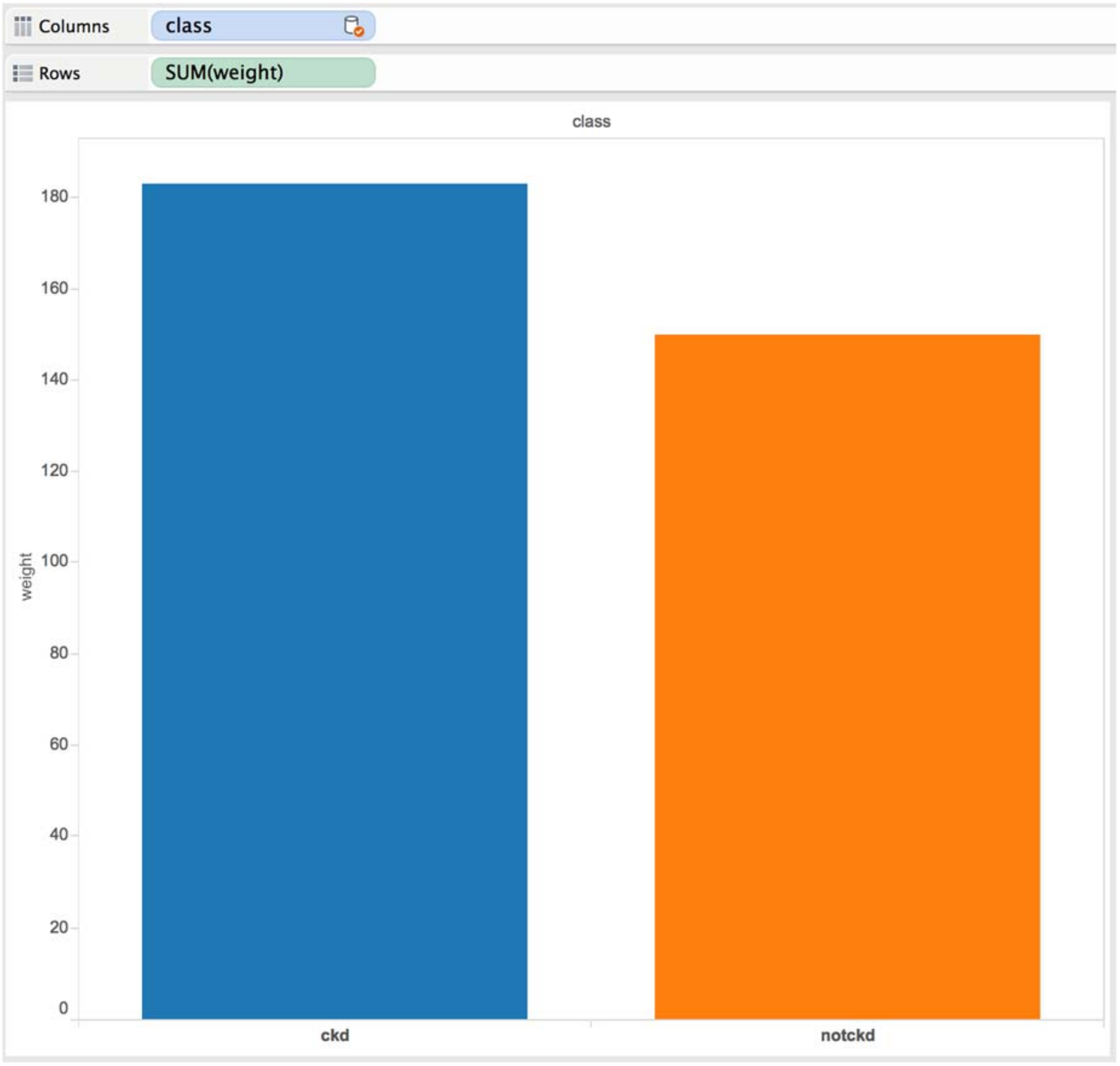
We used most of the learning algorithms in Weka, focusing on ZeroR, OneR, Naive Bayes, IBK and J48.

ZeroR is the simplest classification method which relies on the target and ignores all predictors. It simply predicts the majority category (class). Although there is no predictability power in ZeroR, it was useful for determining a baseline performance as a benchmark for other classification methods. We

run the ZeroR algorithm to make sure that one instance of class wasn't being represented too much in the whole dataset. The results generated by ZeroR helped us put our results in context. We kept in mind that if the 0R accuracy was too high, then we might want to create training data that is less skewed.
OneR as opposed to ZeroR, used the dataset to create rules based on one attribute that has the highest accuracy of classification.
The Naive Bayes classifier which is based on the Bayesian theorem regards all attributes as independent and equally important, whether they are or not then uses the conditional probabilities derived from the training data, to calculate the likelihood of an outcome being a particular class. The Naive Bayes is particularly suited when the dimensionality of the inputs is high. Despite its simplicity, Naive Bayes can often outperform more sophisticated classification methods.
The Nearest Neighbor classifier which is also know as IBK classifies a new sample by calculating the distance to the nearest training instance. As with other algorithms, getting the best accuracy using nearest neighbor depends on many other factors besides the mere cleaning of the data. In this project we were interested in two factors that are associated with nearest neighbor; the "best" subset of attributes and the "best" *k-value*. As we noticed in lab, coming up with these two factors can be very tedious. Therefore, we did some research on whether there are some automated ways of getting these factors in weka. We discovered that this is actually possible. To get the best subset of attributes we played with the *attribute selection* filter. A supervised attribute filter that can be used to select attributes. It is very flexible and allows various searches and evaluations. Under the evaluator options we used the *csfSubsetEval.* It evaluates the worth of subsets of attributes by considering the individual predictive abilities of each feature along with the the degree of redundancy between them. With this, we used the *GreedyStepwise* search option which allowed us to search through the space of attributes backwards and forwards. Upon applying this filter on both the data set 1 and 2, 12 attributes were selected and we got 95% and 97% accuracies respectively. We went further and set the *crossValidate* to true. This feature picks the best *k*-value from a given range of *k-values.* However, even though this filter was applied, the accuracies did not changed from the ones mentioned earlier. Nevertheless, it's quite a handy tool to have if you are only interested in picking the best *k-value* without having to go through all one to ten *k-values.* We did not use the models generated from both experiments owing to the fact that these models won't run on the test sets as they are incompatible.
The J48 which is also known as the decision tree is a predictive machine-learning model that decides the target value (dependent variable) of a new sample based on various attribute values of the available data. With the J48 classifier, the attribute that is to be predicted is known as the dependent variable, since its value depends upon, or is decided by, the values of all the other attributes. The other attributes, which help in predicting the value of the dependent variable, are known as the independent variables in the dataset. The J48 works by building a decision tree to accurately classify a new instance. For all these models, we run them with the ten-fold cross validation, against the 70% training set and 30% test set for both dataset 1 and 2. To see which model that produced the best results, we also decided to look at the confusion matrices of each algorithm, looking out for the one which produced the least number of false negatives and false positives.

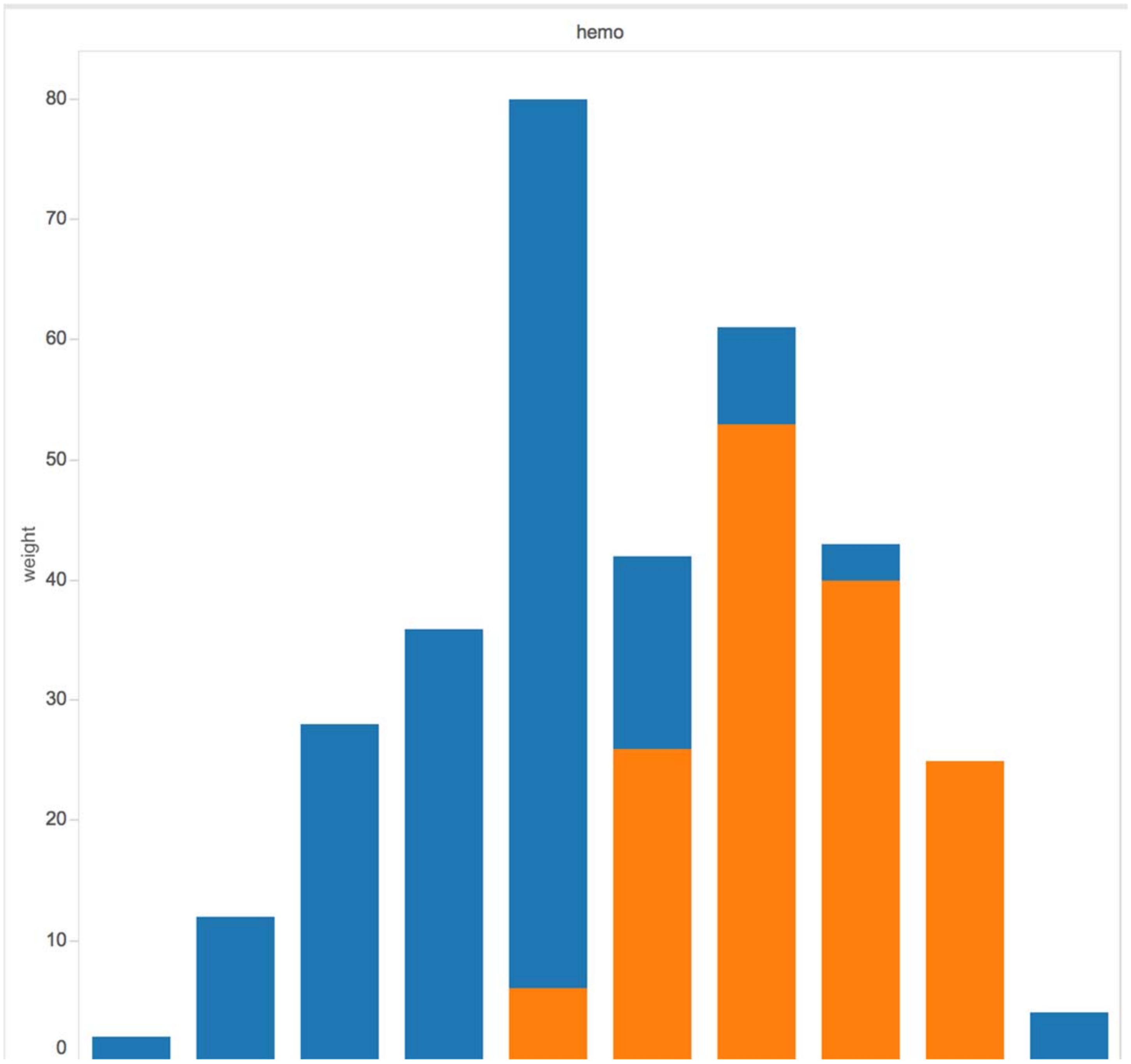
5 Results

We run the ZeroR algorithm to get a better sense of the proportion of attributes that we had in the class attribute. A result of 62.5% for dataset 1 and 59.96% for dataset 2 assured is that we had a good amount of each instance in the class attribute fairly represented.



Dataset has a fair distribution of both instances of the class attribute

Next we run the 1R on both datasets to get a sense of what attributes had the most predictive power. For dataset 1, the most predictive attribute after running OneR for the first time was hemoglobin. We therefore took hemoglobin out and re-run the OneR algorithm again and this gave us packed cell volume. Red blood cells was the next predictive attribute after taking packed cell volume out. For dataset 2 hemoglobin ranked first on OneR but red blood cells came second with serum creatinine ranking next as the third most predictive attribute. Here we saw a difference in the OneR ranking of both dataset 1 and 2. Packed cell volume was the second was predictive attribute but did not even make the first three in dataset 2. Our best guess was that, replacing the missing values in dataset 2 with the most frequent instance in each attribute that had a missing value gave other attributes more predictive power over packed cell volume.



OneR showing Hemoglobin as the highest single predictor of the class attribute
Blue field: chronic kidney disease
Orange field: not chronic kidney disease

Looking at the data run with cross validation, Naive Bayes and J48 seem to be on a tie. It makes sense for Naive Bayes and J48 to have a high accuracy with a dataset 1. J48 is a more complicated classification algorithm and Naive Bayes does a good job on handling missing values. Looking at the results for training set and test set for dataset 1, we realized that we obtained a better accuracy splitting the data into a training set and test set as compared to running cross validation.

Looking at the overall accuracies for both dataset 1 and 2 which were run with cross validation, it seems that our top two models were Naive Bayes and J48. Naive Bayes tied with J48 on dataset 1 but did better than J48 on dataset 2. Here, Naive Bayes, despite its simplicity, outperformed the more complicated and sophisticated J48 algorithm. It seemed that despite the small size of our dataset, splitting the dataset into a training set and test set produced higher results in accuracies but we trusted the results of cross validation more because of the simple fact that cross validation does 10 iterations and averages the result. The accuracies on the training and test split may have been higher than J48, but we suspected that these results were rather skewed. Below is a more detailed accuracy report on both datasets used for our analysis.

Looking at the results for Naive Bayes, it did a better job at classification in dataset 2 than in dataset 1. Because dataset 1 had quite a number of missing values, the naive bayes algorithm simply ignored them when it run and this led to a lower accuracy. Dataset 2 had its missing values replaced with instances that appeared more frequently on each attribute and this boosted the accuracy of dataset 2. The reason why the accuracy of naive bayes for dataset 1 may have been lower than dataset 2 could be because there were a lot of missing values, the algorithm did not have enough information to help in its accurate classification. For dataset 2, which has a somewhat higher accuracy than dataset1, they may be the issue of overfitting simply because, replacing the missing values with the most frequent instances may have given more credit to an attribute, leading to a skewed result. Also for the most part of naive bayes, we realized that increasing the K value led to a decreased accuracy. When the value of K was set to one, most often than not the accuracy was incredibly high with some models even hitting the 100% mark. Increasing the K value as we witnessed, was indirectly proportional to the accuracy obtained. Increasing the K value meant increasing the number of neighbors to compare the new instance to and this decreased the accuracy. The only exception to this was the cross validation set for dataset where the model produced an accuracy of 94.89% when the K value was set to 1, increased to 96.70% when set to 3 and dropped to 95.80% when set to 5. Our judgement was that, having just one neighbor to compare your new instance to may lead to incorrect classification and skewed results. It’s always best to rely on more than one neighbor for classification in Naive Bayes as this helps the model make a more accurate prediction that is less skewed.

Accuracies (%) for Dataset 1 (With Missing Values, Outliers and Extreme Values)

Classification Algorithms	**Cross Validation (Dataset1)**	**Training Data (Training set)**	**Test Data (30%)**
ZeroR	62.5	61.79	62.76
OneR	92	89.64	90.31
NaiveBayes	97.25	97.50	97.45

IBK	K=1=88.75
	K=3=84.5
	K=5=82.25
		K=1=100
		K=3=85.36
		K=5=81.07
			K=1=99.13
			K=3=86.73
			K=5=81.63

J48	97.25	98.2143	98.9796

Accuracies (%) for Dataset 2 (Missing values Replaced, Outliers and Extreme Values Removed)

Classification Algorithms	**Cross Validation (Dataset2)**	**Training Data (Training set)**	**Test Data (30%)**
ZeroR	54.96	56.65	57.32
OneR	99.10	88.84	82.32
NaiveBayes	99.40	99.70	99.39

IBK	K=1=94.89
	K=3=96.70
	K=5=95.80
		K=1=100
		K=3=96.14
		K=5=94.42
			K=1=100
			K=3=95.122
			K=5=93.90

J48	96.94	95.71	98.17

Cost Matrices

The model tries to predict whether a patient has chronic kidney disease or not chronic kidney disease. The false positive means that the patient does not have the disease, but they actually do. While a false negative means that the patient has the disease, but the model predicts that he or she does not have the disease. Subjecting this to critical analysis and scrutiny, we realize that the cost of a false negative was going to be much higher than the cost of a false positive. A patient who's classified as having a false positive chronic kidney disease would still be treated anyways, but the one who is classified as a false negative would not be treated. Even though he or she has the kidney disease, he or she would be not classified as having it. This could be harmful and dangerous

Ensemble Models

In Siegel's book, he talks about ensemble learning and how it was fostered in the Netflix contest to achieve an overall higher accuracy. Going by this we explored a new feature in weka that we hadn't used in class or lab yet:- the experimenter window. With all our labs this semester, we performed analysis using the explorer window in Weka and wanted to try our hands on a cool feature in the experimenter window of Weka. Using the experimenter window in Weka, we wanted to investigate whether we could even further improve upon the results of the J48 algorithm using ensembles methods.We experimented with three popular ensemble methods; boosting, bagging and stacking. **Boosting** is an ensemble method that starts out with a base classifier that is prepared on the training data. A second classifier is then created behind it to focus on the instances in the training data that the first classifier got wrong. The process continues to add classifiers until a limit is reached in the number of models or accuracy. **Bagging** creates separate samples of the training dataset and creates a classifier for each sample. The results of these multiple classifiers are then combined (such as averaged or majority voting). The trick is that each sample of the training dataset is different, giving each classifier that is trained, a subtly different focus and perspective on the problem.
Stacking combines multiple different algorithms prepared on the training data and a meta classifier is prepared that learns how to take the predictions of each classifier and make accurate predictions on unseen data.

The J48 is a powerful decision tree method that performs well on the Chronic kidney disease dataset. In this experiment we wanted to investigate whether we can improve upon the result of the J48 algorithm using ensemble methods. We tried three popular ensemble methods: Boosting, Bagging and Blending. We started out by adding the J48 algorithm to the experiment so that we could compare its results to the ensemble versions of the algorithm. Below are the steps we took in adding all the three forms of ensemble methods to compare to the J48 algorithm.

Boosting

We Clicked on "*Add new…*" in the "*Algorithms*" section.

Clicked the "*Choose*" button.

Clicked "*AdaBoostM1*" under the "*meta*" selection.

Clicked the "*Choose*" button for the "*classifier*" and select "*J48*" under the "*tree*" section and click the "*choose*" button.

Clicked the "*OK*" button on the "*AdaBoostM1*" configuration.

Bagging

Clicked "*Add new…*" in the "*Algorithms*" section.

Clicked the "*Choose*" button.

Clicked "*Bagging*" under the "meta" selection.
Clicked the "*Choose*" button for the "*classifier*" and select "*J48*" under the "*tree*" section and click the "*choose*" button.
Clicked the "*OK*" button on the "*Bagging*" configuration.

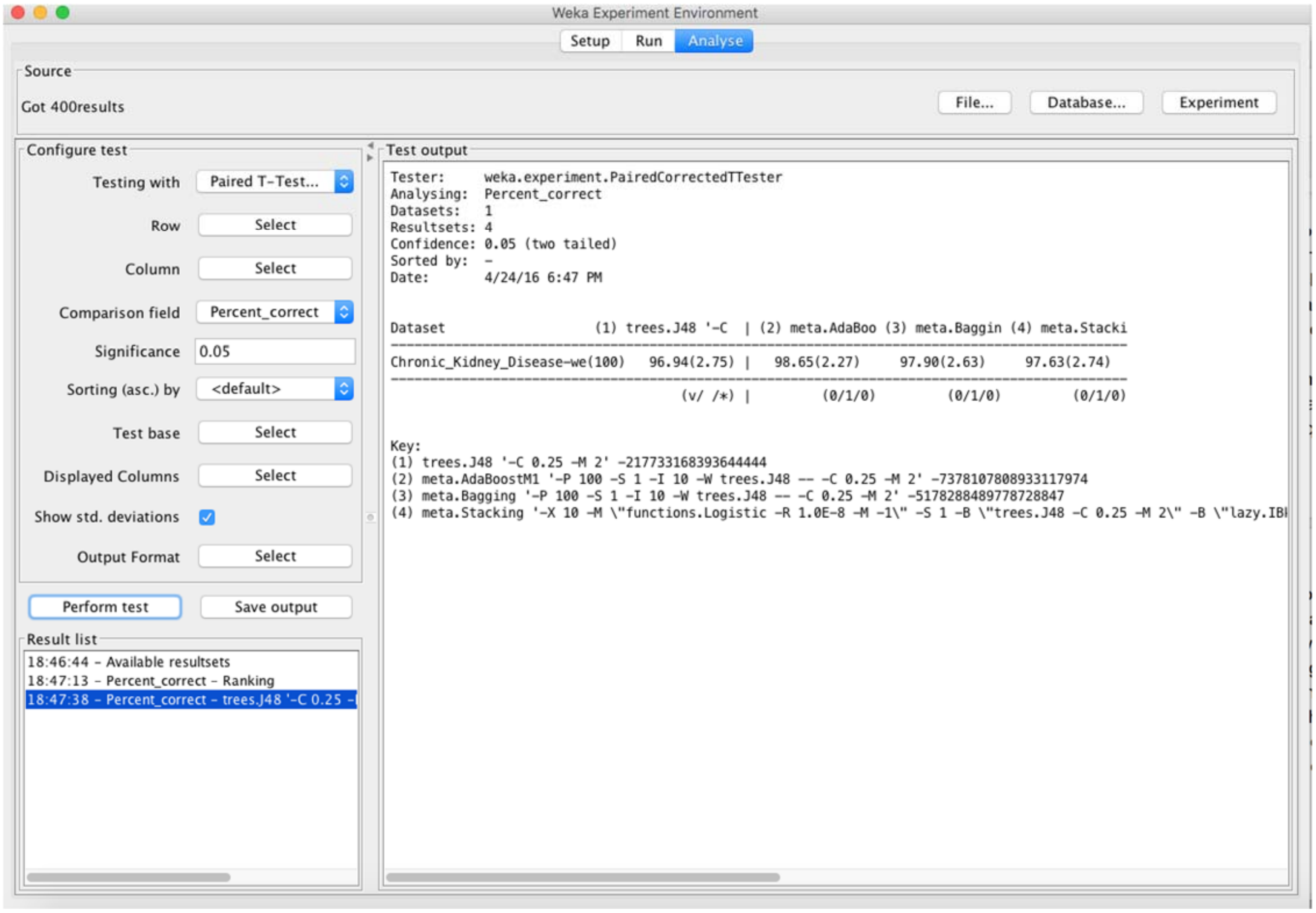
Stacking

Clicked "*Add new…*" in the "*Algorithms*" section
clicked the "*Choose*" button.
Clicked "*Stacking*" under the "*meta*" selection.
Clicked the "*Choose*" button for the "*metaClassifier* and selected "*Logistic*" under the "*function*" section and clicked the "*choose*" button.
Clicked the button for the "*classifiers*".
Clicked "*ZeroR*" and clicked the "*Delete*" button.
Clicked the "*Choose*" button for the "*classifier*" and selected "*J48*" under the "*tree*" section and click the "*Close*" button.
Clicked the "*Choose*" button for the "*classifier*" and select "*IBk*" under the "*lazy*" section and clicked the "*Close*" button.
Clicked the "*X*" to close the algorithm chooser
Clicked the "*OK*" button on the "*Bagging*" configuration.

These steps taken added the three forms of ensemble learning to our experimenter to be able to compare it against the J48 algorithm. For stacking, we stacked the J48 and the IBK algorithm so this method combined the predictive models of two different algorithms.

Ensemble models accuracy:

Next we wanted to know what scores the algorithms achieved so we clicked the "*Select*" button for the "*Test base*" and chose the "*J48*" algorithm in the list and clicked the "*Select*" button. We then checked the box next to "*Show std. deviations*" and then clicked the "*Perform test*" button. This produced the window below:



Source

Got 400results

Configure test

Testing with: Paired T-Test...
Row: Select
Column: Select
Comparison field: Percent_correct
Significance: 0.05
Sorting (asc.) by: <default>
Test base: Select
Displayed Columns: Select
Show std. deviations: ✓
Output Format: Select

Result list

```
18:46:44 - Available resultsets
18:47:13 - Percent_correct - Ranking
18:47:38 - Percent_correct - trees.J48 '-C 0.25 -
```

Test output

```
Tester:     weka.experiment.PairedCorrectedTTester
Analysing:  Percent_correct
Datasets:   1
Resultsets: 4
Confidence: 0.05 (two tailed)
Sorted by:  -
Date:       4/24/16 6:47 PM


Dataset                   (1) trees.J48 '-C  | (2) meta.AdaBoo (3) meta.Baggin (4) meta.Stacki
--------------------------------------------------------------------------------------------
Chronic_Kidney_Disease-we(100)   96.94(2.75) |   98.65(2.27)     97.90(2.63)     97.63(2.74)
--------------------------------------------------------------------------------------------
                                     (v/ /*) |         (0/1/0)         (0/1/0)         (0/1/0)


Key:
(1) trees.J48 '-C 0.25 -M 2' -217733168393644444
(2) meta.AdaBoostM1 '-P 100 -S 1 -I 10 -W trees.J48 -- -C 0.25 -M 2' -7378107808933117974
(3) meta.Bagging '-P 100 -S 1 -I 10 -W trees.J48 -- -C 0.25 -M 2' -5178288489778728847
(4) meta.Stacking '-X 10 -M \"functions.Logistic -R 1.0E-8 -M -1\" -S 1 -B \"trees.J48 -C 0.25 -M 2\" -B \"lazy.IB
```

We can see that the AdaBoostM1 algorithm achieved a classification accuracy of 98.65%. We can see that this a higher value than J48 at 96.94% (+ 1.71 %). We can see a “*” next to the accuracy of J48 in the table and this indicates that the difference between the boosted J48 algorithm is meaningful (statistically significant).We can also see that the AdaBoostM1 algorithm achieved a result with a value higher than Bagging at 97.90% , but we do not see a little “*” which indicates that the difference is not meaningful (not statistically significant). The same applies to stacking which produced a lower accuracy than bagging () but with no “*”, also meaning the difference is not statistically significant.

What we were also interested in was the ranking of these ensemble methods used. AdaBoostM1 version of J48 is ranked the highest against other algorithms. Bagging ranks second, Stacking third and plain old J48l last. It is a good sign that J48 is ranked low, it suggests that at least some of the ensemble methods have increased the accuracy on the problem.

6 Conclusion

Looking at our results, it seemed that J48 and Naive Bayes did a good job at accurately classifying our dataset with Naive Bayes beating J48 in some instances. This goes to affirm that sometimes you can achieve the best results with the simplest of algorithms. We found that hemoglobin levels and red blood cell counts were major predictors of a patient having chronic kidney disease. This makes sense as chronic kidney disease directly affects the production of red blood cells in your body and lowers hemoglobin levels which is exactly what our data showed. Ensemble learning methods are very good ways to boost the accuracy of your models as we found in our dataset experiment. We will however not consider our model error proof since the dataset came with quite a number of missing values and had to be replaced. Replacing the missing values with the instances that appear more frequently in each attribute, though convenient, might not be necessarily helpful at all times as this can lead to skewed results.

Appendix

Dataset: http://archive.ics.uci.edu/ml/datasets/Chronic_Kidney_Disease#
About Chronic Kidney Disease: https://www.kidney.org/kidneydisease/aboutckd