

CS545: Dr. Anderson's Online Section Assignment 4

John Cartmell

October 15, 2009

Contents

1	Problem Definition	1
2	Logistic Regression	2
2.1	Theory	2
2.2	Realization	3
2.3	Model Fidelity	4
3	K-Nearest Neighbors	6
3.1	Number of Operations	7
3.2	Realizations of KNN	7
3.3	Model Fidelity and Real Time Usage	8
3.4	Further Explorations	9
4	Parkinson's Data Analysis	11
4.1	Data Gathering and Model Development	12
4.2	Model Fidelity with 80% Training Fraction	12
4.3	Model Fidelity with Different Training Fractions	14
5	Wine Data Analysis	17
5.1	Data Gathering and Model Development	18
5.2	Model Fidelity with 80% Training Fraction	19
5.3	Model Fidelity with Different Training Fractions	19
6	Conclusion	23

1 Problem Definition

This assignment focused on developing the K-Nearest Neighbors (KNN) and Logical Regression (LR) methods for classification of data sets. Once these models are developed, they are applied to the data sets from the previous assignment, the Parkinson's data set which has been used during Dr. Anderson's lectures, and a data set of my choice. Given my affinity for wine, I used the wine data set. These models are also compared against the Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA) methods which were the focus of the previous assignment. In addition, the KNN method was implemented using different techniques to allow for a comparison of the relative efficiency of each implementation.

It should be noted that much of the code used in the assignment was based on the Dr. Anderson's solution for Assignment 3. I added significant portions to the code as required to make the case for the analysis provided for each data set. That code which I modified or created is included throughout this report. As I have done in previous reports, I remove the comments so that the code can be clearly seen.

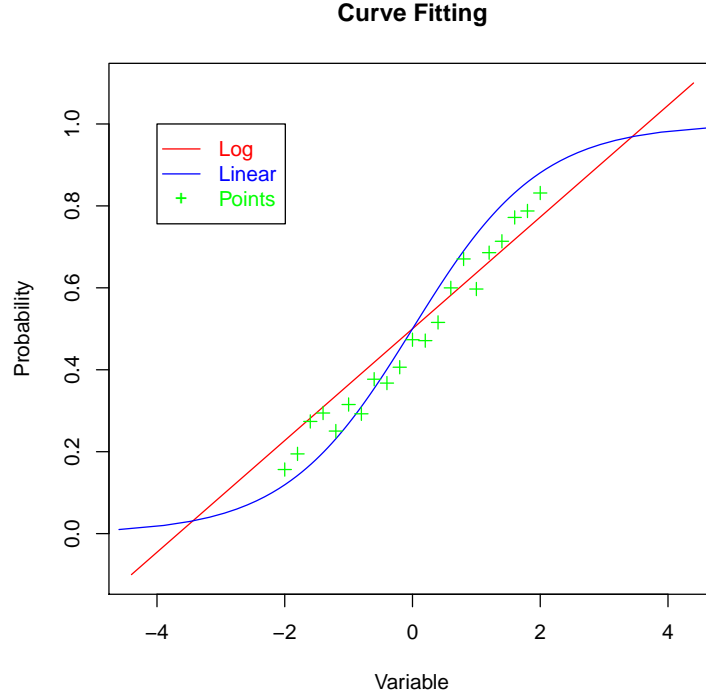


Figure 1: Linear or Logistic Regression Curve Fitting to Data Points

2 Logistic Regression

2.1 Theory

The motivation for Logistic Regression is that a linear model may result in the $P(C=k|x)$ of a particular class being masked. Should this masking occur for a particular class, when the classification is performed by choosing the maximum $P(C=k|x)$ for a particular sample across all the classes, the masked class will never be selected. To attempt to prevent this from occurring, Logistic Regression will force all probabilities to be between 0 and 1 and will force the sum of the probabilities to be 1. Logistic Regression is based on the Logistic Regression equation:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (1)$$

The great feature of this equation is that regardless of the input, x , $f(x)$ will always be between 0 and 1. In many cases, this function also suffices with regard to curve fitting. Given the points displayed in Figure 1, there are an infinite number of curves that can be used to curve fit this data. Two instances are shown, one linear, the other the Logistic Regression equation. Both fit the points reasonably well, however, the Logistic Regression function is bounded, never producing a value less than 0 or greater than 1. The Logistic Regression function can be tweaked so that the error between the fitted curve and the data points is minimized. Let us call the value used to tweak the Logistic Regression function β , therefore, the Logistic Regression function will become $f(x,\beta)$ and β will be unique for each class of the attribute being modeled. This yields:

$$p(C = k|x) = \frac{f(x, \beta_k)}{\sum_{m=1}^k f(x, \beta_m)} \quad \text{for all } k \quad (2)$$

which we will simplify to:

$$g(x, \beta_k) = \frac{f(x, \beta_k)}{\sum_{m=1}^k f(x, \beta_m)} \quad \text{for all } k \quad (3)$$

However, this does not take into account the requirement that the sum of the probabilities must be 1. To accomplish this the above is slightly modified as follows:

$$g(x, \beta_k) = \frac{f(x, \beta_k)}{\sum_{m=1}^k f(x, \beta_m)} \quad \text{for } k < K \quad (4)$$

and:

$$g(x, \beta_k) = \frac{1}{\sum_{m=1}^k f(x, \beta_m)} \quad \text{for } k = K \quad (5)$$

Using these equations, we can build a model based on Logistic Regression. Or I should say almost. How do we figure out the β values for each class within each attribute? We use a least likelihood method that will allow us to optimize each β . We would prefer to set the derivative to zero, however, that will not work given the gradient of the Logistic Regression equation is not linear. At this point, I would prefer to give up and use a simpler method, however, my goal is to get my paper classified at the highest level so I continue.

In order to solve for the set of optimal β values, we will use a technique called gradient descent [1]. There are many techniques to perform this, but the basic concept is to reach a local minimum by traversing along the curve that we wish to find the minimum of by computing the derivative. The gradient is computed at locations as the curve is traversed and when the derivative reaches zero we have found a minima. The algorithm will stop when either some number of iterations has expired without convergence, or the local minimum is found within a tolerance.

There are several techniques that can be used to find this minimum. In class, Dr. Anderson detailed Steepest Descent and Scaled Conjugate Gradient. He also briefly mentioned Newton-Raphson which has several advantages and disadvantages [2]. The method is very costly from a CPU utilization point of view and it does not always converge. However, when it does converge the order of converge is quadratic. Another method I found was Least Mean Square algorithm. This method is very similar to Steepest Descent but is easier to compute [3].

2.2 Realization

The Logistic Regression model was realized reusing R code provided by Dr. Anderson along with code developed by myself. Two functions were created in a similar fashion as was done for all the other models we developed, one to create the model with the training data and the other to use the model to classify a test data set. The first function is `makeLogReg` which creates the model and computes the optimized β values.

```
makeLogReg <- function(X,T, classes) {
  standardize <- makeStandardizeF(X)
  X <- standardize(X)
  X <- cbind(1,X)
  TI <- makeIndicatorVars(T)
  D <- ncol(X)
  K <- classes
  beta <- matrix(0,D,K-1)
  res <- scg(matrix(beta,D*(K-1),1), logregNegMaxLogL, logregGrad, X, TI,
             fPrecision=0.00001, nIterations=1000000, ftracep=TRUE)
  beta <- matrix(res$X,D,K-1)
}
```

This function accepts the training data and the classification values of the training data. It also accepts the number of classes. First, it will standardize the training data and add a bias column to the data. Next it will modify the classification of the training data from a scalar into $k-1$ boolean values where k is the number of classes. After this, it will create the β matrix. The first guess for the β values will be zero. The size of the β matrix will be based on the number of attributes and the number of classes. Once created, it will be sent into the scaled-conjugate gradient (scg) function which will compute the β values based on the training

data and the required precision. The number of iterations are also specified which allows the `scg` function to return should the iterations process not converge. The `scg` function used was provided by the professor. The only tricky part of this code which caused problems was related to the shape of the β matrix. For more than two classes, the β matrix has more than one column which causes problems within the `scg` function. Therefore, the matrix needs to be converted to a column vector before invoking the `scg` function and the resultant matrix needs to be converted back from a column matrix after the `scg` function has completed its processing.

The other function developed to create and use this model is `makeLogReg`. It is executed after the β matrix has been determined when one wants to classify a test data set. As inputs it accepts the data to be classified and the β values. Upon invocation, this function will standardize the data and add a bias column. Then it will compute the `g` function. The `g` function was provided by Dr. Anderson and is reused without change. It models the `g` function in Equations 4 and 5 above. Once this function executes, the maximum value is returned for each sample in the test data set. This is the classification as determined by the Logical Regression model.

```
useLogReg <- function(X, beta, probs=FALSE) {
  X <- standardize(X)
  X <- cbind(1, X)
  if (probs) {
    g(X, beta)
  }
  else {
    apply(g(X, beta), 1, which.max)
  }
}
```

2.3 Model Fidelity

The first two questions from Assignment 2 were rerun using the Logistic Regression model. This requires 4 sets of data:

- 1-dimensional data, segregated training classes - LR
- 2-dimensional data, segregated training classes - LR
- 1-dimensional data, integrated training classes - LR
- 2-dimensional data, integrated training classes - LR

For this data generation I reused the R source code provided by Dr. Anderson. However, instead of using the confusion technique, I changed the mean and standard deviation. For the 1-dimensional segregated data:

```
Xtrain <- matrix(c(rnorm(10, 1, 0.1), rnorm(10, 2, 0.1), rnorm(10, 3, 0.1)))
Ttrain <- matrix(c(rep(1, 10), rep(2, 10), rep(3, 10)))
Xtest <- matrix(seq(0, 4, len=100))
```

For the 1-dimensional integrated data, with the means and standard deviations modified to integrate the data:

```
Xtrain <- matrix(c(rnorm(10, 1, 0.5), rnorm(10, 2, 0.5), rnorm(10, 3, 0.5)))
Ttrain <- matrix(c(rep(1, 10), rep(2, 10), rep(3, 10)))
Xtest <- matrix(seq(0, 4, len=100))
```

For the 2-dimensional segregated data:

```
means <- list(matrix(c(2, 2, 3, 2), 2, 2, byrow=TRUE),
               matrix(c(3, 6, 2, 5), 2, 2, byrow=TRUE),
               matrix(c(8, 2, 7, 2), 2, 2, byrow=TRUE))
```

```

std <- 0.5
Xtrain <- NULL
for (class in 1:3) {
  mus <- means[[class]]
  Xtrain <- rbind(Xtrain, cbind(rnorm(10,mus[1,1],std), rnorm(10,mus[1,2],std)))
  Xtrain <- rbind(Xtrain, cbind(rnorm(10,mus[2,1],std), rnorm(10,mus[2,2],std)))
}
Ttrain <- matrix(c(rep(1,20),rep(2,20),rep(3,20)))

xs <- seq(0,10,len=40)
ys <- xs
Xtest <- as.matrix(expand.grid(xs,ys))

```

For the 2-dimensional integrated data, with the standard deviation modified to integrate the data:

```

means <- list(matrix(c(2,2, 3,2), 2,2,byrow=TRUE),
               matrix(c(3,6, 2,5), 2,2,byrow=TRUE),
               matrix(c(8,2, 7,2), 2,2,byrow=TRUE))

std <- 1.5
Xtrain <- NULL
for (class in 1:3) {
  mus <- means[[class]]
  Xtrain <- rbind(Xtrain, cbind(rnorm(10,mus[1,1],std), rnorm(10,mus[1,2],std)))
  Xtrain <- rbind(Xtrain, cbind(rnorm(10,mus[2,1],std), rnorm(10,mus[2,2],std)))
}
Ttrain <- matrix(c(rep(1,20),rep(2,20),rep(3,20)))

xs <- seq(0,10,len=40)
ys <- xs
Xtest <- as.matrix(expand.grid(xs,ys))

```

For the 1-dimensional data, there seems to be some problem with my implementation. I do not think it is the Logistic Regression model itself, I suspect it is what I am passing it. For both the segregated and non-segregated data, the training data looks correct, Figure 2. However, I expected to the $P(C=k|x)$ to each have areas where they dominated. That is not the case. Also, the predicted classes for the test set looks very bad. Given that it is midnight and I have been up since 4AM, I am going to bed and will try to correct this in the morning. I realize it will be late, but, well I do want to understand what is happening. The Logistic Regression model did appear to work for the harder data sets, so I am at a loss as to what occurred. The code used to plot this is as follows and is common between the two data sets:

```

x11()
par.orig <- par(mfrow=c(1,3))

logReg <- makeLogReg(X,T,3)
pred <- useLogReg(Xtest,logReg)

plot(X,T,type="p",col=T, xlab="x", ylab="Class")

posteriors <- useLogReg(Xtest,logReg,probs=TRUE)
matplot(Xtest,posteriors,type="l", lty=1,lwd=2, xlab="x", ylab="p( C=k | x )")

plot(Xtest,pred,type="p",col=pred,xlab="x",ylab="Predicted Class")

par(par.orig)
dev.copy2eps(file="part1d.eps")

```

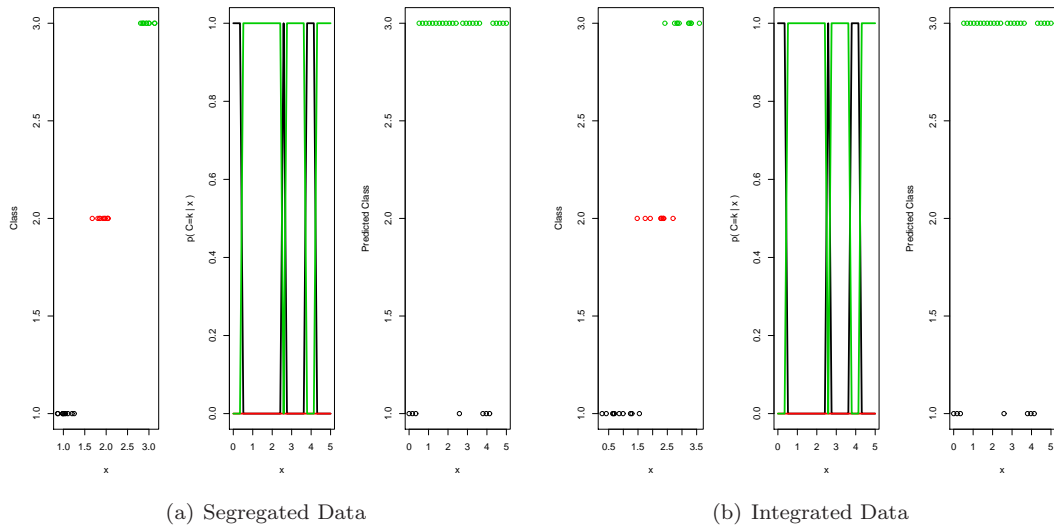


Figure 2: 1 Dimensional Data - Logistic Regression

The 2 dimension data sets exhibited the same problems. I am including the graphs, Figure 3 and the plotting code. I will debug it later. The code to create the model and the plots follows:

```

x11()
par.orig <- par(mfrow=c(1,3))

logReg <- makeLogReg(X,T,3)
pred <- useLogReg(X,logReg)

plot(X[,1],X[,2],col=T+1,pch=paste(T),xlab=expression(x[1]),ylab=expression(x[2]))

N <- length(xs)
colors <- 1+matrix(pred,N,N)[1:(N-1),1:(N-1)]
posterior <- useLogReg(Xtest,logReg,probs=TRUE)
surface <- matrix(apply(posterior,1,max),N,N)
persp(xs,ys,surface,xlab=expression(x[1]),ylab=expression(x[2]),zlab="P( C=k | x )",
col=colors,theta=-30,phi=40,border=NA,shade=0.3)

persp(xs,ys,matrix(pred,N,N),xlab=expression(x[1]),ylab=expression(x[2]),zlab="Predicted
col=colors,theta=-30,phi=40,border=NA,shade=0.3)

par(par.orig)
dev.copy2eps(file="part2b2.eps")

```

3 K-Nearest Neighbors

The K-Nearest Neighbors is a method that is used for classification. It is a unique classification method as the development of a model using KNN requires almost no effort. However, the model requires a significant real-time to classify a sample. When a sample is to be classified, it is compared to all other samples within the training set. The sample to be classified is assigned a classification that is the majority of nearest k-samples from the training set. If only one nearest neighbor is used, the classification of the new sample is

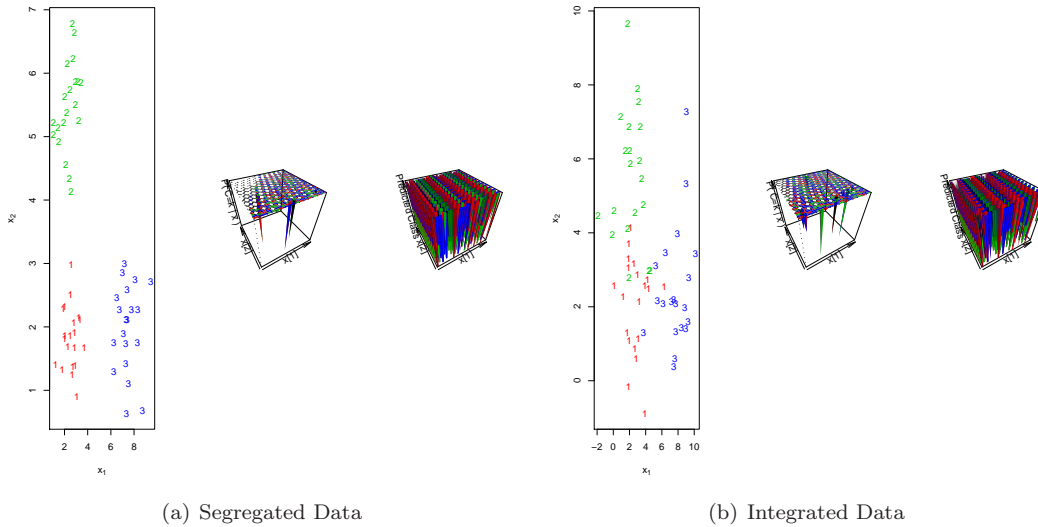


Figure 3: 1 Dimensional Data - Logistic Regression

the class of the single nearest neighbor. If more than one nearest neighbor is used, the majority of the class of these neighbors is the assigned class of the sample. Should there be a tie, it is implementation dependent which classification is selected. Of course, if it is a tie, one may question whether or not KNN is the proper method to use to classify for the data set in question.

3.1 Number of Operations

If there are n test samples and m training samples, then the number of combinations of n and m whose distance must be calculated are:

$$\gamma = n \times m \quad (6)$$

For each of these combinations, the differences of each attribute are computed, then squared and then summed. If there are p attributes in the data samples, then the number of attributes to be compared over both the test and training data sets are:

$$\omega = \gamma \times p \quad (7)$$

Table 1 summarizes the type and quantity of operations, based on the above assumptions. As can be seen in this table, KNN requires lots of processing across many samples!

Operation Type	Number of Operations
Subtraction	ω
Exponentiation	ω
Addition	γ
Comparison	γ

Table 1: Number of Operations and Type for KNN

3.2 Realizations of KNN

The KNN can be implemented in R by using matrices or using for loops. The matrix implementation was provided by Dr. Anderson. I developed the for loop implementation as required by this assignment. The matrix implementation uses matrices to classify all the test samples at once, determining the difference of

each test sample against all the training samples. Given that it was provided by Dr. Anderson, there is little value in discussing this further. The for loop implementation is comprised on two loops. The outer loop is used to traverse through each test set sample. The inner loop is used to compute that test set sample distance from each sample in the training set. For each sample in the test data set, once the differences are computed between it and every sample in the training set. The classification of the training data sample which has the minimum difference to the test data point is then assigned as the classification of the test data sample. A limitation of my KNN implementation is that k must be 1. The logic does not support k being greater than 1. Conversely the matrix version of KNN does allow for the number of neighbors to be any number.

The implementation used three functions to develop the model. These functions are `makeKNN`, `usematrixKNN` and `useforloopKNN`. Since the `makeKNN` is identical to that provided in class, it is not included herein. Also, the `usematrixKNN` function is the original `useKNN` provided in class renamed to discriminate it from the for loop version. The `useforloopKNN` function is new and follows:

```
useforloopKNN <- function(knn, Xtest, nNeighbors=1) {
  Xtest <- knn$standardizeF(Xtest)
  predictions <- NULL
  for (rowi in 1:nrow(Xtest)) {
    nearest <- 999999
    classofNearest <- 0
    for (rowj in 1:nrow(knn$Xtrain)) {
      difference <- computeDiff(Xtest[rowi,], knn$Xtrain[rowj,])
      if (difference < nearest) {
        nearest <- difference
        classofNearest <- knn$Ttrain[rowj]
      }
    }
    predictions <- rbind(predictions, as.numeric(classofNearest))
  }
  predictions
}
```

The above code accepts the `knn` model developed by invoking `makeKNN`. It also accepts the samples to be classified and the number of neighbors. The number of neighbors is not used, however, is provided as it would be simple to extend the model beyond only one neighbor. First, the samples to be classified are standardized. Then for each member of the set to be classified, the distance from all the training set samples is computed, one training set sample at a time. For each member of the set to be classified, the smallest distance is found, and that nearest match is used to classify the sample. Once done for all the test sample set, the predictions are returned, exactly as the `usematrixKNN` does.

3.3 Model Fidelity and Real Time Usage

Once these two implementation of the KNN method were defined, two areas were explored. First, is the model fidelity. Do both of these implementations yield the same results? Second, is the real-time used to perform the classification. Are there any differences in time required to classify between the two approaches? Who implements matrix operations more efficiently? To answer these questions, it is necessary to run sample data through each model, compare the results and compare the time of execution. The two dimensional data development code for Assignment 3 was reused for this purpose, generating different numbers of points, ranging up to 1,000 samples.

$$n_{training} = n_{test}, \quad n_{test} = 10, 100, 250, 500, 1000 \quad (8)$$

$$n_{training} = \frac{1}{2} \times n_{test}, \quad n_{test} = 10, 100, 250, 500, 1000 \quad (9)$$

$$n_{test} = \frac{1}{2} \times n_{training}, \quad n_{training} = 10, 100, 250, 500, 1000 \quad (10)$$

For each of these sample size permutations, both implementations were executed and both the CPU time used as well as the prediction results for each model were captured. The time required to execute each implementation was captured as well as the percent agreement was done with the following code:

```
matrixKNNTime <- system.time(predmatrixc <- usematrixKNN(knn, Xtest, nNeighbors))
forloopKNNTime <- system.time(predforloopc <- useforloopKNN(knn, Xtest, nNeighbors))
algAgreement <- percentCorrect(predmyc, predc)
```

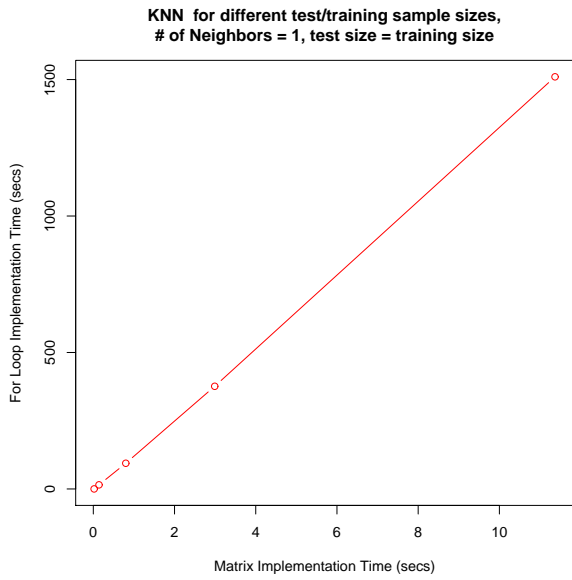
In all cases executed, the results were exact. In order to ensure that this was not the result of some other coding error (or hardcoding `percentCorrect` to always return perfect agreement), I passed a different set to each implementation and verified that the agreement was not perfect. Therefore, I am convinced that the matrix implementation is identical, from a functionality standpoint, to the matrix implementation.

Figure 4 depicts the system time required for each implementation, plotted as a scatter plot. The x-axis is the matrix implementation CPU time while the y-axis is the matrix implementation CPU time. Three plots are shown, one each for the relationship of training to test points. In all cases, the graphs are very linear. As more points are added, the ratio of the time required to perform the for loop implementation versus the matrix implementation timing requirement is constant. This consistency holds true for each of the ratio of test points to training points. For all cases, the for loop implementation of KNN requires about *125 times* more CPU time than the matrix implementation. Based on the above astounding data, I am convinced that R does a much better implementing their matrix operations than I can do with a couple of for loops. For the for loop implementation, it is possible to make some timing improvements, at the expense of some accuracy. If a tolerance is applied to the measure of the distance between the point that is to be classified from each of the points, once a sample from the training set is found that is close (within a tolerance) to the point to be classified then searching through the training set can be stopped. This modification would save some real time for the for loop implementation, however, the Matrix implementation would not benefit from this modification. The risk is that the classification might be incorrect, but as long as the tolerance is chosen to be sufficiently small, misclassifications should be minimized. The code used to develop each of the plots is as follows:

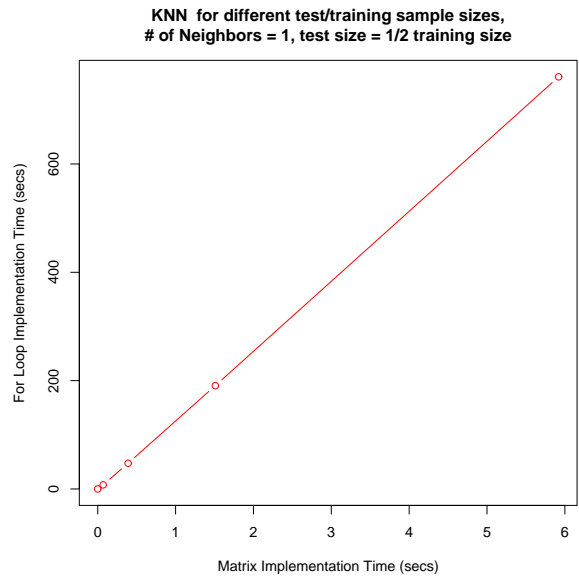
```
x11()
par.orig <- par(mfrow=c(1,1))
plot(timeRes[, "Matrix Time"], timeRes[, "For Loop Time"],
      xlab="Matrix Implementation Time (secs)",
      ylab="For Loop Implementation Time (secs)", col="red",
      main=c("KNN for different test/training sample sizes,\n
            # of Neighbors = 1, test size = training size"),
      type="b")
par(par.orig)
dev.copy2eps(file="part4b.eps")
```

3.4 Further Explorations

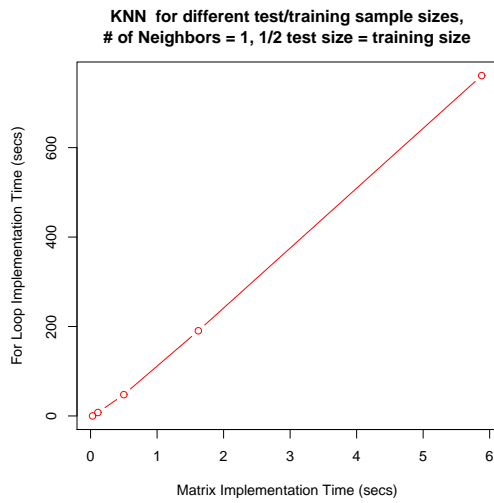
One further area I explored with KNN was the effect of the number of neighbors on the predicting ability of the model. For this section, I only used the matrix implementation as I had no intention of waiting hundreds of minutes for results. I used two sets of data, one segregated, one integrated and evaluated KNN with different numbers of neighbors. The results are shown in Figure 5. There are several interesting conclusions that can be drawn from this figure. First, when the data is well segregated, the prediction ability of KNN is very good regardless of the number of neighbors used in the classification process. Second, for the segregated data, when very few neighbors are used, predicting the training results is very accurate, but the prediction ability of the test set is not good. I think this is because when few neighbors are used, those neighbors could be the outlying points from the other class. The larger the number of neighbors, the better the predictability of the model, but only up to a point. Once the number of neighbors used in the decision process is greater than 10, the predicability of the model is no greater even if more samples are added in the decision process.



(a) Test Samples = Training Samples



(b) Test Samples = 0.5 Training Samples



(c) 0.5 = Test Samples

Figure 4: KNN CPU Usage - Matrix versus For Loop Implementation

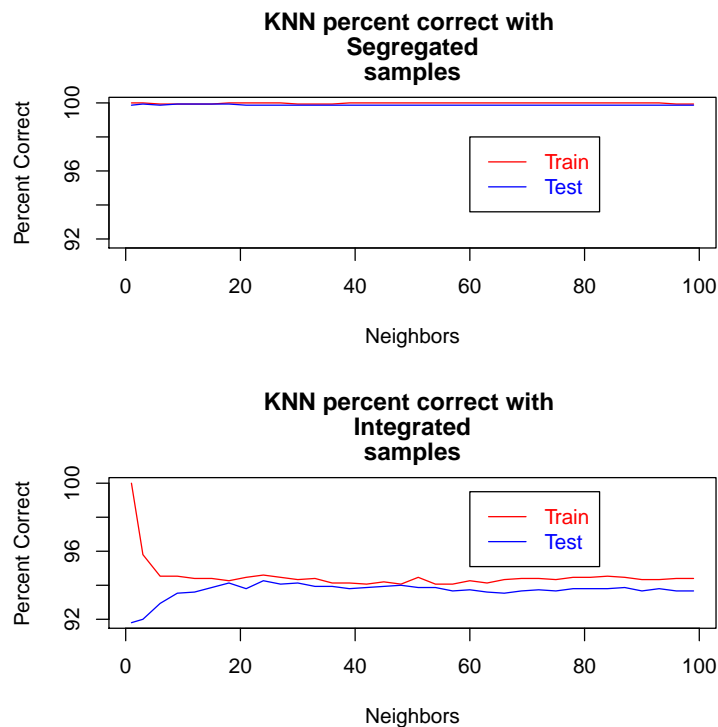


Figure 5: Effect of Neighbor Size on Prediction Accuracy

Learning about KNN, there were several variants I discovered that I would like to look at further. I found a paper which described a distance weighted KNN method where instead of equal weight in majority vote, each of the k nearest neighbors is weighted as a function of its distance from the sample that is being classified [4]. I found another paper where a modification to KNN was proposed where the value of k is a percentage of that class in the training set [5]. Despite wanting to implement these, I did not. Another limitation, other than the time required to execute the model, is that the structure of the training data is pretty much ignored. If a sample to be classified is near an outlying sample from the training set, that outlying point may be the closest and will be used for classification! This may not be optimal.

4 Parkinson's Data Analysis

The Parkinson's data that was used in class is available at the UCI Machine Learning Repository [6]. The data is a collection of samples from two classes, specifically subjects with Parkinson's and those who do not exhibit the disease. For each sample there are 23 attributes of voice information which may or may not be good indicators for classification. The group that collected and posted this data is trying to use measurable voice information as a classifier as they feel Parkinson's disease has a profound effect on subtle voice attributes of those who suffer from the disease. The data set is comprised of almost 200 samples from 31 subjects, 23 with Parkinson's, 8 without. For the previous assignment, this data was used in the development of LDA and QDA models. One of the goals of this assignment is to develop a model using the LR and KNN methods and then use the developed models to classify the data. Finally, the task is to compare the classification ability of each method on this data and make cogent observations.

4.1 Data Gathering and Model Development

For this portion of the assignment, I reused Dr. Anderson's code that accessed the Parkinson's data from the UCI web page. I also reused the code that partitioned the classification from the attributes and reused the code that segmented the data between training data and test data. I then further reused the LDA and QDA code provided by Dr. Anderson to create and use the models. Given that this is reused, I am not including that code here. I also reused the KNN and LR model code developed in the earlier sections of this assignment. A model was developed for LDA, QDA, LR, and KNN with 1 neighbor and 10 neighbors. After the model was developed, it was used with the training and test data sets. Then the fidelity was computed determining how many of the test data samples were correctly classified. This was done 100 times for each of the five models, each time repartitioning the original data set between the training set and test set, ensuring that 80% of the healthy samples and 80% of the Parkinson's samples were in the training set. Once this was done 100 times, the results were averaged for each model. The results are shown in Table 2.

Classification Model	% Correct with Train Data	% Correct with Test Data
LDA	72.87	67.70
QDA	98.10	88.10
KNN-1	100.00	94.17
KNN-10	91.87	88.17
Log Reg	90.52	85.35

Table 2: Summary of Average Percent Correct for Parkinson's Data, Training Frac =0.8

4.2 Model Fidelity with 80% Training Fraction

In analyzing the above data, there are several observations that must be made. First, when the training data is used in the model developed by the training data, each model performs better than for the test data set. This is expected, since the model was created using the training data, the model should be superior when it is given the training data as opposed to any other data set. To wit, for the KNN algorithm with one neighbor, the fidelity of the model is 100% since the data point used for classification is that exact point. Second, when looking at the test data classification percent correct, the algorithms fall into three groupings. First is KNN with one neighbor as it correctly classified about 94% of the test set data samples. Next are QDA, Logistic Regression and KNN with ten neighbors each of which classified between 85% and 89% of the test set samples correctly. Trailing last is LDA, which misclassified about one-third of the test data set samples. LDA only provided a 17% increase over guessing whether the subject had Parkinson's disease or not!

Given the difference between the predicability of the KNN models, one could infer that the data is not noisy, but there is little separation between the classes. Allowing more training samples into the voting process worsened the model. If the prediction accuracy was improved, we might be able to conclude that the data is noisy and there is good segregation between the classes for each attribute. A good next step would be to rerun this test with other values of neighbors to further substantiate my statements. I reran the KNN model using different neighbors sizes, from 1 to 75. Anything greater than 75, makes no sense given the size of the training set. Figure 6 details these results. An interesting observation is that the KNN model does better, for this data set, with about five neighbors used to vote on the classification of each test data point. There is also an area where increasing the voting members of the training set does not add anything to the fidelity of the model. The correct classification is almost the same for between 15 and 50 neighbors. From about 50 neighbors to 75, there is a degradation of the model to an even lower plateau than for between 15 and 50 neighbors. The code to perform the model generation, the model execution and the above plot is as follows:

```
nNeighborList <- c(1,seq(3,75,by=3))
for (nNeighbors in nNeighborList) {
  for (loopy in 1:100) {
```

KNN with varying neighbors against Parkinson's Data

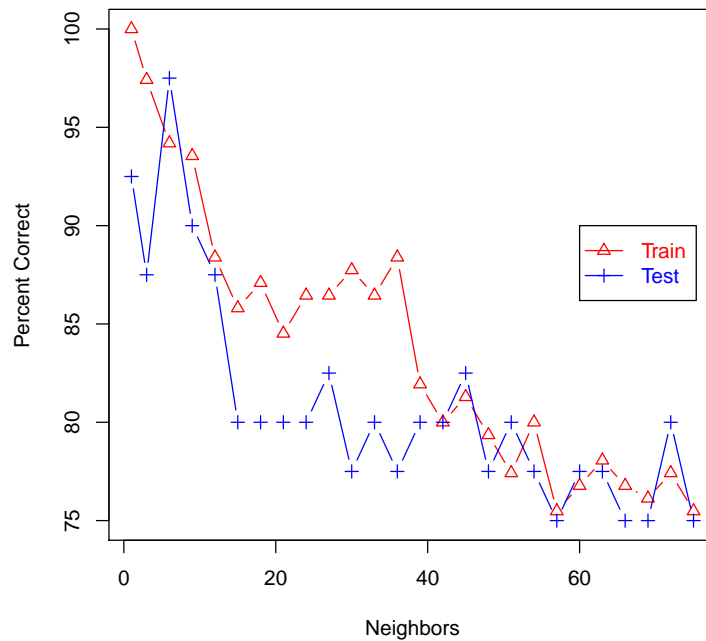


Figure 6: Parkinson's Data Model Fidelity as a Function of Number of Neighbors

```
pcKNNTrain <- NULL
pcKNNTest <- NULL

data <- dataorig
data <- data[sample(nrow(data)),]

status <- data[, "status"]
data <- data[, -which(colnames(data)=="status")]

dataHealthy <- data[status==0,]
dataParks <- data[status==1,]
nHealthy <- nrow(dataHealthy)
nParks <- nrow(dataParks)

trainf <- 0.8
Xtrain <- rbind(dataHealthy[1:floor(trainf*nHealthy),],
               dataParks[1:floor(trainf*nParks),])
Ttrain <- matrix(c(rep(1, floor(trainf*nHealthy)),
                  rep(2, floor(trainf*nParks))))
Xtest <- rbind(dataHealthy[-(1:floor(trainf*nHealthy)),],
               dataParks[-(1:floor(trainf*nParks)),])
Ttest <- matrix(c(rep(1, nHealthy-floor(trainf*nHealthy)),
                  rep(2, nParks-floor(trainf*nParks))))

knn <- makeKNN(Xtrain, Ttrain)
pcKNNTrain <- rbind(pcKNNTrain,
```

```

        percentCorrect ( Ttrain , usematrixKNN (knn , Xtrain , nNeighbors )))
pcKNNTest <- rbind (pcKNNTest ,
                    percentCorrect ( Ttest , usematrixKNN (knn , Xtest , nNeighbors )))
}
pcKNNTrainCombo <- rbind (pcKNNTrainCombo , cbind (mean (pcKNNTrain) , nNeighbors ))
pcKNNTestCombo <- rbind (pcKNNTestCombo , cbind (mean (pcKNNTest) , nNeighbors ))
}

colnames (pcKNNTrainCombo) <- c ("Percent Correct" , "Neighbors")
colnames (pcKNNTestCombo) <- c ("Percent Correct" , "Neighbors")

minmax <- range (pcKNNTrainCombo [, 1] , pcKNNTestCombo [, 1])

x11 ()
par . orig <- par (mfrow = c (1 , 1))

plot (pcKNNTrainCombo [, "Neighbors" ] , pcKNNTrainCombo [, "Percent Correct" ] ,
      xlab = "Neighbors" , ylab = "Percent Correct" ,
      ylim = c (minmax [1] , minmax [2]) ,
      col = "red" ,
      main = "KNN with varying neighbors against Parkinson 's Data" ,
      type = "b" , pch = 2)
points (pcKNNTestCombo [, "Neighbors" ] , pcKNNTestCombo [, "Percent Correct" ] ,
        col = "blue" , type = "b" , pch = 3)
legend (x = 60.0 , y = 90 ,
        legend = c ("Train" , "Test") ,
        col = c ("red" , "blue") , text . col = c ("red" , "blue") ,
        lty = c (1 , 1) , pch = c (2 , 3))

par (par . orig)
dev . copy2eps (file = "part3b . eps")

```

Given the difference between the QDA and LDA accuracy on the Parkinson's data set, I conclude that the assumption of equal Gaussian distributions (mean and shape) for all attributes in this data set is not correct. Given the large number of attributes, this is possible. Also, I do not think the data set is undersampled as in those cases LDA can often be better than QDA, which is not the case here. Finally, the segregation between the classes in each attribute does not appear to be well-segregated by a linear discriminant, or else the prediction ability of the model would be higher. While the data is better classified by quadratic discriminants, there is still weakness in that model. The difference between the training and test data set correction classifications is 10%, the most of any of the model.

The Logistic Regression model is good, but not as good as I would expect. Perhaps some of the attributes in the model do not adhere to the curve shown in Figure 1, i.e. they are better fit with a linear model as opposed to the logistic regression equation.

4.3 Model Fidelity with Different Training Fractions

In order to see the effects of different training fractions on the accuracy of the models, each of the five models were created using the Parkinson's data as above, however, the training fraction was varied. The same R code that was used above was reused, however, was encompassed with a for loop to allow for the changing of the training fraction. For each model/training fraction combination, 100 iterations were performed and the results averaged. This is the code that was used to generate the models:

```

trainfSet <- c (0.5 , 0.6 , 0.7 , 0.8 , 0.9 , 0.95)
for (trainf in trainfSet) {

```

```

for (loopy in 1:100) {
  data <- dataorig
  data <- data[sample(nrow(data)),]
  status <- data[, "status"]
  data <- data[, -which(colnames(data)=="status" )]
  dataHealthy <- data[status==0,]
  dataParks <- data[status==1,]
  nHealthy <- nrow(dataHealthy)
  nParks <- nrow(dataParks)

  Xtrain <- rbind(dataHealthy[1:floor(trainf*nHealthy),],
                 dataParks[1:floor(trainf*nParks),])
  Ttrain <- matrix(c(rep(1, floor(trainf*nHealthy)),
                    rep(2, floor(trainf*nParks))))
  Xtest <- rbind(dataHealthy[-(1:floor(trainf*nHealthy)),],
                 dataParks[-(1:floor(trainf*nParks)),])
  Ttest <- matrix(c(rep(1, nHealthy-floor(trainf*nHealthy)),
                    rep(2, nParks-floor(trainf*nParks))))

  lda <- makeLDA(Xtrain, Ttrain)
  pcLDATrain <- rbind(pcLDATrain, percentCorrect(Ttrain, useLDA(lda, Xtrain)))
  pcLDATest <- rbind(pcLDATest, percentCorrect(Ttest, useLDA(lda, Xtest)))

  qda <- makeQDA(Xtrain, Ttrain)
  pcQDATrain <- rbind(pcQDATrain, percentCorrect(Ttrain, useQDA(qda, Xtrain)))
  pcQDATest <- rbind(pcQDATest, percentCorrect(Ttest, useQDA(qda, Xtest)))

  knn <- makeKNN(Xtrain, Ttrain)
  pcKNN1Train <- rbind(pcKNN1Train,
                      percentCorrect(Ttrain, usematrixKNN(knn, Xtrain, 1)))
  pcKNN1Test <- rbind(pcKNN1Test,
                      percentCorrect(Ttest, usematrixKNN(knn, Xtest, 1)))

  pcKNN10Train <- rbind(pcKNN10Train,
                        percentCorrect(Ttrain, usematrixKNN(knn, Xtrain, 10)))
  pcKNN10Test <- rbind(pcKNN10Test,
                        percentCorrect(Ttest, usematrixKNN(knn, Xtest, 10)))

  logReg <- makeLogReg(Xtrain, Ttrain, 2)
  pcLRTrain <- rbind(pcLRTrain,
                    percentCorrect(Ttrain, useLogReg(Xtrain, logReg)))
  pcLRTest <- rbind(pcLRTest,
                    percentCorrect(Ttest, useLogReg(Xtest, logReg)))
}

pcLDATrainCombo <- rbind(pcLDATrainCombo, cbind(mean(pcLDATrain), trainf))
pcLDATestCombo <- rbind(pcLDATestCombo, cbind(mean(pcLDATest), trainf))
pcQDATrainCombo <- rbind(pcQDATrainCombo, cbind(mean(pcQDATrain), trainf))
pcQDATestCombo <- rbind(pcQDATestCombo, cbind(mean(pcQDATest), trainf))
pcKNN1TrainCombo <- rbind(pcKNN1TrainCombo, cbind(mean(pcKNN1Train), trainf))
pcKNN1TestCombo <- rbind(pcKNN1TestCombo, cbind(mean(pcKNN1Test), trainf))
pcKNN10TrainCombo <- rbind(pcKNN10TrainCombo, cbind(mean(pcKNN10Train), trainf))
pcKNN10TestCombo <- rbind(pcKNN10TestCombo, cbind(mean(pcKNN10Test), trainf))
pcLRTrainCombo <- rbind(pcLRTrainCombo, cbind(mean(pcLRTrain), trainf))

```

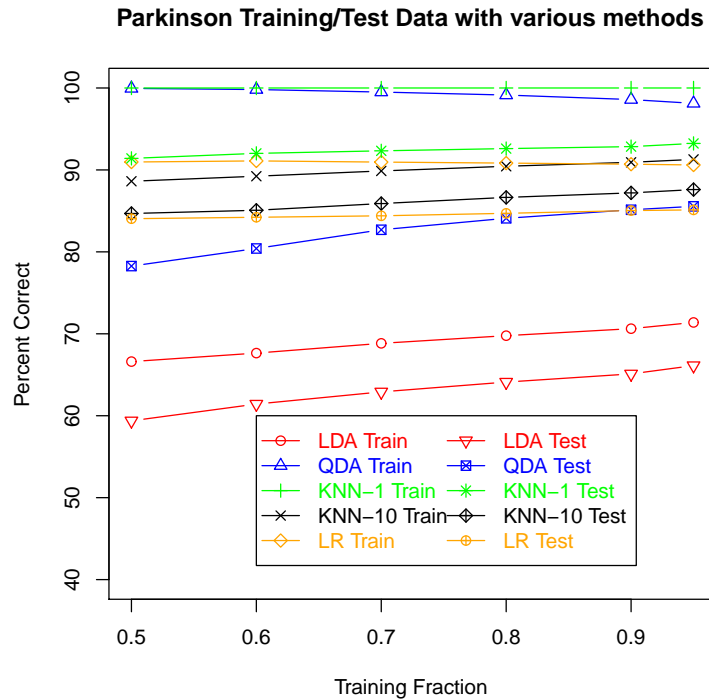


Figure 7: Parkinson’s Data Model Fidelity as a Function of Training Fraction

```
pcLRTestCombo <- rbind(pcLRTestCombo, cbind(mean(pcLRTest), trainf))
}
```

The result of this code is to have 60 prediction accuracy data points. For each of the five models, there is one set of training predicted accuracies and one set of test predicted accuracies. Each of these two predicted accuracy sets include six points, one for each training fraction, 0.5, 0.6, 0.7, 0.8, 0.9 and 0.95. Once this data was collected, it was plotted. The results are shown in Figure 7.

For each classification method, the prediction capability of the model against the test and training data sets are shown. For each, the performance is shown for the varying training fraction. This plot effectuates many observations. The same observations noted in the previous section also apply in this section and are not repeated for the sake of brevity (and sanity). Additionally, there are several other observations that are evident in this plot. First, the training set percent correct classification for each method is better than the same for the test data set, regardless of classification method or the training fraction. Next, for each model, the accuracy of the predictions for the test data set increased as the training fraction increased. For the Logistic Regression model, the improvement is very slight as the percent correct of the predicted values is almost constant despite the increased training fraction. In addition, comparing the performance of each model to each other based on the test data shows results similar to those mentioned in the previous sections. LDA yields poor results and stands out for its (lack of) results. It is not much better than tossing a coin. I believe these results are consistent with the explanation offered in the previous section. The other four methods yield between an 80% and 92% correct prediction. For all four models, there is an improvement as the training fraction increases. In some cases the improvement is dramatic, such as for QDA where a 6% improvement is seen as the training fraction increases. However, for Logistic Regression model no increase in training fraction seems to help the model fidelity. In fact, the Logistic Regression model is rock solid, not varying much for either training data set or test data set.

Between the collection of the data and the generation of the plot, column names were added. I am not including that code for the sake of brevity. The R code used to generate this plot is as follows:

```
x11()
par.orig <- par(mfrow=c(1,1))

minmax <- range(pcLDATrainCombo[, "Train PC"], pcQDATrainCombo[, "Train PC"],
               pcKNN1TrainCombo[, "Train PC"], pcKNN10TrainCombo[, "Train PC"],
               pcLDATestCombo[, "Test PC"], pcQDATestCombo[, "Test PC"],
               pcKNN1TestCombo[, "Test PC"], pcKNN10TestCombo[, "Test PC"],
               pcLRTestCombo[, "Test PC"], pcLRTestCombo[, "Test PC"], 40)
plot(pcLDATrainCombo[, "Trainf"], pcLDATrainCombo[, "Train PC"],
     xlab="Training Fraction", ylab="Percent Correct",
     ylim=c(minmax[1], minmax[2]),
     col="red", pch=1, main=c("Parkinson Training/Test Data with various methods"),
     type="b")
points(pcQDATrainCombo[, "Trainf"], pcQDATrainCombo[, "Train PC"],
       col="blue", pch=2, type="b")
points(pcKNN1TrainCombo[, "Trainf"], pcKNN1TrainCombo[, "Train PC"],
       col="green", pch=3, type="b")
points(pcKNN10TrainCombo[, "Trainf"], pcKNN10TrainCombo[, "Train PC"],
       col="black", pch=4, type="b")
points(pcLRTrainCombo[, "Trainf"], pcLRTrainCombo[, "Train PC"],
       col="orange", pch=5, type="b")
points(pcLDATestCombo[, "Trainf"], pcLDATestCombo[, "Test PC"],
       col="red", pch=6, type="b")
points(pcQDATestCombo[, "Trainf"], pcQDATestCombo[, "Test PC"],
       col="blue", pch=7, type="b")
points(pcKNN1TestCombo[, "Trainf"], pcKNN1TestCombo[, "Test PC"],
       col="green", pch=8, type="b")
points(pcKNN10TestCombo[, "Trainf"], pcKNN10TestCombo[, "Test PC"],
       col="black", pch=9, type="b")
points(pcLRTestCombo[, "Trainf"], pcLRTestCombo[, "Test PC"],
       col="orange", pch=10, type="b")
legendText = c("LDA Train", "QDA Train", "KNN-1 Train", "KNN-10 Train",
              "LR Train", "LDA Test", "QDA Test", "KNN-1 Test",
              "KNN-10 Test", "LR Test"),
colText = c("red", "blue", "green", "black", "orange", "red", "blue",
            "green", "black", "orange"),
legend(x = 0.6, y = 60.0,
       legend = legendText,
       col = colorText,
       text.col = colorText,
       lty = c(1, 1, 1, 1, 1, 1, 1, 1, 1, 1),
       pch = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
       ncol=2)

par(par.orig)
dev.copy2eps(file="part3a.eps")
```

5 Wine Data Analysis

The Wine data that I used in the previous assignment is available at the UCI Machine Learning Repository [7]. The data is a collection of data samples from three classes of wine. For each sample there are 13

attributes of measured parameters which may or may not be good indicators for classification. The group that collected and posted this data stated that this data is well-suited for classification. Therefore, I expect better classification fidelity with the models we are developing for this data set than with the Parkinson's data. The data set is comprised of about 175 samples fairly distributed amongst all three classes. For the previous assignment, this data was used in the development of LDA and QDA models. One of the goals of this assignment is to develop a model using the LR and KNN methods and then use the developed models to classify the data. Finally, the task is to compare the classification ability of each method on this data and make (relevant) observations.

5.1 Data Gathering and Model Development

For this portion of the assignment, I reused Dr. Anderson's code that accessed the Parkinson's data from the UCI web page with some modifications. I also used the code that partitioned the classification from the attributes and used the code that segmented the code between training data and test data with some modifications. This data set has three classes while the Parkinson's data set only had two. I then further reused the LDA and QDA code provided by Dr. Anderson and the code I developed for the KNN and LR code created while analyzing the Parkinson's data. I am only including that code which I developed that is unique. Here is the code that read in and set the column names for the data.

```
dataorig <- NULL
dataorig <- read.table("http://archive.ics.uci.edu/ml/
                      machine-learning-databases/wine/wine.data",
                      header=FALSE, sep=",")
colnames <- c("Class", "Alcohol", "Malic acid", "Ash", "Alcalinity of ash",
              "Magnesium", "Total phenols", "Flavanoids", "Nonflavanoid phenols",
              "Proanthocyanins", "Color intensity", "Hue", "OD280/OD315",
              "Proline")
colnames(dataorig) <- colnames
```

Since this data did not have a header, I assigned the column names. Once the data was read in, the samples from each class were segregated into the training set and test set as follows. The below code was executed 100 times, each time the data was randomized so numerous permutations of training set and test set were evaluated.

```
data <- dataorig
data <- data[sample(nrow(data)),]

class <- data[, 1]
data <- data[, 2:14]

dataC1 <- data[class==1,]
dataC2 <- data[class==2,]
dataC3 <- data[class==3,]
nC1 <- nrow(dataC1)
nC2 <- nrow(dataC2)
nC3 <- nrow(dataC3)

trainf <- 0.8
Xtrain <- rbind(dataC1[1:floor(trainf*nC1),],
               dataC2[1:floor(trainf*nC2),],
               dataC3[1:floor(trainf*nC3),])
Ttrain <- matrix(c(rep(1, floor(trainf*nC1)),
                  rep(2, floor(trainf*nC2)),
                  rep(3, floor(trainf*nC3))))
```

```

Xtest <- rbind(dataC1[-(1:floor(trainf*nC1)),],
              dataC2[-(1:floor(trainf*nC2)),],
              dataC3[-(1:floor(trainf*nC3)),])
Ttest <- matrix(c(rep(1,nC1-floor(trainf*nC1)),
                  rep(2,nC2-floor(trainf*nC2)),
                  rep(3,nC3-floor(trainf*nC3))))

```

Once the data was brought in and segregated, a model was developed for LDA, QDA, LR, and KNN with 1 neighbor and 10 neighbors. After the model was developed, it was used with the training and test data sets. Then the fidelity was computed determining how many of the test data samples were correctly classified. This was done 100 times for each of the five models, each time repartitioning the original data set between the training set and test set, ensuring that 80% of each class were in the training set and 20% of each class were in the test set. Once this was done 100 times, the results were averaged for each model. The results are shown in Table 3.

Classification Model	% Correct with Train Data	% Correct with Test Data
LDA	89.92	88.05
QDA	99.59	99.18
KNN-1	100.00	94.91
KNN-10	97.34	96.05
Log Reg	100.00	97.64

Table 3: Summary of Average Percent Correct for Wine Data, Training Frac =0.8

5.2 Model Fidelity with 80% Training Fraction

In analyzing the above data, there are several observations that must be made. First, as expected, when the training data is used in the model developed by the training data, each model performs better than for the test data set. Since the model was created using the training data, the model should be superior when it is given the training data as opposed to any other data set and it is. However, the spread between the model fidelity for the training and test data sets is much less than with the Parkinson's data set. Just as with the Parkinson's data set, for the KNN algorithm with only one neighbor, the fidelity of the model is 100% since the data point used for classification is that exact point. Second, when looking at the test data classification percent correct, the algorithms fall into three groupings. The best predictors include the model developed by the QDA and Logistic Regression methods. Shortly after that are the two instantiations of the KNN model, with one and ten neighbors. Similar to the Parkinson's data, the worst predictor was the model developed by LDA. However, all the models were excellent or very good.

Given the narrow difference between the predicability of the KNN models, the underlying data is not noisy and there is good separation between the classes. Allowing more training samples into the voting process improved the model, but minutely. Diverging from the comments in the Parkinson's data analysis, other KNN models with different numbers of neighbors is not needed and would not yield any further benefit.

There is about an 11% separation between the QDA and LDA accuracy. The assumption of equal Gaussian distributions (mean and shape) for all attributes in this data set is not completely correct, but more correct than for the Parkinson's data. Since the number of attributes is less, this is possible. Similar to the Parkinson's analysis, I do not believe the data set is undersampled. In opposition to the findings for the Parkinson's data set, the segregation between the classes in each attribute is well defined by a linear discriminant. Given the superior performance of the QDA model, the data is well segregated by quadratic discriminants.

5.3 Model Fidelity with Different Training Fractions

In order to see the effects of different training fractions on the accuracy of the models, each of the five models were created using the Wine data as above but the training fraction was varied. The same R code that was

used above was reused wrapped by a for loop to allow for the slewing of the training fraction. For each model/training fraction combination, 100 iterations were performed and the results averaged. This is the code that was used to generate the models, randomize the order of the data set and partition the data set among the test and training data sets:

```

trainfSet <- c(0.5, 0.6, 0.7, 0.8, 0.9, 0.95)
for (trainf in trainfSet) {
  for (loopy in 1:100) {

    data <- dataorig
    data <- data[sample(nrow(data)),]

    class <- data[, 1]
    data <- data[, 2:14]

    dataC1 <- data[class==1,]
    dataC2 <- data[class==2,]
    dataC3 <- data[class==3,]
    nC1 <- nrow(dataC1)
    nC2 <- nrow(dataC2)
    nC3 <- nrow(dataC3)

    Xtrain <- rbind(dataC1[1:floor(trainf*nC1),],
                   dataC2[1:floor(trainf*nC2),],
                   dataC3[1:floor(trainf*nC3),])
    Ttrain <- matrix(c(rep(1, floor(trainf*nC1)),
                      rep(2, floor(trainf*nC2)),
                      rep(3, floor(trainf*nC3))))
    Xtest <- rbind(dataC1[-(1:floor(trainf*nC1)),],
                  dataC2[-(1:floor(trainf*nC2)),],
                  dataC3[-(1:floor(trainf*nC3)),])
    Ttest <- matrix(c(rep(1, nC1-floor(trainf*nC1)),
                      rep(2, nC2-floor(trainf*nC2)),
                      rep(3, nC3-floor(trainf*nC3))))

    lda <- makeLDA(Xtrain, Ttrain)
    pcLDATrain <- rbind(pcLDATrain, percentCorrect(Ttrain, useLDA(lda, Xtrain)))
    pcLDATest <- rbind(pcLDATest, percentCorrect(Ttest, useLDA(lda, Xtest)))

    qda <- makeQDA(Xtrain, Ttrain)
    pcQDATrain <- rbind(pcQDATrain, percentCorrect(Ttrain, useQDA(qda, Xtrain)))
    pcQDATest <- rbind(pcQDATest, percentCorrect(Ttest, useQDA(qda, Xtest)))

    knn <- makeKNN(Xtrain, Ttrain)
    pcKNN1Train <- rbind(pcKNN1Train,
                        percentCorrect(Ttrain, usematrixKNN(knn, Xtrain, 1)))
    pcKNN1Test <- rbind(pcKNN1Test,
                       percentCorrect(Ttest, usematrixKNN(knn, Xtest, 1)))

    pcKNN10Train <- rbind(pcKNN10Train,
                         percentCorrect(Ttrain, usematrixKNN(knn, Xtrain, 10)))
    pcKNN10Test <- rbind(pcKNN10Test,
                        percentCorrect(Ttest, usematrixKNN(knn, Xtest, 10)))

    logReg <- makeLogReg(Xtrain, Ttrain, 3)

```

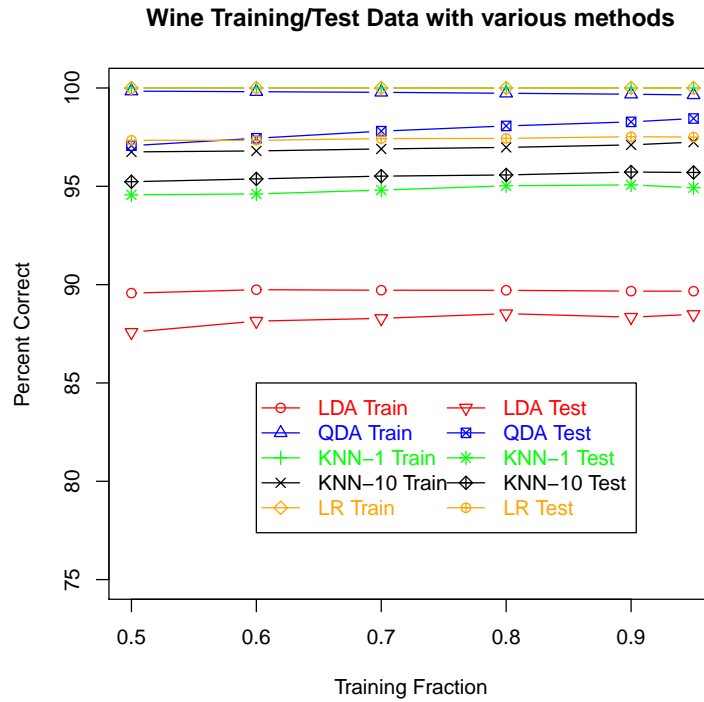


Figure 8: Wine Data Model Fidelity as a Function of Training Fraction

```

pcLRTrain <- rbind(pcLRTrain,
                  percentCorrect(Ttrain, useLogReg(Xtrain, logReg)))
pcLRTest <- rbind(pcLRTest,
                  percentCorrect(Ttest, useLogReg(Xtest, logReg)))
}
pcLDATrainCombo <- rbind(pcLDATrainCombo, cbind(mean(pcLDATrain), trainf))
pcLDATestCombo <- rbind(pcLDATestCombo, cbind(mean(pcLDATest), trainf))

pcQDATrainCombo <- rbind(pcQDATrainCombo, cbind(mean(pcQDATrain), trainf))
pcQDATestCombo <- rbind(pcQDATestCombo, cbind(mean(pcQDATest), trainf))

pcKNN1TrainCombo <- rbind(pcKNN1TrainCombo, cbind(mean(pcKNN1Train), trainf))
pcKNN1TestCombo <- rbind(pcKNN1TestCombo, cbind(mean(pcKNN1Test), trainf))

pcKNN10TrainCombo <- rbind(pcKNN10TrainCombo, cbind(mean(pcKNN10Train), trainf))
pcKNN10TestCombo <- rbind(pcKNN10TestCombo, cbind(mean(pcKNN10Test), trainf))

pcLRTrainCombo <- rbind(pcLRTrainCombo, cbind(mean(pcLRTrain), trainf))
pcLRTestCombo <- rbind(pcLRTestCombo, cbind(mean(pcLRTest), trainf))
}

```

The result of this code is to have 60 prediction accuracy data points. For each of the five models, there is one set of training predicted accuracies and one set of test predicted accuracies. Each of these two predicted accuracy sets include six points, one for each training fraction, 0.5, 0.6, 0.7, 0.8, 0.9 and 0.95. Once this data was collected, it was plotted. The results are shown in Figure 8. The construction of this plot is identical to the one developed for the Parkinson's data. For each classification method, the prediction capability of the

model against the test and training data sets are shown. For each, the performance is shown for the varying training fraction. There are some similarities and some differences between the plots. The same observations noted in the previous section also apply in this section and are not repeated. Additionally, there are several other observations that are evident in this plot. First, the training set percent correct classification for each method is better than the same for the test data set, regardless of classification method or the training fraction. This is expected. Next, for each model, the accuracy of the predictions for the test data set increased very slightly as the training fraction increased. For the QDA model, the improvement is more pronounced than the other four models. In addition, comparing the performance of each model to each other based on the test data shows results similar to those mentioned in the previous sections. LDA yields good results while the other four models are outstanding. For all the models, the prediction accuracy against this data set is much better than that seen with the Parkinson's data set. The Logistic Regression model is very solid, both for the training set and test data sets regardless of the training fraction used which is consistent with the Parkinson's data.

Similar to the process used in the Parkinson's model development and analysis, I am not including that code. The R code used to generate this plot is as follows:

```
x11()
par.orig <- par(mfrow=c(1,1))

minmax <- range(pcLDATrainCombo[, "Train PC"], pcQDATrainCombo[, "Train PC"],
               pcKNN1TrainCombo[, "Train PC"], pcKNN10TrainCombo[, "Train PC"],
               pcLDATestCombo[, "Test PC"], pcQDATestCombo[, "Test PC"],
               pcKNN1TestCombo[, "Test PC"], pcKNN10TestCombo[, "Test PC"], 75)
plot(pcLDATrainCombo[, "Trainf"], pcLDATrainCombo[, "Train PC"],
     xlab="Training Fraction", ylab="Percent Correct",
     ylim=c(minmax[1], minmax[2]),
     col="red", pch=1, main=c("Wine Training/Test Data with various methods"),
     type="b")
points(pcQDATrainCombo[, "Trainf"], pcQDATrainCombo[, "Train PC"],
       col="blue", pch=2, type="b")
points(pcKNN1TrainCombo[, "Trainf"], pcKNN1TrainCombo[, "Train PC"],
       col="green", pch=3, type="b")
points(pcKNN10TrainCombo[, "Trainf"], pcKNN10TrainCombo[, "Train PC"],
       col="black", pch=4, type="b")
points(pcLRTrainCombo[, "Trainf"], pcLRTrainCombo[, "Train PC"],
       col="orange", pch=5, type="b")
points(pcLDATestCombo[, "Trainf"], pcLDATestCombo[, "Test PC"],
       col="red", pch=6, type="b")
points(pcQDATestCombo[, "Trainf"], pcQDATestCombo[, "Test PC"],
       col="blue", pch=7, type="b")
points(pcKNN1TestCombo[, "Trainf"], pcKNN1TestCombo[, "Test PC"],
       col="green", pch=8, type="b")
points(pcKNN10TestCombo[, "Trainf"], pcKNN10TestCombo[, "Test PC"],
       col="black", pch=9, type="b")
points(pcLRTestCombo[, "Trainf"], pcLRTestCombo[, "Test PC"],
       col="orange", pch=10, type="b")
legendText = c("LDA Train", "QDA Train", "KNN-1 Train", "KNN-10 Train",
              "LR Train", "LDA Test", "QDA Test", "KNN-1 Test",
              "KNN-10 Test", "LR Test"),
colText = c("red", "blue", "green", "black", "orange", "red", "blue",
            "green", "black", "orange"),
legend(x = 0.6, y = 60.0,
       legend = legendText,
       col = colText,
```

```

text.col = colorText ,
lty = c(1, 1, 1, 1, 1, 1, 1, 1, 1, 1),
pch = c(1,2,3,4,5,6,7,8,9,10),
ncol=2)

par(par.orig)
dev.copy2eps(file="part5a.eps")

```

6 Conclusion

I learned a great deal about the KNN and LR algorithms by doing this assignment. I was most interested in the different KNN implementations and the stunning difference in CPU time required for each. I am also interested that all of these classification methods resulted in similar prediction results regardless of the training fraction used. Each classifier did exhibit different capabilities for model fidelity which is to be expected.

I had a great deal of difficulty with the LR implementation. It took a while to understand what LR is and how to implement it. For some reason, I have a mental block when it comes to LR. LDA, QDA, and KNN are all fairly obvious methods for classification, but LR, at least to me, requires a greater level of understanding than what I currently possess.

I also selected the wine data set to work with as I was familiar with it, I like wine, it has a fairly small number of attributes, and has no binary fields. It was also well behaved when I used it as my data-set-of-choice for assignment 3.

References

- [1] *Gradient Descent*, http://en.wikipedia.org/wiki/Gradient_descent, Retrieved October 15th, 2009.
- [2] *Unit II, Open Methods*, http://epoch.uwaterloo.ca/~ponnu/syde312/open_methods/page3.htm, Retrieved October 15th, 2009.
- [3] *Adaptive Filters I: Newtons Method, Steepest Descent and LMS*, <http://www.stanford.edu/class/ee264/lectures/mylecture13.pdf>, Retrieved October 15th, 2009.
- [4] *Weighted KNN*, http://sigwww.cs.tut.fi/TICSP/Bio_Seminar/Files/19_09_01_Nicorici.pdf, Retrieved October 13th, 2009.
- [5] *An Improved k-Nearest Neighbor Algorithm for Text Classification*, <http://arxiv.org/ftp/cs/papers/0306/0306099.pdf>, Retrieved October 13th, 2009.
- [6] *Parkinson's Data Set*, <http://archive.ics.uci.edu/ml/datasets/Parkinsons>, Retrieved October 13th, 2009.
- [7] *Wine Data Set*, <http://archive.ics.uci.edu/ml/datasets/Wine>, Retrieved October 14th, 2009.