

CS545 - Assignment 8: Classification Through Clustering

Geoffrey Sewell

December 17, 2009

Contents

| | | |
|----------|--------------------------------|-----------|
| 1 | Introduction | 1 |
| 2 | Set up | 1 |
| 2.1 | R code | 2 |
| 2.1.1 | cclust Function | 2 |
| 2.1.2 | Prediction Function | 5 |
| 2.1.3 | Print Function | 6 |
| 2.2 | C code | 6 |
| 2.2.1 | kmeans Function | 6 |
| 2.2.2 | assign Function | 7 |
| 2.2.3 | reloc Function | 9 |
| 2.2.4 | CC_median function | 10 |
| 2.2.5 | CC_sort Function | 11 |
| 3 | Simple Data Set | 12 |
| 4 | Zip Code Data | 12 |
| 4.1 | Set Up | 12 |
| 4.2 | Predictive Abilities | 14 |
| 4.3 | Hand Drawn Digits | 15 |
| 4.3.1 | Set Up | 15 |
| 4.3.2 | Discussion | 17 |
| 5 | Conclusion | 19 |

1 Introduction

Throughout this semester, as a class, we have learned many different techniques used for classification. However, there are many techniques that we have not used. One of these methods is through the use of clustering. In Section 2, I will review the code used for clustering which involves both R and C code. Finally, Section 4.3 explains the results from applying the clustering method to the hand written numbers data set that was used for assignment 6.

2 Set up

Code for clustering a set of data has already been created and stored in an R package called cclust which was created by Evgenia Dimitriadou. The author makes use of both C and R code for clustering. In this section, I will be discussing what exactly the code is programmed to do in order to make clustering a reality.

As mentioned in Section 1, the code involved in the `cclust` package, has pieces of it implemented in both R and C. The R code is what the user will call while the C code is called from the R code. Below, the R code is examined.

2.1 R code

2.1.1 `cclust` Function

The `cclust` function is what is called by the coder to evaluate a data set. The `cclust` function is pretty flexible in regards to how it is set up. As arguments to the function it takes data set, the centers, the maximum number of iterations, whether or not to print out detailed information, as well as the different parameters for clustering. These parameters include how distance is calculated, the method for clustering, and the learning rate for clustering.

Initially the code first grabs the number of rows and columns of the data set and stores a copy of the data matrix. It then randomizes the rows of the data matrix and stores the new randomized data set. Depending on what is passed to the parameter `centers`, two things can happen. Either the `centers` parameter contains a matrix that explicitly state how many and the location for each center of the initial clusters. If the `centers` parameter happens to be a number, then the centers are randomly chosen from the data matrix passed in.

```
cclust <- function (x, centers, iter.max = 100, verbose = FALSE, dist = "euclidean",
  method = "kmeans", rate.method = "polynomial", rate.par = NULL)
{
  xrows <- dim(x)[1]
  xcols <- dim(x)[2]
  xold <- x
  perm <- sample(xrows)
  x <- x[perm, ]
  # initial values are given
  if (is.matrix(centers))
    ncenters <- dim(centers)[1]
  else {
    # take centers random vectors as initial values
    ncenters <- centers
    centers <- x[rank(runif(xrows))[1:ncenters], ]
  }
}
```

Once the centers have been determined a lot of error checking is done on the parameters such as `dist`, `method`, and `rate.method` to make sure that appropriate string values are given as arguments. Using the `pmatch` function, if the argument is not found in the list of words, then an error message is returned stating an invalid parameter was entered. Finally depending on the `rate.method` selected `rate.par` is initialized to some value if it has not already been given a value.

```
dist <- pmatch(dist, c("euclidean", "manhattan"))
if (is.na(dist))
  stop("invalid distance")
if (dist == -1)
  stop("ambiguous distance")
method <- pmatch(method, c("kmeans", "hardcl", "neuralgas"))
if (is.na(method))
  stop("invalid clustering method")
if (method == -1)
  stop("ambiguous clustering method")
rate.method <- pmatch(rate.method, c("polynomial", "exponentially.decaying"))
if (is.na(rate.method))
  stop("invalid learning rate method")
if (rate.method == -1)
```

```

    stop("ambiguous learning rate method")
  if (method == 2) {
    if (rate.method == 1 && missing(rate.par)) {
      rate.par <- c(1e-00, 0e-00)
    }
    else if (rate.method == 2 && missing(rate.par)) {
      rate.par <- c(0.1, 1e-04)
    }
  }
  if (method == 3 && missing(rate.par)) {
    rate.par <- c(0.5, 0.005, 10, 0.01)
  }
}

```

Once error checking is done, values are set up so that the appropriate C function can be called, which will be discussed in Section 2.2. The initial cluster centers are stored in a new. A variable is used to store the different cluster classifications. Last, a number of integer vectors are created. After all the variables are set up, the method number, determined in the above code, is used to determine which C function to call.

```

  initcenters <- centers
  # dist <- matrix(0, xrows, ncenters)
  # necessary for empty clusters
  pos <- as.factor(1:ncenters)
  rownames(centers) <- pos
  iter <- integer(1)
  changes <- integer(iter.max)
  cluster <- integer(xrows)
  clustersize <- integer(ncenters)
  if (method == 1) {
    retval <- .C("kmeans", xrows = as.integer(xrows),
                xcols = as.integer(xcols),
                x = as.double(x), ncenters = as.integer(ncenters),
                centers = as.double(centers),
                cluster = as.integer(cluster),
                iter.max = as.integer(iter.max), iter = as.integer(iter),
                changes = as.integer(changes),
                clustersize = as.integer(clustersize),
                verbose = as.integer(verbose),
                dist = as.integer(dist - 1), PACKAGE="cclust")
  }
  else if (method == 2) {
    retval <- .C("hardcl", xrows = as.integer(xrows), xcols = as.integer(xcols),
                x = as.double(x), ncenters = as.integer(ncenters),
                centers = as.double(centers),
                cluster = as.integer(cluster),
                iter.max = as.integer(iter.max), iter = as.integer(iter),
                clustersize = as.integer(clustersize),
                verbose = as.integer(verbose),
                dist = as.integer(dist - 1),
                methrate = as.integer(rate.method - 1),
                par = as.double(rate.par), PACKAGE="cclust")
  }
  else if (method == 3) {
    retval <- .C("neuralgas", xrows = as.integer(xrows),
                xcols = as.integer(xcols), x = as.double(x),
                ncenters = as.integer(ncenters),

```

```

        centers = as.double(centers),
        cluster = as.integer(cluster),
        iter.max = as.integer(iter.max), iter = as.integer(iter),
        clustersize = as.integer(clustersize),
        verbose = as.integer(verbose),
        dist = as.integer(dist - 1), par = as.double(rate.par),
        PACKAGE="cclust")
    }
    centers <- matrix(retval$centers, ncol = xcols, dimnames = dimnames(initcenters))
    cluster <- retval$cluster + 1
    cluster <- cluster[order(perm)]
    if (method == 1) {
        methrate <- NA
        par <- NA
    }
    if (method == 3) {
        methrate <- NA
    }
}

```

From the output of the C code, the cluster centers are obtained and the cluster with the number of clusters for each data point is increased by 1 and reordered.

Finally, before returning the results of clustering, the sum of squares is calculated for each cluster in relation to the cluster center. With the original data set, the cluster center is subtracted from the data coordinates and squared. For each cluster, the sum of squared distances is calculated then returned. After the sum of squared distances is calculated, all the variables are placed in to a list and the class of the list is changed to "cclust".

```

withinss <- function(clobj, x){

    retval <- rep(0, nrow(clobj$centers))
    x <- (x - clobj$centers[clobj$cluster, ])^2
    for(k in 1:nrow(clobj$centers)){
        retval[k] <- sum(x[clobj$cluster==k,])
    }
    retval
}

within <- withinss(list(centers = centers, cluster = cluster), xold)

retval <- list(centers = centers, initcenters = initcenters,
    ncenters = ncenters, cluster = cluster, size = retval$clustersize,
    iter = retval$iter - 1, changes = retval$changes, dist = dist,
    method = method, rate.method = rate.method, rate.par = rate.par,
    call = match.call(), withinss = within)
class(retval) <- c("cclust")
return(retval)
}

```

The returned cluster information will allow the user to look at all sorts of information from the clustering algorithm. The can specifically look at the cluster centers, the initial cluster centers, the number of data points for each cluster, and the list goes on. It is extremely helpful in seeing how the data is being split up especially when dealing with higher dimensional data which is more difficult to plot on a graph.

2.1.2 Prediction Function

Thinking back to assignment 6, when a neural network, it was tested with completely new set of data that was not used in training the neural network. Clustering is no different. When clusters are set up using training data, the clusters can be used to predict the classification of a totally new set of data. The function below is R code to actually predict the classification of the new data set.

```
predict.cclust <- function(object , newdata , ...){

  clobj <- object
  x <- newdata
  xrows<-dim(x)[1]
  xcols<-dim(x)[2]
  ncenters <- clobj$ncenters
  cluster <- integer(xrows)
  clustersize <- integer(ncenters)

  if(dim(clobj$centers)[2] != xcols){
    stop("Number of variables in cluster object and x are not the same!")
  }

  retval <- .C("assign",
              xrows = as.integer(xrows),
              xcols = as.integer(xcols),
              x = as.double(x),
              ncenters = as.integer(ncenters),
              centers = as.double(clobj$centers),
              cluster = as.integer(cluster),
              clustersize = as.integer(clustersize),
              dist = as.integer(clobj$dist -1),PACKAGE="cclust")

  clobj$initcenters <- NULL
  clobj$iter <- NULL
  clobj$changes <- NULL
  clobj$cluster <- retval$cluster+1
  clobj$size <- retval$clustersize

  return(clobj)
}
```

The only arguments passed to this function is an object of type cluster that is returned from running the cclust function and a testing data set. The function first grabs all the necessary information from the data passed to it which includes the testing data dimensions as well as the cluster means. Before attempting to do anything, it checks to see that the dimensions of the new data matches up with the dimensions of the cluster means. If not, then there is a problem of dimensionality with the clusters, otherwise it will run the C function that assigns data samples to a particular cluster. Outputted from this function is a cclust object that contains the cluster information for each of the data samples passed to it, as well as the number of test data samples that belong to each cluster.

2.1.3 Print Function

The print function determines what is going to be printed out when a variable of a specific class is entered in on the terminal or is printed via the print command. Specifically, the number of clusters, the size of each cluster, and whether clustering is being done on a training or testing set. If the method used for clustering was not KMeans then the learning rate was printed out, otherwise the number of iterations and the number of changes per iteration were displayed. The code for printing out the clustering information is given below.

```
print.cclust <- function (x, ...)
{
  clobj <- x
  if (!is.null(clobj$iter))
    cat("\n                                Clustering on Training Set\n\n\n")
  else
    cat("\n                                Clustering on Test Set\n\n\n")

  cat("Number of Clusters: ", clobj$ncenters, "\n")
  cat("Sizes of Clusters: ", clobj$size, "\n\n")
  if (clobj$method!=1)
    cat("Learning Parameters:", clobj$rate.par, "\n\n")

  if (clobj$method==1)
  {
    if (!is.null(clobj$iter))
    {
      if (clobj$iter < length(clobj$changes))
        cat("Algorithm converged after", clobj$iter, "iterations.\n")
      else
        cat("Algorithm did not converge after", clobj$iter, "iterations.\n")
      cat("Changes:", clobj$changes[1:clobj$iter], "\n\n")
    }
  }
}
```

2.2 C code

2.2.1 kmeans Function

The code mentioned in Section 2.1, makes use of a call to the C code in order for it to form clusters for the data. The .C function is responsible for making the call to the C code and running that. The very first argument to the .C function determines what C function is to be called while the rest of the parameters correspond with the arguments passed to the C functions being called.

```
retval <- .C("kmeans", xrows = as.integer(xrows),
             xcols = as.integer(xcols),
             x = as.double(x), ncenters = as.integer(ncenters),
             centers = as.double(centers),
             cluster = as.integer(cluster),
             iter.max = as.integer(iter.max), iter = as.integer(iter),
             changes = as.integer(changes),
             clustersize = as.integer(clustersize),
             verbose = as.integer(verbose),
             dist = as.integer(dist - 1), PACKAGE="cclust")
```

Above is code extracted from above that makes a call to the C function kmeans. Using the rest of the parameters, the information is passed from R to the C function. Once in the C code, the function will act like normal C except using the information passed to it from the R function. Below is the C function kmeans that is called by the code above.

```
int kmeans(int *xrows, int *xcols, double *x, int *ncenters,
           double *centers, int *cluster,
           int *itermax, int *iter, int *changes,
           int *clustersize, int *verbose, int *dist)
{
    int m;
    int change;
    int *clustnew;

    clustnew = (int *) R_alloc(*xrows, sizeof(int));
    change = 1;
    *iter=0;
    while(change && ((*iter)++ < *itermax)){
        assign(xrows, xcols, x, ncenters, centers, clustnew, clustersize, dist);
        reloc(xrows, xcols, x, ncenters, centers, clustnew, clustersize, dist);
        change = 0;
        for(m=0; m<*xrows; m++){
            if(cluster[m] != clustnew[m]){
                change++;
                cluster[m] = clustnew[m];
            }
        }
        if(*verbose){
            Rprintf("Iteration: %3d    Changes: %13d \n", *iter, change);
        }
        changes[(*iter)-1] = change;
    }
    return 0;
}
```

In the above function, all the arguments passed from R are stored in pointer variables. Therefore, whenever those variables are used in a calculation they must be dereferenced.

When the functions is first called, variables used for determining the cluster number of the data point. A array of integers is allocated which will store the new cluster for the data point, a change variable is created which will hold the number of data points that changed clusters for each iteration, and finally the iteration number is set to 0 which ensures that the clustering algorithm does not go beyond the max number of iterations. For each iteration, each data point is assigned to a new cluster and the new mean of that cluster is calculated with the data points that are currently a part of that cluster. After the new cluster means are calculated, each sample in the data set is checked to see if they have been placed in a new cluster. If so, then the current cluster that the sample is in is changed and the number of changes is incremented. Depending on whether the user specified that they want text to see how the algorithm is doing for each iteration, a line is printed out stating the iteration number and the number of samples that changed cluster. Lastly, the total number of samples that changed clusters is recorded. This process will repeat until the maximum number of iterations is reached or no samples change clusters.

2.2.2 assign Function

The kmeans function, above, makes use of the functions assign and reloc which are vital for making clustering happen by the kmeans function. The first function called, the assign function, determines which cluster center each data sample is closest to and assigns that sample to that cluster. It is also responsible for determining

how many samples are a part of each cluster. More specifically the cluster cycles through all the samples and calculates the distance between the sample and the cluster mean depending on the dist parameters passed to it. If the variable dist is 0 then euclidean distance is used for calculating the distance, otherwise dist is 1 then Manhattan distance is used. For euclidean distance, the sum of squares is used for calculating the distance. The difference the data sample and the cluster mean is squared for each dimension of the data set and added to the overall distance from the cluster center. For Manhattan distance, the absolute value of the difference between the data sample and the cluster mean for each dimension is calculated and then summed together. As a shorter distance is calculated between the data sample and the cluster mean, the cluster and the minimum distance is stored. Once all data samples' clusters have been determined, the cluster size is calculated by counting up how many data samples are a part of each cluster. The code for the assign function is given below.

```
int assign(int *xrows, int *xcols, double *x, int *ncenters,
          double *centers, int *cluster, int *clustersize,
          int *dist)
{
    int k, m, n;
    double dista, mindist;

    for(k=0; k<*xrows; k++){
        mindist=10e99;

        for(m=0; m< *ncenters; m++){
            dista=0;
            for(n=0;n<*xcols;n++){
                if(*dist == 0){
                    dista += (x[k+(*xrows)*n] - centers[m + (*ncenters)*n])*(x[k+(*xrows)*n] - centers[m + (*ncenters)*n]);
                }
                else if(*dist ==1){
                    dista += fabs(x[k+(*xrows)*n] - centers[m + (*ncenters)*n]);
                }
            }

            if(dista<mindist){
                cluster[k] = m;
                mindist = dista;
            }
        }
    }

    for(k=0;k<*ncenters;k++){
        clustersize[k]=0;
    }
    for(k=0; k<*xrows; k++){
        clustersize[cluster[k]]++;
    }

    return 0;
}
```

2.2.3 reloc Function

Also used by the kmeans function, is the reloc function. The reloc function uses the new clusters, determined from the assign function, and determines the cluster means from the newly created clusters. The reloc function initially resets the cluster centers to 0 for all dimensions to prepare the new centers to be calculated.

```
int reloc(int *xrows, int *xcols, double *x, int *ncenters,
          double *centers, int *cluster, int *clustersize,
          int *dist)
{
    int k,l,m;

    /* Initialize cluster size and centers with 0 */
    for(k=0;k<*ncenters;k++){
/*      clustersize[k]=0;*/
        for(m=0;m<*xcols;m++){
            centers[k+(*ncenters)*m] = 0;
        }
    }
}
```

With the values for all dimensions of the cluster means reset to 0, the calculation of the new cluster centers can start. Depending on the value of dist, the mean is calculated in 2 different ways. If using euclidean distance, the values for each dimension of all the samples of a cluster are added together and then divided by the number of samples within that cluster. R code for determining the new cluster means is given below.

```
/*      for(k=0; k<*xrows; k++)
          clustersize[cluster[k]]++; */

if(*dist==0){
    /* Euclidean distance */
    for(k=0; k<*xrows; k++){
        for(m=0;m<*xcols;m++){
            centers[cluster[k]+(*ncenters)*m] += x[k+(*xrows)*m];
        }
    }

    for(k=0; k<*ncenters; k++){
        for(m=0;m<*xcols;m++){
            centers[k+(*ncenters)*m] /= clustersize[k];
        }
    }
}
```

The actual calculation of the mean is fairly straight forward to me, however it took me a while to determine how a one-dimensional array is holding information from a multidimensional matrix. What I believe happens, is a matrix in R code is converted to single array matrix by reading in entire columns at a time in a similar way to calling the matrix method with 1 argument which is a multidimensional matrix. In order to access data for a particular row of the matrix, the exact position of the values for each dimension of the data must be calculated.

Calculating the cluster mean from Manhattan distance is handled a bit differently. For each cluster, a vector of doubles is created that has a size equal to the number of samples in that cluster. Then for each column, all rows are cycled through. Whenever a data sample is found that belongs to the cluster currently being looked at, the value is stored in the vector of doubles created earlier. Once the vector of doubles is filled, the CC.median function is called, which will be explained a later, which grabs the mean value for that dimension. This process is repeated for all dimensions and all the clusters.

```

else if(*dist == 1){
    for(k=0; k<*ncenters; k++){
        double *xk;
        int i;
        xk = (double *) R_alloc(clustersize[k], sizeof(double));

        for (m=0; m<*xcols; m++){
            i=0;
            for (l=0; l<*xrows; l++){
                if(cluster[l]==k){
                    xk[i] = x[l+(*xrows)*m];
                    i++;
                }
            }
            centers[k+(*ncenters)*m] = CC_median(xk, &clustersize[k]);
        }
    }
}
return 0;
}

```

2.2.4 CC_median function

All the pieces of for recalculating the cluster means is given above; however, the cluster means for Manhattan distance makes use of the CC_median function. The CC_median function calculates the median value of a vector, and is in the reloc method for determining the median value of from a specific dimension for all the samples of a given cluster. First, a variable is created that will contain the median value once it has been determined. Since the subroutine CC_sort is a function defined after the CC_median function, a call to extern for CC_sort is made so that the CC_median function has the ability to call it. As for the parameter adjustment done by subtracting 1 from x, I am not entirely sure the purpose for that. It would seem to me that the x should already be at the beginning of the array and the call to CC_sort grabs the address of the second element of the array. After the CC_sort function sorts the array, the size of the list is divided by 2. Since integer division is being utilized, if the size of the list divided by 2 is even, then multiplying 2 to the resulting quotient would give the same answer as the size of the list. If that is the case, the average is taken between the middle two values. Otherwise, the middle values is considered the median.

```

/* Subroutine */ double CC_median(x, n)
double *x;
Sint *n;
{
    double xmed;
    extern /* Subroutine */ int CC_sort();
    static Sint n2;

    /* Parameter adjustments */
    --x;

    /* Function Body */
    CC_sort(n, &x[1]);
    n2 = *n / 2;
    if (n2 << 1 == *n) {
        xmed = (x[n2] + x[n2 + 1]) * (float).5;
    } else {
        xmed = x[n2 + 1];
    }
}

```

```

    return xmed;
} /* CC_median */

```

2.2.5 CC_sort Function

The CC_sort function returns an array of double values in sorted order. Which will make it easier to extract the median value. I honestly tried my best to understand how and why this function was programmed the way it was but I still do not fully understand it. When I tried to step through what the function does, it appears to place larger values on the boundaries of the array while the numbers that are most likely to be the median are found towards the middle of the list. It did not seem like any attempt was being made to actually sort the number in order from least to greatest or vice versa. Although I feel that using a doubly nested for loop would make it much easier to understand and modify if necessary, there has to be a good reason as to why this was implemented the way it was.

```

/* Subroutine */ int CC_sort(n, ra)
Sint *n;
double *ra;
{

```

```

    static Sint i--, j, l, ir;
    static double rra;
    /* Parameter adjustments */
    --ra;
    /* Function Body */
    l = *n / 2 + 1;
    ir = *n;

```

L10:

```

    if (l > 1) {
        --l;
        rra = ra[l];
    } else {
        rra = ra[ir];
        ra[ir] = ra[l];
        --ir;
        if (ir == 1) {
            ra[l] = rra;
            return 0;
        }
    }
    i-- = l;
    j = l + 1;

```

L20:

```

    if (j <= ir) {
        if (j < ir) {
            if (ra[j] < ra[j + 1]) {
                ++j;
            }
        }
        if (rra < ra[j]) {
            ra[i--] = ra[j];
            i-- = j;
            j += j;
        } else {

```

```

        j = ir + 1;
    }
    goto L20;
}
ra[i_] = rra;
goto L10;
} /* CC_sort */

```

With all the components above, the `cclust` package is able to cluster the data. There are other pieces of the `cclust` data that I have not mentioned. For instance the `cclust` package allows clustering through different means like soft and hard competitive learning. Also, different clustering indexes can be applied to the cluster after it has been run on training data to help determine the appropriate number of clusters needed. For this paper, I will not be delving deeper in to the clustering indexes or other clustering algorithms implemented by the `cclust` package because I will not be using them when calculating the edge data.

3 Simple Data Set

4 Zip Code Data

Next, `kmeans` clustering will be applied to the zip code data and check to see how accurate the clustering was. One of the main things that will be investigated in this section is how the number of clusters affects the accuracy.

4.1 Set Up

To prepare the zip code data for clustering, the same steps have to be followed as was done for preparing the zip code data to be sent to the neural network from assignment 6. First the data for both training and testing purposes had to be read in while being sure to remove the classification column from the data, since those values should not contribute in determining the classification. The training data is then split into partitions based off of how many data samples are classified as 0, 1, 2, et cetera while keeping track of the number of samples for each classification. The number of samples for the classification are important for a later step in clustering which will be explained later on. Finally, all the data samples are recombined back in to one big matrix which will be sent to the clustering algorithm when the time comes. The classifications for those data samples were also compiled together in another matrix which will be used for determining how well clustering did in terms of classification.

library(cclust)

```

XData <- read.table("zip.train", as.is=TRUE)
XData <- as.matrix(XData)
Data <- XData[, -1]
TData <- XData[, 1]

```

```

testData <- read.table("zip.test", as.is=TRUE)
testData <- as.matrix(testData)
testX <- testData[, -1]
testT <- testData[, 1]

```

```

Data0 <- Data[TData == 0,]
Data1 <- Data[TData == 1,]
Data2 <- Data[TData == 2,]
Data3 <- Data[TData == 3,]
Data4 <- Data[TData == 4,]
Data5 <- Data[TData == 5,]

```

```

Data6 <- Data[TData == 6,]
Data7 <- Data[TData == 7,]
Data8 <- Data[TData == 8,]
Data9 <- Data[TData == 9,]

count0 <- nrow(Data0)
count1 <- nrow(Data1)
count2 <- nrow(Data2)
count3 <- nrow(Data3)
count4 <- nrow(Data4)
count5 <- nrow(Data5)
count6 <- nrow(Data6)
count7 <- nrow(Data7)
count8 <- nrow(Data8)
count9 <- nrow(Data9)

trainingData <- rbind( Data0[1:round(count0 * trainingFrac),],
                      Data1[1:round(count1 * trainingFrac),],
                      Data2[1:round(count2 * trainingFrac),],
                      Data3[1:round(count3 * trainingFrac),],
                      Data4[1:round(count4 * trainingFrac),],
                      Data5[1:round(count5 * trainingFrac),],
                      Data6[1:round(count6 * trainingFrac),],
                      Data7[1:round(count7 * trainingFrac),],
                      Data8[1:round(count8 * trainingFrac),],
                      Data9[1:round(count9 * trainingFrac),])

trainingResults <- rbind( matrix(0,round(count0 * trainingFrac),1),
                          matrix(1,round(count1 * trainingFrac),1),
                          matrix(2,round(count2 * trainingFrac),1),
                          matrix(3,round(count3 * trainingFrac),1),
                          matrix(4,round(count4 * trainingFrac),1),
                          matrix(5,round(count5 * trainingFrac),1),
                          matrix(6,round(count6 * trainingFrac),1),
                          matrix(7,round(count7 * trainingFrac),1),
                          matrix(8,round(count8 * trainingFrac),1),
                          matrix(9,round(count9 * trainingFrac),1))

```

One thing I would like to note is that the trainingFrac is set to 1 so that all the data samples are used in training the clustering algorithm. I re-used the code from assignment 6 which also read in the zip code data, however, that code created a validation set from a portion of the training data.

The R code below is vital to setting up clustering. Since all the data for one classification may not fit in 1 cluster, a higher number of clusters may be needed to correctly classify the data. The variable numClustersClass determines the number of clusters that will be created for each class. For each classification of the training data, a set of random numbers are chosen and will be used to pull data samples from each classification in the data set. These samples will be then placed in the matrix initCenters which will serve as the initial clusters for the kmeans clustering algorithm.

```

numClustersClass <- 25
mean0 <- sample(count0, numClustersClass)
mean1 <- sample(count1, numClustersClass)
mean2 <- sample(count2, numClustersClass)
mean3 <- sample(count3, numClustersClass)
mean4 <- sample(count4, numClustersClass)

```

```

mean5 <- sample(count5, numClustersClass)
mean6 <- sample(count6, numClustersClass)
mean7 <- sample(count7, numClustersClass)
mean8 <- sample(count8, numClustersClass)
mean9 <- sample(count9, numClustersClass)

```

```

initCenters <- rbind(Data0[mean0,],
                    Data1[mean1,],
                    Data2[mean2,],
                    Data3[mean3,],
                    Data4[mean4,],
                    Data5[mean5,],
                    Data6[mean6,],
                    Data7[mean7,],
                    Data8[mean8,],
                    Data9[mean9,])

```

With the data all set up to be clustered, all that is left to do is call the clustering method. In the R code given below, the clustering method is called. Depending on what clustering algorithm is needed with specific parameters, only a portion of the parameters need to be set. Otherwise, as mentioned in Section 2.1, all parameters can be set. For this call to the clustering method all that is need is the training data, the initial cluster centers, the number of max iterations, and the boolean TRUE for verbose which prints out details on how the data set is being clustered.

```

cl <- cclust(trainingData, initCenters, iter.max = 500, verbose = TRUE)

```

Finally, to check how accurate clustering was in predicting data, the classification must be determined based off of what cluster the data sample is placed in. The classification method takes a cluster number or returns the classification associated with it. For instance when dealing with 10 clusters per classification, any data sample found in clusters 1 to 10 should be classified as classification 0 because the first 10 initial clusters were taken from data samples with a classification of 0.

```

classification <- function(clusterData)
{
  num <- floor(clusterData/numClustersClass)
  if (clusterData %% numClustersClass == 0)
    num <- num - 1
  num
}

results <- matrix(cl$cluster)
classificationResults <- apply(as.matrix(results), 1, classification)
compareResults <- abs(classificationResults - trainingResults)
correctResults <- apply(as.matrix(compareResults), 1, function(ps) all(ps == 0))
percent <- length(as.matrix(classificationResults)[correctResults,])/length(results)

```

To determine the classification of the data from clustering, the cluster that the data samples belongs in must be passed in to the classification method which will return the classification for it. Once the cluster of the data samples is converted to an actual classification, the predicted classification is compared with the actual classification by subtracting one from the other and seeing if that difference is 0, which indicates that the predicted and actual classification match.

4.2 Predictive Abilities

Table 1 is a table containing the training and testing accuracy for different sets of clusters. As expected the accuracy for both the training and testing set appear to be greatly influenced by the number of clusters

| Number of Clusters | Training Accuracy | Testing Accuracy |
|--------------------|-------------------|------------------|
| 1.0000000 | 0.6846797 | 0.6382661 |
| 5.0000000 | 0.7841174 | 0.7483807 |
| 10.0000000 | 0.8307502 | 0.8021923 |
| 20.0000000 | 0.8949390 | 0.8659691 |
| 30.0000000 | 0.8979564 | 0.8744395 |
| 40.0000000 | 0.9166095 | 0.8844046 |
| 50.0000000 | 0.9281306 | 0.9018435 |
| 80.0000000 | 0.9419833 | 0.9138017 |

Table 1: Training and Testing Accuracy for Different Number of Clusters

for each classification. Initially, having only 1 cluster for each classification did fairly badly in correctly classifying the data. While having a high number of clusters for each classification provided more accurate results. I believe that the results could fluctuate quite a bit depending on what samples are chosen for the initial clusters. While having a high number of clusters for classification may be good generally, if the random samples chosen for each classification are fairly similar and only differ by 1 dimension out of the 256 available, then clustering would probably provide results closer to having a small number of clusters per sample.

I am a little surprised to see how well the clustering algorithm did with 1 one cluster per classification. I expected it to do quite a bit worse than what the results show for 1 cluster in Table 1. Also, there has to be a cap as to how many clusters are used for clustering algorithm. If too many are given, then the algorithm would be extremely similar to the K nearest neighbors algorithm. The classification for test samples would be based off of the closest sample.

4.3 Hand Drawn Digits

Just as was done in Assignment 6, kmeans clustering will attempt to classify hand drawn digits. This proved to be a struggle for the neural network because so many different factors influenced what the expected classification was for the digit drawn. This included the location of the drawing in the window, the actual size of the window, and the overall style such as having the number 4 where the line segments at the top meet.

4.3.1 Set Up

Since the code was used from assignment 6, not much had to be changed. The ability to draw the digit, view the digit drawn, and displaying a bar graph indicating the value chosen still remain the same. However, code had to be implemented to handle the change from using neural networks to using clustering algorithms.

Two files were used in order to have the interactive display function. This function is used to actually classify the data and returns a numeric vector containing a 1 for the classification of the hand drawn digit when it comes. This function takes 2 parameters as arguments to it. The first one refers to the predicted cluster that the drawn data is associated with. The second argument represents the number of clusters for each classification. Similar to the classification function explained in Section 4.1, this function initially creates a numeric vector containing all zeros. From the cluster that the data is associated with, the actual classification of that cluster is determined and a 1 is placed at that position to indicate what predicted classification made by the clustering algorithm was. One thing to note is that the classifications start at 0 but the numerical vector index starts at 1, so 1 must be added to the actual classification so that the bar is drawn in the correct place.

```
classification <- function(clusterData, numClustersClass)
{
  class <- rep(0,10)
  num <- floor(clusterData/numClustersClass)
```

```

    if ((clusterData %% numClustersClass) == 0)
    {
        num <- num - 1
    }
    class[num+1] <- 1
    class
}

```

Another function used, one that creates a set amount of clusters based off of the argument being passed to it. This function wraps the code discussed in Section 4.1 and wraps that in a function so that it can be called by the source R code that contains all the interactive display code. To start the, zip.train and zip.test data set are read in to the same data set and there classification values are removed from the data set. The data set is partitioned based off of their classifications, and a number, specified by the argument num, of random samples are taken from each classification and used as the initial centers for the clustering algorithm. Then, the clustering algorithm is run, and the results are returned in a list.

```

clusterInfo <- function(num)
{
    numClustersClass <- num

    ## Read zip.train and zip.test Data

    ## Split Data according to classification

    ## Randomly select numClustersClass centers from each classification and store in initCenters

    cl <- cclust(trainingData, initCenters, iter.max = 500, verbose = TRUE)

    results <- list(cluster=cl)
    results
}

```

Before anything is done, the source code that contains the code to that clusters the training data must be sourced. The source code provides the code above which is necessary for forming clusters and classifying the hand drawn digit. A number of clusters is passed to the clusterInfo function which forms the clusters and returns the cluster back to be used for predicting the classification of the hand drawn digit.

```

source("clusterInteractiveDisplay.R")
numClusters <- 1

clusterResults <- clusterInfo(numClusters)
cl <- clusterResults$cluster

```

The processAndClassify function is used to update each individual window. One window is used for drawing the actual digit, the other displays a bar graph indicating what the predicted classification for the digit is, and the last one displays a representation of the number just drawn in regards to how the data will represent it. First, the window that contains a representation of the user's drawing is read in to a matrix which represents a data sample. The newly formed data sample is assigned to a particular cluster and then passed to the classification function to determine the classification of the data sample's assigned cluster. Returned from the classification is a vector of numbers which is used to display a bar graph. Since no probabilities are being dealt with, the position with a bar reaching to 1 is the predicted classification for that hand drawn digit.

```

processAndClassify <- function(nnet) {
    dev.set(which=dev.next())
    x <- t(matrix(drawImage()))
    ### make bar plot with probs
}

```

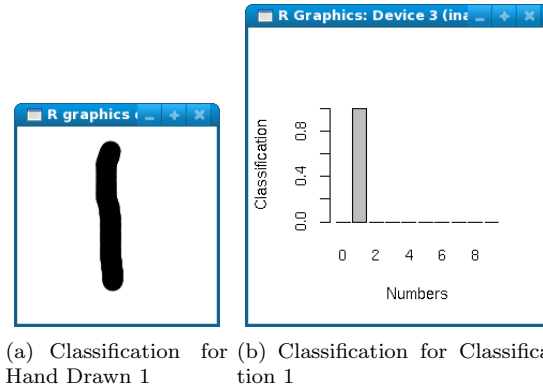


Figure 1: Hand Drawn Classification Based off of 1 Cluster per Classification

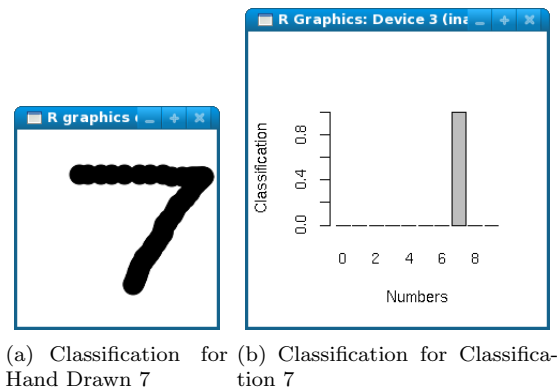


Figure 2: Hand Drawn 7 Classification Based off of 1 Cluster per Classification

```

dev.set(which=dev.next())
probs <- predict(cl, matrix(x, nrow=1))
probs <- classification(probs$cluster, numClusters)
barplot(probs, names.arg=c("0", "1", "2", "3", "4", "5", "6", "7", "8", "9"), xlab="Numbers", ylab="
### draw digit that is class label of x
dev.set(which=dev.next())
}

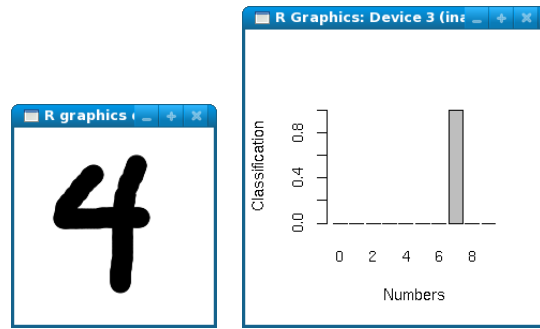
```

4.3.2 Discussion

Classification was done on Hand Drawn Digits using 2 different numbers for clusters per classification. The first experiment used a cluster algorithm with 1 cluster per classification to see how accurate it would be when it was unable to correctly classify outliers. As expected, classification was not very accurate when 1 cluster was used. It appeared that for just about any number drawn, a small subset of the possible classifications were actually predicted as the classification.

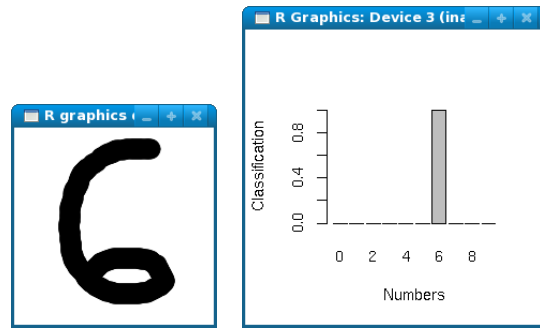
Looking at Figure 1 and Figure 2, it appeared that using 1 cluster per classification was doing great at predicting the actual value. However, this all changed after looking at Figure 3, which was classified incorrectly. After trying various other hand drawn digits in different locations and different sizes, it seemed that a classification of 1 and a classification of 7 were the only possible options for classifications.

Figure 4, Figure 5, and Figure 6 display the hand drawn digit as well as its classification predicted by clustering. From Figure 4 and Figure 5, using 80 clusters per classification provides accurate results for digits that were not classified correctly when using 1 cluster per classification. However, there is still a long way before the classification can be deemed as being accurate. As shown in Figure 6, the classification is



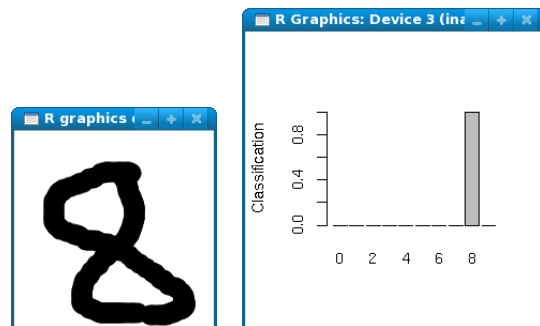
(a) Classification for Hand Drawn 4 (b) Classification for Classification 4

Figure 3: Hand Drawn 4 Classification Based off of 1 Cluster per Classification



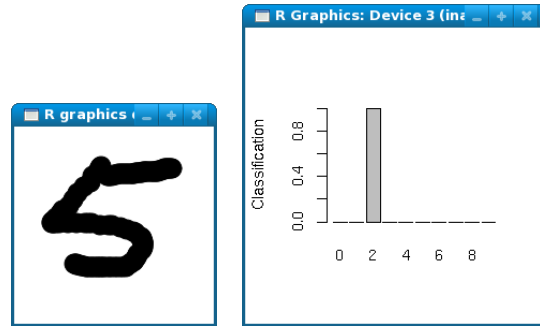
(a) Classification for Hand Drawn 6 (b) Classification for Classification 6

Figure 4: Hand Drawn 6 Classification Based off of 80 Cluster per Classification



(a) Classification for Hand Drawn 8 (b) Classification for Classification 8

Figure 5: Hand Drawn 8 Classification Based off of 80 Cluster per Classification



(a) Classification for Hand Drawn 5 (b) Classification for Classification 5

Figure 6: Hand Drawn 5 Classification Based off of 80 Cluster per Classification

not correct. From my trials, classifications of 3 and 5 were rarely seen if at all. Similar to the results seen from applying the neural network to the zip code data, it seems that a lot of work needs to be done in order to get it correctly classify any digit with any size at any location on the drawing screen.

5 Conclusion

Taking a look at how clustering was implemented was pretty interesting. I must admit that for a majority of their code, I would have had the same approach to attack the goal of each function. However, I would have taken a totally different approach to how the `CC_median` function was implemented. Not to say that their implementation was wrong, but I found it very hard to understand what the benefit for that implementation. Having mentioned that, I'm sure there is a valid reason as to why they took that approach to implementing that function. Given the size of some of the data sets used in the past, it may be very efficient.

I thought that determining the classification of a particular data set was difficult to determine. Clustering assumes that all the data for a particular classification is placed right around each other. This assumption can fail given data samples that are very wide spread or that are mixed with each other which clustering the zip code demonstrates. The only way to possibly combat this, would mean to come up with as many clusters as there are data samples and take a k nearest neighbors approach, but that was not what I was shooting for in this last assignment. As always I was hoping that the data would be effectively classified, but yet again, I must continue my search for an algorithm that will accurately classify the data despite the size and location of a number. The search must go on!