Mark Priestley

Student ID – 19375096

I used this code to acquire matrices “d” and “nbrs256”, and functions “classify” and “findnns”, which are used in the questions.

```
d0=scan("digits/Zl0d.dat",nlines=1000,n=256000)
d1=scan("digits/Zl1d.dat",nlines=1000,n=256000)
d2=scan("digits/Zl2d.dat",nlines=1000,n=256000)
d3=scan("digits/Zl3d.dat",nlines=1000,n=256000)
d4=scan("digits/Zl4d.dat",nlines=1000,n=256000)
d5=scan("digits/Zl5d.dat",nlines=1000,n=256000)
d6=scan("digits/Zl6d.dat",nlines=1000,n=256000)
d7=scan("digits/Zl7d.dat",nlines=1000,n=256000)
d8=scan("digits/Zl8d.dat",nlines=1000,n=256000)
d9=scan("digits/Zl9d.dat",nlines=1000,n=256000)

d=c(d0,d1,d2,d3,d4,d5,d6,d7,d8,d9)

d=matrix(d,256,10000)

findnns =  function(d, x)
{
  d <- d-x # Takes away the image x for each image in d
  d = d^2 # Squares for euclidean distance
  dists=apply(d,2,sum) #Sums the columns (the 2 parameter specifies columsn) for the total distance
  dorder=order(dists)
  dorder[1:40]
}

classify = function(nns, k)
{
  labels =
c(rep("0",1000),rep("1",1000),rep("2",1000),rep("3",1000),rep("4",1000),rep("5",1000),rep("6",1000),rep("7",
1000),rep("8",1000),rep("9",1000))
  digits = labels[nns[2:k]]
  t = table(digits)
  m = which.max(t)
  names(m)
}

nbrs256=matrix(0,40,10000)
system.time(
  for (i in 1:10000)
  {
    nbrs256[,i]=findnns(d,d[,i])
  }
)

saveRDS(nbrs256,file="nbrs256.rds")
saveRDS(d,file="d.rds")
saveRDS(findnns,file="findnns.rds")
saveRDS(classify,file="classify.rds")
```

Question 1

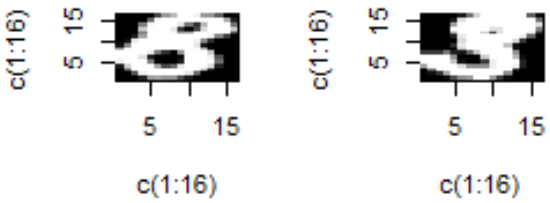
After setting k = 2, I found the confusion matrix. The off diagonal elements are the incorrectly classified digits

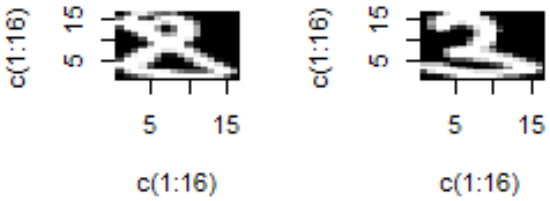
```
nbrs256 = readRDS("nbrs256.rds")
d = readRDS("d.rds")
classify = readRDS("classify.rds")
findnns = readRDS("findnns.rds")

labels =
c(rep("0",1000),rep("1",1000),rep("2",1000),rep("3",1000),rep("4",1000),rep("5",1000),rep("6",1000),rep("7",
1000),rep("8",1000),rep("9",1000))
k = 2
predicted = apply(nbrs256,2, function x classify(x,k))
t = table(predicted,labels)
```

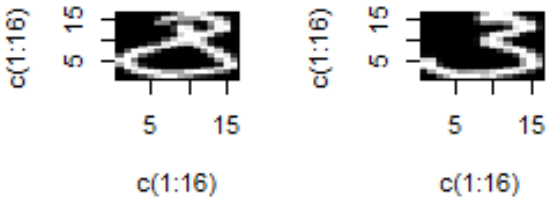
```
> print(t)
         labels
predicted   0   1   2   3   4   5   6   7   8   9
        0 981   0   1   0   1   0   0   1   0   1
        1   8 998   1   0   1   0   0   6   2   2
        2   1   0 991   7   0   0   0   1  13   2
        3   0   0   1 981   0   6   0   0  16  17
        4   1   1   0   0 987   0   0  10   0  12
        5   0   0   1   3   0 984   3   0  11   7
        6   2   0   0   0   1   6 997   0   7   1
        7   1   1   2   1   4   0   0 975   1   3
        8   4   0   1   5   0   2   0   0 943   5
        9   2   0   2   3   6   2   0   7   7 950
```

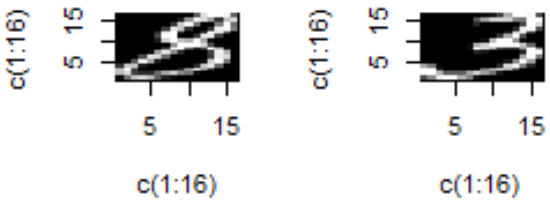
The most incorrectly classified number is 8, which is classified correctly 94.3% of the time. I looked at 4 instances where the nearest neighbour to 8 was a 3.

```
par(mfrow=c(4,2))
group = which(labels == "8" & predicted == "3")
j = 1
for(i in group)
{
 z=matrix(d[,i],16,16)
 image(c(1:16),c(1:16),z[,c(16:1)],col=gray(c(0:255)/255))
 z=matrix(d[,nbrs256[2,i]],16,16)
 image(c(1:16),c(1:16),z[,c(16:1)],col=gray(c(0:255)/255))
 j=j+1
 if (j > 4)
 {
  break
 }
}
```









The stroke width and position of the 3s are very similar to the 8s, which would explain why they were misclassified.

Including more nearest neighbours may improve the effectiveness of the algorithm, as the unlikely similarity of the strokes won't occur with more neighbours.

Question 2

I decided to look at 8s that were misclassified as 3s

```
d3=scan("C:/Users/User/Desktop/ca274/digits/Zl3d.dat",nlines=1000,n=256000)
d8=scan("C:/Users/User/Desktop/ca274/digits/Zl8d.dat",nlines=1000,n=256000)

d=c(d3,d8)
d=matrix(d,256,2000)
d=t(d)

## Need to find the digits that are wrong
classify = readRDS("assessment_1/classify.rds")
nbrs256 = readRDS("assessment_1/nbrs256.rds")

k = 2
predicted = apply(nbrs256,2, function(x) classify(x,k))
t = table(predicted,labels)

## I will look at 8s classified as 3s
misclassified = which(predicted == "3" & labels == "8")
misclassified=misclassified-7000
```

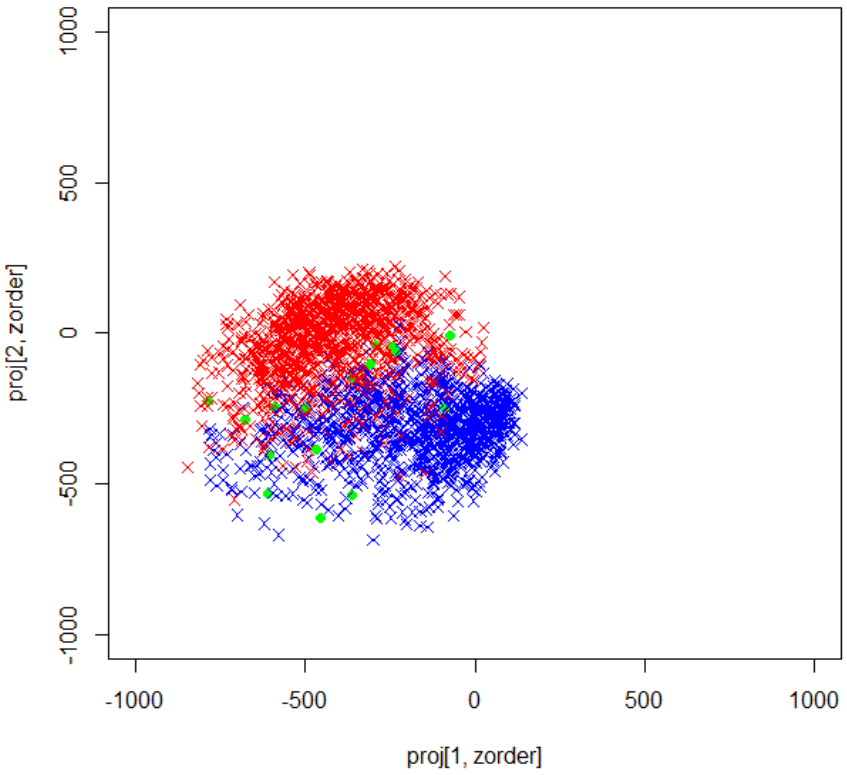
I needed to find the eigenvectors for these digits to plot the points.

I set the misclassified points at the colours vector equal to “green”, and made a vector of plotting symbols. I made all 2000 elements of the ‘mypch’ vector equal to a plus sign (4), and set the misclassified elements equal to 19 (solid circle).

```
c=var(d)
e=eigen(c)

p = d %*% e$vectors

theta = 0.25
rot=matrix(c(cos(theta),sin(theta),-sin(theta),cos(theta)),2,2)
rev=matrix(c(cos(theta),-sin(theta),sin(theta),cos(theta)),2,2)

colours = c(rep("red",1000),rep("blue",1000))
colours[misclassified] = "green"
mypch=c(rep(4,2000))
mypch[misclassified]=19
```

This code plots the points, the green circles are the 8s that were misclassified as 3s., and would produce the image below

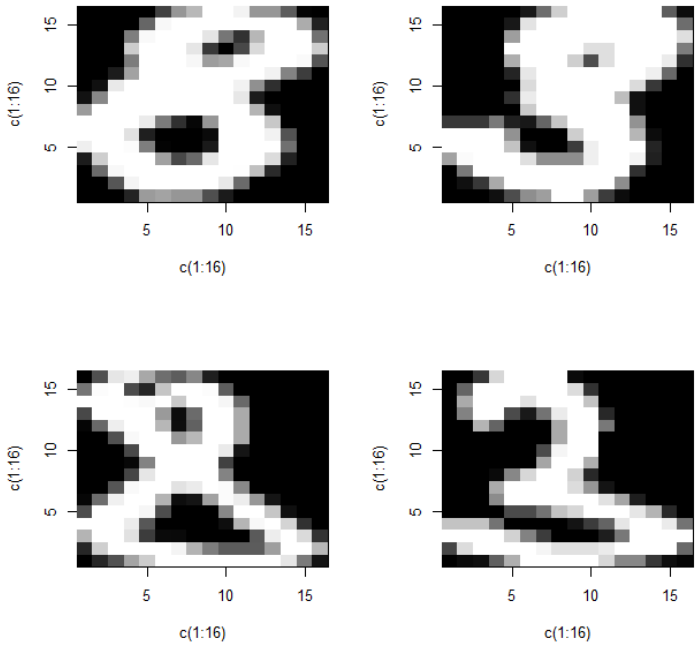
```
p=t(p)
proj=matrix(0,2,10000)

for(i in c(1:3000))
{

pos = locator(1)

if(pos$x < 0)
{
if(pos$y < 0) p[c(1,3),] = rot %*% p[c(1,3),] else  p[c(1,2),] = rot %*% p[c(1,2),]
}
else
{
if(pos$y < 0) p[c(1,3),] = rev %*% p[c(1,3),] else p[c(1,2),] = rev %*% p[c(1,2),]
}


proj[1,] = p[1,]*4000/(9000-p[3,])
proj[2,] = p[2,]*4000/(9000-p[3,])

zorder = order(p[3,])

plot(proj[1,zorder],proj[2,zorder],xlim=c(-1000,1000),ylim=c(-
1000,1000),col=colours[zorder],pch=mypch[zorder])

}
```



The majority of the points of the misclassified images are on the 'border' between the two clouds,

which would explain why they were misclassified. However, there are a few (especially in the bottom left of the cloud of the image above) which are very clearly in the correct cloud. I decided to take 2 of these points and see why were they misclassified.

```
group=identify(proj[1,zorder],proj[2,zorder],n=2)
group=zorder[group]

dev.new()
d=t(d)
par(mfrow=c(2,2))
for(i in group)
{
z=matrix(d[,i],16,16)
image(c(1:16),c(1:16),z[,c(16:1)],col=gray(c(0:255)/255))
z=matrix(d[,nbrs256[2,i+7000]-3000],16,16)
image(c(1:16),c(1:16),z[,c(16:1)],col=gray(c(0:255)/255))
readline()
}
```

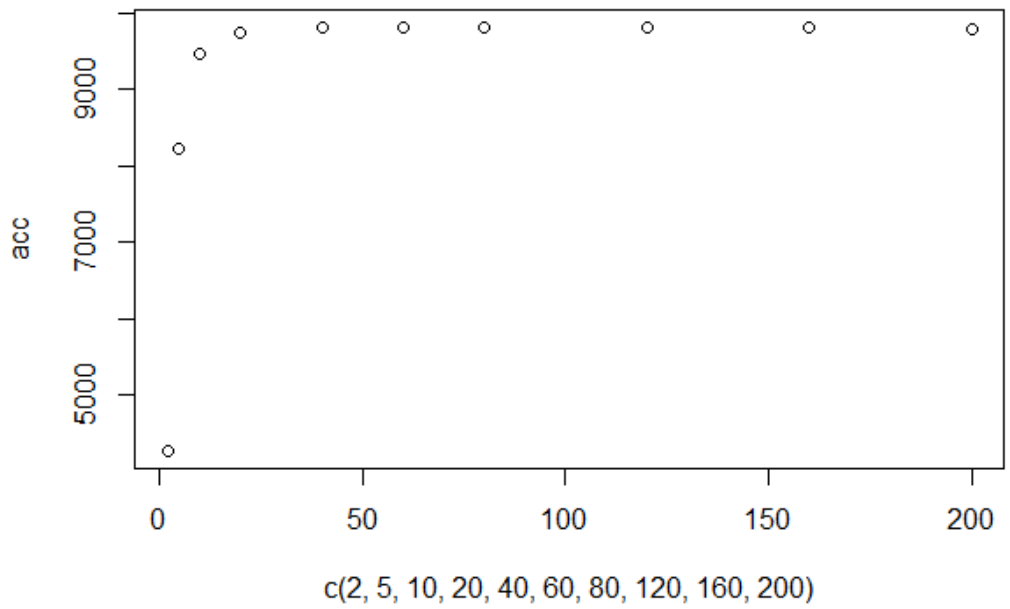


From the images, you can see that these are normal 8s. The nearest neighbour is unusually close to it, in terms of stroke length and position, which explains why they were misclassified.

Question 3

I ran the knn code with 2,5,10,20,40,60,80,120,160,200 eigenvectors. I found the accuracy using these eigenvectors with k = 4.

```
d = readRDS("d.rds")
classify=readRDS("classify.RDS")
findnns=readRDS("findnns.RDS")

dim(d)

d=t(d)
c=var(d)
e=eigen(c)
plot(e$values)
p=d %*% e$vectors[,1:210]


p=t(p)
labels=c(rep("0",1000),rep("1",1000),rep("2",1000),rep("3",1000),rep("4",1000),rep("5",1000),rep("6",1000),re
p("7",1000),rep("8",1000),rep("9",1000))
acc=c()

for(n in c(2,5,10,20,40,60,80,120,160,200))
{

 nbrsp20 = matrix(0,40,10000)
 for(i in c(1:10000))nbrsp20[,i]=findnns(p[1:n,],p[1:n,i])
 results = apply(nbrsp20,2,function(x) classify(x,4)) ## with k = 4
 t1=table(results,labels)
 acc=append(acc,sum(diag(t1)))
}
plot(c(2,5,10,20,40,60,80,120,160,200),acc)
```



The accuracy grows exponentially with <10 eigenvectors. The accuracy levels off and grows minimally after about 10 eigenvectors.

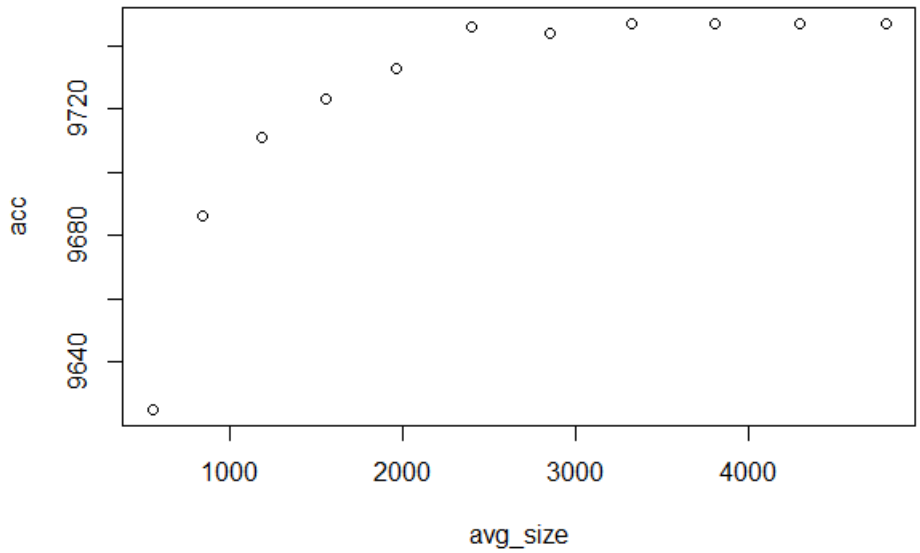
This graph would suggest that between 10-20 eigenvectors is the most effective number of eigenvectors, as it is the start of the point at which the accuracy levels off. Including more eigenvectors would cause execution time to go up with minimal to no improvement in the accuracy of the algorithm.

Question 4

I ran the code selecting digits in a box inside a for loop, which adjust the size of the box at each iteration. I found the size of the box for each point, and appended the average of all points to a vector. I then plotted the average size vs the accuracy of those points.

```
d = readRDS("d.rds")
classify=readRDS("classify.RDS")
findnns=readRDS("findnns.RDS")

d=t(d)
c=var(d)
e=eigen(c)
plot(e$values)
p=d %*% e$vectors[,1:20]
p=t(p)

## Finding the most accurate box size
acc=0
avg_size=0
labels =
c(rep("0",1000),rep("1",1000),rep("2",1000),rep("3",1000),rep("4",1000),rep("5",1000),rep("6",1000),rep("7",
1000),rep("8",1000),rep("9",1000))
j=1
for(x in seq(0,500,50))
{
 nbrsbox = matrix(0,40,10000)
 box_length=0
 system.time(
  for(i in c(1:10000))
  {
   box=which(abs(p[1,]-p[1,i]) < (250 + x) & abs(p[2,]-p[2,i]) < (150 + x))
   box_length[i]=length(box)
   nnb=findnns(p[1:20,box],p[1:20,i])
   nbrsbox[,i]=box[nnb]
  }
 )
 print(paste("Size of box:",length(sum(box_length)/length(box_length))))
 avg_size[j]=sum(box_length)/length(box_length)

 results=0
 for(i in c(1:10000))
 {
  results[i]=classify(nbrsbox[,i],4)
 }
 t=table(results,labels)
 accur=sum(diag(t))
 print(paste("Accuracy:",accur))
 acc[j]=accur
 writeLines("")
 j=j+1
}
plot(avg_size,acc)
```

This code produced the graph below.



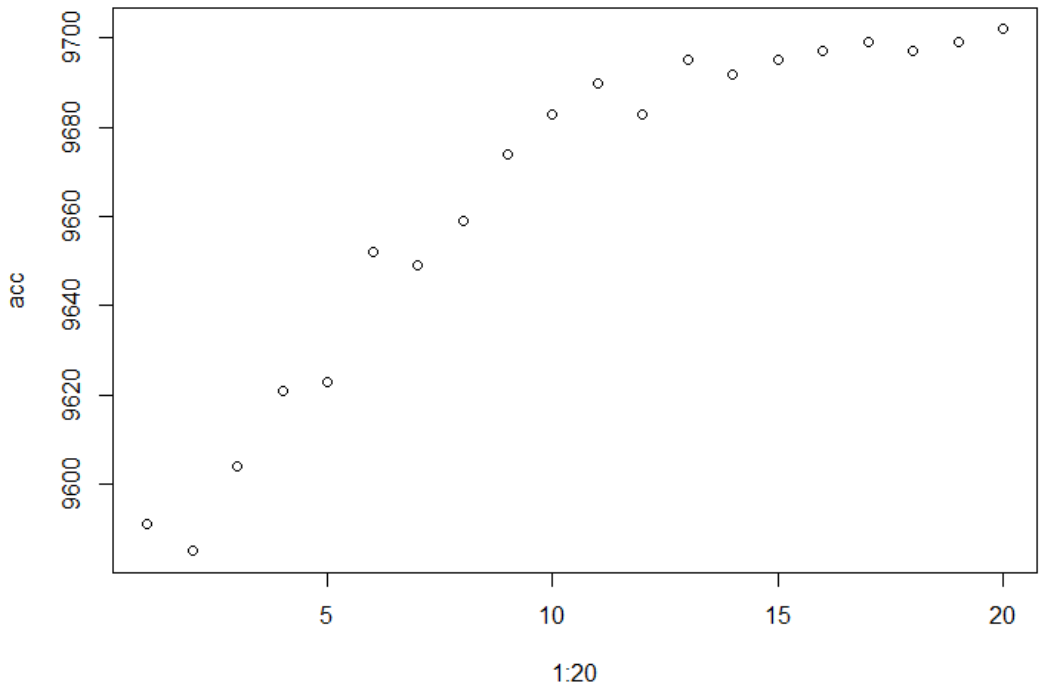
The accuracy levels off when the average box size is about 2500. Therefore, using the line of code "box=which(abs(p[1,]-p[1,i]) < (500) & abs(p[2,]-p[2,i]) < (400))" should produce the best balance of execution speed and accuracy, as the larger the box size is, the longer the execution time is going to be.

Question 5

I firstly ran the knn code with blurring inside a for loop. On each iteration, I changed the level of blurring, and recorded the accuracy.

I then plotted the blurring against the accuracy at that level of blurring.

```
## Try to find optimum blurring
acc=0
for (j in 1:20)
{
width = j
ii = c(0:15)
ii[9:16] = 17 - c(9:16)
ii = ii^2
ex = exp(-ii/width)
gau = ex %*% t(ex)

x2=d
x=d
for(i in c(1:10000))
{
  z = matrix(as.numeric(x[i,]),16,16)
  ft=fft(z)
  ft = ft *gau
  z2=fft(ft,inverse=T)
  x2[i,]=matrix(abs(z2),1,256)
}

#this sets the mean of the blurred data to zero
mx2=apply(x2,2,mean)
x2=sweep(x2,2,mx2)
x2=256*x2/max(x2)

#this calculates the eigenvectors of the blurred data
c=var(x2)
e=eigen(c)
pblr=x2 %*% e$vectors[,1:3]

#this calculates the nearest neighbours within a box determined by the blurred data
pblr=t(pblr)
nbrsblrbox = matrix(0,40,10000)
for(i in c(1:10000))
{

  box=which(abs(pblr[1,]-pblr[1,i]) < 250 & abs(pblr[2,]-pblr[2,i]) < 250)

  if(length(box) < 100)
    box=which(abs(pblr[1,]-pblr[1,i]) < 500 & abs(pblr[2,]-pblr[2,i]) < 500)

  #we use the unblurred eigenvectors to find the neighbours
  nnb=findnns(p[1:20,box],p[1:20,i])
  nbrsblrbox[,i]=box[nnb]
}

results = apply(nbrsblrbox,2,function(x) classify(x,4))
t=table(results,labels)
acc[j]=sum(diag(t))

print(paste("The accuracy of",j,"is",sum(diag(t))))
}
plot(1:20,acc)
```

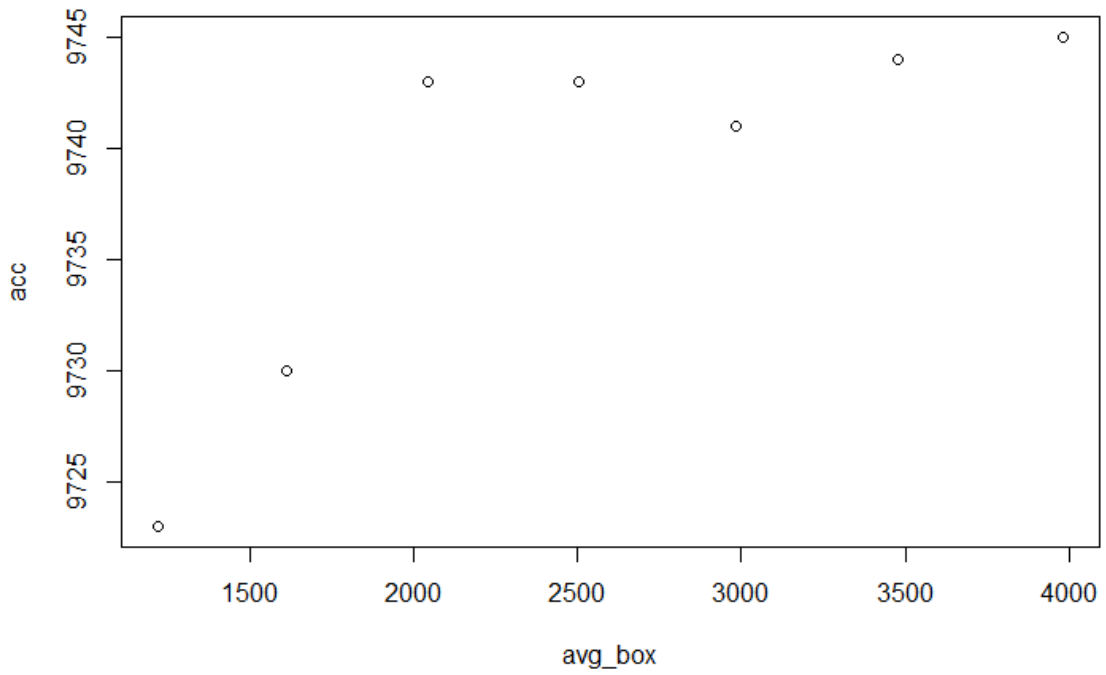


The accuracy levels off when the variable “width” is about 11. Therefore, the optimum level of blurring for the dataset is about 11, as any greater would make execution speed longer, with very little gains in accuracy.

I then tried looking at digits within a box at that blurring level, similarly to q4.

```
## Read in d
d=readRDS("d.RDS")
findnns=readRDS("findnns.RDS")
classify=readRDS("classify.RDS")
d=t(d)

## Find p
c=var(d)
e=eigen(c)
plot(e$values)
p=d %*% e$vectors[,1:20]
p=t(p)

acc=0
avg_box=0
k=1
for(j in seq(250,600,50))
{

 #this calculates the nearest neighbours within a box determined by the blurred data
 nbrsblrbox = matrix(0,40,10000)
 box_length=0
 for(i in c(1:10000))
 {

  box=which(abs(pblr[1,]-pblr[1,i]) < j & abs(pblr[2,]-pblr[2,i]) < j)
  box_length[i]=length(box)

  #we use the unblurred eigenvectors to find the neighbours
  nnb=findnns(p[1:20,box],p[1:20,i])
  nbrsblrbox[,i]=box[nnb]
 }
 results = apply(nbrsblrbox,2,function(x) classify(x,4))
 t=table(results,labels)
 print(paste("Size of box:",length(box)))
 print(paste("Accuracy:",sum(diag(t))))
 avg_box[k]=sum(box_length)/length(box_length)
 acc[k]=sum(diag(t))
 k = k + 1
}
plot(avg_box,acc)
```

The accuracy levels off when the average length of “box” is about 2000. Therefore, the line of code best suited for selecting the box with the blurred data is “box=which(abs(pblr[1,]-pblr[1,i]) < 400 & abs(pblr[2,]-pblr[2,i]) < 400)”



Combining the optimum blurring and box selecting, you get the final code.

```
## Applying optimum blurring and the best sized box

proc.time()
## Read in d
d=readRDS("d.RDS")
findnns=readRDS("findnns.RDS")
classify=readRDS("classify.RDS")
d=t(d)

## Find p
c=var(d)
e=eigen(c)
plot(e$values)
p=d %*% e$vectors[,1:20]
p=t(p)

#this is the code to decide how much the images are blurred
width = 11
ii = c(0:15)
ii[9:16] = 17 - c(9:16)
ii = ii^2
ex = exp(-ii/width)
gau = ex %*% t(ex)

#this is the code that does the blurring
#the blurred images are called x2
x2=d
x=d
for(i in c(1:10000))
{
  z = matrix(as.numeric(x[i,]),16,16)
  ft=fft(z)
  ft = ft *gau
  z2=fft(ft,inverse=T)
  x2[i,]=matrix(abs(z2),1,256)
}

#this sets the mean of the blurred data to zero
mx2=apply(x2,2,mean)
x2=sweep(x2,2,mx2)
x2=256*x2/max(x2)

#this calculates the eigenvectors of the blurred data
c=var(x2)
e=eigen(c)
pblr=x2 %*% e$vectors[,1:11]

#this calculates the nearest neighbours within a box determined by the blurred data
pblr=t(pblr)
nbrsblrbox = matrix(0,40,10000)
for(i in c(1:10000))
{

  box=which(abs(pblr[1,]-pblr[1,i]) < 400 & abs(pblr[2,]-pblr[2,i]) < 400)

  #we use the unblurred eigenvectors to find the neighbours
  nnb=findnns(p[1:20,box],p[1:20,i])
  nbrsblrbox[,i]=box[nnb]
}

labels =
c(rep("0",1000),rep("1",1000),rep("2",1000),rep("3",1000),rep("4",1000),rep("5",1000),rep("6",1000),rep("7",1000),rep("8",1000),rep("9",
1000))


results = apply(nbrsblrbox,2,function(x) classify(x,4))
t=table(results,labels)
print(t)
print(paste("Accuracy:",sum(diag(t))))
proc.time()
```

Running all of this code, you get an accuracy of 97.4%, with time taken being,

user	system	elapsed
63.25	1.18	83.40

This is about 15 mins faster than finding the 40 nearest neighbours using all 256 pixels of each image.