MATH 504 HW10

Jeff Gould

3/28/2020

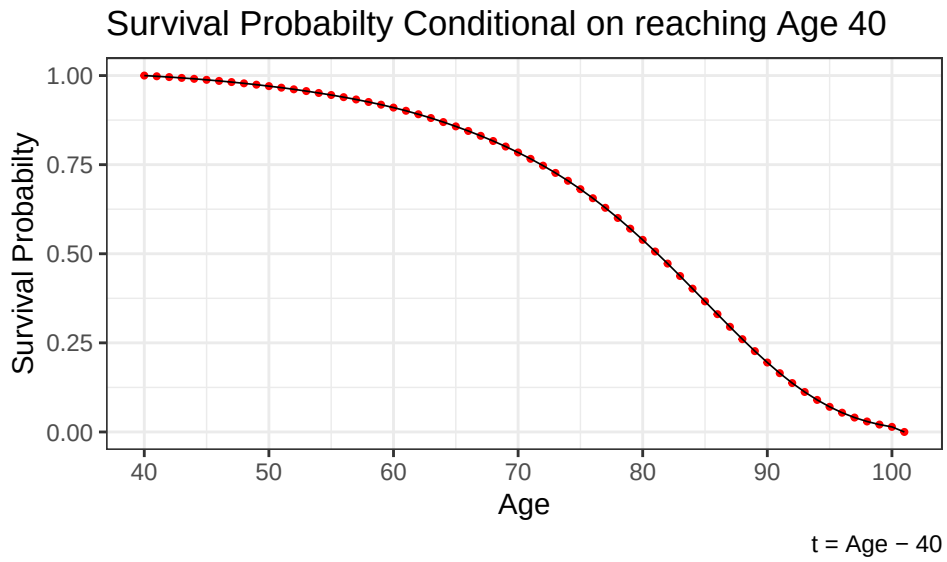
2

a

```
tic("Total Run Time")

life_exp <- readxl::read_xls("US Life Expectancy 2003.xls", range = "A7:B108")

prob_survival_40 <- life_exp %>%
  filter(as.numeric(str_sub(Age,1,2)) >=39 | as.numeric(str_sub(Age,1,3))== 100) %>%
  mutate(qx = ifelse(Age == "39-40",0,qx)) %>%
  mutate(prob_surv = 1 - qx,
         cum_prob_surv = cumprod(prob_surv),
         t = row_number() - 1)
```

	Age	qx	prob_surv	cum_prob_surv	t
1	39-40	0.00000	1.00000	1.00000	0.00000
2	40-41	0.00204	0.99796	0.99796	1.00000
3	41-42	0.00221	0.99779	0.99576	2.00000
4	42-43	0.00239	0.99761	0.99338	3.00000
5	43-44	0.00259	0.99741	0.99081	4.00000
6	44-45	0.00282	0.99718	0.98801	5.00000
7	45-46	0.00306	0.99694	0.98498	6.00000
8	46-47	0.00332	0.99668	0.98171	7.00000
9	47-48	0.00359	0.99641	0.97819	8.00000
10	48-49	0.00386	0.99614	0.97441	9.00000

```
ggplot(data = prob_survival_40, aes(x = t + 40, y = cum_prob_surv)) +
  geom_point(color = "red", size = 0.75) +
  ggformula::geom_spline(size = .3) +
  theme_bw() +
  labs(x = "Age", y = "Survival Probabilty",
       title = "Survival Probabilty Conditional on reaching Age 40",
       caption = "t = Age - 40") +
  scale_x_continuous(breaks = seq(40,100,10))
```



b

The expected value of a function $g(x)$ of discrete random variable X is $\sum g(x)\text{pmf}(X)$ over the support of X. Here our X is time past age 40, $g(X)$ is the Present Value of the payments, $PV(t)$, and the pmf is $L()$. So

$$E[PV] = \sum_{i=1}^{\infty} PV(i)L(i/12) \approx \sum_{i=1}^{732} 200e^{-.05i/12}L(i/12)$$

We use $L(i/12)$ as $L(t)$ is expressed in terms of years past 40, but i is months past 40, so $i/12$ is fraction of years lived past 40. We use 61 years = 732 months as our upper bound as that implies the person lived past 100, the end range of our data, which assigns survival probabilty of 0 for living beyond this point. Since the equation in the HW uses $i = 1$ as the starting point on the sum, we are calculating this with the parameter that the first payment on the policy is made 1 month from the starting date, and there is no payment due upon signing the policy.

```
attach(prob_survival_40)
L_t <- splinefun(x = t, y = cum_prob_surv, method = "natural")
detach(prob_survival_40)

E_PV <- function(m){
  i <- seq(1,m,1)
  e_pv <- sum(200 * L_t(i/12) * exp(-0.05 * i / 12))
  return(e_pv)
}

ExpPresVal <- E_PV(732)
```

The Expected Present Value of the life insurance policy is $39373.04

3

a

```r
tic("Load and Clean mnist_train.csv")
norm <- function(x){sqrt(sum(x^2))}

mnist_train <- data.table::fread("mnist_train.csv", header = F, sep = ",")

X <- as.matrix(mnist_train[,-1])
Numbers <- mnist_train[,1]

mu <- colMeans(X)

X_center <- t(apply(X, 1, function(x) {
  return(x - mu)
}))
rm(mnist_train)
toc()
```

```
## Load and Clean mnist_train.csv: 5.755 sec elapsed
```

Experimented with calculating Θ using `lapply` or in a loop to speed up computation. Calculating Θ in the loop was about twice as fast as using `lapply` (about 5 minutes compared to 2.5). Additionally, the outputs were not identical, presumably due to some loss of data due to memory constraints using `lapply`. The differences were on the order of $1E-10$ or smaller, but when performing several iterations this can be significant. So the Θ calculated from the `for` loop is used for calculations for the rest of the assignment. Also, for time saving purposes, once Θ was computed it was saved as an R data set and is loaded in a muted chunk instead of running the below chunk every knit.

```r
Theta_lapply <- matrix(0, nrow = ncol(X_center), ncol = ncol(X_center))
tic("Time To Calculate Theta with lapply")
start <- 1
index <- 1000
for (j in 1:60) {
  tic(start)
  Theta_list <-
    lapply(c(start:index), function(i) {
      X_center[i, ] %*% t(X_center[i, ])
    })

  Theta_lapply <- Reduce('+', Theta_list) + Theta_lapply
  rm(Theta_list)
  start <- start + 1000
  index <- index + 1000
  toc()
}
toc()

tic("Time to calculate Theta using a for loop")
Theta <- matrix(0, nrow = ncol(X_center), ncol = ncol(X_center))
for (i in 1:nrow(X_center)) {
  Theta <- Theta + X_center[i, ] %*% t(X_center[i, ])
}
toc()
```

Power Iteration Function:

- create random initial column vectors, and center
- Calculate initial λ's off of U
- Perform orthogonalized power iteration until either the difference in `norm`'s is less than ϵ or max iterations are reached

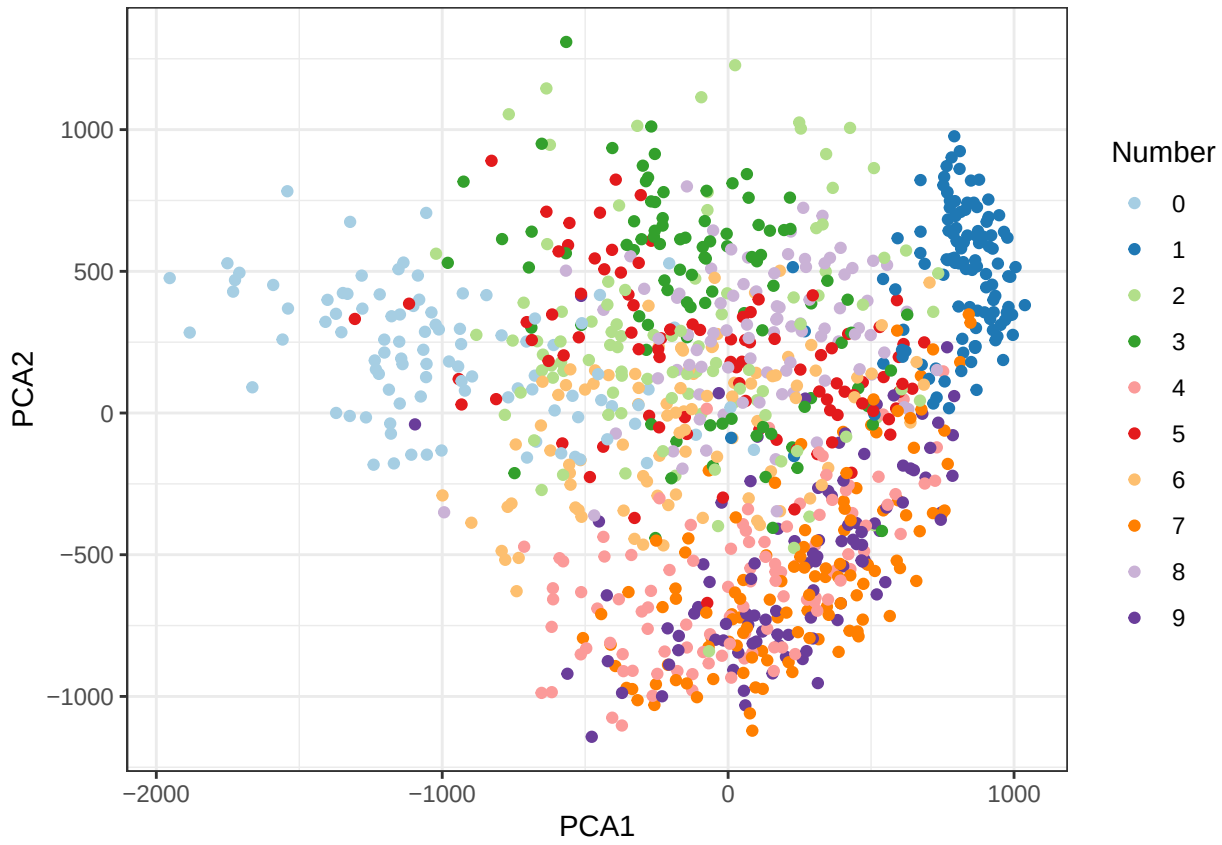
```
tic("Power Iteration")
power_iteration <-
  function(A,
           n,
           max_iterations = 100000,
           epsilon = 10 ^ -5) {
    set.seed(123)
    # Create initial random vectors with columns = n to return n eigenvalues
    U <- matrix(runif(nrow(A) * n), ncol = n)

    ### Normalize columns of U
    U <- apply(
      U,
      2,
      FUN = function(x) {
        x / norm(x)
      }
    )

    lambda <- apply(U, 2, function(X) {t(X) %*% A %*% X})

    for (i in 1:max_iterations) {

      tilde_U <- A %*% U
      qr_out <- qr(tilde_U)
      U <- qr.Q(qr_out)

      lambda_2 <- apply(U, 2, function(X) {t(X) %*% A %*% X})

      if (all(abs(lambda - lambda_2) <= c(rep(epsilon, n)))) {
        lambda <- lambda_2
        break
      }

      lambda <- lambda_2
    }
    eigen_data <- list(
      Eigenvectors = U,
      Eigenvalues = lambda,
      iterations = i)

    return(eigen_data)

  }

eigens_Theta <- power_iteration(A = Theta, n = 2)
eigen_values <- eigen(Theta)$values[c(1, 2)]
toc()
```

```
## Power Iteration: 0.898 sec elapsed
```

Using orthogonal power iteration we get the first two eigenvalues of Θ to be $\lambda_1 = 332719.1$ and $\lambda_2 = 243279.9$. This compares to values of $\lambda_1 = 332719.1$ and $\lambda_2 = 243279.9$ using R's built in `eigen()` function. It took the `power_iteration` function 113 iterations to converge for two eigenvalues.

Apply 2-dimensional PCA using the eigenvectors computed with the orthogonalized power iteration above:

```
PCAs <- X_center %*% eigens_Theta$Eigenvectors

plot_data <- data.frame(Number = Numbers, PCA1 = PCAs[,1], PCA2 = PCAs[,2]) %>%
  rename(Number = V1) %>%
  mutate(Number = as.factor(Number))

plot_sample <- plot_data[1:1000,]

ggplot(data = plot_sample, aes(x = PCA1, y = PCA2, color = Number)) +
  geom_point() +
  scale_color_brewer(palette = "Paired") +
  theme_bw()
```




```
rm(plot_data)
rm(plot_sample)
```

b

We write a robust function for generating k dimension reduction on any dataset, using functions written above. If not already calculated, it will comput Θ. Or user can input already calculated Θ for better efficiency. Gives user option to compute eigendata using either poweriteration or R's built in `eigen()` function.

- Center data, if necessary
- Calculate Θ, if necessary
- Compute eigendata
- Reduce dimensionality of data and print variance captured

```
tic("reduce_dimensions")
reduce_dimensions <- function(k, data,
                              power_iter = F, theta = NULL,
                              centered = F, print_var = T) {
  data <- as.matrix(data)

    ### First: Center Data if necessary
  if(!centered){

  mu <- colMeans(data)

  data <- t(apply(data, 1, function(x) {
    return(x - mu)
  }))
  }

  ## Compute Theta if necessary:
  if (is.null(theta)) {
    Theta <- matrix(0, nrow = ncol(X_center), ncol = ncol(X_center))

    for (i in 1:nrow(X_center)) {
      Theta <- Theta + X_center[i, ] %*% t(X_center[i, ])
    }
  }

  ### Get eigendata and compute Principle Components:
  if(power_iter){

  eigens_Theta <- power_iteration(A = theta, n = nrow(theta))

  PCAs <- data %*% eigens_Theta$Eigenvectors[,1:k]
  }else{
    eigens_Theta <- eigen(theta)

    PCAs <- data %*% eigens_Theta$vectors[,1:k]
  }
  if(print_var){
  cat(
    "The total variance captured in the first ",
    k,
    " dimensions is: ",
    round(100 * sum(eigens_Theta$values[1:k]) / sum(eigens_Theta$values), 2),
    "%\n"
```

```
  )
  }

  return(PCAs)

}

k_vec <- c(1,10,25,50,100,250)

PCAs_list <- lapply(k_vec, reduce_dimensions, data = X, theta = Theta)
```

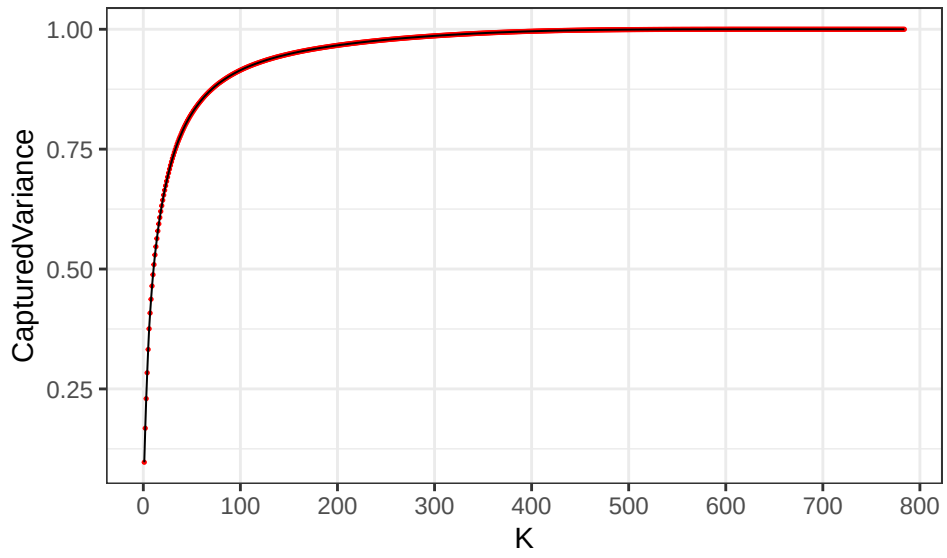
```
## The total variance captured in the first  1  dimensions is:  9.7 %
## The total variance captured in the first  10  dimensions is:  48.81 %
## The total variance captured in the first  25  dimensions is:  69.18 %
## The total variance captured in the first  50  dimensions is:  82.46 %
## The total variance captured in the first  100  dimensions is:  91.46 %
## The total variance captured in the first  250  dimensions is:  97.81 %
```

```
toc()
```

```
## reduce_dimensions: 42.238 sec elapsed
```

Graph the cumulative variance captured with each additional dimension:

```
k_data <- data.frame(K = c(1:ncol(X)), lambda = eigen(Theta)$values)
k_data <- k_data %>%
  mutate(CapturedVariance = cumsum(lambda) / sum(k_data$lambda))

ggplot(k_data, aes(x = K, y = CapturedVariance)) +
  geom_point(size = 0.25, color = "red") +
  geom_line(color = "black", size = 0.35) +
  theme_bw() +
  scale_x_continuous(breaks = seq(0,800,100), minor_breaks = NULL)
```



```
rm(k_data)
```

```
show_image <- function(m, oriented=T)
{
  im <- matrix(m, byrow=T, nrow=28)

  if (oriented) {
    im_orient <- matrix(0, nrow=28, ncol=28)
    for (i in 1:28)
      im_orient[i,] <- rev(im[,i])

    im <- im_orient
  }
  image(im)
}

projected_image <- function(plot_row, k) {

  projected.data <- X_center %*% (eig_vectors[,1:k])
  C <- projected.data[plot_row, 1:k]
  xp <- rep(0, 784)

  for (i in 1:k) {
    xp <- xp + C[i] * eig_vectors[, i]
  }
  cat("K = ",k, "\n")

  return(list(vK = k, picture_data = xp))
}
```

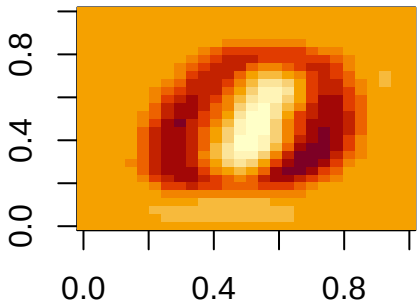
First let's show the projected images for our initial K dimensions, and the last image is the plot of the initial dataset

```
tic("Graph a number 3 over k dimensions")
eig_vectors <- eigen(Theta)$vectors
paste0(k_vec)
```

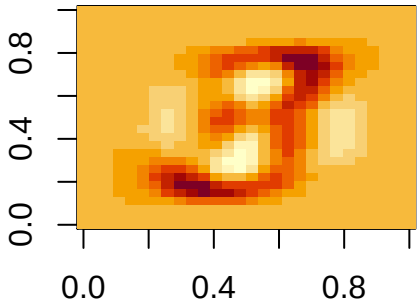
```
## [1] "1"   "10"  "25"  "50"  "100" "250"
```

```
cl <- makeCluster(8)
clusterExport(cl = cl, list("X_center", "eig_vectors"))
test <- parallel::parLapply(cl, k_vec, projected_image, plot_row = 8)

for (i in 1:length(k_vec)) {
  cat("k = ", test[[i]]$vK)
  show_image(test[[i]]$picture_data)
}
```

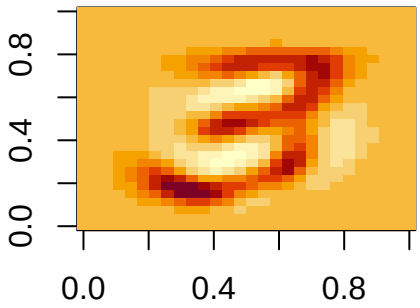
```
## k =  1
```



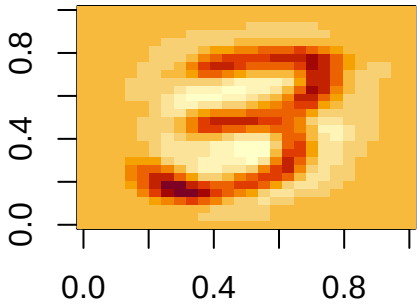

```
## k =  10
```



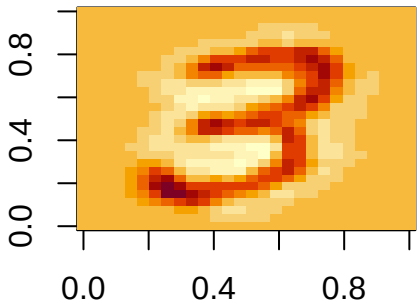

```
## k =  25
```



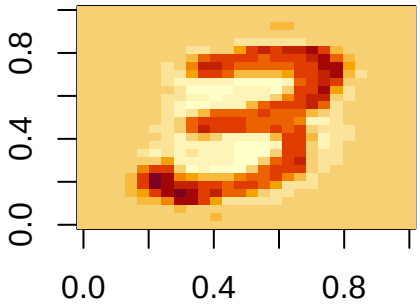

```
## k =  50
```



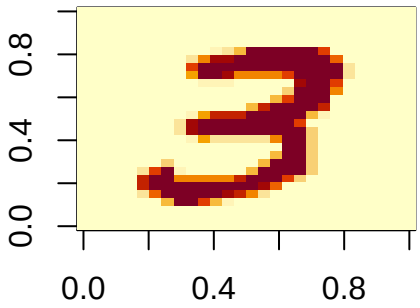

```
## k =  100
```




```
## k =  250
```




```
show_image(X[8,])
```



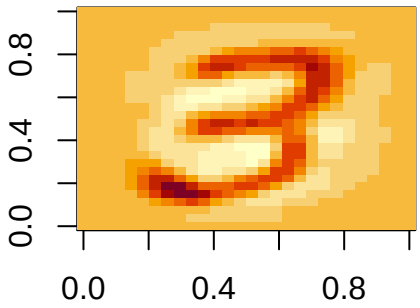

```
toc()
```

```
## Graph a number 3 over k dimensions: 23.005 sec elapsed
```

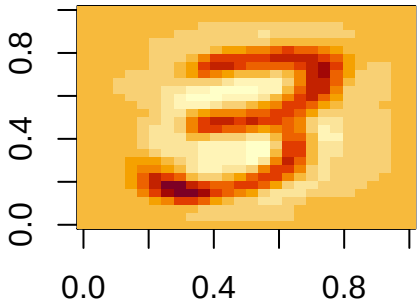
The image starts to come into focus around $k = 50$, and is much clearer at $k = 100$. The added clarity between $k = 100$ and $k = 250$ does not seem significant. Now let's run the projection on k values between 50 and 100, in intervals of 5.

```
tic("Images from k=55 to k = 100")
k_vec <- seq(55,100,5)
test2 <- parallel::parLapply(cl, k_vec, projected_image, plot_row = 8)
for (i in 1:length(k_vec)) {
  cat("k = ", test2[[i]]$vK)
  show_image(test2[[i]]$picture_data)
}
```

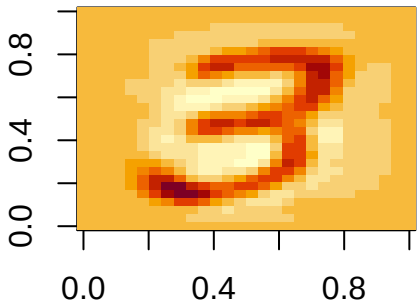
```
## k =  55
```



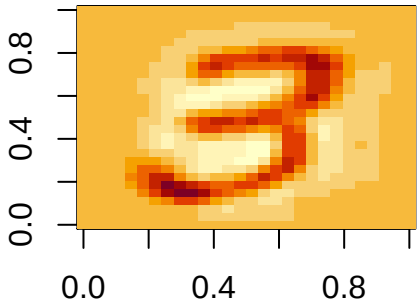

```
## k =  60
```



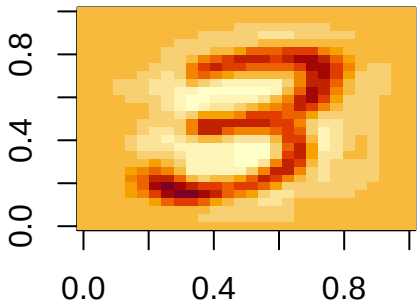

```
## k =  65
```



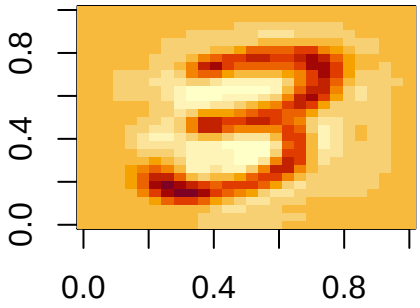

```
## k =  70
```



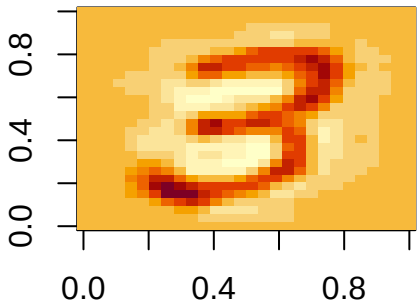

```
## k =  75
```



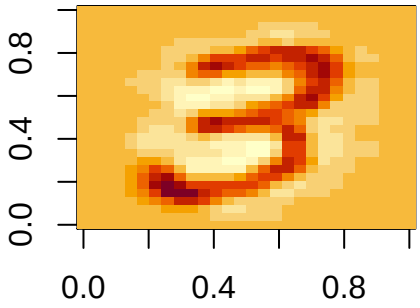

```
## k =  80
```



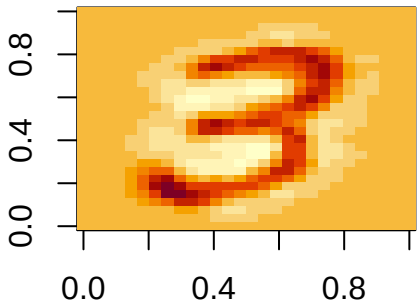

```
## k =  85
```



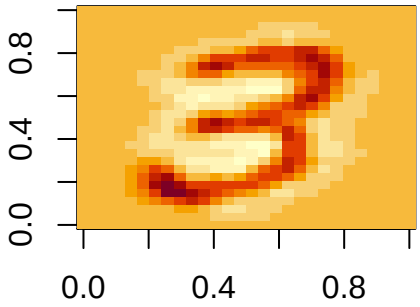

```
## k =  90
```




```
## k =  95
```




```
## k =  100
```




```
toc()
```

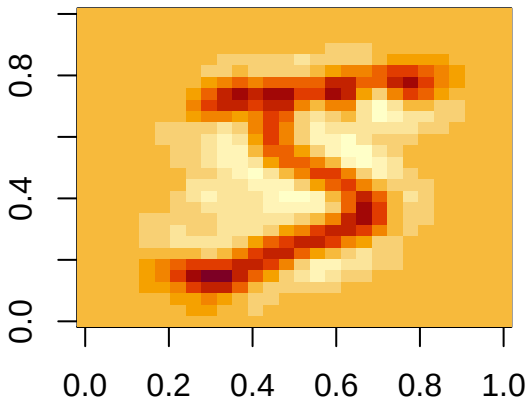
```
## Images from k=55 to k = 100: 13.001 sec elapsed
```

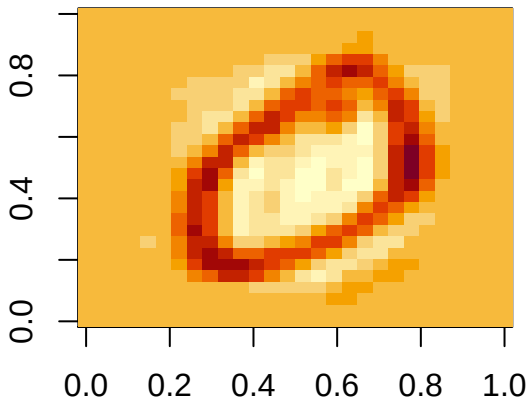
Overall $k = 75$ appears to do a good job of capturing the image, and dimensions in excess of 75 do not seem to add much added value.

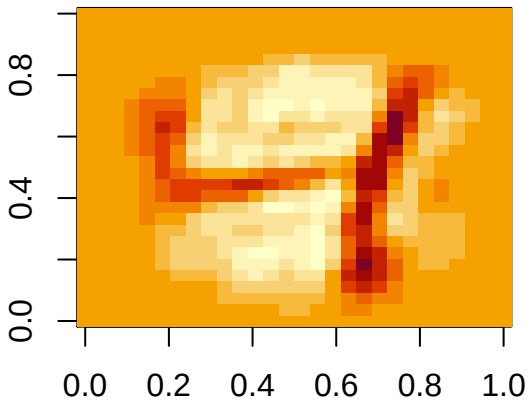
Let's just test this k on a few other numbers:

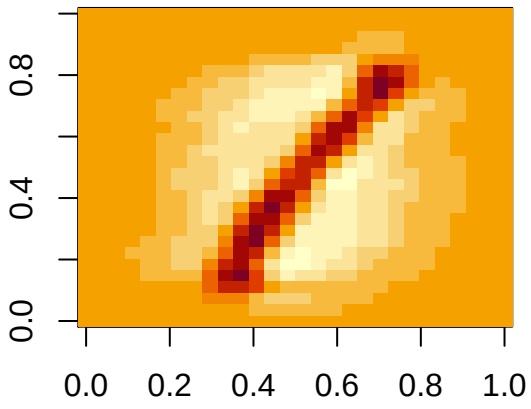
```
tic("Other Numbers")
new_number_rows <- match(unique(Numbers$V1), Numbers$V1)
output <- parLapply(cl, new_number_rows, projected_image, k = 75)
stopCluster(cl)
```

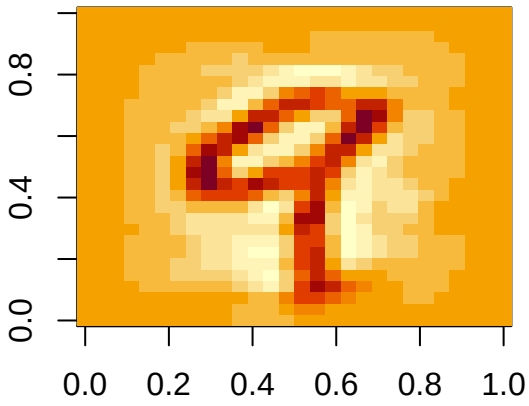
```
for (i in 1:length(new_number_rows)) {
  show_image(output[[i]]$picture_data)
}
```

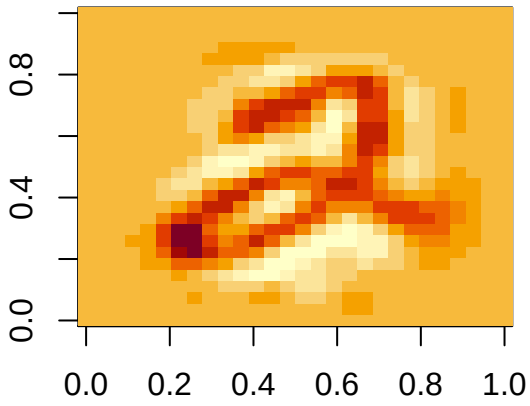


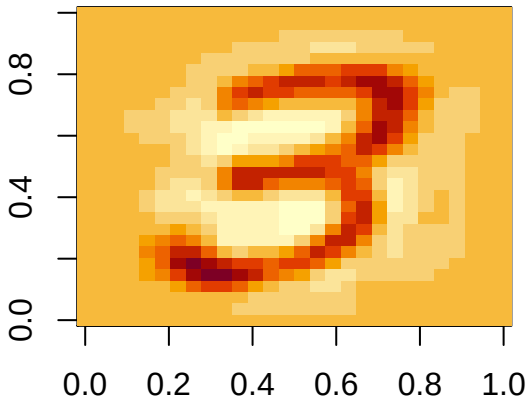


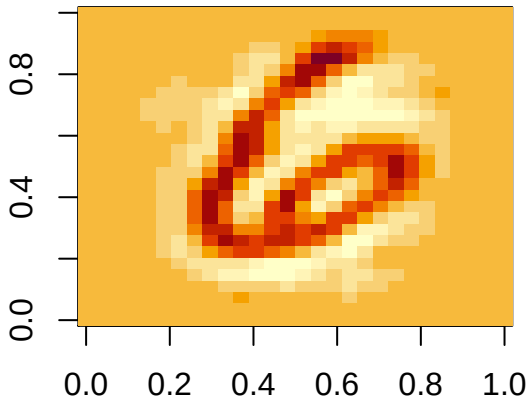


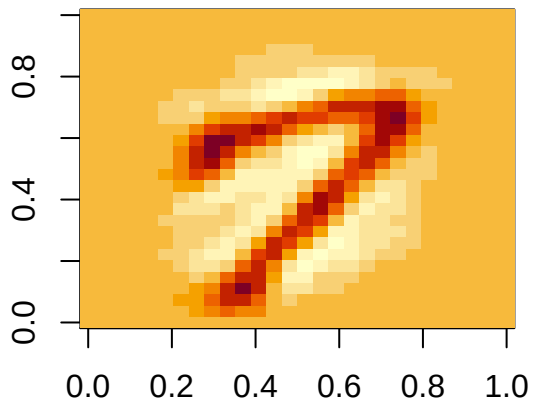


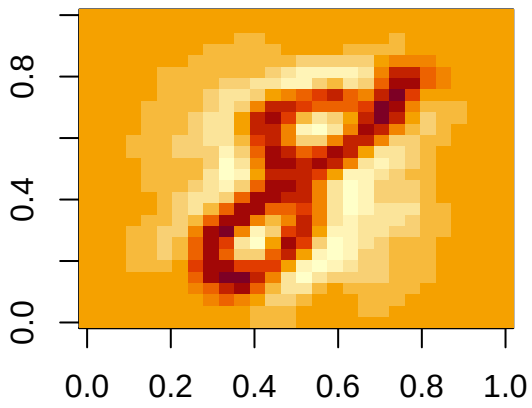

0.0
0.4
0.8
0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.4
0.8
0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.4
0.8
0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.4
0.8
0.0
0.2
0.4
0.6
0.8
1.0






```
toc()
```

```
## Other Numbers: 11.819 sec elapsed
```

And we see that $k = 75$ does a good job of capturing the image for a sample of every number from 0-9.

C

First, write in gradient of L and logL functions from HW7. Gradient has been slightly changed so it only computes for a single vector instead of a matrix. Also, we will use the mini-batch method described in the video, selecting a small sample of observations to sum their gradient. To enhance computation time, we will use the `parallel` package so that we can compute multiple gradients simultaneously. From above, we will let $k = 75$. Using the `detectCores()` function below, we see that there are 12 cores available for parallel computing. So we will set our mini-batch size at 8 for this exercise, and at each step compute one gradient on eight cores. This will be done on data reduced from 784 dimensions down to 75.

```
gradL <- function(alpha, X)
{
  y <- X[1]
  x <- X[-1]

  e_term <- exp(-x %*% alpha)

  grad <- x * e_term / (1 + e_term) - (1 - y) * x

  return (grad)
}

logL <- function(alpha, X, y)
{
```

```
  sum(-(1-y)*(X %*% alpha) - log(1 + exp(-X %*% alpha)))
}

detectCores()
```

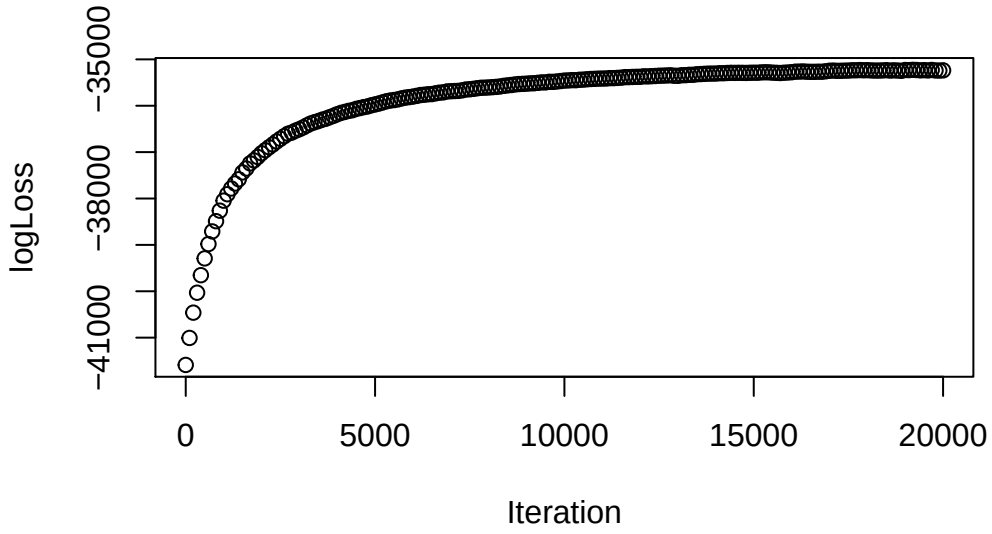
```
## [1] 12
```

```
set.seed(123)
k <- 75

reduced_data <- reduce_dimensions(k = 75, data = X, theta = Theta, print_var = F)

is_three <- as.numeric(Numbers$V1 == 3)

y <- is_three
X <- reduced_data / 250
alpha <- rep(0,ncol(X))

StochasticGradientDescent <- function(alpha, X, y,
                                      max_iter = 1E4,
                                      mini_batch = 8,
                                      with_replacement = FALSE){

  cl <- makeCluster(min(mini_batch, detectCores() - 1))

  initial_sample_rows <- c(1:nrow(X))
  current_sample_rows <- initial_sample_rows
  L <- logL(alpha, X, y)
  for (i in 1:max_iter) {

    step <- 0.001 / mini_batch
    sample_rows <- sample(current_sample_rows, mini_batch)

    if (!with_replacement) {
      ### Remove rows we just sampled
      current_sample_rows <- current_sample_rows[-sample_rows]
      if (length(current_sample_rows) < mini_batch) {
        ### If we have gone through all rows, refill our sample space
        current_sample_rows <- initial_sample_rows
      }
    }

    stochasticSample <- cbind(y[sample_rows], X[sample_rows,])

    gradients <- rowSums(
      parallel::parApply(cl = cl, stochasticSample, MARGIN = 1, FUN = gradL, alpha = alpha)
    )

    alpha <- alpha + step * gradients
    if(i %% 100 == 0){L <- c(L, logL(alpha, X, y))}


  }
```

```
  stopCluster(cl)

  output <- list(LossHistory = L, Alpha = alpha)
  return(output)

}

start_time <- Sys.time()
tictoc::tic("Start Stochastic Gradient Descent ")
testRun <- StochasticGradientDescent(alpha, X, y, max_iter = 2E4)
tictoc::toc()
```

```
## Start Stochastic Gradient Descent : 49.577 sec elapsed
```

```
end_time <- Sys.time()


plot(x = 100 * (c(1:length(testRun$LossHistory) - 1)), y = testRun$LossHistory,
     xlab = "Iteration", ylab = "logLoss")
```




```
rm(X)
rm(X_center)
rm(Numbers)
```

C

```
k <- 75
mnist_test <- data.table::fread("mnist_test.csv")
```

```
Y_test <- as.numeric(mnist_test[,1] == 3)

X_test <- as.matrix(mnist_test[,-1])

X_test <- reduce_dimensions(k = 75, data = X_test, theta = Theta, print_var = F) / 250

alpha_test <- testRun[[2]]

accuracy <- function(p, alpha_star, X, y)
{
  probs <- 1/(1 + exp(-X %*% alpha_star))
  pred_y <- as.numeric(probs >= p)

  sens <- sum(pred_y==1 & y==1)/sum(y==1)
  spec <- sum(pred_y==0 & y==0)/sum(y==0)
  overall <- (sum(pred_y==1 & y==1) + sum(pred_y==0 & y==0))/length(y)

  return (list(sensitivity=sens, specificity=spec, overall=overall))
}

p <- seq(0.05, .95, .05)
specs <- rep(0, length(p))
sens <- rep(0, length(p))
overall <- rep(0, length(p))

for (i in 1:length(p)) {
  acc_out <- accuracy(p[i], alpha_test, X_test, Y_test)
  specs[i] <- acc_out$specificity
  sens[i] <- acc_out$sensitivity
  overall[i] <- acc_out$overall
}

df <- data.frame(p=p,
                 sensitivity=sens,
                 specificiy=specs,
                 overall=paste0(round(100*overall,2),"%"))


print(xtable::xtable(df), comment = F)
```

The time needed to compute the α for the logistic regression on 20,000 iterations with a random mini-batch of 8 was 49.579 seconds, much faster than the Newton's descent method from HW7. Additionally, this proved to be even more accurate when tested on the mtest data.set, achieving 95.8% accuracy at a probabilty threshold of 0.85.

```
toc()
```

```
## Total Run Time: 157.416 sec elapsed
```

	p	sensitivity	specificiy	overall
1	0.05	1.00	0.00	10.12%
2	0.10	1.00	0.00	10.52%
3	0.15	1.00	0.02	12.1%
4	0.20	1.00	0.06	15.69%
5	0.25	1.00	0.12	20.96%
6	0.30	1.00	0.20	28.31%
7	0.35	1.00	0.31	37.65%
8	0.40	0.99	0.44	49.26%
9	0.45	0.99	0.57	60.98%
10	0.50	0.98	0.68	71.13%
11	0.55	0.97	0.77	78.67%
12	0.60	0.96	0.83	84.34%
13	0.65	0.93	0.88	88.47%
14	0.70	0.91	0.92	92.1%
15	0.75	0.87	0.95	94.53%
16	0.80	0.80	0.97	95.64%
17	0.85	0.69	0.99	95.8%
18	0.90	0.56	1.00	95.31%
19	0.95	0.32	1.00	93.11%