Appendix: Code

Christian Pascual, Xinyi Lin, Junting Ren

3/7/2019

```
library(tidyverse)
library(glmnet)
library(modelr)
library(matrixcalc)
set.seed(8160)
```

Data preparation

```
standardize = function(col) {
  mean = mean(col)
  stdev = sd(col)
  return((col - mean)/stdev)
}

# just standardize the covariates
raw = read.csv(file = "breast-cancer-1.csv")
resp = raw %>% dplyr::select(diagnosis) %>% mutate(diagnosis = ifelse(diagnosis == "M", 1, 0))
standardized.data = raw %>%
  dplyr::select(radius_mean:fractal_dimension_worst) %>%
  map_df(.x = ., standardize)

data = cbind(resp, standardized.data)
X = data %>% select(-diagnosis)

y = as.matrix(data$diagnosis)
```

Removing multicollinearity

```
# Many of the columns in the dataset are highly correlated with each other
# Removing any columns that have correlation greater than 0.7
X = data %>% select(-diagnosis, -perimeter_mean, -area_mean, -concave.points_mean,
                    -radius_worst, -area_se, -perimeter_worst, -area_worst,
                    -concave.points_worst, -texture_worst, -smoothness_worst,
                    -compactness_se, -compactness_mean, -compactness_worst,
                    -concavity_worst, -fractal_dimension_worst, -perimeter_se,
                    -concave.points_se, -fractal_dimension_se, -symmetry_worst)
cor.matrix = cor(X)
# Add back in intercept for
X = X %>% mutate(intercept = 1) %>% dplyr::select(intercept, everything())
```

Functions needed for analysis

```r
soft = function(beta, gamma) {
  ### Parameters:
  # beta : the original coefficient beta from a regression
  # gamma : the desired threshold to limit the betas at

  # returns a single adjusted value of the original beta

  return(sign(beta) * max(abs(beta) - gamma, 0))
}

calc.cur.p = function(data, betas) {
  ### Parameters:
  # intercept : the intercept term of the betas (scalar)
  # data : the associated data for each beta in betas (n x p matrix)
  # betas : all the non-intercept beta coefficients (p x 1 array)

  # return n x 1 array of current probabilities evaluated with given betas

  u = data %*% betas
  return(exp(u) / (1 + exp(u)))
}

calc.working.weights = function(p) {
  ### Parameters:
  # p : the working probabilities, calculated by calc.cur.p

  # return n x 1 array of working weights for the data

  # Check for coefficient divergence, adjust for fitted probabilities 0 & 1
  close.to.1  = (1 - p) < 1e-5
  close.to.0 = p < 1e-5
  w = p * (1 - p)
  w[which(close.to.1)] = 1e-5
  w[which(close.to.0)] = 1e-5

  return(w)
}

calc.working.resp = function(data, resp, betas) {
  ### Parameters:
  # intercept : the intercept term of the betas (scalar)
  # data : the associated data for each beta in betas (n x p matrix)
  # resp : the response variable of the dataset (n x 1 array)
  # betas : all the non-intercept beta coefficients (p x 1 array)
  # p : the working probabilities, calculated by calc.cur.p

  # return n x 1 array of working responses evaluated with given betas
  p = calc.cur.p(data, betas)
  w = calc.working.weights(p)
  return((data %*% betas) + ((resp - p) / w))
}
```

```r
calc.obj = function(betas, w, z, data, lambda) {
  ### Parameters:
  # intercept : the intercept term of the betas (scalar)
  # data : the associated data for each beta in betas (n x p matrix)
  # resp : the response variable of the dataset (n x 1 array)
  # betas : all the non-intercept beta coefficients (p x 1 array)

  # return the log-likelihood value for a logistic model
  LS = (2 * nrow(data))^(-1) * sum(w * (z - (data %*% betas))^2)
  beta.penalty = lambda * sum(abs(betas))
  return(LS + beta.penalty)
}

compile = function(data, resp, betas) {
  # Helper function to contain all the calculations for coordinate logistic regression
  p = calc.cur.p(data, betas)
  w = calc.working.weights(p)
  z = calc.working.resp(data, resp, betas)
  return(list(
    p = p,
    w = w,
    z = z
  ))
}

calc.beta.norm = function(beta1, beta2) {
  ### Parameters:
  # beta1, beta2 : beta vectors to compare

  # returns the Frobenius norm between two beta vectors
  return(norm(as.matrix(beta1 - beta2), "F"))
}
```

Newton-Raphson Implementation

```r
logisticstuff <- function(x, y, betavec) {
  u <- x %*% betavec
  expu <- exp(u)
  loglik = vector(mode = "numeric", nrow(x))
  for(i in 1:nrow(x))
    loglik[i] = y[i]*u[i] - log(1 + expu[i])
  loglik_value = sum(loglik)
  # Log-likelihood at betavec
  p <- expu / (1 + expu)
  # P(Y_i=1|x_i)
  grad = vector(mode = "numeric", length(betavec))
  #grad[1] = sum(y - p)
  for(i in 1:length(betavec))
    grad[i] = sum(t(x[,i])%*%(y - p))
  #Hess <- -t(x)%*%p%*%t(1-p)%*%x
  Hess = hess_cal(x, p)
```

```
  return(list(loglik = loglik_value, grad = grad, Hess = Hess))
}

hess_cal = function(x, p) {
  len = length(p)
  hess = matrix(0, ncol(x), ncol(x))
  for (i in 1:len) {
    unit = x[i,] %*% t(x[i,]) * p[i] *(1 - p[i])
    #unit = t(x[i,])%*%x[i,]*p[i]*(1-p[i])
    hess = hess + unit
  }
  return(-hess)
}

NewtonRaphson <- function(x, y, logisticstuff, start, tol = 1e-5, maxiter = 200) {
  i <- 0
  cur <- start
  stuff <- logisticstuff(x, y, cur)
  res = c(0, cur)
  #res <- c(0, stuff$loglik, cur)
  prevloglik <- -Inf      # To make sure it iterates
  #while(i < maxiter && abs(stuff$loglik - prevloglik) > tol && stuff$loglik > -Inf)
  while (i < maxiter && abs(stuff$loglik - prevloglik) > tol) {
    i <- i + 1
    prevloglik <- stuff$loglik
    print(prevloglik)
    prev <- cur
    cur <- prev - solve(stuff$Hess) %*% stuff$grad
    stuff <- logisticstuff(x, y, cur)        # log-lik, gradient, Hessian
    res = rbind(res, c(i, cur))
    #res <- rbind(res, c(i, stuff$loglik, cur))
    # Add current values to results matrix
    }
  return(res)
}
modified <- function(x, y, logisticstuff, start, tol = 1e-5, maxiter = 200){
  i <- 0
  cur <- start
  beta_len <- length(start)
  stuff <- logisticstuff(x, y, cur)
  res = c(0, cur)
  #res <- c(0, stuff$loglik,cur)
  prevloglik <- -Inf # To make sure it iterates
  while(i <= maxiter && abs(stuff$loglik - prevloglik) > tol)
  #while(i <= maxiter && abs(stuff$loglik - prevloglik) > tol && stuff$loglik > -Inf)
    { i <- i + 1
    prevloglik <- stuff$loglik
    prev <- cur
    lambda = 0
    while (is.negative.definite(stuff$Hess-lambda*diag(beta_len)) == FALSE) {
      lambda = lambda + 1
    }
    print(i)
```

```
    cur <- prev - solve(stuff$Hess-lambda*diag(beta_len)) %*% stuff$grad
    #cur <- prev + (diag(beta_len)/10)%*%(stuff$grad)
    #cur = prev + t(stuff$grad)%*%(stuff$grad)
    stuff <- logisticstuff(x, y, cur) # log-lik, gradient, Hessian
    res = rbind(res, c(i, cur))
    #res <- rbind(res, c(i, stuff$loglik, cur))
    }
  return(res)
  }
```

Logistic LASSO implementation

```
LogLASSO.CD = function(X, y, beta, lambda, tol = 1e-10, maxiter = 1000) {
  ### Parameters:
  # X : design matrix, does include an intercept column
  # y : response variable (should be binary)
  # beta : starting beta coefficients to start from
  # lambda : constraining parameter for LASSO penalization
  # tol : how precise should our convergence be
  # maxiter : how many iterations should be performed before stopping

  # return a list containing the matrix of the coefficients and the iteration matrix

  # Convert data into matrix for easier manipulation
  X = as.matrix(X)
  names(beta) = colnames(X) # Assign original covariate names to the betas

  # Iteration setup
  j = 0
  work = compile(X, y, beta)
  obj = calc.obj(beta, work$w, work$z, X, lambda)
  diff.obj = diff.beta = Inf
  path = c(iter = j, obj = obj, beta)
  beta[1] = sum(work$w * (work$z - X %*% beta)) / sum(work$w)
  while (j < maxiter && diff.obj > tol) {

    j = j + 1
    prev.obj = obj
    prev.beta = beta

    # Coordinate descent through all of the betas
    for (k in 2:length(beta)) {
      work = compile(X, y, beta)
      z.rm.k = X[,-k] %*% beta[-k]
      val = sum(work$w * X[,k] * (work$z - z.rm.k))
      beta[k] = (1/sum(work$w * X[,k]^2)) * soft(val, lambda)
    }

    # Recalculate the objective
    work = compile(X, y, beta)
    obj = calc.obj(beta, work$w, work$z, X, lambda)
```

```
    # Convergence check calculation
    # diff.obj = abs(prev.obj - obj) # check difference in log-likelihood
    diff.obj = norm(as.matrix(beta - prev.beta), "F")
    prev.obj = obj

    # Append it to tracking matrix
    path = rbind(path, c(iter = j, obj = obj, beta))
    }

  return(list(
    path = as.tibble(path),
    coefficients = beta,
    iter = j,
    obj = obj)
    )
  }
```

Visualizations

Generating the path of solutions with different lambdas

```
# Function for generating the path of solutions with different lambdas
create.sol.path = function(lambdas, start, data, resp) {
  ### Parameters:
  # lambdas : the sequence of lambdas that you want to create solutions for
  # start : the starting beta coefficients
  # data : the data to estimate the coefficients from
  # resp : the response variable you're trying to predict

  # returns an matrix of the lambda and the computed beta coefficients for that lambda

  coeff.path = NULL
  coeffs = start
  for (l in 1:length(lambdas)) {
    fit = LogLASSO.CD(data, resp, coeffs, lambdas[l])
    iter = fit$iter
    obj = fit$obj
    coeff.path = rbind(coeff.path, c(lambda = lambdas[l], iter = iter, obj = obj, fit$coefficients))
    print(paste("Iter", l, "done,", iter, "loops needed for convergence",  sep = " ")) # progress bar
  }
  return(coeff.path)
}

# Create the solution path for the data
lambda.seq = exp(seq(-3, 6, length = 300))
coeffs = rep(0.001, 12)

path = create.sol.path(lambda.seq, coeffs, X, y)
tidy.path = as.tibble(path) %>%
  gather(., key = "coeff", value = "coeff_est", intercept:symmetry_se) %>%
  mutate(log.lambda = log(lambda))
```

```
ggplot(data = tidy.path, aes(x = log.lambda, y = coeff_est, color = coeff, group = coeff)) +
  geom_line(alpha = 0.5) +
  labs(
    title = "Log-LASSO Coefficient estimates as a function of log(lambda)",
    x = "log(lambda)",
    y = "Coefficient estimate"
  ) +
  theme(legend.position = "bottom", plot.title = element_text(hjust = 0.5))
```

Cross-validation to find the best lambda

```
# Function to do the cross-validation

lambda.cv = function(lambdas, start, data, resp, k = 5) {
  ### Parameters:
  # lambdas : the sequence of lambdas that you want to create solutions for
  # start : the starting beta coefficients
  # data : the data to estimate the coefficients from
  # resp : the response variable you're trying to predict

  # returns a matrix of average test MSES against a sequence of given lambdas

  folds = crossv_kfold(data, k = k)
  path = NULL
  i = 0

  for (l in lambdas) {
    # Reset the storage of the fold MSEs
    fold.mses = NULL
    fold.ses = NULL
    i = i + 1
    for (k in 1:nrow(folds)) {

      # Grab the specific training indexes
      train.idx = folds[k,1][[1]][[toString(k)]]$idx
      test.idx = folds[k,2][[1]][[toString(k)]]$idx

      # Split up the data into the training and test datasets
      train.X = data[train.idx,]
      test.X = data[test.idx,]
      train.y = resp[train.idx]
      test.y = resp[test.idx]

      # Perform the logistic-LASSO
      fit = LogLASSO.CD(X = train.X, y = train.y, beta = start, lambda = l)
      u = exp(as.matrix(test.X) %*% fit$coefficients)
      z = u / (1 + u)

      # Calculate the test MSE for the fold
      fold.mse = mean((test.y - z)^2)
      fold.mses = c(fold.mses, fold.mse)
```

```
    }
    fold.se = sqrt(var(fold.mses)/5)

    path = rbind(path,
                 c(lambda = l, log.lambda = log(l),
                   avg.fold.mse = mean(fold.mses), avg.fold.se = fold.se))
    print(paste("Iteration:", i, "done"))
  }
  return(as.tibble(path))
}

cv.path = lambda.cv(lambda.seq, coeffs, X, y)
min.mse = min(cv.path$avg.fold.mse)
min.lambda = cv.path[which(cv.path$avg.fold.mse == min.mse),]$log.lambda

cv.path %>%
  ggplot(data = ., aes(x = log.lambda, y = avg.fold.mse)) +
  geom_vline(xintercept = min.lambda) +
  geom_line(color = "red") +
  geom_errorbar(aes(ymin = avg.fold.mse - avg.fold.se, ymax = avg.fold.mse + avg.fold.se), color = "gra
  labs(
    title = "Average Test Fold MSE as a function of lambda",
    x = "log(lambda)",
    y = "Average Test MSE"
  ) +
  theme(plot.title = element_text(hjust = 0.5))
```

Tabulation

Assemble all models

```
# Fit all three models
NR.fit = modified(as.matrix(X), y, logisticstuff, coeffs)
LL.fit = LogLASSO.CD(X, y, coeffs, exp(min.lambda))
```

```
coeff.table = tibble(
  `Coefficient` = c("Intercept", "Mean radius", "Mean texture", "Mean smoothness",
                    "Mean concavity", "Mean symmetry", "Mean fractal dimension",
                    "Standard error radius", "Standard error texture",
                    "Standard error smoothness", "Standard error concavity",
                    "Standard error symmetry"),
  `Newton-Raphson` = NR.fit[nrow(NR.fit), 2:ncol(NR.fit)],
  `Logistic-LASSO` = LL.fit$coefficients
)
knitr::kable(coeff.table)
```

Evaluating predictive ability

```
coeffs = rep(0.001, 12)
NR.coefs = NR.fit[nrow(NR.fit), 2:ncol(NR.fit)]
# Set up the datasets for cross-validation
folds = crossv_kfold(X, k = 10)

NR.mses = NULL
LL.mses = NULL
for (k in 1:nrow(folds)) {

  # Grab the specific training indexes
    train.idx = folds[k,1][[1]][[toString(k)]]$idx

    # Split up the data into the training and test datasets
    train.X = X[train.idx,]
    test.X = X[-train.idx,]
    train.y = y[train.idx]
    test.y = y[-train.idx]

  LogLASSO = LogLASSO.CD(X = train.X, y = train.y,
                         beta = coeffs, lambda = exp(min.lambda))
  LL.u = exp(as.matrix(test.X) %*% LogLASSO$coefficients)
  LL.z = LL.u / (1 + LL.u)

  NR.u = exp(as.matrix(test.X) %*% NR.coefs)
  NR.z = NR.u / (1 + NR.u)

  LL.mse = mean((test.y - LL.z)^2)
  LL.mses = cbind(LL.mses, LL.mse)

  NR.mse = mean((test.y - NR.z)^2)
  NR.mses = cbind(NR.mses, NR.mse)
  }
```

```
tidy.mses = tibble(
  `Logistic LASSO` = c(LL.mses),
  `Newton-Raphson` = c(NR.mses)
) %>%
  gather(., key = "model", value = "test.MSE", `Logistic LASSO`:`Newton-Raphson`)

tidy.mses %>%
  ggplot(data = ., aes(x = test.MSE, color = model, fill = model)) +
  geom_density(alpha = 0.5) +
  theme(legend.position = "bottom") +
  labs(
    title = "Distribution of test MSE in 10-fold cross validation by model",
    x = "Test MSE",
    y = "Density"
  ) +
  theme(plot.title = element_text(hjust = 0.5))
```