# MTH 511a - 2020: Lecture 21

Instructor: Dootika Vats

## 1 Resampling Methods

### 1.1 Cross-validation

In this lecture, we will cover some examples of using cross-validation.

#### 1.1.1 Regression model selection

**Ridge regression:**

Recall the regression model:

$$Y = X\beta + \epsilon\,,$$

where $\epsilon \sim N_n(0, \sigma^2 I_n)$. In this example subsection, we will focus our attention only on ridge regression estimators. Similar cross-validations can be done for bridge regression. Recall the ridge objective function for a given $\lambda$ is

$$Q_\lambda(\beta) = \frac{(y - X\beta)^T(y - X\beta)}{2} + \frac{\lambda}{2}\beta^T\beta\,.$$

Recall, the ridge estimator of $\beta$ is

$$\hat{\beta}_\lambda = (X^TX + \lambda I_n)^{-1}X^Ty\,.$$

Here, the solution for $\beta$ depends on the value of $\lambda$. We want to choose $\lambda$ so that prediction error is minimized. We choose the squared error loss function.

To choose the best model in this case, we set a vector of $\lambda : \lambda_1, \dots, \lambda_m$. For each $\lambda_i$, we will implement 10-fold cross-validation and estimate the prediction error. Whichever $\lambda$ minimzies the prediction error, will be the chosen $\lambda$.

Consider the `mtcars` dataset in `R`. The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (197374 models). The response is the miles per gallon of the 32 cars. Run a `?mtcars` in `R` to learn more about the dataset.

---

```r
###################################################
## Choosing lambda in ridge regression
## using LOOCV
## dataset is mtcars
###################################################
data(mtcars)
y <- mtcars$mpg
X <- cbind(1, as.matrix(mtcars[ ,-1]))

n <- dim(X)[1]
p <- dim(X)[2]
lam.vec <- c(10^(seq(-8, 8, by = .1)))
CV.error <- numeric(length = length(lam.vec))

for(l in 1:length(lam.vec))
{
  foo2 <- 0
  lam <- lam.vec[l]
  for(i in 1:n)
  {
    # Making training data
    X.train <- X[-i,] # removing ith X
    y.train <- y[-i] #removing ith y

    # fitting model for training data
    beta.train <- solve(t(X.train) %*% X.train + lam*diag(p)) %*% t(X.train)
        %*% y.train

    # test error
    foo2 <- foo2 + (y[i] - X[i,] %*% beta.train)^2
  }
  CV.error[l] <- foo2/n
}

# lambda that yields minimum error is chosen
(chosen.lam <- lam.vec[which.min(CV.error)] )
# [1] 5.011872

beta.final <- solve(t(X) %*% X + chosen.lam*diag(p)) %*% t(X) %*% y
beta.final
#             [,1]
#       0.247777096
# cyl   0.346710987
# disp -0.007378144
# hp   -0.008715992
# drat  1.420911900
# wt   -1.653648857
# qsec  0.938547096
# vs   -0.077511677
# am    1.443975953
# gear  1.530167865
# carb -0.745846809
```

**Bridge regression:**

For the same dataset, we now implement bridge regression with a grid of values for $\alpha$ and $\lambda$. We will use 4-fold cross validation.

```
##################################################
## Choosing lambda and alpha in bridge regression
## using 4-fold dataset is mtcars
##################################################
set.seed(12)
data(mtcars)
y <- mtcars$mpg
X <- cbind(1, as.matrix(mtcars[ ,-1]))

n <- dim(X)[1]
p <- dim(X)[2]

## MM algorithm for bridge regression
## for any alpha and lambda
bridge <- function(y, X, lambda, alpha, tol = 1e-8)
{
  current <- solve( t(X) %*%X + diag(lambda,p) ) %*% t(X) %*%y # start at
      ridge solution
  iter <- 0
  diff <- 100
  while( (diff > tol) && iter < 1000)
  {
    iter <- iter + 1
    # M matrix diagonals
    ms <- as.vector( lambda/2 *( abs(current))^(alpha - 2) )

    # MM update -- using qr.solve for numerical stability
    update <- qr.solve(t(X) %*% X + diag(ms, p)) %*% t(X) %*% y
    diff <- sum( (current - update)^2 )
    current <- update
  }
  return(current)
}


lam.vec <- 1:10
alpha <- seq(1,2, by =.05)

CV.error <- matrix(0, nrow = length(lam.vec), ncol = length(alpha))
permutation <- sample(1:n, replace = FALSE)
K <- 4

# Making a list of indices for each split
test.index <- split(permutation, rep(1:K, length = n, each = n/K))

for(l in 1:length(lam.vec))
{
  lam <- lam.vec[l]
  for(a in 1:length(alpha))
  {
    foo3 <- 0
    for(k in 1:K)
    {
      X.train <- X[-test.index[[k]], ]
      y.train <- y[-test.index[[k]]]
      X.test <- X[test.index[[k]], ]
      y.test <- y[test.index[[k]]]

      beta.train <- bridge(y.train, X.train, lambda = lam, alpha = alpha[a])

      foo3 <- foo3 + sum( (y.test - X.test %*% beta.train)^2)
    }
    CV.error[l,a] <- foo3/n
  }
}

# lambda that yields minimum error is chosen
ind <- which(CV.error == min(CV.error), arr.ind = TRUE)
(chosen.alpha <- alpha[ind[2]])
(chosen.lam <- lam.vec[ind[1]] )

# Final estimates
bridge(y, X, lambda = chosen.lam, alpha = chosen.alpha)
#              [,1]
#      0.0040899424
# cyl   0.2506111993
# disp  0.0038803710
# hp   -0.0145184786
# drat  1.3829350298
# wt   -2.7347035965
# qsec  1.0656202488
# vs   -0.0001061083
# am    1.8796458185
# gear  1.3880810512
# carb -0.4745591302
```

### 1.1.2 Choosing $C$ in GMM

We have discussed before that for choosing $C$, the usual loss functions don't work since there is not response and covariate. However, we know that models that yield the highest log-likelihood or the lowest negative log-likelihood are preferred. Thus, the loss function is:

$$L(x, \hat{\theta}) = -\log f(x|\hat{\theta}) \,.$$

We observe $X_1, \dots, X_n$ from a mixture of normals and split the data up into testing and training CV sets. We fit the EM algorithm to estimate the MLE from $c = 2, \dots, C$ classes over the training sets and estimate the prediction error.

Note that if use the negative log-likelihood on the full dataset to choose the number of classes $C$, we can keep reducing the loss by continually increasing $C$, until each data point is a cluster in itself. Thus the CV algorithm is critical here.

Below are functions that calculate the negative log-likelihood and that implement the EM algorithm for multivariate GMM.

```
###########################################
## EM algorithm to fit mixture of Gaussians
## to multivariate data (old faithful)
###########################################
set.seed(12)


# calculate negative log-likelihood
# of mixture of multivariate normal
log_like <- function(X, pi.list, mu.list, Sigma.list, C)
{
  foo <- 0
  for(c in 1:C)
  {
    foo <- foo + pi.list[[c]]*dmvnorm(X, mean = mu.list[[c]], sigma =
        Sigma.list[[c]])
  }
  return(-sum(log(foo)))
}

# X is the data
# C is the number of clusters
GLMMforC <- function(X, C, tol = 1e-3, maxit = 1e3)
{
  n <- dim(X)[1]
  p <- dim(X)[2]
  ######## Starting values ###################
  ## pi are equally split over C
  pi.list <- rep(1/C, C)

  mu <- list()
  Sigma <- list()

  # The means for each C cannot be the same,
  # since then the three distributions overlap
  # Hence adding random noise to colMeans(X)
  for(i in 1:C)
  {
    mu[[i]] <- rnorm(p, sd = 3) + colMeans(X)
    Sigma[[i]] <- var(X)
  }
  # Choosing good starting values is important since
  # The GMM likelihood is not concave, so the algorithm
  # may converge to a local optima.

  ######## EM algorithm steps ###################

  iter <- 0
  diff <- 100
  old.mu <- mu
  old.Sigma <- Sigma
  old.pi <- pi.list

  Ep <- matrix(0, nrow = n, ncol = C) # gamma_{i,c}
  save.loglike <- 0
  while((diff > tol) && (iter < maxit) )
  {
    iter <- iter + 1
    ## E step: find gammas
    for(c in 1:C)
    {
      Ep[ ,c] <- pi.list[c]*apply(X, 1, dmvnorm , mu[[c]], Sigma[[c]])
    }
    Ep <- Ep/rowSums(Ep)
```

```r
    ### M-step
    pi.list <- colMeans(Ep)
    for(c in 1:C)
    {
      mu[[c]] <- colSums(Ep[ ,c] * X )/sum(Ep[,c])
    }

    for(c in 1:C)
    {
      foo <- 0
      for(i in 1:n)
      {
        foo <- foo + (X[i, ] - mu[[c]]) %*% t(X[i, ] - mu[[c]]) * Ep[i,c]
      }
      Sigma[[c]] <- foo/sum(Ep[,c])

      # Below is to ensure the estimator is positive definite
      # otherwise next iteration gamma_i,c,k cannot be calculated
      Sigma[[c]] <- Sigma[[c]] + diag(1e-5, p)
    }

    save.loglike <- c(save.loglike, log_like(X = X, pi.list = pi.list,
       mu.list = mu, Sigma.list = Sigma, C = C))
    # Difference in the log-likelihoods as the difference criterion
    diff <- abs(save.loglike[iter+1] - save.loglike[iter])

    old.mu <- mu
    old.Sigma <- Sigma
    old.pi <- pi.list
  }

  # Final allocation updates
  for(c in 1:C)
  {
    Ep[ ,c] <- pi.list[c]*apply(X, 1, dmvnorm , mu[[c]], Sigma[[c]])
  }
  Ep <- Ep/rowSums(Ep)

  return(list("pi" = pi.list, "mu" = mu, "Sigma" = Sigma, "Ep" = Ep,
     "log.like" = tail(save.loglike,1)))
}
```

Next, we will do CV to choose the number of clusters in the `faithful` dataset. We choose between $2, 3, 4$ classes.

**NOTE:** Because EM for GLMM does not converge to a global maxima, for each cross-validation set, we run the algorithm multiple times from different starting values, and choose the run that produced the lowest negative log-likelihood.

```r
 #### Functions needed for cross validation
# My data set
data(faithful)
X <- as.matrix(faithful)
n <- dim(X)[1]

################################################
# 5-fold Cross-validation

permutation <- sample(1:n, replace = FALSE)
K <- 5
# Uneven folds, but that is ok
test.index <- split(permutation, rep(1:K, length = n, each = n/K))

# Testing whether 2-4 classes are needed
potC <- 2:4
CV.errorLike <- numeric(length = length(potC))

# will run EM multiple times for each training data
# since EM convergence to local minima. Setting these
# reps = 7
reps <- 5
model.save <- list()
for(c in 1:length(potC))
{
  foo3 <- 0
  for(k in 1:K)
  {
    print(c(c,k))
    X.train <- X[-test.index[[k]], ]
    X.test <- X[test.index[[k]], ]

    for(r in 1:reps)
    {
      model.save[[r]] <- GLMMforC(X = X.train, C = potC[c])
    }
    # which ever run is the lowest negative log-like
    chosen.run <- which.min(sapply(model.save, function(t) t$log.like))
    model <- model.save[[chosen.run]]
    foo3 <- foo3 + log_like(X = X.test, pi.list = model$pi, mu.list
       =model$mu, Sigma.list = model$Sigma, C = potC[c])
  }
  CV.errorLike[c] <- foo3/n
}
```

```
CV.errorLike #Lowest value is for C = 3,
# [1] 4.217456 4.215619 4.232590

# Thus choose C = 3 classes.
```

When you run the above code you see there are multiple issues with using CV for model selection in this example

- **Time:** The cross-validation is time consuming particularly because we have to run the mode for multiple starting values
- **Final model:** The method does not give you the final estimates, which means you need to run the model again for the chosen $C$. However, it is possible when you run it again that you converge to a different solution!

Thus, an alternative model selection procedure is used, called the *Akaike Information Criterion*:

$$\text{AIC}(\hat{\theta} \mid x) = -2 \log l(\hat{\theta}|x) + 2K$$

where $K$ is the number of parameters being estimated. $\text{AIC}(\hat{\theta} \mid x)$ is calculated by fitting the model on the *full dataset*. Since the negative log-likelihood will increase as you increase the number of clusters, the penalty term of $2K$ penalizes the usage of too many clusters.

In our 2-dimensional GMM example $K = 2 * C + (C - 1) + 3C = 6C - 1$.

Another similar information criterion is the *Bayesian Information Criterion* (BIC) defined as

$$\text{BIC}(\hat{\theta} \mid x) = -2 \log l(\hat{\theta}|x) + \log(n)K$$

where $K$ is again the number of parameters being estimated. The BIC criterion increases the penalty term when the number of data are larger. This makes sense since when large number of data are available, we should be able to find simpler models. It is well known that the BIC criterion is theoretically superior to the AIC criterion and even in this examples works better.

```
########################################
##### Model selection via AIC #######
########################################
aic <- function(X, pi.list, mu.list, Sigma.list, C)
{
  nlike <- log_like(X, pi.list, mu.list, Sigma.list, C)
  rtn <- 2*nlike + 2* (6*C - 1) # No. of params = 3*C - 1
  return(rtn)
}

bic <- function(X, pi.list, mu.list, Sigma.list, C)
{
  n <- dim(X)[1]
  nlike <- log_like(X, pi.list, mu.list, Sigma.list, C)
  rtn <- 2*nlike + log(n)* (6*C - 1) # No. of params = 3*C - 1
  return(rtn)
}

aicLike <- numeric(length = length(potC))
bicLike <- numeric(length = length(potC))
reps <- 5
model.save <- list()
model <- list()
for(c in 1:length(potC))
{
  print(c)
  for(r in 1:reps)
  {
    model.save[[r]] <- GLMMforC(X = X, C = potC[c])
  }

  chosen.run <- which.min(sapply(model.save, function(t) t$log.like))
  model[[c]] <- model.save[[chosen.run]]
  aicLike[c] <- aic(X = X, pi.list = model[[c]]$pi, mu.list =model[[c]]$mu,
      Sigma.list = model[[c]]$Sigma, C = potC[c])
  bicLike[c] <- bic(X = X, pi.list = model[[c]]$pi, mu.list =model[[c]]$mu,
      Sigma.list = model[[c]]$Sigma, C = potC[c])
}

aicLike # Lowest is C = 4 is the best!
# [1] 2282.528 2272.431 2255.077

bicLike # Lowest is C = 2
# [1] 2322.192 2333.730 2338.011
```

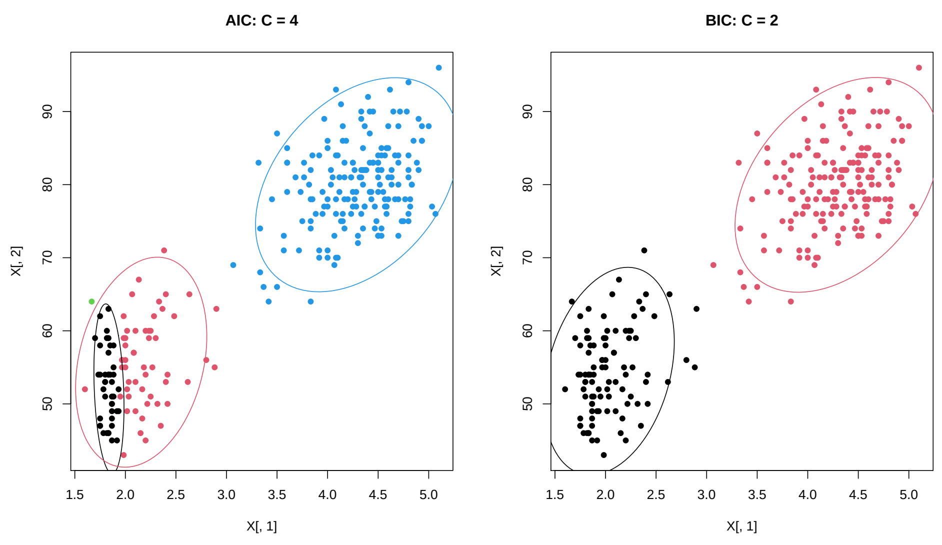
$C = 4$ gives the lowest AIC and $C = 2$ gives the lowest BIC. Note that since we've saved the model, we can now just use that model without rerunning anything. This is a very useful feature of model selection via AIC/BIC.

```
par(mfrow = c(1,2))
chosen <- which.min(aicLike)
allot <- apply(model[[chosen]]$Ep, 1, which.max) ## Final allotment of
    classification
plot(X[,1], X[,2], col = allot, pch = 16, main = "AIC: C = 4") # plot
    allotment

ell <- list()
for(c in 1:potC[[chosen]])
{
  ell[[c]] <- ellipse(model[[chosen]]$Sigma[[c]], centre =
      as.numeric(model[[chosen]]$mu[[c]]))
  lines(ell[[c]], col = c)
}

chosen <- which.min(bicLike)
allot <- apply(model[[chosen]]$Ep, 1, which.max) ## Final allotment of
    classification
plot(X[,1], X[,2], col = allot, pch = 16, main = "BIC: C = 2") # plot
    allotment

ell <- list()
for(c in 1:potC[[chosen]])
{
  ell[[c]] <- ellipse(model[[chosen]]$Sigma[[c]], centre =
      as.numeric(model[[chosen]]$mu[[c]]))
  lines(ell[[c]], col = c)
}
```



Clearly, using AIC, the model chooses $C = 4$, which is clearly not a good model since it has only one point in the fourth cluster. Clearly, the penalty added in AIC is not enough to ensure against over-fitting. Using BIC imposes extra penalty that helps against overfitting.

Thus, we recommend using BIC (and sometimes AIC) over CV for fitting Gaussian mixture models.