MTH 511a - 2020: Lecture 20

Instructor: Dootika Vats

The instructor of this course owns the copyright of all the course materials. This lecture material was distributed only to the students attending the course MTH511a: "Statistical Simulation and Data Analysis" of IIT Kanpur, and should not be distributed in print or through electronic media without the consent of the instructor. Students can make their own copies of the course materials for their use.

1 Resampling Methods

We will take a break from optimization methods to discuss methods for making *inference* in some of the data analysis problems we have discussed.

There are two main questions to ask:

1. *How do we choose model tuning parameters?*: In ridge/bridge/lasso regression, we require choosing λ and/or α. In Gaussian mixture models, we need to choose the number of clusters C.

2. *How good are our estimates?*: MLEs usually yield asymptotic normality for our estimates, so that that

$$\sqrt{n}(\hat{\theta}_{\text{MLE}} - \theta) \xrightarrow{d} N(0, \Sigma_{\text{MLE}}) \,,$$

   where Σ_{MLE} is the asymptotic covariance matrix. If we estimate Σ_{MLE}, we can assess the large-sample error in $\hat{\theta}_{\text{MLE}}$.

   However, for estimators that are not MLEs, like bridge regression, an asymptotic distribution is difficult to construct. So we need a method to assess understand the distribution of estimators.

Both points above can be addressed by *resampling* techniques which we will cover in this week. We will cover two resampling methods each to address two different concepts: *model selection* and *model performance*. Although both the methods can be used to address both things, we will focus on

- **Cross-validation:** for model selection – choosing tuning parameters

- **Bootstrapping:** to estimate model performance – obtain empirical distributions of estimators.

Cross-validation will be used to find the best tuning parameters. However, before we can proceed, we need to understand how to assess the quality of a model. First, we will learn a little about *loss functions*.

1.1 Loss functions

A loss function is a measure of error of an estimator from the true value. Having observed data, y, let $\hat{f}$ be the model fit. Then a loss function is a function $L(y, \hat{f})$ that quantifies the *distance* between y and $\hat{f}$. The form of the loss function may depend on the type of data. We will focus only on binary/Gaussian regression and the Gaussian mixture models.

- **Linear regression:** In penalized or non-penalized regression, we have an observation y, covariates X, and a regression coefficient, β. Let $\hat{\beta}$ be the estimated regression coefficient – this could be obtained via ridge, bridge, or regular regression. Then the estimated response is

  $$\hat{f}(x) = x^T\hat{\beta}\,.$$

  A loss function measures the error between y and the estimated response $\hat{y} = \hat{f}(x)$. For continuous response y, there are two popular loss functions:

  - Squared error: $L(y, \hat{f}(x)) = (y - \hat{f}(x))^2$
  - Absolute error: $L(y, \hat{f}(x)) = |y - \hat{f}(x)|$

  We will focus mainly on the squared error loss.

- **Binary data classification**: For binary data models, like logistic regression, a popular loss function is the *misclassification* also known as the $0-1$ loss. Given response y which are Bernoulli(p) where

  $$p = \frac{e^{X\beta}}{1 + e^{X\beta}}\,,$$

  we can find the estimated p, $\hat{p}$ by setting

  $$\hat{p} = \frac{e^{X\hat{\beta}_{\text{MLE}}}}{1 + e^{X\hat{\beta}_{\text{MLE}}}}\,,$$

  where $\hat{\beta}_{\text{MLE}}$ are the MLE estimates obtained using an optimization algorithm. These $\hat{p}$ are the estimated probability of success. Set $\hat{f}(x) = 1\cdot\mathbb{I}\{\hat{p} \geq .5\} + 0\cdot\mathbb{I}\{\hat{p} < .5\}$. (The cutoff, .5, is the default cutoff, but can be changed depending on the problem). Then the *misclassification* loss function checks whether the model has misclassified the observation

  $$\text{Misclassification or } 0-1 \text{ loss} : L(y, \hat{f}(x)) = \mathbb{I}(y \neq \hat{f}(x))\,.$$

- **Gaussian mixture model:** Where there are no "response" and "covariates", like the Gaussian mixture model (GMM) example, it is difficult to implement the above two loss functions. But recall, that in the GMM example, we used the log-likelihood to see whether our estimates were good. Thus, we can use negative log-likelihood to see how bad our estimates are. Thus, here the loss function is:

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

  We will discuss the specific GMM case in more detail later.

How do we use these loss functions?

Given a dataset, we are interested in estimating the *test error* which is the expected loss, of an independent dataset given estimates from the current dataset. If our given dataset is $\mathcal{D}$

$$\text{Err}_{\mathcal{D}} = \text{E}\left[L(y, \hat{f}(x)) \mid \mathcal{D}\right]\,,$$

where $\hat{f}(x)$ denotes the estimated y for a new independent dataset, given estimators from the current dataset. For example, using $\hat{\beta}$ from a given dataset $\mathcal{D}$, then $\hat{f}(x) = X_{\text{new}}\hat{\beta}$.

Test error is built for a given dataset $\mathcal{D}$. We are interested in the *expected prediction error* which is the expected test error over all such $\mathcal{D}$:

$$\text{Err} = \text{E}\left[L(y, \hat{f}(x))\right] = \text{E}\left[\text{E}\left[L(y, \hat{f}(x)) \mid \mathcal{D}\right]\right] = \text{E}\left[\text{Err}_{\mathcal{D}}\right]$$

The expected prediction error is the expected error in predicting y for new datasets given models built from any dataset.

So, given different values of a tuning parameter, our goal is to compare the prediction error, and choose the tuning parameter which yields the lowest test error. But, the test error requires obtaining the loss on *independent datasets*. We just have one dataset available to use.

This is where cross-validation is critically useful.

1.2 Cross-validation

Cross-validation provides a way to estimate the prediction error. In cross-validation, our original dataset is broken into chunks in order to emulate independent datasets. Then the model is fit on one chunk and tested on another, with the loss recorded. The way these broken into chunks can lead to different methods.

1.2.1 Leave-one-out Cross-Validation

In leave-one-out cross-validation (LOOCV), the data $\mathcal{D}$ of size n is split a *training set* of size $n-1$ and a *test set* of size 1. This is repeated for systematically all observations so that there are n such splits possible.

For each split, the test error is estimated, and the average error over all splits is calculated, which estimates the expected prediction error for a model fit $\hat{f}(x)$. Let $\hat{f}^{-i}(x_i)$ denote the predicted value of y_i using the model that removes the ith data point. Then CV estimate of prediction error is

$$\text{CV}_1(\hat{f}) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{f}^{-i}(x_i))\,.$$

Note that each $\hat{f}^{-i}(x_i)$ represents model fits for different datasets, with testing on one observation. Thus $\text{CV}_1(\hat{f})$ estimates the prediction error, with n observations for the outer expectation and one observation for the inner expectation.

$\text{CV}_1(\hat{f})$ can be calculated for different models or tuning parameters:

$$\text{CV}_1(\hat{f},\lambda) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{f}^{-i}(x_i,\lambda))\,.$$

The chosen model is the one with λ such that

$$\lambda_{\text{chosen}} = \arg\min_{\lambda}\left\{\text{CV}_1(\hat{f},\lambda)\right\}$$

The final model is $\hat{f}(X, \lambda_{\text{chosen}})$ fit to *all* the data. In this way we can accomplish two things: obtain an estimate of the prediction error and choose a model.

Points:

- $CV_1(\hat{f})$ is an approximately unbiased estimator of the expected prediction error
- This is computationally burdensome since the model is fit n times for each λ.

1.2.2 K-fold cross-validation

The data is randomly split into K roughly equal-sized parts. For any kth split, the rest of the $K-1$ parts make up the *training set* and the model is fit to the *training*

set. We then estimate the test error error for each element in the kth part. Repeating this for all $k = 1, 2, \dots, K$ parts, we have an averaged prediction error.

Let $\kappa : \{1, \dots, N\} \mapsto \{1, \dots, K\}$ indicates the partition to which each ith observation belongs. Let $\hat{f}^{-\kappa(i)}(x)$ be the fitted function for the $\kappa(i)th$ partition removed. Then, the estimated prediction error is

$$\text{CV}_K(\hat{f}, \lambda) = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{n/K} \sum_{i \in k^{\text{th}}\text{split}} L(y_i, \hat{f}^{-\kappa(i)}(x_i, \lambda)) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{f}^{-\kappa(i)}(x_i, \lambda))\,.$$

The chosen model is the one with λ such that

$$\lambda_{\text{chosen}} = \arg\min_{\lambda} \left\{ \text{CV}_K(\hat{f}, \lambda) \right\}$$

The final model is $\hat{f}(X, \lambda_{\text{chosen}})$ fit to *all* the data.

Points:

- For small K, the bias is large since the inner loss function is not estimating $\text{Err}_{\mathcal{D}}$ but $\text{Err}_{\mathcal{D}_1}$, where $\mathcal{D}_1$ is considerably smaller than $\mathcal{D}$.
- Also, the computational burden is lesser when K is small.

Usually, for large datasets, 10-fold or 5-fold CV is common. For small datasets, LOOCV is more common.

1.3 Comparing different cross-validations

Before going into examples of using cross-validation, we will demonstrate the estimation quality of various cross-validation techniques: Leave-one-out cross-validation (LOOCV), 10-fold CV, and 5-fold CV.

```
##################################################
## Implementing CV methods for a simulated dataset
## We will generate X and y and generating new independent
## X.new and y.new, we will calculate the "true" prediction error
## Then we implement the LOOCV, and K-fold CV
##################################################
set.seed(10)

n <- 100
```

```r
p <- 50
sigma2.star <- 4
beta.star <- rnorm(p, mean = 2)
beta.star # to output

# repeat 500 times
B <- 5e2
foo <- 0

# Setting size of test data and training data
test.size <- floor(n/5)

# we will use the following code to make our K-fold splits
permutation <- sample(1:n, replace = FALSE) # random permutation of 1:n
K <- 5
# Making a list of indices for each split
(test.index <- split(permutation, rep(1:K, length = n, each = n/K)))

#$`1`
# [1] 31 87 1 59 25 39 16 98 99 65 12 52 58 95 94 24 22 13 9 89
#
#$`2`
# [1] 100 61  85  97  91  47  92  70  75  88  68   4  29  21  30  56
#[17]   7  26  28  57
#
#$`3`
# [1] 62 38 34 53 27 2 73 23 74 55 33 81 10 44 76 60 90 17 20 80
#
#$`4`
# [1] 18 36 48 64 86 46 45 66 11 93 49 84 40 5 19 96 41 77 63 50
#
#$`5`
# [1] 42 3 54 43 35 14 51 37 67 72 79 83 8 78 6 82 15 32 71 69


CV.error <- matrix(0, nrow = B, ncol = 4)
time <- matrix(0, nrow = B, ncol = 3)
colnames(CV.error) <- c( "Truth","LOOCV", "10-fold", "5-fold")
```

In the above code, we have obtained a true β^* with $p = 50$ and we have set $n = 100$. I also provide the code I will use to make CV splits.

Next, we will simulate a dataset, $\mathcal{D} = (y, X)$ multiple times and estimate a regression coefficient β for each of those datasets. Then for every dataset, we will generate new independent copies of new data for testing. This will allow us to understand what the true prediction error will be, and allow us to compare CV estimates with the truth.

```r
for(b in 1:B)
{
    # code takes a while, hence printing for feedback
    if(b %% 100 == 0) print(b)

    # Generate new d
    # Making design matrix, first column is 1
    X <- cbind(1, matrix(rnorm(n*(p-1)), nrow = n, ncol = (p-1)))

    # Generating response
    y <-  X %*% beta.star + rnorm(n, mean = 0, sd = sqrt(sigma2.star))

    beta.mle <- solve( t(X) %*% X) %*% t(X) %*% y

    # independent new X and y
    X.new <- cbind(1, matrix(rnorm(n*(p-1)), nrow = n, ncol = (p-1)))
    y.new <- X.new %*% beta.star + rnorm(n, mean = 0, sd = sqrt(sigma2.star))

    CV.error[b, 1] <- mean( (y.new - X.new %*% beta.mle)^2 )



    ################################################
    # Leave-one-out Cross-validation

    time0 <- proc.time()[3]
    foo2 <- 0
    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) %*% t(X.train) %*% y.train

      # test error
      foo2 <- foo2 + (y[i] - X[i,] %*% beta.train)^2
    }
    CV.error[b, 2] <- foo2/n
    time[b,1] <- proc.time()[3] - time0


    ################################################
    # 10-fold Cross-validation
    permutation <- sample(1:n, replace = FALSE)
```

```
  K <- 10

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

  time0 <- proc.time()[3]
  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 <- solve(t(X.train) %*% X.train) %*% t(X.train) %*% y.train

    foo3 <- foo3 + sum( (y.test - X.test %*% beta.train)^2)
 }
 CV.error[b, 3] <- foo3/n
 time[b,2] <- proc.time()[3] - time0

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

 # Making a permutation from 1:n
 permutation <- sample(1:n, replace = FALSE)
 K <- 5
 test.index <- split(permutation, rep(1:K, length = n, each = n/K))

 time0 <- proc.time()[3]
 foo4 <- 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 <- solve(t(X.train) %*% X.train) %*% t(X.train) %*% y.train

   foo4 <- foo4 + sum( (y.test - X.test %*% beta.train)^2)
 }
 CV.error[b, 4] <- foo4/n
 time[b,3] <- proc.time()[3] - time0
################################################
}
```

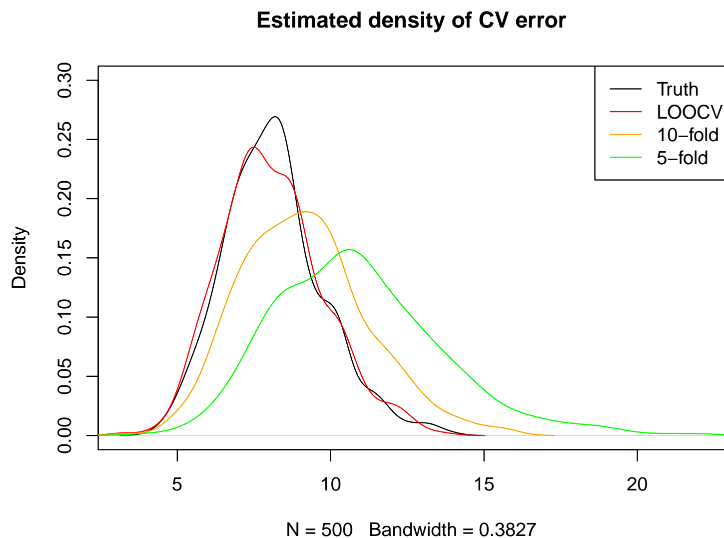
```
# LOOCV is most accurate
colMeans(CV.error)
#    Truth    LOOCV   10-fold   5-fold
# 8.146910 8.101374 9.124679 10.882481

# LOOCV is expensive
colMeans(time)
# [1] 0.069552 0.006628 0.002676
```

We see that LOOCV yields estimates closest to the truth, with 5-fold being the farthest from the truth. However, 5-fold is the fastest to implement and LOOCV is significantly slower.

Since we have $B = 500$ estimates of the prediction error, we can compare their overall distribution as well:

```
# LOOCV is the closest
plot(density(CV.error[,1]), xlim = range(CV.error), ylim = c(0, .30), main
    = "Estimated density of CV error")
lines(density(CV.error[,2]), col = "red")
lines(density(CV.error[,3]), col = "orange")
lines(density(CV.error[,4]), col = "green")
legend("topright", legend = colnames(CV.error), col = c("black", "red",
    "orange", "green"), lty = 1)
```



Thus, through this simulation, we see that LOOCV is the closest to the truth, however, if the dataset is large, it can be too time consuming to implement it. On the other hand, 5-fold CV is not that accurate. If computation allows it, 10-fold CV is a good sacrifice

between computation time and accuracy. We will present examples of implementation of cross-validation for regression and Gaussian mixture model.

2 Questions to think about

- What happens to the log-likelihood for Gaussian mixture model if we keep increasing C?
- Would would happen to the quality of estimation of the prediction error if we increase n, the sample size?