Lab 5: Cross-validation

PSTAT 131/231, Fall 2023

```
# Load libraries
library(ISLR)
library(dplyr)
library(reshape2)
library(ggplot2)
library(class)
```

---

1. Review of Lab2

Recall in Lab 2 we went over the K-nearest neighbor methods. In particular, we used the dataset `Carseats` in package `ISLR` to make an example of classification task.

```
# Obtain Carseats from ISLR using data()
data(Carseats)
# Check the structure by str()
str(Carseats)
# Get dataset info
?Carseats
```

As a reminder, `Carseats` is a simulated data set containing sales of child car seats at 400 different stores on 11 features (3 discrete and 8 numerical). We create a new feature `High` as the response variable following the rule:

$$High = \begin{cases} \text{No}, & \text{if Sales} \leq \text{median(Sales)} \\ \text{Yes}, & \text{if Sales} > \text{median(Sales)} \end{cases}$$

We further delete three categorical variables (`ShelveLoc`, `Urban` and `US`) and the continuous `Sales`[1] from the original data. We call the resulting dataset `seats`:

```
# Create the binary response variable High, drop Sales and 3 discrete independent
# variables as well. Call the new dataset seats
seats = Carseats %>%
    mutate(High=as.factor(ifelse(Sales <= median(Sales), "Low", "High"))) %>%
    select(-Sales, -ShelveLoc, -Urban, -US)
```

In this lab, we will work on the binary response variable `High`, as well as other continuous variables except for `Sales`. The goal is to investigate the relationship between `High` and all continuous variables but `Sales` using K-NN classifiers. Since different values of K correspond to different models, we are mostly interested in using cross-validation to select the best model.

In particular, we will learn how to

- Use LOOCV to select the best number of neighbors by `knn.cv()`
- Use k-fold CV to select the best number of neighbors by `do.chunk()`

[1] Note: we delete `Sales` from the explanatory variables is due to the fact that `High` is derived from it.

- Plot Training and Validation error along with k
- Compute Training and Test error rates

Just as in Lab 2, we do the following training/test split.

```
# Set random seed
set.seed(333)

# Sample 50% observations as training data
train = sample(1:nrow(seats), 200)
seats.train = seats[train,]

# The rest 50% as test data
seats.test = seats[-train,]
```

For later convenience purposes, we create `XTrain`, `YTrain`, `XTest` and `YTest`. `YTrain` and `YTest` are response vectors from the training set and the test set. `XTrain` and `XTest` are design matrices[2].

```
# YTrain is the true labels for High on the training set, XTrain is the design matrix
YTrain = seats.train$High
XTrain = seats.train %>% select(-High) %>% scale(center = TRUE, scale = TRUE)

# YTest is the true labels for High on the test set, Xtest is the design matrix
YTest = seats.test$High
XTest = seats.test %>% select(-High) %>% scale(center = TRUE, scale = TRUE)
```

2. k-fold and Leave-One-Out Cross-validation for selecting best number of neighbors

Cross-validation is a very general method for estimating test error, and essentially can be used with any kind of predictive modeling.

- **k-fold Cross-Validation**

  This approach involves randomly dividing the set of observations into k groups, or folds, of approximately equal size. The first fold is treated as a validation set, and the method is fit on the remaining $k-1$ folds (which is treated as a training set). This procedure is repeated k times; each time, a different group of observations is treated as a validation set and the rest $k-1$ folds as a training set.

  For regression problems, the mean squared error, MSE_k, is then computed on the observations in the k^{th} hold-out fold. This process results in k estimates of the test error: MSE_1, MSE_2, ..., MSE_k. The k-fold CV error is computed by averaging these values,

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} MSE_i$$

  For classification problems, the error rate, Err_k is computed on the observations in the k^{th} hold-out fold. Similarly as in regresssion cases, the CV process yield Err_1, Err_2, ..., Err_k. Then the k-fold CV error rate takes the form

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} Err_i, \text{ where } Err_i = I_{(y_i \neq \hat{y}_i)}$$

- **Leave-One-Out Cross-validation (LOOCV)**

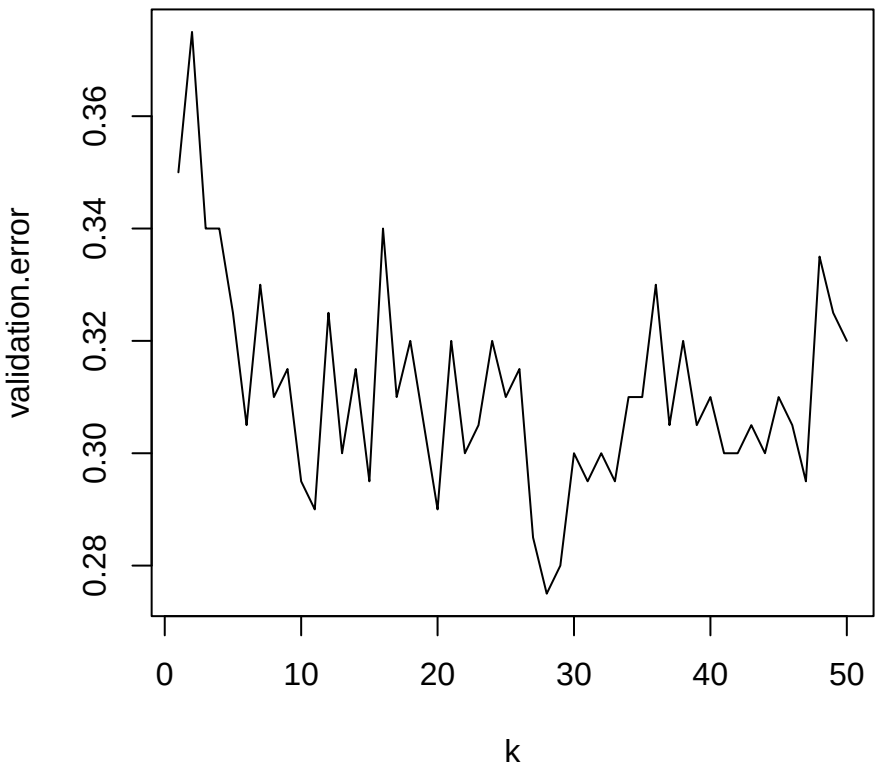
[2]Design matrix, also known as regressor matrix or model matrix, is a matrix of values of explanatory variables, often denoted by X. Each row represents an individual observation, with the successive columns corresponding to the variables and their specific values for that object.

It is not hard to see that LOOCV is a special case of k-fold CV in which k is set to equal n. LOOCV has the potential to be expensive to implement, since the model has to be fit n times. This can be very time consuming if n is large, and if each individual model is slow to fit.

(a). LOOCV

- `knn.cv()` does a k-nearest neighbor classification from training set using LOOCV. There are at at least three arguments to be specified:
  - *train*: a matrix containing the predictors associated with the training data, i.e., design matrix of training set, denoted as `XTrain` below
  - *cl*: a vector containing the class labels for the training observations, labeled as `YTrian` below
  - k: the number of nearest neighbors considered
- Recall that we want to find the best number of neighbors in kNN, which can be completed by the following R chunk:

```r
# Give possible number of nearest neighbours to be considered
allK = 1:50
# Set validation.error (a vector of length 50) to save validation errors in future
# where validation.error[i] is the LOOCV validation when i-NN method is considered
validation.error = rep(NA, 50)

# Set random seed to make the results reproducible
set.seed(66)

# For each number in allK, use LOOCV to find a validation error
for (i in allK){
  # Loop through different number of neighbors
  # Predict on the left-out validation set
  pred.Yval = knn.cv(train=XTrain, cl=YTrain, k=i)
  # Combine all validation errors
  validation.error[i] = mean(pred.Yval!=YTrain)
}

# Validation error for 1-NN, 2-NN, ..., 50-NN
plot(allK, validation.error, type = "l", xlab = "k")
```




```
# Best number of neighbors
#     if there is a tie, pick larger number of neighbors for simpler model
numneighbor = max(allK[validation.error == min(validation.error)])
numneighbor
```

```
## [1] 28
```

- After determining the best number k for kNN, we train the 28-NN classifier and compute the test error rate on the test dataset. Recall that the cross-validation is carried out on the training set.

```
# Set random seed to make the results reproducible
set.seed(67)

# Best k used
pred.YTest = knn(train=XTrain, test=XTest, cl=YTrain, k=numneighbor)

# Confusion matrix
conf.matrix = table(predicted=pred.YTest, true=YTest)
conf.matrix
```

```
##          true
## predicted High Low
##      High   77  38
##      Low    22  63
```

```r
# Test error rate
1 - sum(diag(conf.matrix)/sum(conf.matrix))
```

```
## [1] 0.3
```

Can you compute the TPR, FPR of this 46-NN classifier?

(b). k-fold Cross-validation (k-fold CV)

- `do.chunk()` is the key function (defined below) for carrying out the k-fold Cross-validation[3]. The function takes in `chunkid`, `folddef`, `Xdat`, and `Ydat` as the main argumetns. `folddef` specifies a index mapping for all observations in `Xdat` (design matrix) and `Ydat` (response) into a set of folds (k folds in total for k-fold CV). `chunkid` is the fold id that should be treated as validation set, and `do.chunck()` will train the model using all but `chunkid`-th fold as the training set, and compute the validation error on the `chunkid`-th fold.

  The `do.chunck()` function returns a data frame consisting of all possible values of folds[4], each training error and validation error correspondingly. The purpose is to select the best number of neighbors using a k-fold CV and to calculate the test error rate afterwards. To illustrate the point, we will perform a 3-fold CV.

```r
# do.chunk() for k-fold Cross-validation
do.chunk <- function(chunkid, folddef, Xdat, Ydat, ...){
  # Get training index
  train = (folddef!=chunkid)

  # Get training set by the above index
  Xtr = Xdat[train,]
  # Get responses in training set
  Ytr = Ydat[train]

  # Get validation set
  Xvl = Xdat[!train,]
  # Get responses in validation set
  Yvl = Ydat[!train]

  # Predict training labels
  predYtr = knn(train=Xtr, test=Xtr, cl=Ytr, ...)
  # Predict validation labels
  predYvl = knn(train=Xtr, test=Xvl, cl=Ytr, ...)

  data.frame(fold = chunkid,
             train.error = mean(predYtr != Ytr), # Training error for each fold
             val.error = mean(predYvl != Yvl)) # Validation error for each fold

}
```

- Firstly, we specify $k = 3$ in k-fold CV, and use `cut()` and `sample()` to assign an interval index (1, 2, or 3) to each observation in the training set.
  - `cut()` divides the range of a variable into several intervals and assigns a chunk name (or number) to each value in that variable.

    The first argument in `cut()` should be a numeric vector, which is to be converted to intervals.

[3] '...' notation is used when defining `do.chunk()` function. This syntax is called dot-dot-dot. `do.chunk()` stores all but the first four variables in the ellipsis variable.

[4] The k in k-fold CV is different from the K in K-NN!

breaks: controls how to cut the vector. We can either specify several cut points, or the number of intervals into which the variable is to be cut.

labels: displays the levels of the resulting categories. By default, labels are constructed using (a,b] interval notation. If `labels = FALSE`, simple integer codes are returned. The leftmost interval corresponds to interval 1, the next to interval 2 and so on.

– **`sample()`** takes a sample of the specified size from the elements specified in the first argument.

size: the number of items to choose. The default for `size` is the number of items inferred from the first argument, so that `sample(x)` generates a random permutation of the elements of $x (or\ 1:x)$.

replace: controls whether sampling should be with replacement. By default, `replace=FALSE`, that is, sampling is without replacement.

In the following R chunk, we divide all observations from the training set into 3 intervals, namely, interval 1, 2 and 3. To make the division more random, we sample from all the interval indices without replacement. Call the resulting vector `folds`.

```
# Specify we want a 3-fold CV
nfold = 3

# cut: divides all training observations into 3 intervals;
#      labels = FALSE instructs R to use integers to code different intervals
set.seed(66)
folds = cut(1:nrow(seats.train), breaks=nfold, labels=FALSE) %>% sample()
folds
```

```
##   [1] 2 2 3 2 1 3 3 2 2 1 1 2 1 3 1 1 2 2 3 3 2 1 1 2 3 2 3 2 3 1 1 3 2 1 3 2 3
##  [38] 3 2 1 1 3 1 3 3 2 3 2 2 1 2 3 3 2 3 1 3 1 1 2 1 2 1 1 2 1 2 3 2 1 2 2 3 3
##  [75] 3 3 1 1 1 2 1 3 3 1 1 1 3 3 1 2 1 3 2 3 1 1 2 2 3 3 1 1 3 3 3 2 3 3 3 2 2
## [112] 3 2 2 3 1 2 2 2 3 3 3 2 1 2 2 2 3 1 2 1 1 1 2 3 3 2 1 2 1 1 1 3 3 2 1 1 2
## [149] 3 1 3 2 3 3 1 1 1 3 1 2 1 2 2 1 3 3 2 3 2 1 1 2 1 2 3 1 1 3 3 2 1 2 2 1 2
## [186] 2 1 2 3 3 2 3 1 1 2 3 1 3 3 1
```

```
# Set error.folds (a vector) to save validation errors in future
error.folds = NULL

# Give possible number of nearest neighbours to be considered
allK = 1:50

# Set seed since do.chunk() contains a random component induced by knn()
set.seed(888)

# Loop through different number of neighbors
for (k in allK){
  # Loop through different chunk id
  for (j in seq(3)){
    tmp = do.chunk(chunkid=j, folddef=folds, Xdat=XTrain, Ydat=YTrain, k=k)

    tmp$neighbors = k # Record the last number of neighbor

    error.folds = rbind(error.folds, tmp) # combine results
  }
}
head(error.folds, 10)
```

```
##    fold train.error val.error neighbors
```

```
## 1     1   0.0000000 0.3731343         1
## 2     2   0.0000000 0.3484848         1
## 3     3   0.0000000 0.3134328         1
## 4     1   0.1578947 0.3582090         2
## 5     2   0.2238806 0.3484848         2
## 6     3   0.1654135 0.3731343         2
## 7     1   0.1729323 0.3880597         3
## 8     2   0.1492537 0.3484848         3
## 9     3   0.1954887 0.3283582         3
## 10    1   0.2406015 0.3582090         4
```

Make sure that you check the data frame `error.folds` to see what values are calculated.

- Thirdly, we have to decide the optimal k for kNN based on `error.folds`.
  - **`melt()`** in the package `reshape2` takes wide-format data and melts it into long-format data. By default, `melt()` has assumed that all columns with numeric values are variables with values. Often this is what you want.

    varnames: specifies variable names to use in the molten data frame

    id.vars: the variables that identify individual rows of data
  - **`ungroup()`** in the package `dplyr` is a convenient inline function of removing existing grouping.

For each fold and each neighbor, we want the type of error (training/test) and the corresponding value. We can do it with the help of `melt()` by telling it that we want `fold` and `neighbor` to be `id.vars`.

```
# Transform the format of error.folds for further convenience
errors = melt(error.folds, id.vars=c('fold', 'neighbors'), value.name='error')

# Choose the number of neighbors which minimizes validation error
val.error.means = errors %>%
    # Select all rows of validation errors
    filter(variable=='val.error') %>%
    # Group the selected data frame by neighbors
    group_by(neighbors, variable) %>%
    # Calculate CV error rate for each k
    summarise_each(funs(mean), error) %>%
    # Remove existing group
    ungroup() %>%
    filter(error==min(error))

# Best number of neighbors
#     if there is a tie, pick larger number of neighbors for simpler model
numneighbor = max(val.error.means$neighbors)
numneighbor
```

```
## [1] 31
```

- Fourthly, we train a 31-NN classifier, and calculate the test error rate (on the test set!).

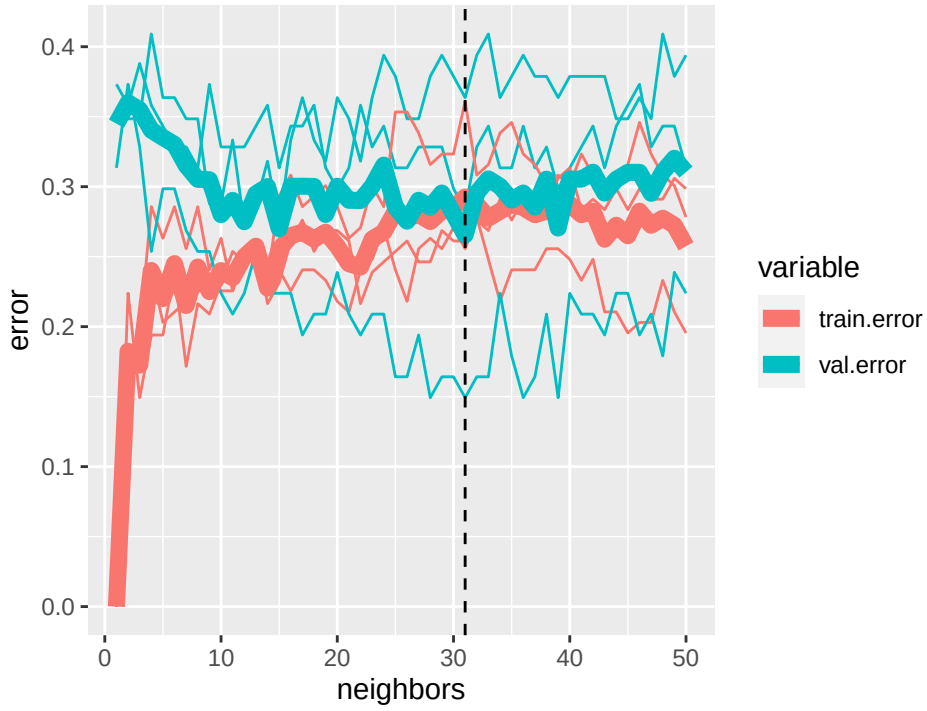
```
set.seed(99)
pred.YTest = knn(train=XTrain, test=XTest, cl=YTrain, k=numneighbor)

# Confusion matrix
conf.matrix = table(predicted=pred.YTest, true=YTest)
```

```
# Test error rate
1 - sum(diag(conf.matrix)/sum(conf.matrix))
```

```
## [1] 0.36
```

- Lastly, we plot training errors and validation errors along with the number of neighbors. In the graph below, thin lines are error rates for each fold in CV; thick lines are mean of the errors; vertical dashed line is the minimum of average validation errors across number of neighbors.

```
# Plot errors
ggplot(errors, aes(x=neighbors, y=error, color=variable))+
    geom_line(aes(group=interaction(variable,fold))) +
    stat_summary(aes(group=variable), fun.y="mean", geom='line', size=3) +
    geom_vline(aes(xintercept=numneighbor), linetype='dashed')
```



Your turn

Perform a 6-fold CV to select the best number of neighbors in kNN.

```
# Code starts here
```