Lab 2: k-Nearest Neighbors

PSTAT 131/231, Spring 2018

Learning Objectives

- k-Nearest Neighbors
  - Training/Test split
  - 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()`
  - Plot Training and Validation error along with k
  - Compute Training and Test error rates

---

1. Install packages and obtatin `Carseats` dataset

In this section, we will primarily focus on performing k-Nearest Neighbors using package `class`, which contains various functions for classification, including k-Nearest Neighbor, Learning Vector Quantization and Self-Organizing Maps. In addition, we will intentionally use `dplyr` as much as possible as it is our main utility package, and we will generate graphs with `ggplot2` and `reshape2`.

The following packages are needed to assist our analysis:

```
# install.packages("ISLR")
# install.packages("ggplot2")
# install.packages("plyr")
# install.packages("dplyr")
# install.packages("class")

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

The dataset `Carseats` in package `ISLR` is adopted again to make an example 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). Last time, we created a new feature `High` as the response variable following the rule:

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

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`.

To achieve this goal, we will 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)

# Check column names of seats
colnames(seats)
str(seats)

# Another way to create seats using quantile()
seats = Carseats %>%
   mutate(High=as.factor(ifelse(Sales <= quantile(Sales, probs=0.5), "Low", "High"))) %>%
   select(-Sales, -ShelveLoc, -Urban, -US)
```

2. k-Nearest Neighbors (a.k.a. kNN, k-NN, knn)

(a). Training/Testing split

Given a data set, the use of a particular statistical learning method is warranted if it results in a low test error. The test error can be easily calculated if a designated test set is available. Therefore, before performing kNN on the data, we first sample 50% of the observations as a training set, and the other 50% as a test set. Note that we set a random seed before applying `sample()` to ensure reproducibility of results.

```
# 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)

# 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)
```

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

[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.

(b). Train a kNN classifier and calculate error rates

- `knn()` function in the package `class` can be used to train a kNN classifier. This function works rather differently from the other model-fitting functions that we have encountered thus far. Rather than a two-step approach in which we first fit the model and then we use the model to make predictions, `knn()` forms predictions using a single command. The function requires at least four inputs.
  - *train*: a matrix containing the predictors associated with the training data, i.e., design matrix of training set, labeled `Xtrain` below
  - *test*: a matrix containing the predictors associated with the data for which we wish to make predictions, i.e., design matrix
  - *cl*: a vector containing the class labels for the training observations, labeled `Ytrain` below.
  - k: the number of nearest neighbors to be used by the classifier.
- Now we apply `knn()` function to train the kNN classifier on the training set and make predictions on training and test sets.

  Notice that we set a random seed before applying `knn()` to ensure reproducibility of results. The random component in `knn()` is that if several observations are tied as nearest neighbors, then R will randomly break the tie, so a fixed seed instructs R to break the tie in the same way as we run the function multiple times.
  - **Calculate training error rate** Recall that there are at least four arguments in `knn()` that should be specified. In order to get the training error, we have to train the kNN classifier on the **training** set and predict `High` on the same **training** set, then we can construct the 2 by 2 confusion matrix to get the training error rate. Based on this idea, we should have `train=XTrain`, `test=XTrain`, and `cl=YTrain` in `knn()`. We assume `k=2` is the best number of nearest neighbors for now, so we train a 2-NN classifier.

```
set.seed(444)

# knn - train the classifier and make predictions on the TRAINING set!
pred.YTtrain = knn(train=XTrain, test=XTrain, cl=YTrain, k=2)

# Get confusion matrix
conf.train = table(predicted=pred.YTtrain, true=YTrain)
conf.train
```

```
##          true
## predicted High Low
##      High   75  15
##      Low    26  84
```

```
# Test accuracy rate
sum(diag(conf.train)/sum(conf.train))
```

```
## [1] 0.795
```

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

```
## [1] 0.205
```

  - **Calculate test error rate** To get the test error, we have to train the kNN classifier on the **training** set and predict `High` on the **test** set, then again we can construct the 2 by 2 confusion matrix to get the test error rate. Based on this idea, we should have `train=XTrain`, `test=XTest`,

and `cl=YTrain` in `knn()`. Similarly as above, we set a random seed and try to train the 2-NN classifier.

```
set.seed(555)

# knn - train the classifier on TRAINING set and make predictions on TEST set!
pred.YTest = knn(train=XTrain, test=XTest, cl=YTrain, k=2)

# Get confusion matrix
conf.test = table(predicted=pred.YTest, true=YTest)
conf.test
```

```
##          true
## predicted High Low
##      High   52  41
##      Low    46  61
```

```
# Test accuracy
sum(diag(conf.test)/sum(conf.test))
```

```
## [1] 0.565
```

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

```
## [1] 0.435
```

The test error rate obtained by 2-NN classifier is not ideal enough, since 43.5% of the test observations are incorrectly predicted. It will be no surprise to us that changing the number of neighbours (k) in the above `knn()` function will lead us to different training/test error rates. Among all possible values of neighbours, which would be the best one and what is the strategy to find this optimal value? The answer, in short, is by Cross-validation.

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

Cross-validation is a very general method, and 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. 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.

  In general, 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,

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

  Specially, 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)**

  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:

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

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

# 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
    pred.Yval = knn.cv(train=XTrain, cl=YTrain, k=i) # Predict on the left-out validation set
    validation.error = c(validation.error, mean(pred.Yval!=YTrain)) # Combine all validation errors
}

# Validation error for 1-NN, 2-NN, ..., 50-NN
validation.error
```

```
##  [1] 0.435 0.400 0.440 0.420 0.445 0.475 0.465 0.470 0.475 0.465 0.500
## [12] 0.475 0.490 0.465 0.480 0.475 0.445 0.440 0.450 0.465 0.460 0.465
## [23] 0.450 0.460 0.450 0.445 0.440 0.440 0.455 0.420 0.435 0.425 0.435
## [34] 0.435 0.440 0.420 0.435 0.460 0.440 0.440 0.430 0.450 0.470 0.465
## [45] 0.480 0.490 0.485 0.455 0.440 0.445
```

```
# 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] 2
```

- After determining the best number k for kNN, we train the 2-NN classifier and compute the test error rate.

```
# 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   57  42
##      Low    41  60
```

```
# Test accuracy rate
sum(diag(conf.matrix)/sum(conf.matrix))
```

```
## [1] 0.585
```

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

```
## [1] 0.415
```

Note: here we get the test error rate as 41.5% from a 2-NN classifier, which is different from the test error rate, 43.5%, in (2b), even though the latter also uses a 2-NN classifier. This is due to the different seed values we set before running `knn()` each time.

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

- `do.chunk()` is the function we defined for k-fold Cross-validation[3]. The function returns a data frame consisting of all possible values of folds[4], each training error and validation error correspondingly. We have seen and used `do.chunk()` in homework 2 and will utilize it again. 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.

  Let's revisit `do.chunk()`:

```
# do.chunk() for k-fold Cross-validation

do.chunk <- function(chunkid, folddef, Xdat, Ydat, ...){ # Function arguments

    train = (folddef!=chunkid) # Get training index

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

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

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

    data.frame(fold = chunkid, # k folds
               train.error = mean(predYtr != Ytr), # Training error for each fold
```

[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 kNN!

```
                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.

    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.
  - `ldply()` applies function for each element of a input list, then combines results into a data frame. It is from the package `plyr`.

  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] 3 3 2 2 2 2 3 3 2 3 1 1 1 3 3 1 2 1 2 1 1 1 1 2 1 1 1 3 1 1 2 1 3 1 2
##  [36] 1 2 3 2 2 2 3 2 3 3 1 3 3 3 2 1 1 2 3 2 1 1 1 1 1 3 3 3 2 3 3 3 1 1 3
##  [71] 2 1 1 2 2 3 3 2 3 2 2 3 1 2 2 2 1 1 2 1 3 3 3 3 3 3 3 1 1 3 3 2 2 3 2
## [106] 1 3 2 1 2 2 3 1 2 3 1 2 3 2 1 1 1 3 2 1 2 3 2 3 3 3 1 3 2 2 3 3 3 1 3
## [141] 1 1 1 2 3 3 2 2 1 2 3 3 1 2 1 1 1 2 1 2 1 2 2 3 1 2 1 2 1 3 3 3 3 1 3
## [176] 1 2 2 2 1 1 1 1 2 2 3 2 2 3 1 2 3 2 2 3 2 3 1 2 1
```

- Secondly, use `do.chunk()` to perform a 3-fold CV for selecting the best number of neighbors. The idea is pretty similar to LOOCV procedure, but instead of doing a 200-fold CV in LOOCV, we only do a 3-fold CV.

```
# 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 (j in allK){

    tmp = ldply(1:nfold, do.chunk, # Apply do.chunk() function to each fold
                folddef=folds, Xdat=XTrain, Ydat=YTrain, k=j)
                # Necessary arguments to be passed into do.chunk

    tmp$neighbors = j # Keep track of each value of neighors

    error.folds = rbind(error.folds, tmp) # combine results

}
```

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`[5] 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))
```

```
## `summarise_each()` is deprecated.
## Use `summarise_all()`, `summarise_at()` or `summarise_if()` instead.
## To map `funs` over a selection of variables, use `summarise_at()`
```

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

[5]Note: if you are not familiar with any of `filter()`, `group_by()` and `summarise_each()`, please look for the details in Lab02 material.

```
## [1] 49
```

- Fourthly, we train a 49-NN classifier, and calculate the test error rate.

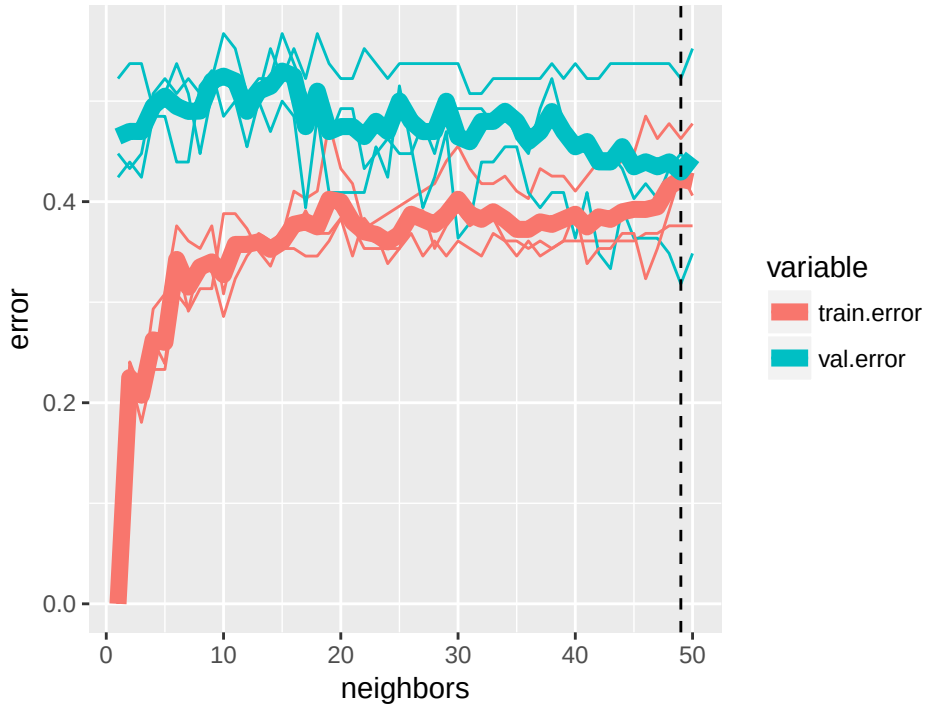
```
set.seed(99)
pred.YTest = knn(train=XTrain, test=XTest, cl=YTrain, k=numneighbor)

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

# Test accuracy rate
sum(diag(conf.matrix)/sum(conf.matrix))
```

```
## [1] 0.535
```

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

```
## [1] 0.465
```

- 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.



Your turn

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

```
# Code starts here
```