Stats 415 Lab3

April Cho

1/24/2018

Today's objectives

1. Learn how to split Training and test data.
2. Practice computing training, test error
3. Learn how to implement K nearest neighbors for regression and classification

Data and necessary packages

We first load some necessary libraries. To illustrate regression, we use `Boston` data from the `MASS` package. We discuss classification using `Default` data from the `ISLR` package.

```
library(ISLR)
library(MASS)
```

Training and Test Data

We split the data into training and test sets. Training data is used to fit a model(i.e. learn model) and test data is used to validate the fitted model. For this lab, we will use data 'Boston' from 'MASS' library. Since `Boston` is relatively large, we'll use about 70% of the data for training, and the remaining 30% for testing. To randomly sample about 70% of the data for training, we use R function `sample()`

```
library(MASS)

set.seed(100) # For reproducibility
# Randomly pick 70% of observations to use as training
train_id <- sample(1:nrow(Boston), floor(nrow(Boston) * .7))
# Subset Boston to include only the selected observations
trainBoston <- Boston[train_id, ]
# Subset Boston again to include only indices NOT sampled
testBoston <- Boston[-train_id, ]
```

MSE - writing your own R function

To write your own function in R,

```
your_function_name <- function(arg1, arg2, arg3){
  # do something using inputs 'arg1', 'arg2', 'arg3'
  sum = arg1 + arg2 + arg3

  # to return sum
  return(sum)

  # or simply
  #sum
}
```

You can also make your R function to return multiple objects at once. The returned object will have sum_all, sum_12, and sum_23, sum12, sum23 and sum variables only exist inside the function definition. To access the returned object with multiple variables, use dollar sign : e.g. (objectname)$sum_all

```
your_function_name <- function(arg1, arg2, arg3){
  # You can also return multiple objects at once
  sum = arg1 + arg2 + arg3
  sum12 = arg1 + arg2; sum23 = arg2 + arg3

  return(list(sum_all = sum, sum_12 = sum12, sum_23 = sum23))
}
```

Common measure of accuracy in a regression problem is mean squared error(MSE). Recall the definition of MSE:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 .$$

We can write a function to compute all of the residuals for the linear regression model (observed - fitted values), square them, and then average those squared values.

```
mse <- function(model, y, data) {
  # model is an lm object (a linear regression)
  # y is the response variable from model
  # data is the dataset we want to use to compute fitted values using our model

  # The predict function computes fitted values when given a model and predictor data
  yhat <- predict(model, data)
  mean((y - yhat)^2)
}
```

Training and Test Error in Linear Regression

Let's calculate training and test error on a linear regression model with `lstat` as a predictor, and `medv` as the response. Recall:

- training mse : fit a model on train data and test the model on train data
- test mse : fit a model on train data and test the model on test data

We can run `lm()` function inside the `mse()` function. It will evaluate `lm()` first and use the returned linear regression model as the input argument for `mse()`.

Notice that we use `trainBoston` to fit the model inside `lm()` for both training and test error.

```
train_mse = mse(lm(medv ~ lstat, data = trainBoston), trainBoston$medv, trainBoston)
test_mse = mse(lm(medv ~ lstat, data = trainBoston), testBoston$medv, testBoston)
train_mse
```

```
## [1] 38.92043
```

```
test_mse
```

```
## [1] 37.63206
```

K Nearest Neighbors

Let's review the regression briefly. Regression is a method to predict a continuous outcome y from predictors $x_1, x_2, \cdots, x_p$. We model the relationship between y and x as $y_i = f(x_i) + \epsilon_i$ The goal of the regression is to learn $f(\cdot)$ from training data in order to do prediction and inference.

- parametric method : assume specific functional form of $f(\cdot)$
- non-parametric method : do not assume a particular functional form of $f(\cdot)$. More flexible.

Linear regression is a parametric method: the goal is to estimate parameters β. Since we assume a linear $f(\cdot)$ between x and y, learning $f(\cdot)$ is equivalent to estimating β.

K nearest neighbors (KNN) is a non-parametric method: For every point in the data, it finds the closest K points (you pick K), and makes predictions by averaging those neighbors. KNN has no parameters unlike β in linear regression. Instead, it has a **tuning parameter**, K. This is a parameter which determines *how* the model is trained, instead of a parameter that is *learned* through training.

Question: Is KNN model more complex with smaller K or bigger K? Why? (Remember that K controls the model complexity!)

Increase model complexity: decrease k

KNN for regression

From 'Boston' data, we'll only use `lstat` as a predictor, and `medv` as the response. We use the function `knn.reg` from the library `FNN` to do regression using KNN.

```r
install.packages("FNN")
```

Load the package. Open the help for `knn.reg` using `?knn.reg`.

```r
library("FNN")
```

We'll do KNN regression for a variety of K using for-loop in R.

We first run KNN regression for training data. Note that we're specifying the argument `test = trainBoston["lstat"]`: this is so that `knn.reg` will give us predictions of the training data so we can get a training MSE.

```r
k_range  = c(1,5,10,25,50,353)
trainMSE = c() #creating null vector

for(i in 1:length(k_range)){
  knnTrain <- knn.reg(train = trainBoston["lstat"], test = trainBoston["lstat"],
y = trainBoston$medv, k = k_range[i])

  trainMSE[i] <- mean((trainBoston$medv - knnTrain$pred)^2)
}
```

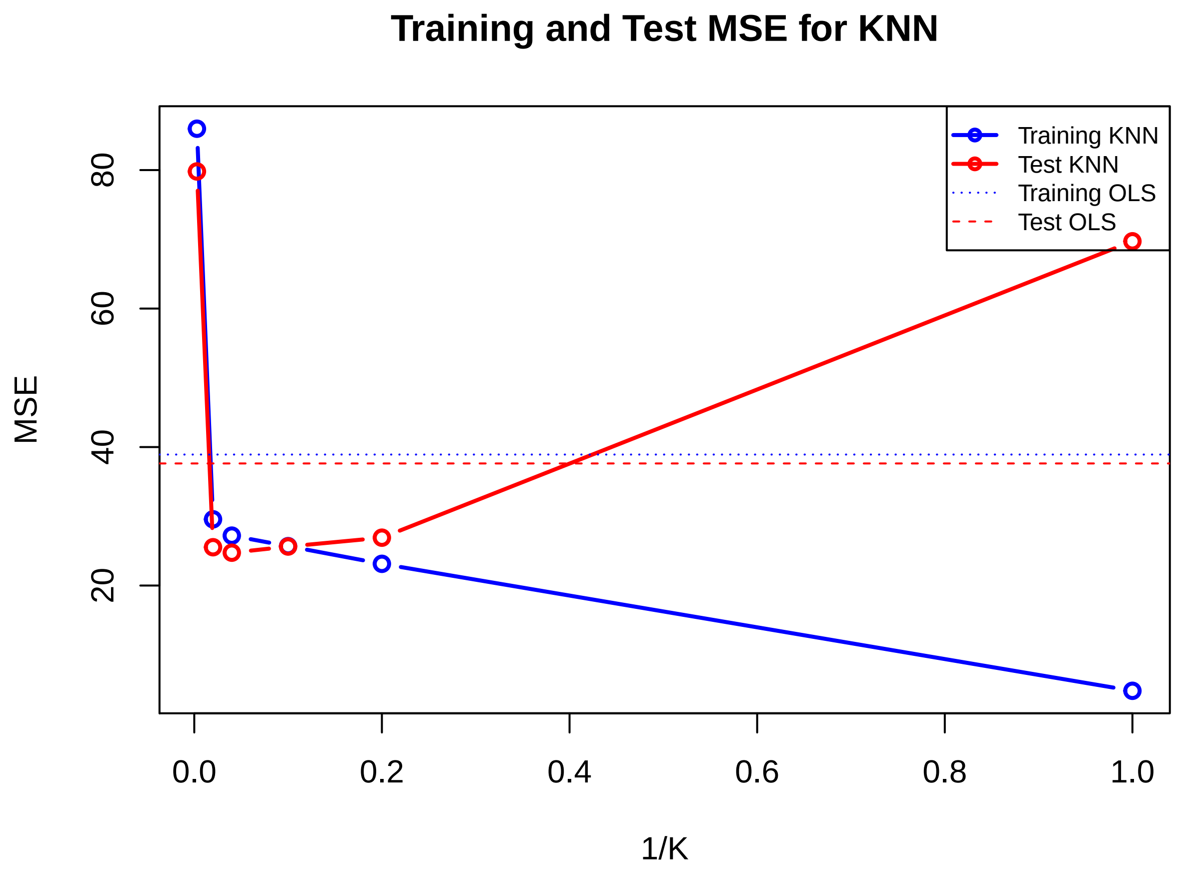
Now let's do the same for the test data (still training using the training set!)

```r
testMSE = c() #creating null vector
```

```
for(i in 1:length(k_range)){
  knnTest <- knn.reg(train = trainBoston["lstat"], test = testBoston["lstat"],
y = trainBoston$medv, k = k_range[i])

  testMSE[i] <- mean((testBoston$medv - knnTest$pred)^2)
}
```

Let's plot training MSEs and test MSEs on the same plot.

```
plot(trainMSE ~ c(1, 1/5, 1/10, 1/25, 1/50, 1/353), type = "b", lwd = 2, col = "blue",
main = "Training and Test MSE for KNN", xlab = "1/K", ylab = "MSE")
# Add the test MSE
lines(testMSE ~ c(1, 1/5, 1/10, 1/25, 1/50, 1/353), type = "b", lwd = 2, col = "red")

# Add the linear regression MSE
abline(a = mse(lm(medv ~ lstat, data = trainBoston), trainBoston$medv, trainBoston),
b = 0, lty = 3, col = "blue")
abline(a = mse(lm(medv ~ lstat, data = trainBoston), testBoston$medv, testBoston),
b = 0, lty = 2, col = "red")

legend("topright", legend = c("Training KNN", "Test KNN", "Training OLS", "Test OLS"),
col = c("blue", "red"), cex = .75,
lwd = c(2, 2, 1, 1), pch = c(1, 1, NA, NA), lty = c(1, 1, 3, 2))
```



Question: What pattern do we observe this graph?

Question: How do we pick K? Can we pick K that minimize the training error?

KNN for Classification

To perform classification with KNN, we use the `knn()` function from the `class` package. Unlike many of our previous methods, `knn()` requires that all predictors be numeric, so we coerce `student` to be a `0` and `1` variable instead of a factor. (We can leave the response as a factor.)

```r
library(class)
```

```
##
## Attaching package: 'class'

## The following objects are masked from 'package:FNN':
##
##     knn, knn.cv
```

```r
set.seed(42)
Default$student = as.numeric(Default$student) - 1
default_index = sample(nrow(Default), 5000)
default_train = Default[default_index, ]
default_test = Default[-default_index, ]
```

Also `knn()` requires the predictors be their own data frame or matrix, and the class labels be a separate factor variable. Note that the y data should be a factor vector, **not** a data frame containing a factor vector.

```r
# training data
X_default_train = default_train[, -1]
y_default_train = default_train$default

# testing data
X_default_test = default_test[, -1]
y_default_test = default_test$default
```

There is very little "training" with k-nearest neighbors. Essentially the only training is to simply remember the inputs. Because of this, we say that k-nearest neighbors is fast at training time. However, at test time, k-nearest neighbors is very slow. For each test case, the method must find the k-nearest neighbors, which is not computationally cheap. (Note that `knn()` uses Euclidean distance.)

The`knn()` function immediately returns estimated classes. So there is no need to use predict() as in linear regression. Here, `knn()` used four arguments:

- `train`, the predictors for the train set.
- `test`, the predictors for the test set. `knn()` will output results for these cases.
- `cl`, the true class labels for the train set.
- `k`, the number of neighbors to consider.

We'll assess how `knn()` works with this data.

```r
actual = y_default_test
predicted = knn(train = X_default_train, test = X_default_test, cl = y_default_train, k = 5)
mean(actual == predicted) #accuracy
```

```
## [1] 0.9684
```

```r
table(predicted, actual) #confusion matrix
```

```
##          actual
## predicted   No  Yes
##       No  4823  146
##       Yes   12   19
```

Often with `knn()` we need to consider the scale of the predictors variables. If one variable contains much larger numbers because of the units or range of the variable, it will dominate other variables in the distance measurements. But this doesn't necessarily mean that it should be such an important variable. It is common in practice to scale the predictors to have 0 mean and unit variance. Be sure to apply the scaling to both the train and test data.

```r
actual = y_default_test
predicted = knn(train = scale(X_default_train), test = scale(X_default_test),
                cl = y_default_train, k = 5)
mean(actual == predicted) #accuracy
```

```
## [1] 0.9722
```

```r
table(predicted, actual) #confusion matrix
```

```
##          actual
## predicted   No  Yes
##       No  4802  106
##       Yes   33   59
```

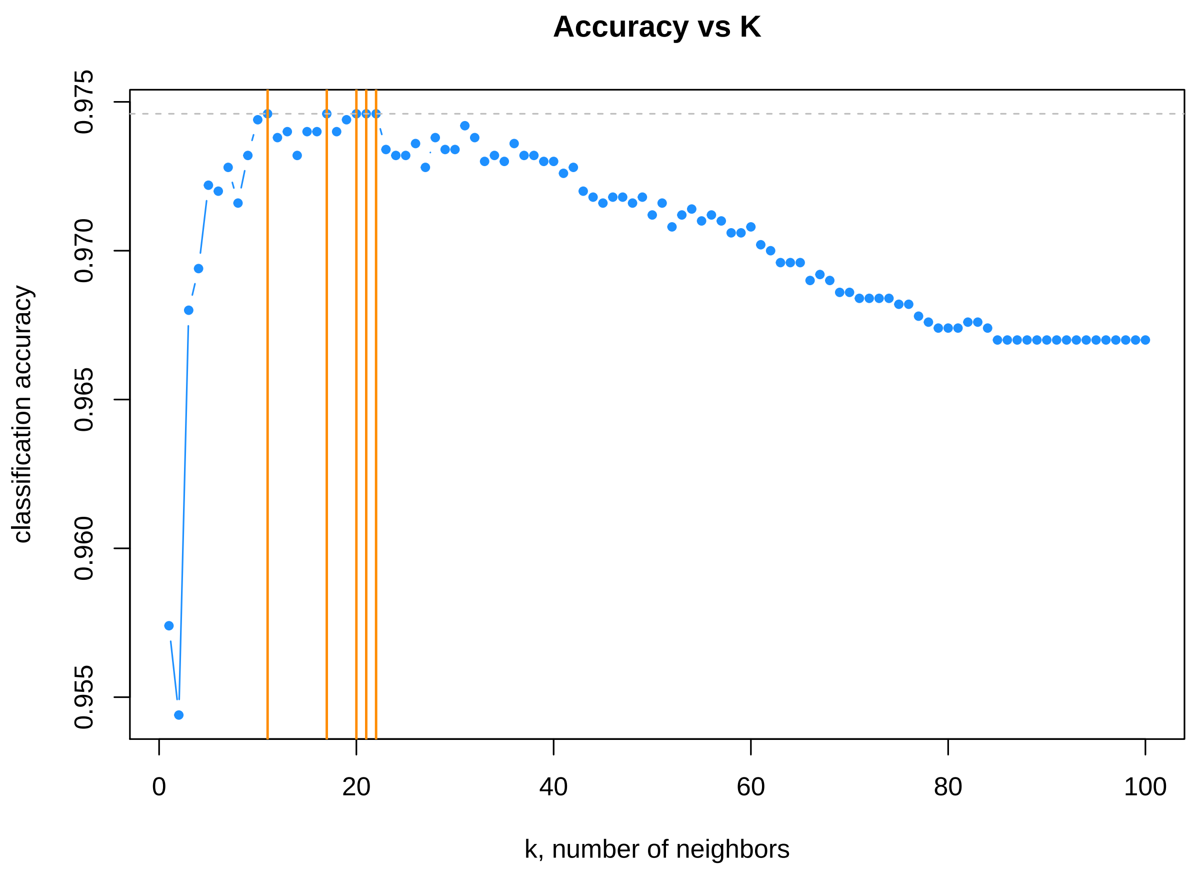
Here we see the scaling improves the classification accuracy. This may not always be the case, and often, it is normal to attempt classification with and without scaling.

How do we choose k? Try different values and see which works best.

```r
set.seed(42)
k_to_try = 1:100
acc_k = rep(x = 0, times = length(k_to_try))

for(i in k_to_try) {
  pred = knn(train = scale(X_default_train),
             test = scale(X_default_test),
             cl = y_default_train,
              k = k_to_try[i])
  acc_k[i] = mean(y_default_test == pred)
}
```

Naturally, we plot the k-nearest neighbor results.

```
# plot accuracy vs choice of k
plot(acc_k, type = "b", col = "dodgerblue", cex = 1, pch = 20,
     xlab = "k, number of neighbors", ylab = "classification accuracy",
     main = "Accuracy vs K")
# add lines indicating k with best accuracy
abline(v = which(acc_k == max(acc_k)), col = "darkorange", lwd = 1.5)
# add line for max accuracy seen
abline(h = max(acc_k), col = "grey", lty = 2)
```



```
max(acc_k)
```

```
## [1] 0.9746
```

```
max(which(acc_k == max(acc_k)))
```

```
## [1] 22
```

We see that five different values of k are tied for the highest accuracy. Given a choice of these five values of k, we select the largest, as it is the least variable, and has the least chance of overfitting.

Classification with multi-class data : Iris Data

KNN can be used for both binary and multi-class problems. As an example, we return to the iris data where the response varaible `Species` has three classes.

```
set.seed(430)
iris_obs = nrow(iris)
iris_index = sample(iris_obs, size = trunc(0.50 * iris_obs))
iris_train = iris[iris_index, ]
iris_test = iris[-iris_index, ]
```

All the predictors here are numeric, so we proceed to splitting the data into predictors and classes.

```
# training data
X_iris_train = iris_train[, -5]
y_iris_train = iris_train$Species

# testing data
X_iris_test = iris_test[, -5]
y_iris_test = iris_test$Species
```

```
iris_pred = knn(train = scale(X_iris_train), test = scale(X_iris_test),cl = y_iris_train,
                k = 10)
```

We look at the confusion matrix that compares the true classes and predicted classes.

```
conf_table = table(iris_pred, y_iris_test)
conf_table
```

```
##              y_iris_test
## iris_pred     setosa versicolor virginica
##   setosa          22          0         0
##   versicolor       0         26         6
##   virginica        0          0        21
```

From the confusion table, We can calculate the test error(classification error) for multi-class data.

```
1 - sum(diag(conf_table))/sum(conf_table)
```

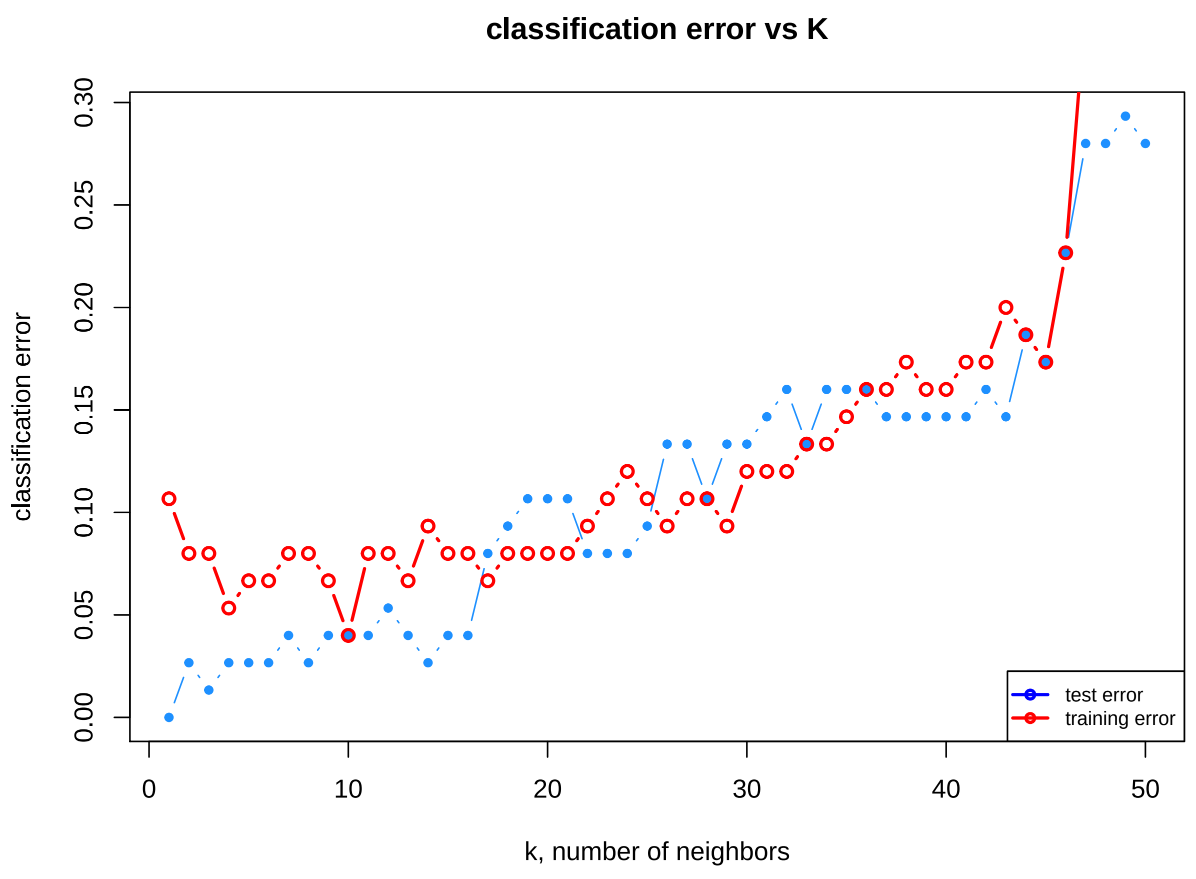
```
## [1] 0.08
```

Now, let's try with different values of K and see how the classification error change with K.

```
k_to_try = 1:50
test_k = rep(x = 0, times = length(k_to_try))
train_k = rep(x = 0, times = length(k_to_try))

for(i in k_to_try) {
  pred1 = knn(train = scale(X_iris_train), test = scale(X_iris_test),
            cl = y_iris_train, k = k_to_try[i])
  test_k[i] = mean(y_iris_test != pred1) #test error

  pred2 = knn(train = scale(X_iris_train), test = scale(X_iris_train),
            cl = y_iris_train, k = k_to_try[i])
  train_k[i] = mean(y_iris_train != pred2) #training error
}
```

We plot the training and test error from k-nearest neighbor results.

```r
# plot classification error vs choice of k
plot(k_to_try, train_k, type = "b", col = "dodgerblue", cex = 1, pch = 20,
     xlab = "k, number of neighbors", ylab = "classification error",
     main = "classification error vs K")
lines(k_to_try, test_k, type = "b", lwd = 2, col = "red")
# add lines indicating k with best accuracy
legend("bottomright", legend = c("test error", "training error"),
col = c("blue", "red"), cex = .75, lwd = c(2, 2), pch = c(1, 1), lty = c(1, 1))
```



Resources and References

1. https://daviddalpiaz.github.io/r4sl/k-nearest-neighbors.html#regression