STAT 542: Homework 2

Spring 2022, by Kamila Makhambetova (kamilam3)

Due: Thursday 11:59 PM CT, Feb 3

Contents

Instruction

Students are encouraged to work together on homework. However, sharing, copying, or providing any part of a homework solution or code is an infraction of the University's rules on Academic Integrity. Any violation will be punished as severely as possible. Final submissions must be uploaded to compass2g. No email or hardcopy will be accepted. For **late submission policy and grading rubrics**, please refer to the course website.

- What is expected for the submission to **Gradescope**
  - You are required to submit one rendered **PDF** file `HWx_yourNetID.pdf`. For example, `HW01_rqzhu.pdf`. Please note that this must be a `.pdf` file generated by a `.Rmd` file. `.html` format cannot be accepted.
  - Please follow the instructions on Gradescope to select corresponding PDF pages for each question.
- Please note that your homework file is a **PDF** report instead of a messy collection of R codes. This report should **include**:
  - Your Name and NetID. (Replace `Ruoqing Zhu(rqzhu)` by your name and NetID if you are using this template).
  - Make all of your `R` code chunks visible for grading.
  - Relevant outputs from your `R` code chunks that support your answers.
  - Provide clear answers or conclusions for each question. For example, you could start with `Answer: I fit SVM with the following choice of tuning parameters ...`
  - Many assignments require your own implementation of algorithms. **Basic comments are strongly encouraged** to explain the logic to our graders. However, line-by-line code comments are unnecessary.
- Requirements regarding the `.Rmd` file.

- You do **NOT** need to submit `Rmd` files. However, your PDF file should be rendered directly from it.
- Make sure that you **set random seeds** for simulation or randomized algorithms so that the results are reproducible. If a specific seed number is not provided in the homework, you can consider using your NetID.
- For some questions, there will be restrictions on what packages/functions you can use. Please read the requirements carefully. As long as the question does not specify such restrictions, you can use anything.

About HW2

For this HW, we mainly try to understand the KNN method in both classification and regression settings and use it to perform several real data examples. Tuning the model will help us understand the bias-variance trade-off. A slightly more challenging task is to code a KNN method yourself. For that question, you cannot use any additional package to assist the calculation.

There is an important package, `ElemStatLearn`, which is the package associated with the ESL textbook for this course. Unfortunately, the package is currently discontinued on CRAN. You can install an earlier version of this package by using

```
require(devtools)
install_version("ElemStatLearn", version = "2015.6.26.2", repos = "http://cran.us.r-project.org")
```

And of course, you will have to install the `devtools` package if you don't already have it.

Question 1 [40 Points] KNN Classification (Diabetes)

Load the Pima Indians Diabetes Database (`PimaIndiansDiabetes`) from the `mlbench` package. If you don't already have this package installed, use the following code. It also randomly splits the data into training and testing. You should preserve this split in the analysis.

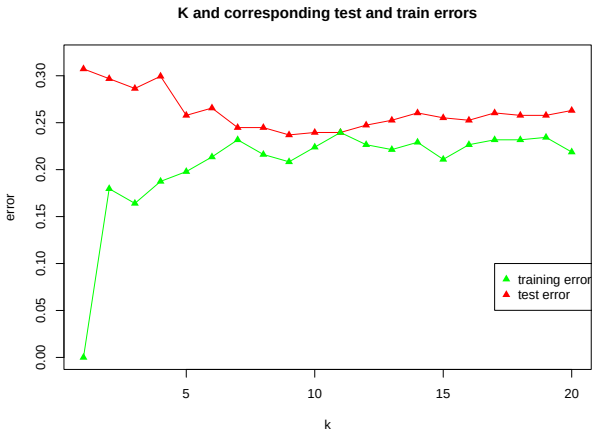
```
#install.packages("mlbench")
library(mlbench)
data(PimaIndiansDiabetes)

set.seed(2)
trainid = sample(1:nrow(PimaIndiansDiabetes), nrow(PimaIndiansDiabetes)/2)
Diab.train = PimaIndiansDiabetes[trainid, ]
Diab.test = PimaIndiansDiabetes[-trainid, ]
```

- **Hints**
  - Read the documentation of this dataset here.
  - Make sure that you understand the goal of this is **classification problem**. Knn algorithms use different prediction strategies for classification and regression.

Use a grid of k values (every integer) from 1 to 20.

a) [10 pts] Fit a KNN model using `Diab.train` and calculate both training and testing errors. For the testing error, use `Diab.test`. Plot the two errors against the corresponding k values. Make sure that you differentiate them using different colors/shapes and add proper legends.

```r
#install.packages("kknn")
#install.packages("class")

library(class)

set.seed(1)
train_error<-numeric(20)
test_error<-numeric(20)

train<-Diab.train[,1:8]
test<-Diab.test[,1:8]


# loop to find train and test errors for k=1,2...,20
for (k in 1:20){

  #Y predicted for train
  knn.fit_train<-knn(train=train,test=train,cl = Diab.train$diabetes,
                     k = k)

   #Y predicted for test
  knn.fit_test<-knn(train=train,test=test,cl = Diab.train$diabetes,
                     k = k)

  #convert all types of Y into numeric data (initially their type is factor).
  fitted_train=as.numeric(knn.fit_train)
  fitted_test=as.numeric(knn.fit_test)

  real_y_train=as.numeric(Diab.train$diabetes)
  real_y_test=as.numeric(Diab.test$diabetes)

  #calculate train and testing errors using proportion.
  train_error[k]=sum(real_y_train != fitted_train)/nrow(Diab.train)
  test_error[k]=sum(real_y_test != fitted_test)/nrow(Diab.test)


}


#Draw graphs
x=1:20

plot(x,test_error, type = "l", ylim=range(c(0,0.32)),xlim=range(c(1,20)), col="red", xlab = "k", ylab =
points(x,test_error, pch=17, col="red")
points(x,train_error, pch=17, col="green")
lines(x,train_error, type = "l", col="green")

title(main="K and corresponding test and train errors")
legend(17, 0.1,legend=c("training error","test error"), col=c("green","red"),
                                    pch=c(17,17), ncol=1)
```




```
#Find minimum test error
min(test_error)
```

```
## [1] 0.2369792
```

```
#Find index of minimum test error
index=match(min(test_error), test_error)
index
```

```
## [1] 9
```

```
#Find the corresponding training error
train_error[index]
```

```
## [1] 0.2083333
```

b) [15 pts] Does the plot match (approximately) our intuition of the bias-variance trade-off in terms of having a U-shaped error? What is the optimal k value based on this result? For the optimal `k`, what is the corresponding degrees-of-freedom and its error?

Answer

I think the plot matches our intuition of the bias-variance trade-off in terms of having a U-shaped error curve. We can see it by looking at test error graph. When K=1 the testing error is relatively high (approximately 0.3) and the training error is 0. It is because when K=1, we need only find 1 closest point from the training set to $x0$ from the test set. Since the test set belongs to training dataset, the closest point to $x0$ is $x0$ itself. That's why the training error = 0 and testing error is high. So we have low bias, but high variance. So the testing error consists of variance mostly.

As K increases bias increases and variance decreases, the difference between bias and variance becomes smaller and the testing error decreases until $K = K_{optimal}$. At $K = K_{optimal}$ variance approximately equals to variance. As $K > K_{optimal}$ testing error begins to increase, as model's bias exceeds variance and it continues to rise and variance continues to drop, so the difference between bias and variance becomes larger.

So the $K_{optimal}$ will be such K that test error is minimum and after $K > K_{optimal}$ the testing error starts to rise.

So $K_{optimal} = 9$ and the corresponding testing error = 0.2369792 and corresponding training error = 0.2083333. $d.f. = \frac{N}{k}$ So d.f. = 384/9=42.67.

c) [15 pts] Suppose we do not have access to `Diab.test` data. Thus, we need to further split the training data into train and validation data to tune `k`. For this question, use the `caret` package to complete the tuning. You are required to

- Train the **knn** model with cross-validation using the `train()` function.
  - Specify the type of cross-validation using the `trainControl()` function. We need to use three-fold cross-validation.
  - Specify a grid of tuning parameters. This can be done using `expand.grid(k = c(1:20))`.
- Report the best parameter with its error. Compare it with your `k` in b).

For details, read either the example from SMLR or the documentation at here to learn how to use the `trainControl()` and `train()` functions. Some examples can also be found at here.

```
library(caret)

#3-fold crossvalidation
train_control <- trainControl(method  = "cv", number = 3)
set.seed(1)

# fit KNN.
fit <- train(diabetes ~ .,
             method     = "knn",
             tuneGrid   = expand.grid(k = c(1:20)),
             trControl  = train_control,
             metric     = "Accuracy",
             data       = Diab.train)
fit
```

```
## k-Nearest Neighbors
##
## 384 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (3 fold)
## Summary of sample sizes: 256, 256, 256
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    1  0.6458333  0.2383891
##    2  0.6588542  0.2493153
##    3  0.7109375  0.3500264
##    4  0.7135417  0.3570747
##    5  0.7213542  0.3625524
##    6  0.7213542  0.3610688
##    7  0.7239583  0.3605471
##    8  0.7213542  0.3740264
##    9  0.7161458  0.3426171
##   10  0.7109375  0.3327837
##   11  0.7187500  0.3306667
##   12  0.7213542  0.3437038
##   13  0.7239583  0.3495692
##   14  0.7213542  0.3457462
```

```
##   15  0.7187500  0.3302849
##   16  0.7083333  0.3107828
##   17  0.7239583  0.3422158
##   18  0.7239583  0.3379770
##   19  0.7265625  0.3412294
##   20  0.7265625  0.3397047
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 20.
```

Answer

The best parameter will be k=20, as it has the highest accuracy (accuracy = 0.7265625). We know that $error = 1 - accuracy = 1 - 0.7265625$. So its corresponding error = 0.2734375. From part (b) I got $K_{optimal} = 9$ and the corresponding testing error = 0.2369792 and corresponding training error = 0.2083333. The optimal K-s in 2 parts are different and the testing error (0.2369792) in part (b) is lower than the testing error (0.2734375) from cross-validation method. But I expected it, because cross validation methods randomly divide the dataset into 3 groups and performs KNN.

Question 2 [40 Points] Write your own KNN for regression

a. [10 pts] Generate $p = 5$ independent standard Normal covariates X_1, X_2, X_3, X_4, X_5 of $n = 1000$ independent observations. Then, generate Y from the regression model

$$Y = X_1 + 0.5 \times X_2 - X_3 + \epsilon,$$

with i.i.d. standard normal error ϵ. Make sure to set a random seed 1 for reproducibility.

- Use a KNN implementation from an existing package. Report the mean squared error (MSE) for your prediction with `k = 5`. Use the first 500 observations as the training data and the rest as testing data. Predict the Y values using your KNN function with `k = 5`. Mean squared error is

$$\frac{1}{N}\sum_i (y_i - \widehat{y}_i)^2$$

. This question also helps you validate your own function in b). a) and b) are expected have similar (possibly not identical) results.
- Hints: this is a **regression** problem instead of a classification one.

```
library(kknn)

#generate 5 predictors from random standard normal distribution
set.seed(1)
X1<-rnorm(n = 1000, mean = 0, sd = 1)
X2<-rnorm(n = 1000, mean = 0, sd = 1)
X3<-rnorm(n = 1000, mean = 0, sd = 1)
X4<-rnorm(n = 1000, mean = 0, sd = 1)
X5<-rnorm(n = 1000, mean = 0, sd = 1)

##generate error from random standard normal distribution and calculate Y
error<-rnorm(n = 1000, mean = 0, sd = 1)

Y<-X1+0.5*X2-X3+error
```

```r
#divide data into training and test datas
data<-data.frame(X1=X1,X2=X2,X3=X3,X4=X4,X5=X5,Y=Y)
data_train=data[1:500, ]
data_test=data[501:1000, ]
Y_test=data[501:1000, 6]

#fit KNN
knn.fit2 <- kknn(Y ~ X1+X2+X3+X4+X5, train = data_train, test = data_test, k=5,  kernel = "rectangular")
test.pred = knn.fit2$fitted.values

#calculate MSE
MSE<-mean((Y_test - test.pred)^2)
print(paste0("MSE for KNN with k=5 : ", MSE))
```

```
## [1] "MSE for KNN with k=5 : 1.38613535548539"
```

Answer

MSE for KNN model with k=5 equals to 1.38613535548539.

b. [30 pts] For this question, you **cannot** use (load) any additional `R` package. Write your own function `myknn(xtrain, ytrain, xtest, k)` that fits a KNN model and predict multiple target points `xtest`. The function should return a variable `ytest`.

- Here, `xtrain` is the training dataset covariate value, `ytrain` is the training data outcome, and `k` is the number of nearest neighbors. `ytest` is the prediction on `xtest`.
- Use Euclidean distance to calculate the closeness between two points.
- Test your code by reporting the mean square error on the testing data.

```r
#Function to calculate Euclidean distance
Eucl_dist <- function(point1, point2){
  dist = 0
  i=1

  while(i<=length(point1)){

     dist = dist + (point1[[i]]-point2[[i]])^2
     i=i+1
  }

  dist = sqrt(dist)
  return(dist)
}

#checking Euclidean distance function
#a=c(3,2,4)
#b=c(7,6,8)
#d=Eucl_dist(a,b)
#print(d)

# My KNN function to calculate predicted Y.
myknn <- function(xtrain, ytrain, xtest, k){
```

```
  #create vector, which will store values from one testing point to all training points
  dist_vect =numeric(nrow(xtrain))
  j=1
  i=1

  #create vector, which stores all predicted Y.
  ytest=numeric(nrow(xtest))

  #create matrix of distances, which will store values from all testing points to all training points
  dist_matrix=matrix(0,nrow(xtest), nrow(xtrain))

  #nested loop to calculate all distance from all training points to all testing points
  while(i<=nrow(xtest)){
    while(j<=nrow(xtrain)){
      #assign distance matrix
      dist_matrix[i,j]=Eucl_dist(xtest[i, ],xtrain[j, ])
      j=j+1
    }

    #sort all calculated distances from one testing point to all training point in increasing order
    sort_dist_vect=sort(dist_matrix[i, ])
    #choose only k smallest distances
    dist_k=sort_dist_vect[1:k]


    index=numeric(k)
    #find corresponding k indexes
    index=match(dist_k, dist_matrix[i, ])

    #calculate sum of k training Ys corresponding to these k indexes
    y_sum=0
    for(l in index){
      y_sum=y_sum+ytrain[l]
    }

    #find average Y. It is our predicted Y.
    ytest[i]=y_sum/k

    i=i+1
    j=1

  }

  return(ytest)

}

#run my Knn algorithm
data_train2=data[1:500, -6]
data_test2=data[501:1000, -6]
Y_train2=data[1:500, 6]
Y_test2=data[501:1000, 6]
```

```
ytest_res=myknn(data_train2, Y_train2, data_test2, 5)

#calculate MSE
MSE2<-mean((Y_test2 - ytest_res)^2)
print(paste0("MSE for my KNN function with k=5 : ", MSE2))
```

```
## [1] "MSE for my KNN function with k=5 : 1.39322652116514"
```

Answer

MSE for my KNN function with k=5 equals to 1.39322652116514. MSE for KNN from package with k=5 equals to 1.38613535548539. So MSE-s from 2 different functions are very close to each other.

Question 3 [30 Points] Curse of Dimensionality

Let's consider a high-dimensional setting. Keep the data-generating model the same as question 2. In addition to the outcomes and covariates from question 2, we will also generate 95 more noisy variables to make p = 100. In this question, you can use a KNN function from any existing package.

We consider two different settings to generate that additional set of 95 covariates. Make sure to set random seeds for reproducibility.

- Generate another 95-dimensional covariates with all independent standard Gaussian entries.
- Generate another 95-dimensional covariates using the formula $X^T A$, where X is the original 5-dimensional vector, and A is a 5×95 dimensional (fixed) matrix that remains the same for all observations. You should generate A just once using i.i.d. uniform $[0, 1]$ entries and then apply A to your current 5-dimensional data.

Fit KNN in both settings (with the total of 100 covariates) and select the best k value. Answer the following questions

```
library(kknn)
set.seed(2)

#Generate another 95-dimensional covariates with all independent standard Gaussian entries

X95_1<-matrix(0, 1000,95)

for(i in 1:95){
  X95_1[, i]=rnorm(n = 1000, mean = 0, sd = 1)
}

#X-1000x5 or 5x1? --> X^T is 1x5 A=5x95


#Generate 95-dimensional covariates using X^T A

A<-matrix(0, 5, 95)

for(i in 1:5){
  A[i, ]=runif(95)
```

```
}

X<-matrix(0, 1000, 5)

for(j in 1:5){
  X[,j ]=data[ , j]
}

#A-5x95 X-1000x5
#Calculate 95 predictors
X95_2=X%*% A


#generate training and testing datasets
data3_1=data.frame(X,X95_1, Y)
data3_2=data.frame(X,X95_2, Y)

data_train31=data3_1[1:500, ]
data_test31=data3_1[501:1000, ]
Y_test31=data3_1[501:1000, 101]

data_train32=data3_2[1:500, ]
data_test32=data3_2[501:1000, ]
Y_test32=data3_2[501:1000, 101]

#Function to calculate MSE
MSE_calc<-function(Y_test, Y_pred){
  MSE3<-mean((Y_test - Y_pred)^2)
  return(MSE3)
}

MSE31<-numeric(100)
MSE32<-numeric(100)


#loop to calculate MSE for 1st and 2nd settings for KNN with k=1,2...,100

for(i in 1:100){

  #predicted Y from KNN with k for 1st setting
  knn.fit31 <- kknn(Y ~ . , train = data_train31, test = data_test31, k=i,  kernel = "rectangular")

  #corresponding MSE
  MSE31[i]=MSE_calc(Y_test31, knn.fit31$fitted.values)

  #predicted Y from KNN with k for 2nd setting
  knn.fit32 <- kknn(Y ~ . , train = data_train32, test = data_test32, k=i,  kernel = "rectangular")

  #corresponding MSE
  MSE32[i]=MSE_calc(Y_test32, knn.fit32$fitted.values)

}
```

```r
#find minimum MSE and corresponding K for 1st setting
min1=min(MSE31)
index1=match(min1, MSE31)

print(paste0("The minimum MSE for 1st method =: ", min1, " and the oprimal K= ", index1))
```

```
## [1] "The minimum MSE for 1st method =: 2.67599075657796 and the oprimal K= 22"
```

```r
#find minimum MSE and corresponding K for 2nd setting
min2=min(MSE32)
index2=match(min2, MSE32)

print(paste0("The minimum MSE for 2nd method =: ", min2, " and the oprimal K= ", index2))
```

```
## [1] "The minimum MSE for 2nd method =: 1.42762023162136 and the oprimal K= 10"
```

a) [10 pts] For the first setting, what is the best k and the best mean squared error for prediction?

Answer

For the first setting, the best K will be such K that corresponding MSE is minimized. The minimum MSE = 2.67599075657796, and the corresponding K = 22. So best MSE = 2.67599075657796 and best K = 22.

b) [10 pts] For the second setting, what is the best k and the best mean squared error for prediction?

Answer

For the second setting, the best MSE = 1.42762023162136 and the best K =10, as MSE = 1.42762023162136 is minimum among all MSEs.

c) [10 pts] In which setting kNN performs better? Why?

Answer

kNN performs better for 2nd setting as the best MSE for 2nd setting = 1.42762023162136 < the best MSE for 1st setting = 2.67599075657796. Also best K=10 for 2nd setting and it is smaller than K=22 for 1st setting. I think it is because in 1st setting we generate 95 predictors from random normal distribution. In 2nd setting we generate 95 predictors using set of 5 previous predictors $X_1, X_2, ..., X_5$. Also, $Y = X_1 + 0.5 * X_2 - X_3 + \epsilon$. So there is some relationship between Y and 95 predictors in 2nd setting, but in 1st setting Y and 95 predictors are independent. That's why $MSE_1 > MSE_2$.