Lab_09T

36-290 – Statistical Research Methodology

Week 9 Tuesday – Fall 2021

Preliminaries

Goal

Today's goal is to learn how to apply K nearest neighbors (KNN) via use of `R`'s `FNN` package, which provides (relatively) convenient functions for determining the optimal number of neighbors k and for generating predictions on a test set.

Questions

Data, Part I

Below we read in the `PHOTO_MORPH` dataset:

```
rm(list=ls())
file.path = "https://raw.githubusercontent.com/pefreeman/36-290/master/EXAMPLE_DATASETS/PHOTO_MORPH/photo_morph.Rdata"
load(url(file.path))
rm(file.path)

predictors = data.frame(scale(predictors))

cat("Sample size: ",length(response),"\n")
```

```
## Sample size:  3419
```

To remind you: if everything loaded correctly, you should see two variables in your global environment: `predictors` and `response`, the former with 16 measurements each for 3,419 galaxies, and the latter being 3,419 spectroscopically determined redshifts (spectroscopic = "effectively no error in the redshift determination," i.e., "we know where the galaxy is with high precision"). For the 16 predictor variables, four represent brightness (one magnitude, three colors), and 12 are morphological statistics, i.e., statistics that encode the galaxy's appearance.

Note that I scaled (i.e., standardized) the predictor data frame, because KNN relies on the Euclidean distance to determine distances to neighbors.

Question 1

Split the data! Carry out a traditional linear regression analysis and compute the test-set MSE. Set it aside.

Then work with `FNN` to derive a value for the same metric. The interface to `FNN`, like that to `xgboost`, is not written in a manner consistent with older, more traditional packages. However, the documentation is a bit more straightforward.

To carry out a KNN regression analysis, you need to do the following:

- Determine the maximum value for the number of nearest neighbors. Call this `k.max`. This should be of order 10-100; realize that if your optimal value of k that you determine is equal to `k.max`, then `k.max` is too small and you'll have to increase it and run the algorithm again.
- Initialize a vector called `mse.k` of length `k.max`.
- Loop over calls `knn.reg()` with training data only and with different values of k. Since the dataset is not that large, use `algorithm="brute"`. In each loop, after the call to `knn.reg()`, save the value of the validation MSE, using the `pred` vector that is embedded in the output from `knn.reg()`, and `resp.train`. The `pred` vector is an output from the cross-validation that `knn.reg()` does internally. Look at the documentation for `knn.reg()` for help.
- The optimal value of k is the one for which `mse.k` achieves its minimum value. Find this value by applying, e.g., `which.min()` to the `mse.k` vector. Again, if the optimal value of k is equal to (or very close to) `k.max`, increase `k.max` and run everything again.
- Using `ggplot()`, plot the validation MSE versus the number of neighbors k. Remember: you'll have to define a data frame, where one column is `1:k.max` and the other column is `mse.k`.
- Call `knn.reg()` again, but this time with both the training and test data, with k set to k_{opt}. Use the output to compute the test-set MSE for KNN.

- Use `ggplot()` to make a regression diagnostic plot.

What is the optimal value of k? Is the test-set MSE substantially smaller for KNN versus linear regression? For these data, which model would you adopt? Would it depend on the needs of the client?

```
#Split the data! Carry out a traditional linear regression analysis and compute the test-set MSE. Set it aside.

set.seed(100)
fraction=.7
sp = sample(nrow(predictors), round(fraction*nrow(predictors)))
pred.train = predictors[sp ,]
pred.test = predictors[-sp ,]

respdf = data.frame(response)
sr = sample(length(response), round(fraction*length(response)))
resp.train = respdf[sr ,]
resp.test = respdf[-sr ,]

model = lm(resp.train~.,data=pred.train)
model
```

```
##
## Call:
## lm(formula = resp.train ~ ., data = pred.train)
##
## Coefficients:
## (Intercept)        mag.i       col.Vi       col.iJ       col.JH          V.G
##    1.287973    -0.001246    -0.014780    -0.025637     0.001899    -0.034443
##       V.M20          V.C       V.size          J.G        J.M20          J.C
##    0.036220    -0.007367     0.039679    -0.108747    -0.082574     0.031386
##      J.size          H.G        H.M20          H.C       H.size
##    0.085145     0.142500     0.064208    -0.031297    -0.105631
```

```
resp.pred.lm = predict(model,newdata=pred.test)

lmtest_MSE = mean((resp.test - resp.pred.lm) ^ 2)
lmtest_MSE
```

```
## [1] 0.8069791
```

```
#Then work with `FNN` to derive a value for the same metric. The interface to `FNN`, like that to `xgboost`, is not written in a manner consistent with older, more traditional packages. However, the documentation is a bit more straightforward.

#To carry out a KNN regression analysis, you need to do the following:

#Determine the maximum value for the number of nearest neighbors. Call this `k.max`. This should be of order 10-100; realize that if your optimal value of $k$ that you determine is equal to `k.max`, then `k.max` is too small and you'll have to increase it and run the algorithm again.

#Initialize a vector called `mse.k` of length `k.max`.

#Loop over calls `knn.reg()` with training data only and with different values of $k$. Since the dataset is not that large, use `algorithm="brute"`. In each loop, after the call to `knn.reg()`, save the value of the validation MSE, using the `pred` vector that is embedded in the output from `knn.reg()`, and `resp.train`. The `pred` vector is an output from the cross-validation that `knn.reg()` does internally. Look at the documentation for `knn.reg()` for help.


library(FNN)

k.max = 50
mse.k = rep(NA, k.max)
for ( kk in 1:k.max ) {
  knn.out = knn.reg(train=pred.train,y=resp.train,k=kk,algorithm="brute")
  mse.k[kk] = mean((knn.out$pred-resp.train)^2)
}
k.min = which.min(mse.k)
cat("The optimal number of nearest neighbors is ",k.min,"\n")
```

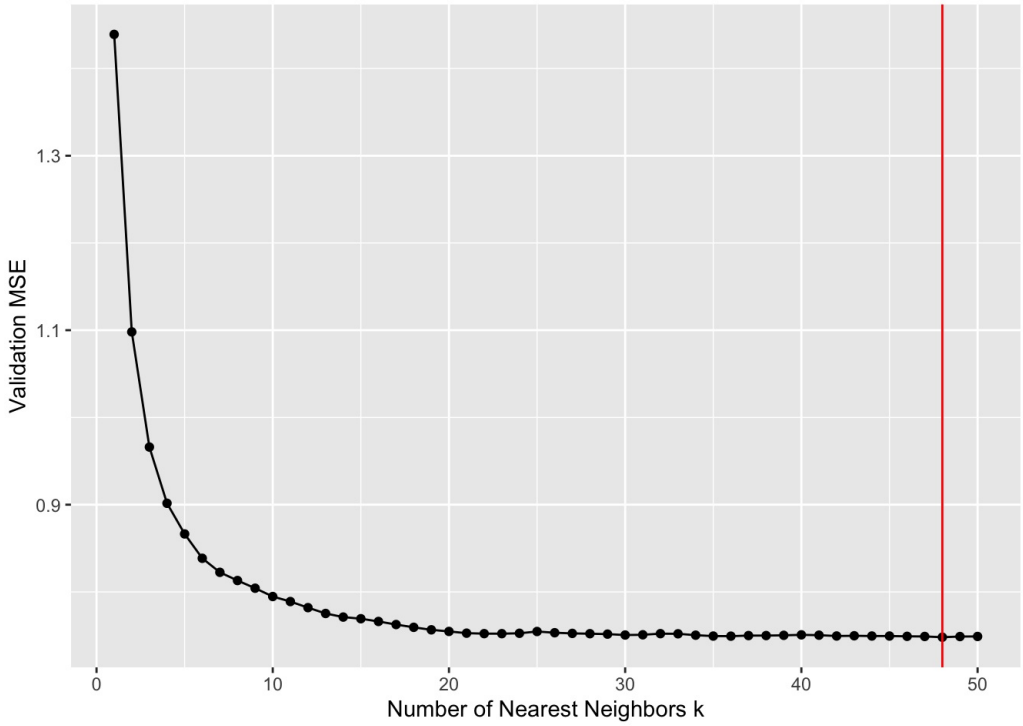
```
## The optimal number of nearest neighbors is  48
```

```
#The optimal value of $k$ is the one for which `mse.k` achieves its minimum value. Find this value by applying, e.g
., `which.min()` to the `mse.k` vector. Again, if the optimal value of $k$ is equal to (or very close to) `k.max`,
increase `k.max` and run everything again.


#optimal number of nearest neighbors is 6


#Using `ggplot()`, plot the validation MSE versus the number of neighbors $k$. Remember: you'll have to define a da
ta frame, where one column is `1:k.max` and the other column is `mse.k`.

suppressMessages(library(tidyverse))
ggplot(data=data.frame("k"=1:k.max,"mse"=mse.k),mapping=aes(x=k,y=mse)) +
  geom_point() + geom_line() +
  xlab("Number of Nearest Neighbors k") + ylab("Validation MSE") +
  geom_vline(xintercept=k.min,color="red")
```



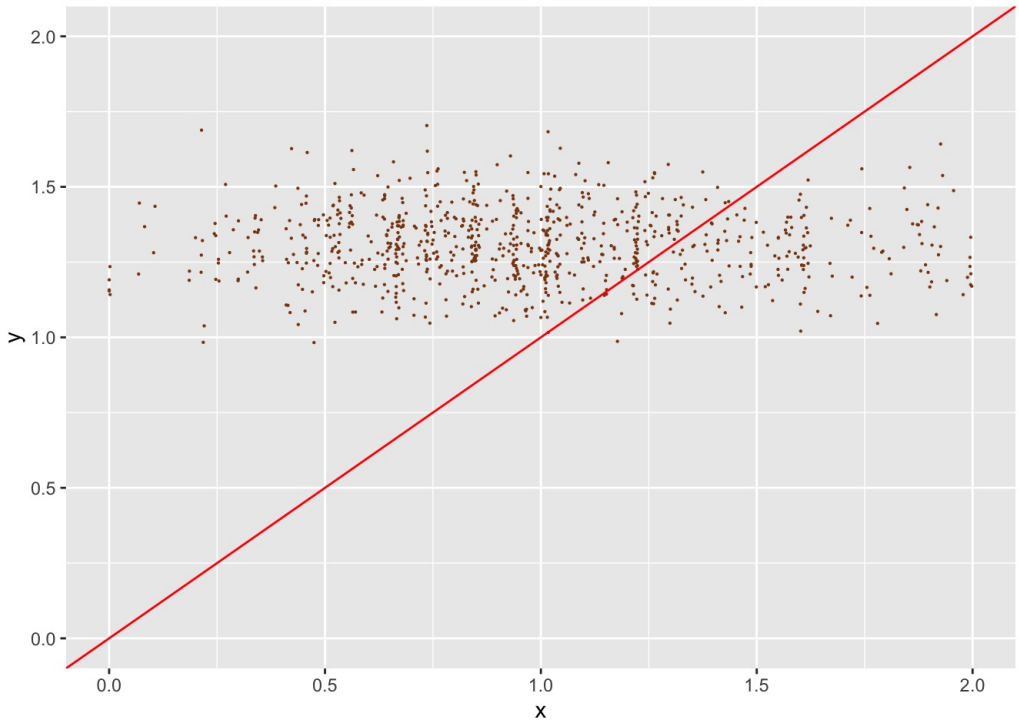

```
#Call `knn.reg()` again, but this time with both the training and test data, with $k$ set to $k_{\rm opt}$. Use the
output to compute the test-set MSE for KNN.

knn.out = knn.reg(train=pred.train,test=pred.test,y=resp.train,k=k.min,algorithm="brute")
(knn.mse = mean((knn.out$pred-resp.test)^2))
```

```
## [1] 0.8105224
```

```
#Use `ggplot()` to make a regression diagnostic plot.
ggplot(data=data.frame("x"=resp.test,"y"=knn.out$pred),mapping=aes(x=x,y=y)) +
  geom_point(size=0.1,color="saddlebrown") + xlim(0,2) + ylim(0,2) +
  geom_abline(intercept=0,slope=1,color="red")
```

```
## Warning: Removed 202 rows containing missing values (geom_point).
```




```
The optimal number of nearest neighbors is k=48.

MSE for linear regression: 0.8069791, MSE for knn:0.8105224

The test set MSE is not substantially smaller for KNN verses linear regression. I would maybe adopt linear regressi
on because the test MSE of KNN is slightly higher and linear regression is better from a interpretability standpoin
t.
```

Now we turn our attention to classification.

Data, Part II

Below we read in a dataset from the UCI Machine Learning Repository. Because it exists elsewhere, I haven't put it on `GitHub`. If you want to see the raw data and documentation, search for the `HTRU2` dataset.

```
rm(list=ls())
file.path = "http://www.stat.cmu.edu/~pfreeman/pulsar.Rdata"
load(url(file.path))
rm(file.path)
set.seed(406)
w.0 = which(response$X9==0)
w.1 = which(response$X9==1)
s = sample(length(w.0),length(w.1))
predictors = predictors[c(w.0[s],w.1),]
response = as.character(response$X9)[c(w.0[s],w.1)]
predictors = scale(predictors)
predictors = data.frame(predictors)
cat("Number of predictor variables: ",ncol(predictors),"\n")
```

```
## Number of predictor variables:  8
```

```
cat("Sample size:                   ",nrow(predictors),"\n")
```

```
## Sample size:                    3278
```

```
response = factor(response,labels=c("NO","YES"))
response = data.frame(response)
```

Note again that I scaled the predictor data frame!

The eight predictors are summary statistics that describe the distribution of brightness measurements of a pulsar candidate (mean, standard deviation, skewness, kurtosis) as well as the distribution of "dispersion measure" readings (also mean, standard deviation, skewness, kurtosis).

The response is either "NO" (the candidate is *not* a pulsar) or "YES".

In order to increase computation speed and to make the results "more interesting," I downsampled the data so as to have 1639 "NO" values and 1639 "YES" values. (There were 1639 "YES" values in the original dataset.)

What's a pulsar, you ask?

When less massive stars die, they slough off their gaseous outer layers and expose an Earth-sized remnant that is kept from collapsing under gravity by the so-called degeneracy pressure of electrons. Basically, the Pauli Exclusion principle is why these remnants don't collapse to, e.g., black holes. These are *white dwarfs*.

When stars that are eight times the mass of the Sun or more massive die, they don't just slowly slough off their gas; they "explode" as supernovae. The remnants left behind are too massive to be held up by electon degeneracy pressure…rather, they collapse to bodies about the size of Pittsburgh that are held up by the degeneracy pressure of neutrons. These are *neutron stars*.

Often times, neutron stars spin. Fast. They also often have really strong magnetic fields that act to "focus" the light they give off into beams. So sometimes, if everything is arranged just so, the neutron star beam of light will hit the Earth, then the star will go dark, then the beam will come around again and hit the Earth. We see pulses of light. Hence…pulsars.

In the end, a pulsar is a Pittsburgh-sized ball of neutrons that whacks us with a radiation beam, perhaps up to tens or hundreds of times per second. You wouldn't want to live on one, given that the surface gravity is so large you'd be literally instantly crushed into something microscopic. Don't say you weren't warned.

Question 2

You know the drill: split the data, learn a logistic regression model as a baseline, then learn an KNN model and output the test-set MCR values and the confusion matrices for both. (Assume a decision threshold of 0.5. This is not optimal but it is easier to code KNN with this default than otherwise.) Also, like above, use `ggplot()` to display the validation-set MCR versus the number of neighbors. I would have you also create a ROC curve, but the way `FNN` is coded currently makes this relatively difficult to do. Comment on the difference between the MCRs for both models. Also, look at the confusion matrices and comment on where KNN does better than logistic regression. As in Q1, would your conclusion about which model to adopt depend on the needs of the client?

Note that here, instead of using `knn.reg()`, you would use `knn.cv()` to determine the optimum value of k, and `knn()` to generate predictions on the test set. Note the `cl` argument for both: this is where `resp.train` goes (since `cl` means "class"). Use the misclassification rate to as your metric for each k value.

Try setting `k.max` to 20.

```
#split data, logistic regression model learned on split data with test-set assessment

set.seed(100)
fraction=.7
sp = sample(nrow(predictors), round(fraction*nrow(predictors)))
pred.train = predictors[sp ,]
pred.test = predictors[-sp ,]
resp.train = response[sp ,]
resp.test = response[-sp ,]

#logistic regression model using training data
glm.fit = glm(resp.train~.,data=pred.train,family="binomial")
summary(glm.fit)
```

```
##
## Call:
## glm(formula = resp.train ~ ., family = "binomial", data = pred.train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -4.1309  -0.3041  -0.0341   0.0252   3.0302
##
## Coefficients:
##                Estimate Std. Error z value Pr(>|z|)
## (Intercept)      1.9772     0.6588   3.001 0.002690 **
## profile.mean     0.7523     0.3943   1.908 0.056374 .
## profile.stddev  -0.2551     0.1872  -1.363 0.172957
## profile.skew    12.8567     1.2340  10.419  < 2e-16 ***
## profile.kurt    -7.2510     1.9033  -3.810 0.000139 ***
## dm.mean         -0.9195     0.2444  -3.762 0.000168 ***
## dm.stddev        1.3820     0.3421   4.040 5.35e-05 ***
## dm.skew          0.6165     0.6739   0.915 0.360233
## dm.kurt         -0.6918     0.4525  -1.529 0.126361
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 3181.39  on 2294  degrees of freedom
## Residual deviance:  768.69  on 2286  degrees of freedom
## AIC: 786.69
##
## Number of Fisher Scoring iterations: 11
```

```
predicted = predict(glm.fit, pred.test, type="response")

glm.pred=rep("NO", length(response))
glm.pred[predicted >.5]="YES"
tab = table(glm.pred,resp.test)
tab
```

```
##         resp.test
## glm.pred  NO YES
##      NO    1   0
##      YES   7 461
```

```
(462)/(462+7)
```

```
## [1] 0.9850746
```

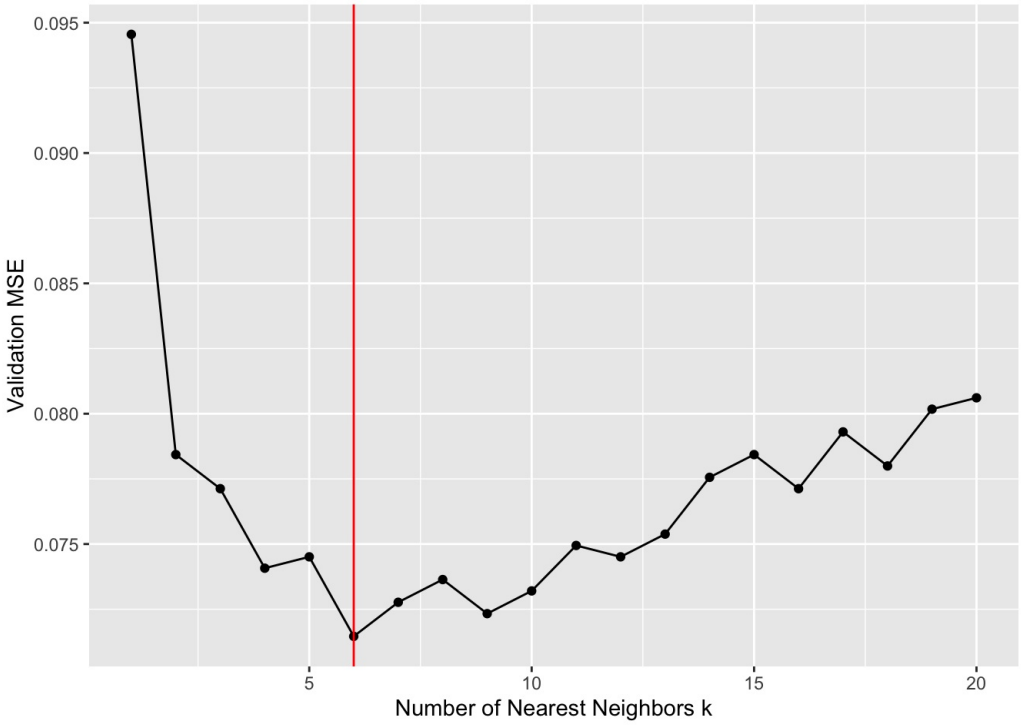
```
#Note that here, instead of using `knn.reg()`, you would use `knn.cv()` to determine the optimum value of $k$, and
`knn()` to generate predictions on the test set. Note the `cl` argument for both: this is where `resp.train` goes (
since `cl` means "class").

library(FNN)
k.max = 20
mse.k = rep(NA,k.max)
for ( kk in 1:k.max ) {
  knn.out = knn.cv(train=pred.train,cl=resp.train,k=kk,algorithm="brute")
  mse.k[kk] = mean(knn.out != resp.train)
}
k.min = which.min(mse.k)
cat("The optimal number of nearest neighbors is ",k.min,"\n")
```

```
## The optimal number of nearest neighbors is  6
```

```
#optimal number of nearest neighbors is 6


#use `ggplot()` to display the validation-set MCR versus the number of neighbors.
suppressMessages(library(tidyverse))
ggplot(data=data.frame("k"=1:k.max,"mse"=mse.k),mapping=aes(x=k,y=mse)) +
  geom_point() + geom_line() +
  xlab("Number of Nearest Neighbors k") + ylab("Validation MSE") +
  geom_vline(xintercept=k.min,color="red")
```




```
#Call `knn.cv()` again, but this time with both the training and test data, with $k$ set to $k_{\rm opt}$. Use the output to compute the test-set MSE for KNN.

knn.out = knn(train=pred.train,test=pred.test,cl=resp.train,k=k.min,prob=TRUE, algorithm="brute")
(knn.mse = mean(knn.out != resp.test))
```

```
## [1] 0.06103764
```

```
tab = table(knn.out, resp.test)
tab
```

```
##        resp.test
## knn.out  NO YES
##     NO  473  51
##     YES   9 450
```

```
(473+450)/(473+450+51+9)
```

```
## [1] 0.9389624
```

```
1-0.9389624
```

```
## [1] 0.0610376
```

```
The MCR for logistic regression is 1.49% and 6.10% for KNN.  KNN does better than logistic regression at identifyin
g true positives. I think like in question 1, the conclusion about which model to adopt does somewhat depend on the
needs of the client, but logistic regression has a lower MCR.



0.9850746
```

```
i=1                          # declaration to initiate for loop
k.optm=1                     # declaration to initiate for loop
for (i in 1:28){
    knn.mod <-  knn(train=train.gc, test=test.gc, cl=train.gc_labels, k=i)
    k.optm[i] <- 100 * sum(test.gc_labels == knn.mod)/NROW(test.gc_labels)
    k=i
    cat(k,'=',k.optm[i],'\n')       # to print % accuracy
}
```