Douglas Rowe

Final Project

M5364–Data Mining   Fall 2016   Tarleton State Univ   Dr. Scott Cook   Assigned 2016-09-30   Due 2016-12-10

This data set is called the abalone data set. Abalone's are a form of sea snails whose age can be determined by boring into the shell and, through a "boring and time consuming" process, by counting the rings much like a tree. In order to save our conchologist friends time, we will attempt to predict the number of rings in the shell by measuring:

1) Sex
2) Length
3) Diameter
4) Height
5) Whole Weight
6) Shucked Weight
7) Viscera weight
8) Shell weight

Preprocessing:

```
data = read.csv("C:/Users/Rowe/Documents/My Repo/Final Project DM/abalone.data.txt",
    header = FALSE)
# We have enough data from the UCI website to know that none of the data
# points are missing values or Nan's.
table(data[, 1])
```

```

   F    I    M
1307 1342 1528
```

```
# Because there is almost even number of 'M', 'F' and 'I' values here, there
# is minimal chance of a class imbalance problem coming from this variable.
my_shapiro(data[, 2:8])
```

```
[1] 7.442090e-29 1.648335e-28 1.181265e-47 1.013778e-27 9.340986e-32
[6] 1.777103e-29 1.565014e-28
```

```
# From this function, we can confidently say that none of the continuous
# variables are normal.
```

Beginning:

```
# This function calculates a 1x2 matrix whose values are the Root Mean
# Square Error (RMSE) and the Mean Absolute Error (MAE) respectively.
RMSE_MAE = function(true, predicted) {
    error_matrix = matrix(nrow = 1, ncol = 2)
    error_matrix[1, 1] = sqrt((1/length(true)) * sum((predicted - true)^2))
    error_matrix[1, 2] = (1/length(true)) * sum(abs(true - predicted))
    return(error_matrix)
}
# I am creating master train and test set so we can compare the acccuracies
# of the models directly
set.seed(5364)
split_data = splitdata(data, 0.7)
train = split_data$train
test = split_data$test
train_rows = split_data$train_rows
```

Artificial Neural Networks:

```
Error_matrix = matrix(ncol = 2, nrow = 15)
for (i in 1:15) {
    crude_ann = nnet(V9 ~ ., data = train, size = as.numeric(i), maxit = 1000,
        linout = TRUE, trace = FALSE)
    predicted_ann = predict(crude_ann, newdata = test)
    Error_matrix[i, ] = RMSE_MAE(test[, 9], predicted_ann)
}
Error_matrix[argmin(Error_matrix[, 1]), ]
```

```
[1] 2.087510 1.464726
```

```
Error_matrix[argmin(Error_matrix[, 2]), ]
```

```
[1] 2.087510 1.464726
```

```
RMSE_size = argmin(Error_matrix[, 1])
MAE_size = argmin(Error_matrix[, 2])
ann_cv = function(size, d, m, data_frame) {
    error_matrix = matrix(nrow = m + 1, ncol = 2)
    for (i in 1:m) {
        d_records = sample(nrow(data_frame), d)
        temp_train = data_frame[d_records, ]
        temp_test = data_frame[-d_records, ]
        model = nnet(V9 ~ ., data = temp_train, size = size, maxit = 1000, linout = TRUE,
            trace = FALSE)
        predicted = predict(model, temp_test)
        error_matrix[i, ] = RMSE_MAE(predicted, temp_test[, 9])
    }
    error_matrix[m + 1, 1] = mean(error_matrix[1:m, 1])
    error_matrix[m + 1, 2] = mean(error_matrix[1:m, 2])
    return(error_matrix)
}
ann_cv(RMSE_size, round(0.7 * nrow(data)), 25, data)
```

```
          [,1]     [,2]
 [1,] 2.095105 1.485442
 [2,] 2.207351 1.541343
 [3,] 2.039443 1.449004
 [4,] 2.065539 1.448992
 [5,] 2.064905 1.466190
 [6,] 2.002281 1.447325
 [7,] 2.123396 1.480538
 [8,] 2.070027 1.465895
 [9,] 2.086959 1.464638
[10,] 2.070136 1.474192
[11,] 2.005938 1.410567
[12,] 2.150864 1.485463
[13,] 2.177452 1.523892
[14,] 2.172037 1.501955
[15,] 1.966703 1.418677
[16,] 2.111945 1.465941
[17,] 1.962255 1.428519
[18,] 2.083109 1.427756
[19,] 1.980216 1.427262
[20,] 2.082509 1.491201
[21,] 2.059137 1.480417
[22,] 2.068889 1.454277
[23,] 2.017523 1.434391
```

```
[24,] 2.146891 1.493057
[25,] 2.134369 1.493402
[26,] 2.077799 1.466413
```

```
ann_cv(MAE_size, round(0.7 * nrow(data)), 25, data)
```

```
            [,1]     [,2]
 [1,]   2.829343 1.497518
 [2,] 165.295623 9.595944
 [3,]   2.155146 1.506562
 [4,]   2.038264 1.465907
 [5,]   2.036226 1.422680
 [6,]   2.074686 1.468382
 [7,]   2.136545 1.486556
 [8,]   2.132773 1.493238
 [9,]   2.097541 1.504436
[10,]   2.083242 1.471270
[11,]   2.155716 1.520520
[12,]   2.111108 1.493397
[13,]   2.071499 1.478480
[14,]   2.046546 1.438408
[15,]   2.038515 1.423895
[16,]   1.974339 1.396308
[17,]   2.105372 1.502165
[18,]   2.082981 1.478904
[19,]   2.127992 1.514478
[20,]   2.071086 1.455224
[21,]   2.243506 1.582867
[22,]   2.017569 1.465865
[23,]   2.208430 1.496113
[24,]   2.159128 1.542066
[25,]   2.192894 1.563071
[26,]   8.659443 1.810570
```

```
ann_n_fold_cross = function(n, data_frame, size) {
    chopping_point = floor(nrow(data_frame)/n)
    stopping_points = c(1/chopping_point, 1:n) * chopping_point
    error_matrix = matrix(ncol = 2, nrow = n + 1)
    for (i in 1:n) {
        lower_bound = stopping_points[i]
        upper_bound = stopping_points[i + 1]
        temp_test = data_frame[lower_bound:upper_bound, ]
        temp_train = data_frame[-(lower_bound:upper_bound), ]
        model = nnet(V9 ~ ., data = temp_train, size = size, maxit = 1000, linout = TRUE,
            trace = FALSE)
        predicted = predict(model, temp_test)
        error_matrix[i, ] = RMSE_MAE(predicted, temp_test[, 9])
    }
    error_matrix[n + 1, 1] = mean(error_matrix[1:n, 1])
    error_matrix[n + 1, 2] = mean(error_matrix[1:n, 2])
    return(error_matrix)
}
ann_n_fold_cross(10, data, RMSE_size)
```

```
         [,1]     [,2]
 [1,] 2.648594 1.955683
 [2,] 3.260283 2.327803
 [3,] 1.612628 1.217336
 [4,] 2.050177 1.595570
```

```
 [5,] 1.587534 1.187856
 [6,] 2.845384 2.046782
 [7,] 1.385622 1.054785
 [8,] 2.188472 1.620859
 [9,] 1.639786 1.196326
[10,] 1.957278 1.408840
[11,] 2.117576 1.561184
```

```
ann_n_fold_cross(10, data, MAE_size)
```

```
          [,1]     [,2]
 [1,] 2.587194 1.911639
 [2,] 3.179929 2.274739
 [3,] 1.722674 1.349745
 [4,] 2.050171 1.595562
 [5,] 1.583929 1.190128
 [6,] 2.856398 2.044390
 [7,] 1.349745 1.027931
 [8,] 3.225724 2.373436
 [9,] 1.656324 1.219490
[10,] 1.959487 1.417368
[11,] 2.217157 1.640443
```

Decision Trees:

```
# First, we're going to create a decision tree to find a good guess as how
# accurate we can get it.
first_tree = rpart(V9 ~ ., data = train)
first_predicted = predict(first_tree, newdata = test)
RMSE_MAE(first_predicted, test[, 9])
```

```
         [,1]     [,2]
[1,] 2.452406 1.754041
```

```
# Now that we have an estimate for accuracy, we're going to use randomForest
# to get a more accurate model using the same test and train data as the
# decision tree did. Note: like all tune functions, it does include a
# 10-fold cross validation test of accuracy.
forestmodel = tune.randomForest(V9 ~ ., data = train)
forestmodel
```

```

Error estimation of 'randomForest' using 10-fold cross validation: 4.709902
```

```
randomforest_cv = function(d, m, data_frame) {
    error_matrix = matrix(nrow = m + 1, ncol = 2)
    for (i in 1:m) {
        d_records = sample(nrow(data_frame), d)
        temp_train = data_frame[d_records, ]
        temp_test = data_frame[-d_records, ]
        model = randomForest(V9 ~ ., data = temp_train)
        predicted = predict(model, temp_test)
        error_matrix[i, ] = RMSE_MAE(predicted, temp_test[, 9])
    }
    error_matrix[m + 1, 1] = mean(error_matrix[1:m, 1])
    error_matrix[m + 1, 2] = mean(error_matrix[1:m, 2])
    return(error_matrix)
}
randomforest_cv(round(0.7 * nrow(data)), 25, data)
```

```
          [,1]     [,2]
 [1,] 2.189761 1.559031
 [2,] 2.032656 1.451119
 [3,] 2.147769 1.502647
 [4,] 2.160276 1.517783
 [5,] 2.155134 1.504519
 [6,] 2.237676 1.545699
 [7,] 2.156094 1.521302
 [8,] 2.065127 1.496686
 [9,] 2.080927 1.502607
[10,] 2.097192 1.522242
[11,] 2.146814 1.506259
[12,] 2.029483 1.470464
[13,] 2.089718 1.517656
[14,] 2.149859 1.547441
[15,] 2.225827 1.560536
[16,] 2.151263 1.543244
[17,] 2.139978 1.500482
[18,] 2.183016 1.509353
[19,] 2.049982 1.459659
[20,] 2.161172 1.491655
[21,] 2.316180 1.586989
[22,] 2.089543 1.470437
[23,] 1.925467 1.424065
[24,] 2.140897 1.521362
[25,] 2.250510 1.603325
[26,] 2.134893 1.513462
```

Support Vector Machines:

```
# Here I'm making a for loop that runs the tune function five times to find
# a good guess at the best parameters.
best_parameters = matrix(ncol = 2, nrow = 5)
obj <- tune.svm(V9 ~ ., data = train, gamma = 2^(0:4), cost = 2^(0:4))
best_parameters[1, ] = as.matrix(obj$best.parameters)
for (i in 1:4) {
    gamma = as.numeric(obj$best.parameters)[1]
    cost = as.numeric(obj$best.parameters)[2]
    obj <- tune.svm(V9 ~ ., data = train, gamma = (0.25 * gamma):(1.75 * gamma),
        cost = (0.5 * cost):(1.5 * cost))
    best_parameters[i + 1, ] = as.matrix(obj$best.parameters)
}
best_model = obj$best.model
svm_cv = function(d, m, data_frame) {
    error_matrix = matrix(nrow = m + 1, ncol = 2)
    for (i in 1:m) {
        d_records = sample(nrow(data_frame), d)
        temp_train = data_frame[d_records, ]
        temp_test = data_frame[-d_records, ]
        model = svm(V9 ~ ., data = temp_train, gamma = best_parameters[5, 1],
            cost = best_parameters[5, 2])
        predicted = as.numeric(predict(model, temp_test))
        error_matrix[i, ] = RMSE_MAE(predicted, temp_test[, 9])
    }
    error_matrix[m + 1, 1] = mean(error_matrix[1:m, 1])
    error_matrix[m + 1, 2] = mean(error_matrix[1:m, 2])
    return(error_matrix)
}
svm_cv(round(0.7 * nrow(data)), 25, data)
```

```
          [,1]     [,2]
 [1,] 2.290255 1.554808
 [2,] 2.192552 1.505933
 [3,] 2.186716 1.510708
 [4,] 2.179284 1.517026
 [5,] 2.182027 1.498752
 [6,] 2.219791 1.533720
 [7,] 2.357776 1.592112
 [8,] 2.323458 1.560145
 [9,] 2.254139 1.545554
[10,] 2.300648 1.561159
[11,] 2.265284 1.539249
[12,] 2.299996 1.529469
[13,] 2.180669 1.470896
[14,] 2.240785 1.510838
[15,] 2.263901 1.561225
[16,] 2.268788 1.529638
[17,] 2.271419 1.541745
[18,] 2.218457 1.482707
[19,] 2.240139 1.545508
[20,] 2.333338 1.609474
[21,] 2.234419 1.524602
[22,] 2.338490 1.618267
[23,] 2.205766 1.516662
[24,] 2.337352 1.556476
[25,] 2.135988 1.479137
[26,] 2.252857 1.535832
```

```r
obj
```

```


Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
      gamma   cost
 0.00390625 1.9375

- best performance: 5.145983
```

Naive Bayes:

```r
nrings = data[, 9]
rings = as.factor(nrings)
sex = data[, 1]
nb_data = data[, -1]
nb_train = train[, -1]
nb_test = test[, -1]
Error_matrix = matrix(nrow = 14, ncol = 2)
# Since the Shapiro-Wilks test suggests that all of the numeric variables
# are nonnormal, I have to find out the best way to turn them into discrete
# factors using a for loop. Thus, I am keeping the train and test set
# constant and only changing number of levels for the newly-factorized
# variables.
colnames(Error_matrix) = c("RMSE", "MAE")
for (i in 2:15) {
    discretized_train = discretizer(nb_train[, -8], i)
    discretized_train = cbind(sex[train_rows], discretized_train)
```

```
    discretized_test = discretizer(nb_test[, -8], i)
    discretized_test = cbind(sex[-train_rows], discretized_test)
    model = naiveBayes(rings[train_rows] ~ ., data = discretized_train)
    predicted_naive_Bayes = predict(model, test)
    predicted_naive_Bayes = as.numeric(predicted_naive_Bayes)
    Error_matrix[i - 1, ] = RMSE_MAE(predicted_naive_Bayes, test[, 9])
}
Error_matrix[argmin(Error_matrix[, 1]), ]
```

```
    RMSE      MAE
3.547534 2.709497
```

```
Error_matrix[argmin(Error_matrix[, 2]), ]
```

```
    RMSE      MAE
3.662323 2.604948
```

```
# We can see that the factor levels that minimizes the Root Mean Square
# error and the Mean Absolute error are very close but different and both of
# them don't change all that much when switching between the two. So, we
# could confidently pick either one and would recieve similar results. I am
# now going to use 10-fold cross-validation to double check my numbers.
data_RMSE = discretizer(nb_data[, -8], argmin(Error_matrix[, 1]))
data_RMSE = cbind(sex, data_RMSE)
data_MAE = discretizer(nb_data[, -8], argmin(Error_matrix[, 2]))
data_MAE = cbind(sex, data_MAE)
data = cbind(sex, data)
naiveBayes_cv = function(d, m, data_frame) {
    error_matrix = matrix(nrow = m + 1, ncol = 2)
    for (i in 1:m) {
        d_records = sample(nrow(data_frame), d)
        temp_test = data_frame[d_records, ]
        temp_train = data_frame[-d_records, ]
        model = naiveBayes(rings[-d_records] ~ ., data = temp_train)
        predicted = as.numeric(predict(model, temp_test))
        error_matrix[i, ] = RMSE_MAE(predicted, nrings[d_records])
    }
    error_matrix[m + 1, 1] = mean(error_matrix[1:m, 1])
    error_matrix[m + 1, 2] = mean(error_matrix[1:m, 2])
    return(error_matrix)
}
naiveBayes_cv(round(0.7 * nrow(data_RMSE)), 25, data_RMSE)
```

```
          [,1]     [,2]
 [1,] 3.060773 2.007182
 [2,] 3.401939 2.136115
 [3,] 3.450897 2.155609
 [4,] 3.254482 2.045828
 [5,] 3.449410 2.161081
 [6,] 3.033838 1.975718
 [7,] 3.504297 2.223324
 [8,] 3.248224 2.117305
 [9,] 3.370275 2.136457
[10,] 3.421436 2.158345
[11,] 2.901837 1.914501
[12,] 3.318583 2.152531
[13,] 3.087362 2.022914
[14,] 2.921160 1.927839
[15,] 3.483693 2.117647
```

```
[16,] 3.427777 2.121067
[17,] 3.236569 2.064295
[18,] 2.952253 1.929207
[19,] 3.490411 2.189124
[20,] 3.168868 2.058824
[21,] 3.285336 2.110807
[22,] 3.155403 2.049590
[23,] 2.829696 1.879959
[24,] 3.365401 2.092681
[25,] 3.086531 2.023940
[26,] 3.236258 2.070876
```

```
naiveBayes_cv(round(0.7 * nrow(data_MAE)), 25, data_MAE)
```

```
          [,1]     [,2]
 [1,] 3.011322 1.985978
 [2,] 3.144491 2.037620
 [3,] 3.029495 2.009576
 [4,] 3.050924 1.981190
 [5,] 3.057866 2.020862
 [6,] 3.150847 2.102257
 [7,] 3.083594 2.044118
 [8,] 3.015464 1.987004
 [9,] 3.121732 2.037278
[10,] 3.077377 2.019494
[11,] 3.058985 1.993502
[12,] 3.053669 2.015048
[13,] 3.073818 2.011286
[14,] 3.190701 2.067031
[15,] 3.012514 1.978796
[16,] 3.085866 2.027360
[17,] 3.190433 2.049590
[18,] 3.079376 2.021546
[19,] 3.086254 2.037278
[20,] 3.029100 2.003762
[21,] 3.043461 2.017100
[22,] 2.996521 1.964090
[23,] 3.256793 2.127223
[24,] 3.266492 2.122777
[25,] 3.044416 2.005130
[26,] 3.088461 2.026676
```

```
nb_n_fold_cross = function(n, data_frame) {
    chopping_point = floor(nrow(data_frame)/n)
    stopping_points = c(1/chopping_point, 1:n) * chopping_point
    error_matrix = matrix(ncol = 2, nrow = n + 1)
    for (i in 1:n) {
        lower_bound = stopping_points[i]
        upper_bound = stopping_points[i + 1]
        temp_test = data_frame[lower_bound:upper_bound, ]
        temp_train = data_frame[-(lower_bound:upper_bound), ]
        model = naiveBayes(rings[-(lower_bound:upper_bound)] ~ ., data = temp_train)
        predicted = as.numeric(predict(model, temp_test))
        error_matrix[i, ] = RMSE_MAE(predicted, nrings[lower_bound:upper_bound])
    }
    error_matrix[n + 1, 1] = mean(error_matrix[1:n, 1])
    error_matrix[n + 1, 2] = mean(error_matrix[1:n, 2])
    return(error_matrix)
}
```

```r
nb_n_fold_cross(10, data_RMSE)
```

```
          [,1]     [,2]
 [1,] 4.236120 3.074519
 [2,] 5.044064 3.930622
 [3,] 3.055311 1.602871
 [4,] 2.486807 1.270335
 [5,] 2.845293 1.507177
 [6,] 4.653033 3.516746
 [7,] 2.730739 1.337321
 [8,] 3.635945 2.497608
 [9,] 2.833920 1.653110
[10,] 3.049041 1.827751
[11,] 3.457027 2.221806
```

```r
nb_n_fold_cross(10, data_MAE)
```

```
          [,1]     [,2]
 [1,] 3.964688 2.843750
 [2,] 5.021961 3.861244
 [3,] 1.599192 1.093301
 [4,] 1.537412 1.143541
 [5,] 1.547494 1.078947
 [6,] 4.469460 3.401914
 [7,] 1.409978 1.069378
 [8,] 3.138357 2.174641
 [9,] 2.336009 1.476077
[10,] 2.664672 1.775120
[11,] 2.768922 1.991791
```

Knn function:

```r
# Because knn can't calculate distance with factor variables, we have to
# create two dummy variables. Also, because knn relies on distance, I scaled
# the data so that variables who have smaller ranges aren't more important.
male = as.numeric(data[, 1] == "M")
female = as.numeric(data[, 1] == "F")
knn_data = cbind(male, female, data[, -1])
knn_train = cbind(male[train_rows], female[train_rows], train[, -1])
knn_test = cbind(male[-train_rows], female[-train_rows], test[, -1])
Error_matrix = matrix(ncol = 2, nrow = 50)
for (i in 1:50) {
    predicted_knn = as.numeric(knn(knn_train[, -10], knn_test[, -10], knn_train[,
        10], k = i))
    Error_matrix[i, ] = cbind(RMSE_MAE(knn_test[, 10], predicted_knn))
}
Error_matrix[argmin(Error_matrix[, 1]), ]
```

```
[1] 3.519414 2.777334
```

```r
Error_matrix[argmin(Error_matrix[, 2]), ]
```

```
[1] 3.585686 2.774142
```

```r
k_RMSE = argmin(Error_matrix[, 1])
k_MAE = argmin(Error_matrix[, 2])
# Delete-d cross validation
knn_cv = function(d, m, data_frame, k) {
    error_matrix = matrix(nrow = m + 1, ncol = 2)
```

```
    for (i in 1:m) {
        d_records = sample(nrow(data_frame), d)
        temp_test = data_frame[-d_records, ]
        temp_train = data_frame[d_records, ]
        predicted_knn = as.numeric(knn(temp_train[, -10], temp_test[, -10],
            cl = temp_train[, 10], k = k))
        error_matrix[i, ] = RMSE_MAE(temp_test[, 10], predicted_knn)
    }
    error_matrix[m + 1, 1] = mean(error_matrix[1:m, 1])
    error_matrix[m + 1, 2] = mean(error_matrix[1:m, 2])
    return(error_matrix)
}
knn_cv(round(0.7 * nrow(knn_data)), 25, knn_data, k = k_RMSE)
```

```
Warning in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NAs introduced by coercion
```

```
Warning in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NAs introduced by coercion
```

```
Error in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NA/NaN/Inf in foreign function
    call (arg 6)
```

```
knn_cv(round(0.7 * nrow(knn_data)), 25, knn_data, k = k_MAE)
```

```
Warning in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NAs introduced by coercion
```

```
Warning in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NAs introduced by coercion
```

```
Error in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NA/NaN/Inf in foreign function
    call (arg 6)
```

```
# 10-fold cross validation
knn_n_fold_cross = function(n, data_frame, k) {
    chopping_point = floor(nrow(data_frame)/n)
    stopping_points = c(1/chopping_point, 1:n) * chopping_point
    error_matrix = matrix(ncol = 2, nrow = n + 1)
    for (i in 1:n) {
        lower_bound = stopping_points[i]
        upper_bound = stopping_points[i + 1]
        temp_test = data_frame[lower_bound:upper_bound, ]
        temp_train = data_frame[-(lower_bound:upper_bound), ]
        predicted = as.numeric(knn(temp_train[, -10], temp_test[, -10], cl = temp_train[,
            10], k = k))
        error_matrix[i, ] = RMSE_MAE(predicted, temp_test[, 10])
    }
    error_matrix[n + 1, 1] = mean(error_matrix[1:n, 1])
    error_matrix[n + 1, 2] = mean(error_matrix[1:n, 2])
    return(error_matrix)
}
knn_n_fold_cross(10, knn_data, k_RMSE)
```

```
Warning in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NAs introduced by coercion
```

```
Warning in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NAs introduced by coercion
```

```
Error in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NA/NaN/Inf in foreign function
    call (arg 6)
```

```
knn_n_fold_cross(10, knn_data, k_MAE)
```

```
Warning in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NAs introduced by coercion
```

```
Warning in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NAs introduced by coercion
```

```
Error in knn(temp_train[, -10], temp_test[, -10], cl = temp_train[, 10], : NA/NaN/Inf in foreign function
    call (arg 6)
```

Weighted Knn-function:

```
k_vector = c(1:50)
# Since the optimal kernel does not always provide the best result, all of
# them are being tested with k values between 1 and 50. Once the optimal
# kernal and k value are calculated, they will cross validated.
optimal_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "optimal",
    ykernel = -1.2:1.2)
triangular_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "triangular",
    ykernel = -1.2:1.2)
gaussian_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "gaussian",
    ykernel = -1.2:1.2)
epanechnikov_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "epanechnikov",
    ykernel = -1.2:1.2)
biweight_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "biweight",
    ykernel = -1.2:1.2)
triweight_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "triweight",
    ykernel = -1.2:1.2)
cosine_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "cos",
    ykernel = -1.2:1.2)
inverted_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "inv",
    ykernel = -1.2:1.2)
rectangular_test = train.kknn(V9 ~ ., data = knn_data, ks = k_vector, kernel = "rectangular",
    ykernel = -1.2:1.2)
optimal_data_frame = rbind(c("optimal", min(optimal_test$MEAN.ABS), min(optimal_test$MEAN.SQU),
    optimal_test$best.parameters$k), c("triangular", min(triangular_test$MEAN.ABS),
    min(triangular_test$MEAN.SQU), triangular_test$best.parameters$k), c("epanechnikov",
    min(epanechnikov_test$MEAN.ABS), min(epanechnikov_test$MEAN.SQU), epanechnikov_test$best.parameters$k),
    c("biweight", min(biweight_test$MEAN.ABS), min(biweight_test$MEAN.SQU),
        biweight_test$best.parameters$k), c("triweight", min(triweight_test$MEAN.ABS),
        min(triweight_test$MEAN.SQU), triweight_test$best.parameters$k), c("cos",
        min(cosine_test$MEAN.ABS), min(cosine_test$MEAN.SQU), cosine_test$best.parameters$k),
    c("inv", min(inverted_test$MEAN.ABS), min(inverted_test$MEAN.SQU), inverted_test$best.parameters$k),
    c("gaussian", min(gaussian_test$MEAN.ABS), min(gaussian_test$MEAN.SQU),
        gaussian_test$best.parameters$k), c("rectangular", min(rectangular_test$MEAN.ABS),
        min(rectangular_test$MEAN.SQU), rectangular_test$best.parameters$k))
# Since train.kknn includes a leave-one-out cross validation technique, I
# will use that method.
optimal_data_frame[argmin(optimal_data_frame[, 2]), ]
```

```
[1] "gaussian"         "1.51924144935422" "4.8439507372699"
[4] "18"
```

```
optimal_data_frame[argmin(optimal_data_frame[, 3]), ]
```

```
[1] "biweight"         "1.52023104377889" "4.82416270611321"
[4] "43"
```

```
optimal_k_abs = optimal_data_frame[argmin(optimal_data_frame[, 2]), 4]
optimal_k_abs = as.numeric(optimal_k_abs)
optimal_kernel_abs = optimal_data_frame[argmin(optimal_data_frame[, 2]), 1]
```

```r
optimal_kernel_abs = as.character(optimal_kernel_abs)
optimal_k_rmse = optimal_data_frame[argmin(optimal_data_frame[, 3]), 4]
optimal_k_rmse = as.numeric(optimal_k_rmse)
optimal_kernel_rmse = optimal_data_frame[argmin(optimal_data_frame[, 3]), 1]
optimal_kernel_rmse = as.character(optimal_kernel_rmse)
# 10 fold cross validation using convenience command
fold_RMSE = cv.kknn(V9 ~ ., data = knn_data, kcv = 10, k = optimal_k_rmse, kernel = optimal_kernel_rmse)
RMSE_MAE(fold_RMSE[[1]][1:4177], fold_RMSE[[1]][4178:(2 * 4177)])
```

```
         [,1]     [,2]
[1,] 2.206253 1.526847
```

```r
fold_MAE = cv.kknn(V9 ~ ., data = knn_data, kcv = 10, k = optimal_k_abs, kernel = optimal_kernel_abs)
RMSE_MAE(fold_MAE[[1]][1:4177], fold_MAE[[1]][4178:(2 * 4177)])
```

```
         [,1]     [,2]
[1,] 2.205438 1.528683
```