# Deeper into k-nearest neighbors

*Brooke Anderson*

*January 26, 2016*

```
knitr::opts_knit$set(fig.path = '../figures/MoreKNN-',
                     root.dir = '..')
library(dplyr)
library(ggplot2)
library(flexclust)  ## To calculate distance matrix
library(StatMatch)
```

## My lurking questions

Some lurking questions about k-nearest neighbors:

- I've seen stuff on how to resolve ties in the majority vote once you've identified the k nearest points, but how do you resolve ties in the distance to points to pick just k?
- How does scaling binary predictive variables influence their weight in the distance measurement? Do you identify different nearest neighbors if you do or don't scale binary variables? If you don't scale, do these variables carry a heavier or lighter weight when picking nearest neighbors than scaled continuous variables?
- What are the implications for how you dealed with ordered multi-level classes like the ticket class of the passengers? What happens when you include that as an ordinal number? If you do, should you scale it? What happens when you include it broken down as different binary indicators (i.e., 0 / 1 for second and third class, in the case of `Pclass`).
- How do measure distance to determine nearest neighbors when values of some parameters are missing?

## Training and testing data

For this, I'll read in the training dataset and split it into one-half `my_train`, to train the model (maybe I should say "train" for k-NN), and one-half `my_test` to test the model. I'll set a seed so you should get the same results. I'll also limit it to the predictive variables of Sex, Pclass, Age, and Fare. For now, I'll remove all missing values and only sample 10 values.

```
train <- read.csv("data/train.csv") %>%
  select(Survived, Sex, Pclass, Age, Fare)

train <- filter(train, complete.cases(train)) %>%
  mutate(nSex = as.numeric(Sex),
         sAge = as.vector(scale(Age)),
         sFare = as.vector(scale(Fare)),
         sSex = as.vector(scale(nSex)))
bPclass <- model.matrix(~ factor(Pclass),
                        data = train)[ , -1]
train[ , c("bPclass2", "bPclass3")] <- bPclass

test <- read.csv("data/test.csv") %>%
  select(Sex, Pclass, Age, Fare)
```

```r
test <- filter(test, complete.cases(test)) %>%
  mutate(nSex = as.numeric(Sex),
         sAge = as.vector(scale(Age)),
         sFare = as.vector(scale(Fare)),
         sSex = as.vector(scale(nSex)))
bPclass <- model.matrix(~ factor(Pclass),
                        data = test)[ , -1]
test[ , c("bPclass2", "bPclass3")] <- bPclass

set.seed(2101)
train_2 <- sample_n(train, 10)

set.seed(21)
train_i <- sample(1:nrow(train_2),
                  size = round(1 / 2 * nrow(train_2)))
(my_train <- train_2[train_i, ])
```

```
##     Survived    Sex Pclass Age    Fare nSex        sAge      sFare
## 504        0   male      3  32   7.925    2  0.15839210 -0.5058589
## 292        0   male      3  35   7.050    2  0.36491125 -0.5223937
## 260        1 female      2  30  12.350    1  0.02071266 -0.4222405
## 692        1 female      2  42  13.000    1  0.84678929 -0.4099575
## 275        1 female      1  24 263.000    1 -0.39232566  4.3142499
##           sSex bPclass2 bPclass3
## 504  0.7585196        0        1
## 292  0.7585196        0        1
## 260 -1.3165110        1        0
## 692 -1.3165110        1        0
## 275 -1.3165110        0        0
```

```r
(my_test <- train_2[-train_i, ])
```

```
##     Survived  Sex Pclass Age    Fare nSex      sAge      sFare      sSex
## 644        0 male      1  39  0.0000    2 0.6402701 -0.6556163 0.7585196
## 408        1 male      1  36 26.2875    2 0.4337510 -0.1588659 0.7585196
## 551        0 male      1  60 26.5500    2 2.0859042 -0.1539055 0.7585196
## 380        0 male      2  34 21.0000    2 0.2960715 -0.2587829 0.7585196
## 369        1 male      1  48 26.5500    2 1.2598276 -0.1539055 0.7585196
##     bPclass2 bPclass3
## 644        0        0
## 408        0        0
## 551        0        0
## 380        1        0
## 369        0        0
```

## "From scratch" k-NN code

To start checking this out, I'll write some of my own code to fit a k-NN model from scratch. This will use the theoretical ideas for a most-basic k-NN model. In some cases, different R packages likely implement the model fit using different algorithms. I'll try to check that out later, but I think this is a good way to try to get a handle on the basics of some of these questions (and maybe appreciate all the fancy things being done by R package code a bit more).

First, I'll calculate distance as Euclidean distance. Euclidean distance between two vectors $p$ and $q$ is:

$$d(\mathbf{p}, \mathbf{q}) = \left( \sum_{i=1}^{n} (p_i - q_i) \right)^{1/2}$$

in R, you can calculate a distance matrix using `dist2` from the `flexclust` package:

```
find_dist_mat <- function(train, test, predictors){
  dist_mat <- dist2(train[ , predictors],
                    test[ , predictors])
  #dist_mat <- gower.dist(train[ , predictors],
  #                       test[ , predictors])
  dist_mat <- t(dist_mat)
  return(dist_mat)
}

find_dist_mat(my_train, my_test, c("Age", "Fare"))
```

```
##          504       292       260      692      275
## 644 10.57382  8.105708 15.281443 13.34166 263.4274
## 408 18.79312 19.263473 15.174120 14.57936 237.0165
## 551 33.62872 31.705678 33.190963 22.53004 239.1748
## 380 13.22708 13.985796  9.530084 11.31371 242.2065
## 369 24.55383 23.436083 22.926840 14.81899 237.6649
```

The default distance metric for `dist2` is Euclidean, but you can also specify alternative distance matrics ("maximum", "manhattan", "canberra", "binary" or "minkowski"). In terms of computational speed (from the help file):

> "The current implementation is efficient only if y has not too many rows (the code is vectorized in x [first matrix] but not in y [second matrix])."

Now I need to identify the indices of the `k` lowest values in each row of this matrix. I'll do a function that can do that for a row, and then apply it across all rows:

```
find_k_indices <- function(x, k){
  order_indices <- order(x)
  nearest_ind <- order_indices[1:k]
  return(nearest_ind)
}

dist_mat <- find_dist_mat(my_train, my_test,
                          c("Age", "Fare"))
find_k_indices(dist_mat[1, ], k = 3)
```

```
## [1] 2 1 4
```

In terms of handling ties in the `sort`, here's a note from the `order` help file:

> "Any unresolved ties will be left in their original ordering."

Each row represents a point in `my_test`, and each column gives the Euclidean distance between that point and a point in `my_train`.

```
find_nn_indices <- function(train,
                            test,
                            predictors,
                            k){

  dist_mat <- find_dist_mat(train, test, predictors)

  index_mat <- apply(dist_mat, 1, find_k_indices, k)
  if(!is.matrix(index_mat)){
    index_mat <- as.matrix(index_mat, nrow = 1)
  } else {
    index_mat <- t(index_mat)
  }
  return(index_mat)
  }
```

```
find_nn_indices(train = my_train, test = my_test,
               predictors = c("Age", "Fare"),
               k = 1)
```

```
##     [,1]
## 644    2
## 408    4
## 551    4
## 380    3
## 369    4
```

```
find_nn_indices(train = my_train, test = my_test,
               predictors = c("Age", "Fare"),
               k = 3)
```

```
##     [,1] [,2] [,3]
## 644    2    1    4
## 408    4    3    1
## 551    4    2    3
## 380    3    4    1
## 369    4    3    2
```

Now I can use this matrix of k-nearest indices to pick out the nearest neighbor votes from the training data:

```
nn_votes <- function(nn_indices, train_y){
  votes <- apply(nn_indices, 1, function(x) train_y[x])
  if(!is.matrix(votes)){
    votes <- as.matrix(votes, nrow = 1)
  } else {
    votes <- t(votes)
  }
  return(votes)
}
```

```r
nn_indices <- find_nn_indices(my_train, my_test,
                              predictors = c("Age", "Fare"),
                              k = 3)
nn_indices
```

```
##     [,1] [,2] [,3]
## 644    2    1    4
## 408    4    3    1
## 551    4    2    3
## 380    3    4    1
## 369    4    3    2
```

```r
my_train$Survived
```

```
## [1] 0 0 1 1 1
```

```r
nn_votes(nn_indices, train_y = my_train$Survived)
```

```
##     [,1] [,2] [,3]
## 644    0    0    1
## 408    1    1    0
## 551    1    0    1
## 380    1    1    0
## 369    1    1    0
```

```r
nn_indices <- find_nn_indices(my_train, my_test,
                              predictors = c("Age", "Fare"),
                              k = 1)
nn_indices
```

```
##     [,1]
## 644    2
## 408    4
## 551    4
## 380    3
## 369    4
```

```r
my_train$Survived
```

```
## [1] 0 0 1 1 1
```

```r
nn_votes(nn_indices, train_y = my_train$Survived)
```

```
##     [,1]
## 644    0
## 408    1
## 551    1
## 380    1
## 369    1
```

Then I can generate the predictions (1 if the average of votes is above 0.5, 0 otherwise – as long as I only use odd `k` values, this average will never be exactly 0.5).

```
my_prediction <- function(votes){
  mean_vote <- apply(votes, 1, mean)
  prediction <- factor(mean_vote < 0.5,
                       levels = c(TRUE, FALSE),
                       labels = c("0", "1"))
  return(prediction)
}

nn_indices <- find_nn_indices(my_train, my_test,
                              predictors = c("Age", "Fare"),
                              k = 3)
votes <- nn_votes(nn_indices,
                  train_y = my_train$Survived)
votes
```

```
##     [,1] [,2] [,3]
## 644    0    0    1
## 408    1    1    0
## 551    1    0    1
## 380    1    1    0
## 369    1    1    0
```

```
my_prediction(votes)
```

```
## 644 408 551 380 369
##   0   1   1   1   1
## Levels: 0 1
```

Putting everything into a function:

```
my_knn_function <- function(train, test,
                            predictors,
                            outcome,
                            k){
  nn_indices <-find_nn_indices(train,
                               test,
                               predictors,
                               k)
  votes <- nn_votes(nn_indices,
                    train_y = train[ , outcome])
  out <- my_prediction(votes)
  return(out)
}

my_knn_function(train = my_train, test = my_test,
                predictors = c("Age", "Fare"),
                outcome = "Survived", k = 3)
```

```
## 644 408 551 380 369
##   0   1   1   1   1
## Levels: 0 1
```

## Role of scaling predictors

### Scaling continuous predictors

Now I can use these functions to check out the influence of some different choices. For example, we know that it's important to scale continuous variables, so we'll ultimately want to do that, but I can check out how important that is. I'll create two new variables with scaled age and fare, `sAge` and `sFare`, to use to check.

First, I calculated the distance matrix using the scaled and unscaled predictors:

```
find_dist_mat(my_train, my_test,c("Age", "Fare"))
```

```
##          504       292       260      692      275
## 644 10.57382  8.105708 15.281443 13.34166 263.4274
## 408 18.79312 19.263473 15.174120 14.57936 237.0165
## 551 33.62872 31.705678 33.190963 22.53004 239.1748
## 380 13.22708 13.985796  9.530084 11.31371 242.2065
## 369 24.55383 23.436083 22.926840 14.81899 237.6649
```

```
find_dist_mat(my_train, my_test,c("sAge", "sFare"))
```

```
##           504       292       260       692      275
## 644 0.5046124 0.3058934 0.6620542 0.3209336 5.076005
## 408 0.4429748 0.3699883 0.4898641 0.4833711 4.548755
## 551 1.9593811 1.7600001 2.0825513 1.2652938 5.109407
## 380 0.2828466 0.2724510 0.3202201 0.5710900 4.624556
## 369 1.1563007 0.9678114 1.2678366 0.4859664 4.763824
```

Then I checked the selection of nearest neighbors, using `k = 3`:

```
find_nn_indices(train = my_train, test = my_test,
                predictors =c("Age", "Fare"), k = 3)
```

```
##     [,1] [,2] [,3]
## 644    2    1    4
## 408    4    3    1
## 551    4    2    3
## 380    3    4    1
## 369    4    3    2
```

```
find_nn_indices(train = my_train, test = my_test,
                predictors =c("sAge", "sFare"), k = 3)
```

```
##     [,1] [,2] [,3]
## 644    2    4    1
## 408    2    1    4
## 551    4    2    1
## 380    2    1    3
## 369    4    2    1
```

While some of the nearest neighbors identified are the same, occasionally some differ. For example, the unscaled analysis identifies the fourth entry in the training data set as a nearest neighbor to the fourth test point, while the scaled analysis suggests the second member of the dataset is closer instead. Here are those passengers:

First, the passenger we need to predict:

```
my_test[4, ]
```

```
##     Survived  Sex Pclass Age Fare nSex      sAge      sFare      sSex
## 380        0 male      2  34   21    2 0.2960715 -0.2587829 0.7585196
##     bPclass2 bPclass3
## 380        1        0
```

Here's the training passenger identified as a near neighbor by the analysis with unscaled predictors but not the analysis with scaled predictors:

```
my_train[4, ]
```

```
##     Survived    Sex Pclass Age Fare nSex      sAge      sFare      sSex
## 692        1 female      2  42   13    1 0.8467893 -0.4099575 -1.316511
##     bPclass2 bPclass3
## 692        1        0
```

Here's the training passenger identified as a near neighbor by the analysis with scaled predictors but not the analysis with unscaled predictors:

```
my_train[2, ]
```

```
##     Survived  Sex Pclass Age Fare nSex      sAge      sFare      sSex
## 292        0 male      3  35 7.05    2 0.3649113 -0.5223937 0.7585196
##     bPclass2 bPclass3
## 292        0        1
```

`Fare` is more similar between the test data point and the training point identified using unscaled predictors; `Age` is more similar with the training point identified using scaled predictors. This makes sense, because the unscaled scale of `Fare` is much larger than that of `Age` and so would cause `Fare` to carry more weight in measuring Euclidean distance if you don't scale these continuous predictors.

### Scaling a categorical predictor

Here's a similar analysis of what happens when you just convert a binary categorical predictor to a numeric value versus when you scale it after you convert. Here `nSex` is a version of the `Sex` variable where the factor levels have been converted to numbers (1 = female, 2 = male), while `sSex` is a version of the same variable, but scaled.

Here are the distance metrics, done for using each of these predictors in conjunction with the `sFare` predictor:

```
find_dist_mat(my_train, my_test,c("nSex", "sFare"))
```

```
##           504       292      260      692      275
## 644 0.1497574 0.1332226 1.026871 1.029732 5.069474
## 408 0.3469930 0.3635278 1.034102 1.031042 4.583532
## 551 0.3519535 0.3684882 1.035376 1.032261 4.578691
## 380 0.2470760 0.2636108 1.013271 1.011362 4.681093
## 369 0.3519535 0.3684882 1.035376 1.032261 4.578691
```

```
find_dist_mat(my_train, my_test,c("sSex", "sFare"))
```

```
##           504       292      260      692      275
## 644 0.1497574 0.1332226 2.088113 2.089522 5.385659
## 408 0.3469930 0.3635278 2.091678 2.090167 4.930975
## 551 0.3519535 0.3684882 2.092309 2.090769 4.926476
## 380 0.2470760 0.2636108 2.081459 2.080530 5.021791
## 369 0.3519535 0.3684882 2.092309 2.090769 4.926476
```

And here are the indices of the nearest neighbors identified based on these predictors, with `k = 3`:

```
(a <- find_nn_indices(train = my_train, test = my_test,
                predictors =c("nSex", "sFare"), k = 3))
```

```
##     [,1] [,2] [,3]
## 644    2    1    3
## 408    1    2    4
## 551    1    2    4
## 380    1    2    4
## 369    1    2    4
```

```
(b <- find_nn_indices(train = my_train, test = my_test,
                predictors =c("sSex", "sFare"), k = 3))
```

```
##     [,1] [,2] [,3]
## 644    2    1    3
## 408    1    2    4
## 551    1    2    4
## 380    1    2    4
## 369    1    2    4
```

For this small dataset, these two methods gave the exact same sets of nearest neighbors:

```
sum(apply(a != b, 1, sum) > 0) ## Number of testing
```

```
## [1] 0
```

```
                                ## points with different
                                ## nearest neighbors
                                ## (order counts)
```

I decided to check and see if it made any difference on the full training and testing datasets. Even in the full dataset, this choice of whether to scale the categorical predictor of `Sex` made absolutely no difference in which training-set points were predicted as the nearest neighbors of each testing point.

```r
a <- find_nn_indices(train = train,
                     test = test,
                predictors =c("nSex", "sFare"), k = 3)
b <- find_nn_indices(train = train,
                     test = test,
                predictors =c("sSex", "sFare"), k = 3)
sum(apply(a != b, 1, sum) > 0)
```

```
## [1] 0
```

**Choosing how to handle an ordinal predictor**

One of the predictors, `Pclass`, is ordinal. How are k-NN predictions affected by your choice of the following two ways to deal with this predictor:

- Convert to a numerical value (`Pclass` in the data). This will give it values that increase by one unit for each change from one category to the next-higher category of the predictor.
- Convert to binary variables (`bPclass2` and `bPclass3` in the data). This essentially loses the information inherent in the predictor about order but removes the assumption that the difference between each set of contiguous categories is the same.

Here are the distance metrics, done for using each of these predictors in conjunction with the `sAge` predictor:

```r
find_dist_mat(my_train, my_test,c("Pclass", "sAge"))
```

```
##          504      292       260       692       275
## 644 2.057233 2.018867 1.1763722 1.0211024 1.0325958
## 408 2.018867 2.001184 1.0819430 1.0819430 0.8260766
## 551 2.777643 2.638526 2.2945623 1.5922958 2.4782299
## 380 1.009433 1.002367 0.2753589 0.5507178 1.2140390
## 369 2.283235 2.191090 1.5922958 1.0819430 1.6521533
```

```r
find_dist_mat(my_train, my_test,c("bPclass2",
                                  "bPclass3", "sAge"))
```

```
##          504      292       260       692       275
## 644 1.110048 1.037219 1.1763722 1.0211024 1.0325958
## 408 1.037219 1.002367 1.0819430 1.0819430 0.8260766
## 551 2.171475 1.990431 2.2945623 1.5922958 2.4782299
## 380 1.420900 1.415888 0.2753589 0.5507178 1.2140390
## 369 1.487669 1.341967 1.5922958 1.0819430 1.6521533
```

And here are the indices of the nearest neighbors identified based on these predictors, with `k = 3`:

```r
(a <- find_nn_indices(train = my_train, test = my_test,
                predictors =c("Pclass", "sAge"), k = 3))
```

```
##     [,1] [,2] [,3]
## 644    4    5    3
```

```
## 408    5    3    4
## 551    4    3    5
## 380    3    4    2
## 369    4    3    5
```

```
(b <- find_nn_indices(train = my_train, test = my_test,
                predictors =c("bPclass2", "bPclass3",
                              "sAge"), k = 3))
```

```
##     [,1] [,2] [,3]
## 644    4    5    2
## 408    5    2    1
## 551    4    2    1
## 380    3    4    5
## 369    4    2    1
```

In this case, there is a good bit of difference in who is identified as a nearest neighbor depending on the choice of method.

For example, here is the second passenger that we need to predict for:

```
my_test[2, ]
```

```
##     Survived  Sex Pclass Age    Fare nSex     sAge      sFare      sSex
## 408        1 male      1  36 26.2875    2 0.433751 -0.1588659 0.7585196
##     bPclass2 bPclass3
## 408        0        0
```

When using the “ordinal” method, the following passenger from the training data are identified as a nearest neighbor:

```
my_train[c(3, 4), ]
```

```
##     Survived    Sex Pclass Age  Fare nSex       sAge      sFare      sSex
## 260        1 female      2  30 12.35    1 0.02071266 -0.4222405 -1.316511
## 692        1 female      2  42 13.00    1 0.84678929 -0.4099575 -1.316511
##     bPclass2 bPclass3
## 260        1        0
## 692        1        0
```

while when using the “binary” method, the following training passengers were identified as a nearest neighbor instead:

```
my_train[c(2, 1), ]
```

```
##     Survived  Sex Pclass Age  Fare nSex      sAge      sFare      sSex
## 292        0 male      3  35 7.050    2 0.3649113 -0.5223937 0.7585196
## 504        0 male      3  32 7.925    2 0.1583921 -0.5058589 0.7585196
##     bPclass2 bPclass3
## 292        0        1
## 504        0        1
```

For the binary method, `Age` is prioritized once you fail on an exact match for `Pclass`. For the ordinal method, a `Pclass` value that is only off by one class can still override age-related distance when finding a nearest neighbor.

It will be interesting to see how these two methods compare in terms of accuracy in the Titanic competition. The ordinal approach seems like it might be a winner– it looks like it's picking out nearest neighbors that aren't too far away in age while helping to grab, for example, second-class passengers to match with a first class passenger.