K Nearest Neighbor

Ruoqing Zhu

Contents

K-Neariest Neighber

K nearest neighbor (KNN) is a simple nonparametric method. It can be used for both regression and classification problems. However, the idea is quite different from models we introduced before. In a linear model, we have a set of parameters $\boldsymbol{\beta}$ and our estimated function value, for any target point x_0 is $x_0^\text{T}\boldsymbol{\beta}$. In KNN, we don't really specify these parameters. Instead, we directed estimate $f(x_0)$. This is traditionally called **nonparametric models** in statistics. Usually these models perform a local averaging technique to estimate $f(x_0)$ using observations close to x_0.

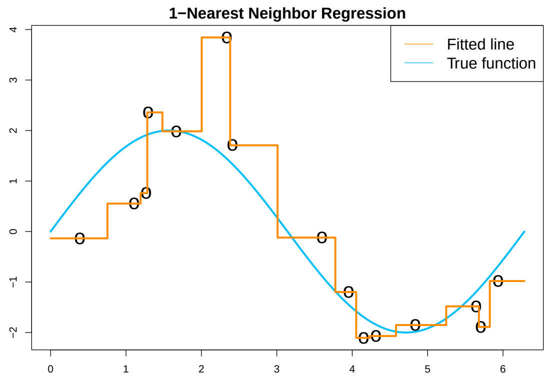
Suppose we collect a set of observations $\{x_i, y_i\}_{i=1}^n$, the prediction at a new target point x_0 is

$$\widehat{y} = \frac{1}{k} \sum_{x_i \in N_k(x_0)} y_i,$$

where $N_k(x_0)$ defines the k samples from the training data that are closest to x_0. As default, closeness is defined using a distance measure, such as the Euclidean distance. Here is a demonstration of the fitted regression function in a one-dimensional case. We use the sin function as the true underlying function.

```r
# generate training data with 2*sin(x) and random Gaussian errors
set.seed(1)
x <- runif(15, 0, 2*pi)
y <- 2*sin(x) + rnorm(length(x))

# generate testing data points where we evaluate the prediction function
test.x = seq(0, 1, 0.001)*2*pi

# "1-nearest neighbor" regression using kknn package
```

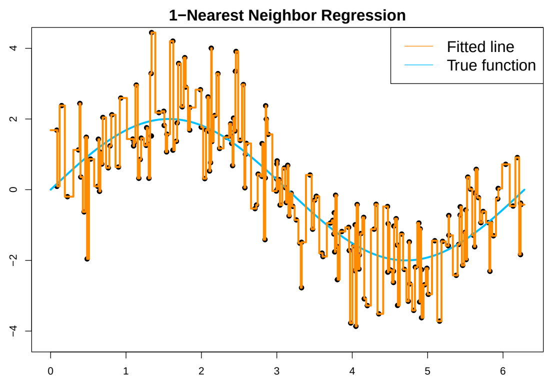
```r
library(kknn)
knn.fit = kknn(y ~ ., train = data.frame(x = x, y = y),
               test = data.frame(x = test.x),
               k = 1, kernel = "rectangular")
test.pred = knn.fit$fitted.values

# plot the data
par(mar=rep(2,4))
plot(x, y, xlim = c(0, 2*pi), pch = "o", cex = 2,
     xlab = "", ylab = "", cex.lab = 1.5)
title(main="1-Nearest Neighbor Regression", cex.main = 1.5)

# plot the true regression line
lines(test.x, 2*sin(test.x), col = "deepskyblue", lwd = 3)

# plot the fitted line
lines(test.x, test.pred, type = "s", col = "darkorange", lwd = 3)
legend("topright", c("Fitted line", "True function"),
       col = c("darkorange", "deepskyblue"), lty = 1, cex = 1.5)
```



Tuning k

Tuning k is crucial for kNN. Let's observe its effect. This time, I generate 200 observations. For 1NN, the fitted regression line is very jumpy.

```r
# generate more data
set.seed(1)
n = 200
x <- runif(n, 0, 2*pi)
y <- 2*sin(x) + rnorm(length(x))

test.y = 2*sin(test.x) + rnorm(length(test.x))

# 1-nearest neighbor
knn.fit = kknn(y ~ ., train = data.frame("x" = x, "y" = y),
               test = data.frame("x" = test.x),
               k = 1, kernel = "rectangular")
test.pred = knn.fit$fitted.values
```

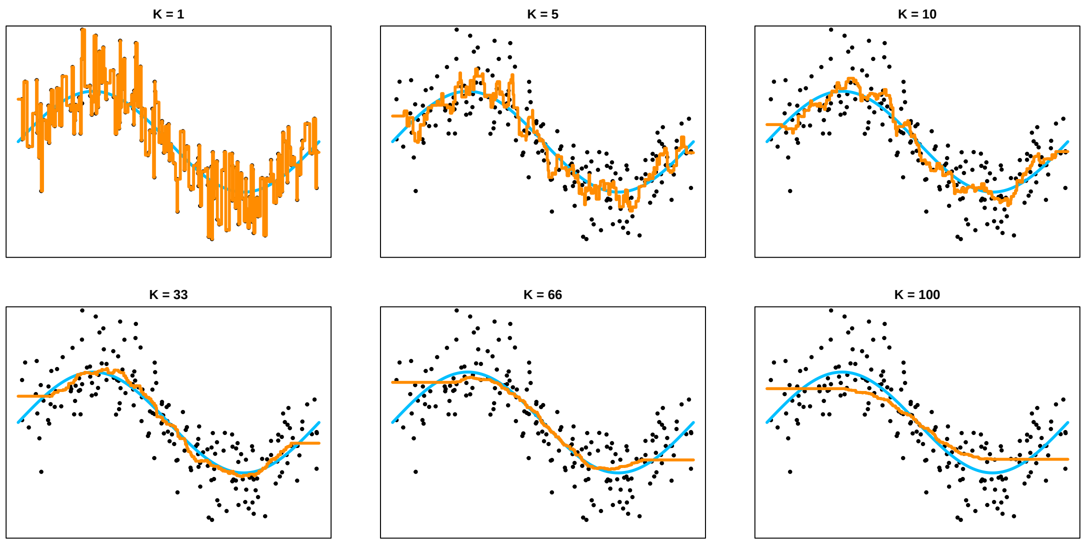


We can evaluate the prediction error of this model:

```
  # prediction error
  mean((test.pred - test.y)^2)
## [1] 2.097488
```

If we consider different values of k, we can observe the trade-off between bias and variance.

```
## Prediction Error for K = 1 : 2.09748780881932
## Prediction Error for K = 5 : 1.39071867992277
## Prediction Error for K = 10 : 1.24696415340282
## Prediction Error for K = 33 : 1.21589627474692
## Prediction Error for K = 66 : 1.37604707375972
## Prediction Error for K = 100 : 1.42868908518756
```



As k increases, we have a more stable model, i.e., smaller variance. However, the bias is also increased. As k decreases, the bias decreases, but the model is less stable.

The Bias-variance Trade-off

Formally, the prediction error (at a given target point x_0) can be broke down into three parts: the **irreducible error**, the **bias squared**, and the **variance**.

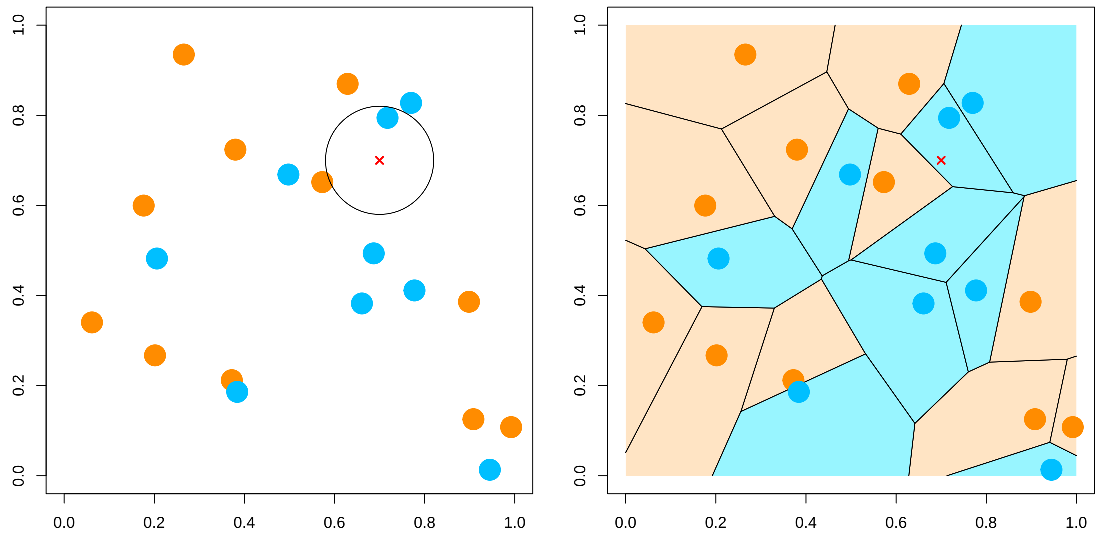
$$\begin{aligned}
\mathrm{E}\Big[\big(Y - \widehat{f}(x_0)\big)^2\Big] &= \mathrm{E}\Big[\big(Y - f(x_0) + f(x_0) - \mathrm{E}[\widehat{f}(x_0)] + \mathrm{E}[\widehat{f}(x_0)] - \widehat{f}(x_0)\big)^2\Big] \\
&= \mathrm{E}\Big[\big(Y - f(x_0)\big)^2\Big] + \mathrm{E}\Big[\big(f(x_0) - \mathrm{E}[\widehat{f}(x_0)]\big)^2\Big] + \mathrm{E}\Big[\big(E[\widehat{f}(x_0)] - \widehat{f}(x_0)\big)^2\Big] + \text{Cross Terms} \\
&= \underbrace{\mathrm{E}\Big[(Y - f(x_0))^2\Big]}_{\text{Irreducible Error}} + \underbrace{\Big(f(x_0) - \mathrm{E}[\widehat{f}(x_0)]\Big)^2}_{\text{Bias}^2} + \underbrace{\mathrm{E}\Big[\big(\widehat{f}(x_0) - \mathrm{E}[\widehat{f}(x_0)]\big)^2\Big]}_{\text{Variance}}
\end{aligned}$$

As we can see from the previous example, when $k = 1$, the prediction error is about 2. This is because for all the testing points, the theoretical irreducible error is 1 (variance of the error term), the bias is almost 0 since the function is smooth, and the variance is the variance of 1 nearest neighbor, which is again 1. On the other extreme side, when $k = n$, the variance should be in the level of $1/n$, the bias is the difference between the sin function and the overall average. Again, to summarize the trade-off:

- As k increases, bias increases and variance decreases
- As k decreases, bias decreases and variance increases

KNN for Classification

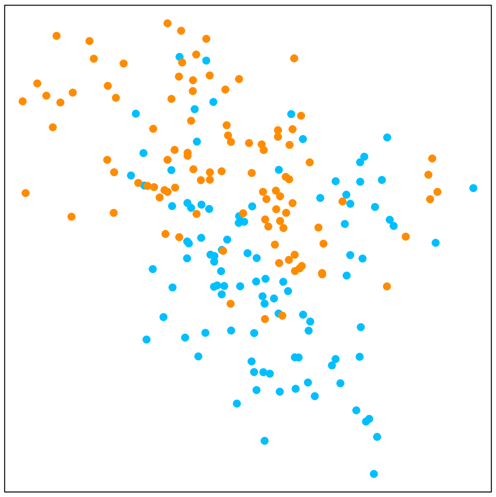
For classification, kNN is different from the regression model in term of finding neighbors. The only difference is to majority voting instead of averaging. Majority voting means that we look for the most popular class label among its neighbors. For 1NN, it is simply the class of the closest neighbor. The visualization of 1NN is a Voronoi tessellation. The plot on the left is some randomly observed data in $[0,1]^2$, and the plot on the right is the corresponding 1NN classification model.



Example: An artificial data

We use artificial data from the `ElemStatLearn` package.

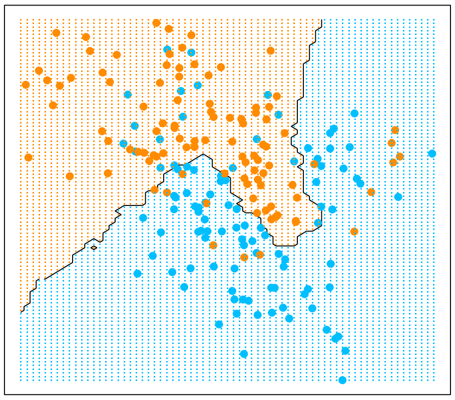
```
library(ElemStatLearn)

x <- mixture.example$x
y <- mixture.example$y
xnew <- mixture.example$xnew

par(mar=rep(2,4))
plot(x, col=ifelse(y==1, "darkorange", "deepskyblue"),
     axes = FALSE, pch = 19)
box()
```



The decision boundary seems to be highly nonlinear. We can utilize the `contour()` function to demonstrate the result of $k = 15$.

```
# knn classification
k = 15
knn.fit <- knn(x, xnew, y, k=k)

px1 <- mixture.example$px1
px2 <- mixture.example$px2
pred <- matrix(knn.fit == "1", length(px1), length(px2))

contour(px1, px2, pred, levels=0.5, labels="",axes=FALSE)
box()
title(paste(k, "-Nearest Neighbor", sep= ""))
points(x, col=ifelse(y==1, "darkorange", "deepskyblue"), pch = 19)
mesh <- expand.grid(px1, px2)
points(mesh, pch=".", cex=1.2, col=ifelse(pred, "darkorange", "deepskyblue"))
```

15−Nearest Neighbor



We can evaluate the in-sample prediction result (since there is no testing sample available) of this model using a confusion matrix:

```
  # the confusion matrix
  knn.fit <- knn(x, x, y, k = 15)
  xtab = table(knn.fit, y)

  library(caret)
  confusionMatrix(xtab)
## Confusion Matrix and Statistics
##
##        y
## knn.fit  0  1
##       0 82 13
##       1 18 87
##
##                Accuracy : 0.845
##                  95% CI : (0.7873, 0.8922)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.69
##
##  Mcnemar's Test P-Value : 0.4725
##
##             Sensitivity : 0.8200
##             Specificity : 0.8700
##          Pos Pred Value : 0.8632
##          Neg Pred Value : 0.8286
##              Prevalence : 0.5000
##          Detection Rate : 0.4100
##    Detection Prevalence : 0.4750
##       Balanced Accuracy : 0.8450
##
##        'Positive' Class : 0
```

```
##
```

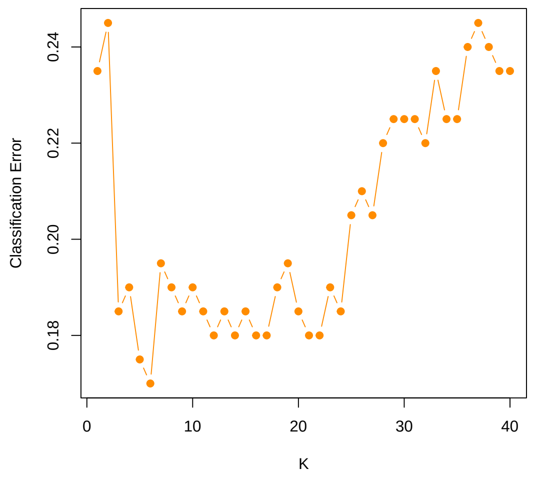
Tuning with the caret Package

The `caret` package has some built-in feature that can tune some popular machine learning models using cross-validation. The cross-validation settings need to be specified using the `trainControl()` function.

```r
library(caret)
control <- trainControl(method = "cv", number = 10)
```

There are other cross-validation methods, such as `repeatedcv` the repeats the CV several times, and leave-one-out CV `LOOCV`. For more details, you can read the documentation. We can then setup the training by specifying a grid of k values, and also the CV setup. Make sure that you specify `method = "knn"` and also construct the outcome as a factor in a data frame.

```r
set.seed(1)
knn.cvfit <- train(y ~ ., method = "knn",
                   data = data.frame("x" = x, "y" = as.factor(y)),
                   tuneGrid = data.frame(k = seq(1, 40, 1)),
                   trControl = control)

plot(knn.cvfit$results$k, 1-knn.cvfit$results$Accuracy,
     xlab = "K", ylab = "Classification Error", type = "b",
     pch = 19, col = "darkorange")
```



Print out the fitted object, we can see that the best k is 6. And there is a clear "U" shaped pattern that shows the potential bias-variance trade-off.

Distance Measures

Closeness between two points needs to be defined based on some distance measures. By default, we use the squared Euclidean distance (ℓ_2 norm) for continuous variables:

$$d^2(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|_2^2 = \sum_{j=1}^{p} (u_j - v_j)^2.$$

However, this measure is not scale invariant. A variable with large scale can dominate this measure. Hence, we often consider a normalized version:

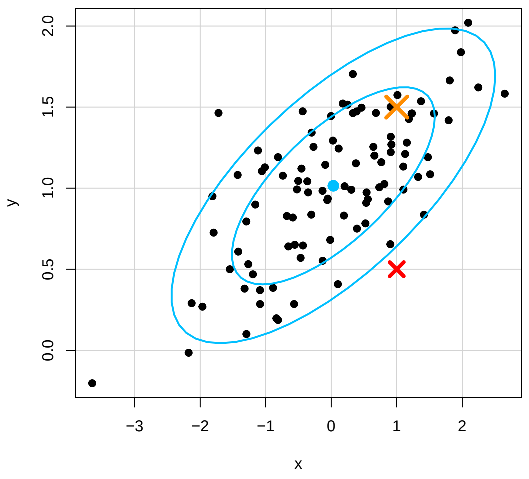
$$d^2(\mathbf{u}, \mathbf{v}) = \sum_{j=1}^{p} \frac{(u_j - v_j)^2}{\sigma_j^2},$$

where σ_j^2 can be estimated using the sample variance of variable j. Another choice that further taking the covariance structure into consideration is the **Mahalanobis distance**:

$$d^2(\mathbf{u}, \mathbf{v}) = (\mathbf{u} - \mathbf{v})^{\mathrm{T}} \Sigma^{-1} (\mathbf{u} - \mathbf{v}),$$

where Σ is the covariance matrix, and can be estimated using the sample covariance. In the following plot, the red cross and orange cross have the same Euclidean distance to the center. However, the red cross is more of a "outlier" based on the joint distribution. The Mahalanobis distance would reflect this.

```
x=rnorm(100)
y=1 + 0.3*x + 0.3*rnorm(100)

library(car)
dataEllipse(x, y, levels=c(0.6, 0.9), col = c("black", "deepskyblue"), pch = 19)
points(1, 0.5, col = "red", pch = 4, cex = 2, lwd = 4)
points(1, 1.5, col = "darkorange", pch = 4, cex = 3, lwd = 4)
```



For categorical variables, the Hamming distance is commonly used. It simply counts how many entries are not the same.

$$d(\mathbf{u}, \mathbf{v}) = \sum_{j=1}^{p} I(u_j \neq v_j).$$

Computational Issues

KNN can be quite computationally intense for large sample size because to find the nearest neighbors, we need to calculate and compare the distances to each of the data point. In addition, it is not memory friendly because we need to store the entire training dataset for future prediction. In contrast, for linear model, we only need to store the estimated $\boldsymbol{\beta}$ parameters. Some algorithms have been developed to search for the neighbors more efficiently. You can try the `FNN` package for these faster computations when n is large.

Different kNN functions

You should be careful about which function to use when performing kNN. Some packages are for regression and some are for classification. Not many functions can be used for both purposes. Try using `??knn` to see a summary of different functions.