KNN Regression

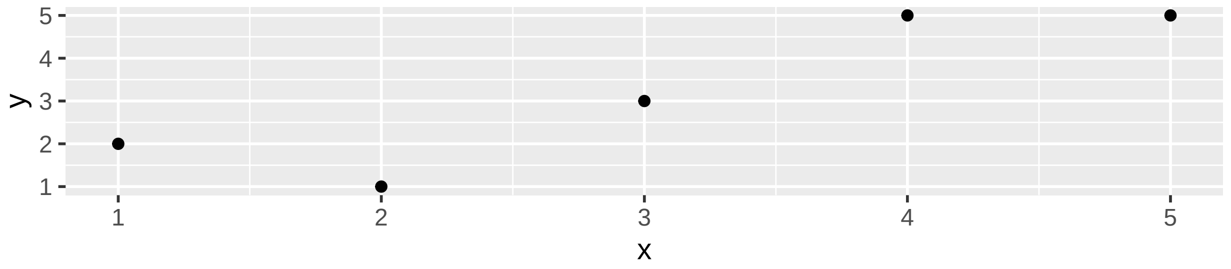
Most examples here are adapted from examples discussed at https://daviddalpiaz.github.io/r4sl/knn-reg.html (this is a useful companion to our text).

Suppose we have the following data:

```
train_data
```

```
##   x y
## 1 1 2
## 2 2 1
## 3 3 3
## 4 4 5
## 5 5 5
```

```
ggplot(data = train_data, mapping = aes(x = x, y = y)) +
  geom_point()
```



2 Basic Approaches to Estimating $f(x)$:

1. Specify a model like $f(x) = \beta_0 + \beta_1 x$. Estimate the *parameters* β_0 and β_1.
2. Local Approach: $f(x_0)$ should look like the data in a neighborhood of x_0

Models that don't specify a specific parametric form for f are often called *nonparametric*.

K Nearest Neighbors

- The predicted value at a test point x_0 is the average of the K training set observations that are closest to x_0 (the K nearest neighbors).

$$\hat{f}(x_0) = \frac{1}{K} \sum_{i \in N_0^{(k)}} y_i$$

- Here $N_0^{(k)}$ is a set of indices for the k observations that have values x_i nearest to the test point x_0.

Using our example data above:

- Suppose we want to make a prediction at the test point $x_0 = 3.75$
- Set $k = 3$
- In our training data set, what are the k nearest neighbors to the test point?

- What is the fitted/predicted value at x_0?

The `train` and `predict` functions in the `caret` package will do these calculations for us:

```r
library(caret)

# "train" the KNN model -- but note there are no parameters to estimate like in linear regression!
knn_fit <- train(
  form = y ~ x,
  data = train_data,
  method = "knn",
  trControl = trainControl(method = "none"),
  tuneGrid = data.frame(k = 3)
)

# get test set predictions.  here we have just one point in the test set
test_data <- data.frame(
  x = 3.75
)
test_data
```

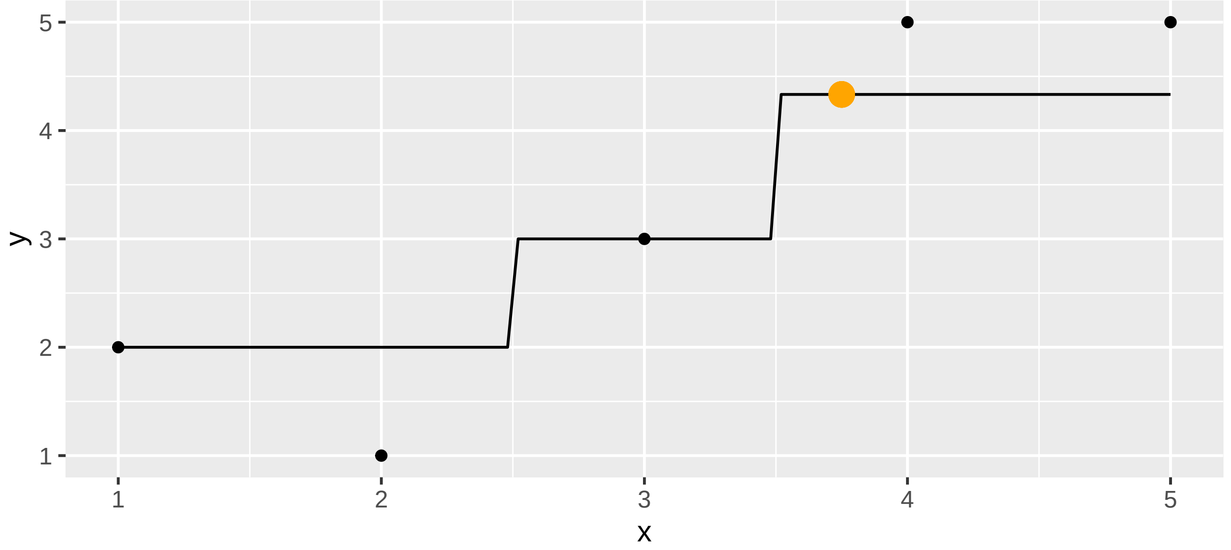
```
##      x
## 1 3.75
```

```r
predict(knn_fit, newdata = test_data)
```

```
## [1] 4.333333
```

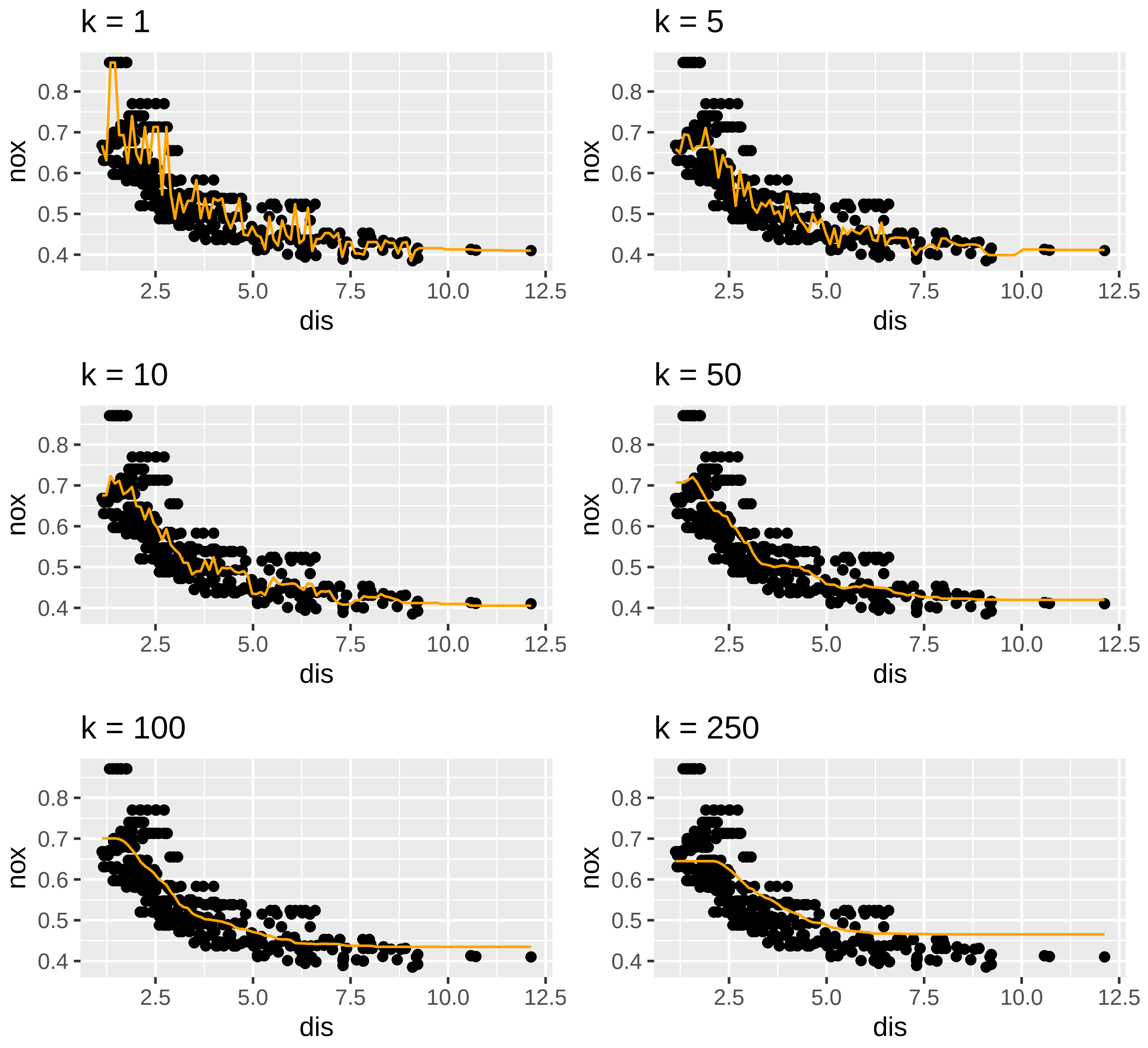
```r
# to make a plot of the fitted function f hat, we can set up a function to get predicted values
predict_knn <- function(x) {
  predict(knn_fit, newdata = data.frame(x = x))
}

ggplot(data = train_data, mapping = aes(x = x, y = y)) +
#  geom_line(data = test_data_for_plot, mapping = aes(x = x, y = y_hat)) +
  stat_function(fun = predict_knn) +
  geom_point() +
  geom_point(mapping = aes(x = 3.75, y = 4.333333), color = "orange", size = 4)
```



Flexibility is determined by k

Recall we previously fit models for the relationship between nitrogen oxides concentrations and distance from Boston employment centers in 506 neighborhoods around Boston. Below are predictions from KNN models with varying values of k.



- Is the bias of KNN lower for large k or small k?

- Is the variance of KNN lower for large k or small k?

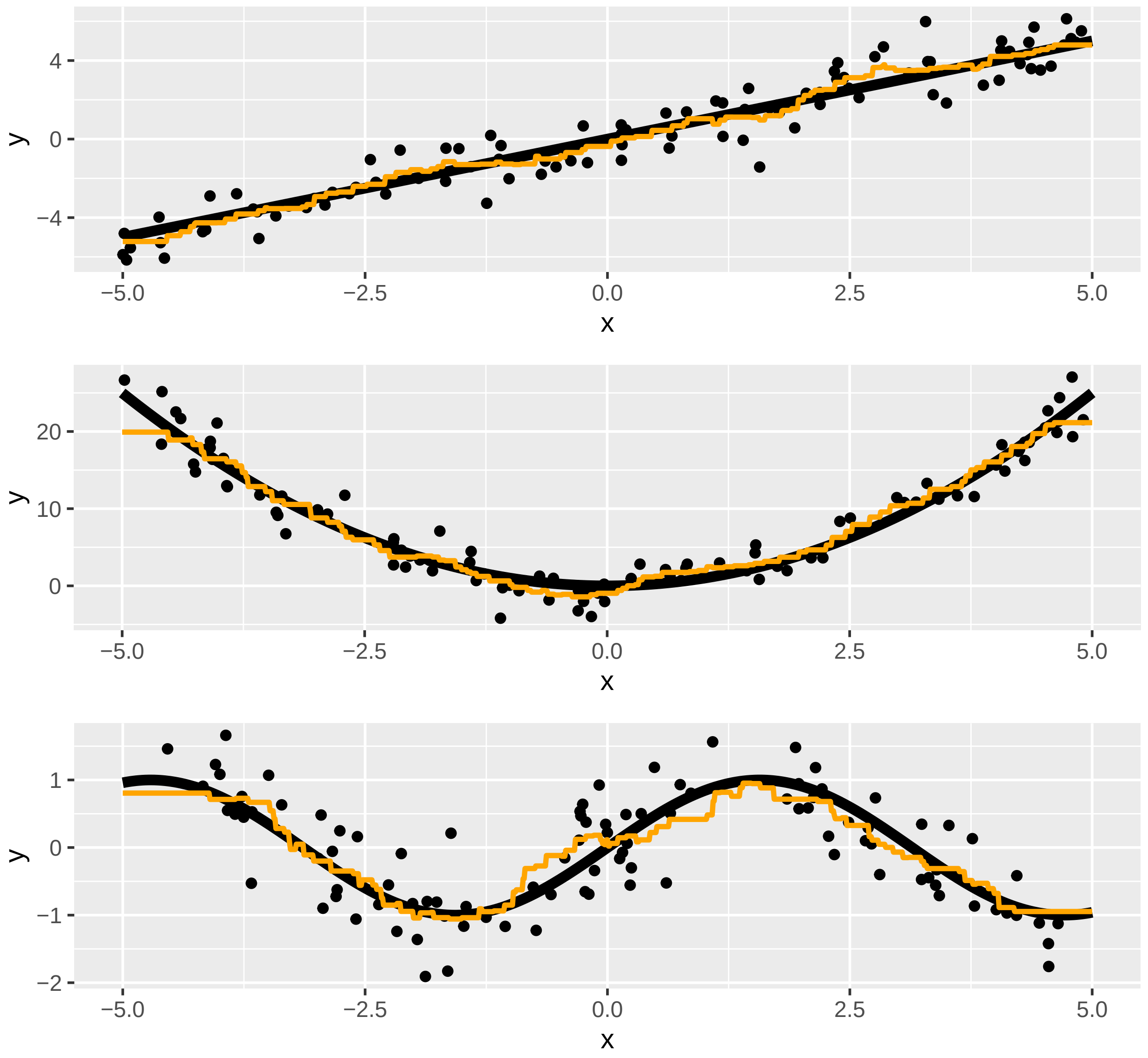
- How should we choose the value of k?

KNN Automatically Adjusts to Different Functional Forms

I simulated three fake data sets of size $n = 100$:

- One where the true function is linear
- A second where the true function is quadratic
- A third where the true function is sinusoidal

In all cases, a KNN fit with $k = 10$ nearest neighbors (shown in orange) does a good job at recovering the underlying function (shown in black).



KNN with multiple explanatory variables

The basic idea is the same, but now we use Euclidean distance to find the nearest neighbors.

In our fake example, suppose we now have two x variables:

```r
train_data
```

```
##   x1 x2 y
## 1  1  2 2
## 2  2  3 1
## 3  3  1 3
## 4  4  5 5
## 5  5  4 5
```

We now want to make a prediction at the test set point $x_0 = (3.75, 2)$ using the $k = 3$ nearest neighbors.

We need to first find the Euclidean distance of each training set observation from the test set point:

```r
train_data <- train_data %>% mutate(
  distance_from_test = sqrt((x1 - 3.75)^2 + (x2 - 2)^2)
)
train_data
```

```
##   x1 x2 y distance_from_test
## 1  1  2 2           2.750000
## 2  2  3 1           2.015564
## 3  3  1 3           1.250000
## 4  4  5 5           3.010399
## 5  5  4 5           2.358495
```

- What are the $k = 3$ nearest neighbors to the test point?

- What is the fitted/predicted value at the test point?

If $p > 1$, KNN performance often improves if we scale the explanatory variables

- Divide by standard deviation, so the rescaled variable has standard deviation 1
- Consider an example predicting based on two variables:
    - `tax`: full-value property-tax rate per $10,000
    - `dis`: weighted mean of distances to five Boston employment centres

```
# train/test split
set.seed(76520)
train_inds <- caret::createDataPartition(Boston$nox, p = 0.8)
train_boston <- Boston %>% slice(train_inds[[1]])
test_boston <- Boston %>% slice(-train_inds[[1]])

# rescale.  Here I rescale based on just training set data standard deviations,
# but actually it's ok to rescale based on combined train/test standard deviations
scale_sds <- train_boston %>%
  summarize(
    dis_sd = sd(dis),
    tax_sd = sd(tax)
  )
scale_sds
```

```
##     dis_sd   tax_sd
## 1 2.106811 167.5882
```

```
train_boston_scaled <- train_boston %>%
  mutate(
    dis = dis / scale_sds$dis_sd,
    tax = tax / scale_sds$tax_sd
  )
train_boston_scaled %>%
  summarize(
    dis_sd = sd(dis),
    tax_sd = sd(tax)
  )
```

```
##   dis_sd tax_sd
## 1      1      1
```

```
test_boston_scaled <- test_boston %>%
  mutate(
    dis = dis / scale_sds$dis_sd,
    tax = tax / scale_sds$tax_sd
  )
```

```
# KNN fit from scaled data
knn_fit_scaled <- train(
  form = nox ~ dis + tax,
  data = train_boston_scaled,
  use.all = FALSE,
  method = "knn",
  trControl = trainControl(method = "cv"),
  tuneGrid = data.frame(k = 5)
)

# KNN fit from original data
knn_fit_orig <- train(
  form = nox ~ dis + tax,
  data = train_boston,
  use.all = FALSE,
  method = "knn",
  trControl = trainControl(method = "cv"),
  tuneGrid = data.frame(k = 5)
)
```

```
# test set RMSE, scaled data fit
(test_boston_scaled$nox - predict(knn_fit_scaled, newdata = test_boston_scaled))^2 %>%
  mean() %>%
  sqrt()
```

```
## [1] 0.03893054
```

```
# test set RMSE, original data fit
(test_boston$nox - predict(knn_fit_orig, newdata = test_boston))^2 %>%
  mean() %>%
  sqrt()
```

```
## [1] 0.04375844
```

Actually, we could have had `caret` do the scaling for us by passing in a `preProcess = "scale"` argument to `train`:

```
# KNN fit from scaled data, but caret does the scaling
knn_fit_scaled_by_caret <- train(
  form = nox ~ dis + tax,
  data = train_boston, # note I'm giving train my original data frame
  use.all = FALSE,
  method = "knn",
  preProcess = "scale", # this is the only new line
  trControl = trainControl(method = "none"),
  tuneGrid = data.frame(k = 5)
)

# test set RMSE, scaled data fit, scaling done by caret
# note that since caret is handling scaling, I give it my original data for prediction
(test_boston$nox - predict(knn_fit_scaled_by_caret, newdata = test_boston))^2 %>%
  mean() %>%
  sqrt()
```

```
## [1] 0.03893054
```

Curse of Dimensionality: KNN performance degrades relatively quickly as the number of explanatory variables p increases

- Degradation in performance affects all methods, but affects non-parametric methods more
  - Parametric models assume a restricted parametric form for f and are trying to learn only a few parameters
  - Non-parametric methods are trying to learn the functional form. This is more difficult in higher dimensions
  - You will have a homework problem about this.

To explore this, I pulled out the top 10 explanatory variables in the Boston data set that were most correlated with `nox`, and sorted them in decreasing order of (absolute value of) correlation:

```
vars_to_include
```

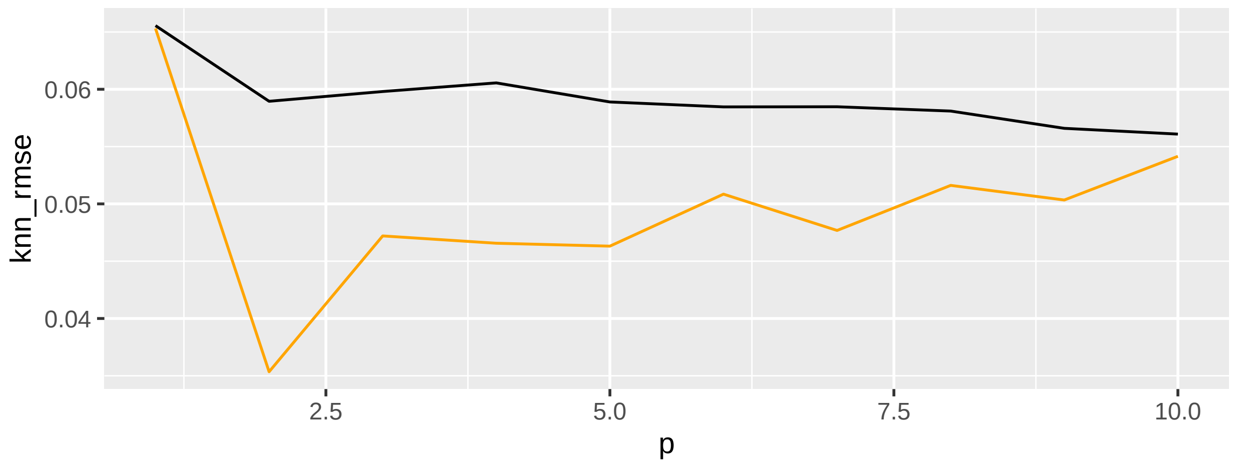
```
##    x_var    cor_nox
## 1    dis -0.7692301
## 2  indus  0.7636514
## 3    age  0.7314701
## 4    tax  0.6680232
## 5    rad  0.6114406
## 6  lstat  0.5908789
## 7     zn -0.5166037
## 8   medv -0.4273208
## 9   crim  0.4209717
## 10 black -0.3800506
```

I then fit a sequence of KNN models (k = 10) and linear models where each explanatory variable entered with a degree 2 polynomial term (no interactions). A model with p features used the p features that were most correlated with `nox`. Here are the RMSE for each of these models:

```
rmse_results
```

```
##     p   knn_rmse    lm_rmse
## 1   1 0.06530539 0.06555301
## 2   2 0.03535338 0.05895207
## 3   3 0.04720556 0.05980434
## 4   4 0.04656691 0.06055764
## 5   5 0.04631183 0.05889248
## 6   6 0.05085073 0.05846330
## 7   7 0.04768184 0.05847480
## 8   8 0.05161320 0.05809841
## 9   9 0.05033840 0.05659314
## 10 10 0.05415012 0.05608830
```

```
ggplot(data = rmse_results, mapping = aes(x = p)) +
  geom_line(mapping = aes(y = knn_rmse), color = "orange") +
  geom_line(mapping = aes(y = lm_rmse))
```


knn_rmse
p