Stats 401: Homework 3

Miles Chen

Overview

In this homework assignment, you will manually create some basis splines and try to fit them to some synthetic data. Once we are done with the basis splines, we will then model some real data with a Generalized Additive Model (using splines). [Week 4 content]

After that, I will lead you through coding up algorithms for a Naive Bayes Classifier and a k-nearest neighbors classifier. We will apply your coded algorithms to the iris dataset. [Week 5 content]

As always, the hope and goal of the homework assignment is to give you a deeper understanding of how each of these tools work.

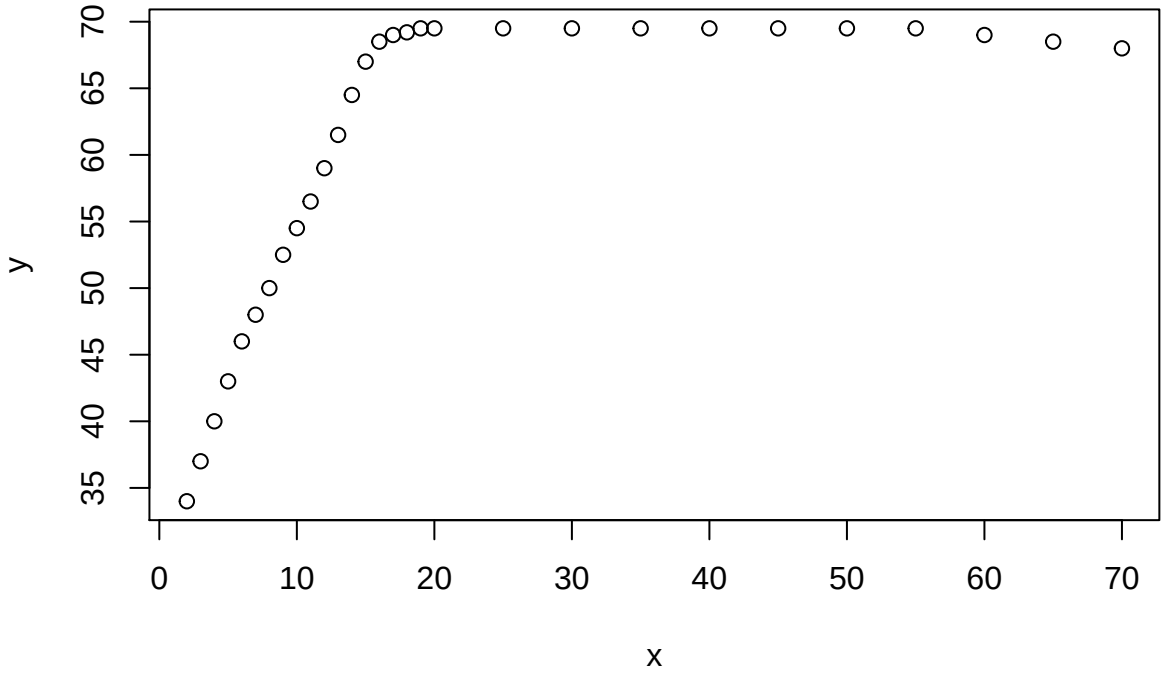
1) Cubic Basis Splines

A cubic basis spline allows us to fit a series of cubic functions in a way that is smooth and continuous.

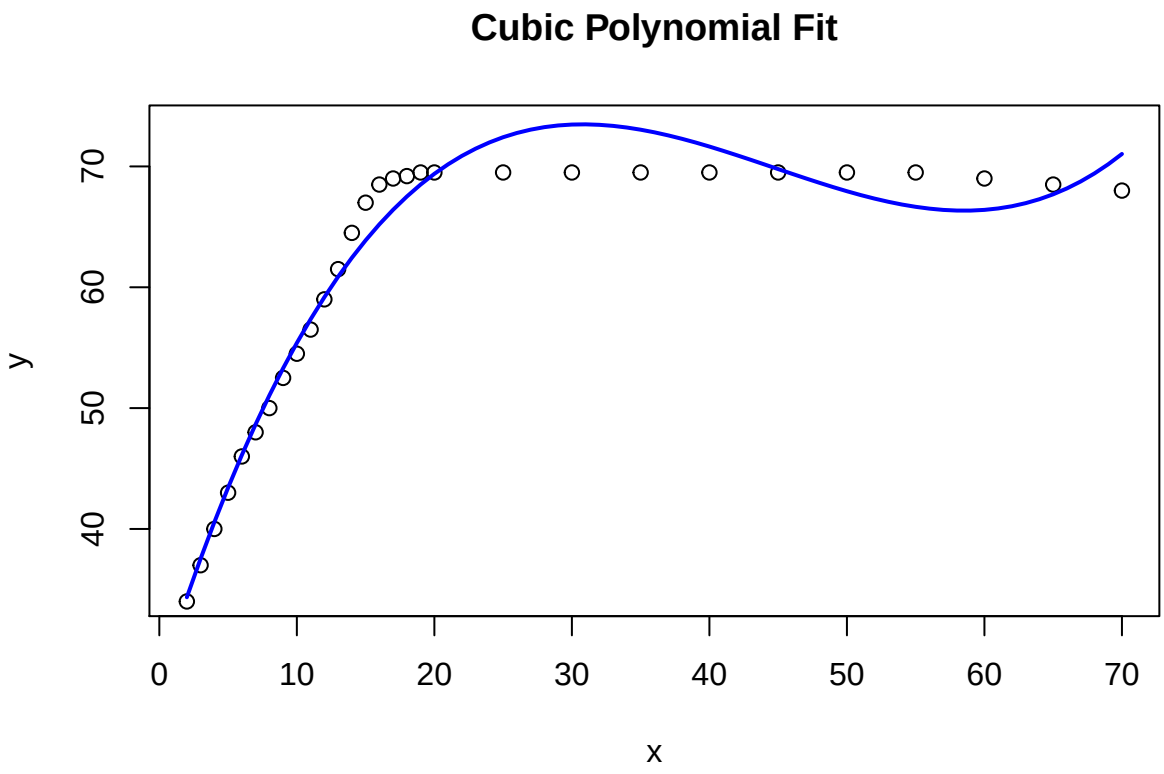
It is similar to least-squares polynomial regression in the way it selects coefficients – that is, it tries to minimize the total sum of squares of residuals. The difference is that the basis spline will have knots.

Follow the instructions outlined below to create a cubic basis spline with two knots manually. In a real-world seting, I do not recommend manual creation of basis splines. I recommend using R's spline functions.

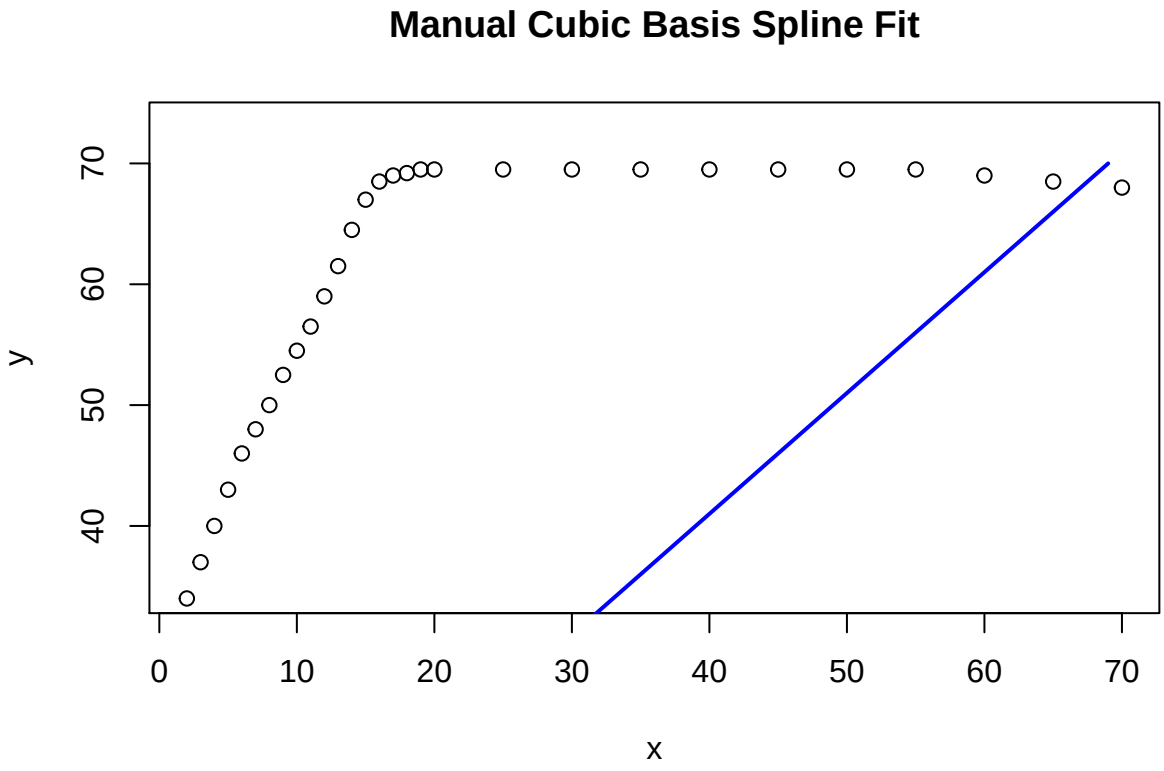
```
# Here is our data. We will pretend y is the height of a person, and x is his age.
y <- c(34, 37, 40, 43, 46, 48, 50, 52.5, 54.5, 56.5, 59, 61.5, 64.5, 67, 68.5,
       69, 69.2, rep(69.5, 9), 69, 68.5, 68)
x <- c(2:20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70)
plot(x, y) # the plot shows that there is not a linear relationship between x and y
```




```
## We begin doing regular polynomial regression with a cubic function.
## We define x_2 and x_3 as the squares and cubes of the x vector
x_2 <- x^2
x_3 <- x^3

## We perform least squares regression with the cubic fit, and extract the coefficients
cubic_fit <- lm(y ~ x + x_2 + x_3)
cubic_cf <- cubic_fit$coefficients

## To plot our fit, we create a vector xnew, and the predicted values yhat
xnew <- seq(2,70)  # vector of x values to plot
yhat <- cubic_cf[1] + cubic_cf[2] * xnew + cubic_cf[3] * xnew ^ 2 +
  cubic_cf[4] * xnew ^ 3
plot(x, y, ylim = range(yhat), main = "Cubic Polynomial Fit")
# result of the fitted cubic basis spline
lines(xnew, yhat, type = "l", lwd = 2, col = "blue")
```




```
## We will fit a basis spline with two knots, one at 15, and another at 20
## We must create two new vectors of x.
## The first, will be x_a, which is effectively (x-15)^3, but all negative values
## are made to be zero
x_a <- (x - 15) ^ 3   # the cubic function that begins at xi = 15
x_a[ x_a < 0 ] <- 0   # we force all the values below 15 (x_a is negative) to be 0

## create another vector of x for the knot at x=20


## fit a least squares regression model on all of your x vectors
basis_fit <- # ..


## Plot your fitted spline
s <- seq(2,70)  # vector of x values to plot
hs_a <- (s-15)^3 # must make a vector of values to plot for x_a
hs_a[s<15] <- 0
hs_b <- # ...
yhat <- # ...
plot(x, y, ylim = range(yhat), main = "Manual Cubic Basis Spline Fit")
lines(s, yhat, type = "l", lwd = 2, col = "blue")
```



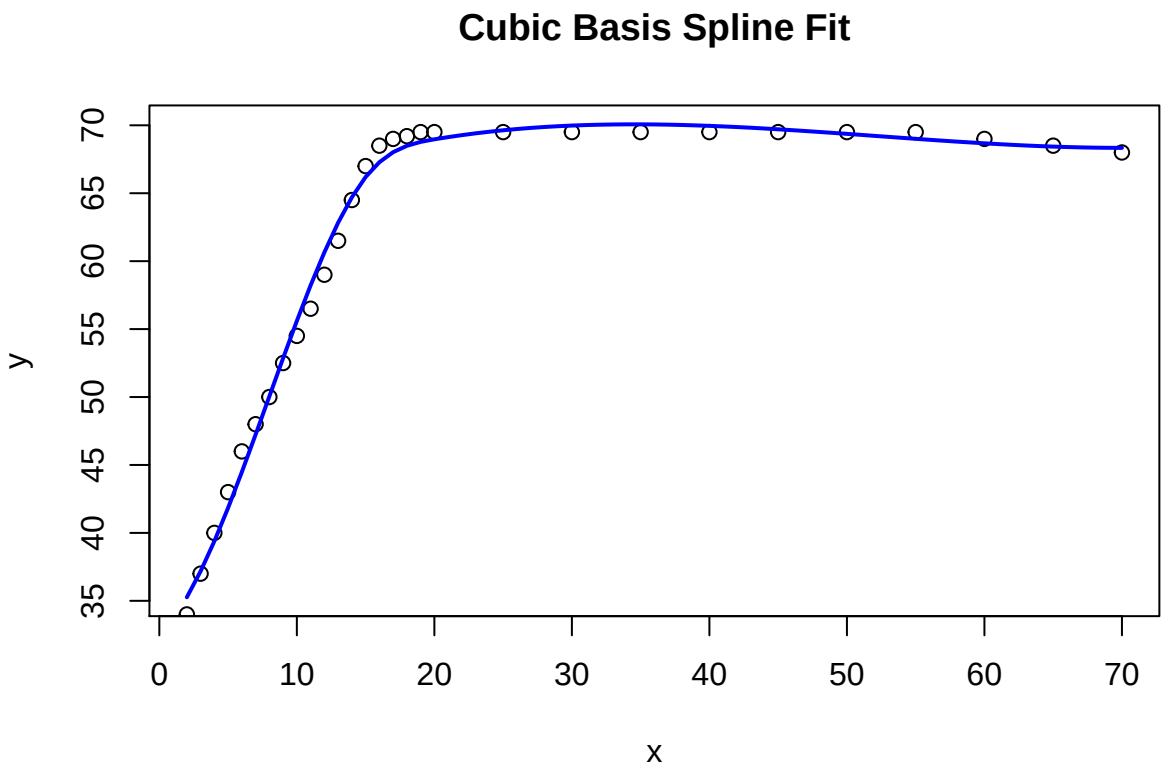

```
# what is your predicted value when x = 50?



## Compare results to using R's built-in basis spline function bs() in library(gam)
library(gam)
```

```
## Loading required package: splines
```

```
## Loading required package: foreach
```

```
## Loaded gam 1.20
```

```
bs_fit <- lm(y ~ bs(x, knots = c(15, 20)))
s <- seq(2,70)
yhat <- predict(bs_fit, newdata = data.frame(x = xnew))
plot(x, y, ylim = range(yhat), main = "Cubic Basis Spline Fit")
lines(s, yhat, type = "l", lwd = 2, col = "blue")
```




```
# what is your predicted value when x = 50?
```

2) Generalized Additive Model

We will now use `library(gam)` to create a generalized additive model for the data `Auto`, using gas mileage `mpg` as the response variable.

We start by learning a bit about the data.

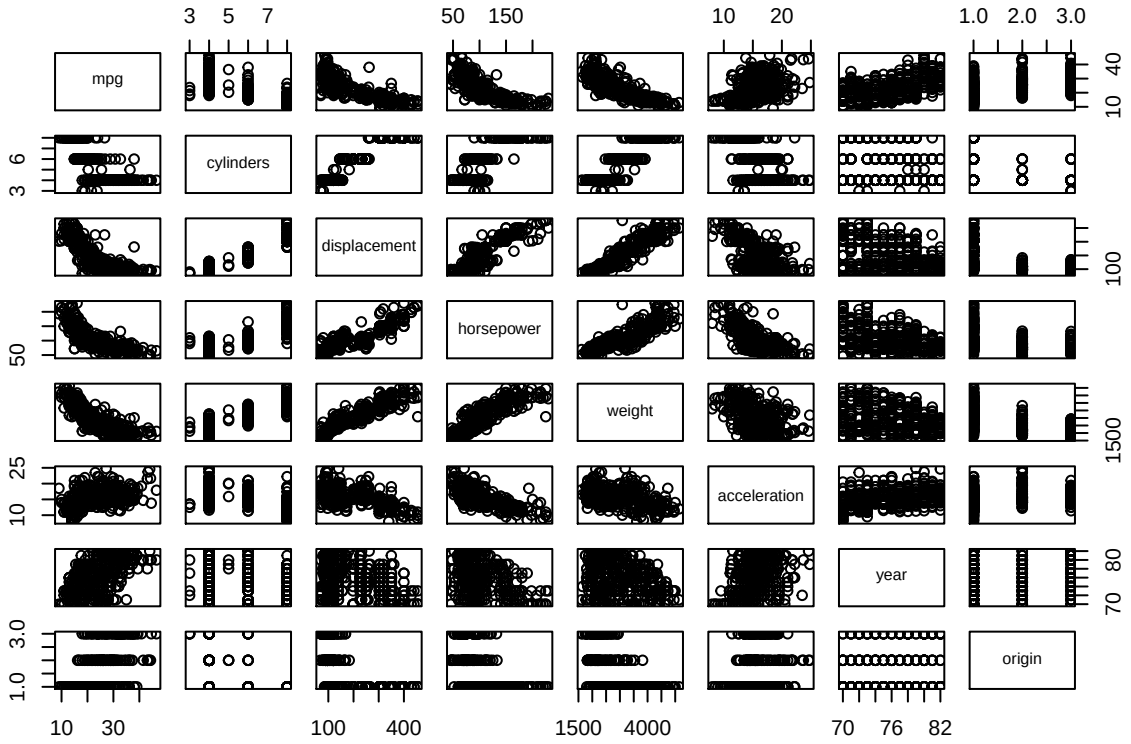
```
library(gam)
library(ISLR)
data(Auto)
names(Auto)
```

```
## [1] "mpg"          "cylinders"    "displacement" "horsepower"   "weight"
## [6] "acceleration" "year"         "origin"       "name"
```

```
# help(Auto) # You should read this
```

First things first. Let's make some plots and see how they are related to mpg.

```
# We create a pairwise plot to see all the variable relationships
# I removed the 9th column because it contains vehicle names, which will not be helpful
# This plot is too small here, so run it in your console and make it large
pairs(Auto[,-9])
```



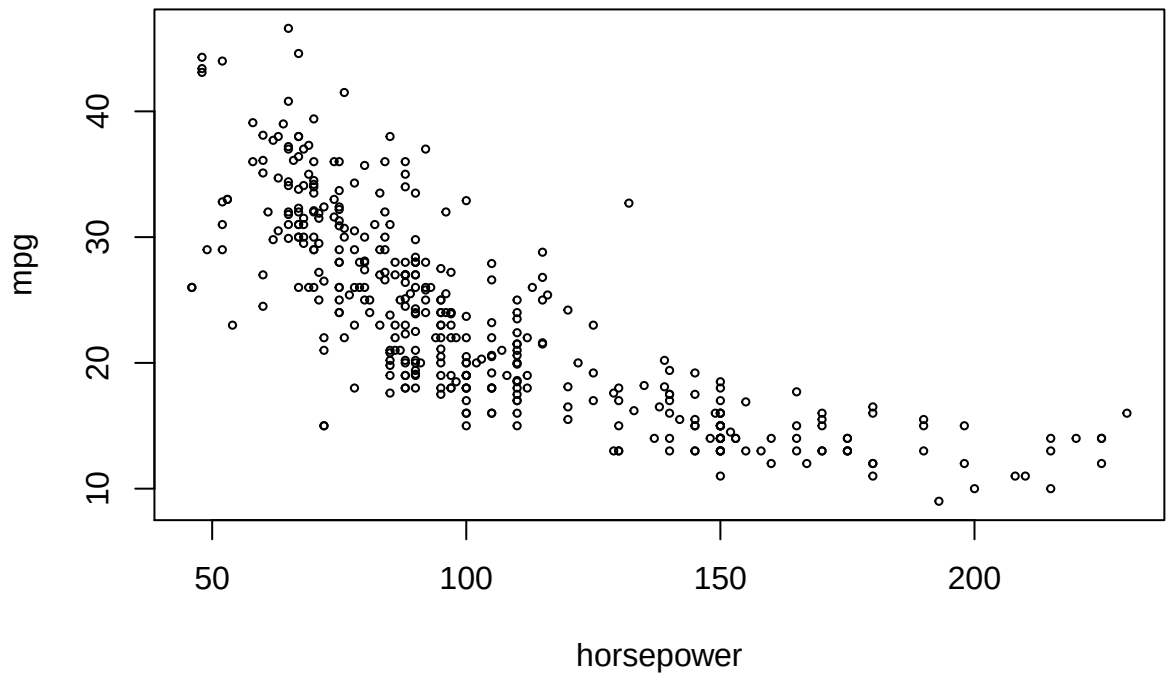
From the pair-wise variable plot, we see that there is a non-linear relationship between mpg and the variables `displacement`, `horsepower`, `weight`, and possibly `acceleration`.

However, the variables `displacement`, `horsepower`, and `weight` seem to have very high colinearity, so including all three variables might not be beneficial to our model. We can compare models using `anova()` as we add additional variables.

The variable origin is categorical (not ordinal), so if we want to include them we have to make dummy variables.

I will begin by exploring the replationship between `horsepower` and `mpg`.

```
plot(mpg ~ horsepower, data = Auto, cex = 0.5)
```



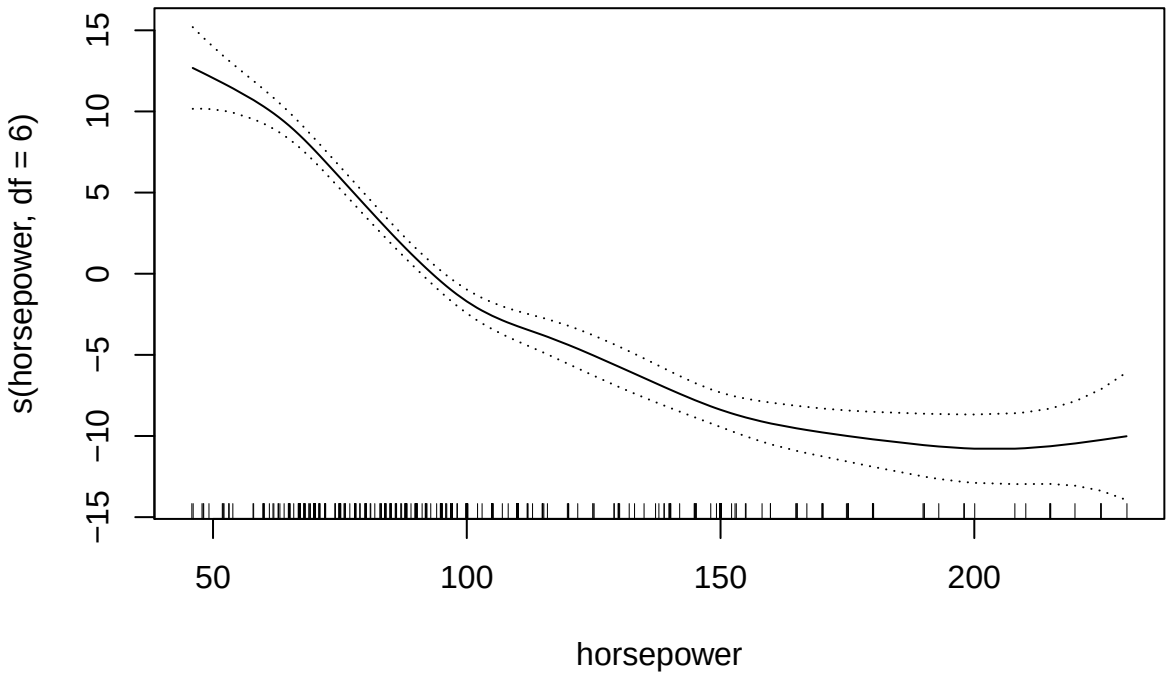
The plot shows a relationship that is clearly not linear. We may choose to explore a polynomial relationship or may consider using splines.

I start by fitting a few GAMs - a quadratic (poly 2) function, and splines with 2, 4, and 6 degrees of freedom. The choice of degrees of freedom is arbitrary.

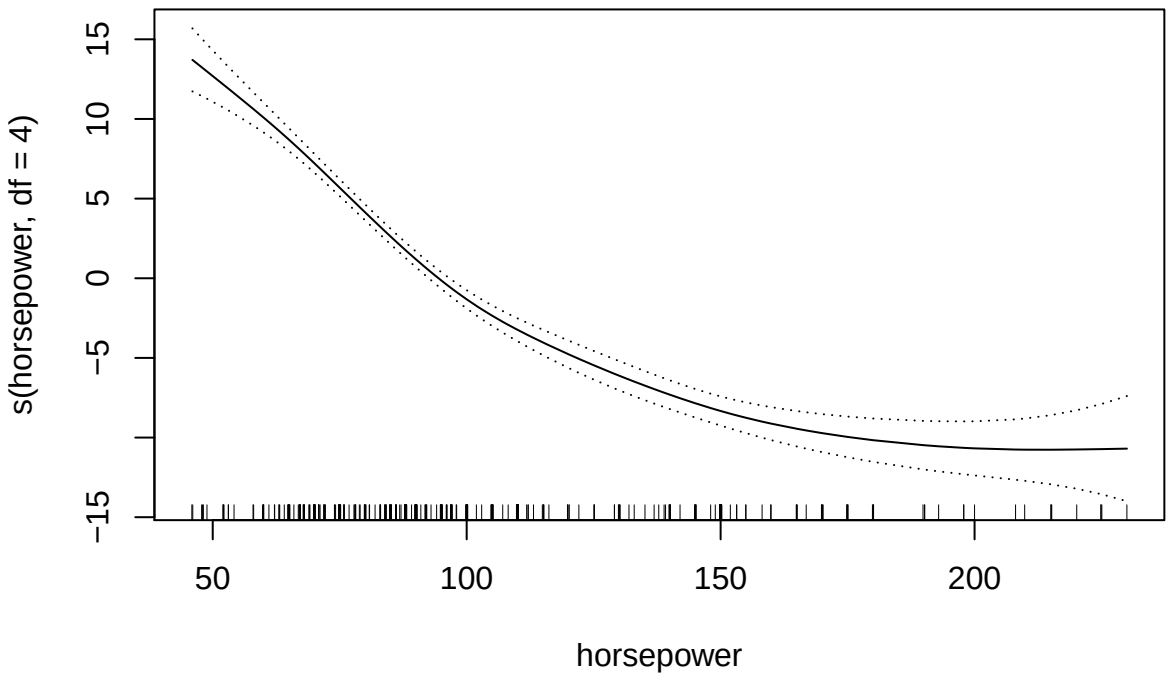
```
poly2_fit <- gam(mpg ~ poly(horsepower, 2) , data = Auto)
spline2_fit <- gam(mpg ~ s(horsepower, df = 2) , data = Auto)
spline4_fit <- gam(mpg ~ s(horsepower, df = 4) , data = Auto)
spline6_fit <- gam(mpg ~ s(horsepower, df = 6) , data = Auto)
anova(poly2_fit, spline2_fit, spline4_fit, spline6_fit)
```

```
## Analysis of Deviance Table
##
## Model 1: mpg ~ poly(horsepower, 2)
## Model 2: mpg ~ s(horsepower, df = 2)
## Model 3: mpg ~ s(horsepower, df = 4)
## Model 4: mpg ~ s(horsepower, df = 6)
##   Resid. Df Resid. Dev          Df Deviance  Pr(>Chi)
## 1       389     7442.0
## 2       389     7638.5 -8.8147e-05  -196.48 3.537e-08 ***
## 3       387     7267.7  2.0002e+00   370.82 4.474e-05 ***
## 4       385     7127.5  1.9997e+00   140.15    0.0227 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

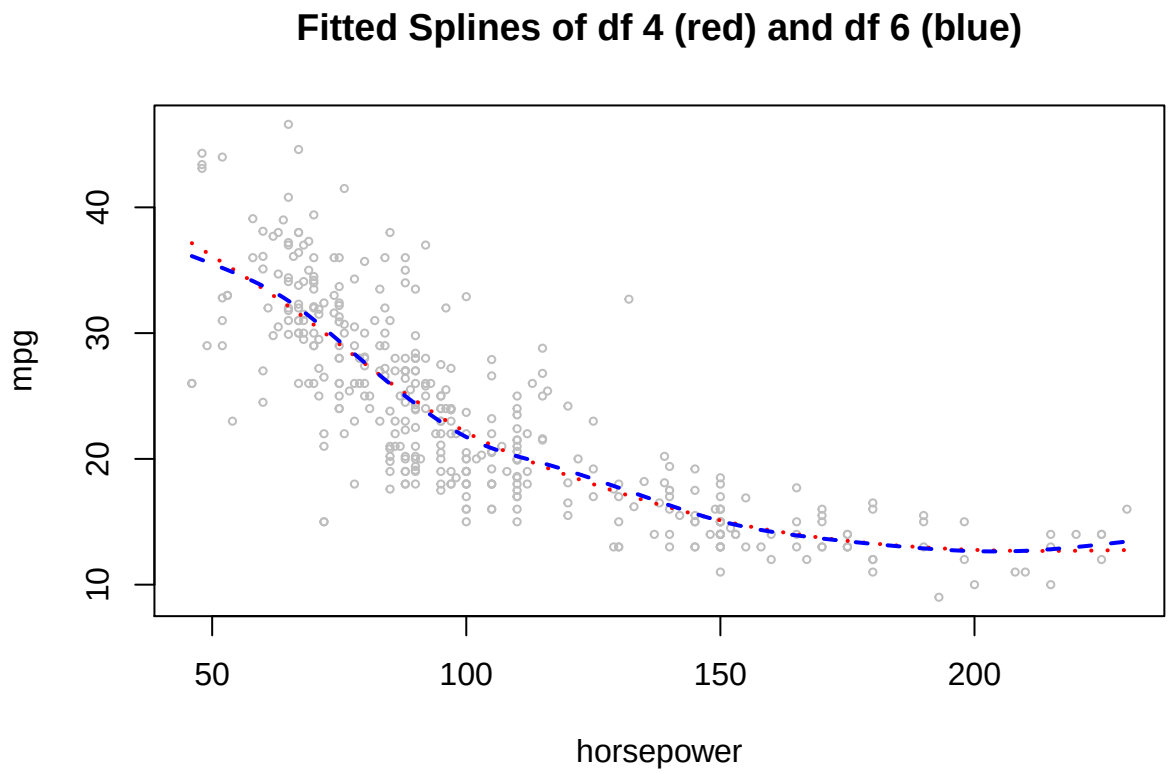
```r
plot(spline6_fit, se = TRUE)
```




```r
plot(spline4_fit, se = TRUE)
```




```
s <- seq(min(Auto$horsepower), max(Auto$horsepower), by = 1)
test_cases <- data.frame(horsepower = s) # a new data frame with just one variable
spline2_predict <- predict(spline2_fit, newdata = test_cases) # predicted values
spline4_predict <- predict(spline4_fit, newdata = test_cases)
spline6_predict <- predict(spline6_fit, newdata = test_cases)
plot(mpg ~ horsepower, data = Auto, cex = 0.5, col = "gray", main = "Fitted Splines of df 4 (red) and d
lines(test_cases$horsepower, spline4_predict, col = "red", lwd = 2, lty = 3)
lines(test_cases$horsepower, spline6_predict, col = "blue", lwd = 2, lty = 2)
```

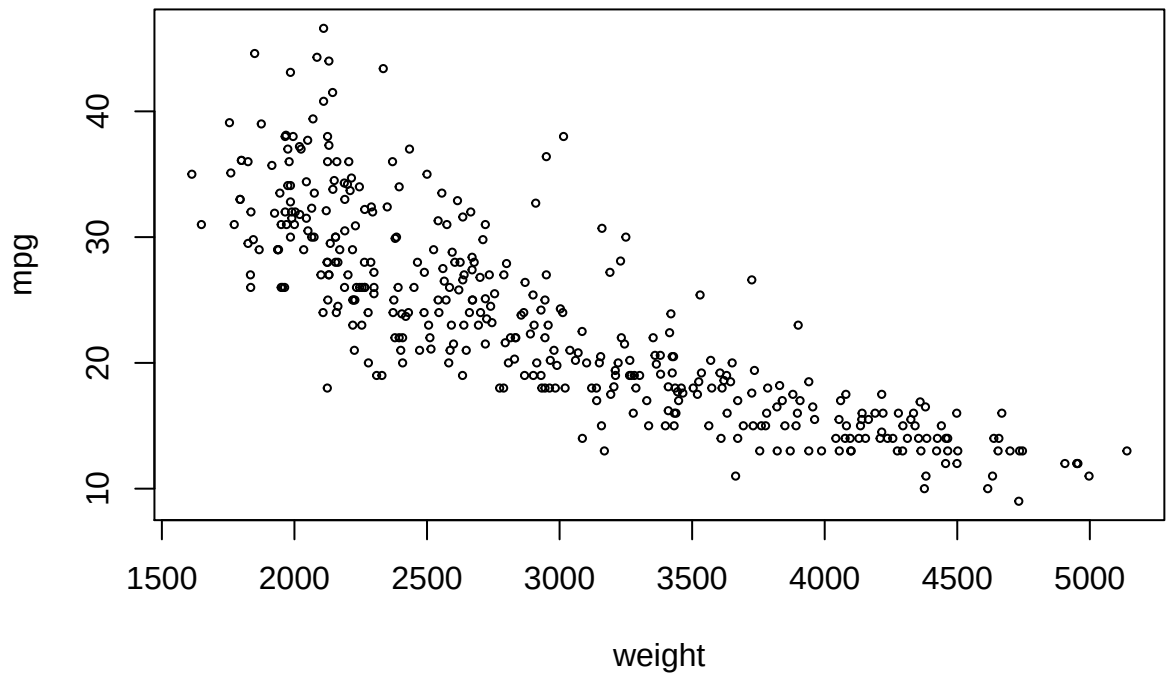


The ANOVA analysis shows that using a spline out-performs the quadratic polynomial. It also shows difference between the spline with 6 df and 4 df is statistically significant. The plots, however, do not seem to show much of a difference between 4 df and 6 df.

Based on the anova results, I'll choose a spline with 6 df, but if someone decides to use the 4df spline to keep the shape simpler, they could justify their reasoning. We could compare our choice to other splines with fewer or more degrees of freedom as well. The exact choice of degrees of freedom will depend on the researcher's opinion. Keep in mind, that the goal is not to find the exact relationship between the x and y variable, but to be able to create an additive model that relates each variable to the response with the flexibility that a spline offers.

Now let's begin exploring the relationship with an additional variable. We look at the relationship between `mpg` and `weight`.

```
plot(mpg ~ weight, data = Auto, cex = 0.5)
```



Like before, I start by fitting a few GAMs - a quadratic (poly 2) function, and splines with 2, 4, and 6 degrees of freedom. The choice of degrees of freedom is arbitrary.

```
poly2_fit <- gam(mpg ~ poly(weight, 2) , data = Auto)
spline2_fit <- gam(mpg ~ s(weight, df = 2) , data = Auto)
spline4_fit <- gam(mpg ~ s(weight, df = 4) , data = Auto)
spline6_fit <- gam(mpg ~ s(weight, df = 6) , data = Auto)
anova(poly2_fit, spline2_fit, spline4_fit, spline6_fit)
```

```
## Analysis of Deviance Table
##
## Model 1: mpg ~ poly(weight, 2)
## Model 2: mpg ~ s(weight, df = 2)
## Model 3: mpg ~ s(weight, df = 4)
## Model 4: mpg ~ s(weight, df = 6)
##   Resid. Df Resid. Dev          Df Deviance  Pr(>Chi)
## 1       389     6784.9
## 2       389     6841.0 -5.2063e-05  -56.095 2.216e-06 ***
## 3       387     6759.5  2.0001e+00   81.451   0.09662 .
## 4       385     6709.1  2.0000e+00   50.393   0.23554
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

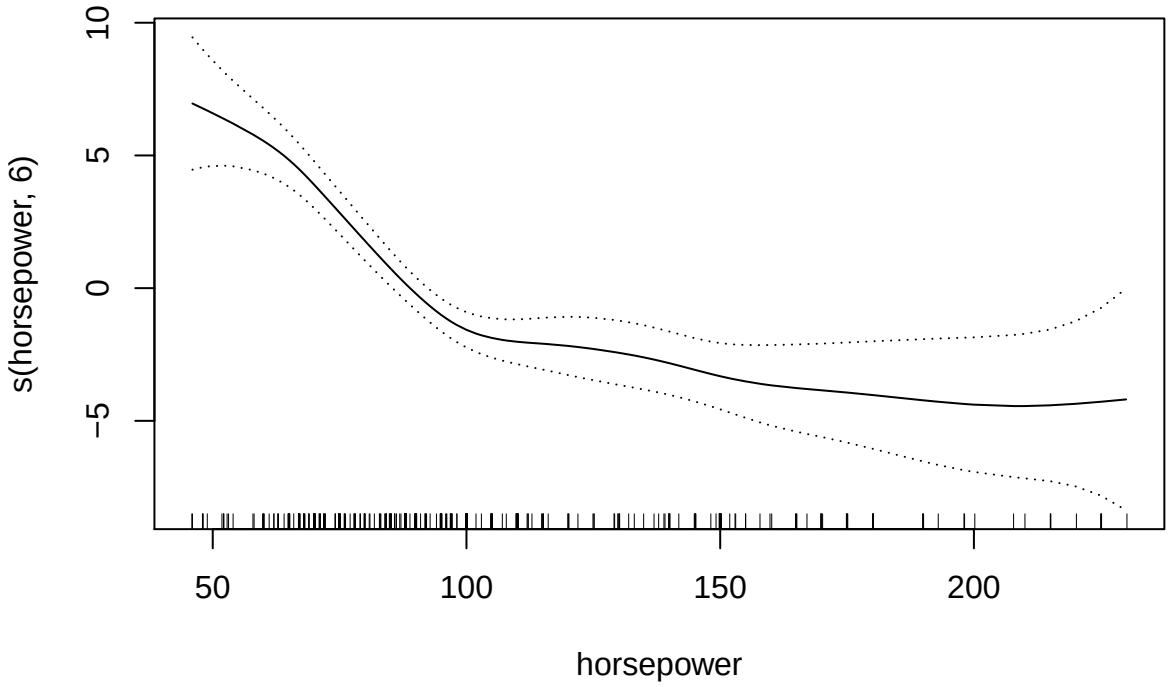
The anova analysis seems to indicate that a spline with 2 df is the fit we should choose.

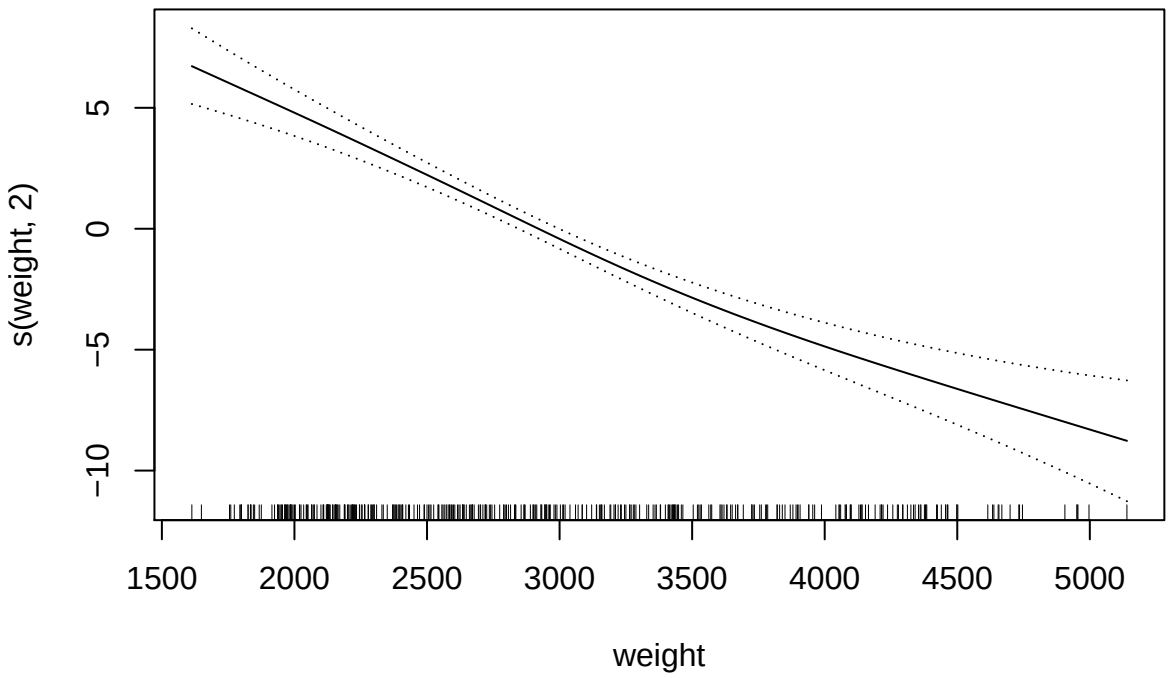
I create a gam with both variables.

```r
gam_fit <- gam(mpg ~ s(horsepower, 6) + s(weight, 2), data = Auto)
summary(gam_fit)
```

```
##
## Call: gam(formula = mpg ~ s(horsepower, 6) + s(weight, 2), data = Auto)
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -10.6915  -2.3129  -0.1792   1.7568  14.3039
##
## (Dispersion Parameter for gaussian family taken to be 15.0831)
##
##     Null Deviance: 23818.99 on 391 degrees of freedom
## Residual Deviance: 5776.845 on 383.0002 degrees of freedom
## AIC: 2187.065
##
## Number of Local Scoring Iterations: NA
##
## Anova for Parametric Effects
##                   Df  Sum Sq Mean Sq F value    Pr(>F)
## s(horsepower, 6)   1 14607.1 14607.1  968.44 < 2.2e-16 ***
## s(weight, 2)       1  1549.9  1549.9  102.76 < 2.2e-16 ***
## Residuals        383  5776.8    15.1
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Anova for Nonparametric Effects
##                   Npar Df Npar F     Pr(F)
## (Intercept)
## s(horsepower, 6)       5 12.275 4.715e-11 ***
## s(weight, 2)           1  6.526   0.01102 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
plot(gam_fit, se = TRUE)
```

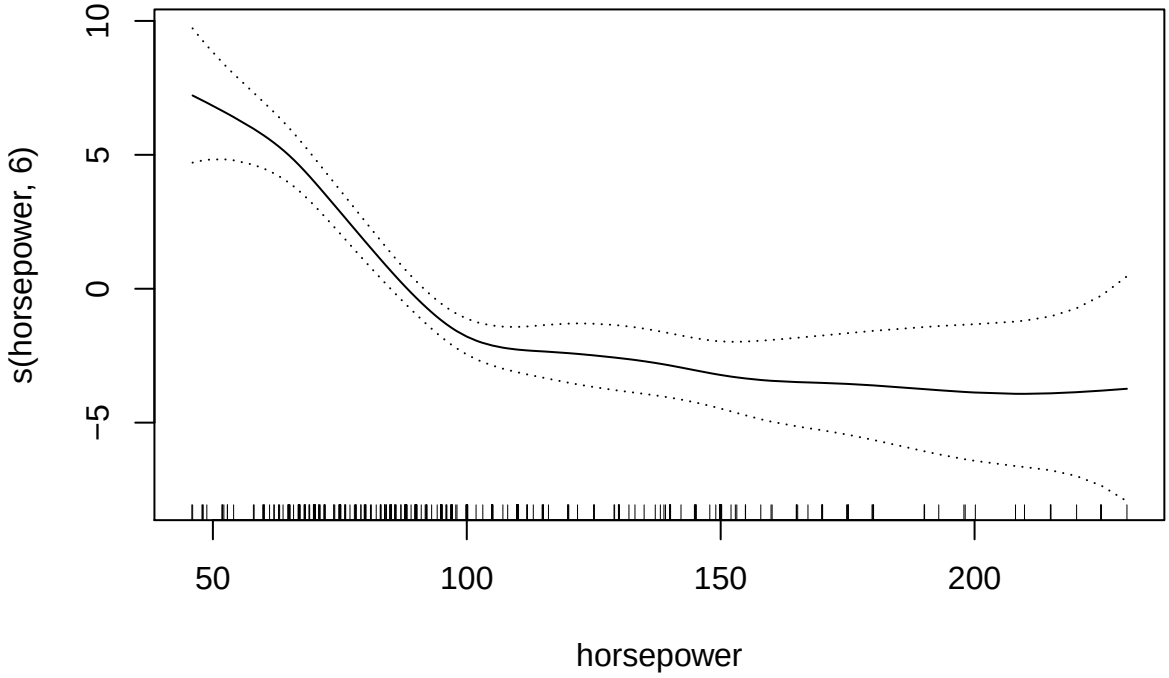

s(horsepower, 6)
10
5
0
−5
50
100
150
200
horsepower

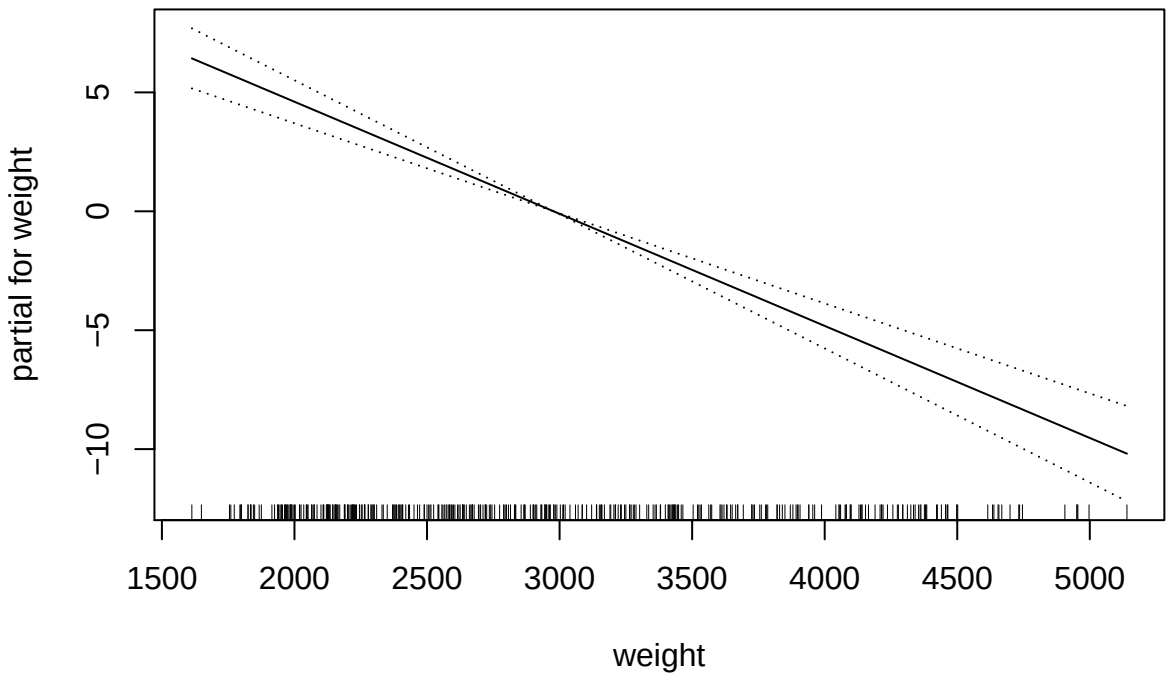


The summary of the fitted GAM model indicates that both variables have a significant relationship with `mpg`.

Inspection of the plots, however, seems to show that perhaps the relationship between `weight` and `mpg` becomes linear if we have already included `horsepower` in the model. We can check this by fitting another gam where `weight` is added as a linear variable.

```
gam_fit2 <- gam(mpg ~  s(horsepower,6) + weight, data = Auto)
plot(gam_fit2, se = TRUE)
```


s(horsepower, 6)
10
5
0
−5
50
100
150
200
horsepower




```
anova(gam_fit2, gam_fit)
```

```
## Analysis of Deviance Table
##
## Model 1: mpg ~ s(horsepower, 6) + weight
## Model 2: mpg ~ s(horsepower, 6) + s(weight, 2)
##   Resid. Df Resid. Dev      Df Deviance Pr(>Chi)
## 1       384     5858.3
## 2       383     5776.8 0.99995   81.498   0.0201 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The ANOVA table compares the two models: `gam_fit2` uses a linear fit for weight and `gam_fit` uses a spline fit for weight. Comparing these two models provides a p-value of 0.0201. While this is not as extreme as a value on the order of 10^-16, it is still a small value, and we will conclude that the relationship between weight and mpg is better modeled as a non-linear relationship.

Continue the data analysis to model the relationship between mpg and the other variables in the Auto dataset. Comment on what you find

Your resulting model does not need to be perfect, and I do not want you to stress over what is right versus wrong. Data analysis is very open-ended. I would encourage at least exploring the possibility of including `displacement` and/or `acceleration` as variables in the model. Your ANOVA comparison may reveal that some of these variables are not worth including.

Check the example in chapter 7 of ISLR - pages 294-297 for guidance.

3) Naive Bayes Classifier for Iris data

In this section, we will write a function that performs Naive Bayes classification for the iris dataset. See `help(iris)` if you are not familiar with the data. The function will output probability estimates of the species for a test case.

The function will accept three inputs: a row matrix for the x values of the test case, a matrix of the training data x values, and the labels of the training data.

The function will create the probability estimates based on the training data it has been provided. Use a Gaussian model with estimates for the mean and standard deviation based on the training data provided.

Overview

The Naive Bayes classifier outputs a probability. The probability is a fraction, where the numerator is the **likelihood of the data given a particular class** $\times$ **the prior probability of the class**. The denominator is the total probability of the data, but is easily calculated by summing up all the numerator values across the classes.

$$P(Y_n = C_i|X_n, \mathbf{X}, y) = \frac{P(X_n \cap Y_n = C_i|\mathbf{X}, y)}{P(X_n|\mathbf{X}, y)} = \frac{P(X_n|Y_n = C_i, \mathbf{X}, y)P(Y_n = C_i|\mathbf{X}, y)}{\sum_{i=1}^{K} P(X_n \cap Y_n = C_i|\mathbf{X}, y)}$$

$\mathbf{X}$ represents the matrix of training data X, and y is the class labels of the training data.

$$P(Y_n = C_i|X_n, \mathbf{X}, y) = \frac{P(X_n|Y_n = C_i, \mathbf{X}, y)P(Y_n = C_i|\mathbf{X}, y)}{\sum_{i=1}^{K} P(X_n|Y_n = C_i, \mathbf{X}, y)P(Y_n = C_i|\mathbf{X}, y)}$$

The **prior probability of the class** will be based on the training data. In our scenario, we will count how many observations belong to class i, and divide by the total observations in the training data.

The **likelihood of the data for a particular class** will depend on the model we use and the parameters estimated from the training data. In our scenario, we will use a Naive (independent) normal model. This means that the total joint probability of our data is the product of a few univariate normal densities (as opposed the single result of a multivariate normal density).

A little more info on the **Naive** part of the Naive bayes classifier: Take a look at the likelihood term: $P(X_n|Y_n = C_i, X, y)$. There could be multiple X variables, so observation n, may have a likelihood of:

$$P(X_{1n}, X_{2n}, X_{3n}|Y_n = C_i, \mathbf{X}, y)$$

Joint probabilities can be found by multiplying conditional probabilities:

$$P(X_{1n}, X_{2n}, X_{3n}|Y_n = C_i, \mathbf{X}, y) = P(X_{1n}|Y_n = C_i, \mathbf{X}, y) \times P(X_{2n}|X_{1n}, Y_n = C_i, \mathbf{X}, y) \times P(X_{3n}|X_{1n}, X_{2n}, Y_n = C_i, \mathbf{X}, y)$$

The conditional probability of variable X_2 given X_1 may be difficult to find or calculate. Similarly, finding the conditional probability of X_3 given variables X_2 and X_1 can be difficult to find.

The **Naive** part is to assume that X_{1n} and X_{2n} and X_{3n} are all independent of each other. The above thus reduces to just the product of each variable's probability, conditioned on each class label:

$$P(X_{1n}, X_{2n}, X_{3n}|Y_n = C_i, \mathbf{X}, y) = P(X_{1n}|Y_n = C_i, \mathbf{X}, y) \times P(X_{2n}|Y_n = C_i, \mathbf{X}, y) \times P(X_{3n}|Y_n = C_i, \mathbf{X}, y)$$

To condition each class using our training data, we will use the sample's summary statistics. The parameter estimates of each univariate normal density will just be the sample mean and the standard deviation of the sample for each X variable.

Naive Bayes Classifier for Iris data

Task: Write a function that performs Naive Bayes classification for the iris data. The function will output probability estimates of the species for a test case.

The function will accept three inputs: a row matrix for the x values of the test case, a matrix of x values for the training data, and a vector of class labels for the training data.

The function will create the probability estimates based on the training data it has been provided.

Within the function use a Gaussian model and estimate the mean and standard deviation of the Gaussian populations based on the training data provided. (Hint: You have 24 parameters to estimate: the mean and standard deviation of each of the 4 variables for each of the three species. With the naive assumption, you do not have to estimate any covariances.)

The data has three classes of flowers and four input variables.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
# we would call the funtion like: iris_nb(test_case1, train)

iris_nb <- function(test, trainx, trainy){
  # the function will accept three arguments:
  # test, which will be the values of the four input variables
  # trainx, which will be the data.frame of the training data x values
  # trainy, which will be the class labels of the training data
  test <- as.matrix(test) # this converts the test case from a data.frame to a matrix

  # Using the training data, calculate the mean and standard
  # deviation for each variable for each class of flower.
  # There will be a total of 24 summary values, that will be used to calculate the
  # normal distributions

  # one way is to manually calculate each of these values...
  # mean sepal length of class setosa
  m_sep_l_set <- mean(train$Sepal.Length[train$Species == "setosa"])
  # sd sepal length of class setosa
  s_sep_l_set <- sd(train$Sepal.Length[train$Species == "setosa"])
  # ...
```

```
  # another way uses dplyr. Either method works.
  summary <- train %>% group_by(Species) %>% summarise(msl = mean(Sepal.Length), ssl = sd(Sepal.Length)

  # Also, calculate the relative frequency of each species in the training data
  # these will serve as the 'prior' probabilities of each class
  pr_set <- # proportion of class setosa in the training data
  pr_ver
  pr_vir

  # Now that we have the summary statistics, we can calculate the numerator
  # for each class.
  # This will be the product of the likelihoods of each variable.
  # The Naive Part is that we are assuming each variable is independent,
  # So that the joint probability is just the product of probabilities.
  # To make our lives easier, we will use the dnorm() function to calculate
  # normal probability densities.

  # Numerator for the Setosa Class
  # test[1] corresponds to the first value in the test vector: the sepal length
  # dnorm(test[1], mu_sl_set, sd_sl_set) is the probability of observing that
  # test[1] value based on the summary stats we found for class Setosa's Sepal Length.
  num_set <- dnorm(test[1], m_sep_l_set, s_sep_l_set) *
             dnorm(test[2], ...) *
             dnorm(test[3], ...) *
             dnorm(test[4], ...) *
             pr_set
  # repeat for the other classes ...


  # The denominator will be the sum of the three numerator values
  denom <- sum(num_set, num_ver, num_vir)

  # output: a named vector of each probability.
  results <- round(c(Setosa = num_set / denom,
    Versicolor = num_ver / denom,
    Virginica = num_vir / denom), 6)

  return(results) # what the function outputs
}
```

```
### output should be a named vector that looks something like this:
## [these numbers are completely made up btw]
    setosa versicolor  virginica
 0.9518386  0.0255936  0.0225678
```

```
set.seed(1)
training_rows <- sort(c(sample(1:50, 40), sample(51:100, 40), sample(101:150, 40)))
training_x <- as.matrix(iris[training_rows, 1:4])
training_y <- iris[training_rows, 5]
```

```
# test cases
test_case_a <- as.matrix(iris[24, 1:4]) # true class setosa
test_case_b <- as.matrix(iris[73, 1:4]) # true class versicolor
test_case_c <- as.matrix(iris[124, 1:4]) # true class virginica

# class predictions of test cases
iris_nb(test_case_a, training_x, training_y)
```

Testing it out

```
## Error in mean(train$Sepal.Length[train$Species == "setosa"]): object 'train' not found
```

```
iris_nb(test_case_b, training_x, training_y)
```

```
## Error in mean(train$Sepal.Length[train$Species == "setosa"]): object 'train' not found
```

```
iris_nb(test_case_c, training_x, training_y)
```

```
## Error in mean(train$Sepal.Length[train$Species == "setosa"]): object 'train' not found
```

```
# should work and produce slightly different estimates based on new training data
set.seed(10)
training_rows2 <- sort(c(sample(1:50, 25), sample(51:100, 25), sample(101:150, 25)))
training_x2 <- as.matrix(iris[training_rows2, 1:4])
training_y2 <- iris[training_rows2, 5]

iris_nb(test_case_a, training_x2, training_y2)
```

```
## Error in mean(train$Sepal.Length[train$Species == "setosa"]): object 'train' not found
```

```
iris_nb(test_case_b, training_x2, training_y2)
```

```
## Error in mean(train$Sepal.Length[train$Species == "setosa"]): object 'train' not found
```

```
iris_nb(test_case_c, training_x2, training_y2)
```

```
## Error in mean(train$Sepal.Length[train$Species == "setosa"]): object 'train' not found
```

Naive Bayes with R

While instructive and education (I hope) to write your own NaiveBayes function, in practical settings, I recommend using the production ready code from some time-tested packages.

I've included some code for using the `naiveBayes()` function that is part of the `e1071` package. No need to modify anything. The results prediced by `naiveBayes()` should match the results from the function you wrote.

```r
# code provided. no need to edit. These results should match your results above.
library(e1071)
nb_model1 <- naiveBayes(training_x, training_y)
predict(nb_model1, newdata = test_case_a, type = 'raw')
```

```
##      setosa   versicolor    virginica
## [1,]      1 1.029887e-13 4.098385e-18
```

```r
predict(nb_model1, newdata = test_case_b, type = 'raw')
```

```
##             setosa versicolor  virginica
## [1,] 2.980587e-115  0.9034742 0.09652578
```

```r
predict(nb_model1, newdata = test_case_c, type = 'raw')
```

```
##             setosa versicolor virginica
## [1,] 6.078393e-136 0.09540725 0.9045928
```

```r
nb_model2 <- naiveBayes(training_x2, training_y2)
predict(nb_model2, newdata = test_case_a, type = 'raw')
```

```
##      setosa  versicolor    virginica
## [1,]      1 2.60018e-10 4.610354e-17
```

```r
predict(nb_model2, newdata = test_case_b, type = 'raw')
```

```
##             setosa versicolor  virginica
## [1,] 1.735964e-131  0.9195738 0.08042615
```

```r
predict(nb_model2, newdata = test_case_c, type = 'raw')
```

```
##             setosa versicolor virginica
## [1,] 3.400709e-149  0.1887116 0.8112884
```

4) K-nearest neighbors Classifier for the Iris data

Task: Write a classifier using the K-nearest neighbors algorithm for the iris data set. A nice feature of this dataset is that it has numeric predictors, which is necessary for the k-nearest neighbors classifier.

First write a function that will calculate the euclidean distance from a vector A (in 4-dimensional space) to another vector B (also in 4-dimensional space).

Use that function to find the k nearest neighbors to then make a classification. If there is a tie, use the 1-nearest neighbor to break the tie.

The function will accept four inputs: a row matrix for the x values of the test case, a data frame of the training data, and the k parameter.

The output to this function will be a single label.

```r
distance <- function(a, b){
  # I've doce this part for you
  dis <- sum( (a - b) ^ 2 )
  sqrt(dis)
}

iris_knn <- function(testx, trainx, trainy, k){
  # Write your code here

  # establish n as the number of rows in the training data
  # create a vector d (of length n) that will contain the distances
  # from our test case x to the different training cases
  n <- nrow(train)
  d <- rep(NA, n)
  for(i in 1:n){
    # for every data point in the training set
    # calculate the distance and store it in d
    d[i] <- x# insert calculated distance
  }

  # attach the d column to the training data labels
  # Sort this resulting table in ascending order
  dist_table <- data.frame(d, train$Species)
  # look at help(order).
  # Also see: http://adv-r.had.co.nz/Subsetting.html#applications on ordering

  # select the top k rows
  top_k <- ...

  # with those top k rows, you can use table() to tabulate a count of labels
  tab <- table(top_k)

  # choose the label that has the largest count
  result <- which() # find which values equal the max value in the table

  # if results has a length of 1, then that is the label
  # if results has a length greater than 1, go back to the top_k table, and select
  # the nearest row as the label
  label <- ...

  return(label)
}
```

```r
iris_knn(test_case1, train, 5)
```

```
## Error in nrow(train): object 'train' not found
```

```r
iris_knn(test_case2, train, 5)
```

```
## Error in nrow(train): object 'train' not found
```

```
iris_knn(test_case3, train, 5)
```

```
## Error in nrow(train): object 'train' not found
```

I hope this exercise helps you better understand the inner workings of a knn algorithm.

KNN with R

Again, if you plan on using KNN in real-life, use a function from a package.

I've included some code for using the `knn()` function that is part of the `class` package. No need to modify anything. The results prediced by `knn()` should match the results from the function you wrote, including the misclassification of some of the test cases based on the training data.

```
library(class)
knn(train = training_x, cl = training_y, test = test_case_a, k = 5)
```

```
## [1] setosa
## Levels: setosa versicolor virginica
```

```
knn(train = training_x, cl = training_y, test = test_case_b, k = 5)
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

```
knn(train = training_x, cl = training_y, test = test_case_c, k = 5)
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

```
knn(train = training_x2, cl = training_y2, test = test_case_a, k = 5)
```

```
## [1] setosa
## Levels: setosa versicolor virginica
```

```
knn(train = training_x2, cl = training_y2, test = test_case_b, k = 5)
```

```
## [1] versicolor
## Levels: setosa versicolor virginica
```

```
knn(train = training_x2, cl = training_y2, test = test_case_c, k = 5)
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```