# Spline regression

Mattias Villani, Department of Statistics, Stockholm University

This notebook introduces the concept of **spline regression** - a very useful non-linear regression model that can be fitted with least squares. Let's get started by loading some libraries and setting colors.

```r
suppressMessages(library(dplyr)) # Data transformations and tables
suppressMessages(library(tidyverse))
library("RColorBrewer") # for pretty colors
options(repr.plot.width = 12, repr.plot.height = 12, repr.plot.res = 100) # plot size
colors = brewer.pal(12, "Paired")[c(1,2,7,8,3,4,5,6,9,10)];
set.seed(12332)         # set the seed for reproducability
```

**Polynomial regression** Consider the basic polynomial regression model of order $K$:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_k x^k + \varepsilon, \qquad \varepsilon \sim \mathrm{N}(0, \sigma^2).$$

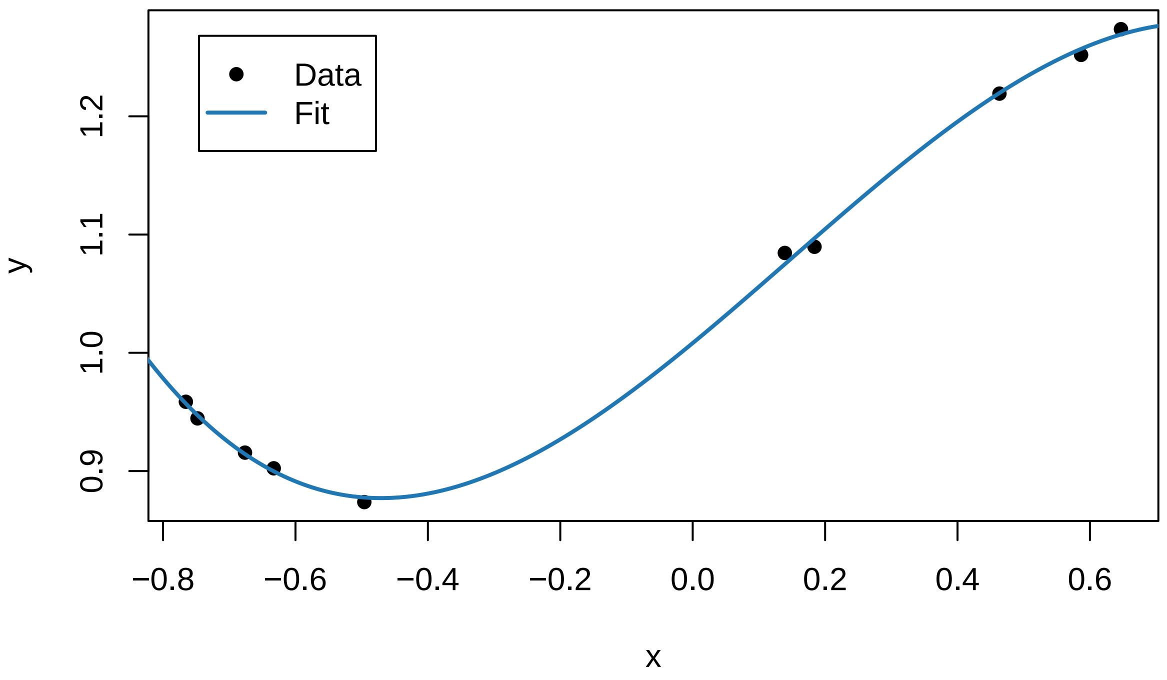
A problem with polynomials is that they are **global basis functions**, meaning that changes in the data in one part of the space (for example, changing a $y$ data point for one of the large $x$ values) can affect the fit in a completely different part of the space (for example the fit at the smallest values of $x$). Let's us see this is action by simulating some data, fitting the polynomial regression by least squares and then plotting the predictions over a fine grid of $x$-values:

```r
# Simulate data and plot
n = 10
sigma = 0.01
x = runif(n, min = -1, max = 1)
y = 1 + 0.5*x + 0.2*x^2 - 0.5*x^3 + rnorm(n, sd = sigma)
X = cbind(1,x,x^2,x^3) # lm adds intercept automatically
plot(x,y, pch = 16)

# Least squares fit
betaHat = solve(crossprod(X),crossprod(X,y)) # Least squares (X'X)^{-1}(X'y)

# Prediction
xt = seq(-1, 1, length = 1000)
Xtest = cbind(1, xt, xt^2, xt^3)

yPred = Xtest%*%betaHat
lines(xt, yPred, col = colors[2], lwd = 2)
legend(x = "topleft", inset=.05, legend = c("Data", "Fit"),
       lty = c(NA, 1), lwd = c(2, 2), pch = c(16,NA),
       col = c("black", colors[2]))
```

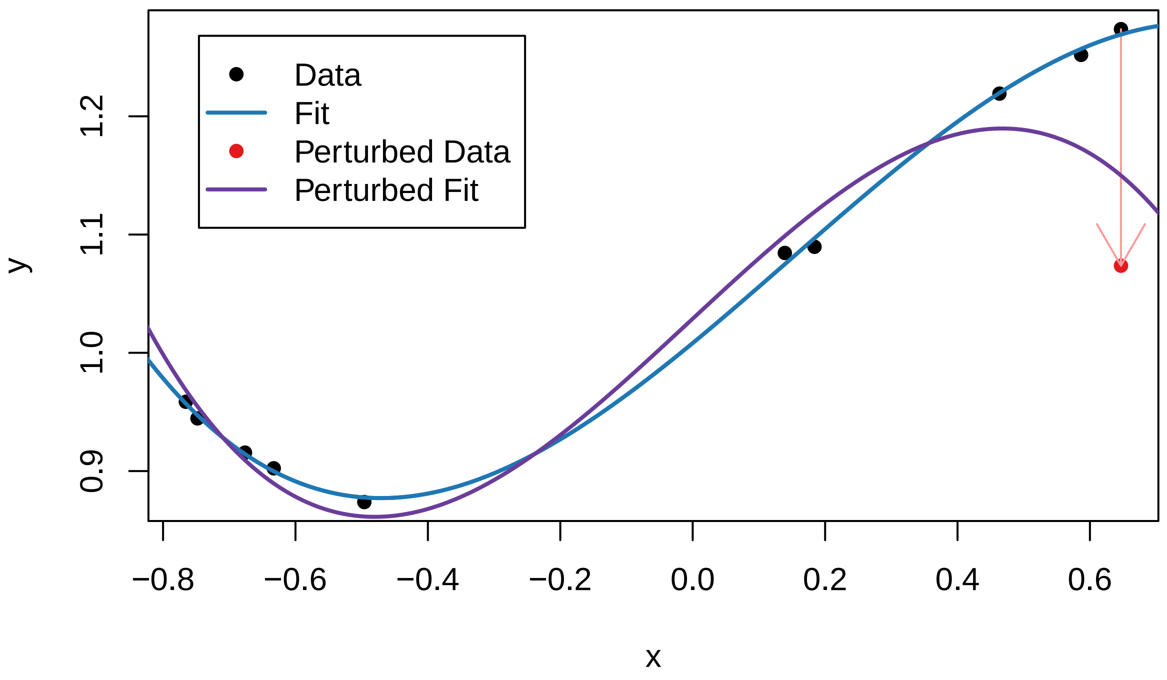


Now, let's change the $y$-value for the largest $x$ in the data, re-fit the model and plot the predictions again.

```r
ind = which.max(x)
yPerturb = y
yPerturb[ind] = y[ind] - 0.2
plot(x,y, pch = 16)
points(x[ind], yPerturb[ind], col = colors[8], pch = 16)
arrows(x[ind], y[ind], x[ind], yPerturb[ind], col = colors[7])
lines(xt, yPred, col = colors[2], lwd = 2)

# refit
betaHatPerturb = solve(crossprod(X),crossprod(X,yPerturb))

# Prediction
yPredPerturb = Xtest%*%betaHatPerturb
lines(xt, yPred, col = colors[2], lwd = 2)
lines(xt, yPredPerturb, col = colors[10], lwd = 2)
legend(x = "topleft", inset=.05,
       legend = c("Data", "Fit", "Perturbed Data", "Perturbed Fit"),
       lty = c(NA, 1, NA, 1), lwd = c(2, 2, 2, 2), pch = c(16,NA,16, NA),
       col = c("black", colors[2], colors[8], colors[10]))
```



Note how the fit at lower $x$ is also affected even though we only changed a $y$ for the largest $x$. Polynomials are global basis functions. It is usually better to use more **local basis functions** than powers of $x$. One such local basis function model is **spline regression**.

**Spline regression** An spline regression defines a set of basis functions locally around certain pre-determined positions in $x$-space called **knots**. Here is a simple example of spline regression with **truncated quadratic** basis functions defined at the $M$ knots $\kappa_1, \ldots, \kappa_M$:

$$y = \beta_0 + \beta_1 x + \beta_2 (x - \kappa_1)_+^2 + \ldots + \beta_{M+1}(x - \kappa_M)_+^2 + \varepsilon, \qquad \varepsilon \sim \mathrm{N}(0, \sigma^2).$$

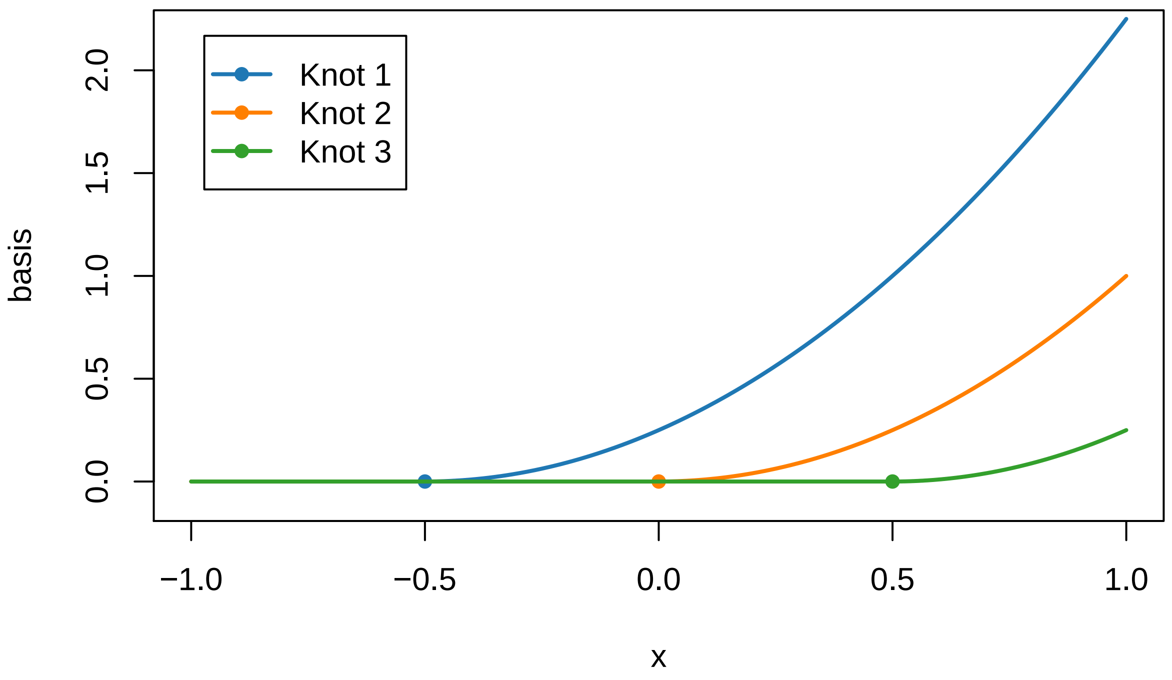
where

$$(x - \kappa_m)_+^2 = \begin{cases} 0, & \text{if } x \leq \kappa \\ (x - \kappa)^2, & \text{if } x > \kappa \ . \end{cases}$$

These new basis functions are perhaps easiest to understand visually. Let us plot the basis functions for three different knots: $\kappa_1 = -0.5, \kappa_2 = 0.0$ and $\kappa_3 = 0.5$:

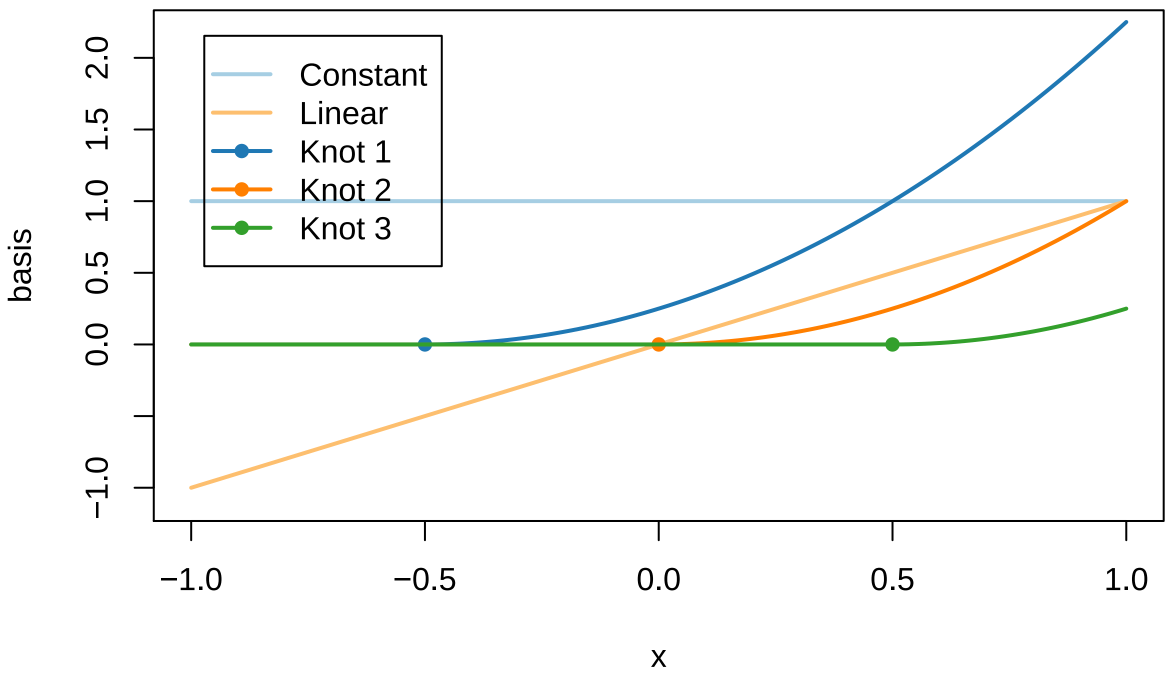
```r
# Function that computes the basis function for a vector of x-values.
# The order=2 (quadratic) was used in the example above.
TruncPoly <- function(x, knot, order){
    basis = (x-knot)^order
    basis[(x-knot)<=0] = 0
    return(basis)
}

plot(1, type="n", xlab="x", ylab="basis", xlim=c(-1, 1), ylim=c(-0.1, 2.2))
knots = c(-0.5,0,0.5)
for (i in 1:length(knots)){
    lines(xt, TruncPoly(xt, knots[i], 2), type = "l", lwd = 2, col = colors[2*i])
    points(knots[i], 0, col = colors[2*i], pch = 16)
}
legend(x = "topleft", inset=.05, legend = c("Knot 1", "Knot 2", "Knot 3"),
       lty = c(1, 1, 1), lwd = c(2, 2, 2), pch = c(16, 16, 16),
       col = c(colors[2], colors[4], colors[6]))
```



We can also think about linear regression as being regression on the two basis function 1 and $x$, i.e. the constant and linear basis functions. Let's add also those to the plot:

```r
plot(1, type="n", xlab="x", ylab="basis", xlim=c(-1, 1), ylim=c(-1.1, 2.2))
knots = c(-0.5,0,0.5)
lines(xt, rep(1,length(xt)), lwd = 2, col = colors[1])
lines(xt, xt, col = colors[3], lwd = 2)
for (i in 1:length(knots)){
    lines(xt, TruncPoly(xt, knots[i], 2), type = "l", lwd = 2, col = colors[2*i])
    points(knots[i], 0, col = colors[2*i], pch = 16)
}
legend(x = "topleft", inset=.05, legend = c("Constant","Linear","Knot 1", "Knot 2", "Knot 3"),
       lty = c(1, 1, 1, 1, 1), lwd = c(2, 2, 2, 2, 2), pch = c(NA, NA, 16, 16, 16),
       col = c(colors[1], colors[3], colors[2], colors[4], colors[6]))
```



We can estimate the spline regression by least squares, just like we did for the polynomial regression. We let each basis function be a new covariate by adding it as a new column in the covariate matrix $X$. Let's make a function `TruncPolyMatrix`that computes all covariates for the knots, including the constant (intercept) and linear covariate:

```
TruncPolyMatrix <- function(x, knots, order){
    X = cbind(1,x)
    for (knot in knots){
        X = cbind(X, TruncPoly(x, knot, order))
    }
    return(X)
}
X = TruncPolyMatrix(x, knots = c(-0.5,0,0.5), order = 2)
```

To clearly see how the local basis functions are constructed, we will sort the observations with respect to their value on the linear covariate. In order to do that, we set up the data as a dataframe and sort by the linear column:

```
data = data.frame(cbind(X,y))
names(data) <- c("const", "linear", "kappa=-0.5", "kappa=0.0", "kappa=0.5", "y")
arrange(data, linear) # sorting the rows with respect to the column 'linear'
```

```
##    const     linear   kappa=-0.5  kappa=0.0   kappa=0.5         y
## 1      1 -0.7655907 0.0000000000 0.00000000 0.000000000 0.9586053
## 2      1 -0.7479613 0.0000000000 0.00000000 0.000000000 0.9445658
## 3      1 -0.6762505 0.0000000000 0.00000000 0.000000000 0.9156011
## 4      1 -0.6328205 0.0000000000 0.00000000 0.000000000 0.9023558
## 5      1 -0.4960179 0.0000158574 0.00000000 0.000000000 0.8737876
## 6      1  0.1391588 0.4085239951 0.01936518 0.000000000 1.0844386
## 7      1  0.1839904 0.4678429269 0.03385248 0.000000000 1.0896732
## 8      1  0.4633416 0.9280269670 0.21468540 0.000000000 1.2191652
## 9      1  0.5866791 1.1808715212 0.34419240 0.007513271 1.2519275
## 10     1  0.6469014 1.3153828038 0.41848141 0.021580019 1.2736584
```

Note how: - each knot corresponds to a new covariate and - how a knot covariate is zero whenever $x \leq \kappa$, i.e. when $x$ (the linear basis function) is below the value of the knot.

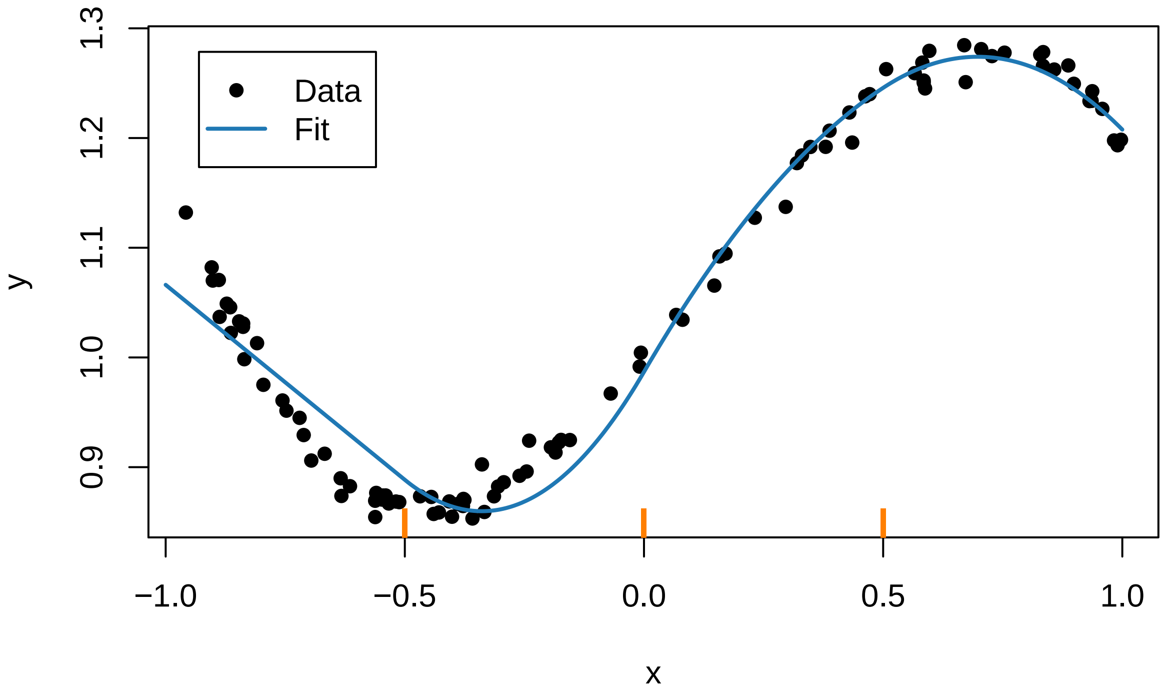
The function `SplineTrainTestPlot` below does: - Constructs the matrix of basis covariates for the training data using our `TruncPolyMatrix` function - Estimates the $\beta$-coefficients in the spline regression using training data - Constructs the matrix of basis covariates for the test data (xt is typically a fine grid of $x$-values) using again our `TruncPolyMatrix` function - Computes mean predictions for all test data points - Plots the training dat, the test predictions as a curve and the knots as orange bars above the $x$-axis.

```r
SplineTrainTestPlot <- function(x, y, knots, order, xt){
    X = TruncPolyMatrix(x, knots, order)
    betaHat = solve(crossprod(X),crossprod(X,y))
    Xtest = TruncPolyMatrix(xt, knots, order)
    yPred = Xtest%*%betaHat
    plot(x,y, pch = 16)
    lines(xt, yPred, col = colors[2], lwd = 2)
    points(knots,rep(par("usr")[3],length(knots)), pch = "|", cex = 2, col = colors[4])
    legend(x = "topleft", inset=.05, legend = c("Data", "Fit"),
       lty = c(NA, 1), lwd = c(2, 2), pch = c(16, NA),
       col = c("black", colors[2]))
    return(list(betaHat = betaHat, yPred = yPred))
}
```

Let us now simulate a little larger dataset and fit the spline regression model. Note that the knots are plotted by orange bars above the $x$-axis.

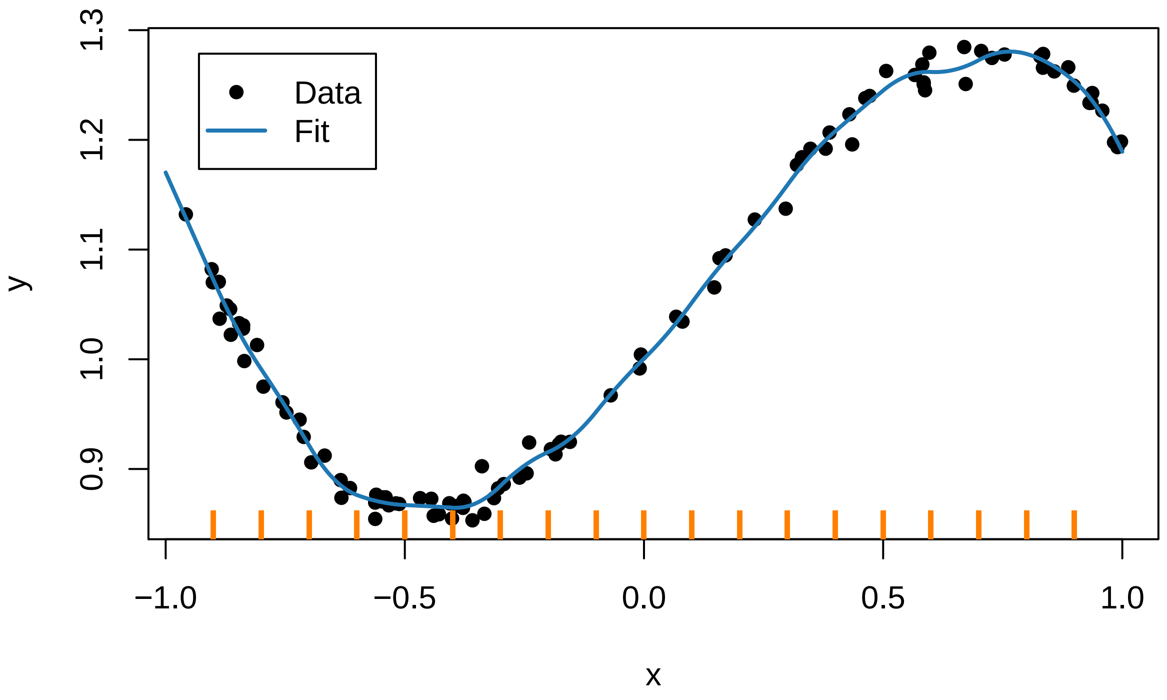
```r
# Simulate data and plot
n = 100
sigma = 0.01
x = runif(n, min = -1, max = 1)
y = 1 + 0.5*x + 0.2*x^2 - 0.5*x^3 + rnorm(n, sd = sigma)
plot(x,y, pch = 16)

# Set up spline, fit, predict and plot
knots = c(-0.5, 0.0, 0.5)
res = SplineTrainTestPlot(x, y, knots, order = 2, xt)
```



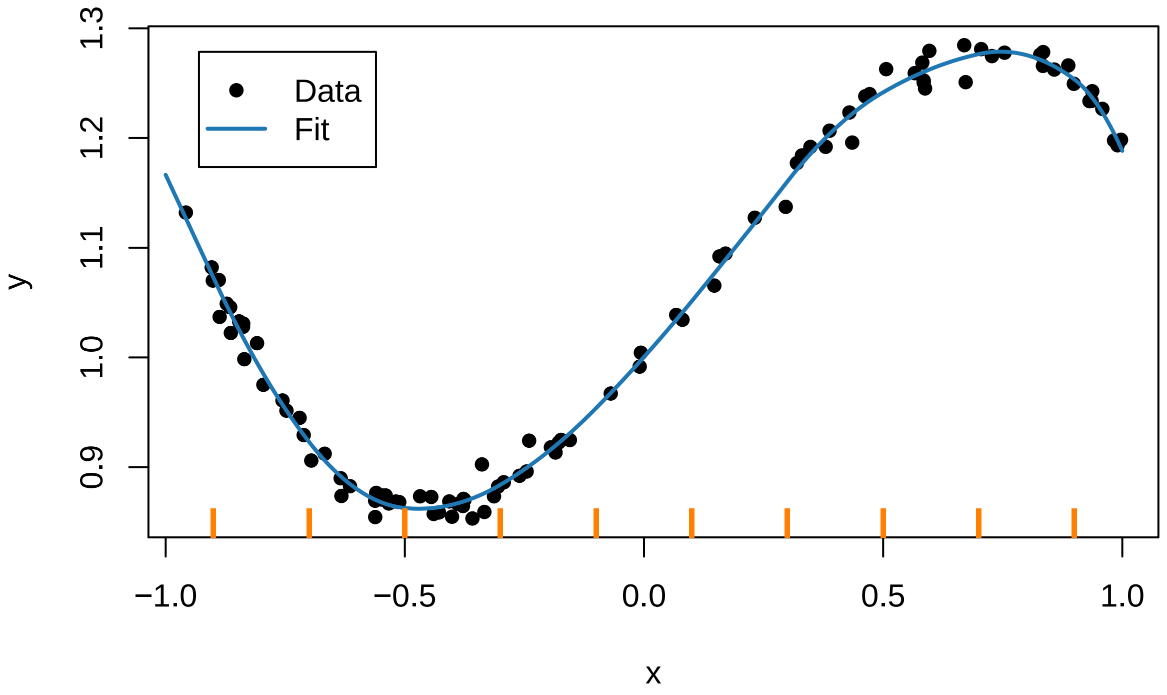
We set that we do not fit the data well for smaller x values. In particular since we do not have a knot below x=-0.5 we cannot fit the data for such x-values. Let's add more knots!

```
knots = seq(-0.9,0.9, by = 0.1)
res = SplineTrainTestPlot(x, y, knots, order = 2, xt)
```



Oops, clearly too many knots now! We are overfitting the data. How about just 10 of them?

```
knots = seq(-0.9,0.9, by = 0.2)
res = SplineTrainTestPlot(x, y, knots, order = 2, xt)
```



We seem to have hit the sweet spot! But is there a better, more automatic, way to achieve this? Yes! There are two common methods for making sure that spline regression don't overfit when (too) many knots are used:

- **Variable selection**. Since each knot corresponds to a covariate, we can do any form of variable selection to remove knot covariates that are not needed, i.e. that insignificant. This could be the usual forward or backward selection methods from basic regression courses, or more sophisticated methods like Bayesian variable selection from Markov Chain Monte Carlo (MCMC) simulation methods (see Lecture 12 in my course Bayesian learning).
- **Regularization** using L1 or L2 shrinkage. More about this later.

---

Prepared for the course Machine Learning, 7.5 credits at Stockholm University by Mattias Villani