# Nonlinear Regression

Mark Andrews

## Contents

## Introduction

In Chapter 11 (Generalized Linear Models) and Chapter 12 (Multilevel Models), we saw how the basic or default regression model, which we referred to as the normal linear model, can be generalized and extended so that it can apply to a much wider range of problems than would otherwise be the case. In this chapter, we will consider another important generalization of the basic regression model. In order to introduce this generalization, let us again return to the normal linear model. In the normal linear model, we assume we have a set of $n$ univariate observations:

$$y_1, y_2 \ldots y_i \ldots y_n.$$

Corresponding to each $y_i$, we have a set of $K$ predictor variables $x_{i1}, x_{i2} \ldots x_{ik} \ldots x_{iK}$. For simplicity, we usually refer to these $K$ variables that correspond to $y_i$ as either $\vec{x}_i$, $\mathbf{x}_i$, or even simply $x_i$. The normal linear model assumes that each $y_i$ is a sample from a normal distribution with a mean $\mu_i$ and a variance $\sigma^2$, and the mean $\mu_i$ is a linear function of the predictors $x_i$. We can write that statement more formally as follows:

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = \beta_0 + \sum_{k=1}^{K} \beta_k x_{ik}, \quad \text{for } i \in 1 \ldots n.$$

In other words, this model assumes that each $y_i$ is a sample from a normal distribution whose mean $\mu_i$ is a (deterministic) linear function of the predictors.

The assumption of a linear relationship between the mean of the outcome variable and the predictor variables is a strong one. It means that we are assuming that the average value of the outcome variable will change as a constant proportion of a change in any predictor variable. For example, if were to change $x_{ik}$ by $\Delta_{x_{ik}}$, we assume that $\mu_i$ will change by exactly $\beta_k \times \Delta_{x_{ik}}$. We assume that this holds for all predictor variables and for any value of the change $\Delta_{x_{ik}}$. Clearly, this is a restrictive assumption that will not generally hold. In Figure 1, we show some of the many ways in which this assumption can be violated. In each of these examples, the average value of the outcome variable does not change by a constant amount with any constant change in the predictor variable. Clearly, the average trends in each of these scatterplots are adequately described by straight lines.

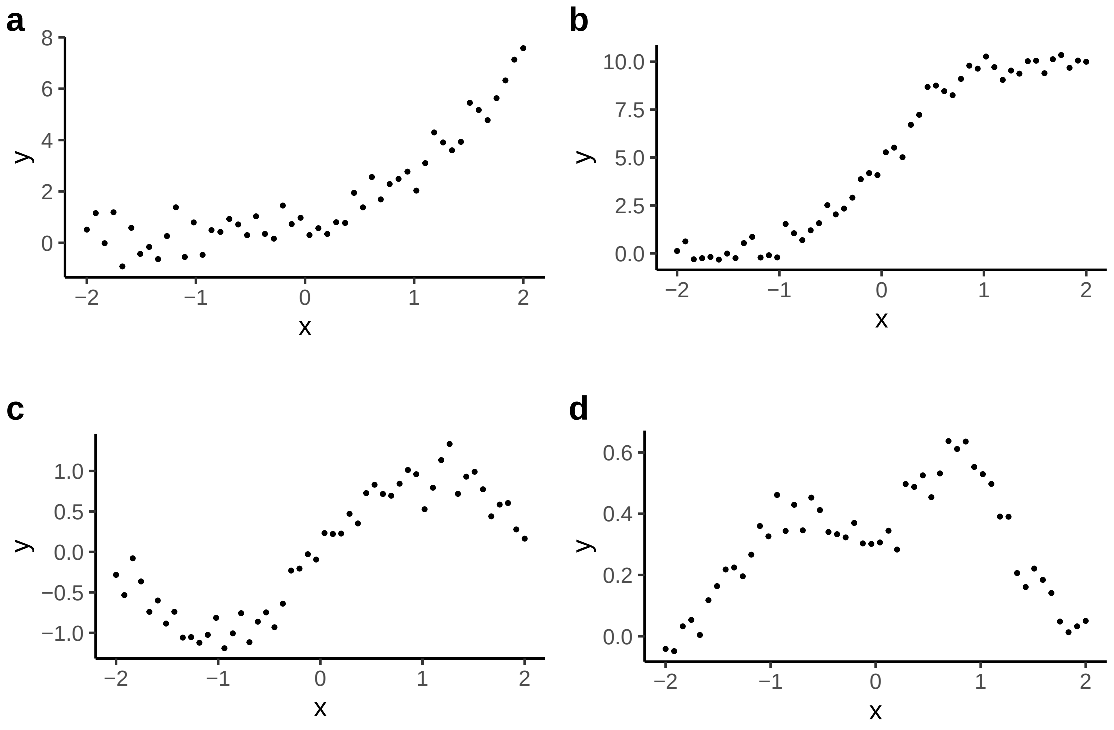

Figure 1: Examples of nonlinear regression models. In each case, the values of the outcome variables are sampled from normal distributions whose means are nonlinear functions of the the predictor variable.

We can deal with these situations by no longer modelling the average value of the outcome variable as linear function of the predictors. For example, we can define a normal *nonlinear* regression model as follows:

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = f(x_i, \theta), \quad \text{for } i \in 1 \ldots n,$$

where $x_i$ is the vector of $K$ predictors as before, $f$ is some (deterministic) nonlinear function of $x_i$, and $\theta$ is a set of parameters of $f$. More generally, of course, we do not have to assume our outcome variable is normally distributed. In fact it could be any parameterized probability distribution. Moreover, we do not have to assume that it is the mean of the outcome distribution that varies as the deterministic function of $x_i$. This leads to the following more general nonlinear regression model:

$$y_i \sim D(\mu_i, \psi), \quad \mu_i = f(x_i, \theta), \quad \text{for } i \in 1 \ldots n,$$

where $D$ is some probability distribution with parameters $\mu_i$ and $\psi$, and $\mu_i$ is the nonlinear function of $x_i$, which is paramterized by $\theta$.

Using a nonlinear regression model, whether the normal or the more general one, introduces considerable conceptual and practical difficulties. First, we must choose the nonlinear function, or functional form, $f$.

There are indescribably many possibilities for $f$ and choosing between them or even choosing a plausible set of candidate functions can be very challenging, as we will see. Second, inference of the values of the unknown parameters of the model can also be very challenging. In the normal linear regression model, inference of, for example, the maximum likelihood estimator, can be accomplished using algebra. Likewise, assuming appropriate choices of prior distributions for the parameters, the Bayesian posterior distribution over these parameters can be obtained through algebra. This is generally not the case when we use nonlinear functions, even if our outcome variable is normally distributed, and for arbitrary nonlinear functions $f$ and arbitrary probability distributions for the outcome variable $D$, inference of the values of the unknown parameters in both $f$ and $D$ may lead to insurmountable computational challenges. Typically, different numerical algorithms are employed in different situations. Often these work well, but it is not at all uncommon to encounter computational problems or failures with these algorithms too. Finally, even assuming that inference was succesful, the interpretion of parameters in nonlinear regression models may also be challenging or even impossible. In a linear model, each regression coefficient has a simple interpretation: it gives the rate of change of the outcome variable for any unit change of the predictor. In nonlinear models, it is not so simple. Certainly, in some cases of nonlinear regression, the parameters can have a clear and meaningful interpretations. But in other cases, individual parameters can not be understood independently of other parameters. This sometimes entails that it can be difficult to assess whether one predictor variable has any statistically meaningful relationship (e.g., a statistically significant relationship) with the outcome variable. More generally, it can be challenging to summarize and explain the relationship, if any, between the predictors and the outcome variable. Simply put, nonlinear regression models can be opaque, and as such, are sometimes preferred to be treated just as *black box* models, used for the accuracy of their predictions rather than for any conceptual or explanatory insights.

For all of these reasons, nonlinear regression should be treated with some caution and as a relatively advanced statistical topic. That is emphatically not to say that it should be avoided. It is in general a very powerful tool that can allow us to do either more advanced analyses than were otherwise possible, or even to be able to deal with problems that were otherwise impossible. It is just that we should also be clear that nonlinear regression raises additional conceptual and practical challenges compared to those we have encountered when using general or generalized linear models.

## Parametric nonlinear regression

In *parametric nonlinear regression*, we choose a particular parametric functional form for the nonlinear function $f$ and then infer the values of its unknown parameters from data. The other types of nonlinear regression that we will cover in this chapter also have parameters, which are inferred from data, and so can also be termed parametric models. However, in these cases, the functional form of the nonlinear function in the regression model is essentially unknown and is being approximated by a more flexible nonlinear model.

### Example 1: A sigmoidal regression

To begin with this topic, let us start with an esssentially arbitrarily chosen nonlinear function, such as the following nonlinear function of a single variable $x$, as an example:

$$f(x, b, \alpha, \beta) \triangleq b \tanh(\alpha + \beta x).$$

The tanh (hyperbolic tangent) function is defined as follows:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

It is a *sigmoidal* or S-shaped function that maps the real line to the interval $[-1, 1]$, see Figure 2a, and so $f(x, b, \alpha, \beta)$ is a linear function of a nonlinear function of another linear function of $x$. Functions of this type are widely used in artificial neural network models (see Bishop 1995). With this nonlinear function, our nonlinear regression model would then be

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = b \tanh(\alpha + \beta x_i), \quad \text{for } i \in 1 \ldots n.$$

In order to explore this regression model in practice, we will begin by simply generating data according to the model.

```
Df <- local({
  b <- 3.0
  alpha <- 0.75
  beta <- 0.25
  tibble(x = seq(-10, 10, length.out = 50),
         y = b * tanh(alpha + beta*x) + rnorm(length(x), sd = 0.5))
})
```

This data is shown in Figure 2b.

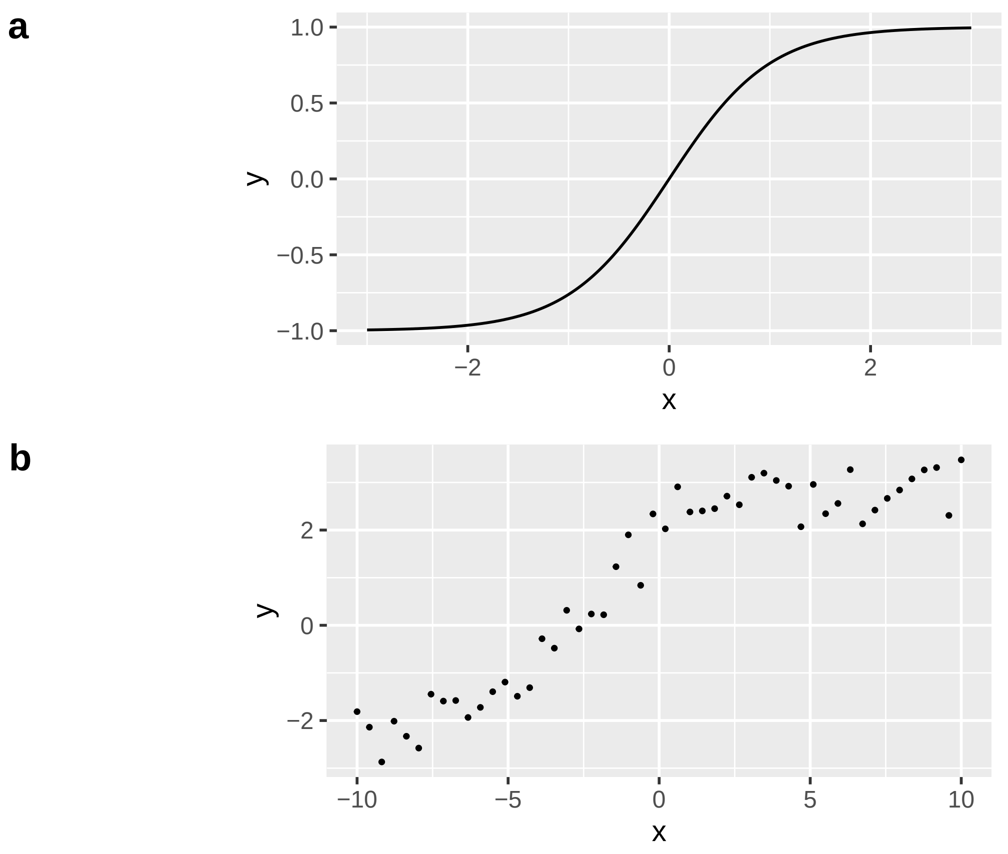

Figure 2: a) The tanh, or hyperbolic tangent, function. b) Data normally distributed around a linear function of a tanh function.

To perform this regression analysis, we can use the `nls` function that is part of the base R `stats` package, which is always preloaded. With `nls`, in its R formula, we state the parametric form of the function we are assuming. In our case, this will be `y ~ b * tanh(alpha + beta * x)`. The `nls` function then attempts to find the *least squares* solution for `alpha` and `beta`. In other words, and more formally, it attempts to find the values of $b$, $\alpha$ and $\beta$ that minimize the following formula.

$$\sum_{i=1}^{n} (y_i - b \tanh(\alpha + \beta x_i))^2$$

Usually, `nls` requires us to also provide an initial guess of the values of the unknown parameters. In the following example, we'll set `alpha` to 0 and `b` and `beta` to 1.

```
M <- nls(y ~ b * tanh(alpha + beta * x),
         start = list(b = 1, alpha = 0, beta = 1),
         data = Df)
```

We can view the results of this analysis using `summary` just as we did with `lm`, `glm`, `lmer` etc models.

```
summary(M)
#>
#> Formula: y ~ b * tanh(alpha + beta * x)
#>
#> Parameters:
#>       Estimate Std. Error t value Pr(>|t|)
#> b      2.84163    0.12458  22.809  < 2e-16 ***
#> alpha  0.77170    0.10205   7.562 1.15e-09 ***
#> beta   0.24426    0.02902   8.416 6.14e-11 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.4478 on 47 degrees of freedom
#>
#> Number of iterations to convergence: 9
#> Achieved convergence tolerance: 4.778e-06
```

As can we see, the least squares estimates of $b$, $\alpha$ and $\beta$, which we'll label as $\hat{b}$, $\hat{\alpha}$ and $\hat{\beta}$, are 2.842, 0.772 and 0.244. The estimate for $\sigma$, the standard deviation of the normal distribution around each $\mu_i$, also known as the *residual standard error*, has a value of 0.448. All of these values match well the parameters that we used to generate the data.

We may now view the predictions of the model using the `predict` function. Recall that by default, `predict` will return the predicted values of the outcome variable for each value of the predictor variables using the estimated values of the parameters. In other words, it will calculate $\hat{y}_i = \hat{b}\tanh(\hat{\alpha} + \hat{\beta}x_i)$ for each $x_i$. We'll use the following code to perform this prediction and plot the results, which is shown in Figure 3

```
Df %>%
  mutate(y_pred = predict(M)) %>%
  ggplot(aes(x = x)) +
  geom_point(aes(y = y), size = 0.5) +
  geom_line(aes(y = y_pred), colour = 'red')
```

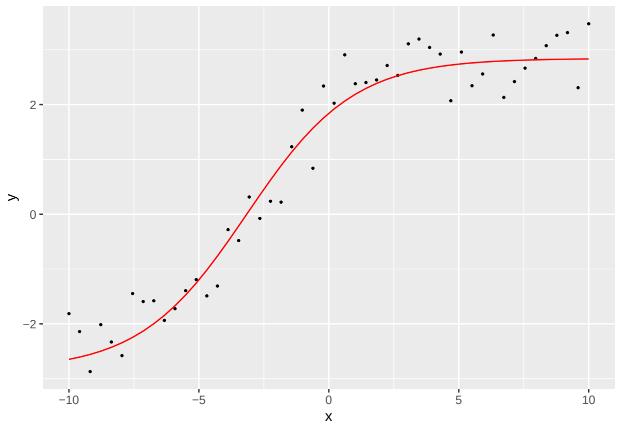

Figure 3: The fitted tanh regression model.

One practical issue with `nls` is that choosing starting values may not always be a simple matter. Even in the above example, starting values relatively close to the true values, say 0 for all parameters, will lead to the algorithm failing.

```
M_tmp <- nls(y ~ b * tanh(alpha + beta * x),
             start = list(b = 0, alpha = 0, beta = 0),
             data = Df)
#> Error in nlsModel(formula, mf, start, wts): singular gradient matrix at initial parameter estimates
```

In the case of some functions, however, the `stats` package provides functions that use heuristics to roughly estimate some reasonable starting values of parameters. These so-called `selfStart` functions only exist for a restricted set of functions. While it is possible to create new `selfStart` functions for new functions, creating the heuristics is itself not simple. For our $f(x, b, \alpha, \beta)$ function, no `selfStart` function exists. However, a `selfStart` function, `SSlogis`, does exist for a related function:

$$\texttt{SSlogis}(x, \text{Asym}, x_{\text{mid}}, \text{scal}) \triangleq \frac{\text{Asym}}{1 + e^{\frac{x_{\text{mid}} - x}{\text{scal}}}}$$

This is also a sigmoidal function, but it is bounded between 0 and $a$. Because it can not take on negative values, to use it with our data, we must first substract the minimum value of all $y_i$ values from each $y_i$. Having done so, we can use the `getInitial` function to return some plausible starting values for Asym, $x_min$, and scal.

```
inits <- getInitial(y ~ SSlogis(x, Asym, xmid, scal),
                    data = mutate(Df, y = y - min(y))
)
inits
#>      Asym      xmid      scal
#>  5.850634 -3.055370  2.180384
```

With some algebra, we can then map the parameters of `SSlogis` to those of our tanh based function as follows:

$$\alpha = \frac{\text{Asym}}{2\text{scal}}, \quad \beta = \frac{1}{2\text{scal}}, \quad b = \frac{\text{Asym}}{2}$$

We may now use these starting values in our `nls` model.

```
#>
#> Formula: y ~ b * tanh(alpha + beta * x)
#>
#> Parameters:
#>       Estimate Std. Error t value Pr(>|t|)
#> b      2.84163    0.12458  22.809  < 2e-16 ***
#> beta   0.24426    0.02902   8.416 6.14e-11 ***
#> alpha  0.77170    0.10205   7.562 1.15e-09 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.4478 on 47 degrees of freedom
#>
#> Number of iterations to convergence: 10
#> Achieved convergence tolerance: 3.725e-06
```

## Example 2: Modelling golf putting successes

Thus far, our aim was simply to introduce the basic principles of how to do nonlinear regression using `nls`, and for that we used a seemingly arbitrary nonlinear function and some data generated using this model. Let us now consider some real world data, but also consider how the choice of the nonlinear function can be, and ideally ought to be, theoretically motivated. The data we will use concerns the relative frequencies with which professional golfers successfully putt the golf ball into the hole as a function of their distance from the hole. This data, which is available in the `golf_putts.csv` file, is taken from Berry (1995), and was discussed further in Gelman and Nolan (2002).

```
golf_df <- read_csv('golf_putts.csv')
golf_df
#> # A tibble: 19 x 3
#>    distance attempts success
#>       <dbl>    <dbl>   <dbl>
#>  1        2     1443    1346
#>  2        3      694     577
#>  3        4      455     337
#>  4        5      353     208
#>  5        6      272     149
#>  6        7      256     136
#>  7        8      240     111
#>  8        9      217      69
#>  9       10      200      67
#> 10       11      237      75
#> 11       12      202      52
#> 12       13      192      46
#> 13       14      174      54
#> 14       15      167      28
#> 15       16      201      27
#> 16       17      195      31
#> 17       18      191      33
#> 18       19      147      20
#> 19       20      152      24
```

As we can see, this data provides the number of putting attempts and the number of successful putts at various distances (in feet) from the hole. For example, there were 1443 recorded putting attempts at a distance of 2 feet from the hole. Of these 1443 attempts, 1346, or around 93%, were successful. On the other hand, there were 200 recorded attempts at a distance of 10 feet, or which 67, or around 34%, were successful. The absolute number of attempts and successes at each distance is vital information and so ideally we should base our analysis on this data, using a binomial logistic regression or a related model. However, for simplicity here, we will just use the relative frequencies of successes at each distance. To do so, we will first create a new variable, `prob`, that is the ratio of successes to attempts at each distance. This is done in the following code, and the plot of these probabilities as a function of distance is show in Figure 4.

```
golf_df %<>% mutate(prob = success/attempts)
```

In order to fit a parametric nonlinear regression model to this data, we must begin by choosing the nonlinear function. We could choose functions whose shape appears to roughly match the decaying pattern of the points. However, when it is possible, it is preferrable to choose a function based on a principled model of the phenomenon. As an example, let us use the simple model described in Gelman and Nolan (2002). We will treat the golf ball as a circle of radius $r$ and the hole as the circle of radius $R$. In reality, the values of $r$ and $R$ are 21.33mm and 53.975mm, respectively. In Figure 5, we draw these two circles to scale, positioned a distance of $d$ apart. If the golf ball travels in a straight line along the line from its centre to the centre of the hole, it will fall into the hole. If it deviates slightly from this straight line, it will still fall in if the angle of the trajectory to the left or the right is no greater than $\theta_{\text{crit}} = \sin^{-1}\left(\frac{R}{d}\right)$, which is the angle between the vertical line of length $d$ and the tangent line from the centre of the circle representing the ball to the circle representing the hole[1]. We can assume that deviation from a perfect straight line of a professional golfer's

[1]The tangent line from the centre of the small circle intersects the line from the centre of the larger circle at a right angle. In the right angled triangle that is formed, see Figure 5, the side opposite $\theta_{\text{crit}}$ is length $R$, and the hypotheneuse is of length $d$.

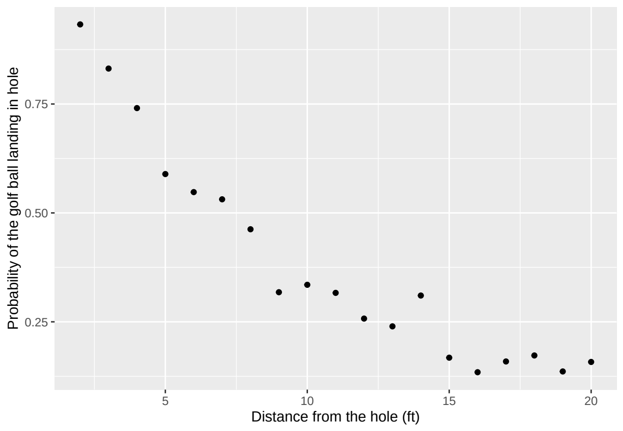

Figure 4: The relative frequencies of successful putts by professional golfers as a function of distance (ft) from the hole.

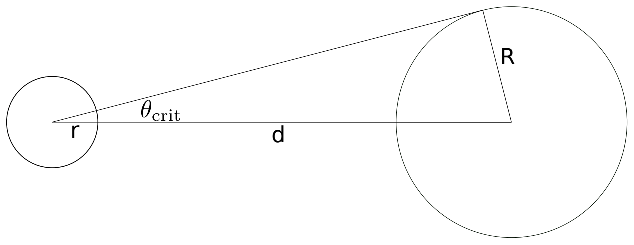

Figure 5: A golf ball of radius $r$ (left) and the golf hole of radius $R$ (right). The centres of these two circles are $d$ apart. If the golf ball travels in a straight vertical line to the hole, it will fall in. If its trajectory deviates, either to the right or to the line, greater than an angle of $\theta_{\text{crit}}$, it will miss. The angle $\theta_{\text{crit}}$ is the angle between the vertical line of length $d$ and the tangent line from the centre of the ball to the hole. The line from the centre of the hole meets the tangent line at a right angle. As such, $\theta_{\text{crit}} = \sin^{-1}\left(\frac{R}{d}\right)$.

putt will be normally distributed with a mean of zero and a standard deviation of $\sigma$, where the value of $\sigma$ is unknown. Given this, the probability that the angle of their putt will be between 0 and $\theta_{\text{crit}}$ is

$$\mathrm{P}(0 < \theta \leq \theta_{\text{crit}}) = \Phi(\theta_{\text{crit}}|0, \sigma^2) - \tfrac{1}{2},$$

where

$$\Phi(\theta_{\text{crit}}|0, \sigma^2) \triangleq \int_{-\infty}^{\theta_{\text{crit}}} N(\theta|0, \sigma^2),$$

which is the value at $\theta_{\text{crit}}$ of the cumulative distribution function of a normal distribution of mean 0 and standard deviation $\sigma$. We simply double the quantity $\Phi(\theta_{\text{crit}}|0, \sigma^2) - \frac{1}{2}$ to get $\mathrm{P}(\theta_{\text{crit}} < \theta \leq \theta_{\text{crit}})$. Therefore, the probability of a sucessful putt is

$$2\Phi\left(\sin^{-1}\left(\tfrac{R}{d}\right)|0, \sigma^2\right) - 1.$$

This is a nonlinear parameteric function of distance $d$, where $R$ is known to have a value of 53.975mm, and $\sigma$ is the single unknown parameter.

This nonlinear function is easily implemented as follows.

```
successful_putt_f <- function(d, sigma){
  R <- 0.17708333 # 53.975mm in feet
  2 * pnorm(asin(R/d), mean=0, sd=abs(sigma)) -1
}
```

The `nls` based model using this `successful_putt_f` function is as follows.

```
M_putt <- nls(prob ~ successful_putt_f(distance, sigma),
              data = golf_df,
              start = list(sigma = 0.1)
)
summary(M_putt)
#>
#> Formula: prob ~ successful_putt_f(distance, sigma)
#>
#> Parameters:
#>       Estimate Std. Error t value Pr(>|t|)
#> sigma 0.041427   0.001289   32.15   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.04233 on 18 degrees of freedom
#>
#> Number of iterations to convergence: 4
#> Achieved convergence tolerance: 9.886e-06
```

As we can see, the estimate for `sigma` is 0.041. This is the estimated standard deviation in the angle of errors the golfer's putts. It is given in radians, and corresponds to 2.374 degrees. The nonlinear function with this estimated value for $\sigma$ is plotted in Figure 6. This appears to be a good fit to the data, which is not necessarily expected given that the physical model that we used was a very simple one.

Before we conclude, however, that our `M_putt` model provides a good explanation of golf putting success rates, it is necessary to peform model evaluation. There are countless techniques for model evaluation, and we have already met some of these in previous chapters. Here, we will look at some widely used methods for comparing the model fit of one model against some alternatives. One alternative model to which we can compare `M_putt` model is a negative exponential model which assumes that the probability of a successful putt decreases exponentially with distance from the hole. In other words, this model is

$$y_i \sim N(\mu, \sigma^2), \quad y_i = a + be^{-\beta x_i}, \quad \text{for } i \in 1 \ldots n$$

The sin of $\theta_{\text{crit}}$ is $\sin(\theta_{\text{crit}}) = \frac{R}{d}$, and so $\theta_{\text{crit}} = \sin^{-1}\left(\frac{R}{d}\right)$

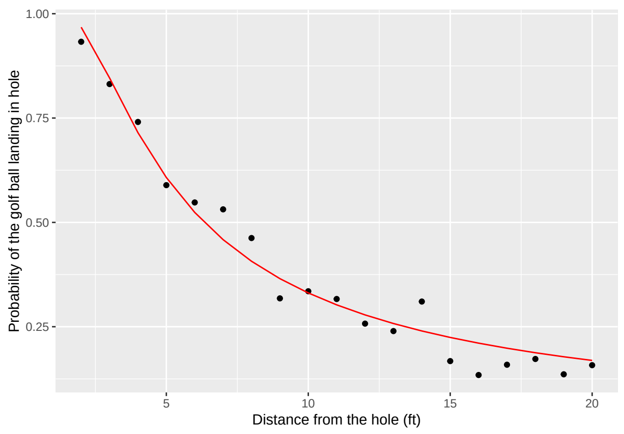

Figure 6: The predictions of the nonlinear regression model based on probabilities of errors in putting angles being normally distributed.

where $y_i$ is the probability of a successful putt for each of the $n$ observed distances. This is simple to set up using `nls`.

```
M_putt_exp <- nls(prob ~ a + b * exp(-beta * distance),
                  data = golf_df,
                  start = list(a = 0, b = 1, beta = 1)
)
summary(M_putt_exp)
#>
#> Formula: prob ~ a + b * exp(-beta * distance)
#>
#> Parameters:
#>      Estimate Std. Error t value Pr(>|t|)
#> a     0.07647    0.03540   2.160   0.0463 *
#> b     1.16510    0.04703  24.774 3.45e-14 ***
#> beta  0.14881    0.01750   8.502 2.50e-07 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.03727 on 16 degrees of freedom
#>
#> Number of iterations to convergence: 8
#> Achieved convergence tolerance: 3.964e-07
```

We plot the predictions of this model in Figure 7. As we can see, it appears to have a fit to the data as good if not better than the original model.

We can begin quantitatively comparing `M_putt` and `M_putt_exp` by looking at the residuals from each model. Recall that the residuals are defined as follows:

$$r_i = y_i - \hat{y}_i, \quad \text{for } i \in 1 \ldots n,$$

where $\hat{y}_i$ is defined, as above, as the predicted value of the outcome variable according to the model. The *residual sum of squares* (RSS) for a given model is the sum of the squares of the residuals:

$$RSS \triangleq \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

There are usually numerous ways in which we can obtain the residuals and the from a model. In the case of `nls` models, we can obtain the residuals as follows using the generic `residuals` function as follows.

```r
residuals(M_putt)
```

However, it is straightforward and instructive to calculate them explicitly, as we do the following code.

```r
get_residuals <- function(model){
  y <- model$m$lhs() # the outcome variable values
  y_hat <- predict(model)
  y - y_hat
}

get_rss <- function(model){
  sum(get_residuals(model)^2)
}
```

The RSS can be used a measures of model fit. The smaller the RSS, the closer the predictions of the model are to the observed data. For `M_putt`, the RSS is 0.032, while for the `M_putt_exp`, the RSS is 0.022. Clearly, `M_putt_exp` has a better fit than `M_putt` with, for example, an RSS being approximately 1.45 times lower.

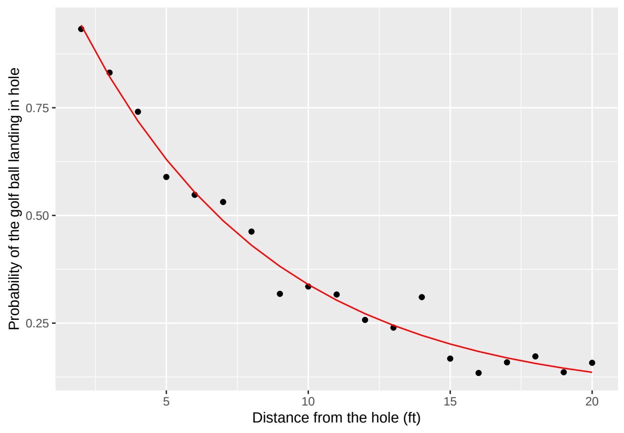

Figure 7: The predictions of the nonlinear regression model based on a negative exponential function.

A related quantity to RSS is the log-likelihood of the model. This is defined as the logarithm to base $e$ of the likelihood function at the maximum likelihood estimates of the unknown parameters, including of $\sigma$, which is the standard deviation of the outcome variables. This is defined as follows:

$$LL \triangleq \log_e\left(\prod_{i=1}^{n} N(y_i|\hat{y}_i, \hat{\sigma})\right),$$

where $y_i$ is as defined above, and $\hat{\sigma}$ is the maximum likelihood estimator of $\sigma$ which is $\hat{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$. The log-likelihood simplifies as follows:

$$LL = -\tfrac{n}{2}\left(1 + \log(2\pi) + \log(RSS) - \log(n)\right).$$

Note that when are comparing two models of the same data, all the terms in this formula except RSS will be the same in the two models. As such, the log-likelihood is just a linear transform (addition of a constant, multiplication by a constant) of the RSS values of the model[^This relationship between RSS and the log-likelihood will not necessarily hold across all models, but does so here because of the normally distributed outcome variables]. We can explicitly calculate the log-likelhood for an `nls` model using the following function.

```
log_likelihood_nls <- function(model){
  n <- length(model$m$lhs())
  -n/2 * (1 + log(2*pi) + log(get_rss(model)) - log(n))
}
```

It is also available using the `logLik` generic function. The log-likelihoods of `M_putt` and `M_putt_exp` are 33.637 and 37.177, respectively. Given that the log-likelihood is essentially the logarithm of the probability density of the observed data given the maximum likelihood estimators of the parameters, and given that the logarithm is monontonic transformation, the *higher* the log-likelihood, the higher the probability of the observed data. Therefore, the data are more likely according to the `M_putt_exp` model than the `M_putt_model`. More precisely, the probability density of data is $e^{LL_{\texttt{M\_putt\_exp}} - LL_{\texttt{M\_putt}}} \approx e^{3.54} \approx 34.47$ higher according to the `M_putt_exp` than the `M_putt` model.

A final goodness of fit measure that we will consider here, which related to $LL$ and so also related to $RSS$, is *Akaike Information Criterion* (AIC). We will cover AIC and related measures in later chapters, and for now we will provide a brief description. For any given model, AIC is defined as follows:

$$AIC = \underbrace{-2LL}_{\text{badness of fit}} + \underbrace{2k}_{\text{penalty}},$$

where $LL$ is the log-likelihood and $k$ is the number of parameters in the model. It is ultimately a approximate measure of out-of-sample predictive performance of the model. In order words, AIC aims to quantify how well a model will generalize to new data that has been generated according to the same data generating process. As we will see, it is related to cross-validation, which is also a method to assess out-of-sample predictive performance. The first term in the AIC formulua, $-2LL$, indicates the lack of fit of the model, simply because it is a constant times the negative of the log-likelihood. The higher the $-2LL$, the lower the log-likelihood. The second term is penalty for model complexity because models with more parameters can overfit data and so generalize poorly. Therefore, AIC balances model fit, or lack thereof, taking in account model complexity. The *lower* the overall value of the AIC, the better the generalization performance of the model. Once, we have the log-likelihood, the AIC is easy to calculate as we see in the following code.

```
aic_nls <- function(model){
  k <- 1 + length(coef(model))
  -2*log_likelihood_nls(model) + 2*k
}
```

It is also available using the `AIC` generic function.

The AIC of `M_putt` and `M_putt_exp` are -63.274 and -66.354, respectively. Clearly, the AIC of `M_putt_exp` is lower `M_putt` by 3.08 and so we expect `M_putt_exp` to generalize to new data better than `M_putt`. Just as there are conventional standards to interpret p-values or Bayes factor, so too are there conventional standards to interpret difference in the AIC values. According to these standards (see, for example, Burnham and Anderson 2003, Chapter 2) AIC difference of greater than between 4 or 7 indicate clear superiority of the predictive power of the model with the lower AIC, while differences of 10 or more indicate that the model with the higher value has essentially no predictive power relative to the model with the lower value. By these standards, the AIC difference that we have observed, i.e. 3.08 indicates that `M_putt_exp` is the model to be preferred, but the case for preferring it is not necessarily clear or overwhelming.

## Polynomial regression

Using `nls` required that we knew or could propose a specific functional form for the nonlinear function in our regression model. We saw the case of the golf putting data where, using some basic principles and knowledge of the domain, we could obtain a specific nonlinear function for our regression model. There are many other examples like this throughout science where theoretical descriptions of the phenomenon of interest, even if sometimes simplified ones, can lead to nonlinear functions that we can use in the regression model. When we are in these situations, `nls` and related tools are very useful. Often, however, we simply do not have theoretically motivated nonlinear functions that describe the phenomenon of interest. Certainly, it may be possible in principle to derive these functions, but this is essentially a type of scientific theory building, and so is not at all a simple matter, especially where the phenomenon and the related data are complex. In these situations then, we often proceed by using a flexible class of nonlinear regression models that essentially attempt to approximate the nonlinear function in regression model. These models are sometimes termed *nonparametric* nonlinear regression models, although the validity of that term is debatable, and include spline and radial basis function regression models, generalied additive models, and Gaussian process regression models. We will cover these models in subsequent sections of this chapter. As a bridge to these topics, we will first cover polynomial regression models.

Polynomial regression models can be seen as just another parametric nonlinear regression model like those we used with `nls` in the previous section because we have a specific nonlinear function, namely a polynomial function of specified degree, that has parameters that we then infer from data. Indeed, it is perfectly possible to use a polynomial function in `nls`. However, polynomial functions are often chosen for their apparent ability to approximate other functions. In other words, in situations where we are modelling data whose trends are nonlinear but where there no known parametric form to this nonlinear, polynomial regression models are often the default choice. The reason why they are chosen is because, as we will see, they are particularly easy to use, being essentially a linear regression on transformed predictor variables. However, we will argue here that polynomial regression should be used with caution, and perhaps should not be a default choice when doing nonlinear regression, because they can often lead to poor fit to the data, either by underfitting but more commonly by *overfitting*, as we will see.

## The nature of polynomial regression

For simplicity, let us begin with a regression model with a single outcome variable and single predictor variable. As we've seen, a normal linear regression model in this case is defined as follows:

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = \alpha + \beta x_i, \quad \text{for } i \in 1 \ldots n.$$

The function $\mu_i = \alpha + \beta x_i$ is a *polynomial of degree one.* In other words, we can write

$$\begin{aligned}\mu_i &= \alpha + \beta x_i,\\ &= \alpha x_i^0 + \beta x_i^1,\end{aligned}$$

where the superscripts in $x_i^0$ and $x_i^1$ are exponents. In other words, $x_i^0$ is $x_i$ raised to the power of 0, which is $x_i^0 = 1$, and $x_i^1$ is $x_i$ raised to the power of 1, which is $x_i^1 = x_i$. The *degree* of the polynomial is the highest power in any of its terms. We can easily extend the linear model to become a polynomial of any degree. For example, a degree 2 polynomial regression version of the above model could be as follows:

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = \alpha + \beta x_i + \gamma x_i^2, \quad \text{for } i \in 1 \ldots n.$$

We could rewrite this as

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = \alpha x_i^0 + \beta x_i^1 + \gamma x_i^2, \quad \text{for } i \in 1 \ldots n,$$

or, to avoid proliferation of Greek letters for the coefficients, by

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = \beta_0 x_i^0 + \beta_1 x_i^1 + \beta_2 x_i^2, \quad \text{for } i \in 1 \ldots n,$$

where the subscripts in $\beta_0$, $\beta_1$, $\beta_2$ are simply indices. Continuing like this to any finite degree is easy. For example, here's a degree $K = 5$ polynomial of each $x_i$.

$$\begin{aligned} y_i \sim N(\mu_i, \sigma^2), \quad \mu_i &= \beta_0 x_i^0 + \beta_1 x_i^1 + \beta_2 x_i^2 + \beta_3 x_i^3 + \beta_4 x_i^4 + \beta_5 x_i^5,\\ \mu_i &= \sum_{k=0}^{K} \beta_k x_i^k, \quad \text{for } i \in 1 \ldots n.\end{aligned}$$

Viewed this way, a polynomial regression model of single predictor is simply a linear regression model with multiple predictors, each of which being the predictor raised to a different power.

Raising a given predictor to successive powers from $0, 1\ldots$ leads to a different function of $x$. Some example as shown in Figure 8. When the power is 0, the function is a flat line at 1 ($y = x^0 = 1$). When the power is 1, the function is a line with slope 1 ($y = x^1 = x$). When the power is 2, the function is a parabolic, or quadratic, function, and so on. The polynomial function $\sum_{k=0}^{K} \beta_k x_i^k$ is a weighted sum of these functions, with the weightings of the sum being the coefficients $\beta_0, \beta_1 \ldots \beta_K$. These weighted sums can approximate different functions. For example, in Figure 9, we provide multiple plots, each with multiple functions of the form $y = \sum_{k=0}^{5} \beta_k x^k$ where $\beta_0, \beta_1 \ldots \beta_5$ are random. In polynomial regression, therefore, we aim to infer a set of coefficients to essentially approximate the function that represents the curve from which our data was generated along with additive normally distributed noise. In other words, we assume that are data are generated as follows:

$$y_i \sim N(f(x_i), \sigma^2), \quad \text{for } i = 1, 2 \ldots n,$$

where $f$ is a nonlinear function that we will approximate with a polynomial of degree $K$, i.e. each $x_i$, we assume that $f(x_i)$ can be approximated by $\sum_{k=0}^{K} \beta_k x_i^k$ for some unknown values of $\beta_0, \beta_1 \ldots \beta_K$.

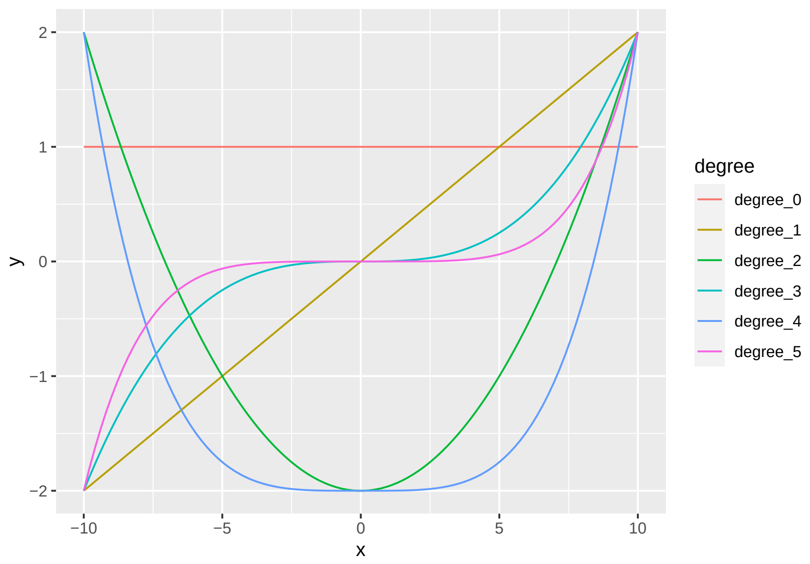

Figure 8: Plots of polynomial functions. For each degree $k \in 0, 1, \ldots 5$, we plot $y = x^k$. We have scaled the value of $y$ in each case so that it occurs within the range $(-2, 2)$, which is done to aid the visualization of each function.

## Polynomial regression in practice

Let use polynomial regression to model some eye-tracking data[^Data from D. Mirman and J. Magnussen.], which is based on averaging data that was obtained from a eye-tracking based cognitive psychology experiment, and is availble in the file `funct_theme_pts.csv`.

```
eyefix_df <- read_csv('funct_theme_pts.csv')
```

The data provides the number of times over the duration of a few seconds that participants look at certain key object in their visual scenes under different experimental conditions. To begin, we will look at the average number of eye fixations at one of the three different types of object in each (50ms) time window, averaging over experimental subjects and experimental conditions. The fixation proportions for all three object is shown in Figure 10. We will begin our analysis with just the data on the *Target* object.

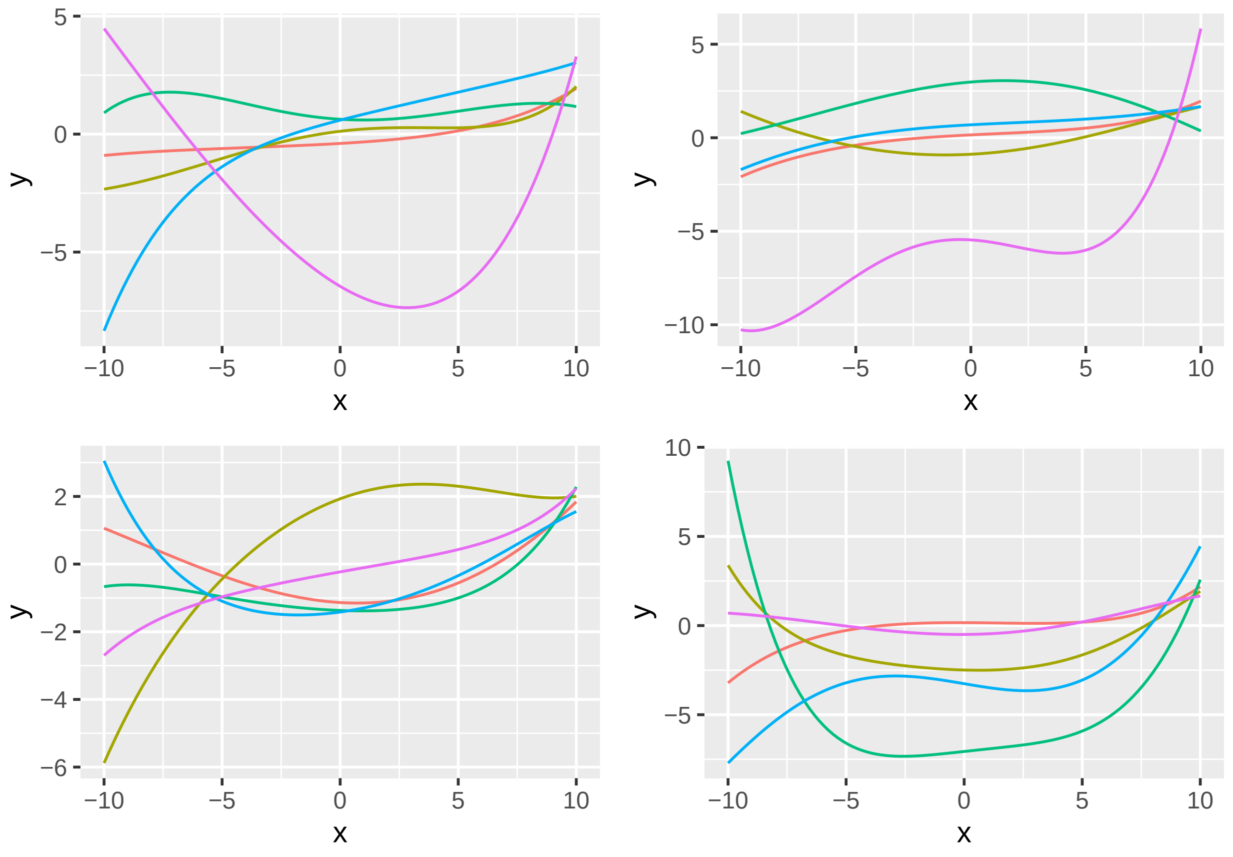

Figure 9: Examples of random polynomial functions. In each subplot, we have five random polynomials of degree $K = 5$. In other words, each function shown in each subplot is defined as $y = \sum_{k=0}^{5} \beta_k x^k$ for some random vector $\beta_0, \beta_1 \dots \beta_5$.

```
eyefix_df_avg <- eyefix_df %>%
  group_by(Time, Object) %>%
  summarize(mean_fix = mean(meanFix)) %>%
  ungroup()

eyefix_df_avg_targ <- filter(eyefix_df_avg, Object == 'Target')
```

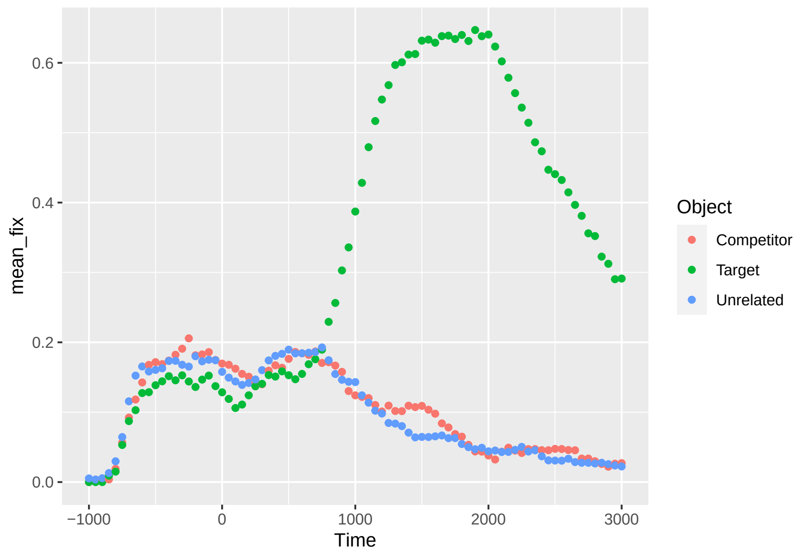

Figure 10: Average proportion of eye fixations at different types of objects (named *Competitor*, *Target*, *Unrelated*) in each time window in multisecond experimental trial.

To perform a polynomial regression in R, we can use `lm`. All we need to do is to create new variables that are our original variable raised to different powers. For example, for a predictor variable `x` and outcome `y`, a degree 3 polynomial regression using `lm` could be written using the following `lm` formula.

```
y ~ x + I(x^2) + I(x^3)
```

Note that we must use `I()`, known as *as is*, here. This essentially transforms e.g. `x^2` to be a new variable in the regression. We can create the same formula more easily using `poly` as follows.

```
y ~ poly(x, degree=3, raw=T)
```

We will explain the meaning of `raw=T` in due course, but for now, we will note that by default `raw=F`.

In the following code, we perform polynomial regression on this data from degree 1 (which is a standard linear model) to degree 9. Note that we are using `purrr::map` here to re-run the same `lm` 9 times. On the first iteration, the `degree` parameter in `poly` takes a value of `1`. On the second iteration, it takes the value of `2`, and so on for each value in the sequence `1` to `9`. The 9 resulting models are stored in the list `M_eyefix_targ`.

```
M_eyefix_targ <- map(seq(9),
                     ~lm(mean_fix ~ poly(Time, degree = ., raw = T),
                         data = eyefix_df_avg_targ)
)
```

The fitted models can be seen in Figure 11 and the model fit statistics are show in Table **??**. For the model fit statistics, we provide $R^2$, adjusted $R^2$, the log of the likelihood, and the AIC. We have covered all four of these statistics in previous chapters, or in the previous section of this chapter. As we can see both in the figures and from the model summaries, we obtain better fits to the data as we increase the degree of the

Table 1: Model fit statistics for polynomial regression models.

| degree | Rsq | Adj Rsq | LL | AIC |
|---:|---:|---:|---:|---:|
| 1 | 0.57 | 0.56 | 44.35 | -82.70 |
| 2 | 0.70 | 0.69 | 58.72 | -109.44 |
| 3 | 0.89 | 0.89 | 100.49 | -190.99 |
| 4 | 0.90 | 0.90 | 103.76 | -195.53 |
| 5 | 0.98 | 0.97 | 161.26 | -308.53 |
| 6 | 0.98 | 0.98 | 163.50 | -311.01 |
| 7 | 0.99 | 0.99 | 202.69 | -387.39 |
| 8 | 0.99 | 0.99 | 203.22 | -386.43 |
| 9 | 0.99 | 0.99 | 207.39 | -392.78 |

polynomial. This may seem to imply that higher order polynomials are more flexible and so generally lead to better fits to data. This is an important point to which we will return later in this section.

Note that because the polynomial regression is essentially a linear model, everything that we know about linear models and `lm` apply to polynomial regression too. For example, the summary output of the model, which we show below for the case of degree $K = 3$, is interpreted identically to any other `lm` model summary.

```
summary(M_eyefix_targ[[3]])
#>
#> Call:
#> lm(formula = mean_fix ~ poly(Time, degree = ., raw = T), data = eyefix_df_avg_targ)
#>
#> Residuals:
#>       Min        1Q    Median        3Q       Max
#> -0.140940 -0.053969  0.002774  0.062525  0.105314
#>
#> Coefficients:
#>                                     Estimate Std. Error t value Pr(>|t|)
#> (Intercept)                        1.043e-01  1.421e-02   7.336 1.91e-10 ***
#> poly(Time, degree = ., raw = T)1  2.220e-04  1.512e-05  14.681  < 2e-16 ***
#> poly(Time, degree = ., raw = T)2  1.623e-07  2.016e-08   8.051 8.13e-12 ***
#> poly(Time, degree = ., raw = T)3 -7.497e-11  6.359e-12 -11.790  < 2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.07177 on 77 degrees of freedom
#> Multiple R-squared:  0.8921, Adjusted R-squared:  0.8878
#> F-statistic: 212.1 on 3 and 77 DF,  p-value: < 2.2e-16
```

Thus, for example, the rate of the change of the average value of the outcome variable for a unit change on `Time`$^2$ is given by the coefficient for this variable, which is $1.6231344 \times 10^{-7}$.

## Orthogonal polynomials

In the polynomials we have just run, we used `poly` with `raw = T`. By setting `raw = F`, which is the default, we obtain *orthogonal polynomials*. This means that the predictor variables that represent the various powers of the predictor `Time` predictor are uncorrelated with one another. To understand the basics of orthogonal versus raw polynomials, let us generate a set of $K = 5$ polynomial functions of both types as follows.

```
x <- seq(-1, 1, length.out = 100)
y <- poly(x, degree = 5) # orthogonal
```

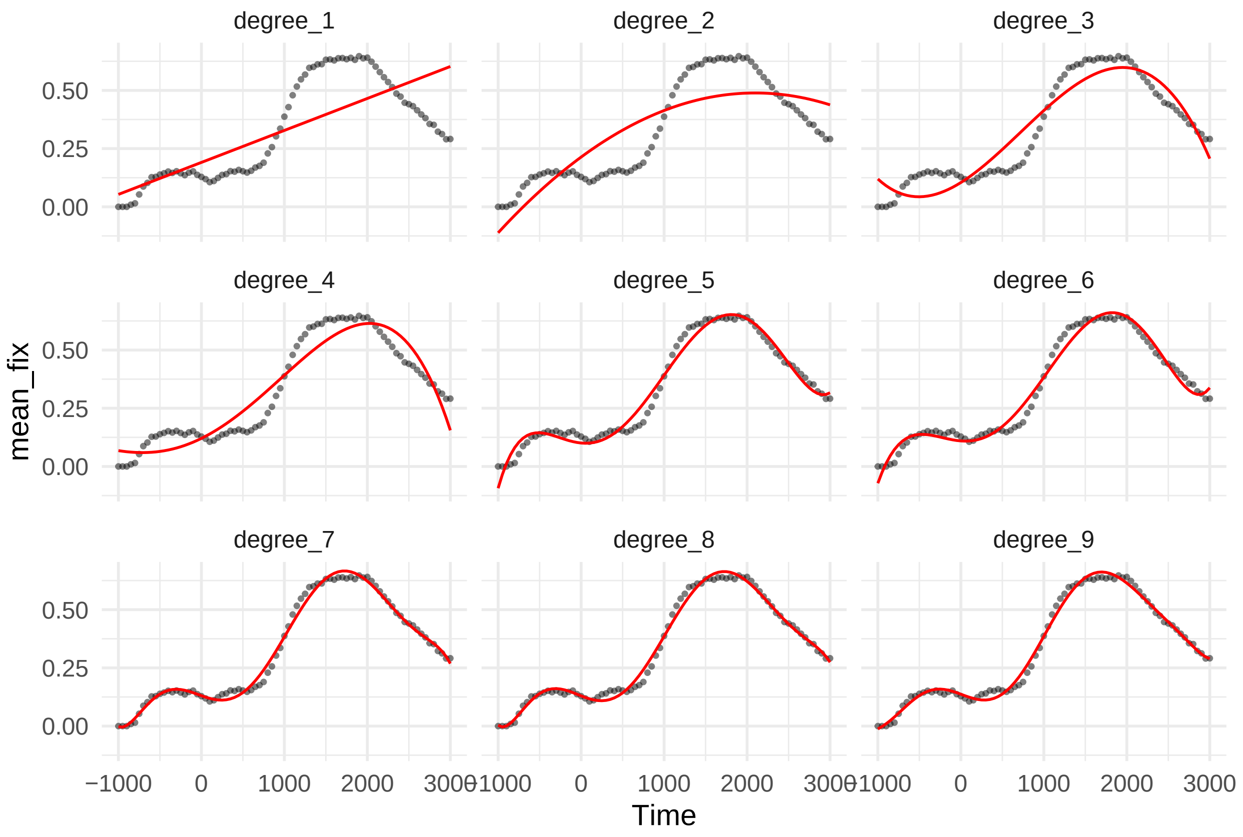

Figure 11: The predicted functions in a sequences of polynomial models, from degree 1 to degree 9.

```
y_raw <- poly(x, degree = 5, raw = T) # raw
```

Now, let us look at the inter-correlation matrix of the 5 orthogonal vectors.

```
cor(y) %>%
  round(digits = 2)
#>   1 2 3 4 5
#> 1 1 0 0 0 0
#> 2 0 1 0 0 0
#> 3 0 0 1 0 0
#> 4 0 0 0 1 0
#> 5 0 0 0 0 1
```

As we can see, they all have zero correlation with one another. By contrast, the raw polynomials are highly intercorrelated.

```
#>      1    2    3    4    5
#> 1 1.00 0.00 0.92 0.00 0.82
#> 2 0.00 1.00 0.00 0.96 0.00
#> 3 0.92 0.00 1.00 0.00 0.98
#> 4 0.00 0.96 0.00 1.00 0.00
#> 5 0.82 0.00 0.98 0.00 1.00
```

Orthogonal polynomials are also usually rescaled so that each vector has a Euclidean length of exactly 1.0.

```
euclidean <- function(x) sqrt(sum(x^2))
apply(y, 2, euclidean)
#> 1 2 3 4 5
#> 1 1 1 1 1
apply(y_raw, 2, euclidean)
#>        1        2        3        4        5
#> 5.831529 4.562178 3.893977 3.467986 3.167594
```

We provide a plot of orthogonal polynomials from degree 1 to degree 5 in Figure 12.

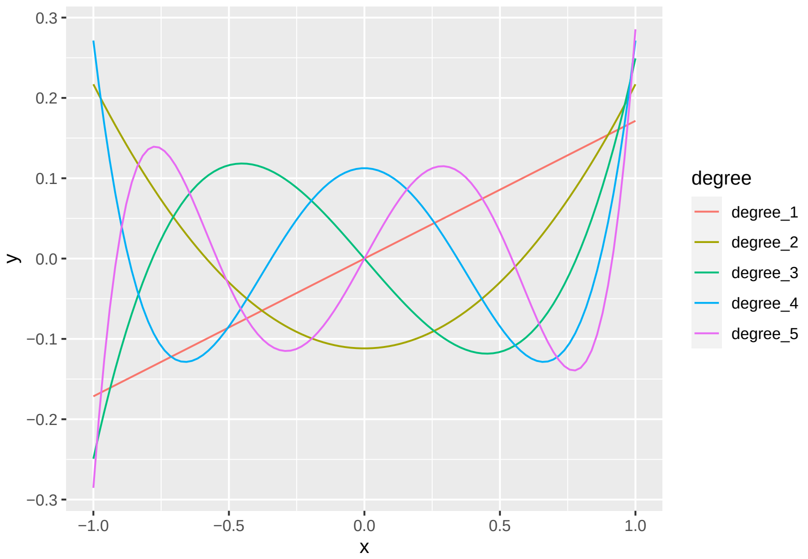

Figure 12: Plots of orthogonal polynomial functions of $x \in (-1, 1)$ for degree 1 to 5.

Using orthogonal polynomials has computational and conceptual consequences. Computationally, it avoids any multicollinearity. Multicollinearity arises when any predictor variables can be predicted by some or all of the other predictors. In practice, there is almost always some multicollinearity because predictors in real world data sets are often correlated, at least to a minimal extent. Extreme multicollinearity can lead to numerical instability in the inference algorithms. But more generally, multicollinearity leads to the variance of the estimates of the coefficients (i.e., the square of the standard error of the estimate) being inflated. Conceptually, on the other hand, orthogonal predictor variables entail that the coefficient for each predictor is the same regardless of which of the other predictors are present, or indeed whether any of the other predictor are present. In other words, the coefficient for the predictor corresponding to `Time` raised to the $k$th power will be the same in every regression that includes this term or if it were used as the only predictor. The coefficient for each predictor in the orthogonal polynomial regression gives the independent contribution of the various powers of the original predictor.s

In the following, we rerun the above analysis using orthogonal predictors.

```
M_eyefix_targ_orth <- map(seq(9),
                          ~lm(mean_fix ~ poly(Time, degree = .),
                              data = eyefix_df_avg_targ)
)
```

Let us look, for example, at the coefficients in the first models from degree 1 to degree 4.

```
map(M_eyefix_targ_orth[1:4],
    ~coef(.) %>% unname())
#> [[1]]
#> [1] 0.3280476 1.4449036
#>
#> [[2]]
#> [1]  0.3280476  1.4449036 -0.6884575
#>
#> [[3]]
#> [1]  0.3280476  1.4449036 -0.6884575 -0.8461469
#>
#> [[4]]
#> [1]  0.3280476  1.4449036 -0.6884575 -0.8461469 -0.1753745
```

As we can see, the coefficient corresponding to any given particular power is the same in each model. Moreover, because the orthogonal polynomials are all normalized, they are on the same scale. As such, from the coefficient alone we can immediately see the effect size of each power of the original predictor.

It is important to remember, however, that the orthogonal predictors are no longer simply $x^0, x^1, x^2, \ldots x^K$. They are based on a (relatively complex) transformation of this $K+1$ dimensional vector space in a manner comparable to what is done in *principal components analysis*. In addition, it should also be noted that othogonal polynomials of any given degree will still lead to an *identical* model fit to that obtained using the raw polynomials. Overall then, the choice of orthogonal polynomials is motivated by the potential advantages in their interpretation, and also for the numerical properties.

## Polynomial regression using other variables

Because we are using `poly` inside `lm` effectively to create a new set of predictors, how we use and interpret the model results in a polynomial regression are no different to how they are when using `lm` generally. For example, in the following, see analyse how the average value of `mean_fix` varies as polynomial function of time for each of the three different object categories, rather than just the `Target` category.

We show the model fit for this in Figure 13.

In this model, the `Object` variable is a categorical variable and so we are creating a different polynomial function for each `Object` category (`Target`, `Competitor`, `Unrelated`) by inferring polynomial coefficients for the base category (which is, by default, `Competitor`, based on the alphabetical order of the category

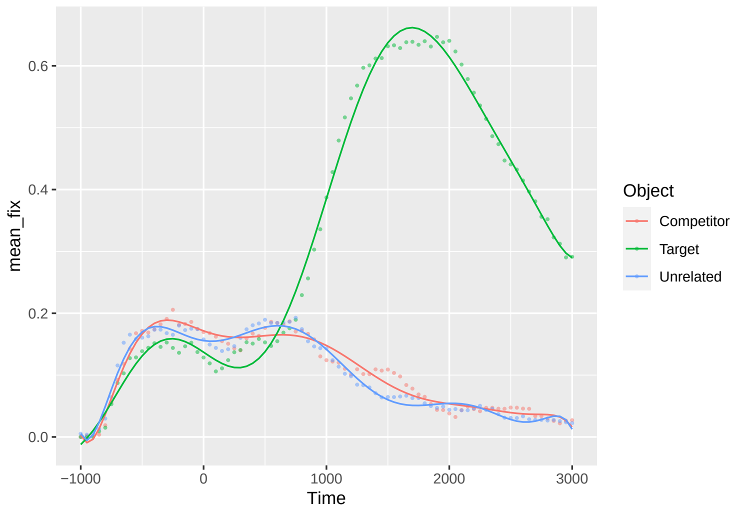

Figure 13: The predicted functions for proportion of looking times to each category using a 9 degree polynomial. In this case, we use the categorical predictor `Object` to vary to coefficients for the polynomial regression for the three object categories.

names) and then differences in these coefficients corresponding to the `Target` and `Unrelated` categories. For example, in the following, we show the difference in the quadratic term, i.e. representing `Time` raised to the power of 2, between the `Competitor` and the `Target` category.

```
coef(M_eyefix) %>%
  extract('poly(Time, 9)2:ObjectTarget')
#> poly(Time, 9)2:ObjectTarget
#>                  -0.6477649
```

This tells us that that coefficient changes by a value of approximately -0.648 from the from the `Competitor` to the `Target` category.

Although this analysis was as easy to perform as a varying slopes and varying intercept linear model, the interpretion of the model is more challenging. We can easily compare the `M_eyefix` model against a null alternative model, where `Object` is used as an intercept only term.

```
M_eyefix_null <- lm(mean_fix ~ Object + poly(Time, 9), data=eyefix_df_avg)
anova(M_eyefix_null, M_eyefix)
#> Analysis of Variance Table
#>
#> Model 1: mean_fix ~ Object + poly(Time, 9)
#> Model 2: mean_fix ~ poly(Time, 9) * Object
#>   Res.Df    RSS Df Sum of Sq      F    Pr(>F)
#> 1    231 3.3021
#> 2    213 0.0470 18    3.2551 819.82 < 2.2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This shows that the polynomial functions for the `Target`, `Competitor`, and `Unrelated` `Object` categories do not differ simply in terms of their intercept terms. However, beyond that, it is not simple matter to say where and how the three different polynomial functions differ from one another, and so it is not a simple matter to explain the effect of `Object` on the time course of the eye fixation proportions in this experiment. This is common issue with nonlinear regression models that we mentioned in the introduction to this chapter. Ultimately, the polynomial regression model, as with nonlinear regression more generally, provides us with a relatively complex probabilistic model of the phenomenon we are studying. However, key features of interest, such as the role played by one predictor variable, can not be isolated as easily to role of individual parameters.

## Overfitting in polynomial regression

Let us consider the following simple data set, which is also plotted in Figure 14.

```r
set.seed(101)
Df <- tibble(x = seq(-2, 2, length.out = 20),
             y = 0.5 + 1.5 * x + rnorm(length(x))
)
Df %>% ggplot(aes(x,y)) + geom_point()
```

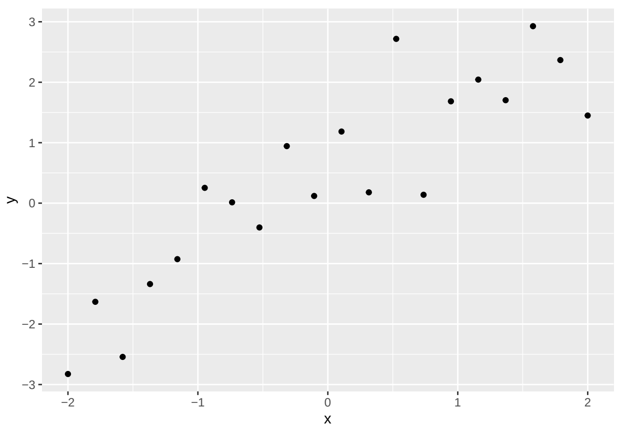

Figure 14: Data generated by a simple linear model, with intercept 0.5, slope 1.5, and noise standard deviation $\sigma = 1$.

Although we know, because we have generated it, that this data is nothing more than data from a linear normal model, if this were real world data, we simply would not know this. We could therfore see how well each in a sequence of increasingly complex polynomial regression models fits this data. These fits are shown Figure 15, where we also show the $R^2$ value for each polynomial.

As the degree of the polynomial model increases, so too does its fit to the data. However, this fit to the data is essentially an *overfit*. There is no precise general definition of overfitting, but in this example, it is clearly the case that the overfitted model is fitting the noise rather than the true underlying function, which we know in this case is just a linear model. We can see how the functions in higher order polynomials are bending to fit individual data points. Overfitted models do not generalize well to new data. This is easy to observe in this example. We generate a new set of observations from the same true model that was used to generate the original data, and then see how well this new data is predicted by each of the 9 previously fitted models. We can measure the model predicts the new data sets by calculating $R^2_{\text{new}}$ scores for each new data set as follows:

$$R^2_{\text{new}} = 1 - \frac{RSS_{\text{new}}}{TSS_{\text{new}}} = 1 - \frac{\sum_{i=1}^{n}(y_i^{\text{new}} - \hat{y}_i^{\text{new}})^2}{\sum_{i=1}^{n}(y_i^{\text{new}} - \bar{y}^{\text{new}})^2},$$

where $y_1^{\text{new}} \ldots y_n^{\text{new}}$ are the outcome variable values in the new data set, $\bar{y}^{\text{new}}$ is the mean of these values, and $\hat{y}_1^{\text{new}} \ldots \hat{y}_n^{\text{new}}$ are the predicted values of the outcome variable for the model. Obviously, the lower the value of $R^2_{\text{new}}$, the worse the model generalization performance. We will generate 1000 new data sets from the same data generating process as the original model, and for each data set calculated $R^2_{\text{new}}$ for each of the 9 polynomial models. The boxplot of the distribution of the $R^2_{\text{new}}$ scores for each polynomial model are shown in Figure 16. Clearly, as the degree of the polynomial increases, its out-of-sample generalization performance decreases.

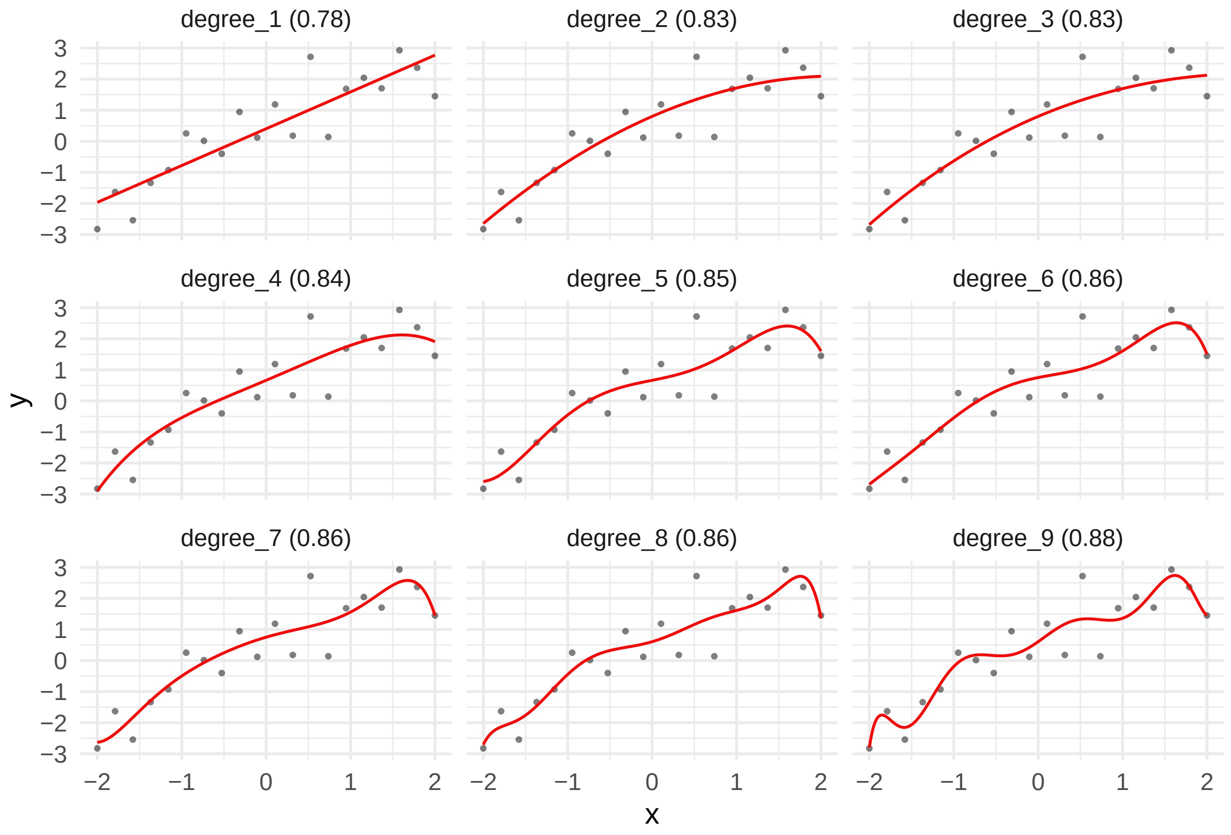

Figure 15: Fitting a data with a sequence of polynomials from degree 1 to degree 9. Shown in the label for each model is its $R^2$ value. Clearly, the fit to the data is increasing but at a cost of *overfitting*.

It should be noted that model evaluation methods such as AIC are explicitly designed to measure out-of-sample generalization and hence help us to identify overfitted models. We may calculate the AIC value for each of the model above as follows.

```
map_dbl(M_overfits, AIC) %>%
  set_names(paste0('degree_', seq(M_overfits))) %>%
  round(2)
#> degree_1 degree_2 degree_3 degree_4 degree_5 degree_6 degree_7 degree_8
#>    51.40    48.41    50.40    51.73    51.79    53.49    55.31    56.51
#> degree_9
#>    56.25
```

As we can see, although the AIC model is lower for polynomials of degree 2 and 3 than for degree 1, the drop is not by much and after that, as the degree of the polynomial increases, so too does the AIC value.

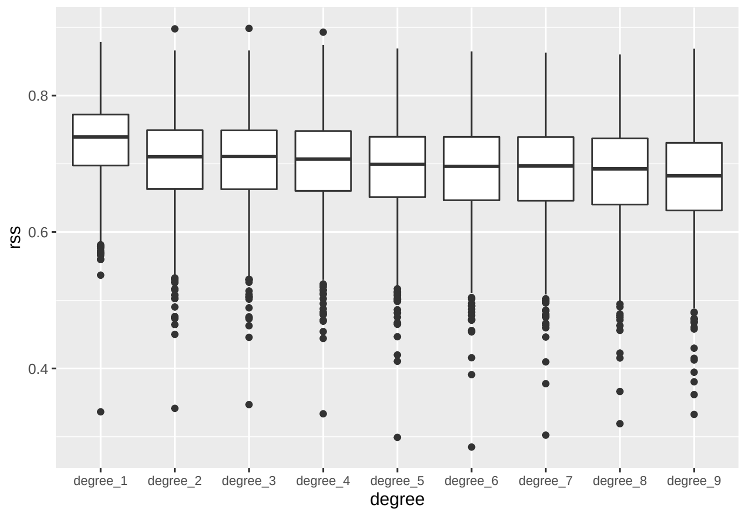

Figure 16: For each polynomial model, we show the boxplot of the distribution of $R^2$ scores of the predictions of new data sets generated from the same true data generating process as the original data, which was a normal linear model. The lower the value, the poorer the out of sample generalization of the model. As such, the more complex the polynomial model, the worse its generalization performance despite the fact that the more complex models fit the original data better.

Overfitting is a general problem in statistical modelling. However, polynomial regression is especially prone to overfitting. This is because higher order polynomials are too unconstrained. The higher the order of the polynomial, the more it can twist and bend to fit the data. This is not always avoided by simply sticking to lower order polynomials because lower order polynomials *underfit* the data, having insufficient flexibility to fit the function. Thus a common problem with polynomial regression is that the lower order polynomials are not flexible enough, and the higher order ones are too unconstrained. Moreover, polynomial regression is also prone to a pathology related *Runge's phenomenon*, which is where there is excessive oscillation in the polynomial function paricularly at its edges. We can see this easily by increasing the order of the polynomial to 16. This model is shown in Figure 17.

## Spline and basis function regression

Polynomial regression can be seen as a type of *basis function* regression. In basis function regression, we model the nonlinear functions of the predictors variables using linear combinations of simpler functions that are known as the basis functions. For example, in the case of a nonlinear regression with one predictor variable, and assuming normally distributed outcome variables, we would write our basis function regression

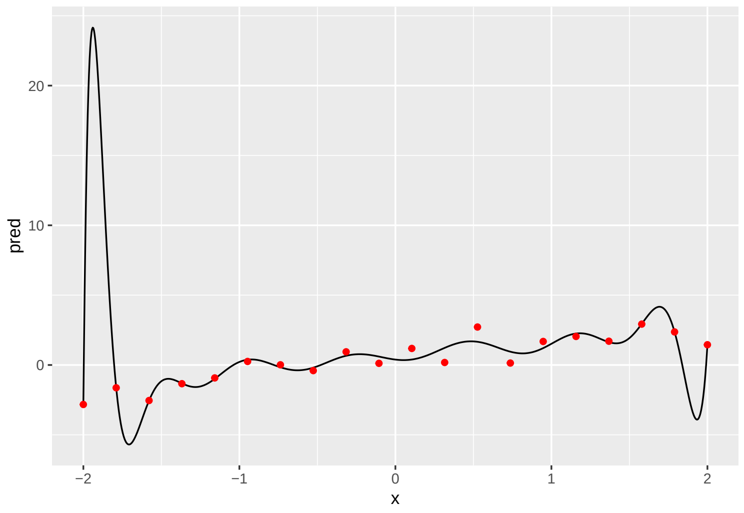

Figure 17: In higher order polynomials, there is often excessive oscillation between fitted points, particularly at the edge of the function. Here, we plot a 16th order polynomial. At edges of the function, we extreme oscilliation between the fitted points.

model as follows:

$$y_i \sim N(\mu_i, \sigma^2), \quad \mu_i = f(x_i) = \sum_{k=1}^{K} \beta_k \phi_k(x_i), \quad \text{for } i \in 1 \ldots n,$$

or if we include an explicit intercept term

$$\mu_i = f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k \phi_k(x_i), \quad \text{for } i \in 1 \ldots n.$$

Here, $\phi_1(x_i), \phi_2(x_i) \ldots \phi_k(x_i) \ldots \phi_K(x_i)$ are (usually) simple deterministic functions of $x_i$. In polynomial regression, our basis functions are defined as follows:

$$\phi_k(x_i) \triangleq x_i^k.$$

We saw in Figure 9 what weighted sums of these functions look like.

## Cublic b-splines

Using basis functions for nonlinear regression is widely practiced. There are many different types of basis functions that are possible to use, but one particularly widely used class of basis functions are *spline* functions. The term *spline* is widely used in mathematics, engineering, and computer science and may refer to many different types of related functions, but in the present context, we are defining splines as piecewise polynomial functions that are designed in such a way that each piece or segment of the function joins to the next one without a discontinuity. As such, splines are smooth functions composed of multiple pieces, each of which is a polynomial. Even in the context of basis function regression, there are many types of spline functions that can be used, but one of the most commonly used types is *cubic b-splines*. The *b* refers to *basis* and the *cubic* is the order of the polynomials that make up the pieces. Each cubic b-spline basis function is defined by 4 curve segments that join together smoothly. The breakpoints between the intervals on which these curves are defined are known as *knots*. If these knots are equally spaced apart, then we say that the spline is *uniform*. For basis function $k$, its knots can be stated as

$$t_0^k < t_1^k < t_2^k < t_3^k < t_4^k,$$

so that the curve segments are defined on the intervals $(t_0^k, t_1^k]$, $(t_1^k, t_2^k]$, $(t_2^k, t_3^k]$, $(t_0^3, t_4^k)$. Spline basis function $k$ takes the value of 0 for values less than $t_0^k$ or values greater than $t_4^k$. The cubic b-spline is then defined as follows:

$$\phi_k(x_i) = \begin{cases} \frac{1}{6}u^3, & \text{if } x_i \in (t_0^k, t_1^k], \quad \text{with } u = (x_i - t_0^k)/(t_1^k - t_0^k) \\ \frac{1}{6}(1 + 3u + 3u^2 - 3u^3), & \text{if } x_i \in (t_1^k, t_2^k], \quad \text{with } u = (x_i - t_1^k)/(t_2^k - t_1^k) \\ \frac{1}{6}(4 - 6u^2 + 3u^3), & \text{if } x_i \in (t_2^k, t_3^k], \quad \text{with } u = (x_i - t_2^k)/(t_3^k - t_2^k) \\ \frac{1}{6}(1 - 3u + 3u^2 - u^3), & \text{if } x_i \in (t_3^k, t_4^k), \quad \text{with } u = (x_i - t_3^k)/(t_4^k - t_3^k) \\ 0 & \text{if } x_i < t_0^k \text{ or } x_i > t_4^k \end{cases}$$

In Figure 18, we plot a single cubic b-spline basis function defined on the knots $\{-\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}\}$. In this figure, we have colour coded the curve segments.

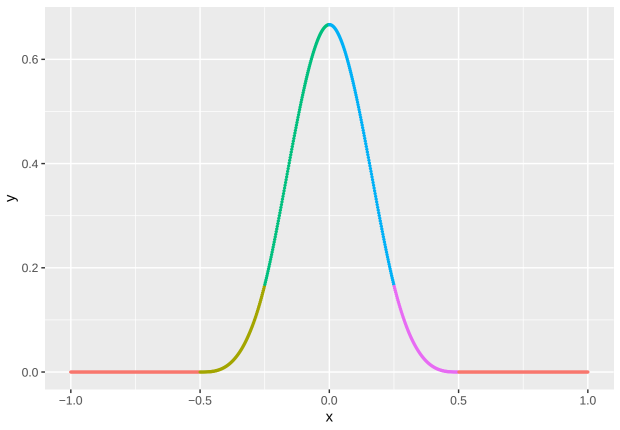

Figure 18: A single cubic b-spline basis function defined on the knots $\{-\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}\}$.

In any basis function regression, including spline regression, we usually have many basis functions. In the case of spline regression, the number and location of the basis functions are defined by the position of the knots. In the case of any one basis function, as we've seen, only five knots are used. However, if there are many more knots, a basis function is defined on each set of 5 consecutive knots. In other words, if our knots are $t_0, t_1, t_2 \ldots t_K$, one basis function is defined on knots $t_0, t_1, t_2, t_3, t_4$, the next is defined on knots $t_1, t_2, t_3, t_4, t_5$, and so on. In Figure 19a, we provide examples of multiple basis functions defined on different consecutive sequences of a set of knots spaced $\frac{1}{2}$ apart, from -2 to 2. In practice, sometimes lower order spline are fitted to the sequences of less 5 knots at the start and end of the set of knots, e.g., the sequences $t_0, t_1, t_3$, etc. The function `bs` available from the `splines` package creates b-spline basis functions in this way. In Figure 19b, we provide the basis functions created by `splines::bs` for the same set of knots as were used in 19a.

The simplest way to perform spline regression in R is to use `splines::bs`, or a related function from the `splines` package, just as we used `poly` for polynomial regression. For example, to perform a cubic b-spline regression on our eye-fixation rates to the three object categories, we could do the following.

```r
library(splines)
knots <- seq(-500, 2500, by = 500)
M_bs <- lm(mean_fix ~ bs(Time, knots = knots)*Object,
           data=eyefix_df_avg)
```

The model fit for this model is shown in Figure 21. Notice that we spaced the knots evenly from -500 to 2500 in steps of 500 ms. This gives us 7 explicitly supplied knots, at which there will be a basis function. In addition, however, the `bs` function provides 3 (assuming `degree=3`, which is the default) extra lower degree basis functions at the boundaries. From the model summary, we can see that we have an extremely high model fit, $R^2 = 0.994$. As before, the very high $R^2$ value must be treated with some initial caution, as it may indicate model overfit, a point to which we will return momentarily.

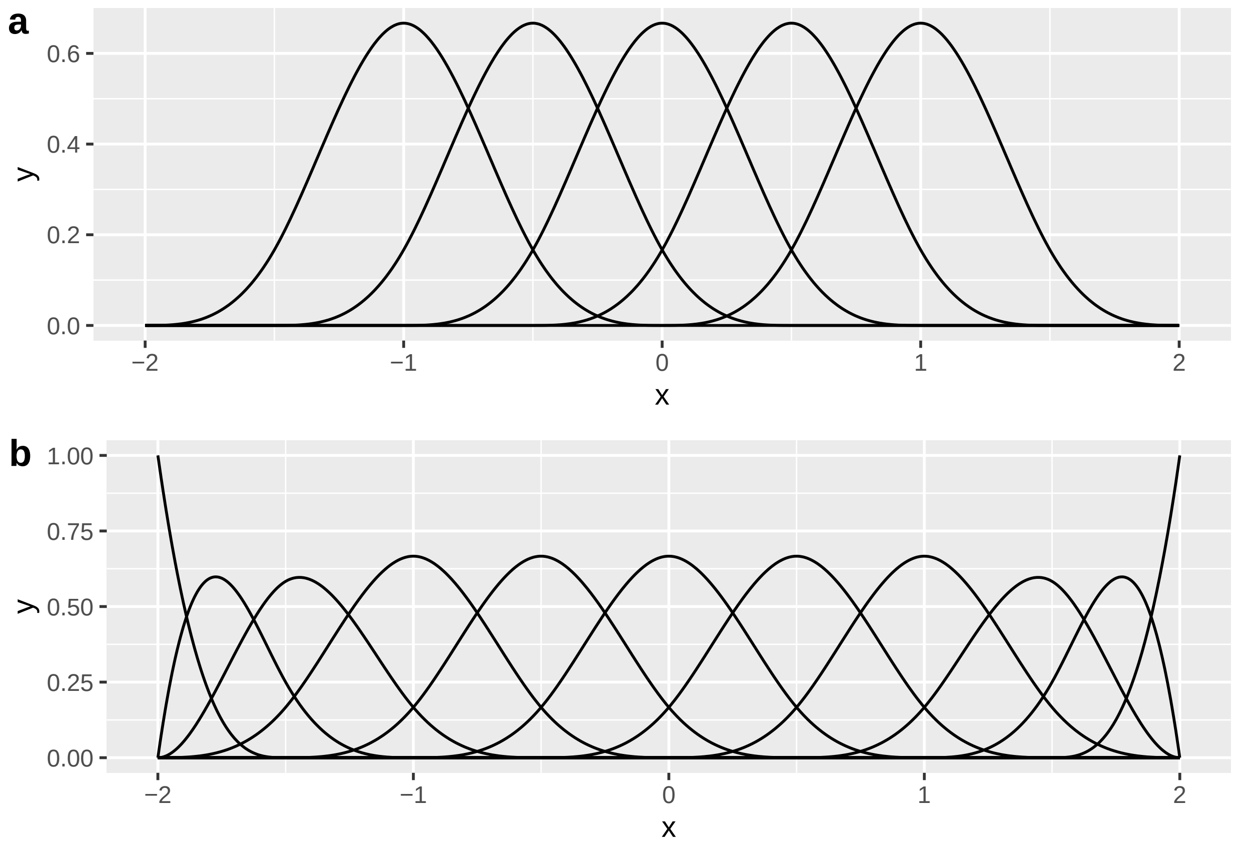

Figure 19: a) A set of cubic b-spline basis functions defined on each consective set of 5 knots, spaced $\frac{1}{2}$ apart, from -2 to 2. b) The set of cubic b-spline basis functions defined on the set same knots as in a) generated by `splines::bs` function, which includes lower order b-splines at the edges of the set of knots.

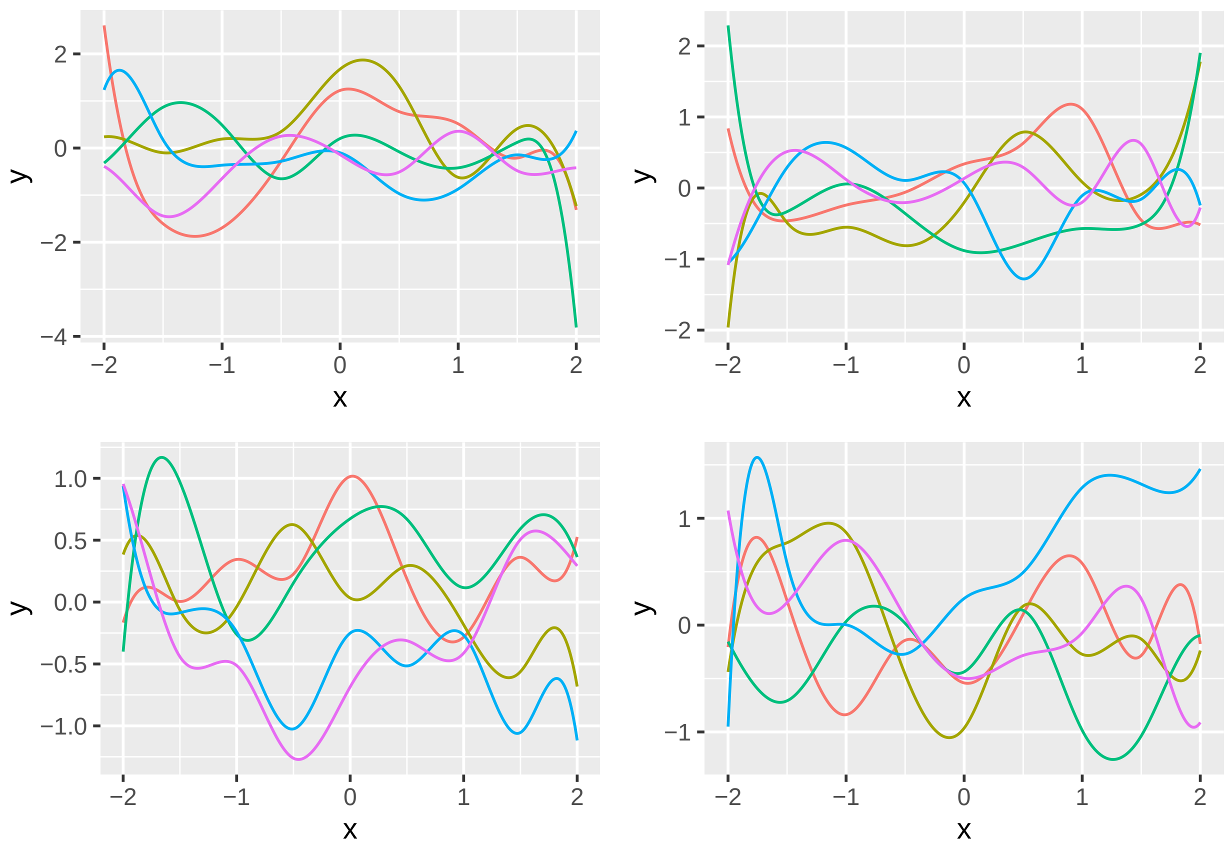

Figure 20: Examples of random sums of cubic b-splines basis functions. In each subplot, we have five random functions generated by different weightings of the same set of basis functions, which were defined on knots spaced $\frac{1}{2}$ apart, from -2 to 2. .

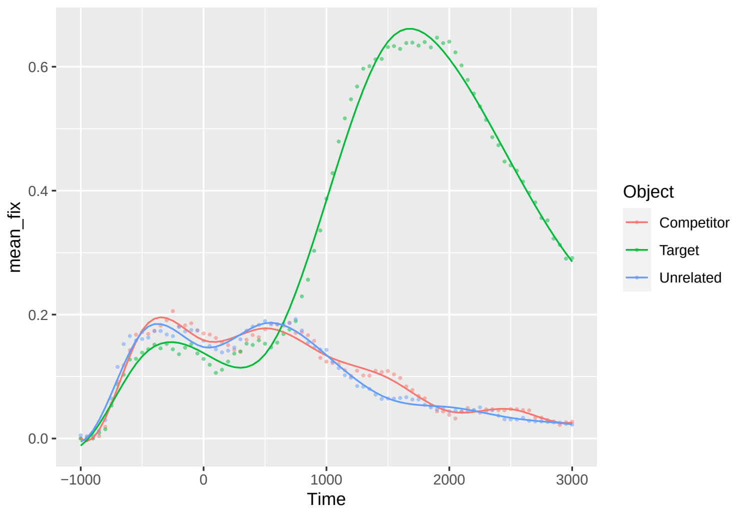

Figure 21: The fit of a cubic b-spline, with evenly spaced basis functions every 500ms, to the average eye fixation rates to each `Object` category.

An alternative approach to using `bs` with `lm` to perform b-spline regression is to not explicitly choose the location of the knots, but rather to let `bs` choose them at evenly spaced quantiles. We can accomplish this using the `df` parameter. If we set `df` to some $K$, then `bs` will find $K - 3$ knots. For example, the following code will perform a spline regression very similar to `M_bs`, and with the same number of basis function, but with the knot locations chosen based on quantiles.

```
M_bs_df10 <- lm(mean_fix ~ bs(Time, df = 10)*Object,
                data=eyefix_df_avg)
```

We can see the location of the `df - 3` knots by extracting the `knots` attribute from the object created by `bs` as follows.

```
with(eyefix_df_avg,
     attr(bs(Time, df = 10), 'knots')
)
#> 12.5%   25% 37.5%   50% 62.5%   75% 87.5%
#>  -500     0   500  1000  1500  2000  2500
```

In this case, the knots happen to be in the same location as when explicitly made them.

## Radial basis functions

An alternative, though related, class of basis functions to spline basis functions are *radial basis functions* (RBF). In these basis functions, the value the function takes is defined by the distance of the input value from a fixed center. As an example, one of the most commonly used RBF models is the *Gaussian* or *squared exponential* RBF defined as follows.

$$\phi(x) = e^{-\frac{|x-\mu|^2}{2\sigma^2}}.$$

An example of Gaussian RBF, with $\mu = 0$ and $\sigma = 1$, is shown in Figure 22.

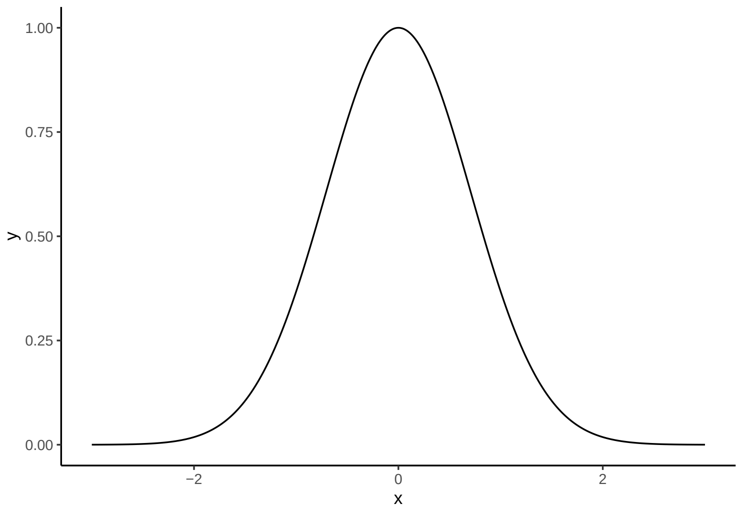

Figure 22: A Gaussian radial basis function (RBF) is essentially an unnormalized Normal distribution. In this figure, we display a Gaussian RBF that is centered at $\mu = 0$ and has a width parameter $\sigma = 1.0$. This is identical to an unnormalized Normal distribution with a mean of 0 and standard deviation of 1.0.

These functions are identical to unnormalized Normal distributions where the two parameters of the RBF play the same role as the mean and the standard deviation of the Normal distribution. In Figure 23, we display random sums of 9 RBF functions defined at defined the centres -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2. In subfigures a-d in 23, we set the $\sigma$ parameter of the RBFs to $0.25, 0.5, 1.0, 2.0$, respectively. As is clear from these subfigures, as the $\sigma$ parameter increases, the resulting functions become smoother.

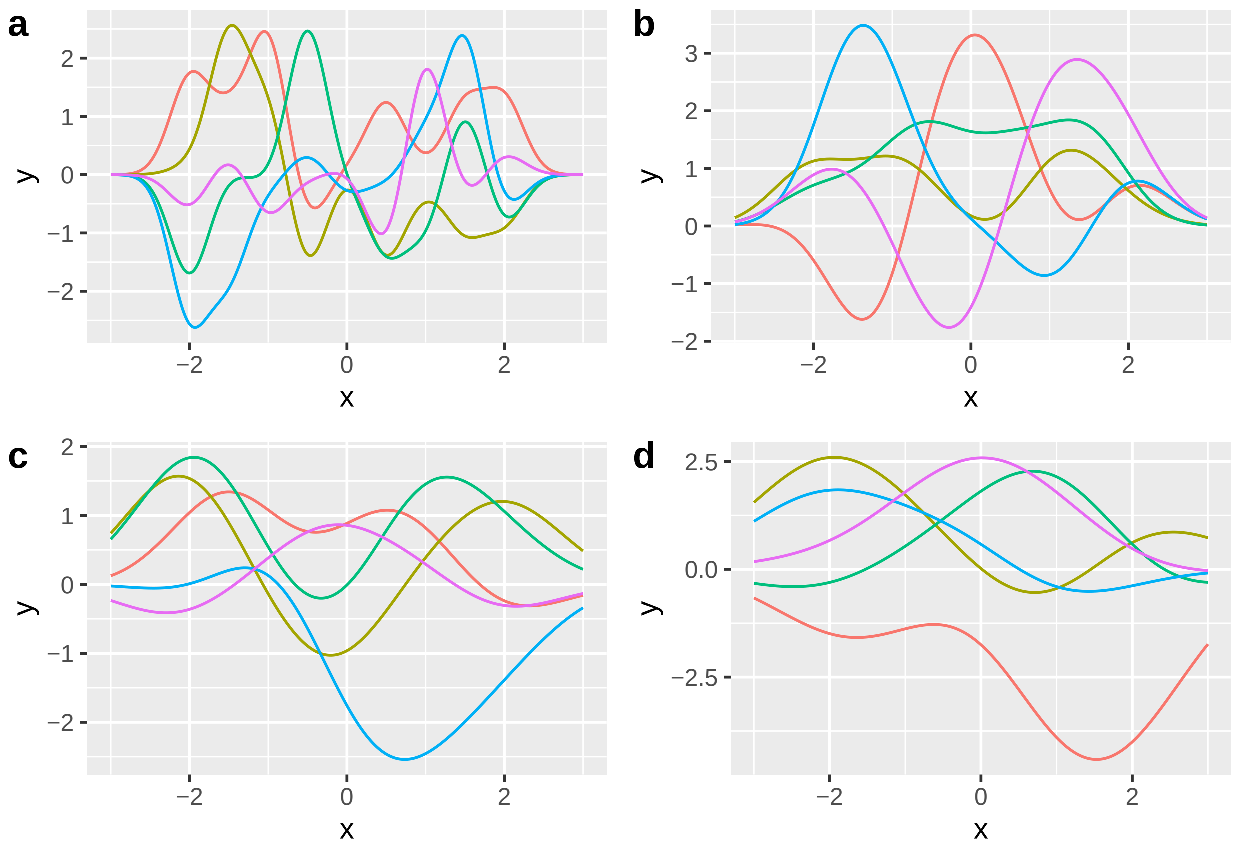

Figure 23: Examples of random sums of Gaussian RBFs. In each subplot, we have five random functions generated by different weightings of the same set of RBFs, which were defined on centres spaced $\frac{1}{2}$ apart, from -2 to 2. The width parameter $\sigma$ of the RBF*s*, on the other hand, take the values of $0.25, 0.5, 0.75, 1.0$ in subfigures a-d, respectively. Clearly, as $\sigma$ increases, the resulting functions become more smooth.

We can perform a RBF regression using `lm` similarly to how we used `lm` with `poly` or `splines:bs`. To do so, we will create a custom `rbf` function that returns the values of set of Gaussian RBF functions defined at specified centres and with a common width parameter.

```r
rbf <- function(x, centres, sigma = 1.0){
  map(centres,
      ~exp(-(x-.)^2/(2*sigma^2))
  ) %>% do.call(cbind, .)
}
```

We may then use this `rbf` function inside `lm` by choosing the location of the centres, which we set to be at every 250ms beween -1000 and 3000 ms, and the width parameter, which we set to be 500. We will use the `eyefix_df_avg` data set as before.

```r
centres <- seq(-1000, 3000, by = 250)
M <-lm(mean_fix ~ rbf(Time, centres = centres, sigma = 500)*Object,
       data=eyefix_df_avg)
```

The predictions of this model are shown in Figure 24.

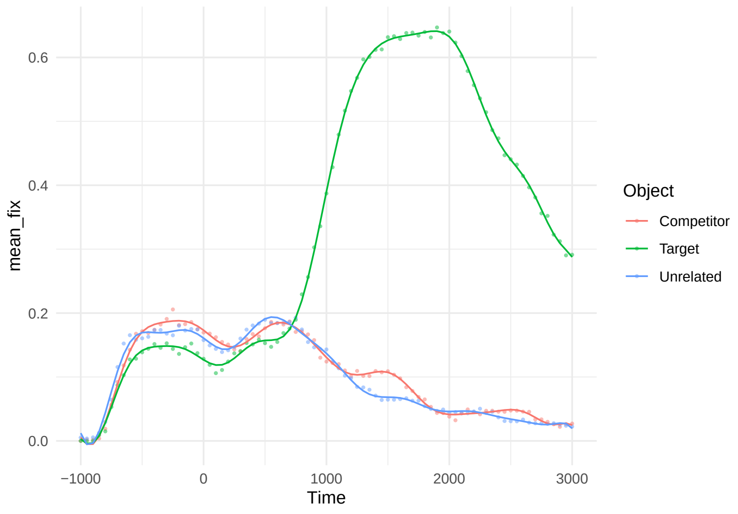

Figure 24: The fit of a Gaussian RBF, with centres at every 250ms and $\sigma = 500$, to the average eye fixation rates to each `Object` category.

## Choosing basis function parameters

A persistent and major issue in basis function regression is choosing between or evaluating the different parameters of the basis functions. In the case of cubic b-splines, for example, this would primarily concern the choice of the number and location of the knots. Other basis functions, as we will see, have other parameters whose values must also be chosen. Although this issue can in principle be treated as just another type of parametric inference, i.e., where the basis function parameters are inferred along with the standard regression coefficients and the standard deviation of the outcome variable, doing so can often be technically very difficult. As a result, more commonly, this issue is treated as a model evaluation issue. In other words, evaluating or choosing between the number and location of knots in a spline regression can be seen as analogous, for example, to choosing between different parametric functions performing parametric nonlinear regression using `nls`. To do so, we typically employ a wide range of model evaluation methods including model fit statistics to guide our choices. Efficient methods for making these choices, as well as making other modelling choices with basis function regression, will be discussed in the next section on *generalized additive models*. However, for now, we will begin looking at this issue using some relatively simple general methods. For example, we evaluate choices concerning knots in spline regression by using the AIC values for each model in a sequence of spline models with increasing numbers of basis functions. Rather than using the standard AIC, however, we will use $\text{AIC}_c$ defined as follows:

$$\text{AIC}_c = \text{AIC} + \frac{2k(k+1)}{n-k-1}.$$

This can be implemented as follows:

```r
aic_c <- function(model){
  K <- length(coef(model))
  N <- nrow(model$model)
  AIC(model) + (2*K*(K+1))/(N-K-1)
}
```

This correction is generally advised when the ratio of sample size $n$ to number of parameters $k$ is relatively low, as it would be in the case.

Here, we will consider a new data set GSSvocab, available from GSSvocab.csv, that provides scores on a vocabulary test for a relatively large number of people, collected over the course of a few decades. For simplicity, we will examine how vocabulary test scores vary with age, excluding the other variables in the data set. For this, we obtain the average vocabulary score for each year of age for a sequence of years from 18 to 89.

```r
GSSvocab <- read_csv('GSSvocab.csv')
gssvocab <- GSSvocab %>%
  group_by(age) %>%
  summarize(vocab = mean(vocab, na.rm=T)) %>%
  ungroup() %>%
  drop_na()
```

This data is shown in Figure 25.

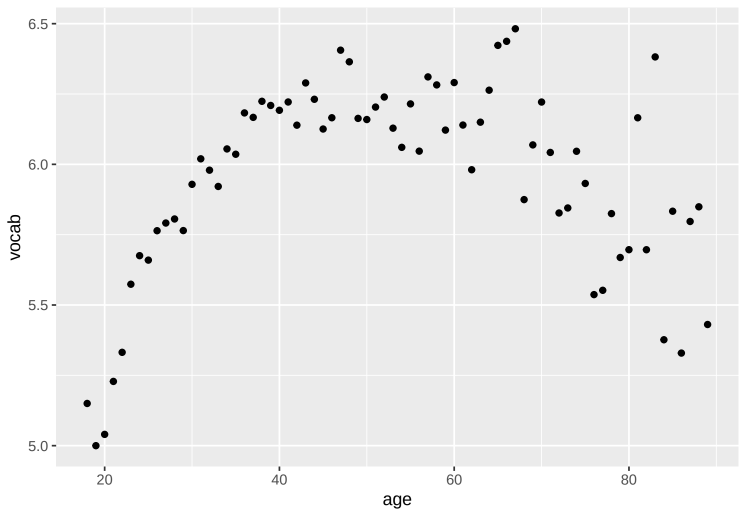

Figure 25: Average score on a vocabulary test for each year of age in a sequences of years from 18 to 89.

Let us now fit a sequence of cubic b-spline regression model to this data, where we vary the number of knots from a minimum of 3 to 30. This is done in the following code where we use so-called *natural* cubic b-splines, using the `splines::ns` function. Cubic b-splines of this kind forces the constraint that the function is linear beyond the knot boundaries.

```r
df_seq <- seq(3, 30) %>%
  set_names(.,.)

M_gssvocab <- map(df_seq,
                  ~lm(vocab ~ ns(age, df = .),
                      data = gssvocab)
)


aic_results <- map_dbl(M_gssvocab, aic_c)%>%
  enframe(name = 'df',
          value = 'aic')
```

For comparison with the AIC$_c$ result, we will also perform a *leave-one-out cross-validation* (LOOCV). This is where we remove one observation from the data, fit the model on the remaining data, and then see how well the we can predict the outcome variable's value in the held-out data. In a data set with $n$ observations, we leave each observation out and so it requires $n$ repetitions of the model fitting process.

```r
loocv <- function(K){
  map_dbl(seq(nrow(gssvocab)),
          function(i){
            Df_train <- gssvocab %>% slice(-i)
            Df_test <- gssvocab %>% slice(i)
            M <- lm(vocab ~ ns(age, df = K), data = Df_train)
            Df_test %>%
              add_predictions(M) %>%
              summarize(rss = (vocab - pred)^2) %>%
              unlist()
          }
  ) %>% sum()
}

loocv_results <- map_dbl(df_seq, loocv) %>%
  enframe(name = 'df',
          value = 'loocv')
```

The minimum AIC$_c$ score is -38.35, which occurs at a `df` of 6. The minimum LOOCV score is 2.38, which occurs at a `df` of 6.

In Figure 26a, we show the difference between the AIC$_c$ scores at all `df` values and the minimum AIC$_c$ score, and in Figure 26b, we how the differences between the LOOCV scores at all `df` values and the minimum LOOCV score. As we can see, the best models have a low number of knots. High values of `df` tend to become extremely poor.

## Generalized additive models

The polynomial and spline regression models that we have covered in the previous two sections can be regarded as special cases of a more general type of regression model known as a *generalized additive model* (GAM). GAMs are quite a general class of regression models, incorporating many special cases, and because of this, a single formal definition, while technically possible, may not be particularly clear. It is better, therefore, to define GAMs by a series of different definitions. We may begin our definitin of GAMs as follows. Given $n$ observations of a set of $L$ predictor variabes $x_1, x_2 \ldots x_l \ldots x_L$ and outcome variable $y$, where $y_i, x_{1i}, x_{2i} \ldots x_{li} \ldots x_{Li}$ are the values of the outcome and predictors on observation $i$, then a GAM regression model of this data is:

$$y_i \sim D(\mu_i, \psi), \quad \mu_i = f_1(x_{1i}) + f_2(x_{2i}) + \ldots + f_L(x_{Li}), \quad \text{for } i \in 1 \ldots n,$$

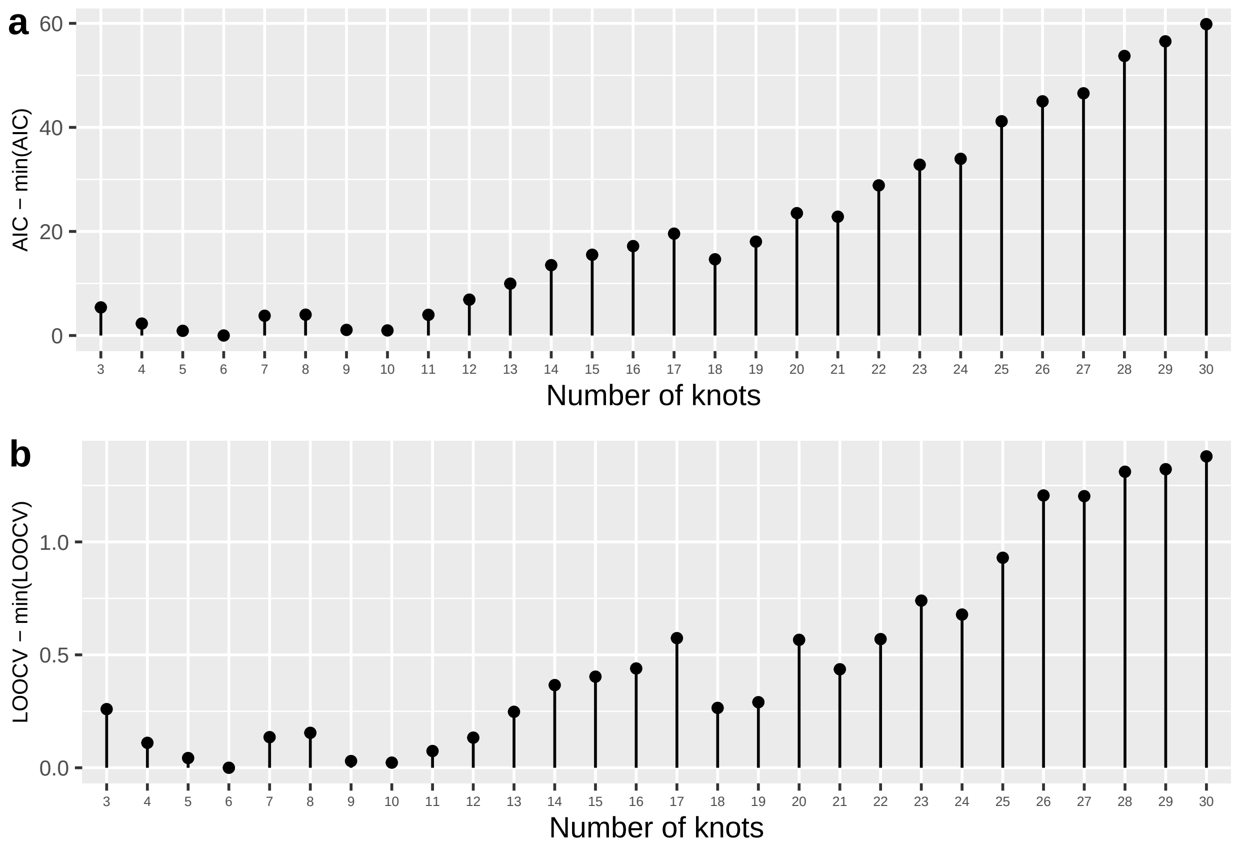

Figure 26: Differences in a) AIC$_c$ scores and b) LOOCV scores between each model and the best fitting model.

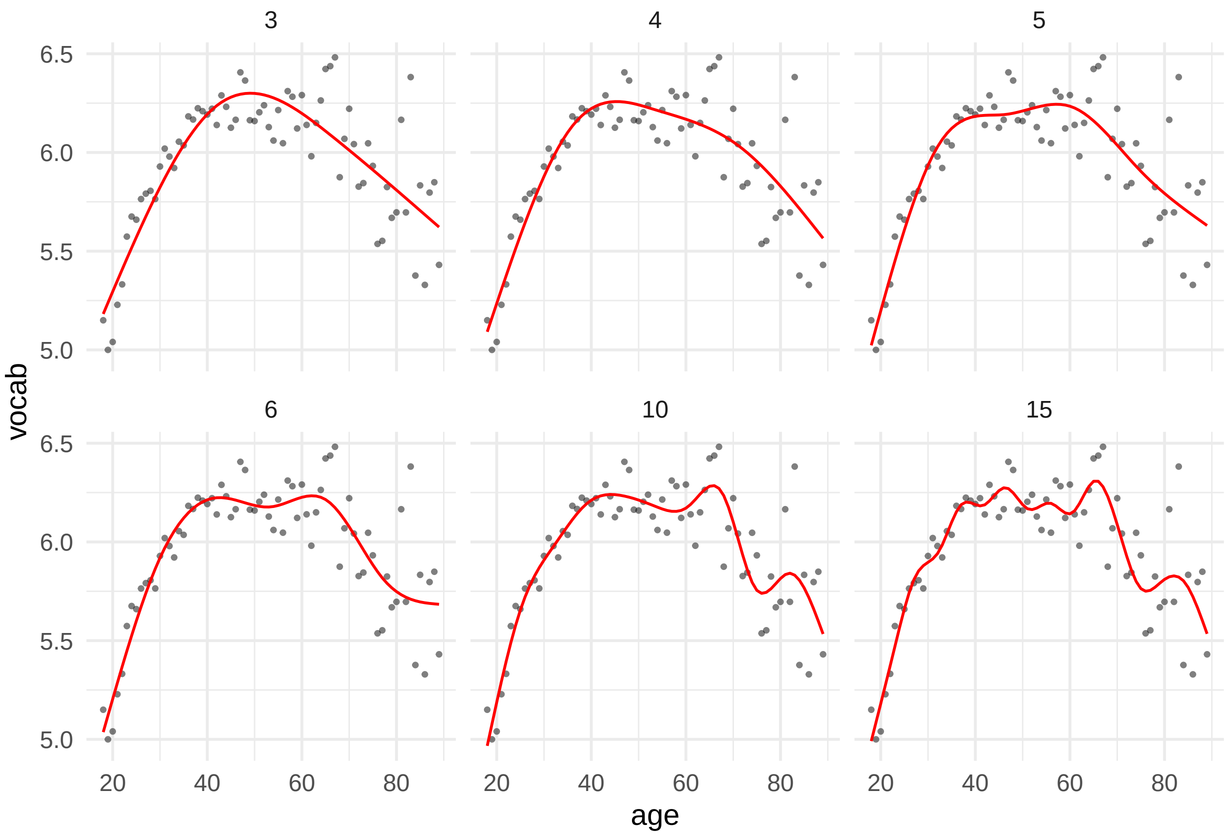

Figure 27: Cubic b-spline regression models of varying `df` values fitted to the vocabulary test data. The model with the `df` of 6 has the lowest AIC$_c$ and LOOCV score, but model 5 has a practically indistinguishable AIC$_c$ or LOOCV score.

where $D$ is some probability distribution with parameters $\psi$, and each predictor variable $f_l$ is a *smooth function* of the predictor variable's values. Usually each smooth function $f_l$ is a weighted sum of basis functions such as spline basis functions or other common types, some of which we describe below. In other words, the smooth function $f_l$ might be defined as follows:

$$f_l(x_{li}) = \beta_{l0} + \sum_{k=1}^{K} \beta_{lk}\phi_{lk}(x_{li}),$$

where $\phi_{lk}$ is a basis function of $x_{li}$. Although this definition is quite general in its scope, we can in fact be more general. For example, instead of the outcome variable being described by a probability distribution $D$ where the value of $\mu_i$ is the sum of smooth functions of the values of predictor variable at observation $i$, just as in the case of generalized linear models, we could transform $\mu_i$ by a deterministic *link function* $g$ as follows:

$$y_i \sim D(g(\mu_i), \psi), \quad \mu_i = f_1(x_{1i}) + f_2(x_{2i}) + \ldots + f_L(x_{Li}), \quad \text{for } i \in 1 \ldots n.$$

More generally still, each smooth function may in fact be a multivariate function, i.e. a function of multiple predictor variables. Thus, for example, a more general GAM than above might be as follows:

$$y_i \sim D(g(\mu_i)), \quad \mu_i = f_1(x_{1i}) + f_2(x_{2i}, x_{3i}, x_{4i}) + \ldots + f_L(x_{Li}), \quad \text{for } i \in 1 \ldots n,$$

where in this case, $f_2$ is a 3-dimensional smooth function. From these definitions so far, we can view GAMs as extensions of the general and generalized linear models that are based on sums of smooth functions of predictors. However, multilevel GAMs are also possible. Recall that an example of a simple multilevel normal linear model can be defined as follows:

$$\begin{aligned} y_{ji} \sim N(\mu_{ji}, \sigma^2), \quad \mu_j = \alpha_j + \beta_j x_{ji}, \quad \text{for } i \in 1 \ldots n \\ \text{with} \quad \alpha_j \sim N(a, \tau_\alpha^2), \quad \beta_j \sim N(b, \tau_\beta^2) \quad \text{for } j \in 1 \ldots J. \end{aligned}$$

This model can be rewritten as

$$\begin{aligned} &y_{ji} \sim N(\mu_{ji}, \sigma^2), \\ &\mu_{ji} = a + \nu_j + b x_{ji} + \xi_j x_{ji}, \quad \text{for } i \in 1 \ldots n, \quad j \in 1 \ldots J, \\ \text{with} \quad &\nu_j \sim N(0, \tau_\alpha^2), \quad \xi_j \sim N(0, \tau_\beta^2), \quad \text{for } j \in 1 \ldots J. \end{aligned}$$

A GAM version of this model might be as follows.

$$\begin{aligned} &y_{ji} \sim N(\mu_{ji}, \sigma^2), \\ &\mu_{ji} = a + \nu_j + f_1(x_{ji}) + f_{2j}(x_{ji}), \quad \text{for } i \in 1 \ldots n, \quad j \in 1 \ldots J, \\ \text{with} \quad &\nu_j \sim N(0, \tau_\alpha^2), \quad f_{2j} \sim F(\Omega), \quad \text{for } j \in 1 \ldots J. \end{aligned}$$

Here, $f_{21}, f_{22} \ldots f_{2j} \ldots f_{2J}$ are *random smooth functions*, sampled from some function space $F(\Omega)$, where $\Omega$ specifies the parameters of that function space.

## Using `mgcv`

The R package `mgcv` (Wood 2019, 2017) is a powerful and versatile toolbox for using GAMs in R. Here, we will introduce some of the main features of `mgcv`. We will use a classic data-set often used to illustrate nonlinear regression, namely the `mycle` data set, available in the `MASS` package and elsewhere. This data set gives head acceleration measurements over time in a simulation of a motorcycle crash.

We illustrate this data set in Figure 28.

The main function we will use from `mgcv` is `gam`. By default, `gam` behaves just like `lm`. In other words, to do a normal linear regression on the `mcycle` data, we could use `gam` as follows.

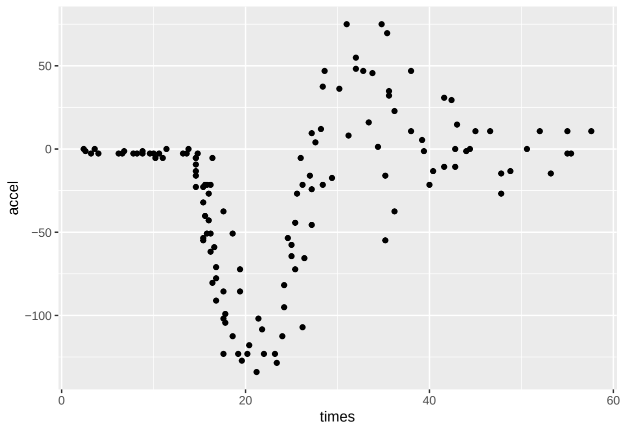

Figure 28: Head acceleration over time in a simulated motorcycle crash.

```r
library(mgcv)

M_0 <- gam(accel ~ times, data = mcycle)
```

In other to use `gam` to do basis function regression, we must apply what `mgcv` calls *smooth terms*. There are many smooth terms to choose from in `mgcv` and there are many methods to specify them. Here, we will use the function simply named `s` to set up the basis functions. The default basis functions used with `s` are *thin plate splines*. These are a type of radial basis functions, not identical but similar to those we describe above. Therefore, to do thin plae spline basis function regression using `gam`, we simply apply `s` to our predictor as follows.

```r
M_1 <- gam(accel ~ s(times), data = mcycle)
```

The plot of the fit of this model can be accomplished using the base R `plot` function. This plot in shown in Figure 29.

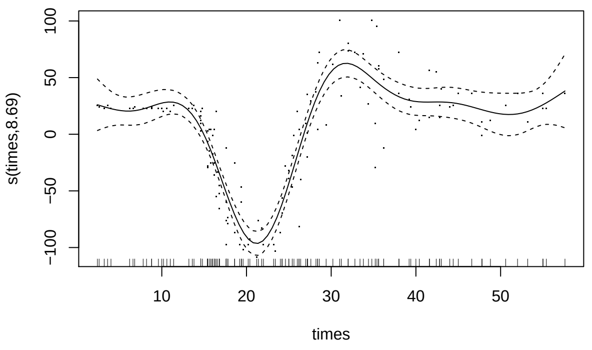

Figure 29: A thin plate spline basis function regression model applied to the `mycle` data set.

The model summary output, using the generic `summary` function, gives us some different output to what we normally get from linear or generalized linear models, even when we use basis functions. In paricular, it provides us the following table for the basis functions

```
summary(M_1)$s.table
#>                edf   Ref.df        F      p-value
#> s(times) 8.693314 8.971642 53.51503 2.957613e-71
```

The `edf` is the effective degrees of freedom of the smooth term. We can interpret it values in terms of polynomial terms. In other words, a `edf` close to one means the smooth terms is effectively a linear function, while a `edf` close to 2 or close to 3, and so on, are effectively quadratic, cubic, and so on, models. The F statistic and p-value that accompanies this value tells us whether the function is significantly different to a horizontal line, which is a linear function with a zero slope. Even if the `edf` is greater than 1, the p-value may be not significant because there is too much uncertainty in the nature of the smooth function.

The number of basis functions used by `s` is reported by the `rank` attribute of the model. In our model, we see that it is 10.

```
M_1$rank
#> [1] 10
```

In general, `mgcv` will use a number of different methods and constraints, which differ depending on the details of the model, in order to optimize the value of `k`. We can always, however, explicitly control the number of basis functions used by setting the value of `k` in the `s` function. For example, in the following, we set the value of `k` to be 5.

```
M_2 <- gam(accel ~ s(times, k = 5), data = mcycle)
M_2$rank
#> [1] 5
```

How models with different numbers of bases differ in terms of AIC can be easily determined using the `AIC` function. To illustrate this, we will fit the same model with a range of value of `k` from 3 to 30.

```
M_k_seq <- map(seq(3, 20),
               ~gam(accel ~ s(times, k = .), data = mcycle))
model_aic <- map_dbl(M_k_seq, AIC)
```

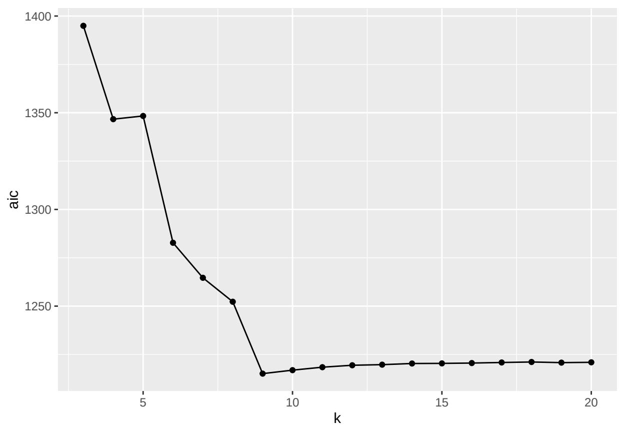

Figure 30: The AIC values of `gam` models of the `mcycle` data with different values of `k` from 3 to 20 within the exttts function.

We plot these AIC values in the following Figure 30. As we can see, the AIC values drop to a minimum at 9, and then slowly increases after this minimum over the remaining values of `k` we have examined.

In addition to explicitly setting the number of basis functions, we can also explicitly set the *smoothing penalty* with the `sp` parameter used inside the `s` function. In general, the higher the smoothing penalty, the *less* flexibility in the nonlinear function. For example, very high values of the smoothing penalty effectively force the model to be a liner model. On the other hand, low values of the smoothing penalty may be overly flexible and overfit the data, as we saw above. In Figure 31, we display the model fits of `gam` models applied to `mcycle` data for three different values of *sp*.

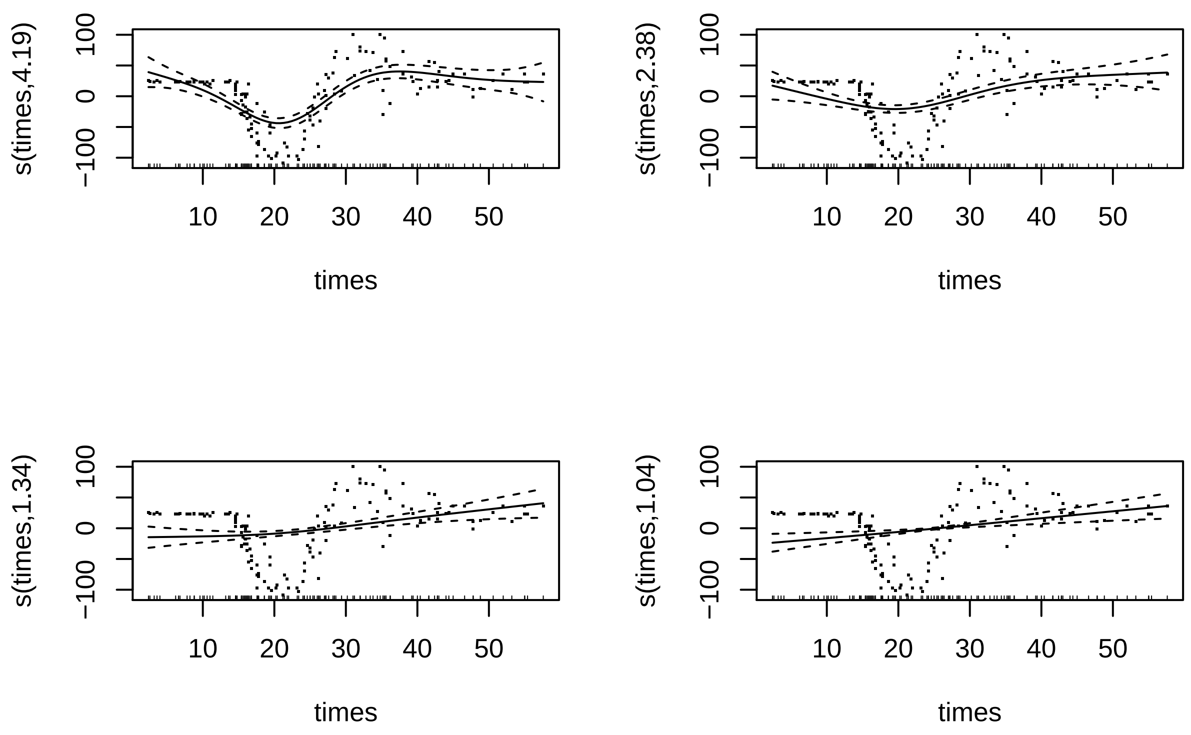

Figure 31: Plots of the fits of GAM models to the `mcycle` data with different values of `sp` from, left to right and upper to lower, $10^{-1}$, 1, 10, 100.

```
#> null device
#>           1
```

From the Figure, it is clear that values of `sp` greater than 1 tend to underfit the data. Therefore, it seems likely that much lower values of `sp` will be optimal in terms of AIC. In the following, we evaluate the AIC of a set of models whose `sp` ranges from $10^{-5}$ to 1 in powers of 10.

```
sp_seq <- 10^seq(-5, 0)
M_sp_seq <- map(sp_seq,
                ~gam(accel ~ s(times, sp = .), data = mcycle)
)
model_sp_aic <- map_dbl(M_sp_seq, AIC) %>%
  set_names(sp_seq)
```

We may then subtract the minimum value of AIC to see the difference in AIC from each model to the optimal model.

```
model_sp_aic - min(model_sp_aic)
#>       1e-05       1e-04       0.001        0.01         0.1           1
#>   0.1529868   0.0656411   0.0000000  18.3651318  89.8999526 146.1496912
```

As we can see, the optimal model is at 0.001, with the AIC values for the lower orders of magnitude being very close, but a rapidly increasing AIC value for higher orders of magnitude.

As with `k`, if `sp` is not explicitly set, `mgcv` uses a different methods, including cross-validation, to optimize the value of `sp` for any given model. For example, for models `M_1` and `M_2` above, we can see that their `sp` values are as follows.

```
c(M_1$sp, M_2$sp)
#>     s(times)     s(times)
#> 0.0006195886 0.0213462068
```

These optimized `sp` values are close to the `sp` value we estimated using `AIC` above.

## References

Berry, Donald A. 1995. *Statistics: A Bayesian Perspective.* Duxbury Press.

Bishop, Christopher M. 1995. *Neural Networks for Pattern Recognition.* Oxford University Press.

Burnham, Kenneth P, and David R Anderson. 2003. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach.* Springer Science & Business Media.

Gelman, Andrew, and Deborah Nolan. 2002. *Teaching Statistics: A Bag of Tricks.* Oxford University Press.

Wood, Simon. 2019. *Mgcv: Mixed Gam Computation Vehicle with Automatic Smoothness Estimation.* https://CRAN.R-project.org/package=mgcv.

Wood, S. N. 2017. *Generalized Additive Models: An Introduction with R.* 2nd ed. Chapman; Hall/CRC.