Lecture 18- Splines

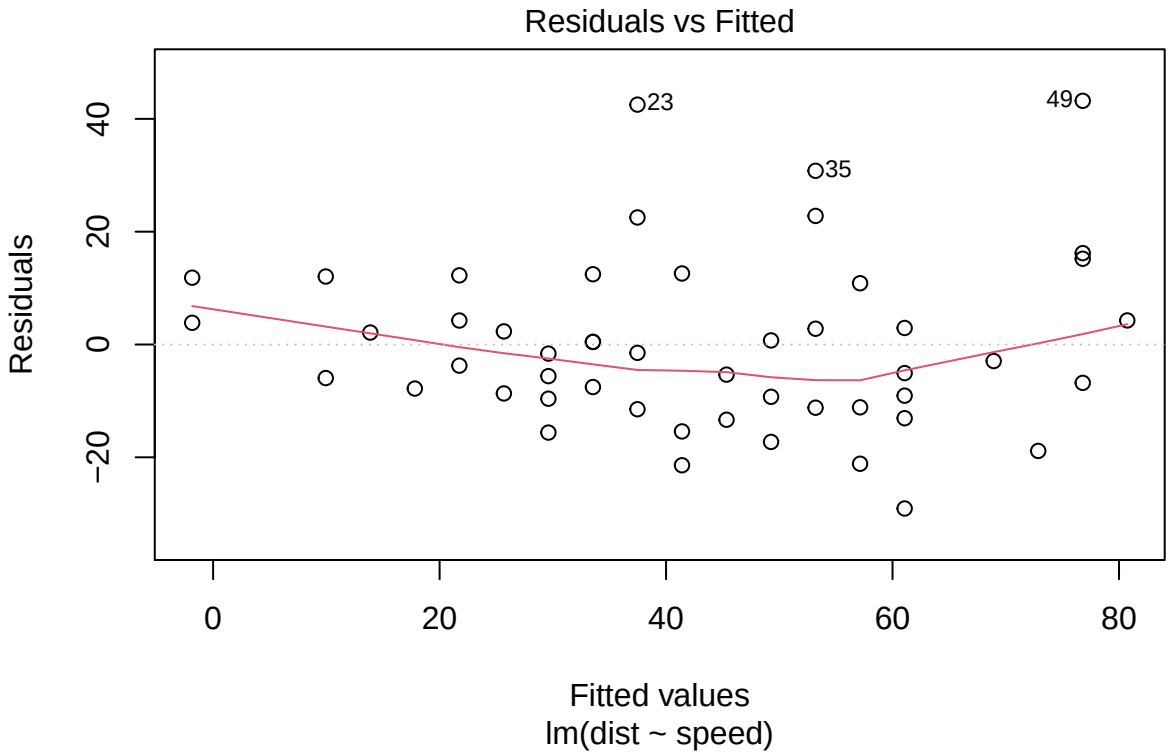
Peter Shaffery

3/13/2021

Introduction - What is Smoothing?

Today we'll talk about something that we've actually already seen quite a lot of: smoothing.

```
library(tidyverse)
library(magrittr)

linear.mod = lm(dist~speed, data=cars)
plot(linear.mod,which=1) # the red line is a "smoothed" estimate of the residuals
```



A *smoother* is (roughly) any curve fit to datapoints (x_i, y_i) trend which is "smoother" than the y_i, in the sense of having less variance.

The term smoother typically also carries a connotation of being **non-parametric**. Non-parametric models are weirdly named, because they absolutely DO have "parameters". What sets them apart from the linear models that we have studied so far, is that they can have (potentially) **infinite** numbers of such parameters. The saturated models from the definition of deviance, which assign a parameter for each data point, are examples of non-parametric models.

The basic theoretical form of these non-parametric smoothers is already familiar to us from GLMs:

$$E[y_i|x_i] = f(x_i) + \epsilon_i$$

Where we allow f to be **any** function, which will attempt to estimate using a flexible non-parametric framework. Notice that this is really just a type of GLM, where the link function g has been re-written in terms of its inverse, $g^{-1} = f$.

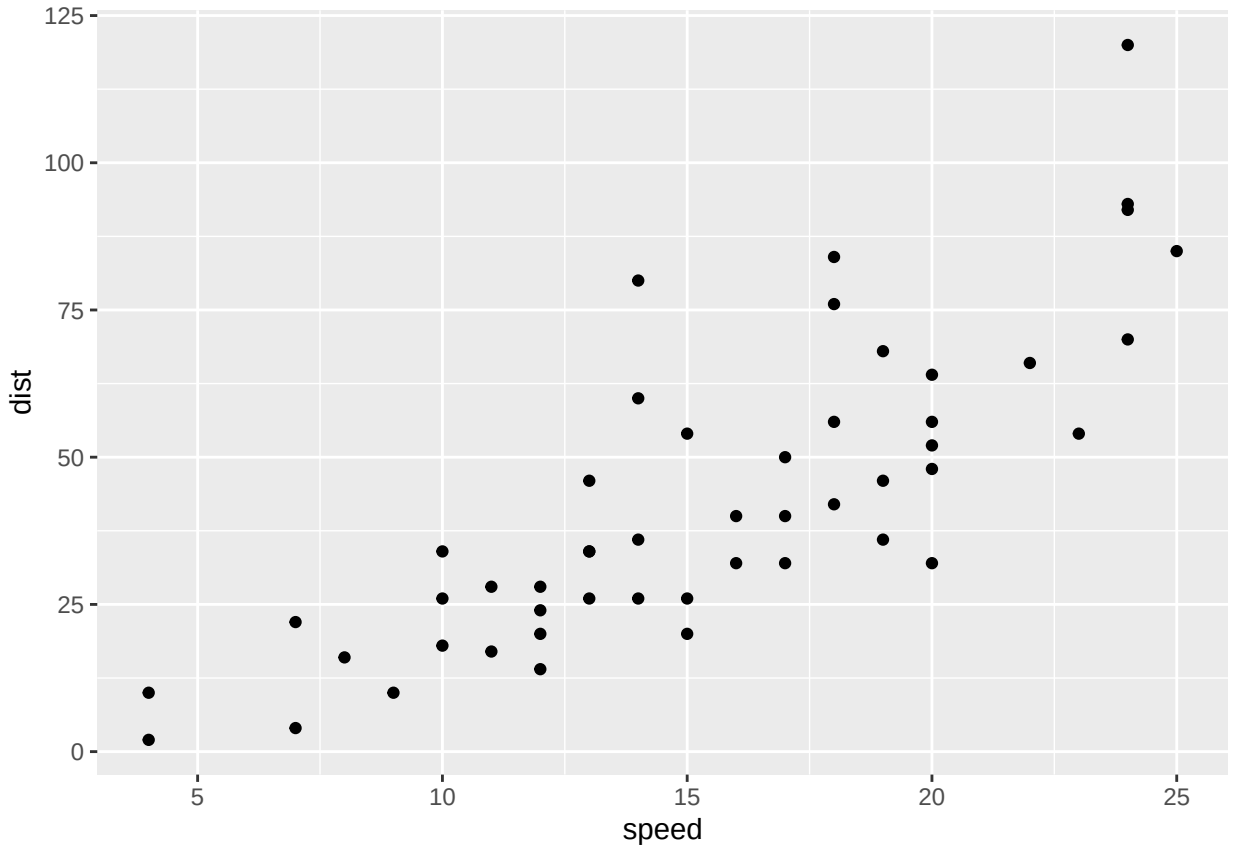
Smoother models are differentiated from each other by the choice of estimate f. This lecture is really just a primer to familiarize you with existence of this whole model type, and is in no way exhaustive. For some more theoretical details check out Hastie and Tibshirani (1990).

Example: `cars`

```
cars %>% head # speed and stopping distance of 50 cars
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

```
ggplot(cars, aes(x=speed,y=dist))+geom_point()
```

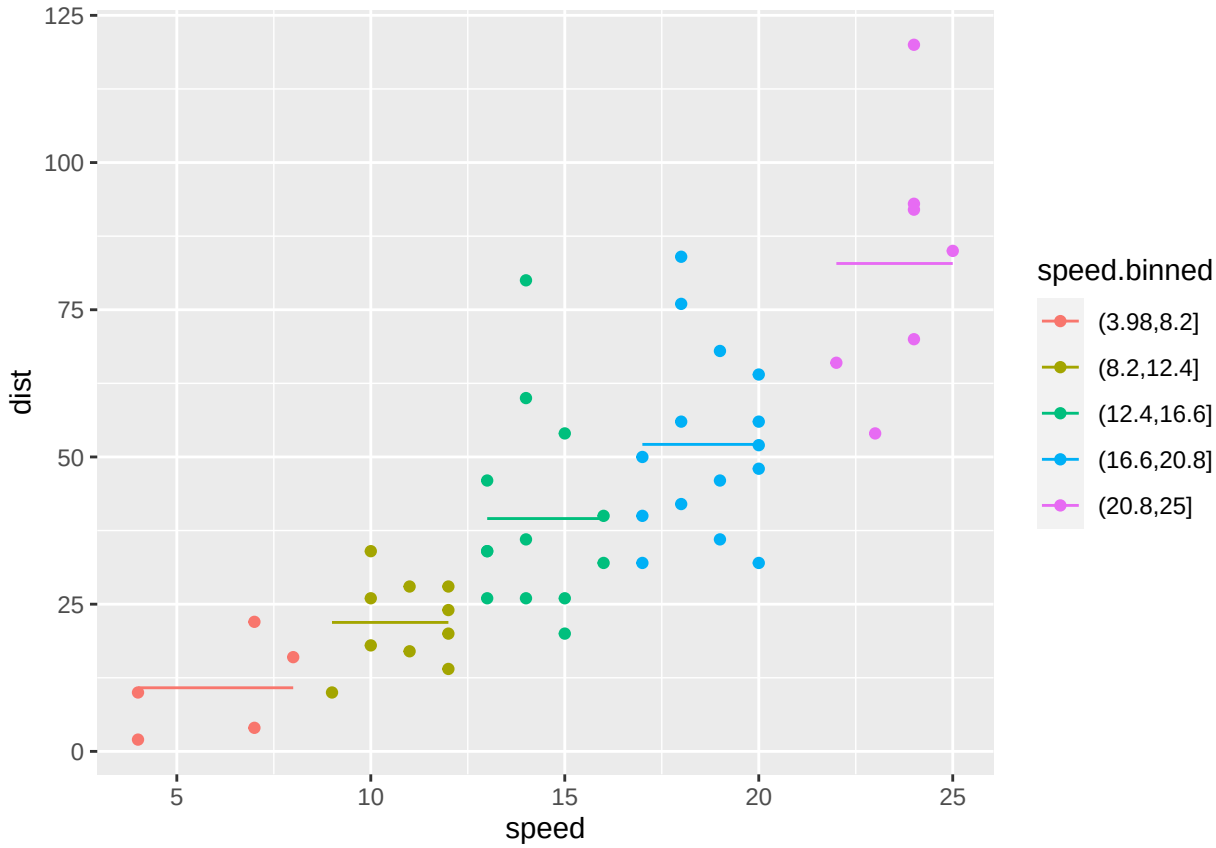


Types of Smoothers

Binned Mean

This one is sometimes presented alongside other smoothing models, but is in no way actually a smooth estimate of f. The idea is to chunk up the range of the x_i into some bins of constant width, and then just take the mean of the data points in each bin. The estimate of f is the piecewise constant function defined by each bin's mean.

```
plot.df = cars

plot.df$speed.binned = cut(plot.df$speed,5)
plot.df %<>% group_by(speed.binned) %>% mutate(pred = mean(dist))
ggplot(plot.df, aes(x=speed,color=speed.binned))+geom_point(aes(y=dist)) + geom_step(aes(y=pred))
```



Although this type of smoother is unappealing for most "linear regression" contexts, it's evolved form the *classification and regression tree* (CART) is actually a very popular model type for *classification problems*. This type of model gets a lot smarter about bin construction, but from 10,000 feet is fundamentally the same thing as what we did above.

Rolling Mean

The binned mean was kind of gnarly, it wasn't at all a "smooth" function (in the sense of being continuous and having continuous derivatives), and it's barely statistical at all.

The next rung up the sophistication ladder is a *rolling mean* model. The idea here is very similar, we're still grouping and averaging the data, but instead of slicing up x_i into non-overlapping bins we allow the bins to overlap.

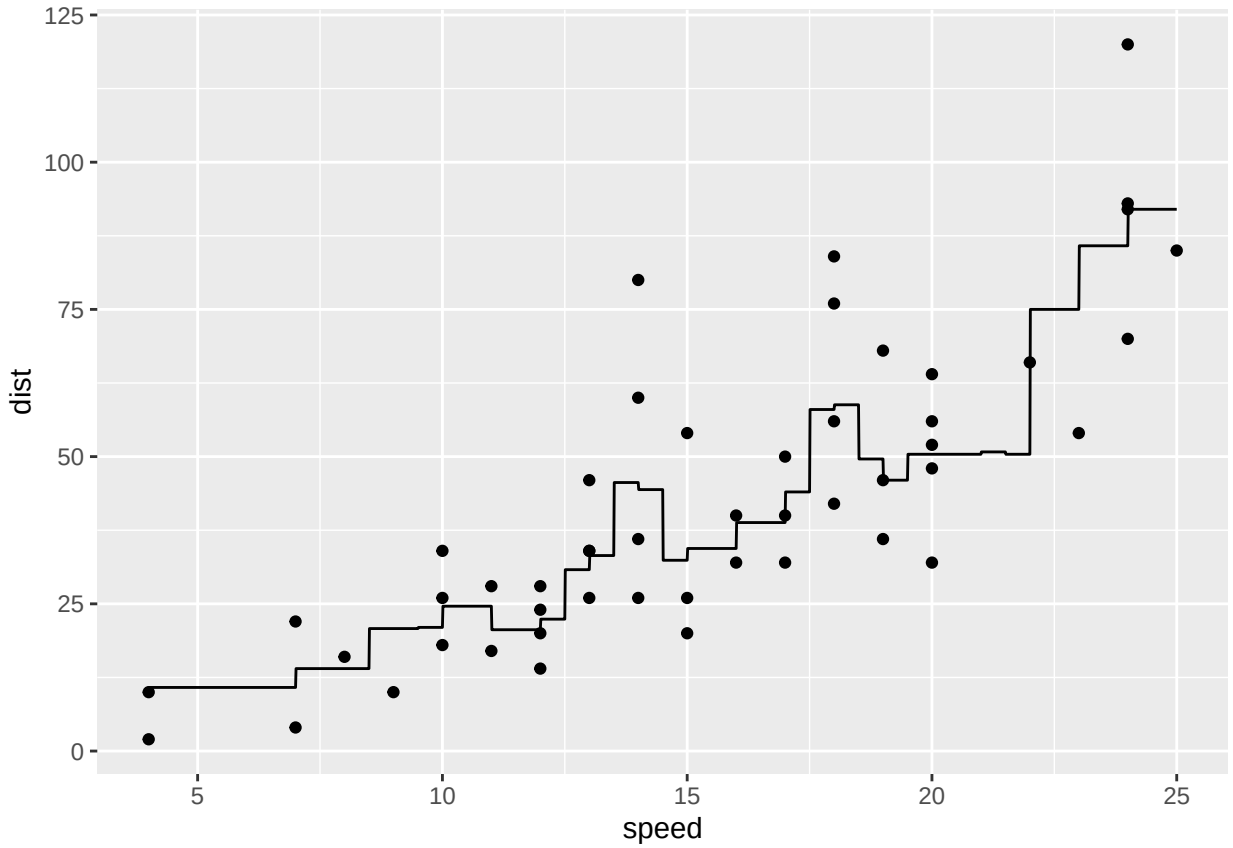
Say that we want to estimate the value of $f(x^*)$, which we'll denote as $\hat{f}$. The idea here is to take k of the

closest x_i to x^*, which we'll say have the index values $j, ..., j+k$. Our estimate is then:

$$\hat{f} = \frac{1}{k} \sum_{i=j}^{j+k} y_i$$

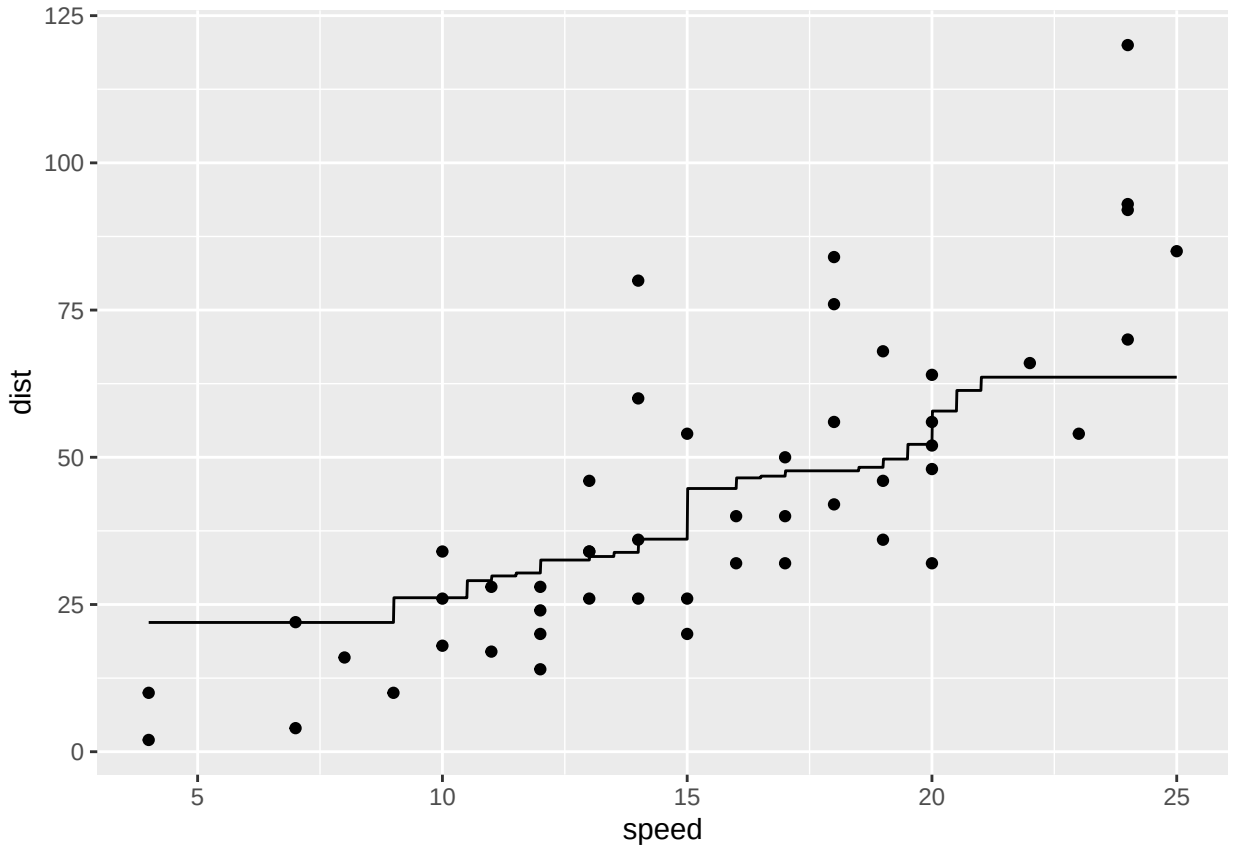
By allowing the averaging window to move with the point x^*, this estimate will be a lot smoother than our piecewise model:

```
rolling.avg = function(x0,k,x,y){
  dist = abs(x0-x)
  pts = y[order(dist)][1:k]
  return(mean(pts))
}

k=5
xgrid = seq(min(cars$speed),max(cars$speed),.01)
pred = sapply(xgrid, function(x){ rolling.avg(x,k,cars$speed,cars$dist) })
ra.df = data.frame(xgrid=xgrid, pred=pred)
ggplot(plot.df) +
  geom_point(aes(x=speed,y=dist)) +
  geom_line(data=ra.df, aes(x=xgrid,y=pred))
```

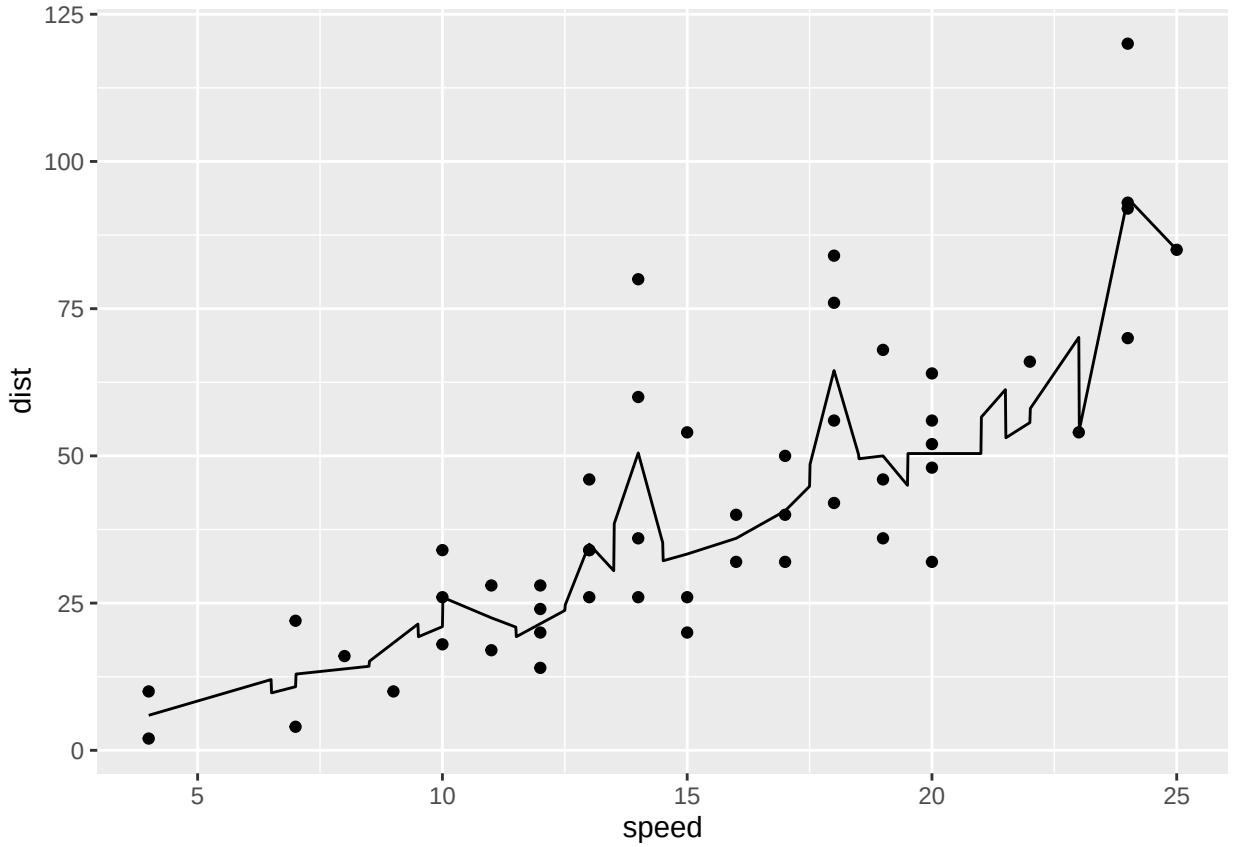



```
k=20
xgrid = seq(min(cars$speed),max(cars$speed),.01)
pred = sapply(xgrid, function(x){ rolling.avg(x,k,cars$speed,cars$dist) })
ra.df = data.frame(xgrid=xgrid, pred=pred)
ggplot(plot.df) +
```

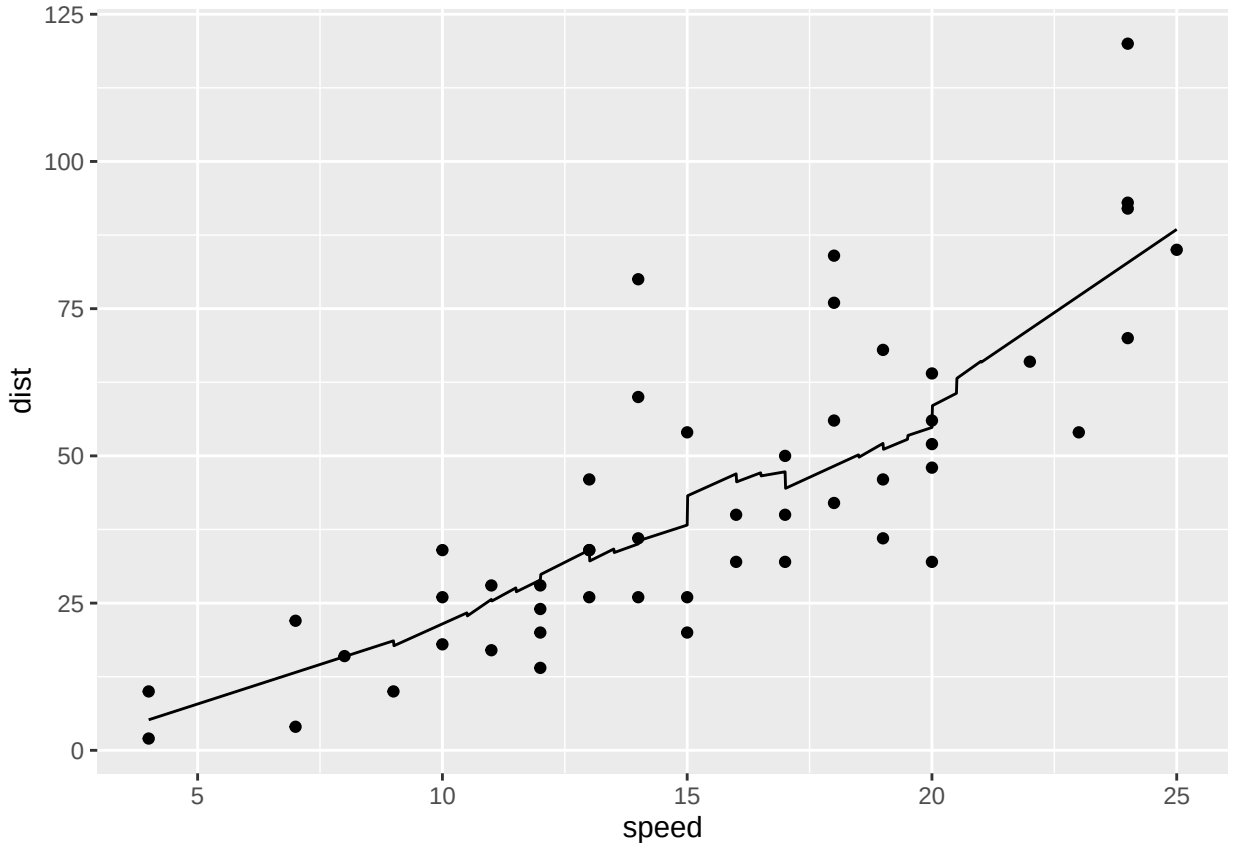
```
  geom_point(aes(x=speed,y=dist)) +
  geom_line(data=ra.df, aes(x=xgrid,y=pred))
```




```
rolling.regression = function(x0,k,x,y){
  dist = abs(x0-x)
  y.vals = y[order(dist)][1:k]
  x.vals = x[order(dist)][1:k]

  mod = lm(y.vals~x.vals)
  ret = predict(mod,data.frame(x.vals=c(x0)))

  return(as.numeric(ret))
}

k=5
xgrid = seq(min(cars$speed),max(cars$speed),.01)
pred = sapply(xgrid, function(x){ rolling.regression(x,k,cars$speed,cars$dist) })
ra.df = data.frame(xgrid=xgrid, pred=pred)
ggplot(plot.df) +
  geom_point(aes(x=speed,y=dist)) +
  geom_line(data=ra.df, aes(x=xgrid,y=pred))
```




```
k=20
xgrid = seq(min(cars$speed),max(cars$speed),.01)
pred = sapply(xgrid, function(x){ rolling.regression(x,k,cars$speed,cars$dist) })
ra.df = data.frame(xgrid=xgrid, pred=pred)
ggplot(plot.df) +
  geom_point(aes(x=speed,y=dist)) +
  geom_line(data=ra.df, aes(x=xgrid,y=pred))
```



LOESS

So far our models have been fairly crude. This had the advantage of being cheap to compute, just group the data and take an average, but the predictions were kind of "ugly".

A more complex scheme, which does not have these problems is called local regression, sometimes abbreviated as LOESS (Locally Estimated Scatterplot Smoothing). Again, the idea is similar to the previous type of smoother, but with a little extra wrinkle of complexity.

Previously we just binned k of the nearest points and took their mean. We could have gone one step further even and fit a little regression to the k points. What's annoying about this is that it's challenging to choose a sensible k in some cases.

Consider the above example with $k = 20$. In the middle of the plot it's okay, it produces a reasonable trend, but near the edges it seems to be missing some of the behavior.

A smarter model would incorporate information from the entire plot in some way, properly accounting for how "relevant" far-away datapoints are. This is the idea of local regression.

Say that we are trying to estimate the value $f(x^*)$, denoted $\hat{f}$. Local regression does so by fitting a regression to the datapoints, **weighted** by their distance from x^*. That is, for a given value of x^* it estimates β_0^* and β_1^* by:

$$\beta_0^*, \beta_1^* = \text{argmin} \sum_{i=1}^{n} w(|x_i - x^*|)(\beta_0 + \beta_1 x_i - y_i)^2$$

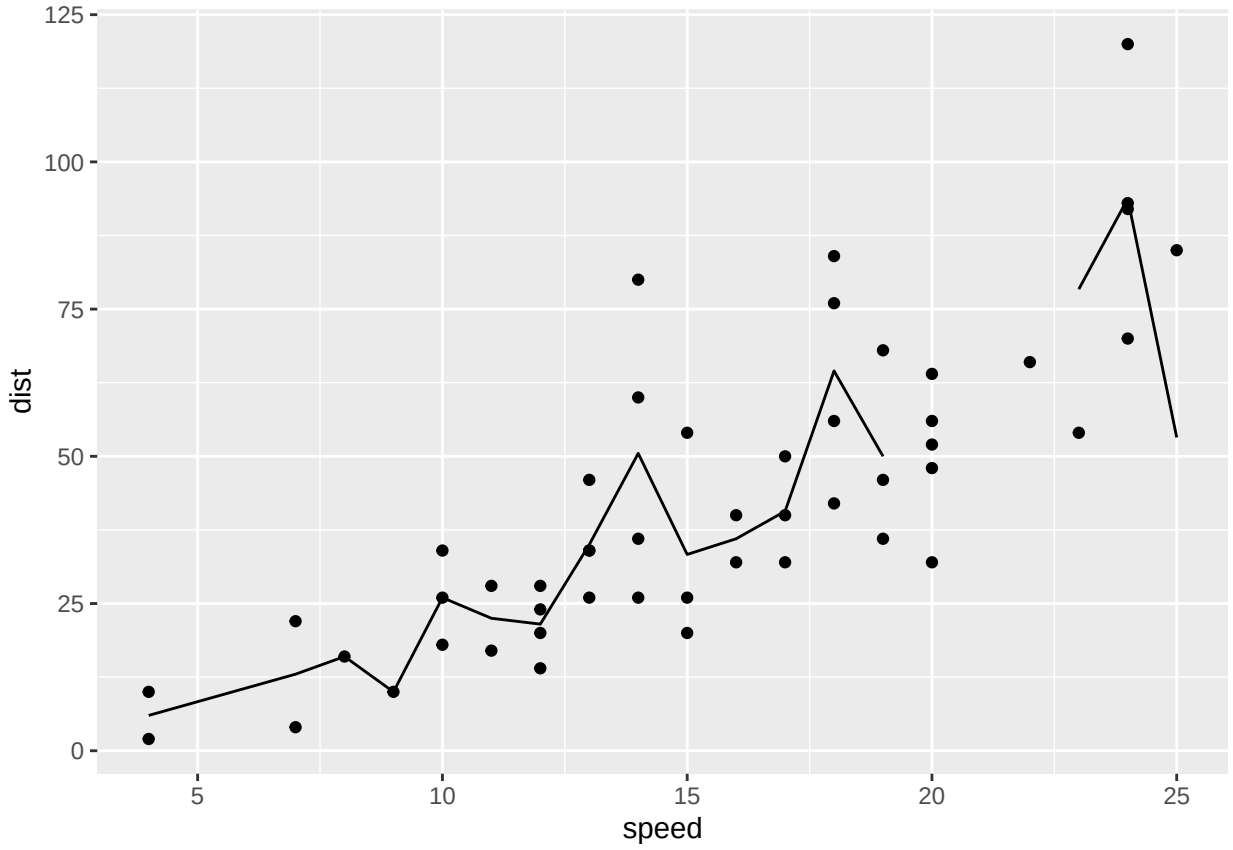
Where the function $w(d)$ is the *weighting function* that depends on the distance $d = |x_i - x^*|$. Typically $w(d)$ is chosen to prioritize points close to x^* and ignore points further away.

Now, to control the computational cost of this algorithm typically not every point x_i is included in the above sum. Typically only a percentage α of the available data is included. This is usually equivalent to setting $w(d) = 0$ when d is outside some radius, $d > R$.

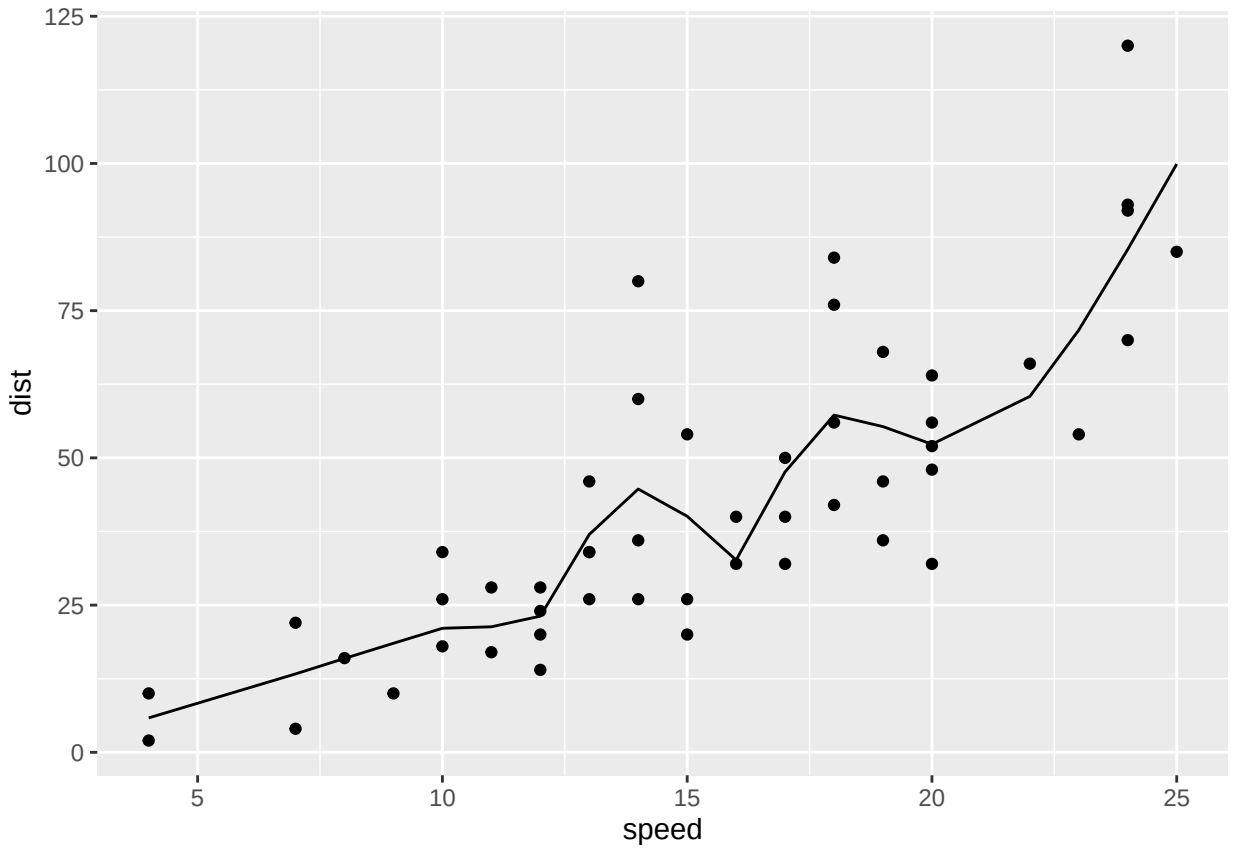
There are lots of flavors of local regression, which tweak (among other things):

1. The weighting function (most common is tri-cube, $w(d) = (1 - \left[\frac{d}{d_{\max}}\right]^3)^3$)
2. The distribution of the $n\alpha$ fitting points (do you just want close points, or do you want some near the edges?)
3. The degree of the regression curve (here we're just looking at a linear fit, but a quadratic polynomial is also a popular choice)

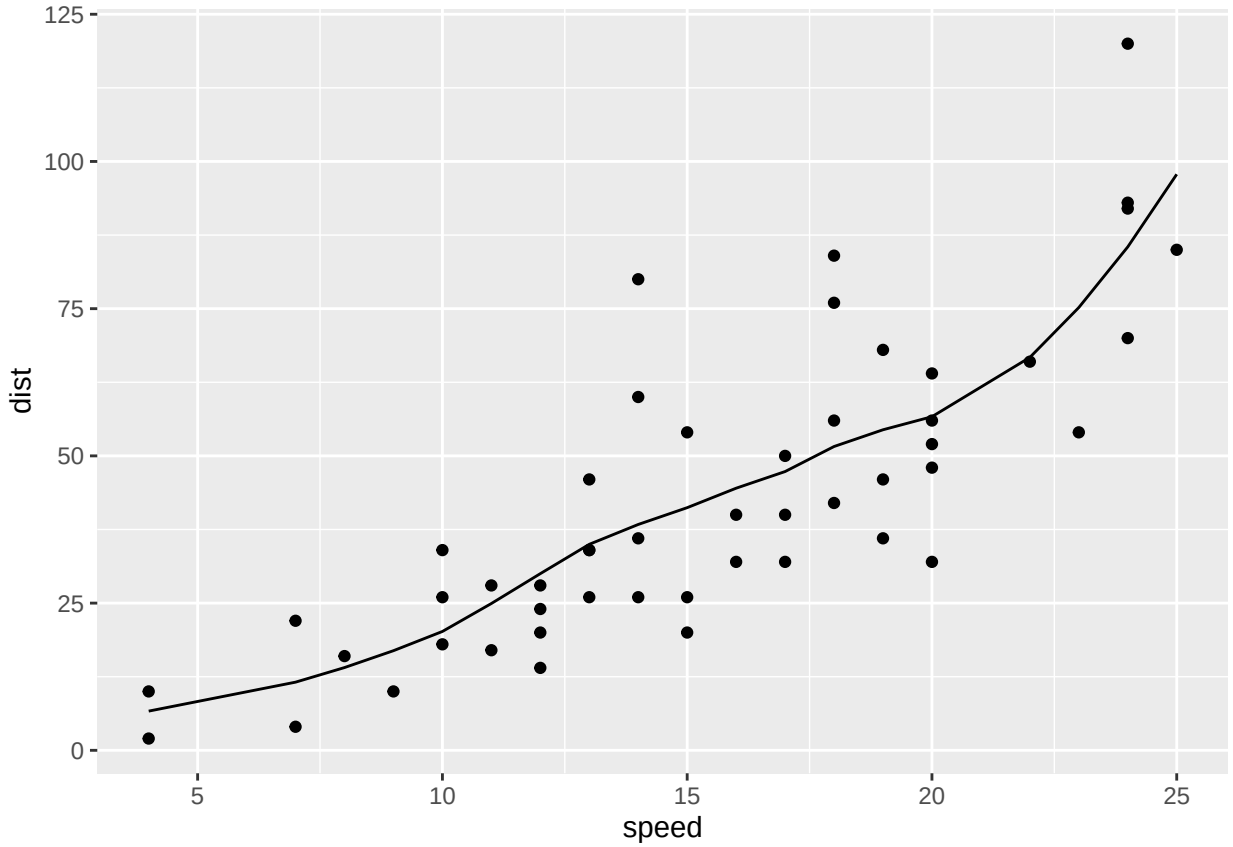
```
plot.df = cars
alpha = .1 # k = 5
mod = loess(dist~speed, data=cars, span=alpha)
plot.df$pred = predict(mod)
ggplot(plot.df, aes(x=speed)) +
  geom_point(aes(y=dist)) +
  geom_line(aes(y=pred))
```




```
alpha = .4 # k = 20
mod = loess(dist~speed, data=cars, span=alpha)
plot.df$pred = predict(mod)
ggplot(plot.df, aes(x=speed)) +
  geom_point(aes(y=dist)) +
  geom_line(aes(y=pred))
```




```
alpha = .6 # k = 20
mod = loess(dist~speed, data=cars, span=alpha)
plot.df$pred = predict(mod)
ggplot(plot.df, aes(x=speed)) +
  geom_point(aes(y=dist)) +
  geom_line(aes(y=pred))
```



Terminology Weirdness: Sometimes people will distinguish between LOWESS (weighted local regression) and LOESS. I have no idea why. As far as I can tell these are basically equivalent models, and any difference is lost in the wash.

Interlude: Wiggliness and Bias-Variance Tradeoff

An important, and highly technical, concept in smoothing is *wiggliness*, how "smooth" the smoother actually is.

Typically smoothers have a single *hyperparameter* which control how wiggly the final product is. In the binned mean and rolling mean models, this was k. In the LOESS model, it's α. Observe that as we increase this parameter, in both cases, the smoother becomes smoother. If we set $k = n$ or $\alpha = 1$, then it's just a straight line.

Wiggliness has an important connection to a concept which we talked about briefly back in lecture 3: *the bias-variance tradeoff*.

- **Bias:** The expected difference between an estimator and the estimand
- **Variance:** The variance of an estimator

Recall that the *overall mean square error* (MSE) can be written:

$$MSE = \text{Bias}^2 + \text{Variance}$$

So our goal in selecting a wiggliness parameter value is to strike a balance between bias and variance to achieve minimum overall error.

Let's see a quick example of this with the rolling mean smoother. Recall that we have an estimate:

$$\hat{f} = \frac{1}{k}\sum_{i=j}^{j+k} y_i$$

Which we would like to use to estimate $f(x^*)$. For simplicity, let's say that x^* is the middle point x_j, that is that $x^* = x_{j+k/2}$.

Thus:

$$\begin{aligned} E[f(x_{j+k/2}) - \hat{f}] &= f(x_{j+k/2}) - \frac{1}{k}\sum_{i=j}^{j+k} E[y_i] \\ &= f(x_{j+k/2}) - \frac{1}{k}\sum_{i=j}^{j+k} f(x_i) \\ &= \frac{k-1}{k} f(x_{j+k/2}) - \frac{1}{k}\sum_{i \neq j+k/2} f(x_i) \end{aligned}$$

It's not hard to convince yourself that this will get bigger (in absolute value) as k grows large, or that the minimum possible bias is achieved with $k = 1$.

On the other hand, let's look at the variance:

$$\begin{aligned} \mathrm{Var}[\hat{f}] &= \mathrm{Var}[\frac{1}{k}\sum_{i=j}^{j+k} y_i] \\ &= \frac{1}{k^2}\sum_{i=j}^{j+k} \mathrm{Var}[y_i] \\ &= \frac{1}{k^2}\sum_{i=j}^{j+k} \sigma^2 \\ &= \frac{\sigma^2}{k} \end{aligned}$$

Hence the variance *decreases* as k grows large. And this makes sense, with $k = n$ then $\hat{f}$ is just the overal sample mean of the y_i, whose variance goes to 0 as n goes to infinity.

Hence *decreasing* the wiggliness (by increasing k) *increases* the bias and *decreases* the variance.

Cubic Splines

So far all of our smoother methods have been somewhat "ad hoc". We've just been thinking about what kind of smoothing schemes seem sensible and easy. Splines, on the other hand, have a more theoretical foundation.

The basic idea for a cubic regression spline is that we'd like to pick a smoothing function which:

a. Fits the data well
b. Isn't too wiggly

Regression splines make this explicit, by choosing a smoothing function $\hat{f}$ which minimizes the following loss function:

$$\sum_i (y_i - f(x_i))^2 + \lambda \int f''(t)^2 dt$$

Notice that this loss function meets both criteria. It penalizes smoothers which don't fit the data (through the $\sum_i (y_i - f(x_i))^2$ term), as well as smoother which are too wiggly (through the $\lambda \int f''(t)^2 dt$ term). The

parameter λ in this case takes on the role that k and α took on in our previous models- namely it controls *how intensely* we penalize wiggliness. Obviously $\lambda = 0$ has no wiggliness penalty, so the resulting $\hat{f}$ will just satisfy $\hat{f}(x_i) = y_i$ (or $\hat{f}(x_i) = \bar{y}_i$ where there are multiple y_i per x_i). On the other hand, if we make λ absurdly large then $\hat{f}$ will just be (roughly) the usual least-squares regression line (since if $\hat{f}'' = 0$ then $\hat{f}$ is just a line).

For some freakish reason, it turns out that (for any value of λ) the function $\hat{f}$ which minimizes this loss function is a *piecewise, third-order (cubic) polynomial*, which is known as a **cubic spline**. This has the functional form:

$$f(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \sum_{i=2}^{n-1} \theta_i (x - x_i)_+^3$$

Here the notation $(z)_+$ is meant to imply a function, defined as:

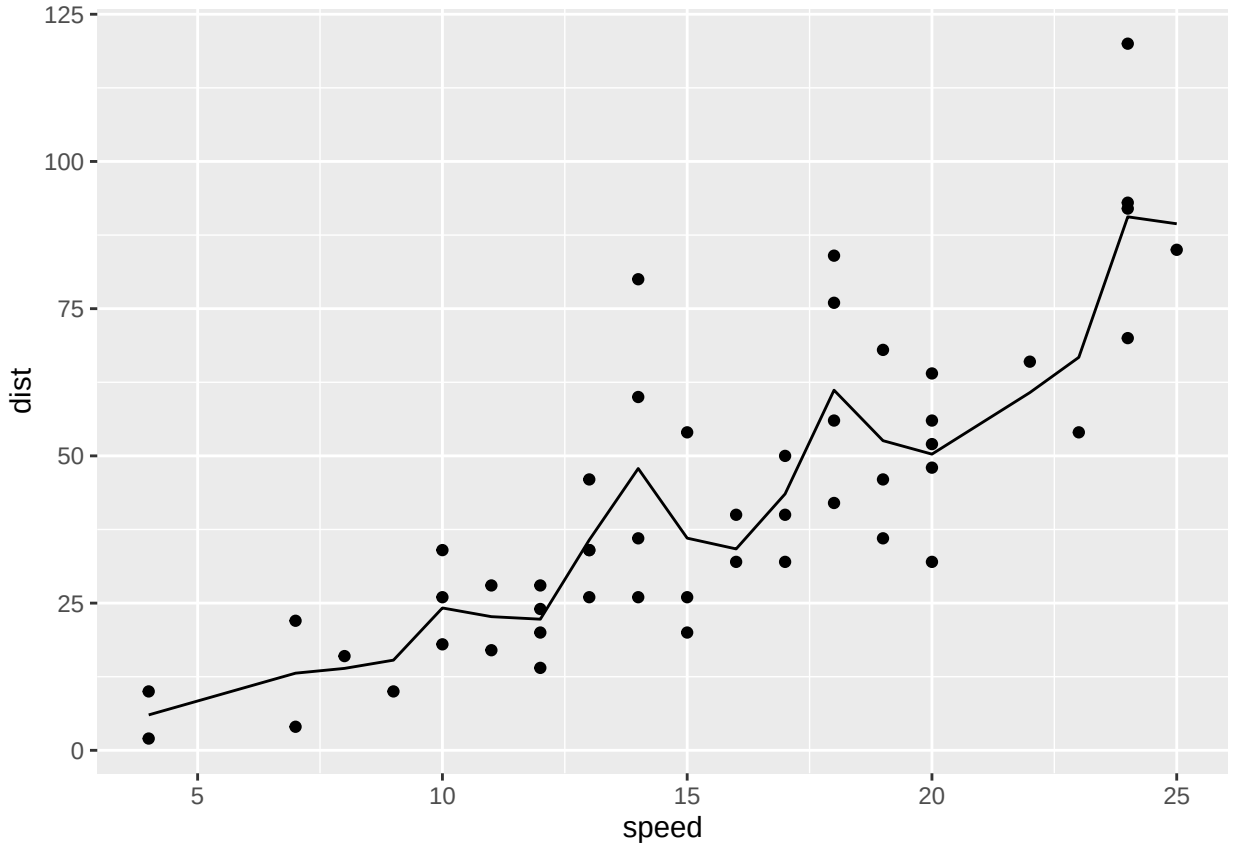
$$(z)_+ = \begin{cases} 0, z < 0 \\ z, z \geq 0 \end{cases}$$

Now, it turns out that fitting the above cubic function can cause some numerical errors, so it's common to rewrite a cubic spline in terms of *basis functions* B_i:

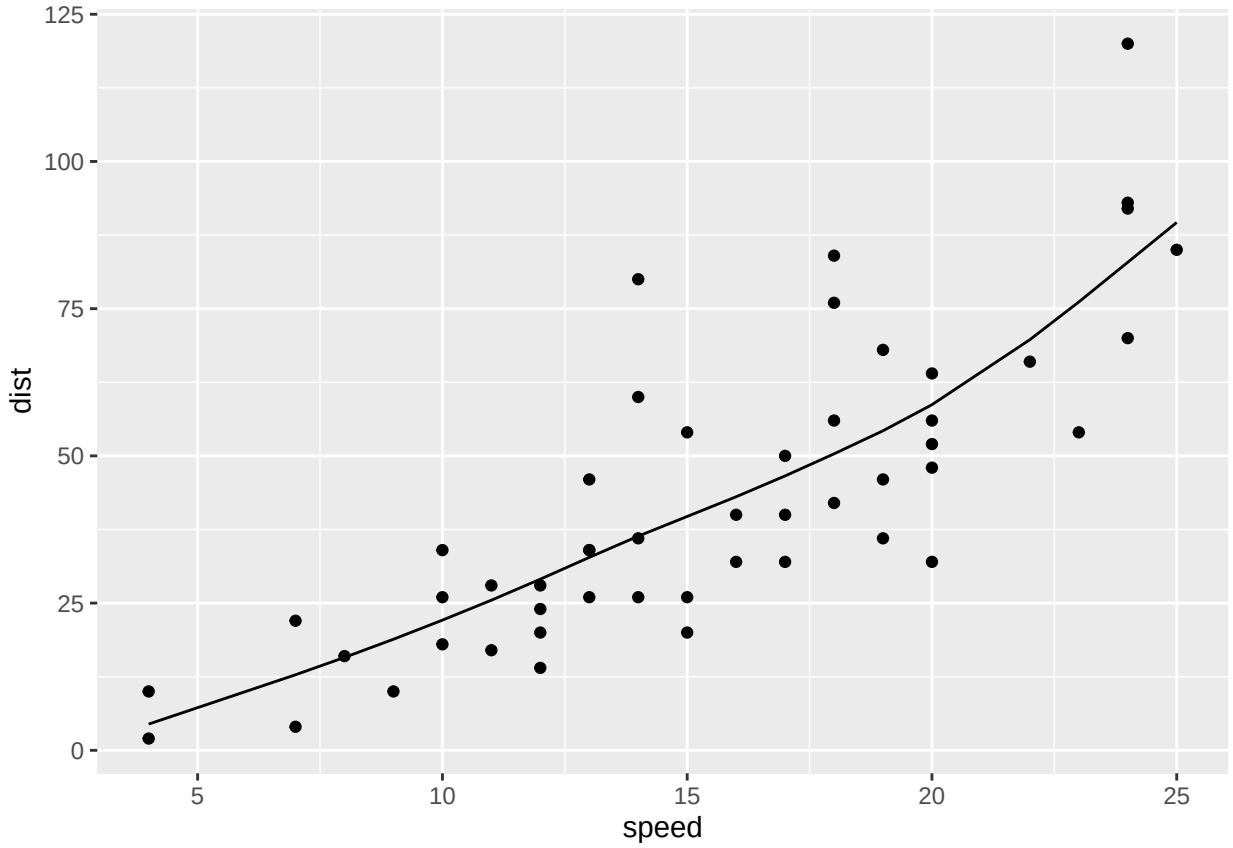
$$f(x) = \sum_{i=2}^{n-1} \gamma_i B_i(x)$$

The actual definition of these basis functions is a little complicated, but the basic idea is that they are constructed such that, when added all together, you get the same thing as the cubic version of the spline. Regression splines defined in this way are called B-splines (B for "basis").

```
lambda = .00001
mod = smooth.spline(cars$speed, cars$dist, lambda=lambda)
plot.df = data.frame(speed=mod$x, pred=mod$y)
ggplot(plot.df, aes(x=speed)) +
  geom_point(data=cars, aes(y=dist)) +
  geom_line(aes(y=pred))
```




```
lambda = .01
mod = smooth.spline(cars$speed, cars$dist, lambda=lambda)
plot.df = data.frame(speed=mod$x, pred=mod$y)
ggplot(plot.df, aes(x=speed)) +
  geom_point(data=cars, aes(y=dist)) +
  geom_line(aes(y=pred))
```



Cross Validation

So how do we choose how wiggly to make our smoother?

Well, we saw above that the best value of wiggliness balances bias and variance, minimizing out-of-sample MSE. Our best option would therefore be to pick the option which predicts new data well.

Unfortunately we don't have any new data, since if we did it would just be old data. So we'll do the next best thing: cross-validation.

Cross validation (CV) is one of the most powerful tools in the stats, machine learning, and data science toolbox. At it's core, the idea is very simple. Split your data into two chunks: a "training set" and a "testing set". We fit our model to the training set, and then measure its error on the testing set. The idea here is that, assuming your data is truly random and independent, the testing set does a good job representing the future data.

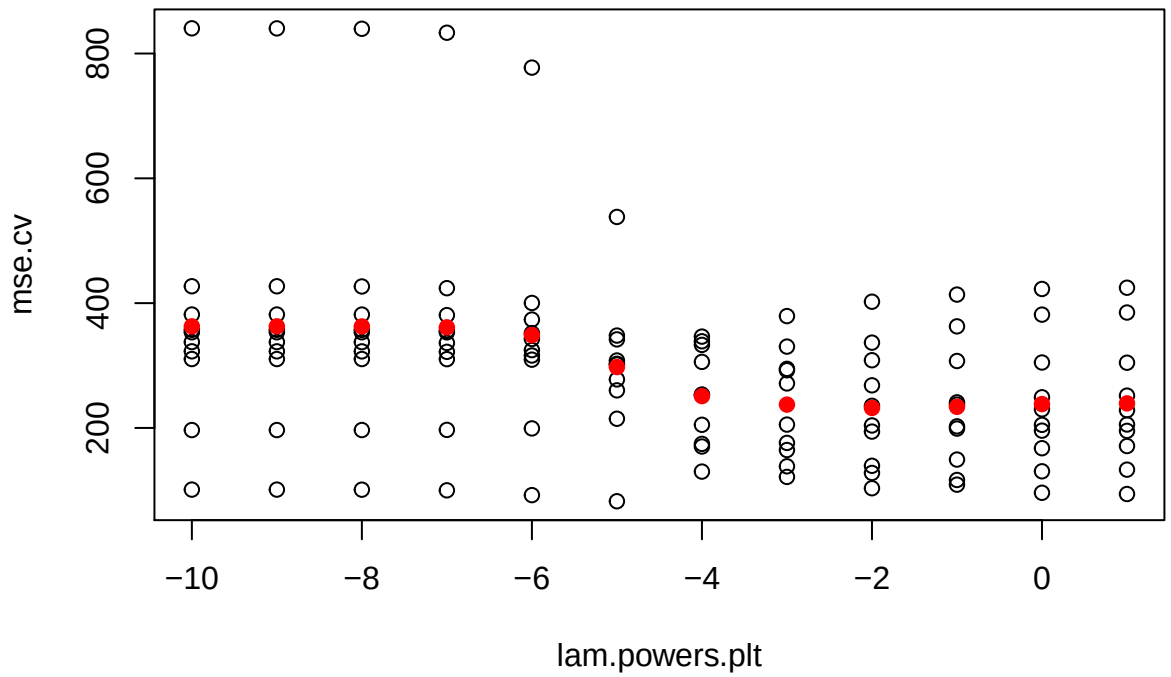
By choosing a wiggliness parameter which minimizes the CV error, we can effectively balance the bias and variance of our model without having to compute either explicitly.

Often CV is done in *folds*. The idea here is that, while one train/test split might be good, it could (by random chance) also be unrepresentative of future data. To hedge against this we can split our data multiple times, into k (usually $k = 10$) distinct chunks (*folds*). We then go down the list of chunks, designating each one as the training data set and the rest as testing data. For each fold, we compute a separate CV error.

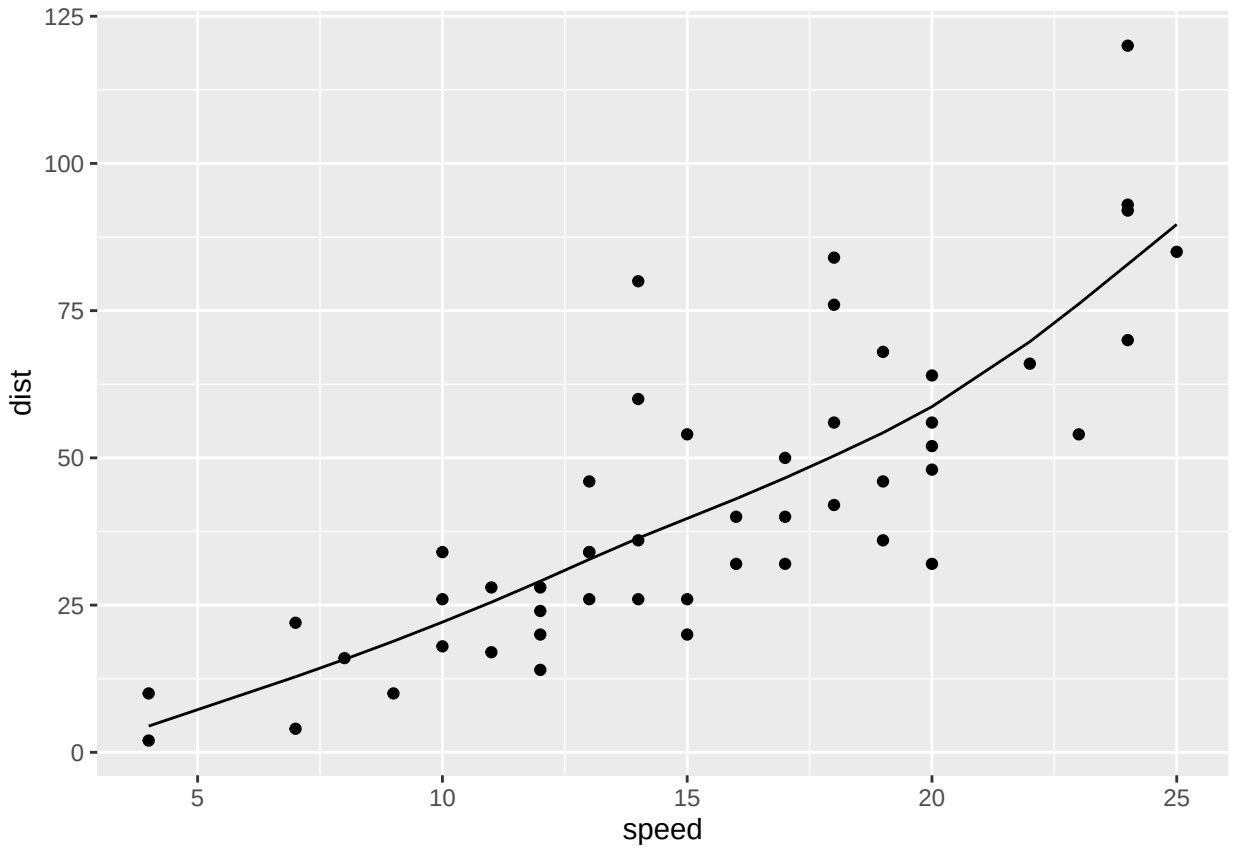
Let's 10-fold cross validation to choose the value of λ for our cubic spline:

```
set.seed(777)

cubic.spline.mse = function(lam, dat, testing.inds){
  train = dat[-testing.inds,]
```

```
  test = dat[testing.inds,]

  mod = smooth.spline(train[,1], train[,2], lambda=lam)
  preds = predict(mod, test[,1])$y
  res = preds - test[,2]

  return( mean(res^2) )
}

n.folds = 10
folds = caret::createFolds(cars$speed,k=n.folds)
lam.powers = seq(-10,1)
lam.grid = 10^(lam.powers)
mse.cv = c()

for (l in lam.grid){
  for (fold in folds) {
    mse.cv = c(mse.cv, cubic.spline.mse(l, cars, fold))
  }
}

lam.powers.plt = rep(lam.powers, each=n.folds)
mean.cv.mse = apply(matrix(mse.cv,nrow=n.folds),2,mean)
plot(lam.powers.plt,mse.cv)
points(lam.powers, mean.cv.mse,col='red', pch=19)
```




```
best.lambda = lam.powers[which(min(mean.cv.mse)==mean.cv.mse)]
best.mod = smooth.spline(cars$speed, cars$dist, lambda=10^best.lambda, tol=1e-10)
plot.df = data.frame(speed=best.mod$x, pred=best.mod$y)
ggplot(plot.df, aes(x=speed)) +
  geom_point(data=cars, aes(y=dist)) +
  geom_line(aes(y=pred))
```


125
100
75
50
25
0
dist
5
10
15
20
25
speed