R for Data Science (IV)

Koji Mizumura

2018/08/03

# IV Model

# Chapter 22. Introduction

Now that you are equippted with powerful programming tools, we can finally return to modeling. You will use your new tools of data wrangling and programming, to fit many models and understand how they work.

The focus of this book is on exploration, not confimation or formal inference. But you will leanr a few basic tools to help you understand the variation within your models.

The gola of a model is to provide a simple low-dimensional summary of a dataset. Ideally, the model will capture true “signals” (i.e. patterns generated by the phenomenon of interest), and ignore “noise” (i.e. random variation that you’re not interested in). Here we only cover “predictive” models, which, as the name suggests, generate predictions. There is another type of model that we’re not going to discuss: “data discovery” models. These models don’t make predictions, but instead help you discover interesting relationships within your data. (These two categories of models are sometimes called supervised and unsupervised, but I don’t think that terminology is particularly illuminating.)

This book is not going to give you a deep understanding of the mathematical theory that underlies models. It will, however, build your intuition about how statistical models work, and give you a family of useful tools that allow you to use models to better understand your data:

* In [model basics](http://r4ds.had.co.nz/model-basics.html), you’ll learn how models work mechanistically, focussing on the important family of linear models. You’ll learn general tools for gaining insight into what a predictive model tells you about your data, focussing on simple simulated datasets.
* In [model building](http://r4ds.had.co.nz/model-building.html), you’ll learn how to use models to pull out known patterns in real data. Once you have recognised an important pattern it’s useful to make it explicit in a model, because then you can more easily see the subtler signals that remain.
* In [many models](http://r4ds.had.co.nz/many-models.html), you’ll learn how to use many simple models to help understand complex datasets. This is a powerful technique, but to access it you’ll need to combine modeling and progamming tools.

These topics are notable because of what they don’t include: any tools for quantitatively assessing models. This is deliberate : precisely quantifying a model requires a couple of big ideas that we just don’t have the space to cover here. For now, you’ll rely on qualitative assessment and your natural speticism. In [Learning more about models](http://r4ds.had.co.nz/model-building.html#learning-more-about-models), we’ll point you to other resources where you can learn more.

## 22.1 Hypothesis generation vs. Hypothesis confirmation

In this book, we are going to use models as a tool for exploration, completing the trifecta of the tools for EDA that were introduced in Part 1. his is not how models are usually taught, but as you will see, models are an important tool for exploration. Traditionally, the focus of modelling is on inference, or for confirming that an hypothesis is true. Doing this correctly is not complicated, but it is hard. There is a pair of ideas that you must understand in order to do inference correctly:

1. Each observation can either be used for exploration or confimration, not both.
2. You can use an observation as many times as you like for exploration, but you can only use it once for confirmation. As soon as you use an observation twice, you’ve switched from confirmation to exploration.

This is necessary because to confirm a hypothesis you must use data independent of the data you used to generate the hypothesis. Otherwise you will be over optimistic. There is absolutely nothing wrong with exploration, but yo ushould neve sell an exploratory analysis as a confirmatory analysis because it is fundamentally misleading.

If you are serious about doing an confirmation analysis, one approach is to split your data into three pieces before you begin the analysis:

1. 60% of your data goes into a **training** (or exploration) set. You’re allowed to do anything you like with this data: visualise it and fit tons of models to it.
2. 20% goes into a **query** set. You can use this data to compare models or visualisations by hand, but you’re not allowed to use it as part of an automated process.
3. 20% is held back for a **test** set. You can only use this data ONCE, to test your final model.

# Chapter 23: Model basics

## 23.1 Introduction

The goal of a model is to provide a simple low-dimensional summary of a dataset. In the context of this book we’re going to use models to partition data into patterns and residuals. Strong patterns will hide subtler trends, so we’ll use models to help peel back layers of structure as we explore a dataset.

However, before we can start using models on interestin, real, datasets, you need to understand the basics of how model works. For that reason, this chapter of the book is unique because it uses only simulated datasets. These datasets are very simple, and not at all interesting, but they will help you understand the essence of modelling before you apply the same techniques to real data in the next chapter.

There are two parts to a model: 1.First, you define a **family of models** that express a precise, but generic, pattern that you want to capture. For example, the pattern might be a straight line, or a quadratic curve. You will express the model family as an equation like . Here, x and y are known variables from your data, and and are parameters that can vary to capture different patterns.

1. Next, you generate a **fitted model** by finding the model from the family that is the closest to your data. This takes the generic model family and makes it specific, like .

It’s important to understand that a fitter model is just the closest model from afamily of models. That implies that you have the “best” model (according to some criteria); it does not imply that you have a good model and it certainly doesn’t imply that the model is “true”. George Box puts this well in his famous aphrism:

All models are wrong, but some are useful

It’s worth reading the fuller context of the quote:

Now it would be very remarkable if any system existing in the real world could be exactly represented by any simple model. However, cunningly chosen parsimonious models often do provide remarkably useful approximations. For example, the law PV = RT relating pressure P, volume V and temperature T of an “ideal” gas via a constant R is not exactly true for any real gas, but it frequently provides a useful approximation and furthermore its structure is informative since it springs from a physical view of the behavior of gas molecules.

For such a model there is no need to ask the question “Is the model true?”. If “truth” is to be the “whole truth” the answer must be “No”. The only question of interest is “Is the model illuminating and useful?”.

The goal of a model is not to uncover truth, but to discovera simple approaximation that is still useful.

### 23.1.1. Pre-requisites

In this chapter, we will use the modelr package which wraps around base R’s modeling functions to make them work naturally in a pipe.

- echo: show/hide - eval: evaluate/not code - include: evaluate, but include/not include

library(tidyverse)  
library(modelr)  
options(na.action=na.warn)

## 23.2 A simpler model

Lets take a look at the simulated dataset sim1, included with the modelr package. It contains two continuous variables, x, y. Let’s plot them to see how they’re related:

library(ggplot2)

## Warning: package 'ggplot2' was built under R version 3.5.1

library(modelr)  
library(tidyverse)

## Warning: package 'tidyverse' was built under R version 3.5.1

## -- Attaching packages --------------------------------- tidyverse 1.2.1 --

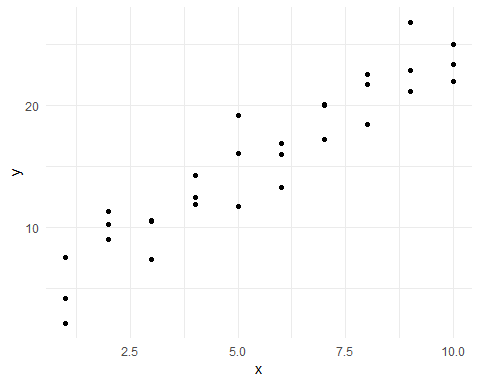
## √ tibble 1.4.2 √ purrr 0.2.5  
## √ tidyr 0.8.1 √ dplyr 0.7.6  
## √ readr 1.1.1 √ stringr 1.3.1  
## √ tibble 1.4.2 √ forcats 0.3.0

## Warning: package 'purrr' was built under R version 3.5.1

## Warning: package 'dplyr' was built under R version 3.5.1

## -- Conflicts ------------------------------------ tidyverse\_conflicts() --  
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag() masks stats::lag()

ggplot(sim1,aes(x,y))+  
 geom\_point()+theme\_minimal()



You can see a strong pattern in the data. Let’s use a model to capture that pattern and make it explicit. It’s our job to supply the basic form of the model. In this case, the relationship looks linear, i.e., . Let’s start by getting a fell for what models from that family look like by randomly generating a few and overlaying them on the data.

For this simple case, we can use geom\_abline() which takes a slope and intercept as parameters. Later on we’ll learn more general techniques that work with any model.

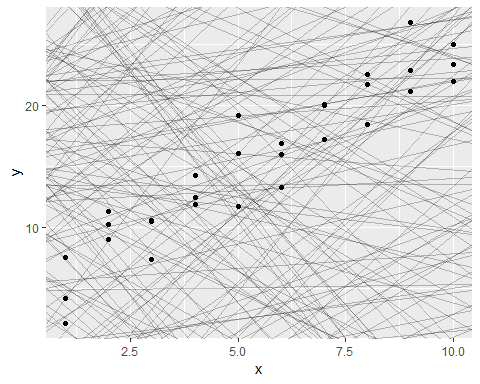
models <- tibble(  
 a1=runif(250,-20,40),  
 a2=runif(250,-5,5)  
)  
# runif(250,min=-20,max=40)  
  
modelsX <- tibble(  
 a1=runif(1,-20,40),  
 a2=runif(1,-5,5)  
) %>% print()

## # A tibble: 1 x 2  
## a1 a2  
## <dbl> <dbl>  
## 1 -2.66 1.25

head(models)

## # A tibble: 6 x 2  
## a1 a2  
## <dbl> <dbl>  
## 1 -4.99 -4.86   
## 2 34.8 -4.21   
## 3 20.4 3.60   
## 4 -19.6 2.90   
## 5 -3.00 0.0264  
## 6 20.1 4.79

ggplot(data=sim1,aes(x,y))+  
 geom\_abline(aes(intercept=a1,slope=a2),data=models,alpha=1/4)+  
 geom\_point()



There are 250 models on the above plot, but a lot are really bad! We need to find the good models by making precise our intuition that a good model is “close” to the data. We need a way to quantify the distance between the data and a model. Then we can fit the model by finding the value of a\_0 and a\_1 that generate the model with the smallest distance from this data.

One easy place to start is to find the vertical distance between each point and the model, as in the following diagram. (Note that I’ve shifted the x values slightly so you can see the individual distances.)

This distance is just the difference between the y value given by the model (the **prediction**), and the actual y value in the data (the **response**).

To compute this distance, we first turn our model family into an R function. This takes the model parameters and the data as inputs, and gives values predicted by the model as output:

model1 <- function(a,data){  
 a[1]+data$x\*a[2]  
}  
model1(c(7,1,5),sim1)

## [1] 8 8 8 9 9 9 10 10 10 11 11 11 12 12 12 13 13 13 14 14 14 15 15  
## [24] 15 16 16 16 17 17 17

Next, we need some way to compute an overall distance between the predicted and actual values. In other words, the plot above shows 30 distances: how do we collapse that into a single number?

One common way to do this in statistics to use the *“root-mean-squared deviation”*. We compute the difference between actual and predicted, square them, average them, and the take the square root. This distance has lots of appealing mathematical properties, which we’re not going to talk about here. You’ll just have to take my word for it!

measure\_distance <- function(mod,data){  
 diff <- data$y - model1(mod,data)  
 sqrt(mean(diff^2))  
}  
measure\_distance(c(7,1.5),sim1)

## [1] 2.665212

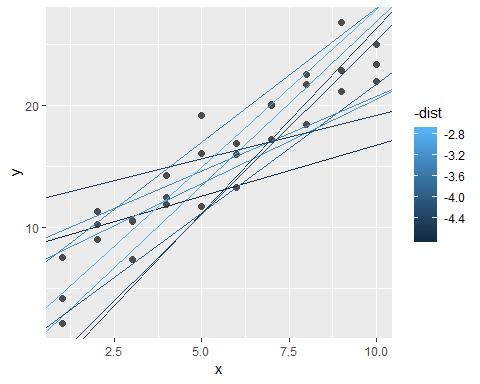
Now we can use purrr to compute the distance for all the models defined above. We need a helper function because our distance function expects the model as a numeric vector of length 2.

sim1\_dist <- function(a1,a2){  
 measure\_distance(c(a1,a2),sim1)  
}  
  
models <- models %>%   
 mutate(dist=purrr::map2\_dbl(a1,a2,sim1\_dist))  
models

## # A tibble: 250 x 3  
## a1 a2 dist  
## <dbl> <dbl> <dbl>  
## 1 -4.99 -4.86 51.3  
## 2 34.8 -4.21 18.5  
## 3 20.4 3.60 25.1  
## 4 -19.6 2.90 19.5  
## 5 -3.00 0.0264 19.4  
## 6 20.1 4.79 32.0  
## 7 39.4 3.01 40.6  
## 8 30.1 4.10 37.7  
## 9 37.3 1.02 27.7  
## 10 37.8 -1.38 17.8  
## # ... with 240 more rows

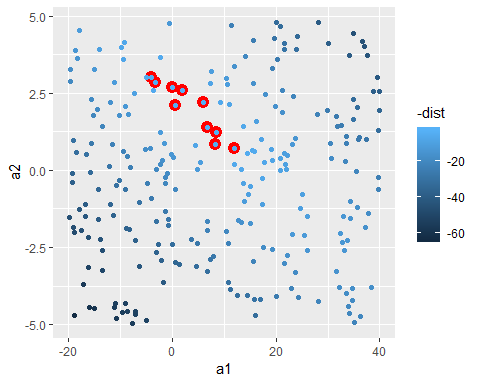
Next, let’s overlay the 10 best models on the data. I’ve coloured the models by -dist: this is an easy way to make sure that the best models (i.e., the ones with the smallest distance) get the highest colours.

ggplot(sim1,aes(x,y))+  
 geom\_point(size=2,colour="grey30")+  
 geom\_abline(  
 aes(intercept=a1,slope=a2,colour=-dist),  
 data=filter(models,rank(dist)<=10)  
 )



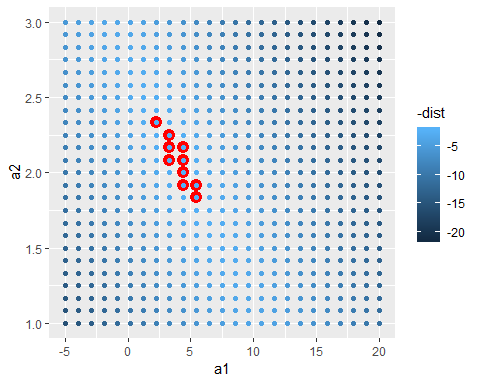
We can also think about these models as observations, and visualising with a scatterplot of a1 vs a2, again coloured by -dist. We can no longer directly see how the model compares to the data, but we can see many models at once. Again, I’ve highlighted the 10 best models, this time by drawing red circles underneath them.

ggplot(models,aes(a1,a2))+  
 geom\_point(data=filter(models,rank(dist)<=10),size=4,colour="red")+  
 geom\_point(aes(colour=-dist))



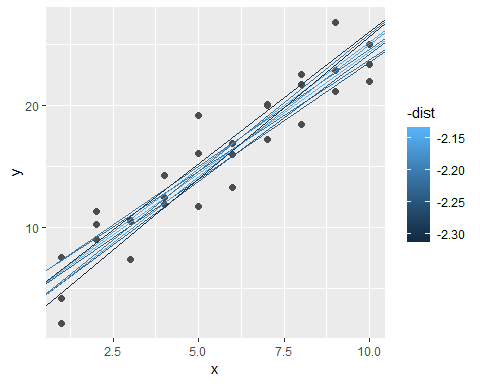
Instead of trying lots of random models, we could be more systematic and generate an evenly spaced grid of points (this is called a grid search). I picked the parameters of the grid roughly by looking at where the best models were in the plot above.

grid <- expand.grid(  
 a1=seq(-5,20,length=25),  
 a2=seq(1,3,length=25)  
) %>%   
 mutate(dist=purrr::map2\_dbl(a1,a2,sim1\_dist))  
  
grid %>%   
 ggplot(aes(a1,a2))+  
 geom\_point(data=filter(grid,rank(dist)<=10),size=4,colour="red")+  
 geom\_point(aes(colour=-dist))



When you overlay the best 10 models back on the original data, they all look pretty good:

ggplot(sim1,aes(x,y))+  
 geom\_point(size=2,colour="grey30")+  
 geom\_abline(  
 aes(intercept = a1, slope = a2, colour=-dist),  
 data=filter(grid,rank(dist)<=10)  
 )

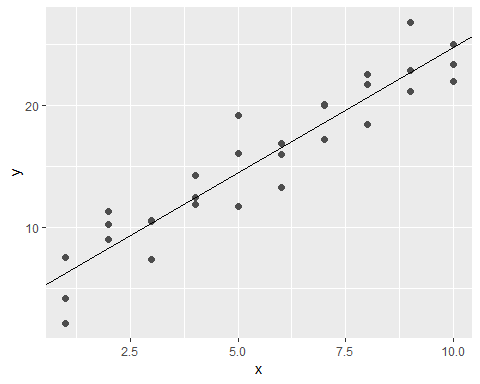


You could imagine iteratively making the grid finer and finer until you narrowed in on the best model. But there’s a better way to tackle that problem: a numerical minimisation tool called Newton-Raphson search. The intuition of Newton-Raphson is pretty simple: you pick a starting point and look around for the steepest slope. You then ski down that slope a little way, and then repeat again and again, until you can’t go any lower. In R, we can do that with optim():

best <- optim(c(0,0),measure\_distance,data=sim1)  
best$par

## [1] 4.222248 2.051204

#> [1] 4.22 2.05  
  
# ggplot(sim1,aes(x,y))+  
# geom\_point(size=2,colour="light grey")+  
# geom\_abline(intercept=c(3,2),slope=c(2,3),colour="blue")  
  
ggplot(sim1,aes(x,y))+  
 geom\_point(size=2,colour="grey30")+  
 geom\_abline(intercept=best$par[1],slope=best$par[2])



Don’t worry too much about the details of how optim() works. It’s the intuition that’s important here. If you have a function that defines the distance between a model and a dataset, an algorithm that can minimise that distance by modifying the parameters of the model, you can find the best model. The neat thing about this approach is that it will work for any family of models that you can write an equation for.

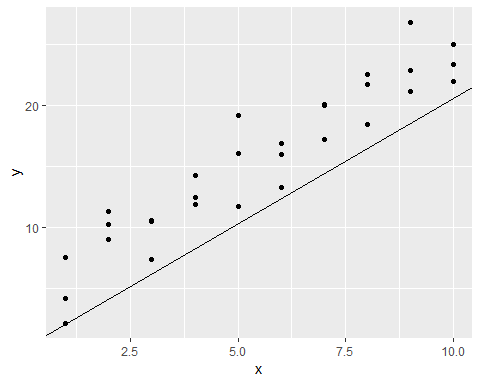
There’s one more approach that we can use for this model, because it’s a special case of a broader family: linear models. A linear model has the general form . So this simple model is equivalent to a general linear model where n is 2 and x\_1 is x. R has a tool specifically designed for fitting linear models called lm(). lm() has a special way to specify the model family: formulas. Formulas look like y ~ x, which lm() will translate to a function like y = a\_1 + a\_2 \* x. We can fit the model and look at the output:

sim1\_mod <- lm(y~x,data=sim1)  
coef(sim1\_mod)

## (Intercept) x   
## 4.220822 2.051533

ggplot(data=sim1,aes(x,y))+  
 geom\_point()+  
 geom\_abline(yintercept=coef(sim1\_mod)[1],slope=coef(sim1\_mod)[2])

## Warning: Ignoring unknown parameters: yintercept



These are exactly the same values we got with optim()! Behind the scenes lm() doesn’t use optim() but instead takes advantage of the mathematical structure of linear models. Using some connections between geometry, calculus, and linear algebra, lm() actually finds the closest model in a single step, using a sophisticated algorithm. This approach is both faster, and guarantees that there is a global minimum.

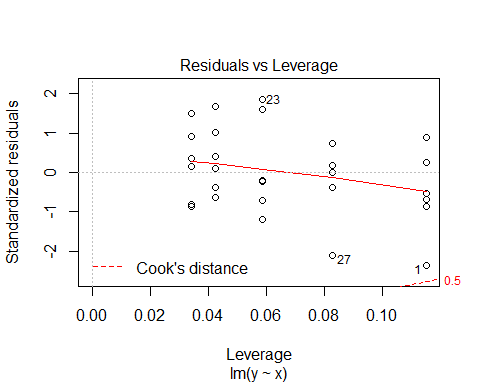
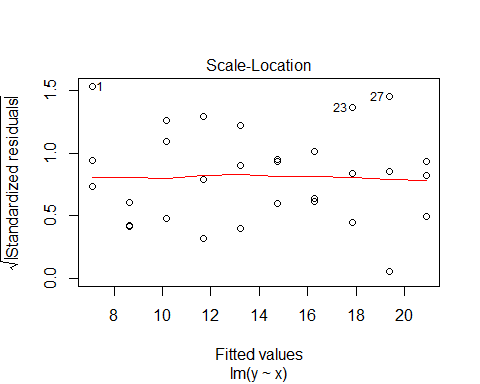
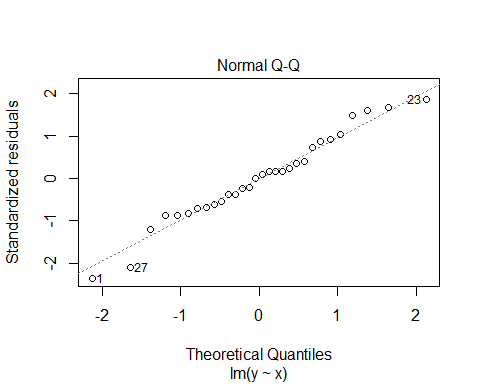
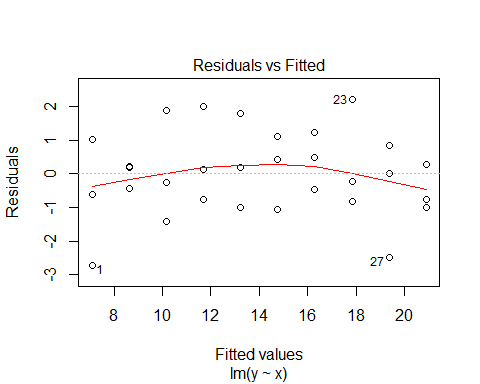
### 23.2.1 Exercises

1. One downside of the linear model is that it is sensitive to unusual values because the distance incorporates a squared term. Fit a linear model to the simulated data below, and visualise the results. Rerun a few times to generate different simulated datasets. What do you notice about the model?

sim1a <- tibble(  
 x = rep(1:10, each = 3),  
 y = x \* 1.5 + 6 + rt(length(x), df = 2)  
)  
sim1a\_mod <- lm(y~x,data=sim1a)  
coef(sim1a\_mod)

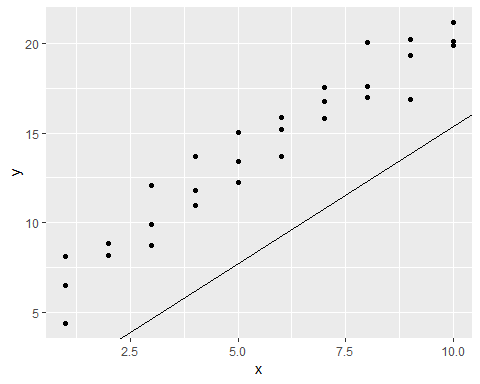
## (Intercept) x   
## 5.548376 1.535891

plot(sim1a\_mod)



ggplot(data=sim1a,aes(x,y))+  
 geom\_point()+  
 geom\_abline(yintercept=coef(sim1a\_mod)[1],slope=coef(sim1a\_mod)[2])

## Warning: Ignoring unknown parameters: yintercept



1. One way to make linear models more robust is to use a different distance measure. For example, instead of root-mean-squared distance, you could use mean-absolute distance:

measure\_distance <- function(mod, data) {  
 diff <- data$y - model1(mod, data)  
 mean(abs(diff))  
}

Use optim() to fit this model to the simulated data above and compare it to the linear model.

1. One challenge with performing numerical optimisation is that it’s only guaranteed to find one local optimum. What’s the problem with optimising a three parameter model like this?

model1 <- function(a, data) {  
 a[1] + data$x \* a[2] + a[3]  
}

## 23.3 Visualizing models

For simple models, ike the one above, you can figure out what pattern the model captures by carefully studying the model family and the fitted coefficients. And if you ever take a statistics course on modelling, you’re likely to spend a lot of time doing just that. Here, however, we’re going to take a different tack. We’re going to focus on understanding a model by looking at its predictions. This has a big advantage: every type of predictive model makes predictions (otherwise what use would it be?) so we can use the same set of techniques to understand any type of predictive model.

It’s also useful to see what the model doesn’t capture, the so-called residuals which are left after subtracting the predictions from the data. Residuals are powerful because they allow us to use models to remove striking patterns so we can study the subtler trends that remain.

### 23.3.1 Predictions

To visualise the predictions from a model, we start by generating an evenly spaced grid of values that covers the region where our data lies. The easiest way to do that is to use modelr::data\_grid(). Its first argument is a data frame, and for each subsequent argument it finds the unique variables and then generates all combinations:

grid <- sim1 %>%   
 data\_grid(x)  
grid

## # A tibble: 10 x 1  
## x  
## <int>  
## 1 1  
## 2 2  
## 3 3  
## 4 4  
## 5 5  
## 6 6  
## 7 7  
## 8 8  
## 9 9  
## 10 10

(This will get more interesting when we start to add more variables to our model.)

Next we add predictions. We’ll use modelr::add\_predictions() which takes a data frame and a model. It adds the predictions from the model to a new column in the data frame:

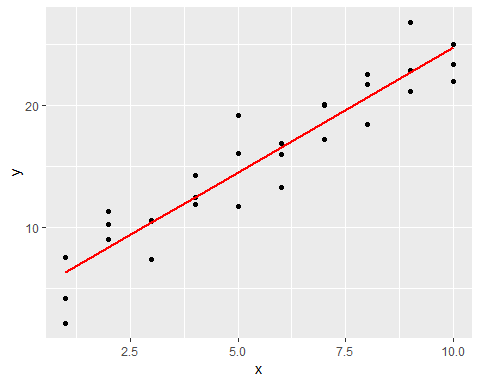
grid <- grid %>%   
 add\_predictions(sim1\_mod)   
grid

## # A tibble: 10 x 2  
## x pred  
## <int> <dbl>  
## 1 1 6.27  
## 2 2 8.32  
## 3 3 10.4   
## 4 4 12.4   
## 5 5 14.5   
## 6 6 16.5   
## 7 7 18.6   
## 8 8 20.6   
## 9 9 22.7   
## 10 10 24.7

You can also use this function to add predictions to your original dataset.)

Next, we plot the predictions. You might wonder about all this extra work compared to just using geom\_abline(). But the advantage of this approach is that it will work with any model in R, from the simplest to the most complex. You’re only limited by your visualisation skills. For more ideas about how to visualise more complex model types, you might try <http://vita.had.co.nz/papers/model-vis.html>.

ggplot(sim1, aes(x))+  
 geom\_point(aes(y=y))+  
 geom\_line(aes(y=pred),data=grid,colour="red",size=1)



### 23.3.2 Residuals

The flip-side of predictions are residuals. The predictions tells you the pattern that the model has captured, and the residuals tell you what the model has missed. The residuals are just the distances between the observed and predicted values that we computed above.

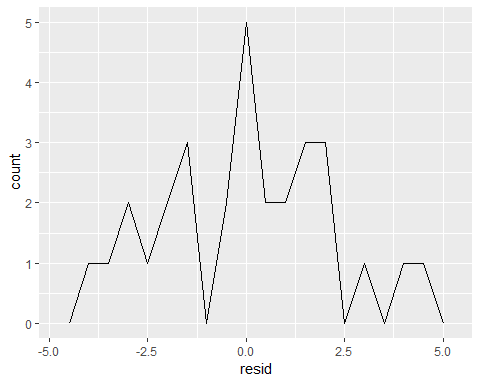
We add residuals to the data with add\_residuals(), which works much likeadd\_predictions(). Note, however, that we use the original dataset, not a manufactured grid. This is because to compute residuals we need actual y values.

sim1 <- sim1 %>%   
 add\_residuals(sim1\_mod)  
sim1

## # A tibble: 30 x 3  
## x y resid  
## <int> <dbl> <dbl>  
## 1 1 4.20 -2.07   
## 2 1 7.51 1.24   
## 3 1 2.13 -4.15   
## 4 2 8.99 0.665   
## 5 2 10.2 1.92   
## 6 2 11.3 2.97   
## 7 3 7.36 -3.02   
## 8 3 10.5 0.130   
## 9 3 10.5 0.136   
## 10 4 12.4 0.00763  
## # ... with 20 more rows

There are a few different ways to understand what the residuals tell us about the model. One way is to simply draw a frequency polygon to help us understand the spread of the residuals:

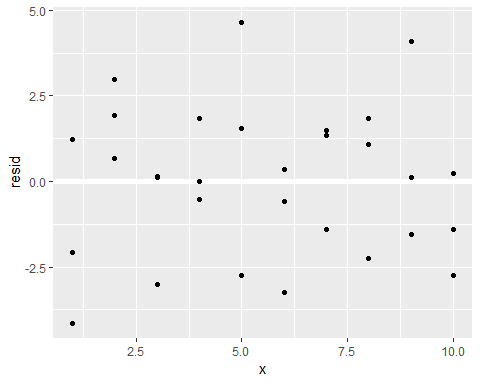
ggplot(sim1,aes(resid))+  
 geom\_freqpoly(binwidth=0.5)



This helps you calibrate the quality of the model: how far away are the predictions from the observed values? Notee average of the residual will always be 0.

You’ll often want to recreate plots using the residuals instead of the original predictor. You’ll see a lot of that in the next chapter.

library(ggplot2)  
library(tidyverse)  
library(modelr)  
  
ggplot(sim1,aes(x,resid))+  
 geom\_ref\_line(h=0)+  
 geom\_point()

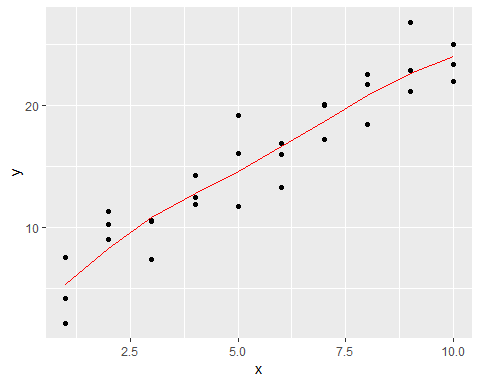


This looks like random noise, suggesting that our model has done a good job of capturing the patterns in the dataset.

### 23.3.3 Exercises

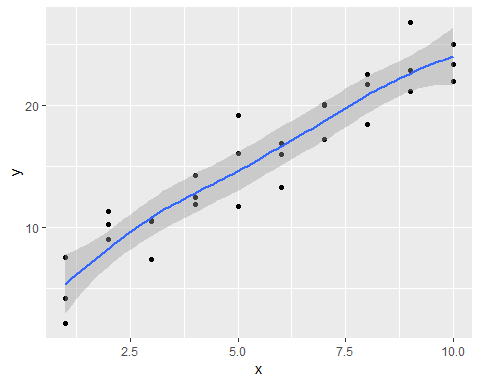
1. Instead of using lm() to fit a straight line, you can use loess() to fit a smooth curve. Repeat the process of model fitting, grid generation, predictions, and visualisation on sim1 using loess() instead of lm(). How does the result compare to geom\_smooth()?

mod1 <- loess(y~x,data=sim1)  
sim1 %>%   
 add\_predictions(mod1) %>%   
 ggplot(aes(x,y))+  
 geom\_point()+  
 geom\_line(aes(y=pred),colour="red")



sim1 %>%   
 ggplot(aes(x,y))+  
 geom\_point()+  
 geom\_smooth()

## `geom\_smooth()` using method = 'loess' and formula 'y ~ x'

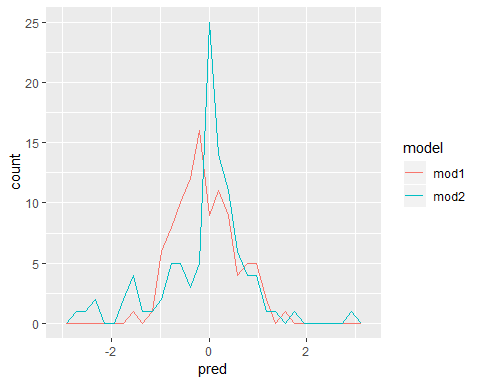


1. add\_predictions() is paired with gather\_predictions() and spread\_predictions(). How do these three functions differ?

gather\_predictions works for gathering serveral models into a tidy data.

new\_data <- tibble(  
 y=rnorm(100),  
 x=y+rnorm(100,mean=5),  
 z=y\*runif(100,max=100)  
)  
  
mod1 <- lm(y~x,data=new\_data)  
mod2 <- lm(y~z,data=new\_data)  
  
final\_data <-   
 new\_data %>%   
 gather\_predictions(mod1,mod2)  
  
final\_data %>%   
 ggplot(aes(pred, colour = model)) +  
 geom\_freqpoly()

## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.



spread\_predictions does the same but adds the predictions as columns rather than as tidy dataset.

new\_data %>%   
 spread\_predictions(mod1,mod2)

## # A tibble: 100 x 5  
## y x z mod1 mod2  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 -0.115 4.06 -9.69 -0.465 -0.0703  
## 2 1.11 5.99 46.8 0.362 0.738   
## 3 -1.05 6.69 -44.5 0.660 -0.568   
## 4 1.31 7.36 63.0 0.949 0.969   
## 5 -1.99 1.68 -169. -1.48 -2.35   
## 6 0.333 3.49 15.8 -0.713 0.294   
## 7 0.620 5.55 14.3 0.171 0.272   
## 8 0.0916 3.12 4.83 -0.871 0.137   
## 9 1.13 6.14 45.4 0.427 0.718   
## 10 0.309 4.29 23.8 -0.369 0.409   
## # ... with 90 more rows

1. What does geom\_ref\_line() do? What package does it come from? Why is displaying a reference line in plots showing residuals useful and important?

geom\_ref\_line is a nice addition to ggplot2 although it comes from modelr. It’s purpose is just adding a reference line in a plot. It’s very practical for analyzing residuals because that way you can figure out if many points are above/below a certain point, and whether the models is worse/better at being overly pessimistic or overly positive.

Why might you want to look at a frequency polygon of absolute residuals? What are the pros and cons compared to looking at the raw residuals?

Looking at absolute residuals would work really well to distinguish the magnitude of bad or good predictions. Moreover, it server well to identiy strong outliers. On the other hand, the con side is that you don’t know whether that strong prediction is either positive or negative. That’s why it’s better to look at raw residuals for that different question.

## 23.4 Formulas and model families

You have seen formulas before when using facet\_wrap() and facet\_grid(). In R, formulas provide a general way of getting “special behaviour”. Rather than evaluating the values of the variables right away, they capture them so they can be interpreted by the function.

The majority of modelling functions in R use a standard conversion from formulas to functions. You’ve seen one simple conversion already: is translated to . If you want to see what R actually does, you can use the model\_matrix() function. It takes a data frame and a formula and returns a tibble that defines the model equation: each column in the output is associated with one coefficient in the model, the function is always . For the simplest case of y ~ x1 this shows us something interesting:

df <- tribble(  
 ~y,~x1,~x2,  
 4,2,5,  
 5,1,6  
)  
  
model\_matrix(df,y~x1)

## # A tibble: 2 x 2  
## `(Intercept)` x1  
## <dbl> <dbl>  
## 1 1 2  
## 2 1 1

The way that R adds the intercept to the model is just by having a column that is full of ones. By default, R will always add this column. If you don’t want, you need to explicitly drop it with -1:

model\_matrix(df,y~x1-1)

## # A tibble: 2 x 1  
## x1  
## <dbl>  
## 1 2  
## 2 1

The model matrix grows in an unsurprising way when you add more variables to the the model:

model\_matrix(df,y~x1+x2)

## # A tibble: 2 x 3  
## `(Intercept)` x1 x2  
## <dbl> <dbl> <dbl>  
## 1 1 2 5  
## 2 1 1 6

This formula notation is sometimes called “Wilkinson-Rogers notation” and was initially described in Symbolic Description of Factorial Models for Analysis of Variance, by G. N. Wilkinson and C. E. Rogers <https://www.jstor.org/stable/2346786>. It’s worth digging up and reading the original paper if you’d like to understand the full details of the modelling algebra.

The following sections expand on how this formula notation works for categorical variables, interactions, and transformation.

### 23.4.1 Categorical variables

Generating a function from a formula is straight forward when the predictor is continuous, but things get a bit more complicated when the predictor is categorical. Imagine you have a formula like y ~ sex, where sex could either be male or female. It doesn’t make sense to convert that to a formula like because sex isn’t a number - you can’t multiply it! Instead what R does is convert it to where sex\_male is one if sex is male and zero otherwise:

df <- tribble(  
 ~sex,~response,  
 "male",1,  
 "female",2,  
 "male",1  
)  
  
model\_matrix(df,response~sex)

## # A tibble: 3 x 2  
## `(Intercept)` sexmale  
## <dbl> <dbl>  
## 1 1 1  
## 2 1 0  
## 3 1 1

You might wonder why R also doesn’t create a sexfemale column. The problem is that would create a column that is perfectly predictable based on the other columns (i.e. sexfemale = 1 - sexmale). Unfortunately the exact details of why this is a problem is beyond the scope of this book, but basically it creates a model family that is too flexible, and will have infinitely many models that are equally close to the data.

Fortunately, however, if you focus on visualizing predictions you don’t need to worry about the exact parameterisation. Let’s look at some data and models to make that concrete. Here’s the sim2 dataset from modelr:

library(ggplot2)  
library(modelr)  
  
ggplot(sim2)+  
 geom\_point(aes(x,y))



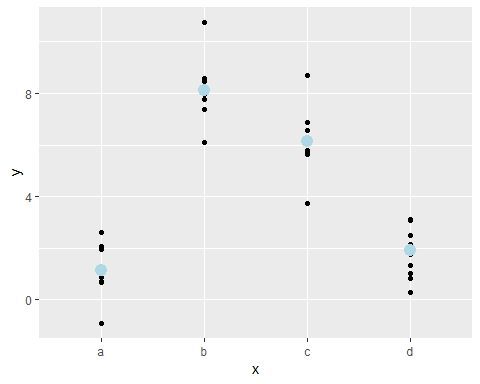
We can fit a model to it, and generate predictions:

mod2 <- lm(y~x,data=sim2)  
  
grid <- sim2 %>%   
 data\_grid(x) %>%   
 add\_predictions(mod2)  
grid

## # A tibble: 4 x 2  
## x pred  
## <chr> <dbl>  
## 1 a 1.15  
## 2 b 8.12  
## 3 c 6.13  
## 4 d 1.91

Effectively, a model with a categorical x will predict the mean value for each category. (Why? Because the mean minimises the root-mean-squared distance.) That’s easy to see if we overlay the predictions on top of the original data:

ggplot(sim2, aes(x))+  
 geom\_point(aes(y=y))+  
 geom\_point(data=grid,aes(y=pred),colour="light blue",size=4)



You can’t make predictions about levels that you didn’t observe. Sometimes you’ll do this by accident so it’s good to recognise this error message:

### 23.4.2 Interactions (Continuous and categorical)

What happens when you