MQIM R BOOTCAMP

R for Data Science (tidyverse package) and Model Building

Monday, Dec 7, 2020

1. Bootcamp Week Schedule
2. Data manipulation and Visualization
3. Model Building

Section 1

Bootcamp Week Schedule

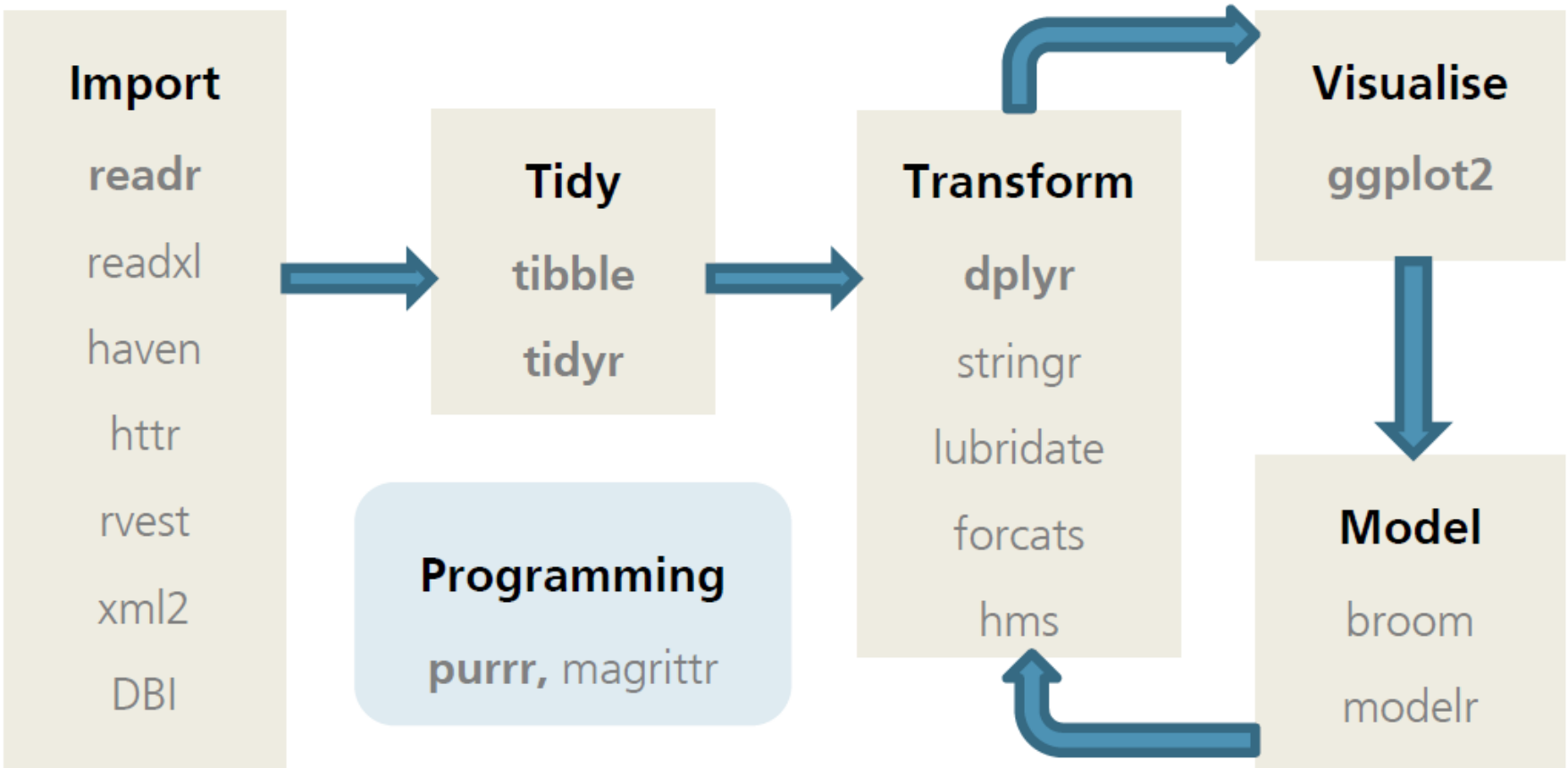
Table of Content

Section 2

Data manipulation and Visualization

Introduction to *tidyverse*

The *tidyverse* is a set of packages for doing data science, it loads six "core" libraries that provide tools for importing(*readr*), tidying(*tidyr*,*tibble*), manipulation(*dplyr*) and visualizing(*ggplot*) data, as well as support for functional programming(*purrr*).



Data Manipulation (*dplyr*)

key verbs:

- **select** for subsetting variables (columns)
- **mutate** for creating new variables from existing ones
- **filter** for subsetting observations (rows)
- **arrange** for rearranging/ordering observations
- **summarise** for computing various summary statistics

Along with the **group_by**, combining multiple simple pieces with the pipe operator %>% is a powerful way of solving complex problems.

Let's see a simple example by loading *data_df.RData*, which contains information about 40 large-cap companies.

```
> library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------- tid

## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

Data Manipulation (*dplyr*)

Suppose we have the following task:

Calculate the mean ROE by sector and 12-month momentum (positive or negative), making sure that there are at least 3 observations per group.

Finally, arrange the data in descending order of ROE.

Data Manipulation (*dplyr*)

What would you do it without advanced package except base R?

```
> data_df$Momentum <- ifelse(data_df$Return12m > 0, "Positive",
+                            "Negative")
> data_df <- transform(data_df, freq = as.numeric(as.character(
+ ave(Sector, Sector, Momentum, FUN = length))))
> data_df_filtered <- subset(data_df, freq >= 3)
> mean_roe <- aggregate(ROE ~ Sector + Momentum,
+                       data = data_df_filtered,
+ FUN = mean, na.rm = T)
> mean_roe <- mean_roe[order(mean_roe$ROE, decreasing = TRUE), ]
> head(mean_roe,3)
```

```
##                    Sector Momentum      ROE
## 3       Consumer Staples Positive 59.76600
## 2 Consumer Discretionary Positive 39.30333
## 5 Information Technology Positive 25.20818
```

Data Manipulation(*dplyr*)

It's cleaner and straight forward by using *dplyr*

```
> mean_roe <- data_df %>%
+ mutate(Momentum = ifelse(Return12m > 0, "Positive",
+                           "Negative")) %>%
+ group_by(Sector, Momentum) %>%
+ filter(n() >= 3) %>% # n() returns row count in each group
+ summarise(ROE = mean(ROE, na.rm = TRUE)) %>%
+ arrange(desc(ROE))
```

Data Manipulation(*dplyr*)

```
> head(mean_roe,3)
```

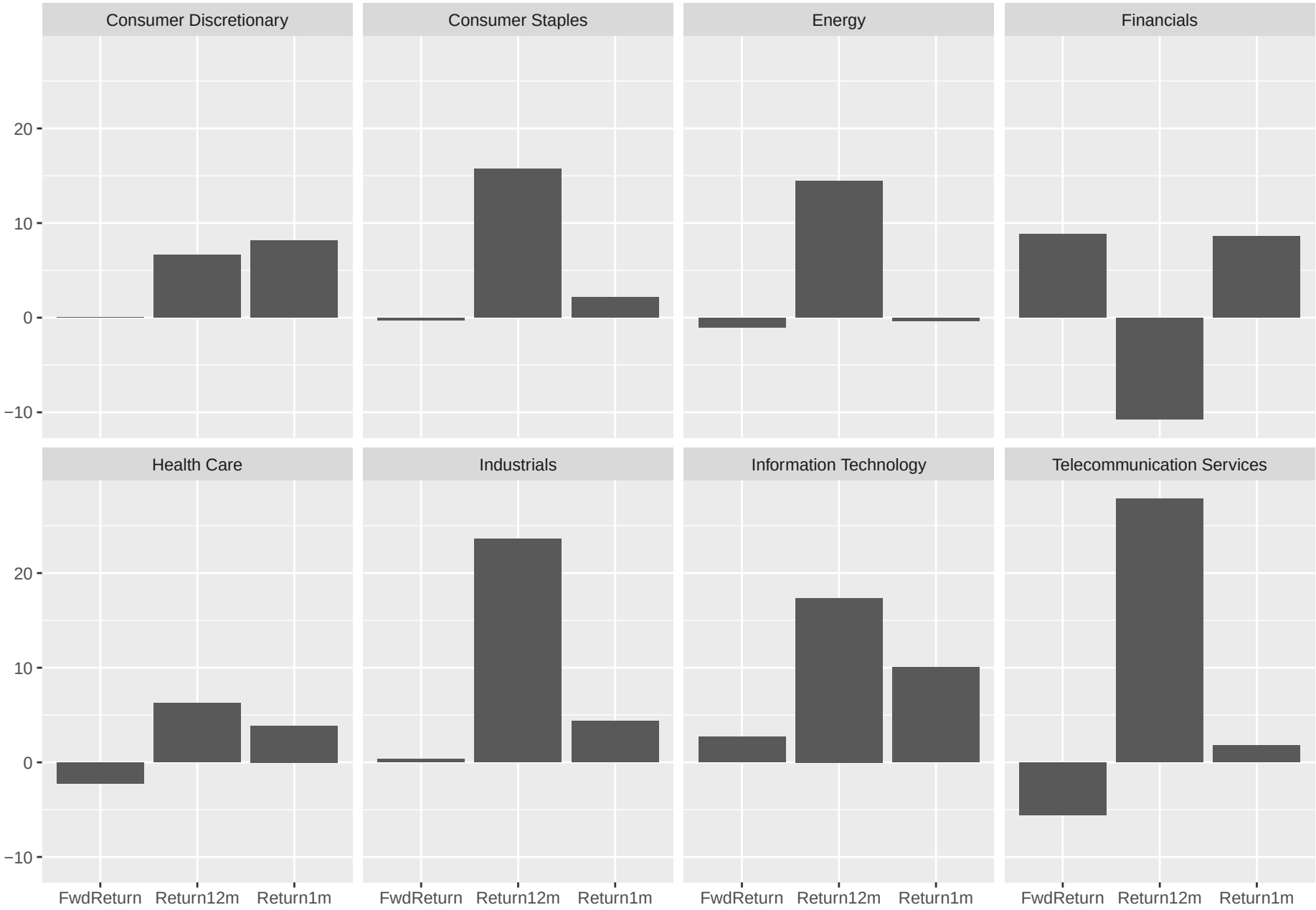
```
## # A tibble: 3 x 3
## # Groups:   Sector [3]
##   Sector                 Momentum   ROE
##   <chr>                  <chr>    <dbl>
## 1 Consumer Staples       Positive  59.8
## 2 Consumer Discretionary Positive  39.3
## 3 Information Technology Positive  25.2
```

Data Visualization(*ggplot*)

Charts are great tools to help you understand the data from the initial data research. If you don't want to spend too much time writing your own data manipulating and plotting functions, there is a nice and quick way to do it by combining *dplyer* and *ggplot*. Here is an example to visualize the sector mean returns of the data:

```
> myplot <- data_df %>%
+   group_by(Sector) %>%
+   summarise_at(vars(contains("Return")), funs(mean)) %>%
+   tidyr::gather(Variable, Return, -Sector) %>%
+   ggplot(aes(x = Variable, y = Return)) +
+   geom_bar(stat = "identity") + ylab("") +
+   facet_wrap(~ Sector, ncol = 4)
```

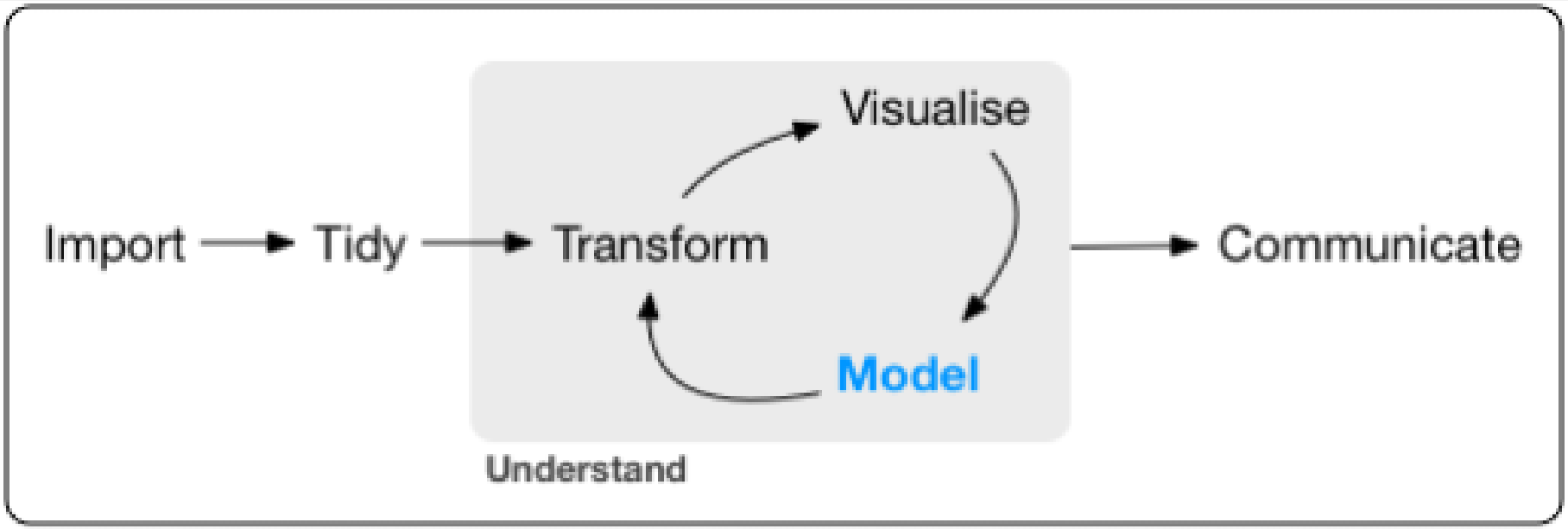
Data Visualization (*ggplot*)



Section 3

Model Building

Model Basics with *modelr*



Model Basics with *modelr*

The goal 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" (random variation that you're not interested in).

There are two parts to a model:

1. Define a *family of models* that express generic pattern that you want to capture. For example, the pattern might be a straight line, or a quadratic curve. You express the model family as an equation like $y = a_1 x + a_2$ or $y = a_1 x^{a_2}$. Here, x and y are known variables from your data, and a_1 and a_2 are parameters that can vary to capture different patterns.
2. Generate a *fitted model* by finding the model from the family that is the closet to your data. this takes the generic model family and makes it specific, like $y = 2x + 8$ or $y = 5x^2$

Note: fitted model only implies that you have the "closest" model based on some criteria, doesn't imply it's a true/good model.

Model Basics with *modelr*

Let's begin with a simple model.

load *modelr* package

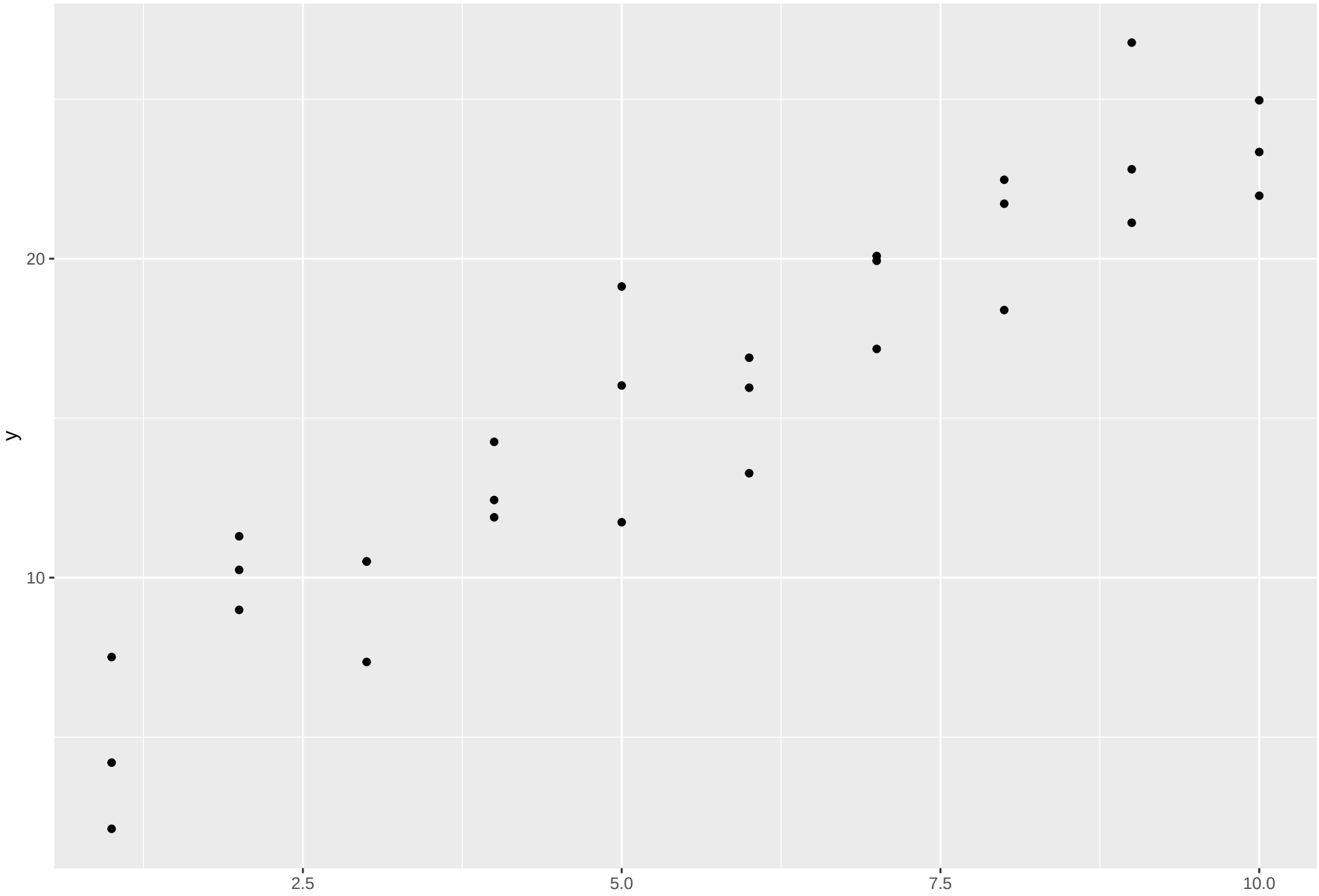
```
> library(tidyverse)
> library(modelr)
```

Model Basics with *modelr*

The simulated dataset *sim1*.

```
> ggplot(sim1, aes(x,y)) +
+   geom_point()
```

Model Basics with *modelr*



Model Basics with *modelr*

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 supple the basic form of the model. In this case, the relationship looks linear, i.e., $y = a_1 + a_2 * x$.

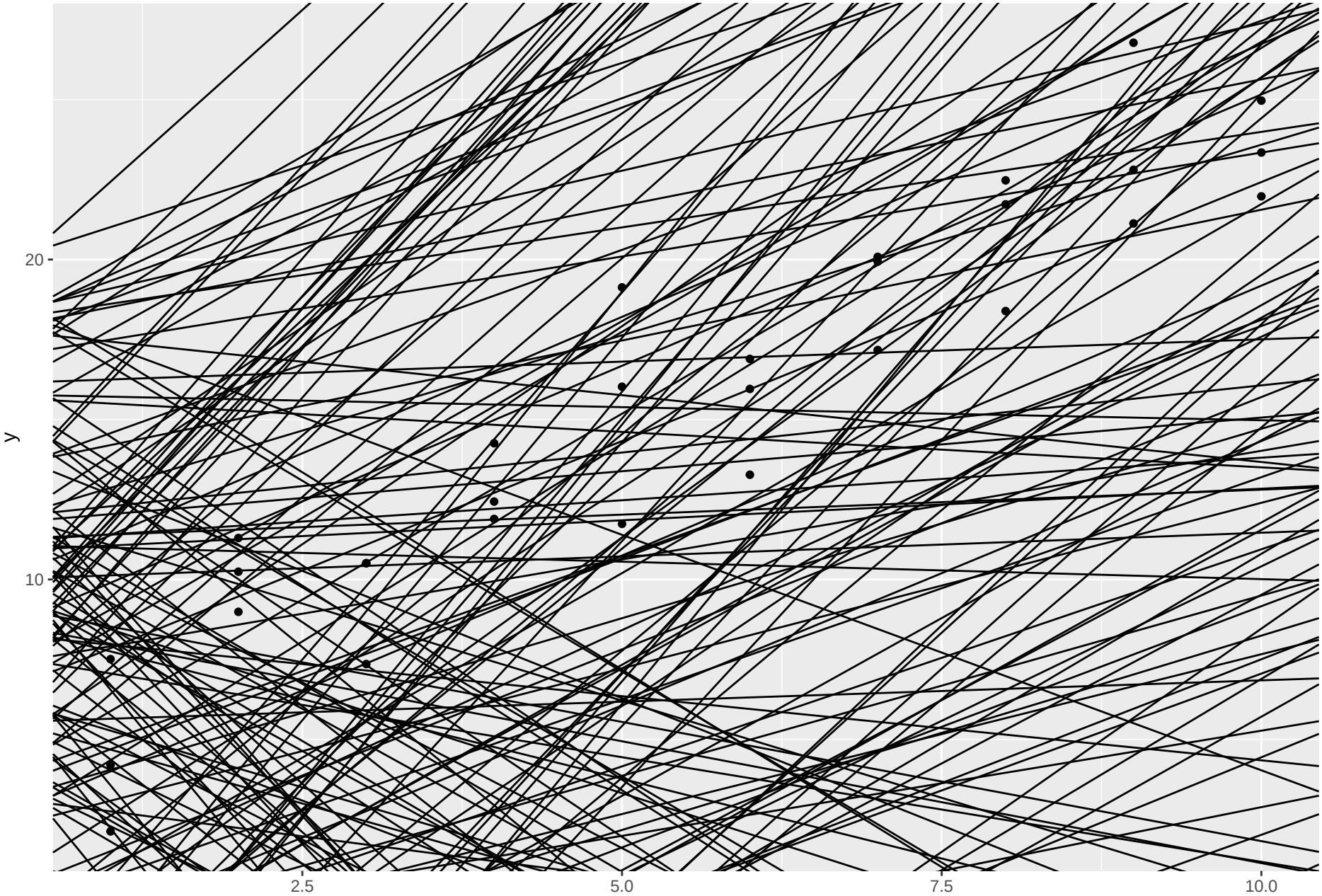
To see what models from that family look like, we can start to generate some random data. Then use the *geom_abline()*, which takes a slope and intercept as parameters.

```
> models <- tibble(
+   a1 = runif(250,-20,20),
+   a2 = runif(250,-5,5)
+ )
```

Model Basics with *modelr*

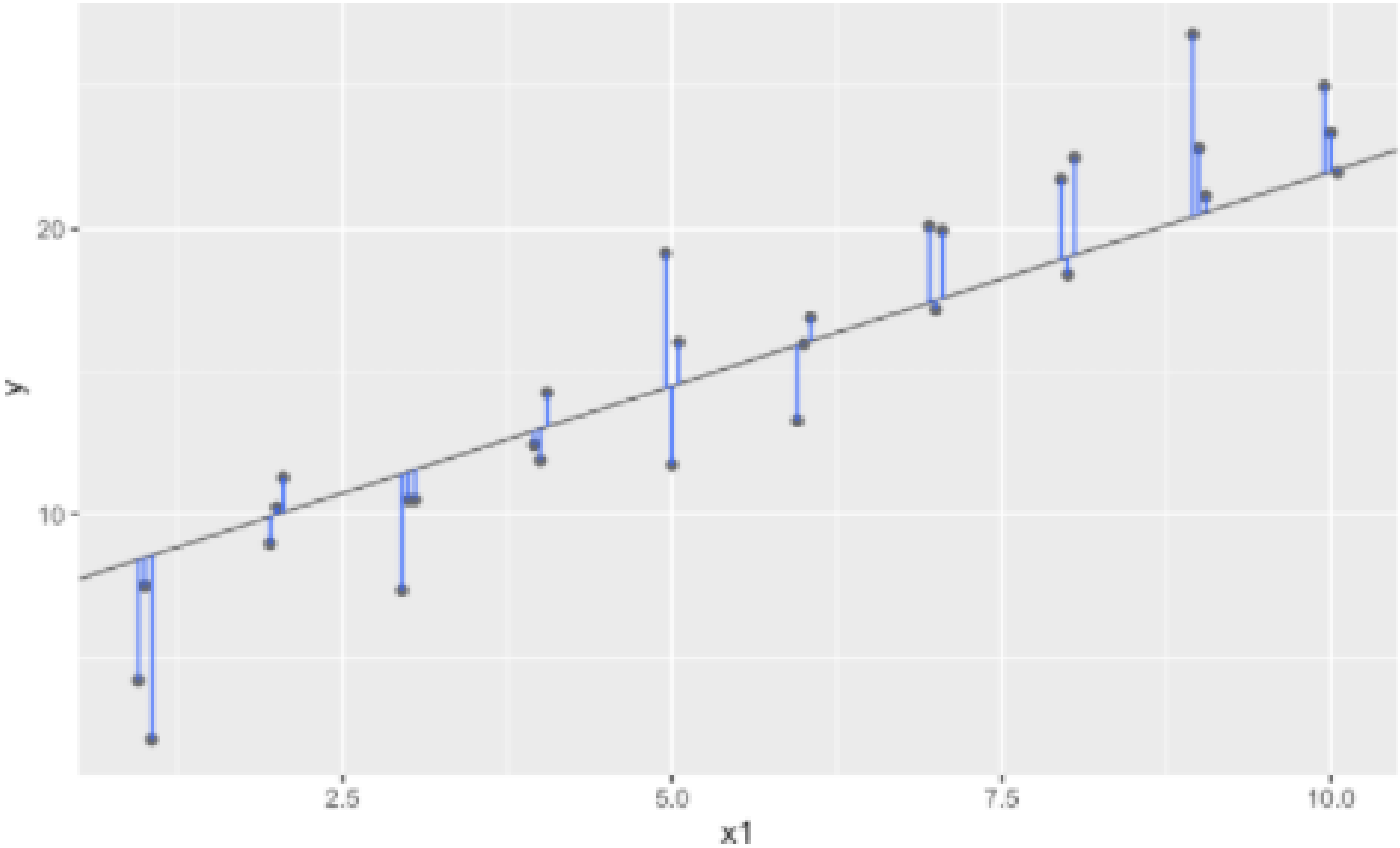
```
> ggplot(sim1, aes(x,y)) +
+   geom_abline(
+     aes(intercept = a1, slope = a2),
+     data = models) +
+   geom_point()
```

Model Basics with *modelr*



Model Basics with *modelr*

There are 250 models on this plot. We need a way to quantify the distance between the data and a model. Then we can fit the model by finding the values of a_1 and a_2 that generate the model with the smallest distance from this data.



Model Basics with *modelr*

One easy place to start is to find the vertical distance between the y value given by the model (the *prediction*), and the actual y value in the data (the *response*).

To compute this distance, need to turn the 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(8,2), sim1)
```

```
##  [1] 10 10 10 12 12 12 14 14 14 16 16 16 18 18 18 20 20 20 22 22
## [26] 26 26 28 28 28
```

Model Basics with *modelr*

```
> meansure_dist <- function(mod, data){
+   diff <- data$y -  model1(mod, data)
+   sqrt(mean(diff^2))
+ }
> meansure_dist(c(8,2), sim1)
```

```
## [1] 4.095278
```

Model Basics with *modelr*

Now if we compute the distance for all the models defined previously.

```
> sim1_dist <- function(a1,a2){
+   meansure_dist(c(a1,a2), sim1)
+ }
>
> models <- models %>%
+   mutate(dist = purrr::map2_dbl(a1,a2,sim1_dist))
```

Model Basics with *modelr*

```
> head(models)
```

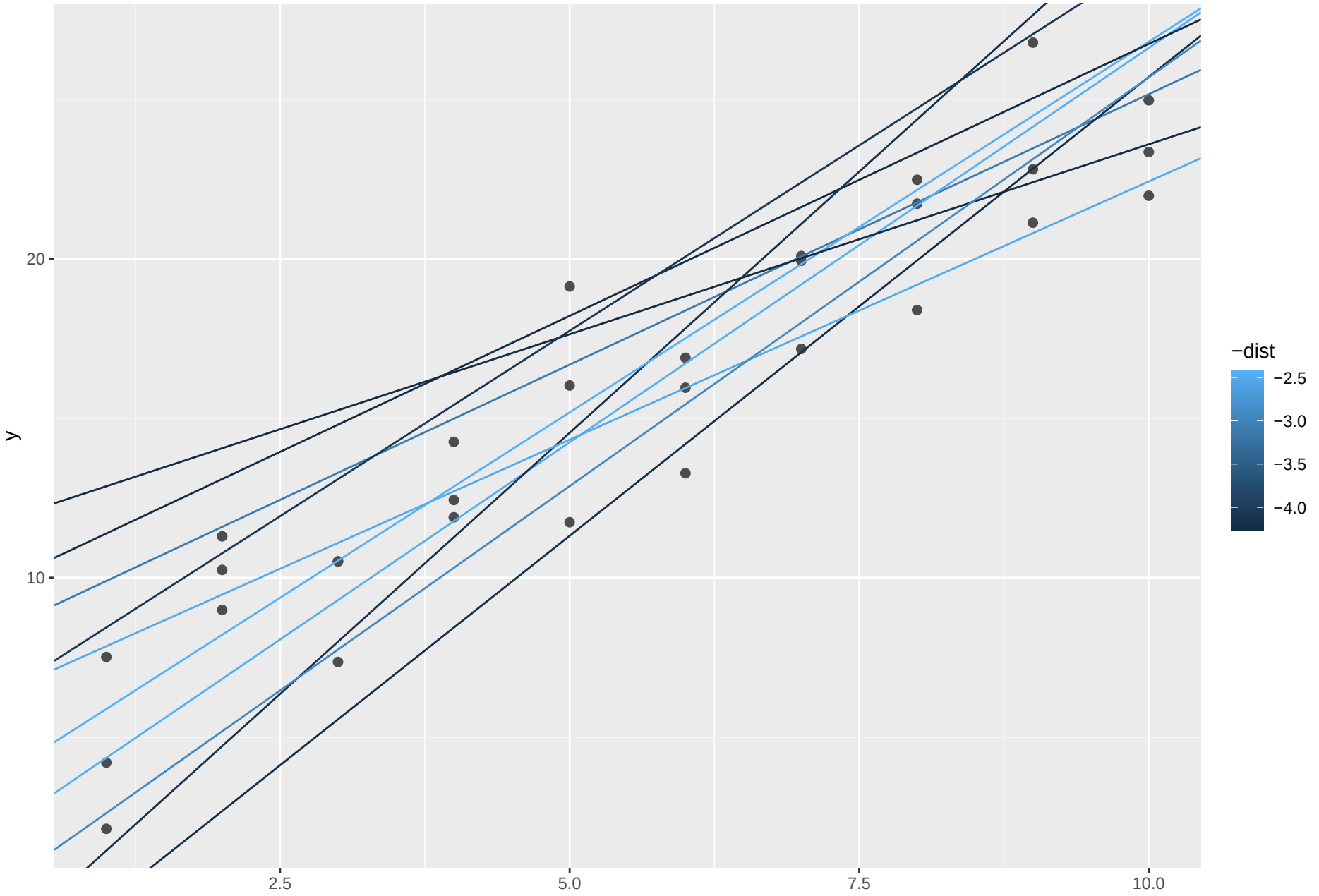
```
## # A tibble: 6 x 3
##      a1      a2  dist
##   <dbl>   <dbl> <dbl>
## 1 -19.8 -2.21    49.1
## 2 -13.5 -3.29    49.6
## 3 -19.7  3.59    16.2
## 4 -19.8  0.545   32.6
## 5 -15.1 -2.83    48.3
## 6 -18.2  4.34    12.0
```

Model Basics with *modelr*

Let's overlay the 10 best models on the the data, and visualize it. The ones with the smallest distance models get the brightest colors:

```
> ggplot(sim1, aes(x,y)) +
+   geom_point(size = 2, color = "grey30") +
+   geom_abline(
+     aes(intercept = a1, slope = a2, color = -dist),
+     data = filter(models, rank(dist) <= 10)
+   )
```

Model Basics with *modelr*

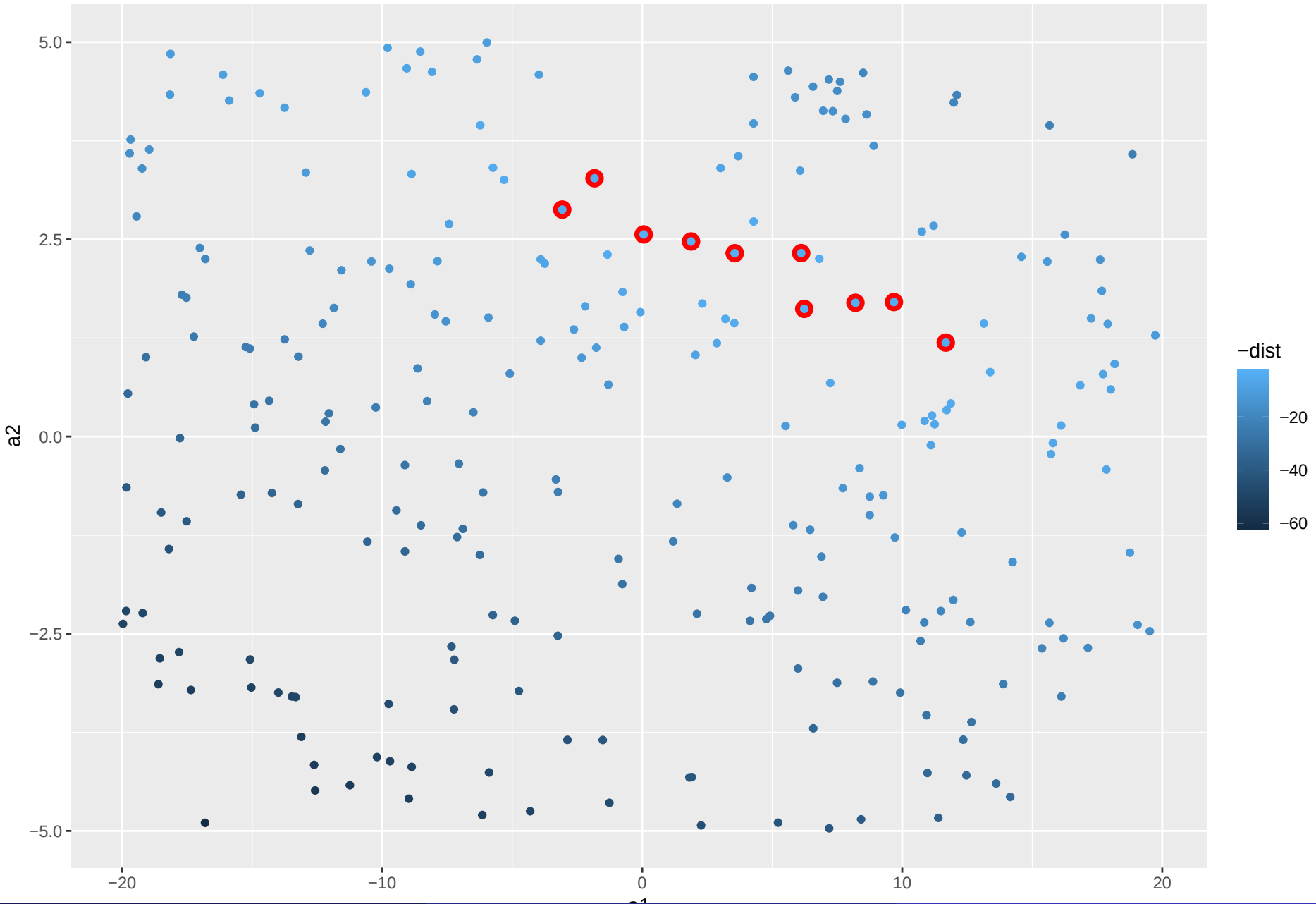


Model Basics with *modelr*

We can also think about these models as observations and visualize them with a scatterplot of a1 vs a2.

```
> ggplot(models, aes(a1,a2)) +
+   geom_point(
+     data = filter(models, rank(dist) <= 10),
+     size = 4, color = "red"
+   ) +
+   geom_point(
+     aes(color = -dist)
+   )
```

Model Basics with *modelr*



Model Basics with *modelr*

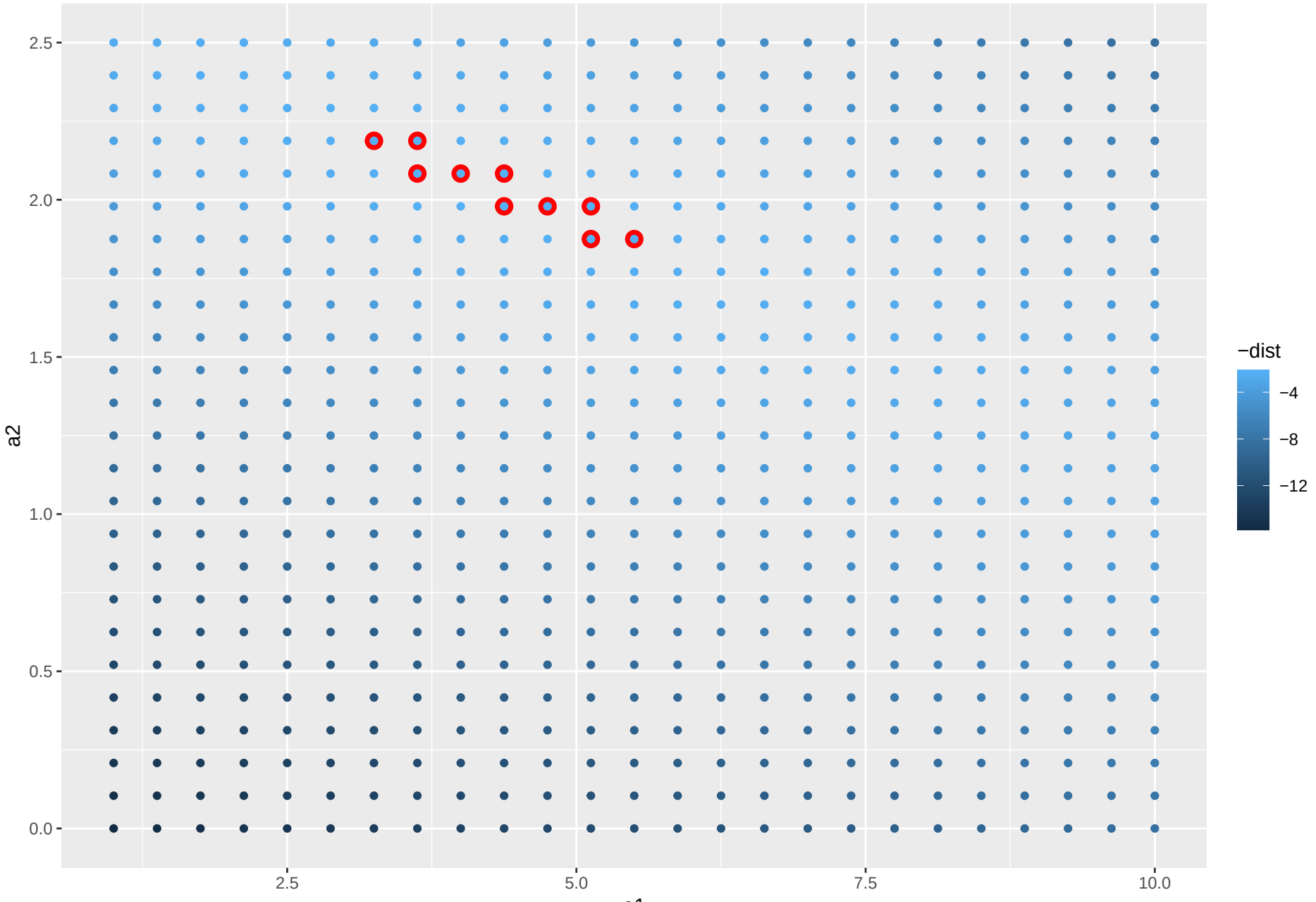
Instead of trying lots of random models, we could be more systematic and generate an evenly spaced grid of points (grid search).

```
> grid <- expand.grid(
+   a1 = seq(1,10, length = 25),
+   a2 = seq(0,2.5, length = 25)) %>%
+   mutate(dist = purrr::map2_dbl(a1,a2,sim1_dist))
```

Model Basics with *modelr*

```
> grid %>%
+   ggplot(aes(a1,a2))+
+   geom_point(
+     data = filter(grid, rank(dist) <=10),
+     size = 4, color = "red"
+   ) +
+   geom_point(aes(color = -dist))
```

Model Basics with *modelr*

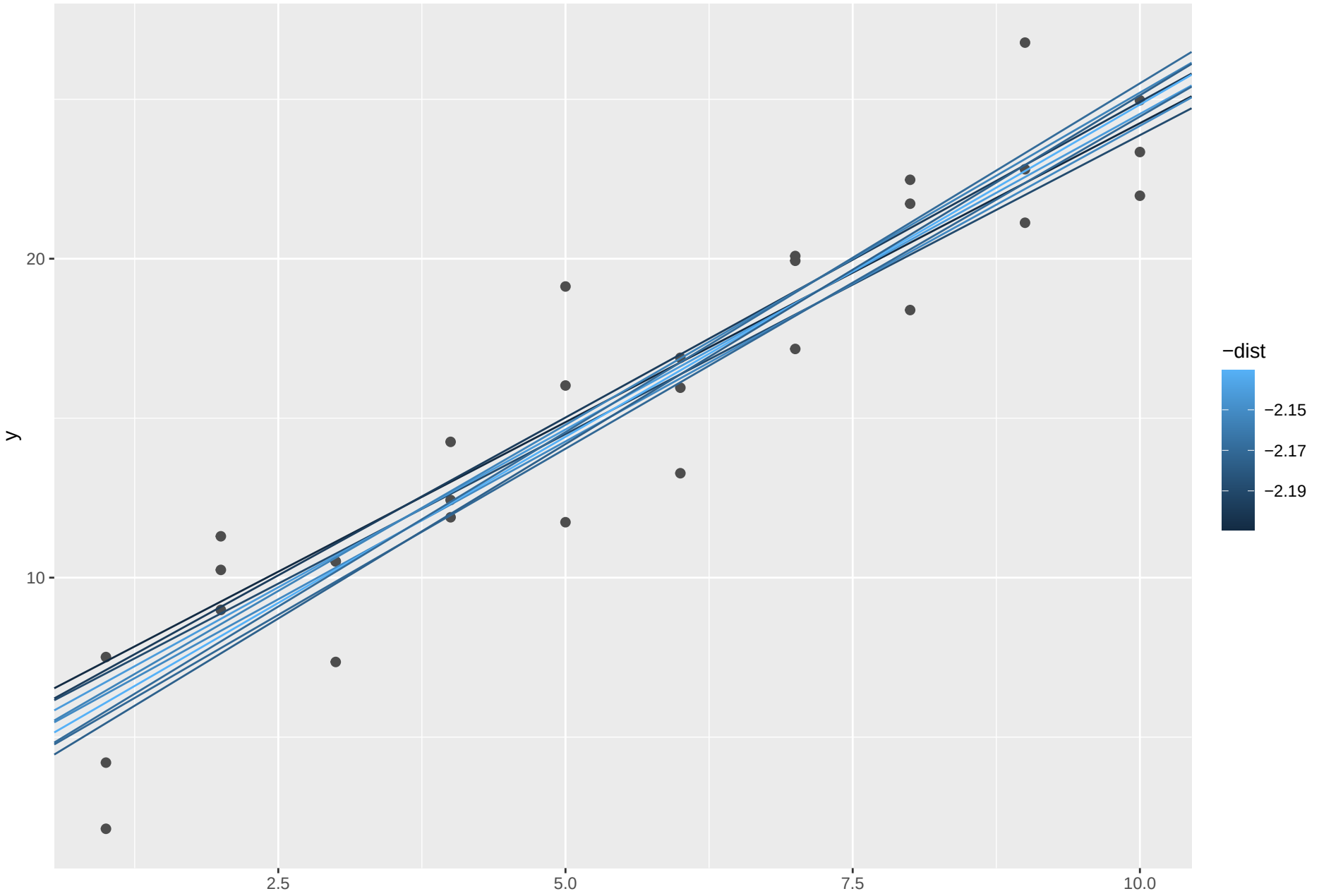


Model Basics with *modelr*

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, color = "grey30") +
+   geom_abline(
+     aes(intercept = a1, slope = a2, color = -dist),
+     data = filter(grid, rank(dist) <= 10)
+   )
```

Model Basics with *modelr*



Model Basics with *modelr*

You could imagine iteratively making the grid finer and finer until you narrowed in on the best model.

But there are other numerical minimization tools to solve the problem. You can pick a starting point and look around for the steepest slope, then ski down that slope a little way, then repeat again and again, until no way down further. In R, we can do that with *optim()*.

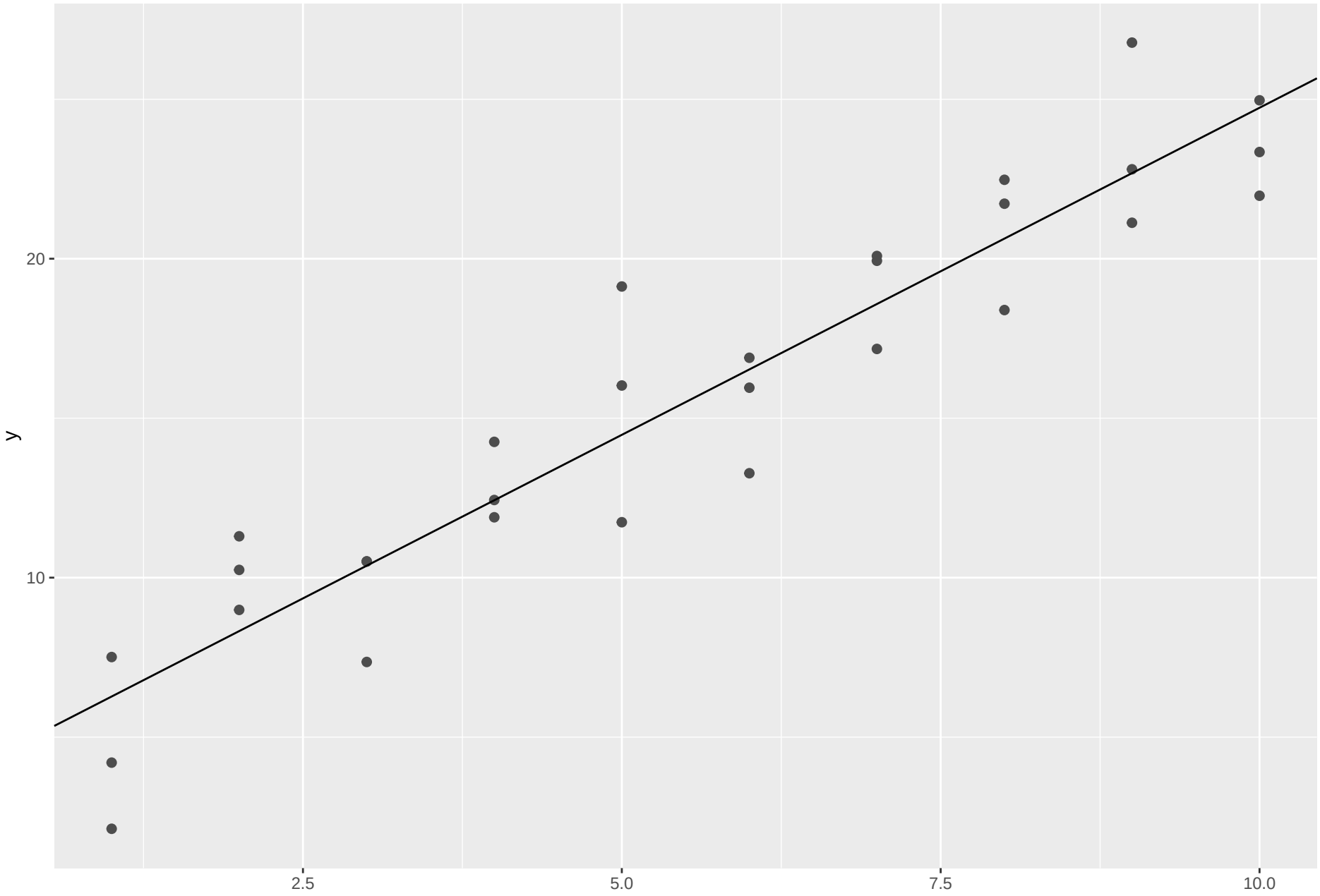
```
> best_guess <- optim(c(0,0), meansure_dist, data = sim1)
> best_guess$par
```

```
## [1] 4.222248 2.051204
```

Model Basics with *modelr*

```
> ggplot(sim1,aes(x,y))+
+   geom_point(size = 2, color ="grey30") +
+ geom_abline(intercept = best_guess$par[1],
+              slope = best_guess$par[2])
```

Model Basics with *modelr*



Model Basics with *modelr*

Linear model with *lm()*

```
> sim1_mod <- lm(y ~x , data = sim1)
> coef(sim1_mod)
```

```
## (Intercept)           x
##    4.220822    2.051533
```

Behind the scenes *lm()* doesn't use *optim()*, but instead takes advantage of the mathematic structure of linear models. *lm()* actually finds the closest model in a single step, the approach is faster and it guarantees that there is a global minimum.

Mode Building

Previously, we have focused on simulated datasets to help you learn about how models work, now we will focus on real data, how you can progresssively build up a model to help you better understand the data.

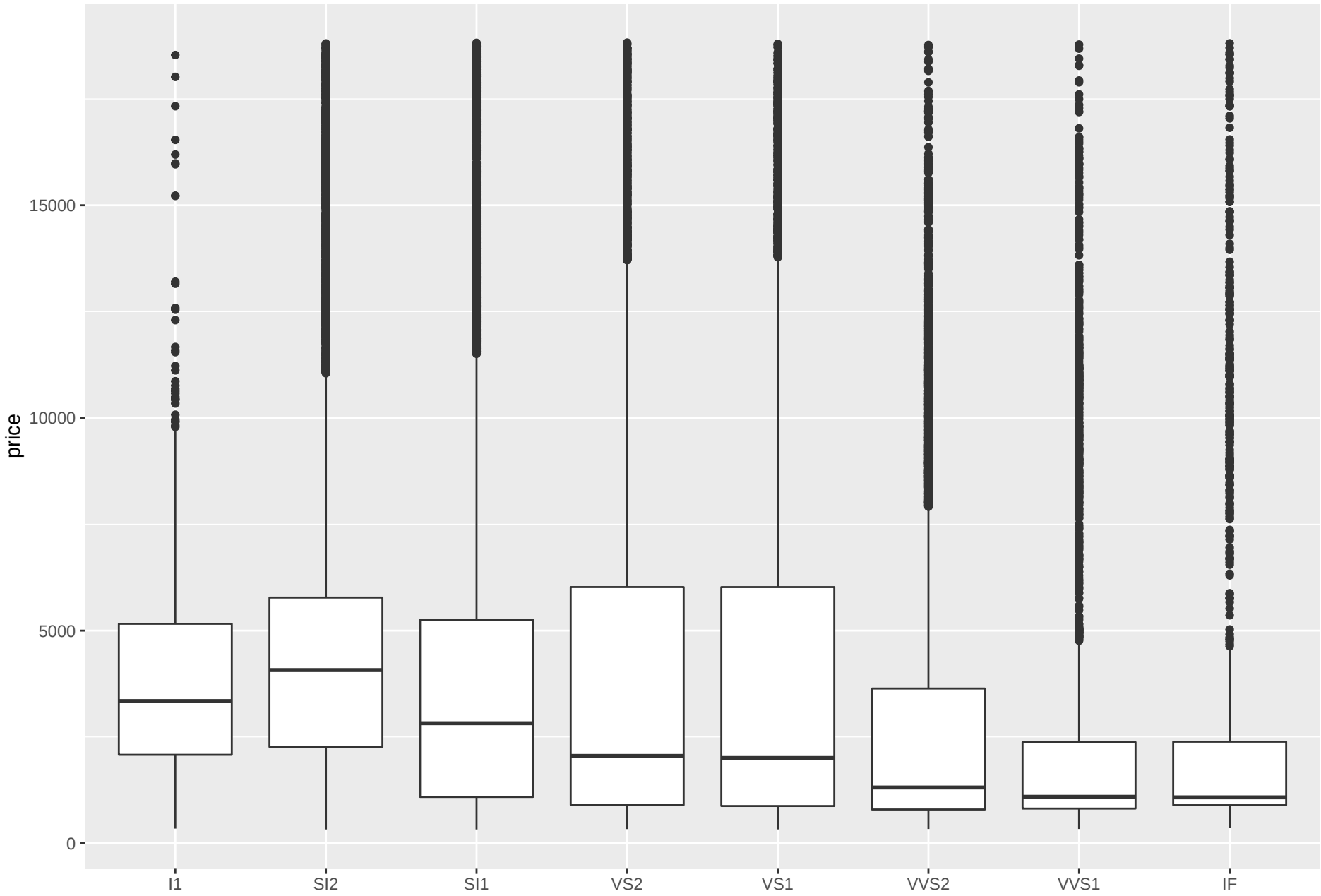
We will use the *diamonds* data set to investigate the relationship between diamonds' quality and it's price.

Mode Building

Firstly, visualize the relationship

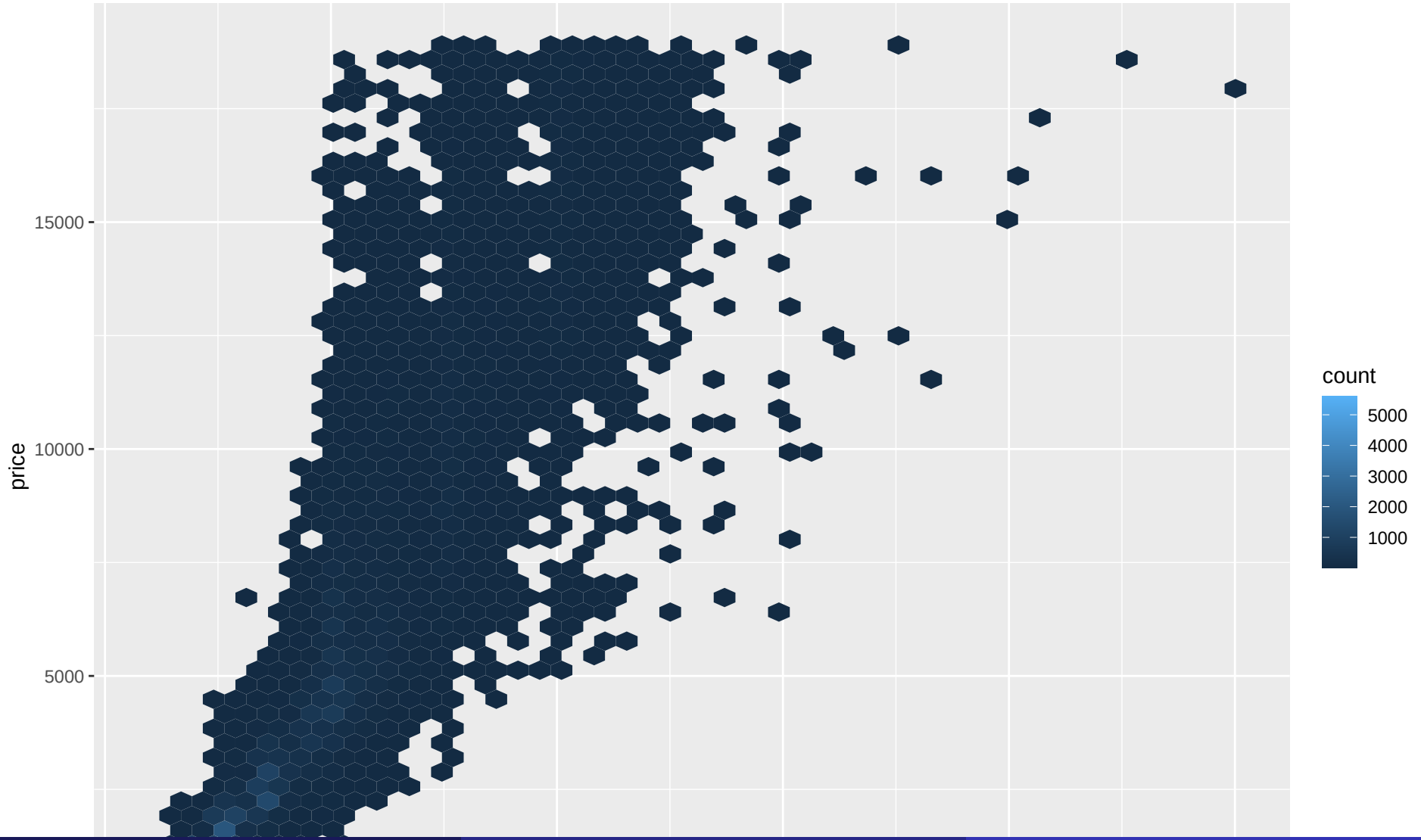
```
> ggplot(diamonds, aes(clarity, price)) +
+ geom_boxplot()
```

Mode Building



Mode Building

It looks like lower-quality diamonds have higher prices because there is an important confounding variable: the weight (*carat*) of the diamond.

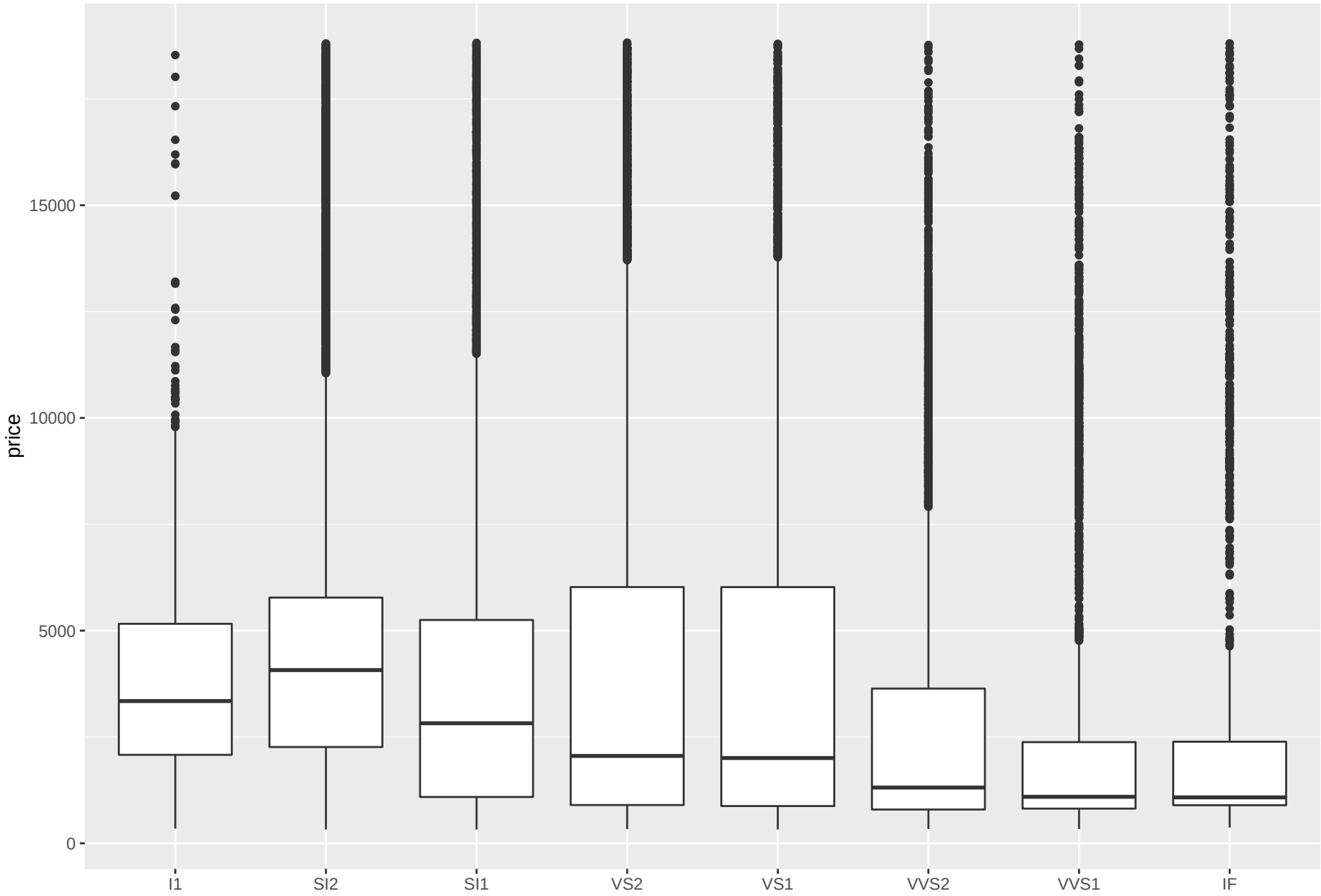


Mode Building

The weight of the diamond is the single most important factor for determining the price of the diamond, and lower quality diamonds tent to be larger.

```
> ggplot(diamonds, aes(clarity, price)) +
+ geom_boxplot()
```

Mode Building



Mode Building

We need to remove the "carat effect" to see how the *clarity* or other attributes of a diamond affect its relative price.

Firstly, we fit a linear model with *price* to *carat* to get the residuals, or get the separate out the effect of *carat*.

```
> mod_diamond <- lm(price ~ carat, data = diamonds)
>
> diamonds2 <- diamonds %>%
+   add_residuals(mod_diamond, "resid")
```

Mode Building