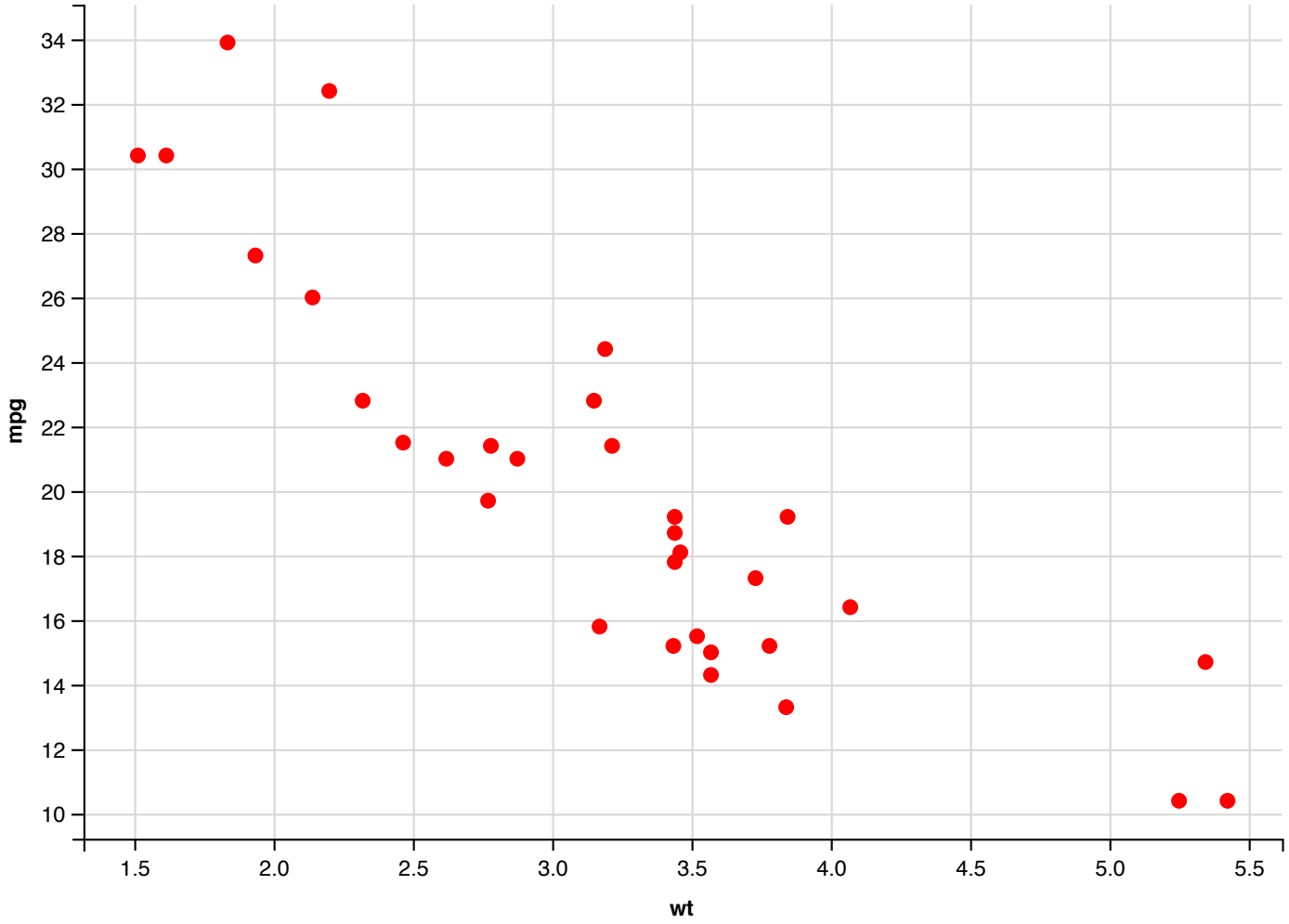
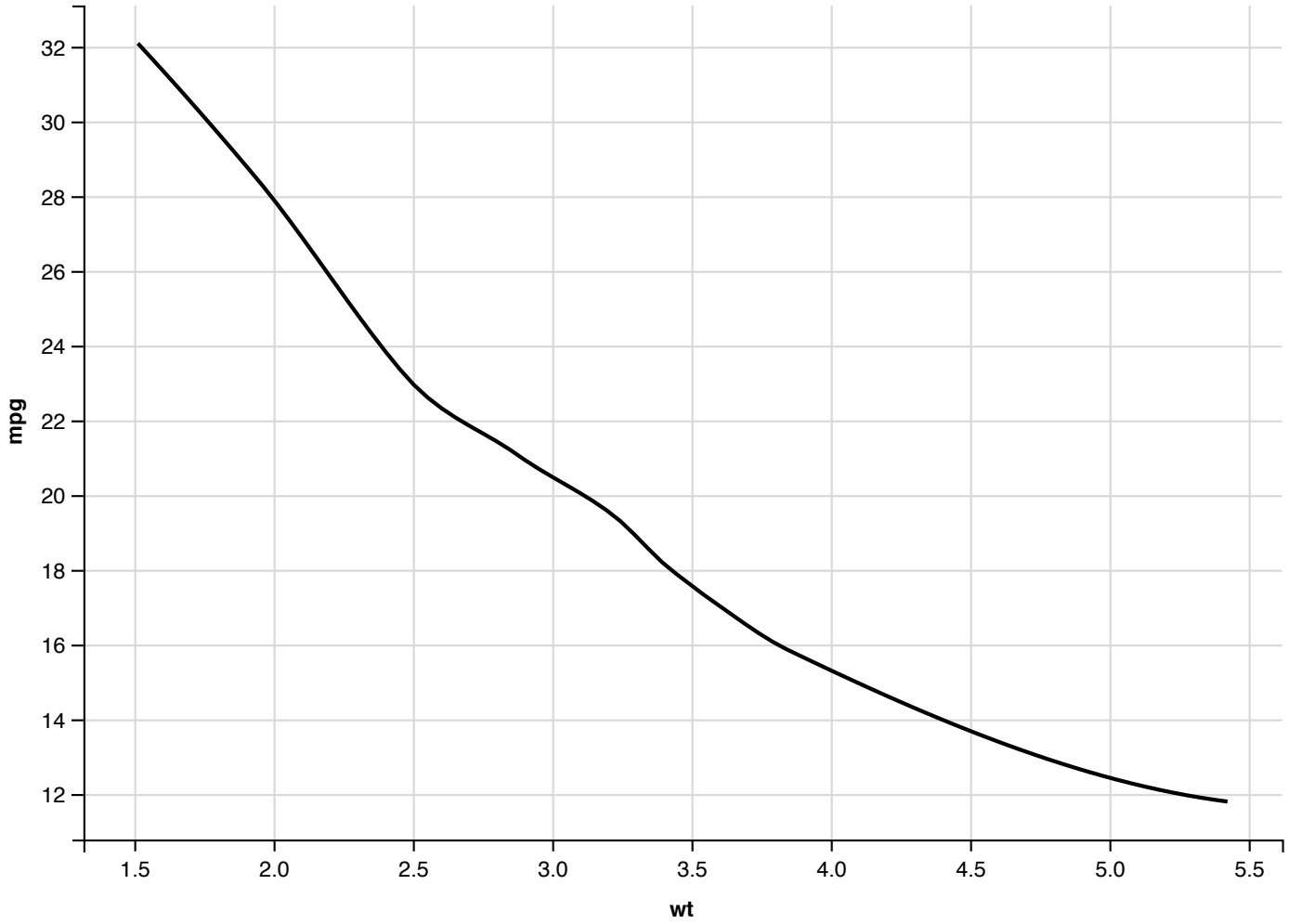
Datacamp GGVIS tutorial

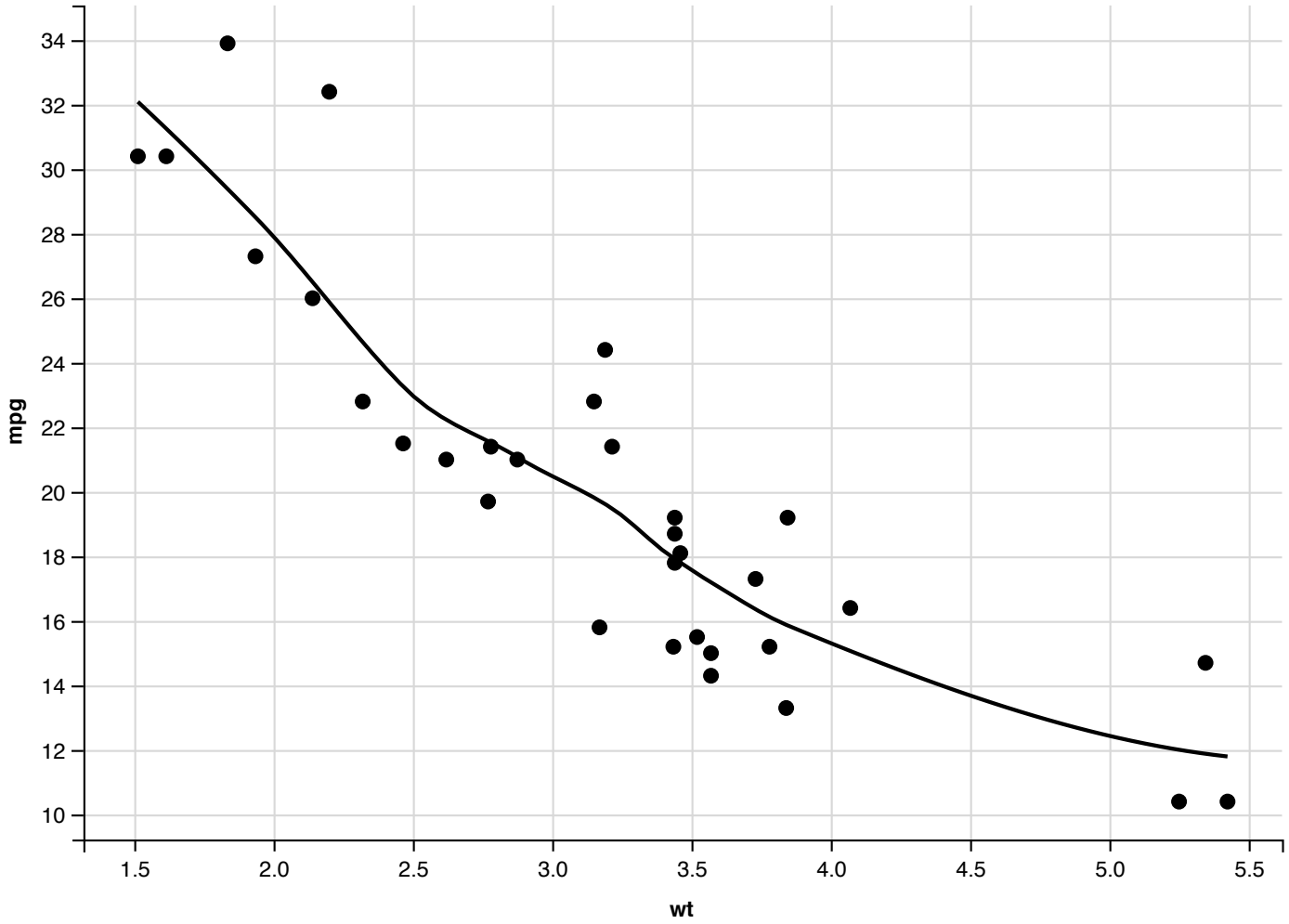
```
library(ggvis)
# Change the code below to make a graph with red points
mtcars %>% ggvis(~wt, ~mpg, fill := "red") %>% layer_points()
```




```
# Change the code below draw smooths instead of points
mtcars %>% ggvis(~wt, ~mpg) %>% layer_smooths()
```

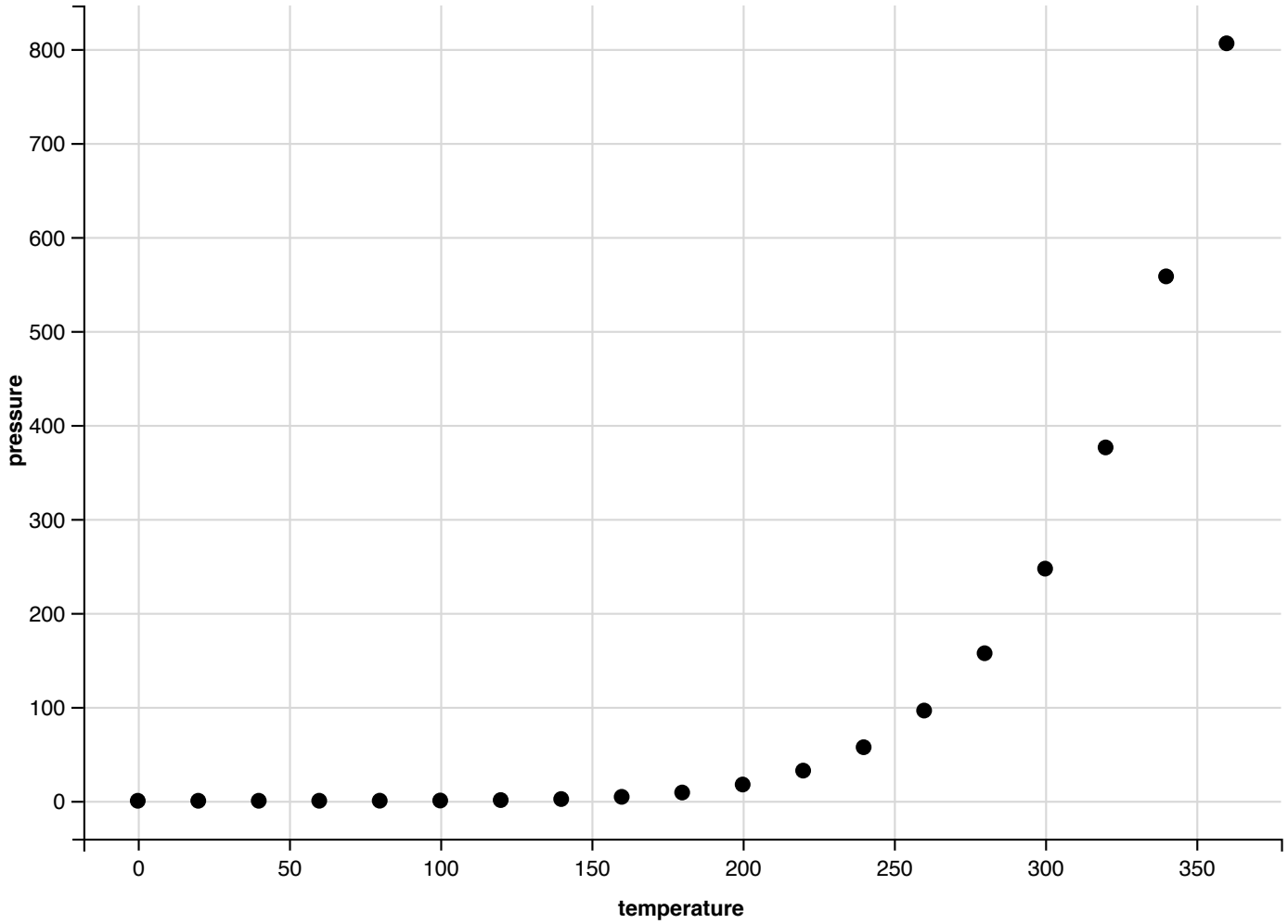



```
# Change the code below to make a graph containing both points and a smoothed summary line
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points() %>% layer_smooths()
```

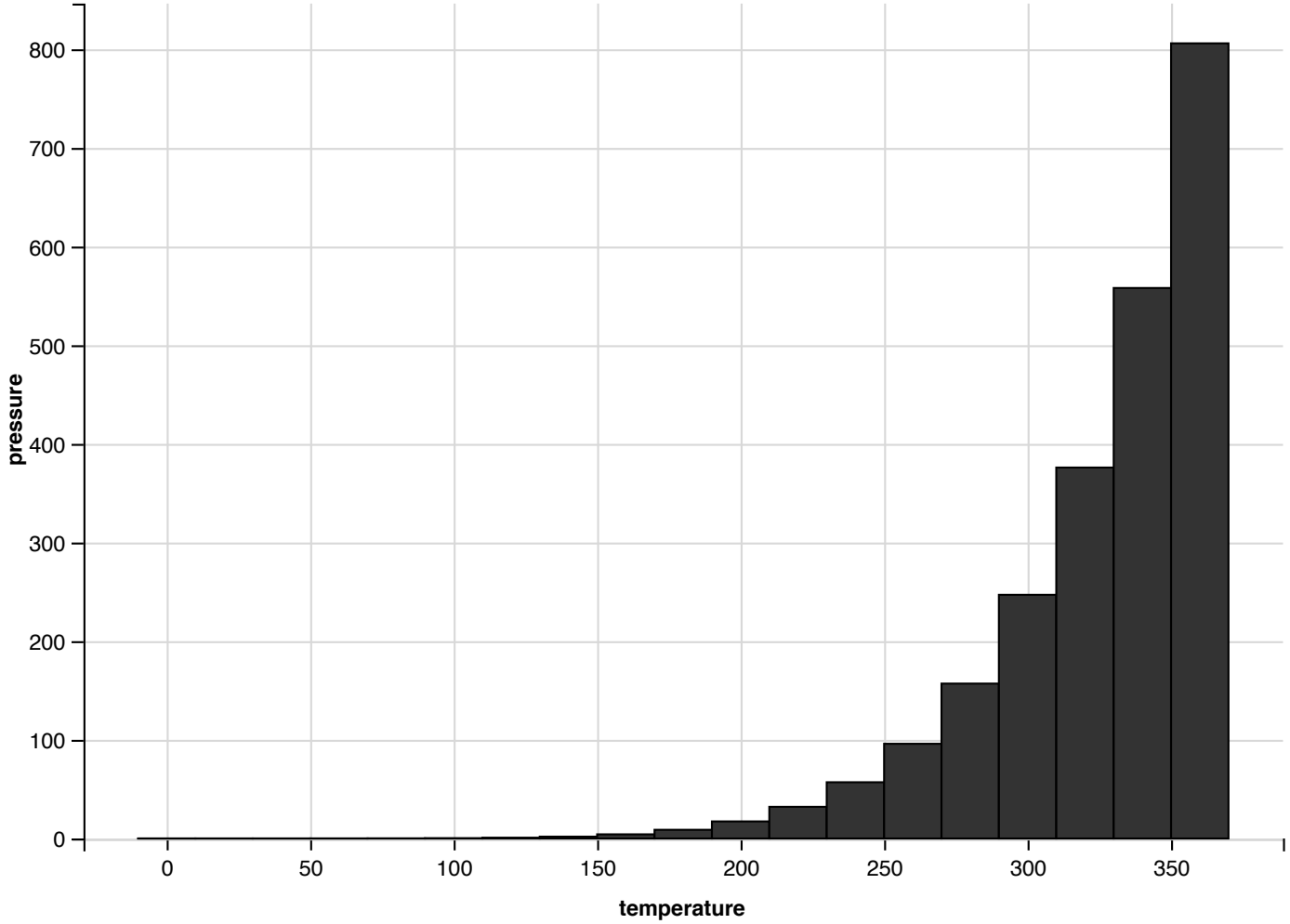


SCATTERPLOT

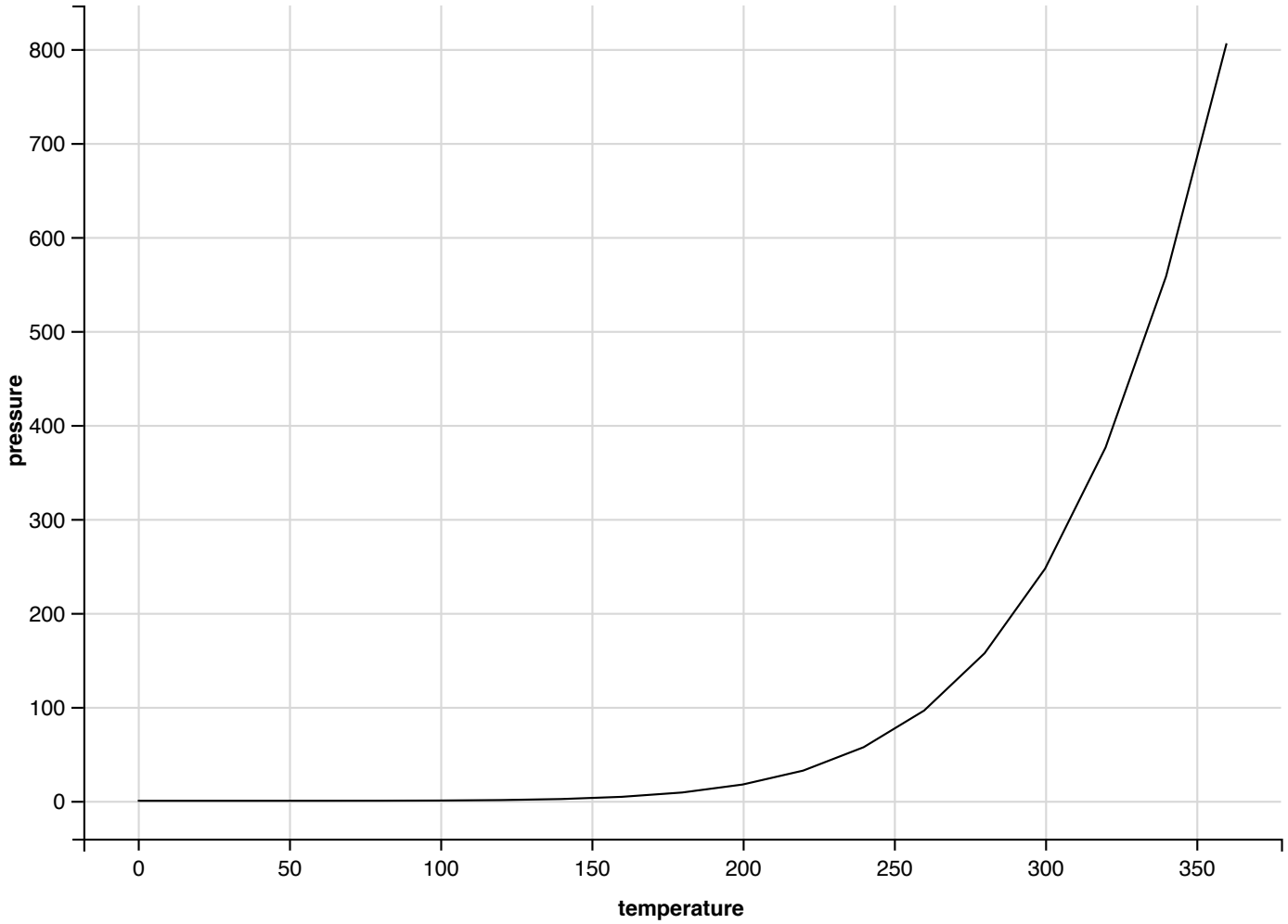
```
# Make a scatterplot of the pressure dataset
pressure %>% ggvis(~temperature, ~pressure) %>% layer_points()
```



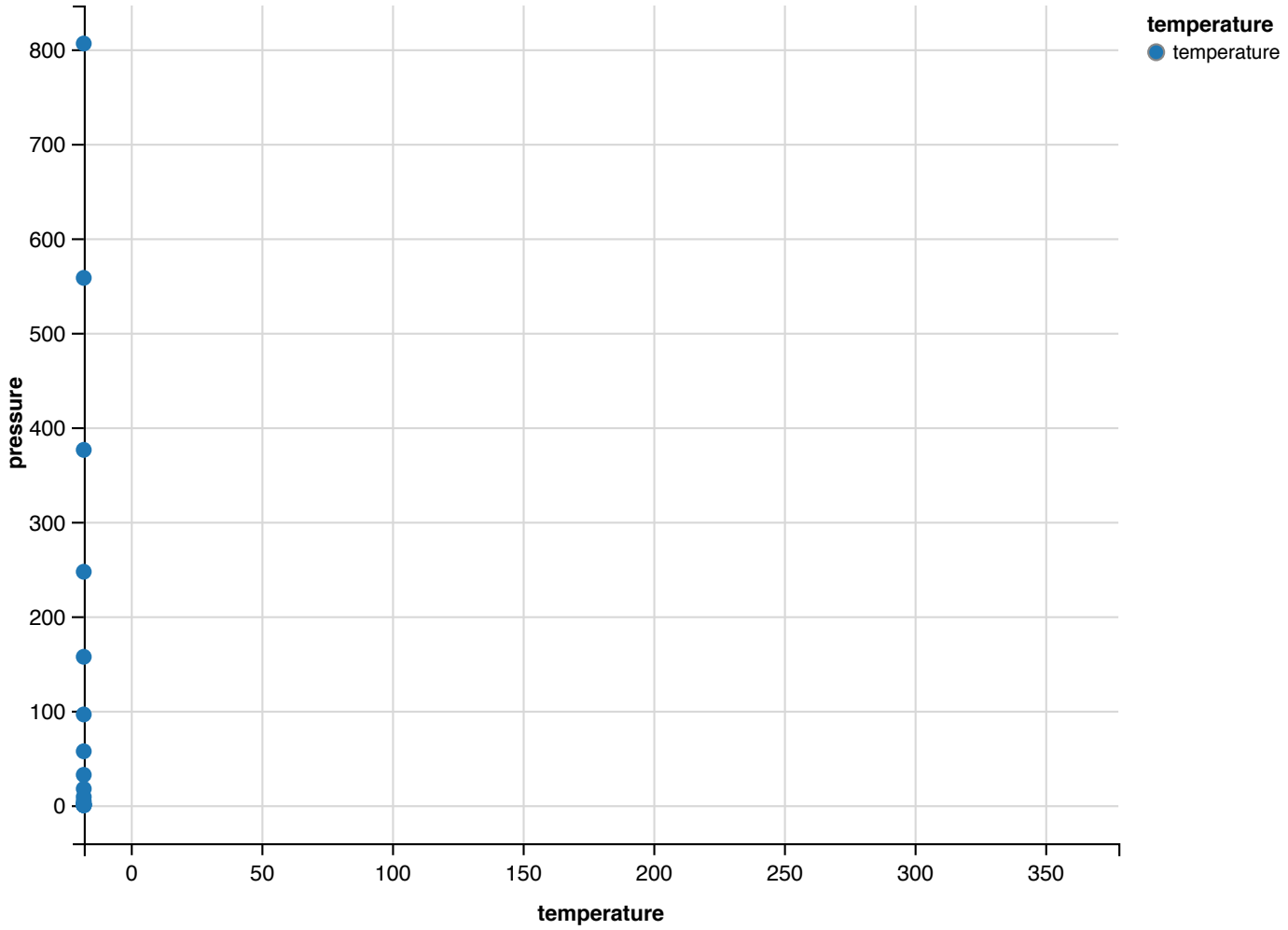

```
# Adapt the code you wrote for the first challenge: show bars instead of points
pressure %>% ggvis(~temperature, ~pressure) %>% layer_bars()
```




```
# Adapt the code you wrote for the first challenge: show lines instead of points
pressure %>% ggvis(~temperature, ~pressure) %>% layer_lines()
```

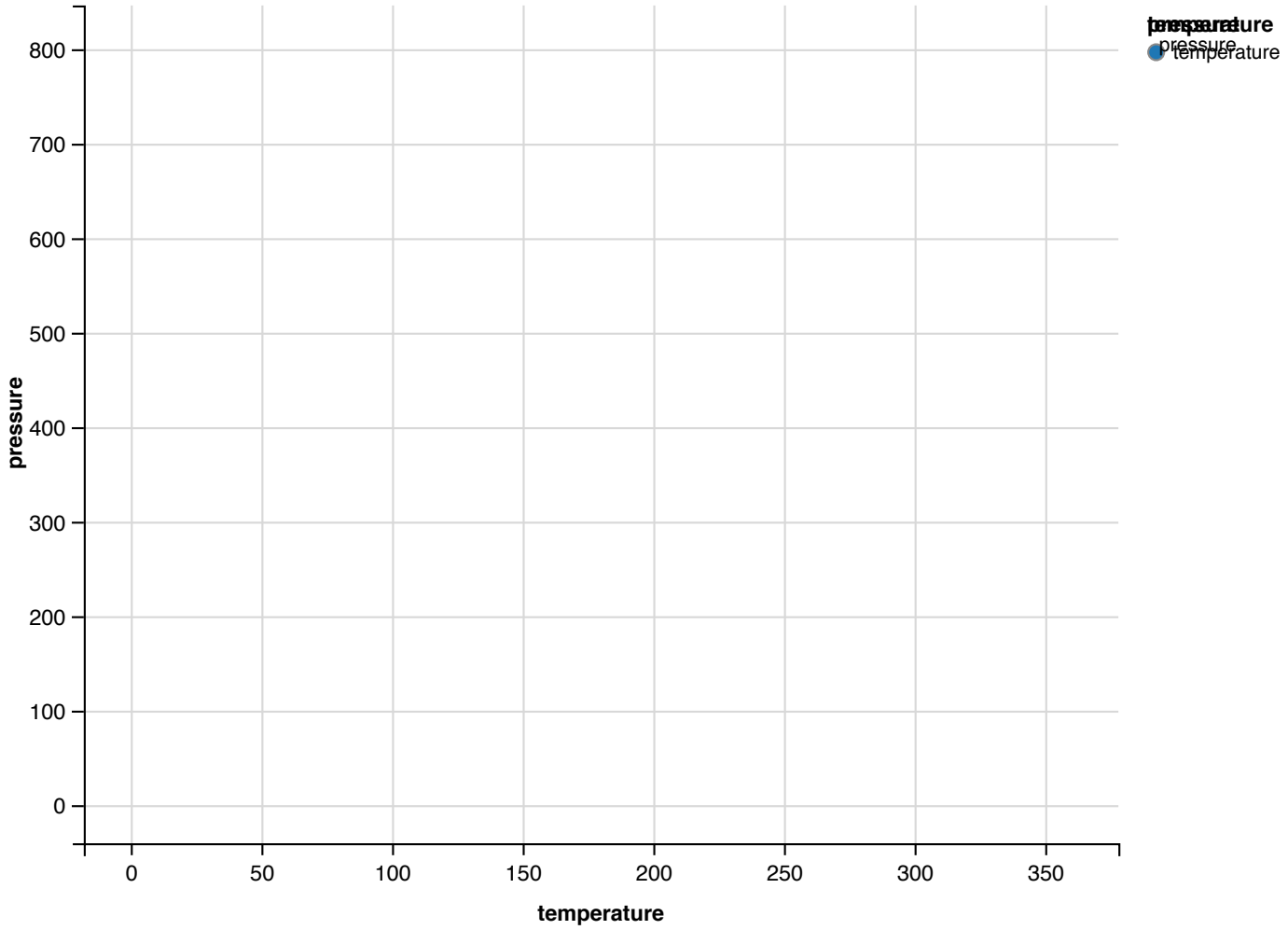



```
# Adapt the code you wrote for the first challenge: map the fill property to the temperatur
e variable
pressure %>% ggvis(~temperature, ~pressure, fill = ~ "temperature") %>% layer_points()
```

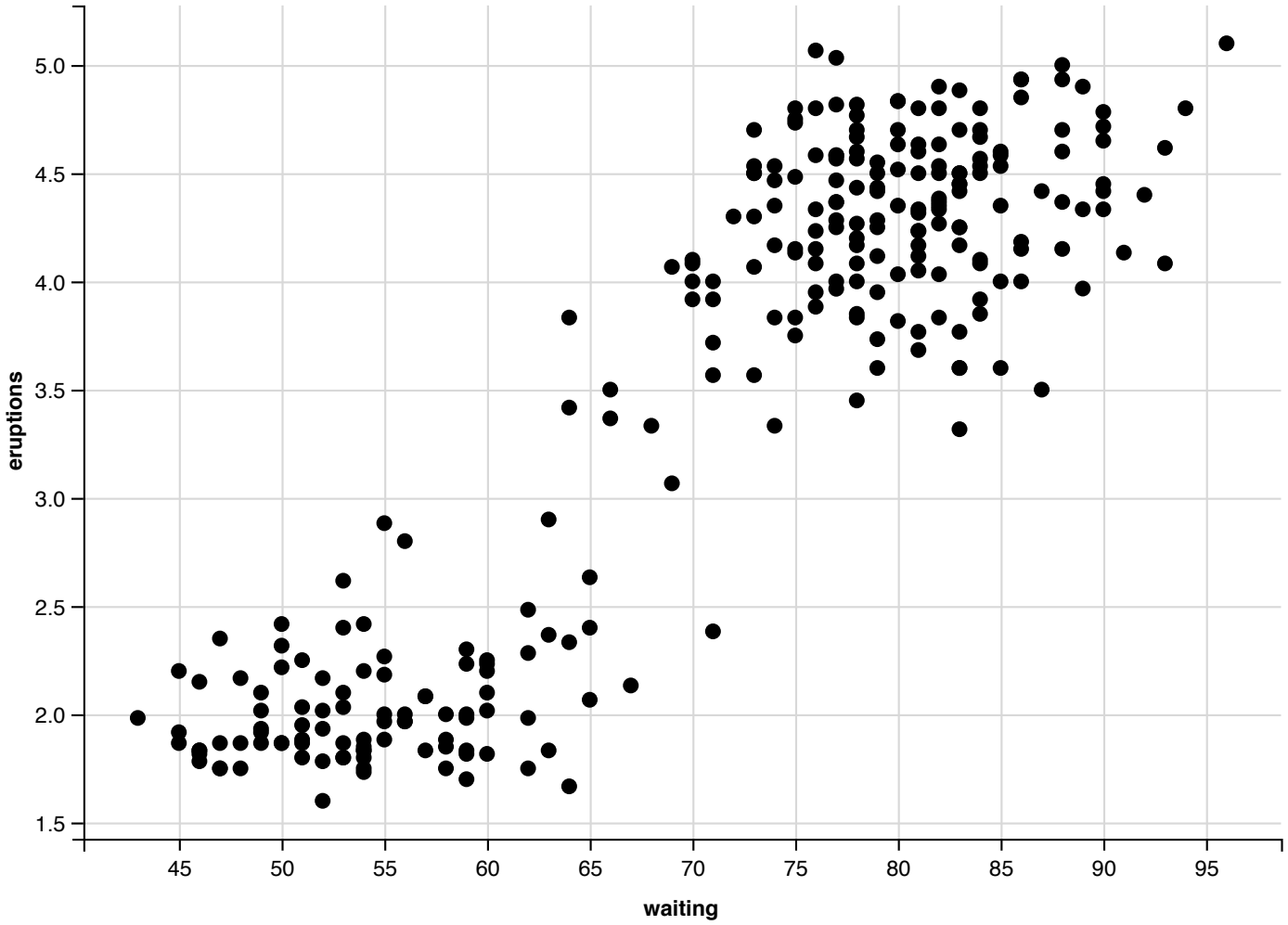



```
# Extend the code you wrote for the previous challenge: map the size property to the pressu
re variable
pressure %>% ggvis(~temperature, ~pressure, fill =~ "temperature", size =~ "pressure") %>%
layer_points()
```

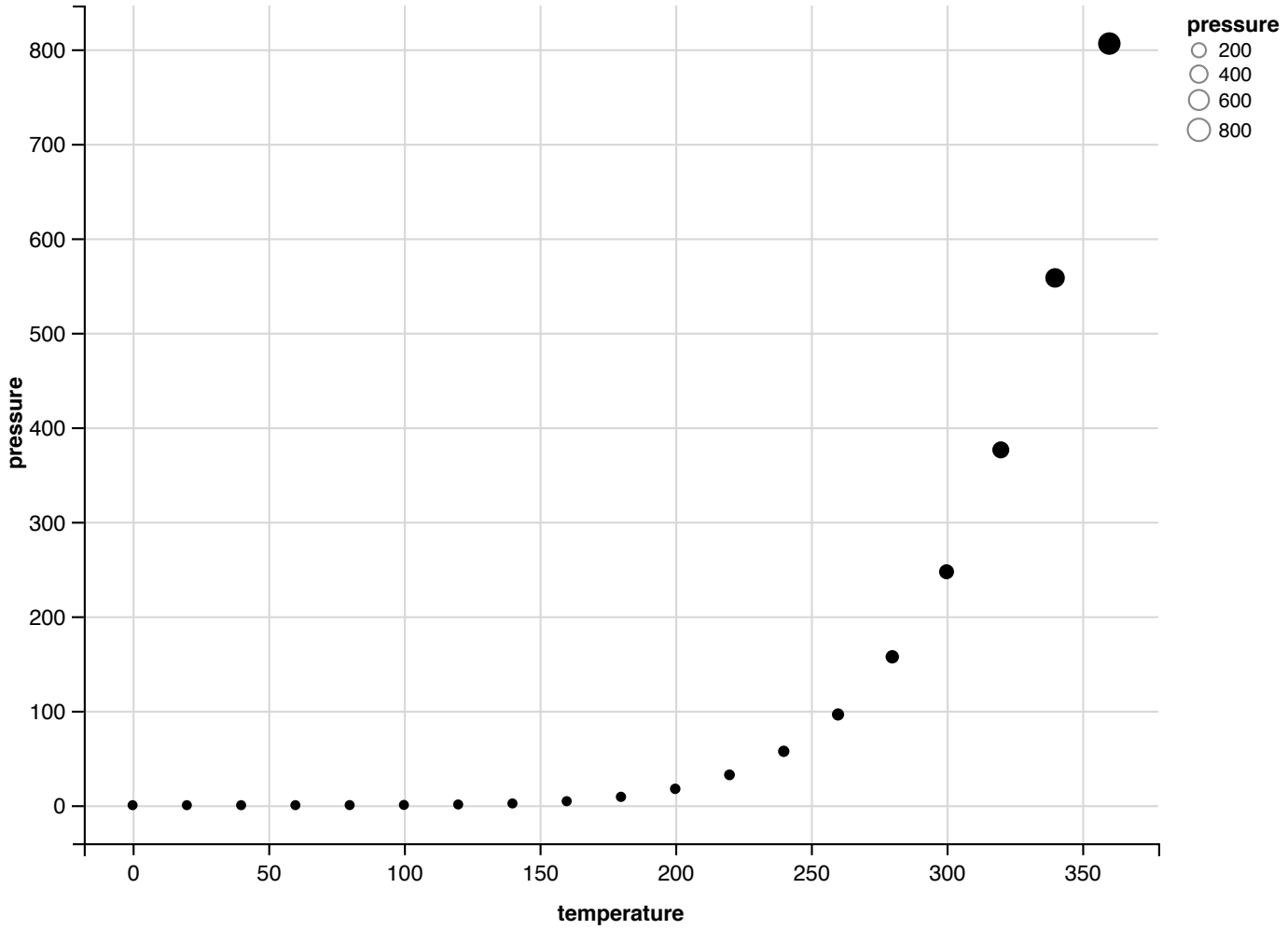
```
## Warning: Scale 'size' for property 'size' is missing a range. ggvis tries
## to automatically provide ranges for scales, but it doesn't know how in
## this case. You must specify the range manually. See ?scale_nominal or ?
## scale_numeric for more information and examples.
```



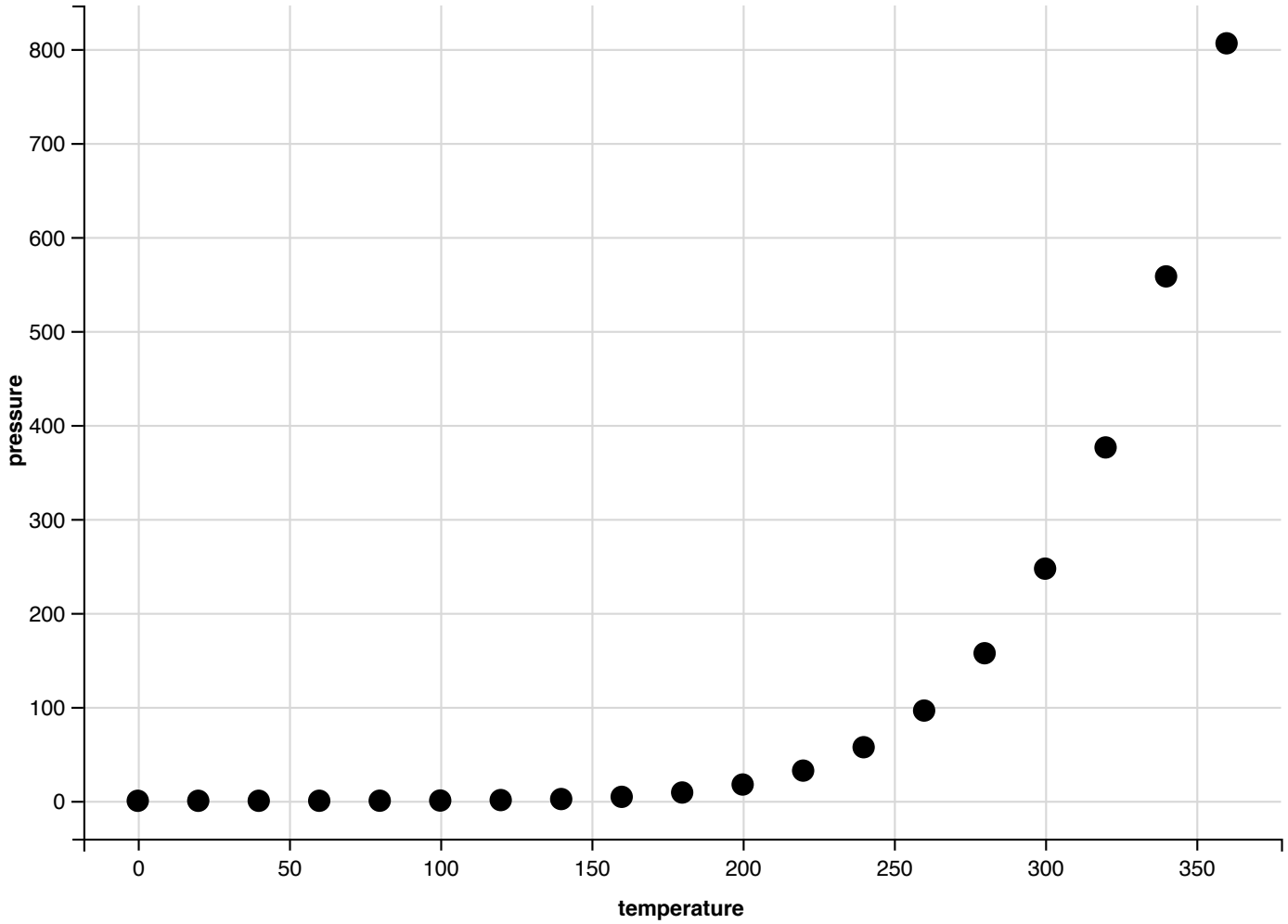

```
# Rewrite the code with the pipe operator
faithful %>%
  ggvis(~waiting, ~eruptions) %>%
  layer_points()
```



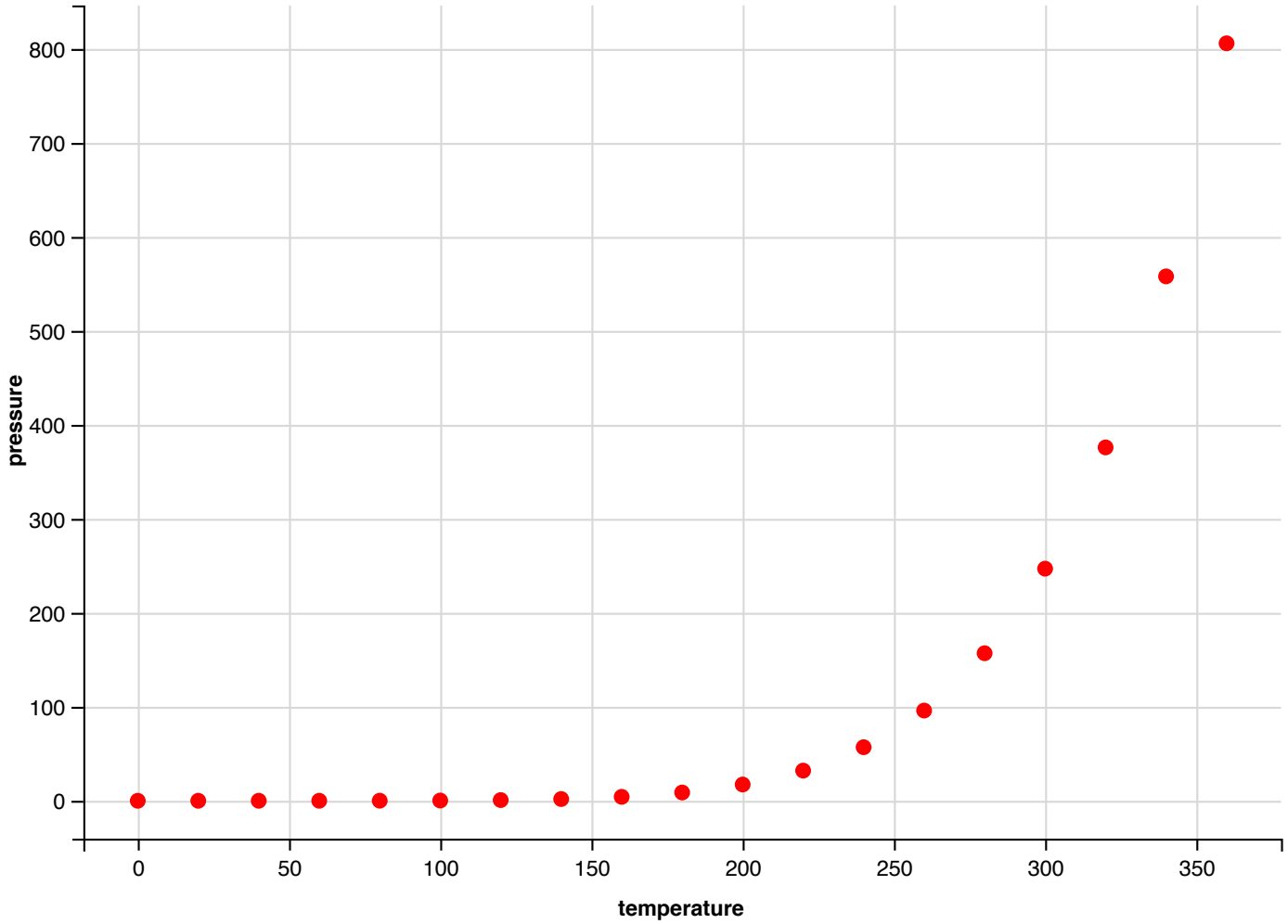

```
# Modify this graph to map the size property to the pressure variable
pressure %>% ggvis(~temperature, ~pressure, size = ~pressure) %>% layer_points()
```




```
# Modify this graph by setting the size property
pressure %>% ggvis(~temperature, ~pressure, size := 100) %>% layer_points()
```




```
# Fix this code to set the fill property to red
pressure %>% ggvis(~temperature, ~pressure, fill := "red") %>% layer_points()
```

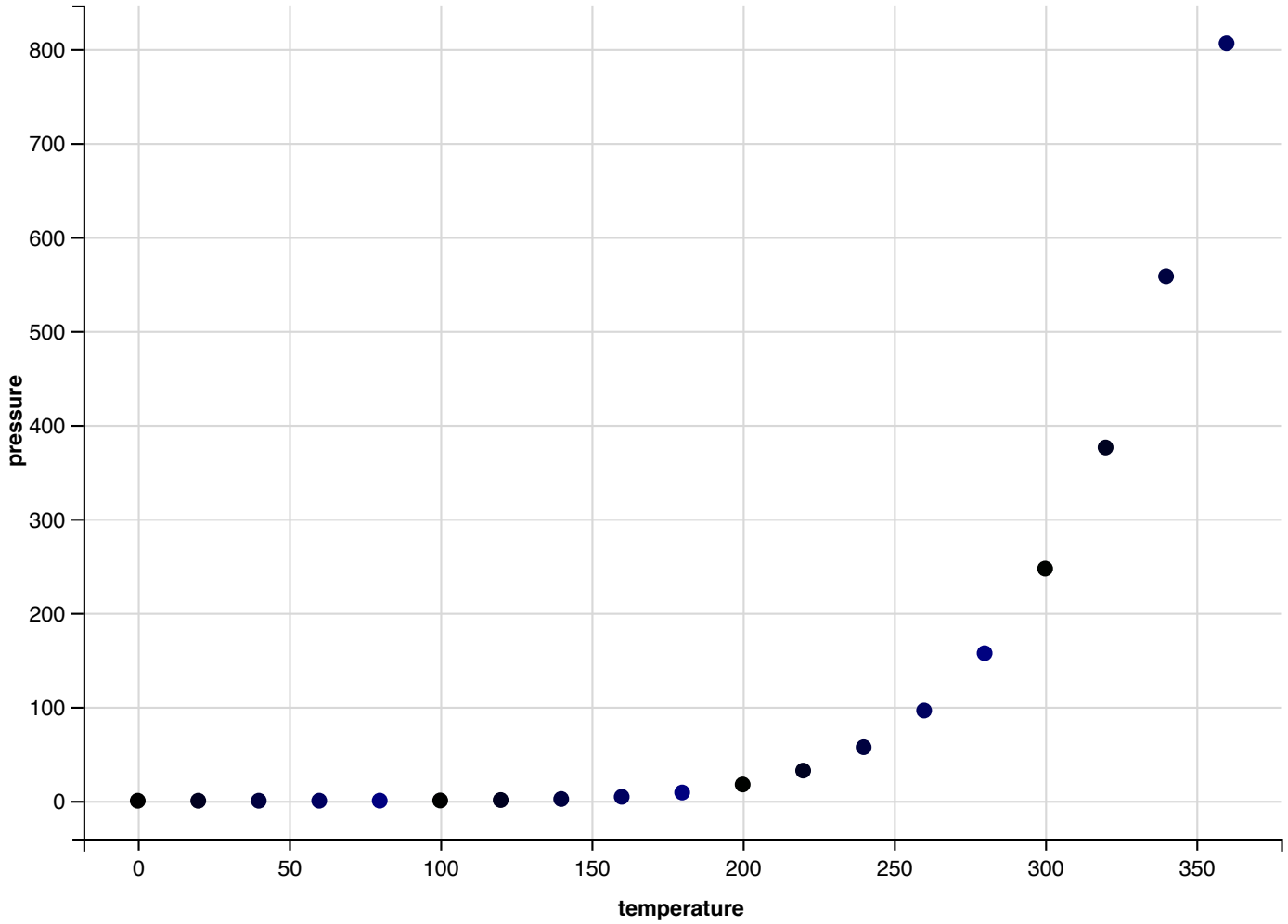


PROPERTIES FOR POINTS

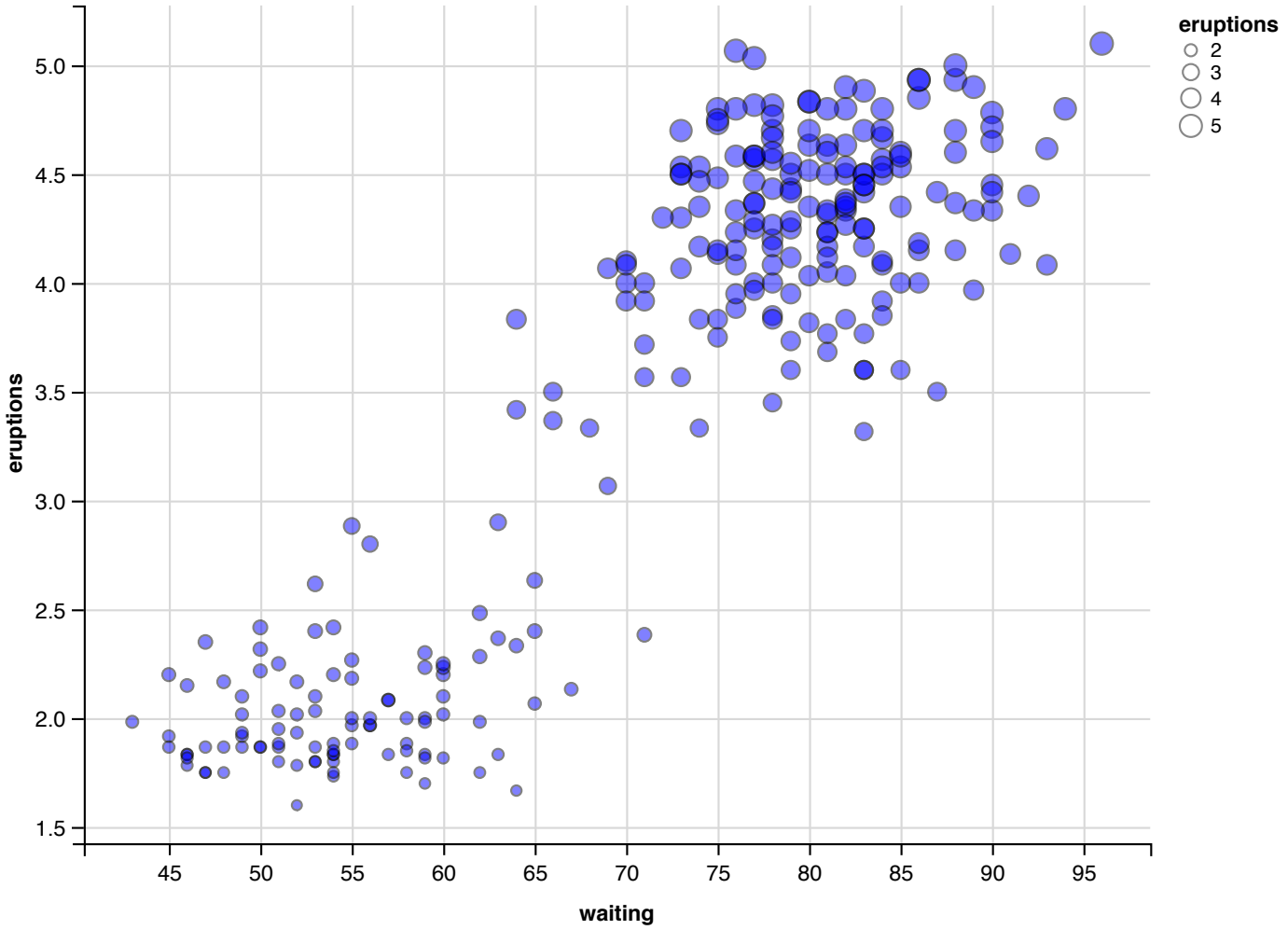
You can manipulate many different properties when using the points mark, including x, y, fill, fillOpacity, opacity, shape, size, stroke, strokeOpacity, andstrokeWidth.

The shape property in turn recognizes several different values: circle (default), square, cross, diamond, triangle-up, and triangle-down.

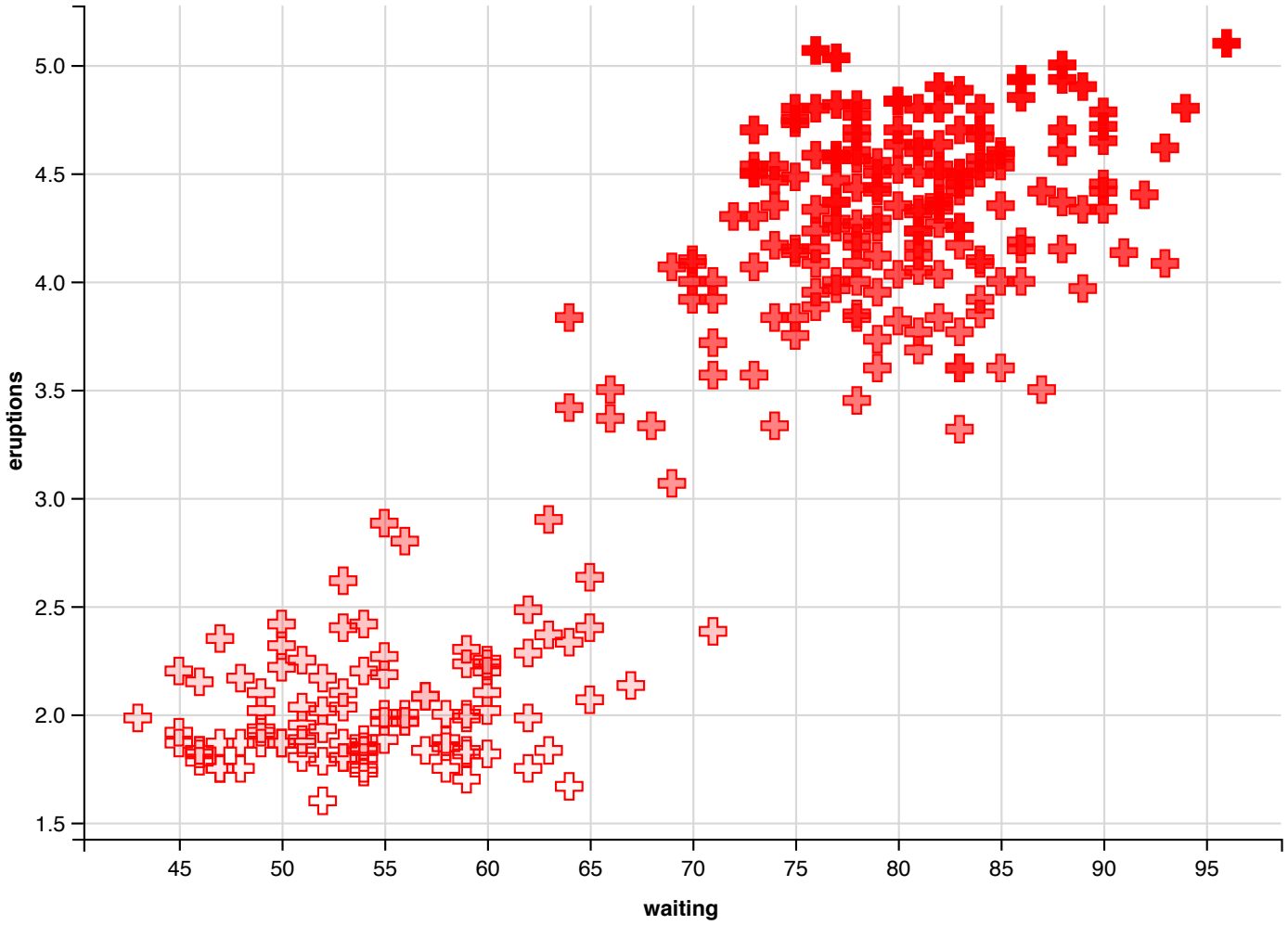
```
# Change the code to set the fills using pressure$black. pressure$black is loaded into work space.
black <- "black"
pressure$black <- pressure$temperature
pressure %>%
  ggvis(~temperature, ~pressure,
        fill := ~black) %>%
  layer_points()
```




```
# Plot the faithful data as described in the second instruction
faithful %>%
  ggvis(~waiting, ~eruptions,
        size = ~eruptions, opacity := 0.5,
        fill := "blue", stroke := "black") %>%
  layer_points()
```




```
# Plot the faithful data as described in the third instruction
faithful %>%
  ggvis(~waiting, ~eruptions,
        fillOpacity = ~eruptions, size := 100,
        fill := "red", stroke := "red", shape := "cross") %>%
  layer_points()
```

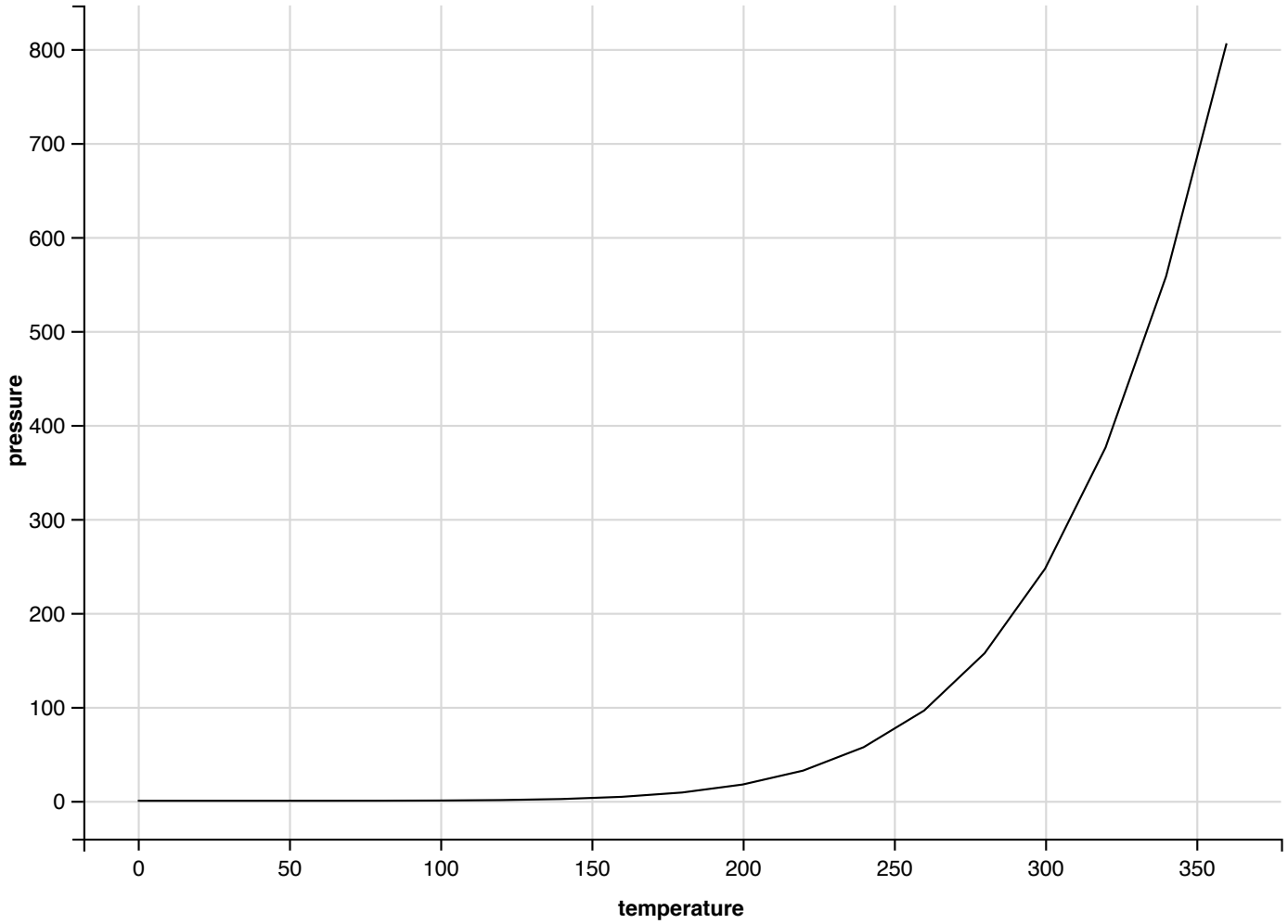


PROPERTIES FOR LINES

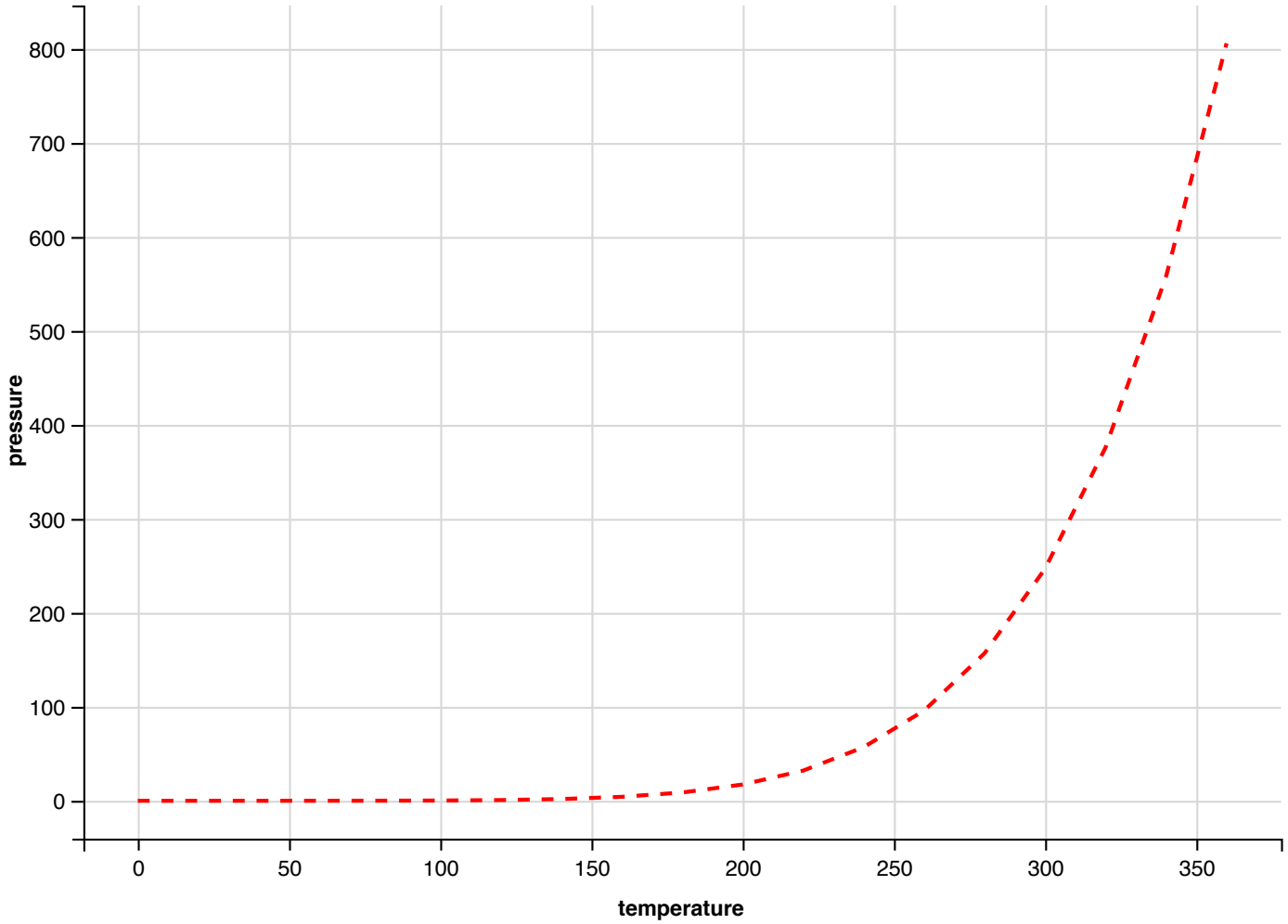
This mark type responds to, among others, x, y, fill, fillOpacity, opacity, shape, size, stroke, strokeOpacity, andstrokeWidth.

Similar to points, lines have specific properties; they respond to: x, y, fill, fillOpacity, opacity, stroke, strokeDash, strokeOpacity, and strokeWidth. As you can see, most of them are common to the properties for points, some are missing (e.g., no size property) and others are new (e.g., strokeDash).

```
# Change the code below to use the lines mark
pressure %>% ggvis(~temperature, ~pressure) %>% layer_lines()
```




```
# Set the properties described in the second instruction in the graph below
pressure %>%
  ggvis(~temperature, ~pressure,
        stroke := "red", strokeWidth := 2, strokeDash := 6) %>%
  layer_lines()
```



###DISPLAYING MODEL FITS

compute_model_prediction() is a useful function to use with line graphs. It takes a data frame as input and returns a new data frame as output. The new data frame will contain the x and y values of a line fitted to the data in the original data frame.

For example, the code below computes a line that shows the relationship between the eruptions and waiting variables of the faithful data set.

Notice that compute_model_prediction() takes a couple of arguments. First we use the pipe operator to pass it the data set faithful. Then we provide an R formula, eruptions ~ waiting. An R formula contains two variables connected by a tilde, ~. compute_model_prediction() will use the variable on the left as the y variable for the line, and it will use the variable on the right as the x variable for the line.

Finally, compute_model_prediction() takes a model argument. This is the name of the R modelling function that compute_model_prediction() should use to calculate the line. lm() is R's function for building linear models. compute_smooth() is a special case of compute_model_prediction() where the model argument is set to loess by default. In this case the function will create a smoothed set of points.

```
faithful %>%
  compute_model_prediction(eruptions ~ waiting, model = "lm")
```

```
##       pred_    resp_
## 1  43.00000 1.377986
## 2  43.67089 1.428724
## 3  44.34177 1.479461
## 4  45.01266 1.530199
## 5  45.68354 1.580937
## 6  46.35443 1.631674
## 7  47.02532 1.682412
## 8  47.69620 1.733150
## 9  48.36709 1.783888
## 10 49.03797 1.834625
## 11 49.70886 1.885363
## 12 50.37975 1.936101
## 13 51.05063 1.986839
## 14 51.72152 2.037576
## 15 52.39241 2.088314
## 16 53.06329 2.139052
## 17 53.73418 2.189790
## 18 54.40506 2.240527
## 19 55.07595 2.291265
## 20 55.74684 2.342003
## 21 56.41772 2.392741
## 22 57.08861 2.443478
## 23 57.75949 2.494216
## 24 58.43038 2.544954
## 25 59.10127 2.595691
## 26 59.77215 2.646429
## 27 60.44304 2.697167
## 28 61.11392 2.747905
## 29 61.78481 2.798642
## 30 62.45570 2.849380
## 31 63.12658 2.900118
## 32 63.79747 2.950856
## 33 64.46835 3.001593
## 34 65.13924 3.052331
## 35 65.81013 3.103069
## 36 66.48101 3.153807
## 37 67.15190 3.204544
```

```
## 38 67.82278 3.255282
## 39 68.49367 3.306020
## 40 69.16456 3.356758
## 41 69.83544 3.407495
## 42 70.50633 3.458233
## 43 71.17722 3.508971
## 44 71.84810 3.559708
## 45 72.51899 3.610446
## 46 73.18987 3.661184
## 47 73.86076 3.711922
## 48 74.53165 3.762659
## 49 75.20253 3.813397
## 50 75.87342 3.864135
## 51 76.54430 3.914873
## 52 77.21519 3.965610
## 53 77.88608 4.016348
## 54 78.55696 4.067086
## 55 79.22785 4.117824
## 56 79.89873 4.168561
## 57 80.56962 4.219299
## 58 81.24051 4.270037
## 59 81.91139 4.320775
## 60 82.58228 4.371512
## 61 83.25316 4.422250
## 62 83.92405 4.472988
## 63 84.59494 4.523725
## 64 85.26582 4.574463
## 65 85.93671 4.625201
## 66 86.60759 4.675939
## 67 87.27848 4.726676
## 68 87.94937 4.777414
## 69 88.62025 4.828152
## 70 89.29114 4.878890
## 71 89.96203 4.929627
## 72 90.63291 4.980365
## 73 91.30380 5.031103
## 74 91.97468 5.081841
## 75 92.64557 5.132578
## 76 93.31646 5.183316
## 77 93.98734 5.234054
## 78 94.65823 5.284792
## 79 95.32911 5.335529
```

```
## 80 96.00000 5.386267
```

```
mtcars %>% compute_smooth(mpg~wt)
```

```
##       pred_    resp_
## 1  1.513000 32.08897
## 2  1.562506 31.68786
## 3  1.612013 31.28163
## 4  1.661519 30.87037
## 5  1.711025 30.45419
## 6  1.760532 30.03318
## 7  1.810038 29.60745
## 8  1.859544 29.17711
## 9  1.909051 28.74224
## 10 1.958557 28.30017
## 11 2.008063 27.83462
## 12 2.057570 27.34766
## 13 2.107076 26.84498
## 14 2.156582 26.33229
## 15 2.206089 25.81529
## 16 2.255595 25.29968
## 17 2.305101 24.79115
## 18 2.354608 24.29542
## 19 2.404114 23.81818
## 20 2.453620 23.36514
## 21 2.503127 22.95525
## 22 2.552633 22.61385
## 23 2.602139 22.32759
## 24 2.651646 22.08176
## 25 2.701152 21.86167
## 26 2.750658 21.65260
## 27 2.800165 21.43987
## 28 2.849671 21.20875
## 29 2.899177 20.95334
## 30 2.948684 20.71584
## 31 2.998190 20.49571
## 32 3.047696 20.28293
## 33 3.097203 20.06753
## 34 3.146709 19.83950
## 35 3.196215 19.58885
## 36 3.245722 19.29716
```

```
## 37 3.295228 18.94441
## 38 3.344734 18.56700
## 39 3.394241 18.20570
## 40 3.443747 17.90090
## 41 3.493253 17.62060
## 42 3.542759 17.34002
## 43 3.592266 17.07908
## 44 3.641772 16.81759
## 45 3.691278 16.55757
## 46 3.740785 16.30833
## 47 3.790291 16.07916
## 48 3.839797 15.87937
## 49 3.889304 15.70181
## 50 3.938810 15.52594
## 51 3.988316 15.35173
## 52 4.037823 15.17933
## 53 4.087329 15.00894
## 54 4.136835 14.84072
## 55 4.186342 14.67484
## 56 4.235848 14.51148
## 57 4.285354 14.35082
## 58 4.334861 14.19302
## 59 4.384367 14.03826
## 60 4.433873 13.88672
## 61 4.483380 13.73856
## 62 4.532886 13.59396
## 63 4.582392 13.45310
## 64 4.631899 13.31614
## 65 4.681405 13.18326
## 66 4.730911 13.05464
## 67 4.780418 12.93045
## 68 4.829924 12.81086
## 69 4.879430 12.69604
## 70 4.928937 12.58617
## 71 4.978443 12.48143
## 72 5.027949 12.38198
## 73 5.077456 12.28799
## 74 5.126962 12.19966
## 75 5.176468 12.11713
## 76 5.225975 12.04060
## 77 5.275481 11.97023
## 78 5.324987 11.90620
```

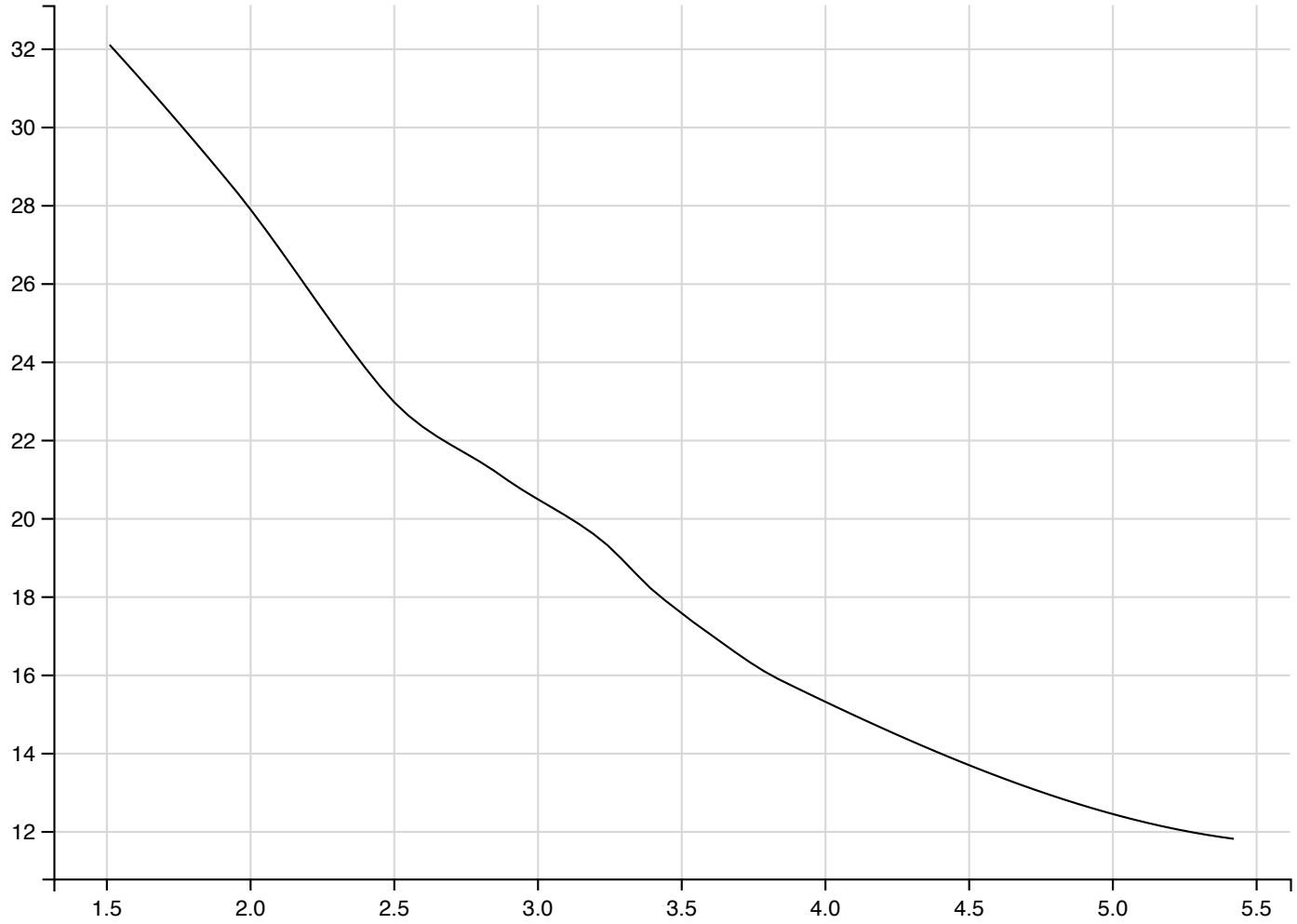
```
## 79 5.374494 11.84868
## 80 5.424000 11.79784
```

Compute_smooth() model fits

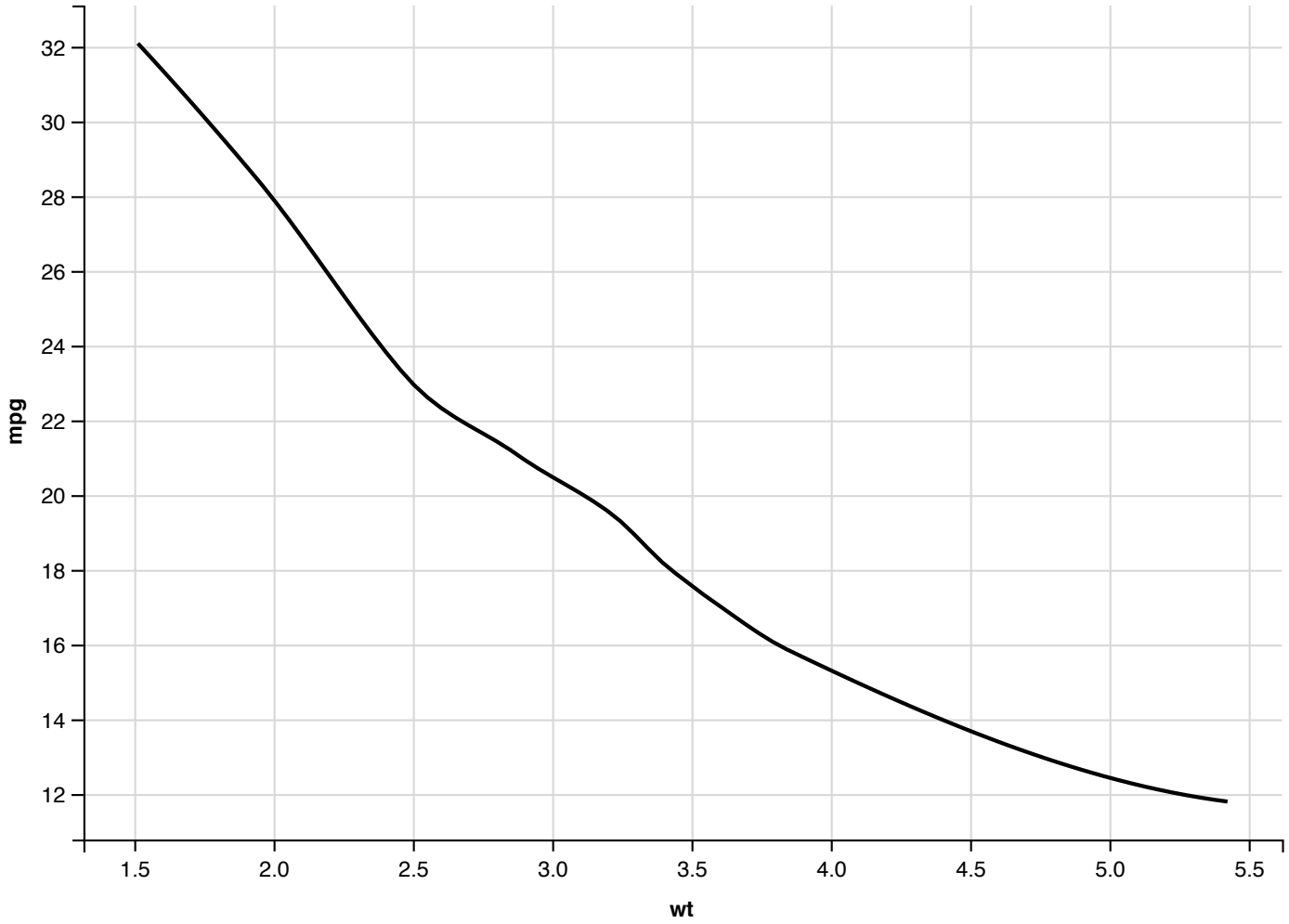
compute_smooth() always returns a data set with two columns, one named pred_ and one named resp_. As a result, it is very easy to use compute_smooth() to plot a smoothed line of your data. For example, you can extend your code from the last exercise to plot the results of compute_smooth() as a line graph.

Calling compute_smooth() can be a bit of a hassle, so ggvis includes a layer that automatically calls compute_smooth() in the background and plots the results as a smoothed line. That layer is layer_smooths().

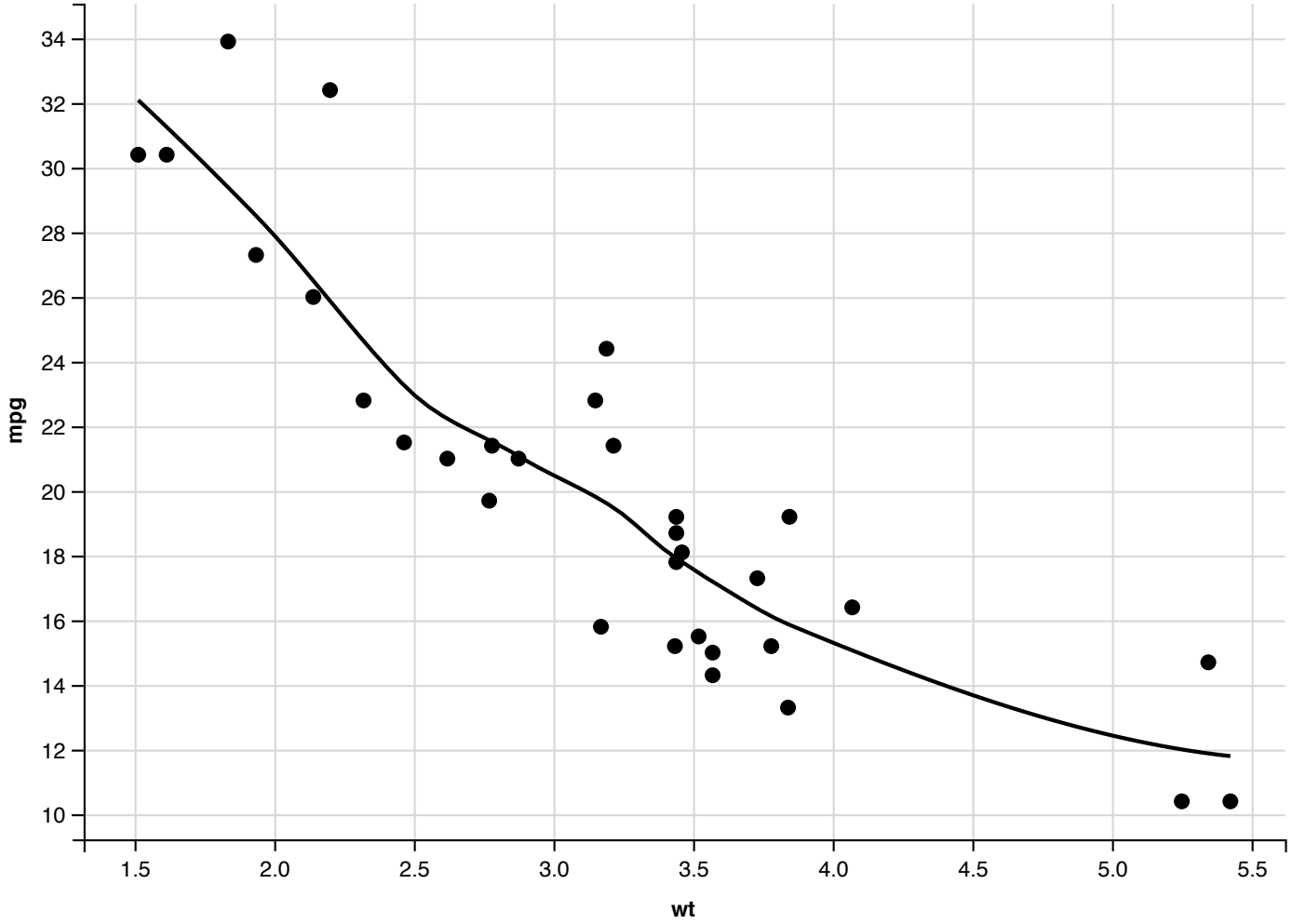
```
# Use 'ggvis()' and 'layer_lines()' to plot the results of compute smooth
mtcars %>% compute_smooth(mpg ~ wt) %>% ggvis(~pred_, ~resp_) %>% layer_lines()
```




```
# Recreate the graph you coded above with 'ggvis()' and 'layer_smooths()'
mtcars %>% ggvis(~wt, ~mpg) %>% layer_smooths()
```



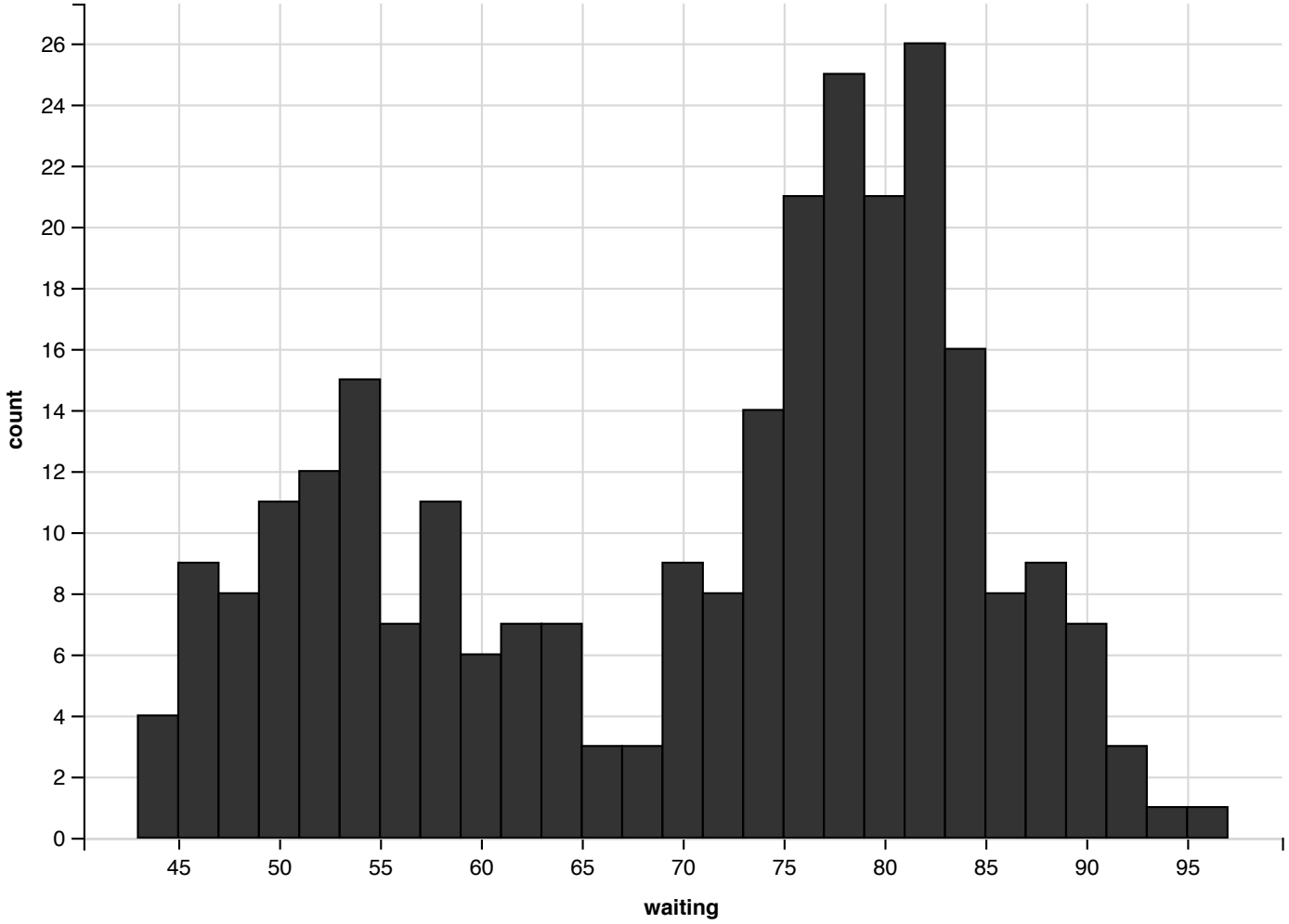

```
# Extend the code for the second plot and add 'layer_points()' to the graph
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points() %>% layer_smooths()
```



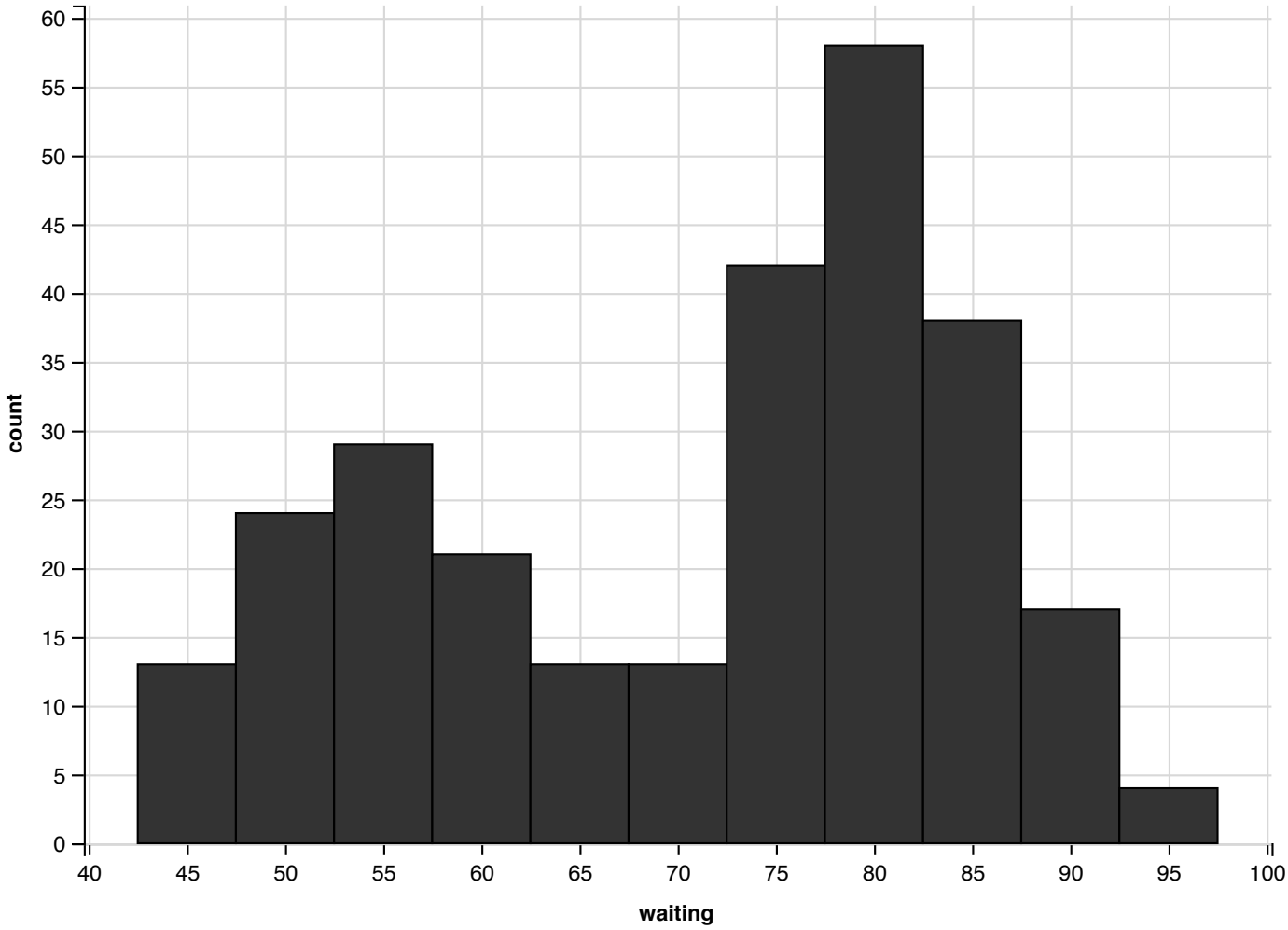
####HISTOGRAMS

```
# Build a histogram of the waiting variable of the faithful data set.
faithful %>% ggvis(~waiting) %>% layer_histograms()
```

```
## Guessing width = 2 # range / 27
```




```
# Build the same histogram, but with a binwidth of 5 units
faithful %>% ggvis(~waiting) %>% layer_histograms(width = 5)
```



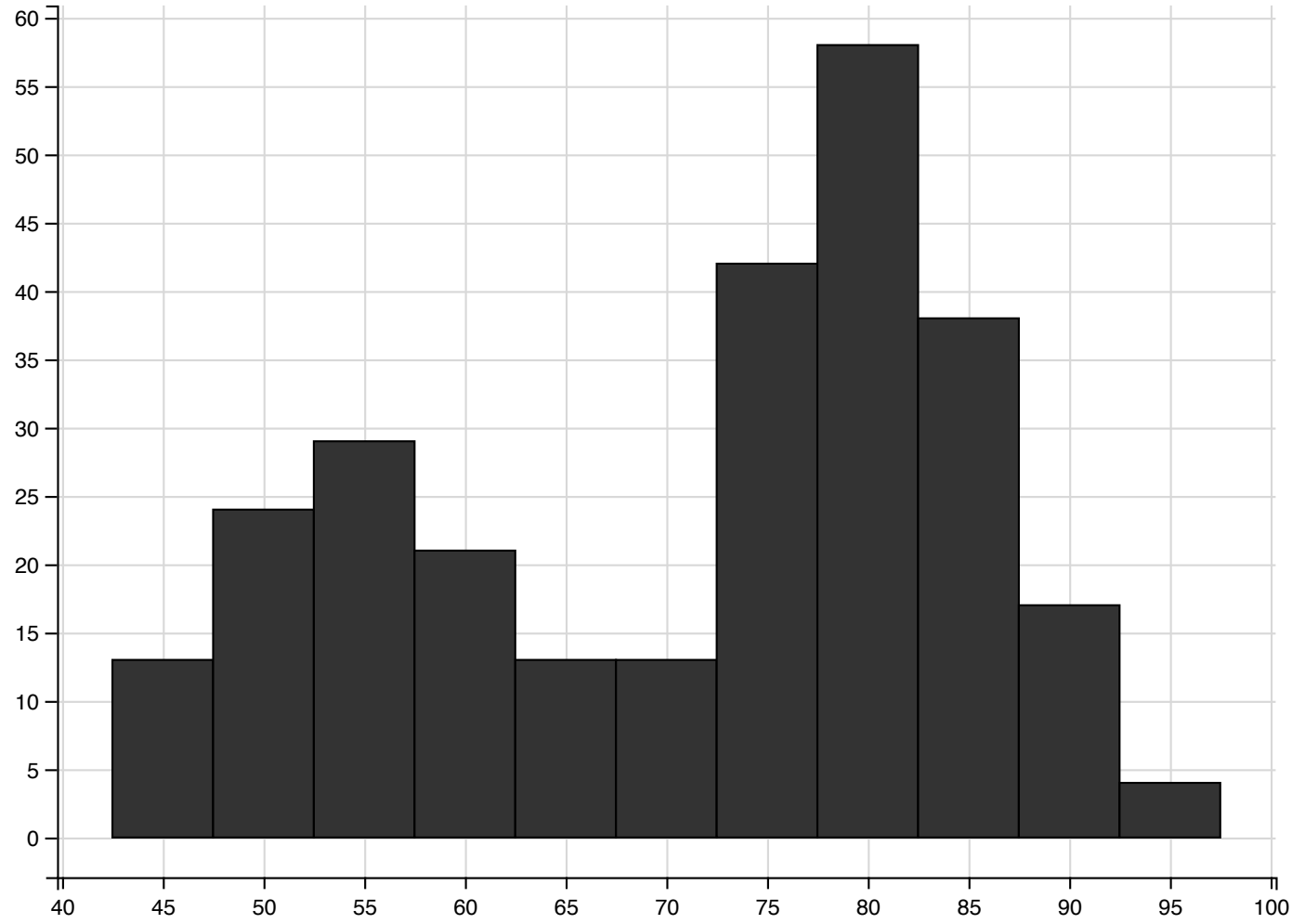
Behind the scenes, layer_histograms() calls compute_bin() to calculate these counts. You can calculate the same values by calling compute_bin() manually. compute_bin() requires at least two arguments: a data set (which you will provide with the %>% syntax), and a variable name to bin on. You can also pass compute_bin() a binwidth argument, just as you pass layer_histograms() a binwidth argument.

compute_bin() returns a data frame that provides everything you need to build a histogram from scratch. Notice the similarity with previous cases: combining compute_smooth() and layer_points() had the exact same result as using layer_smooths() directly!

```
# Transform the code below: just compute the bins instead of plotting a histogram.
faithful %>% compute_bin(~waiting, width = 5)
```

```
##    count_ x_ xmin_ xmax_ width_
## 1      13 45  42.5  47.5      5
## 2      24 50  47.5  52.5      5
## 3      29 55  52.5  57.5      5
## 4      21 60  57.5  62.5      5
## 5      13 65  62.5  67.5      5
## 6      13 70  67.5  72.5      5
## 7      42 75  72.5  77.5      5
## 8      58 80  77.5  82.5      5
## 9      38 85  82.5  87.5      5
## 10     17 90  87.5  92.5      5
## 11      4 95  92.5  97.5      5
```

```
# Combine the solution to the first challenge with layer_rects() to build a histogram.
faithful %>%
  compute_bin(~waiting, width = 5) %>%
  ggvis(x = ~xmin_, x2 = ~xmax_, y = 0, y2 = ~count_) %>%
  layer_rects()
```



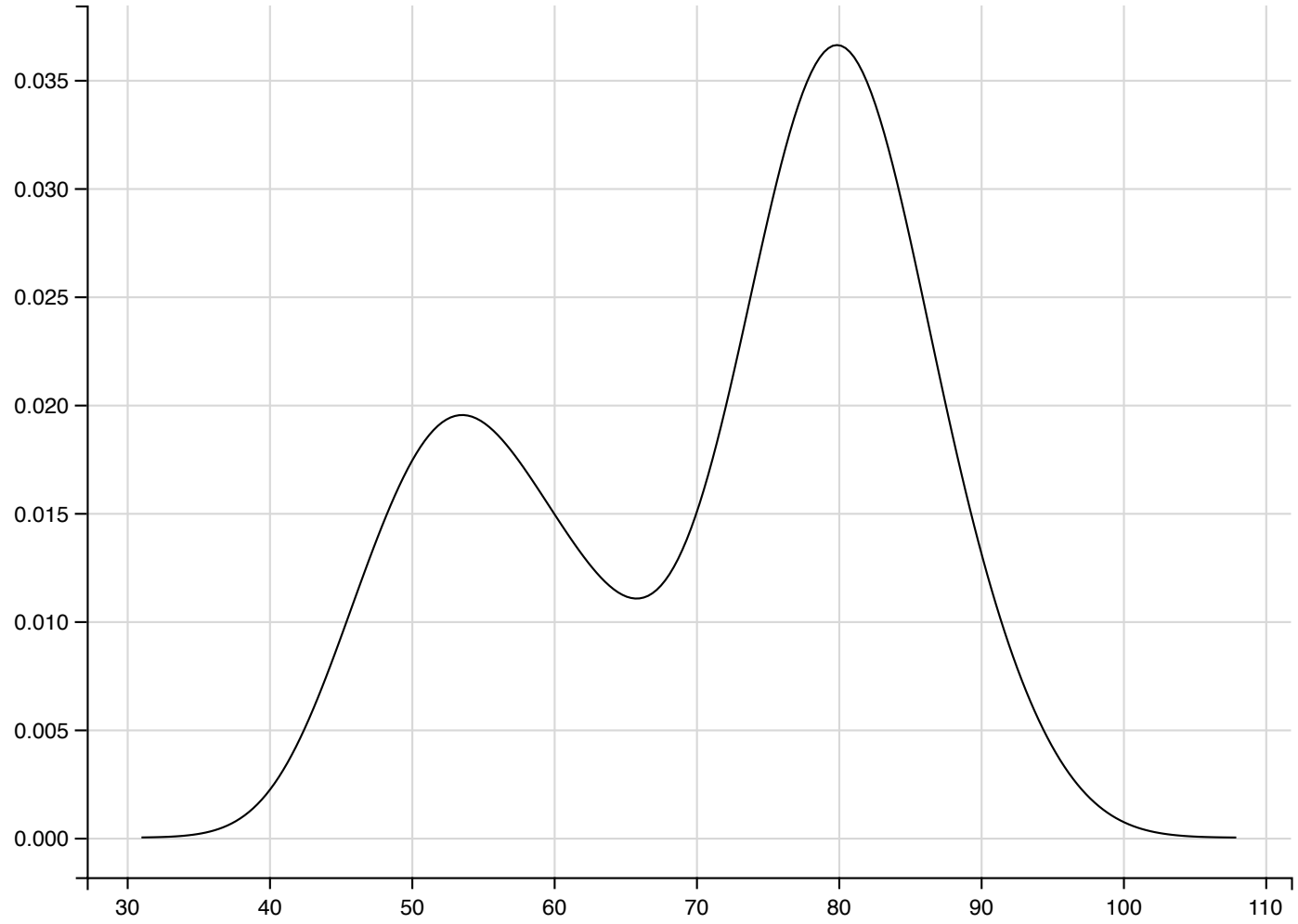
DENSITY PLOTS

Density plots provide another way to display the distribution of a single variable. A density plot uses a line to display the density of a variable at each point in its range. You can think of a density plot as a continuous version of a histogram with a different y scale (although this is not exactly accurate).

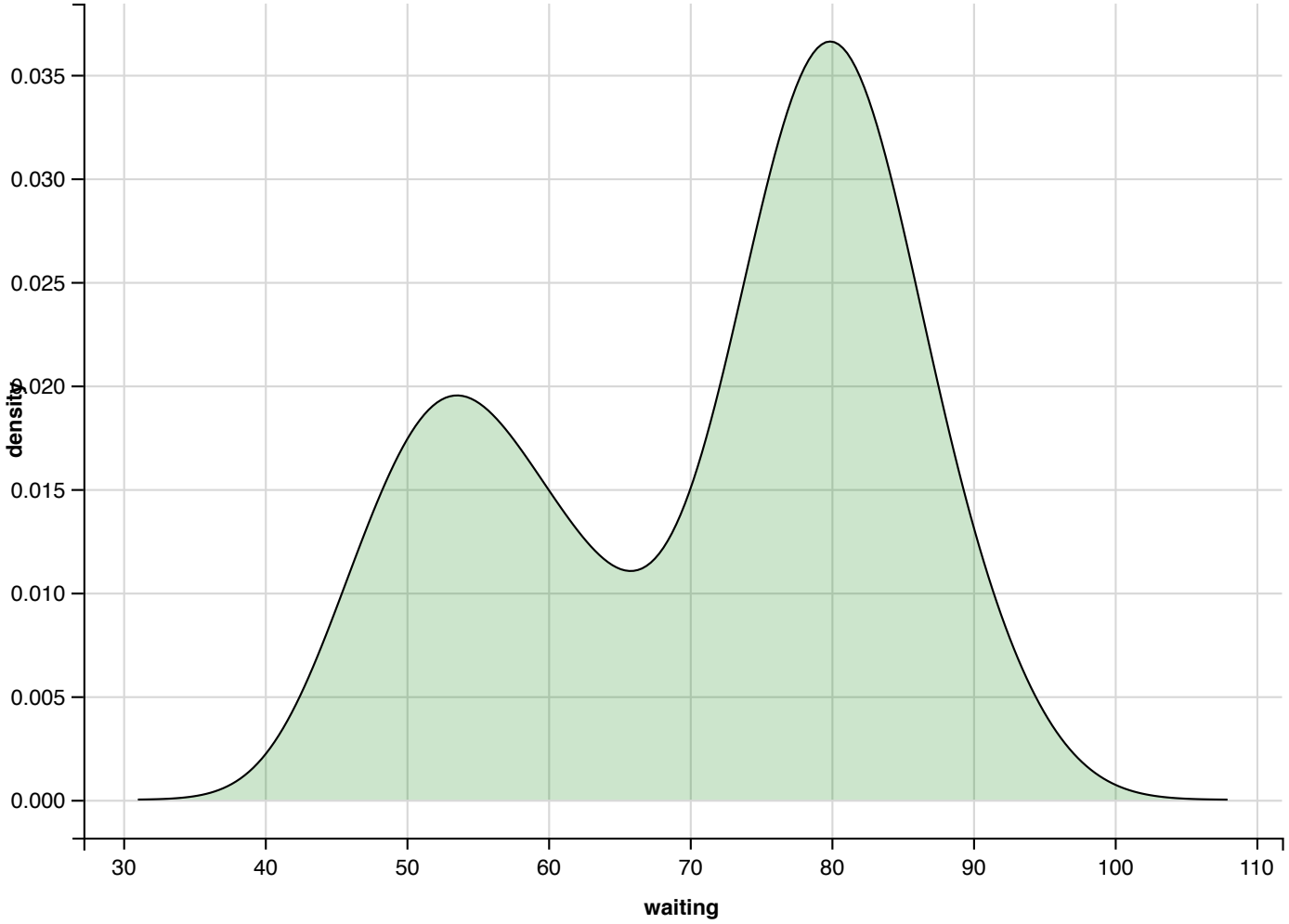
compute_density() takes two arguments, a data set and a variable name. It returns a data frame with two columns: pred_, the x values of the variable's density line, and resp_, the y values of the variable's density line.

You can use layer_densities() to create density plots. Like layer_histograms() it calls the compute function that it needs in the background, so you do not need to worry about calling compute_density().

```
# Combine compute_density() with layer_lines() to make a density plot of the waiting variab
le.
faithful %>% compute_density(~waiting) %>% ggvis(~pred_, ~resp_) %>% layer_lines()
```




```
# Build a density plot directly using layer_densities. Use the correct variables and proper ties.
faithful %>% ggvis(~waiting, fill := "green") %>% layer_densities()
```


0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
density
30
40
50
60
70
80
90
100
110
waiting