© Thomas Mailund 2022

T. Mailund, *Beginning Data Science in R 4*

https://doi.org/10.1007/978-1-4842-8155-0_5

5. Working with Large Data Sets

Thomas Mailund[1]

(1) Aarhus, Denmark

The concept of Big Data refers to enormous data sets, sets of sizes where you need data warehouses to store it, where you typically need sophisticated algorithms to handle the data and distributed computations to get anywhere with it. At the very least, we talk many gigabytes of data but also often terabytes or exabytes.

Dealing with Big Data is also part of data science, but it is beyond the scope of this book. This chapter is on large data sets and how to deal with data that slows down your analysis, but it is not about data sets so large that you cannot analyze it on your desktop computer.

If we ignore the Big Data issue, what a large data set is depends very much on what you want to do with the data. That comes down to the complexity of what you are trying to achieve. Some algorithms are fast and can scan through data in linear time—meaning that the time it takes to analyze the data is linear in the number of data points—while others take exponential time and cannot be applied to data sets with more than a few tens or hundreds of data points. The science of what you can do with data in a given amount of time, or a given amount of space (be it RAM or disk space or whatever you need), is called complexity theory and is one of the fundamental topics in computer science. In practical terms, though, it usually boils down to how long you are willing to wait for an analysis to finish, and it is a very subjective measure.

In this chapter, we will just consider a few cases where I have found in my own work that data gets a bit too large to do what I want, and I have

had to deal with it in various ways. Your cases are likely to be different, but maybe you can get some inspiration, at least, from these cases.

Subsample Your Data Before You Analyze the Full Data Set

The first point I want to make, though, is this: you very rarely need to analyze a complete data set to get at least an idea of how the data behaves. Unless you are looking for very rare events, you will get as much feeling for the data looking at a few thousands of data points as you would from looking at a few million.

Sometimes, you do need extensive data to find what you are looking for. This is the case, for example, when looking for associations between genetic variation and common diseases where the association can be very weak, and you need lots of data to distinguish between chance associations and genuine associations. But for most signals in data that are of practical importance, you will see the signals in smaller data sets. So before you throw the full power of all your data at an analysis, especially if that analysis turns out to be very slow, you should explore a smaller sample of your data.

Here, you must pick a random sample. There is often structure in data beyond the columns in a data frame. This could be a structure caused by when the data was collected. If the data is ordered by when the data was collected, then the first data points you have can be different from later data points. This isn't explicitly represented in the data, but the structure is there nevertheless. Randomizing your data alleviates problems that can arise from this. Randomizing might remove a subtle signal, but with the power of statistics, we can deal with random noise. It is much harder to deal with consistent biases we just don't know about.

If you have a large data set, and your analysis is being slowed down because of it, don't be afraid to pick a random subset and analyze that. You may see signals in the subsample that is not present in the full data

set, but it is much less likely than you might fear. When you are looking for signals in your data, you always have to worry about false signals. But it is not more liable to pop up in a smaller data set than in a larger. And with a more extensive data set to check your results against later, you are less likely to stick with wrong results at the end of your analysis.

Getting spurious results is mostly a concern with traditional hypothesis testing. If you set a threshold for when a signal is significant at 5% for p-values, you will see spurious results one time out of twenty. If you don't correct for multiple testing, you will be almost guaranteed to see false results. These are unlikely to survive when you later throw the complete data at your models.

In any case, with large data sets, you are more likely to have statistically significant deviations from a null model, which are entirely irrelevant to your analysis. We usually use simple null models when analyzing data, and any complex data sets are not generated from a simple null model. With enough data, the chances are that anything you look at will have significant deviations from your simple null model. The real world does not draw samples from a simple linear model. There is always some extra complexity. You won't see it with a few data points, but with enough data, you can reject any null model. It doesn't mean that what you see has any practical importance.

If you have signals you can discover in a smaller subset of your data, and these signals persist when you look at the full data set, you can trust them that much more.

So, if the data size slows you down, downsample and analyze a subset.

You can use `dplyr` functions `sample_n()` and `sample_frac()` to sample from a data frame. Use `sample_n()` to get a fixed number of rows and `sample_frac()` to get a fraction of the data:

```
iris |> sample_n(size = 5)
##   Sepal.Length Sepal.Width Petal.Length
## 1          5.5         3.5          1.3
```

```
## 2          6.4         2.8          5.6
## 3          5.7         2.8          4.5
## 4          5.0         3.4          1.5
## 5          5.1         3.8          1.6
##   Petal.Width    Species
## 1         0.2     setosa
## 2         2.2  virginica
## 3         1.3 versicolor
## 4         0.2     setosa
## 5         0.2     setosa
iris |> sample_frac(size = 0.02)
##   Sepal.Length Sepal.Width Petal.Length
## 1          6.3         2.5          5.0
## 2          6.4         2.8          5.6
## 3          6.3         3.3          4.7
##   Petal.Width    Species
## 1         1.9  virginica
## 2         2.2  virginica
## 3         1.6 versicolor
```

(Your output will be different, since these are random functions, but it should look similar.)

Of course, to sample using `dplyr`, you need your data in a form that `dplyr` can manipulate, and if the data is too large even to load into R, then you cannot have it in a data frame to sample from, to begin with. Luckily, `dplyr` has support for using data that is stored on disk rather than in RAM, in various back-end formats, as we will see later. It is, for example, possible to connect a database to `dplyr` and sample from a large data set this way.

Running Out of Memory During an Analysis

R can be very wasteful of RAM. Even if your data set is small enough to fit in memory and small enough that the analysis time is not a substantial problem, it is easy to run out of memory because R remembers more than is immediately apparent.

In R, all objects are immutable,[1] so whenever you modify an object, you are actually creating a new object. The implementation of this is smart enough that you only have independent copies of data when it is different. Having two different variables to refer to the same data frame doesn't mean that the data frame is represented twice. Still, if you modify the data frame in one of the variables, then R will create a copy with the modifications, and you now have the data twice, accessible through the two variables. If you only refer to the data frame through one variable, then R is smart enough not to make a copy, though.

You can examine memory usage and memory changes using the `pryr` package:

```
library(pryr)
```

For example, we can see what the cost is of creating a new vector:

```
mem_change(x <- rnorm(10000))
## 83.9 kB
```

(The exact value you see here will depend on your computer and your installation, so don't be surprised if it differs from mine.)

R doesn't allow modification of data, so when you "modify" a vector, it makes a new copy that contains the changes. This doesn't significantly increase the memory usage because R is smart about only copying when more than one variable refers to an object:

```
mem_change(x[1] <- 0)
## 528 B
```

If we assign the vector to another variable, we do not use twice the memory, because both variables will just refer to the same object:

```
mem_change(y <- x)
## 584 B
```

but if we modify one of the vectors, we will have to make a copy, so the other vector remains the same:

```
mem_change(x[1] <- 0)
## 80.6 kB
```

This is another reason for using pipelines rather than assigning to many variables during an analysis. You are fine if you assign back to a variable, though, so the `%<>%` operator does not lead to a lot of copying.

Even using pipelines, you still have to be careful, though. Many functions in R will again copy data.

If a function does any modification to data, the data is copied to a local variable. There might be some sharing, so, for example, just referring to a data frame in a local variable does not create a copy. Still, if you, for example, split a data frame into training and test data in a function, then you will be copying and now represent all the data twice. This memory is freed after the function finishes its computations, so it is really only a problem if you are very close to the limit of RAM.

If such copied data is returned in some way from the function, it is not freed. It is, for example, not unusual that model fitting functions will save the entire fitting data in the returned object. If it is copied without modification, again we do not see a memory increase. Yet, if the function modifies it in any way, we are now using twice the memory as before.

When you have problems with running out of memory in data analysis in R, it is usually not that you cannot represent your data initially but that you end up having many copies. You can avoid this to some extent by not storing temporary data frames in variables and by not implicitly storing copies of data frames in the output of functions, or you can explicitly remove stored data using the `rm()` function to free up memory.

Too Large to Plot

The first point where I typically run into problems with large data sets is not that I run out of RAM, but when I am plotting, especially when making scatter plots; box plots or histograms summarize the data and are usually not a problem.

There are two problems when making scatter plots with a lot of data. The first is that if you create files from scatter plots, you will create a plot that contains every single individual point. That can be a huge file. Worse, it will take forever to plot, since a viewer will have to consider every sin-

gle point. You can avoid this problem by creating raster graphics instead of PDFs, but that takes us to the second issue. With too many points, a scatter plot is just not informative any longer. Points will overlap, and you cannot see how many individual data points fall on the plot. This usually becomes a problem long before the computational time becomes an issue.

If, for example, we have a data frame with 10,000 points

```
d <- data.frame(x = rnorm(10000), y = rnorm(10000))
```

we can still make a scatter plot, and if the plot is saved as raster graphic instead of PDF, the file will not be too large to watch or print:

```
d |> ggplot(aes(x = x, y = y)) +
  geom_point()
```

The result will just not be all that informative; see Figure 5-1. The points are shown on top of each other, making it hard to see if the big black cloud of points has different densities in some places than others.

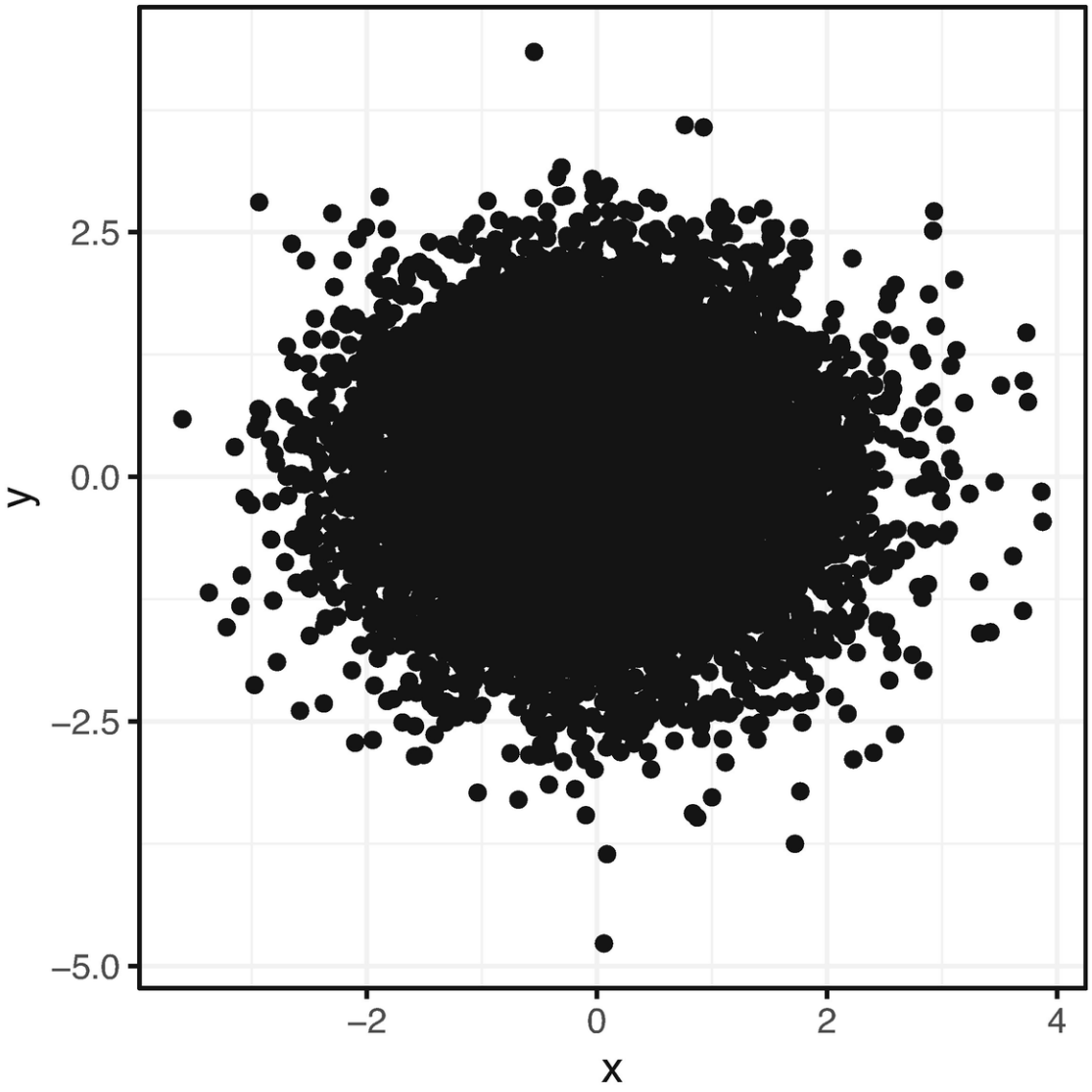

Figure 5-1 A scatter plot with too many points

The solution is to represent points such that they are still visible when there are many overlapping points. If the points are overlapping because they have the same x or y coordinates, you can jitter them; we saw that in the previous chapter. Another solution to the same problems is plotting the points with alpha levels, so each point is partly transparent. You can see the density of points because they are slightly transparent, but you still end up with a plot with very many points; see Figure 5-2.

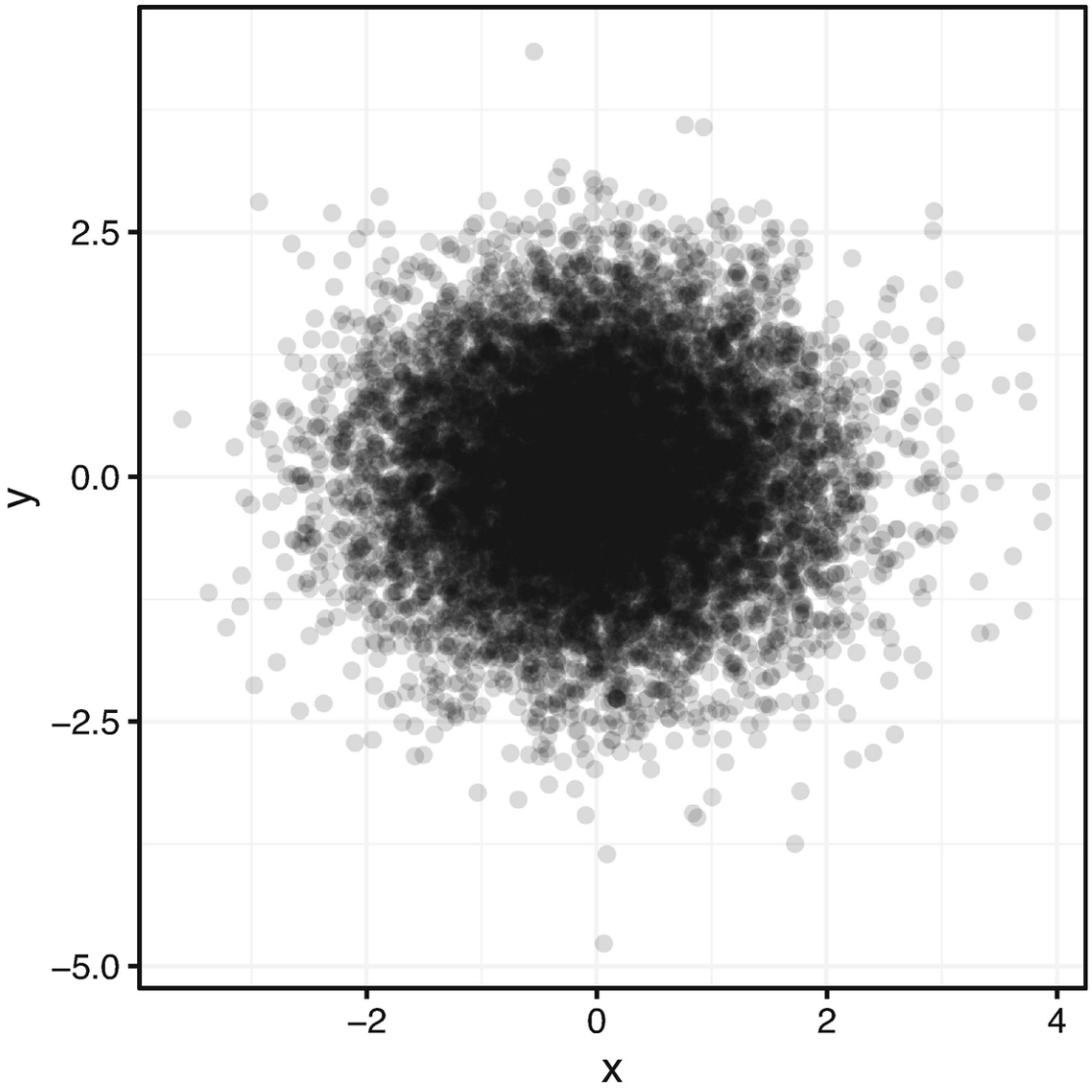

Figure 5-2 A scatter plot with alpha values

```
d |> ggplot(aes(x = x, y = y)) +
  geom_point(alpha = 0.2)
```

This, however, doesn't solve the problem that files will draw every single point and cause printing and file size problems. A scatter plot with transparency is just a way of showing the 2D density, though, and we can do that directly using the `geom_density_2d()` function; see Figure 5-3.

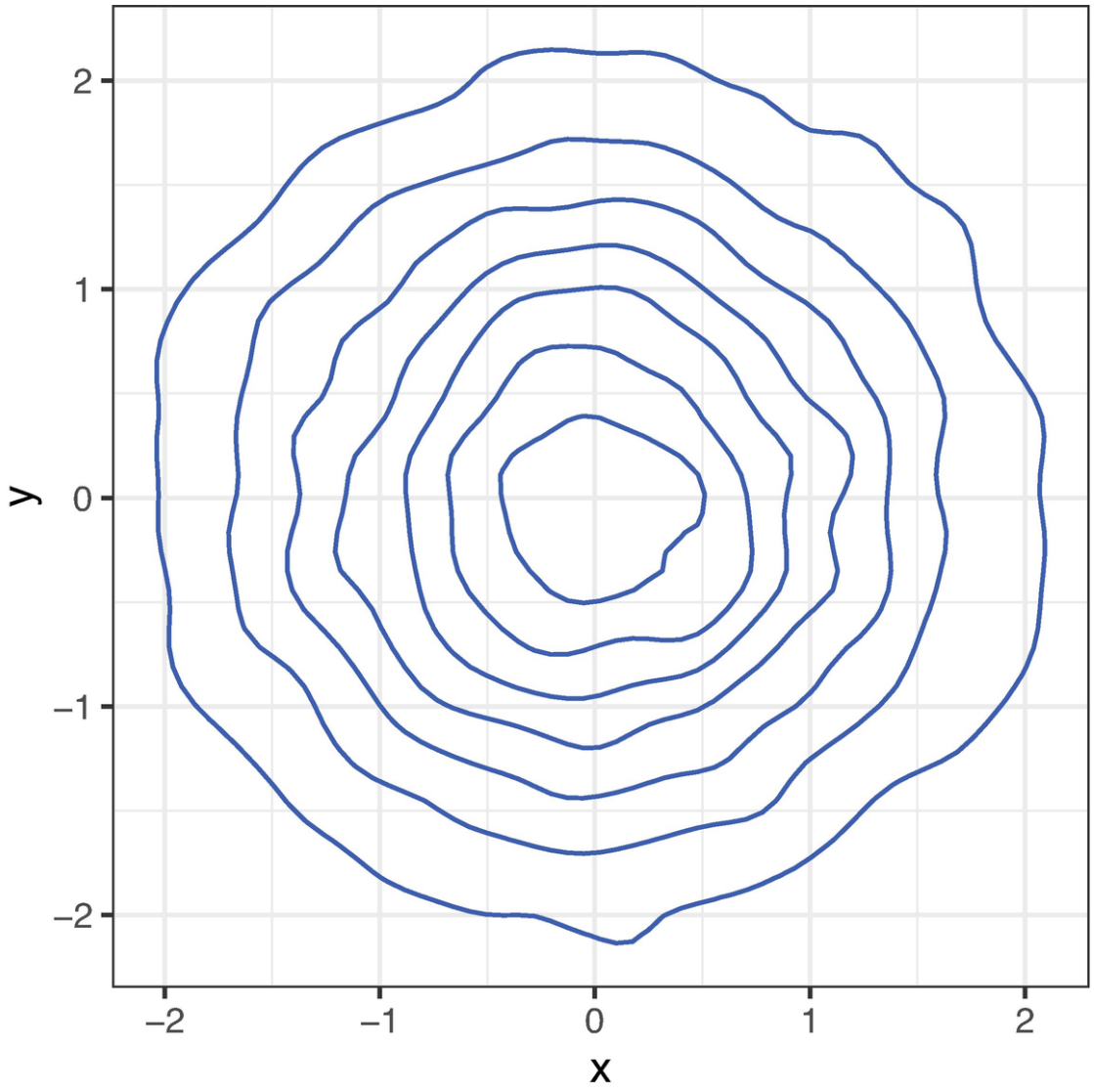

Figure 5-3 A 2D density plot

```
d |> ggplot(aes(x = x, y = y)) +
  geom_density_2d()
```

The plot shows the contour of the density.

An alternative way of showing a 2D density is using a so-called hex plot, the 2D equivalent of a histogram. The plot splits the 2D plane into hexagonal bins and displays the count of points falling into each bin.

To use it, you need to install the package `hexbin` and use the `ggplot2` function `geom_hex()`; see Figure 5-4.

```
d |> ggplot(aes(x = x, y = y)) +
  geom_hex()
```

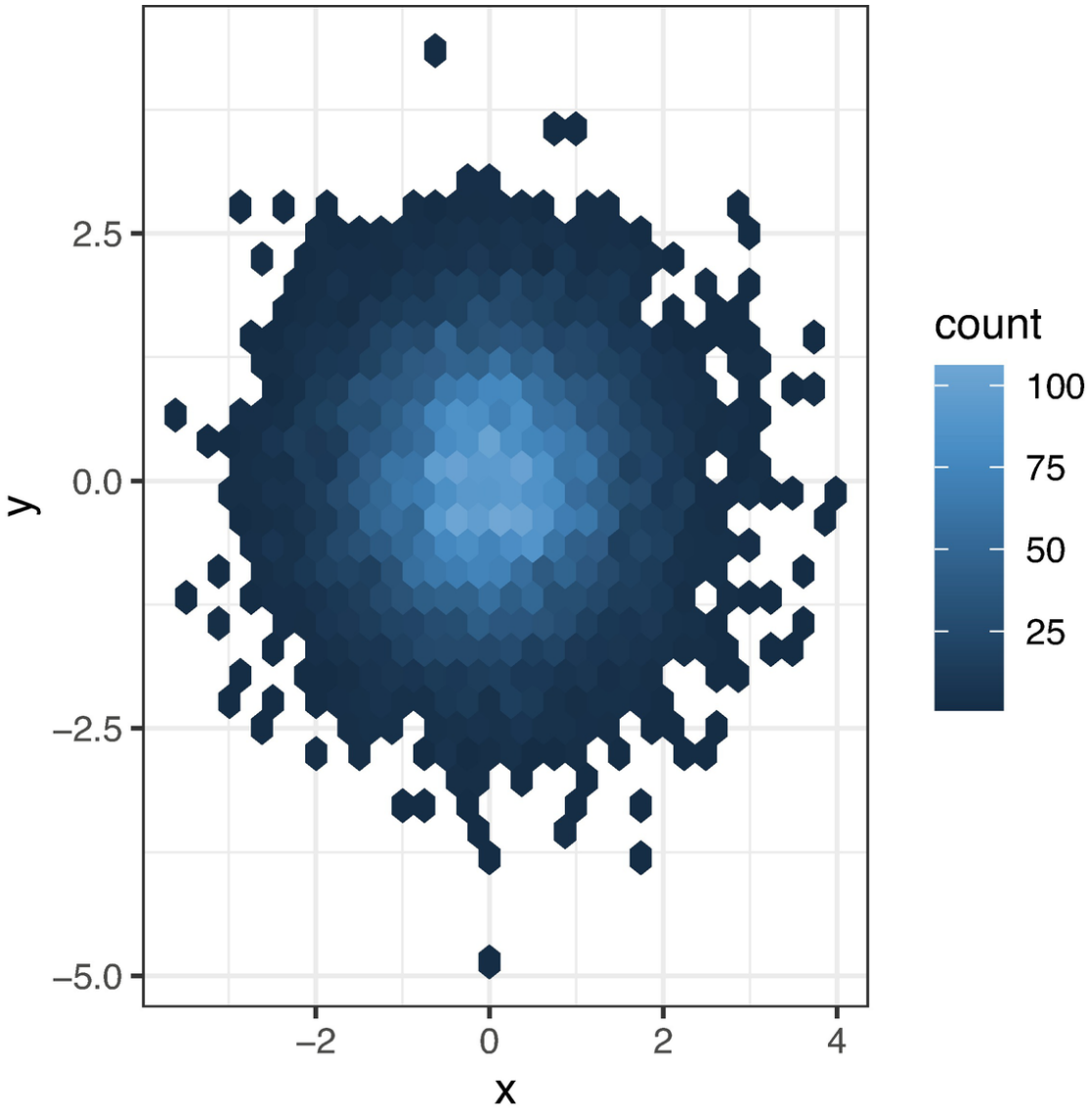

Figure 5-4 A hex plot

The colors used by `geom_hex()` are the fill colors, so you can change them using the `scale_fill` functions. You can also combine hex and 2D density plots to get both the bins and contours displayed; see Figure 5-5.

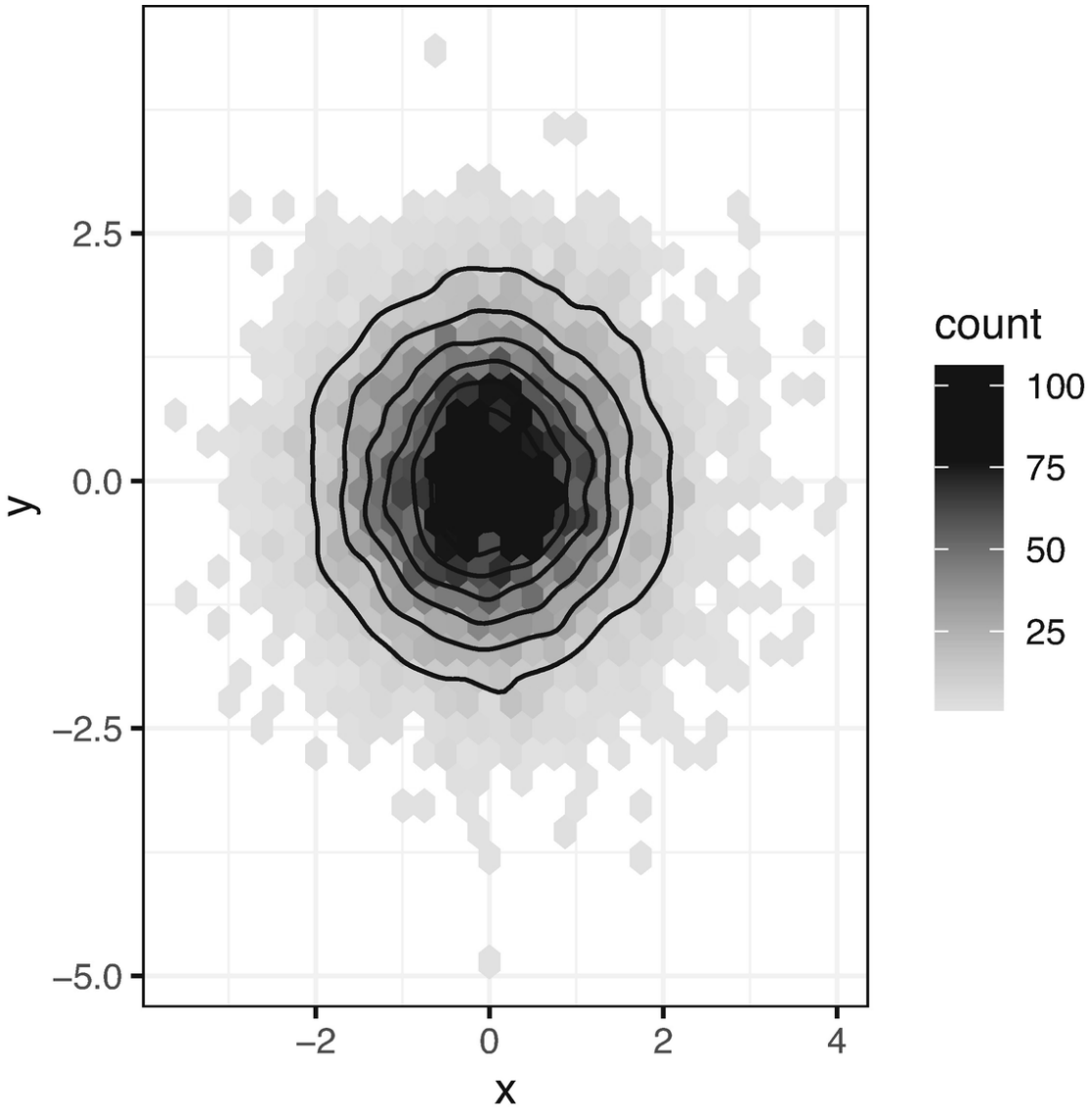

Figure 5-5 A plot combining hex and 2D density

```
d |> ggplot(aes(x = x, y = y)) +
  geom_hex() +
  scale_fill_gradient(low = "lightgray", high = "grey10") +
  geom_density2d(color = "black")
```

Too Slow to Analyze

When plotting data, the problem is usually only in scatter plots. Otherwise, you don't have to worry about having too many points or too large plot files. Even when plotting lots of points, the real problem doesn't show up until you create a plot and load it into your viewer or send it to the printer.

With enough data points, though, most analyses will slow down, and that can be a problem.

The easy solution is again to subsample your data and work with that. It will show you the relevant signals in your data without slowing down your analysis.

If that is not a solution for you, you need to pick analysis algorithms that work more efficiently. That typically means linear time algorithms. Unfortunately, many standard algorithms are not linear time, and even if they are, the implementation does not necessarily make it easy to fit data in batches where the model parameters can be updated one batch at a time. You often need to find packages specifically written for that or make your own.

One package that provides both a memory-efficient linear model fitting (it avoids creating a model matrix that would have rows for each data point and solving equations for that) and functionality for updating the model in batches is the `biglm` package:

```
library(biglm)
```

You can use it for linear regression using the `biglm()` function instead of the `lm()` function, and you can use the `bigglm()` function for generalized linear regression instead of the `glm()` function (see Chapter 6 for details on these).

If you are using a data frame format that stores the data on disk and has support for `biglm` (see the next section), the package will split the data into chunks it can load into memory and analyze. If you do not have a package that handles this automatically, you can split the data into chunks yourself. As a toy example, we can consider the `cars` data set and try to fit a linear model of stopping distance as a function of speed but do this in batches of ten data points. Of course, we can easily fit such a small data set without splitting it into batches, we don't even need to use the `biglm()` function for it, but as an example, it will do.

Defining the slice indices requires some arithmetic, and after that, we can extract subsets of the data using the `slice()` function from `dplyr`. We can create a linear model from the first slice and then update using the following:

```
slice_size <- 10
n <- nrow(cars)
slice <- cars |> slice(1:slice_size)
model <- biglm(dist ~ speed, data = slice)
for (i in 1:(n/slice_size-1)) {
  slice <- cars |> slice((i*slice_size+1):((i+1)*slice_size))
  model <- update(model, moredata = slice)
}
Model
## Large data regression model: biglm(dist ~ speed, data =
slice)
## Sample size = 50
```

Bayesian model fitting methods have a (somewhat justified) reputation for being slow, but Bayesian models based on conjugate priors are ideal for this. Having a conjugate prior means that the posterior distribution you get out of analyzing one data set can be used as the prior distribution for the next data set. This way, you can split the data into slices and fit the first slice with a real prior and the subsequent slices with the result of the previous model fits.

The Bayesian linear regression model in the second project, the last chapter of this book, is one such model. There, we implement an `update()` function that fits a new model based on a data set and a previously fitted model. Using it on the `cars` data, splitting the data into chunks of size 10, would look very similar to the `biglm` example.

Even better are models where you can analyze slices independently and then combine the results to get a model for the full data set. These can not only be analyzed in batches, but the slices can be handled in parallel, exploiting multiple cores or multiple computer nodes. For gradient descent optimization approaches, you can compute gradients for slices independently and then combine them to make a step in the optimization.

There are no general solutions for dealing with data that is too large to be efficiently analyzed, though. It requires thinking about the algorithms used and usually also some custom implementation of these unless you are lucky and can find a package that can handle data in batches.

Too Large to Load

R wants to keep the data it works on in memory. So if your computer doesn't have the RAM to hold it, you are out of luck. At least if you work with the default data representations like 'data.frame'. R usually also wants to use 32-bit integers for indices. Since it uses both positive and negative numbers for indices, you are limited to indexing around two billion data points even if you could hold more in memory.

There are different packages for dealing with this. One such is the `ff` package that works with the kind of tables we have used so far but uses memory-mapped files to represent the data and loads data chunks into memory as needed:

```
library(ff)
```

It essentially creates flat files and has functionality for mapping chunks of these into memory when analyzing them.

It represents data frames as objects of class `ffdf`. These behave just like data frames if you use them as such, and you can translate a data frame into an `ffdf` object using the `as.ffdf()` function.

You can, for example, convert the `cars` data into an `ffdf` object using

```
ffcars <- as.ffdf(cars)
summary(ffcars)
##       Length Class     Mode
## speed 50     ff_vector list
## dist  50     ff_vector list
```

Of course, if you can already represent a data frame in memory, there is no need for this translation, but `ff` also has functions for creating `ffdf` objects from files. If, for example, you have a large file as comma-separated values, you can use `read.csv.ffdf()`.

With `ff`, you get various functions for computing summary statistics efficiently from the memory-mapped flat files. These are implemented as generic functions (we will cover generic functions in Chapter **12**), and this means that for most common summaries, we can work efficiently with `ffdf` objects. Not every function supports this, however, so sometimes functions will (implicitly) work on an `ffdf` object as if it was a plain `data.frame`. This means that the full data might be loaded into memory. This usually doesn't work if the data is too large to fit.

To deal with data that you cannot load into memory, you will have to analyze it in batches. This means that you need special functions for analyzing data, and, unfortunately, this quite often means that you have to implement analysis algorithms yourself.

You cannot use `ffdf` objects together with `dplyr`, which is a main drawback of using `ff` to represent data. However, the `dplyr` package itself provides support for different back ends, such as relational databases. If you can work with data as flat files, there is no benefit for putting it in databases, but large data sets are usually stored in databases that are accessed through the Structured Query Language (SQL). This is a language that is worth learning, but beyond the scope of this book. In any case, `dplyr` can be used to access such databases. This means that you can write `dplyr` pipelines of data manipulation function calls; these will be translated into SQL expressions that are then sent to the database system, and you can get the results back.

With `dplyr`, you can access commonly used database systems such as MySQL (**`www.mysql.com`**) or PostgreSQL (**`www.postgresql.org`**). These require that you set up a server for the data, though, so a simpler solution, if your data is not already stored in a database, is to use LiteSQL (**`https://en.wikipedia.org/wiki/LiteSQL`**).

LiteSQL databases sit on your filesystem and provide a file format and ways of accessing it using SQL. You can open or create a LiteSQL file using the `src_sqlite()` function:[2]

```
iris_db <- DBI::dbConnect(RSQLite::SQLite(),
                          path = "iris_db.sqlite3")
```

and load a data set into it using `copy_to()`:

```
copy_to(iris_db, iris, temporary = FALSE)
```

Of course, if you can already represent a data frame in RAM, you wouldn't usually copy it to a database. It only slows down analysis to go through a database system compared to keeping the data in memory—but the point is, of course, that you can populate the database outside of R and then access it using `dplyr`.

Setting the `temporary` option to `FALSE` here ensures that the table you fill into the database survives between sessions. If you do not set `temporary` to `FALSE`, it will only exist as long as you have the database open; after you close it, it will be deleted. This is useful for many operations but not what we want here.

Once you have a connection to a database, you can pull out a table using `tbl()`:

```
iris_db_tbl <- tbl(iris_db, "iris")
iris_db_tbl
## # Source:   table<iris> [?? x 5]
## # Database: sqlite 3.38.0 []
##    Sepal.Length Sepal.Width Petal.Length
##           <dbl>       <dbl>        <dbl>
## 1           5.1         3.5          1.4
## 2           4.9         3            1.4
## 3           4.7         3.2          1.3
## 4           4.6         3.1          1.5
## 5           5 3.        6            1.4
## 6           5.4         3.9          1.7
## 7           4.6         3.4          1.4
## 8           5 3.        4            1.5
## 9           4.4         2.9          1.4
## 10          4.9         3.1          1.5
## # … with more rows, and 2 more variables:
## #   Petal.Width <dbl>, Species <chr>
```

and use `dplyr` functions to make a query to it:

```
iris_db_tbl %>% group_by(Species) %>%
```

```
  summarise(mean.Petal.Length = mean(Petal.Length, na.rm =
TRUE))
## # Source:   lazy query [?? x 2]
## # Database: sqlite 3.38.0 []
##   Species    mean.Petal.Length
##   <chr>                  <dbl>
## 1 setosa                  1.46
## 2 versicolor              4.26
## 3 virginica               5.55
```

Using `dplyr` with SQL databases is beyond the scope of this book, so I will just refer you to the documentation for the package.

Manipulating data using `dplyr` with a database back end is only useful for doing analysis exclusively using `dplyr`, of course. To fit models and such, you will still have to batch data, so some custom code is usually still required.

Exercises

Subsampling

Take the data set you worked on the last two chapters and pick a subset of the data. Summarize it and compare to the results you get for the full data. Plot the subsamples and compare that to the plots you created with the full data.

Hex and 2D Density Plots

If you have used any scatter plots to look at your data, translate them into hex or 2D density plots.

Footnotes

1 This is not entirely true; it is possible to make mutable objects, but it requires some work. Unless you go out of your way to create mutable objects, it is true.

2 You will have to install the package `RSQLite` to run this code, since that package implements the underlying functionality.
