The following Recipes are covered by this Chapter:

* Installing and loading graphic packages;
* Using ggplot2, plotly and ggvis;
* Making plots using primitives;

# Introduction

On our way to make sweat graphs and deliver kick-ass data visualization first I want to share some context. This book was made when R was under the 3.4.0 version. RStudio, a very popular R IDE (Integrated Development Enviroment) was used under the version 1.0.143.

By the time you are reading this book both R and RStudio may or may not be already updated, but still, all the main functionalities we are about to navigate through are must likely to still be working just the same way. For regular R users that are already conversant with R and RStudio please skip to the next paragraph. Newcomers please, suit up, download the latest version of R and RStudio before proceeding.

Why to visualize data? The very only reason you are reading this book should tell me there is no need to convince you about its importance, guess what, call me stubborn. There are a couple reasons you must want to visualize data, but mostly you fit in one of two (or both).

**Reason one:** you are conducting an exploratory/descriptive data analysis and some insight are welcomed. Here we have two things:

A – Graphs can really delivery the amount of insight needed.

B – Never underestimate the power of a well done descriptive analysis.

**There is reason two:** you own the insight, you have the story and you need to communicate it the best possible way – for data driven insights graphs usually fit the category of best possible way. When done the right way, graphs can grab and hold audience’s attention and tell in only a couple of seconds just what is need to be taught. Graphs are great efficient storytellers.

Now you may be asking, why should I use R? By definition R was made for it, it’s a language and environment for statistical computing and graphics. It’s also very easy to learn, minimal programming background is required. But that is not all.

R is open source, very versatile and there is a great and friendly community. A quick online search will prove me on this, R’s community is always willing to help. This reason is also a tip, whenever you got a problem check stack overflow and you must likely got aid.

Community is also improving R itself by making and sharing tons of packages. This own book explores a bunch of them that made themselves proven to be very useful, conquering huge popularity among users. You can write packages yourself, https://cran.r-project.org/manuals.html gives you a manual on how to do it.

This chapter covers basic aspects of ggplot2, plotly and ggvis graphical packages. Beginning with the installation and loading, it goes though framework to some basic examples done by using ggplot2’s primitives, with related ggvis and plotly codes.

# Install and load graphics packages

Before start there are some habits you may want to keep up in order to keep improving you R skills. First of all, when you are programming you may face some challenges, and tackle then either by outthinking the problem or doing some research. You might want to remember what the problem was about and how did you solved, be that for times you face it again later or even for studying hours.

Speaking for me, making a library-like folder gathering some commented examples on those problems and resolutions was and still is of great help. Naming files properly and taking good use of comments (# are used to assign comments with R) makes the revision so much easier.

R Markdown documents are pretty useful if want to keep a track on your own development and publish for others to see. Publishing the learning process it’s a good way to self-promote. Also keep in mind that as a programming language, tasks can be done correctly by multiple approaches, set your mind open into learning new ones.

First things first, in order to make good use of a package you need to install the package and know how to call a package`s function. With you are already conversant with the process of installing, loading, and calling an specific function linked to certain package you may like to jump ahead to the next section – **Using ggplot2, plotly and ggvis**.

Most of the book covers three graphic packages – **ggplot2**, **plotly** and **ggvis**. In order to install a new package you can type the function **install.packages()** into the console. That function works for packages made available into CRAN-like repositories and local files. In order to install packages from local files setting you need to name more than just the first argument, entering **?install.packages** into RStudio will show you a documentation about the function on the help tab.

All the packages covered by this book are widely popular CRAN-avaliable packages, so the first argument only we are naming. **Plotly** recommends using the developer version of **ggplot2**. The dev version is available at Hadley’s GitHub, an useful package called **devtools** can download and install right from git repositories.

If there is an R Session running for long time now there is a good chance a bunch of packages are already loaded. It’s a good practice to restart R so the installation want mess with related loaded packages. **Recipe 1.1** takes care of the installation.

## How to do it…

**Recipe 1.1 – Installing packages**

> install.packages(c(‘devtools’,’plotly’,’ggvis’))

> devtools::install\_github(‘hadley/ggplot2’)

## How it works…

By running the recipe all the packages (**devtools** included) covered by this chapter might already be properly installed, check the **Packages tab** into your RStudio application (speed up the search by typing into the search engine), if everything went fine theses four may be shown under **User Library**. Other than that you may first check the spelling and internet connection.

This function also gives some outputs that stands for warnings, progress reports and results, “look for package ‘<Package Name>’ successfully unpacked and MD5 sums checked” into your console to make sure it was all right. Checking the output is a good practice in order to know if the plan did work, it also give good clues into the troubleshooting.

You may want to call a non-existing package (be creative here) and a package already installed and see what happens. Sometimes incompatibilities avoid the properly download and installation. For example, missing Java or the proper architecture of Java prevents you from installing rJava package.

Perceive that package’s name must be in the string format in order to work (remember to use ‘ ‘). It’s also important to check the spelling, the function (calling and arguments) is case sensitive, miss one letter or case and will not find the desired package.

Also notice that the arguments where drew into a **c()** function. That is a vector (try ?c in the console). As first argument of the **install.packages()** function, a vector of strings were given. That said, multiple packages can be downloaded and installed simultaneously. That same function might not install only the packages asked, but all the packages each of them rely on.

Installation is also possible via **RStudio features** – that may seen more user friendly for newcomers. Open you RStudio, go for **Tools > Install Packages…**, type packages names (separate them with space or comma) and hit install. It fills the **install.package()** function and shows it in your console.

This is most indicated when you are not absolutely sure about the package name, but have a good clue, it gives you suggestion on what you are typing. You can also install packages from local files by using this feature. Look for an option called Install from and switch it to **Package Archive File** instead of Repository.

RStudios also gives you a **Check For Packages Updates…** option right below **Install Packages…** Hit it once a while to make sure your packages are properly updated. Along with the packages to be updated it shows what is new about it.

Once the packages are installed you have a bunch of new functions at your disposal. In order to get to know this functions you can seek packages documentation online. For CRAN-avaliable packages, documentations can be found very easily, they also follow standards that are very helpful.

Now with a bunch of new functions at hand the next step is to call a function from an specific package. There are several ways of doing that. One possible way is to call using **<package name>::<package function>**, **Recipe 1.1** done that when called the install\_github() function from devtools (**devtools::install\_github()**).

There are pros and cons about calling a function that way. For pros you mostly avoid any name conflict that could possible happen between packages. Other than that, you also avoid loading the whole package when you only need to call one function. Calling a function that way may be useful in two occasions: there is a name conflict or only few functions from that package may be requested and only a few times.

Otherwise, if a package is required many times, typing <package name>:: before every function is anti-productive. It’s possible to load and attach the whole package at once. Via RStudio interface, right below the window that shows environment objects, there is a window with a package tab. Down the package tab it’s possible to check the box in order to load the package and uncheck to detach them.

Try to detach **ggplot2** by unchecking the box and keep one eye on that box. You can load packages using functions. The **require()** and **library()** functions can be assigned to this task. Both don’t need ‘ ‘ in order to function well, but if you call the package name by a string stills works. Notice that both functions can only load one package a time.

Although **require()** and **library()** works in a very similar way they do not work exactly the same. By miss using **require()** it throws an warning, against an error thrown by **library()**. Also **require()** returns a logical value that stands for TRUE when the load success and FALSE for fails – **library()** returns no value. For common loading that is not a difference that should made into account, but if you want to create a function or loop that depends on loading a package and checking if it succeed, you may find easier to make it using **require()**.

Using the logical operator **&** (and) it’s possible to load all three packages at once and store the result to a single variable, calling this variable will state TRUE if there is success for all and FALSE if just one fails. This is done by Recipe 1.2.

## There is more

**Recipe 1.2 – Loading and Logical Check**

> lcheck <- require(ggplot2) & require(plotly) & require(ggvis)

> lcheck

For now you might be able to install R packages – either locally or CRAN-like repository located – load and call a functions from a package. Now that you are familiarized with R packages installation and loading procedures the next section gives an introduction to the **ggplot2** package framework.

# Using ggplo2, plotly and ggvis

Among the vast universe of R graph packages, **ggplot2** might be the most popular, between 2012 and 2017 it had been download over 10 million times. Authored by Hadley Wickman and Winston Chang, **ggplot2** brought a whole new way to brew graphs using R. As consequence of its popularity, it works along with several other graph packages – this is the case of **plotly**.

The “gg” from **ggplot2** stands for Grammar of Graphics, which is the very theory that underlies the package. There are several advantages related to the use of ggplot2 instead of base R base graphical functions.

The package brings flexibility to the graph brewing process, meaning that when circumstances change, new discovers come out, it’s not needed to change much of the code to bring new information, a much appreciated virtue by data science. Also **ggplot2** is very mature – 10 years old since June 2017, meaning it’s stable – and holds an active mailing list.

As great as it sounds there are some limitations, as the creation of interactive graphics should not be done by using only **ggplot2**. That one is easily overcome by combining it with **plotly** package, more on that later.

A bunch of graphing tasks could easily done by using only the ggplot2 package. That would be:

* Drawing scatterplots, boxplots, violin-plots, maps and more;
* Faceting, scaling and adding a theme;
* Drawing regression and smooth lines;
* Modify and add texts and labels;
* Grouping variable by the adoption of symbols and colors.

For people who already knows some of Hadley’s works, it’s not difficult to recognize his packages for making solutions much easier, intuitive, elegant and efficient – for example, check dplyr package.

The same applies to **ggplot2**. The core idea behind it is to build graphs layer by layer, think of it as playing lego or making a cake. Each block or layer can come from a different dataset or hold different aesthetic mapping, the process of building graphs become by that way iterative.

Each layers either adds, modify or remove one or more of the five main graph components: the dataset, aesthetic mapping, geometry object, statistical transformation and position adjustment.

Some authors might consider scales, coordinate system and faceting as different components from the ones just cited, others would include within one of them. Truth is, they are also easily manipulated by **ggplot2**.

In order to use **ggplot2** plot functions is necessary to have data stored into a data frame. Every layer must be associated with a data, and this data has to be stored into a data frame. It`s possible to have multiple data sources but each one must come from a data frame type object. That in fact is a strong restriction, but there is some reasons.

First of all, gathering your data into a data frame makes your data to be more explicit. Other than that storing your data into a single data frame other than multiple vectors make it organized, as result, sending your data or reproducing the results get easier too.

Also make a clear cut into **ggplot2** concerns – it take cares of data visualization, that is, turn data frames into visuals, to make data frames into the right format is a task for model visualization packages. It’s important to outline that there are several ways of combining multiple vectors into a single data frame.

## Getting ready

That said, in order to get started with ggplot2 we must adopt either one or more data frames. R come along with base data sets. To get a brief introduction to all of them you can go into the console and type **library(help = ‘datasets’)**.

The output gives you the information on the package datasets. This information comes with an index about all the data sets R Core Team made available. Using one of this data sets spares you time and health as there is no need either to download or import data.

One small tip on importing data, specifying the **colClasses** parameter allows **read.table()** and **read.csv()** to work faster as it doesn’t have to check each column’s classes. Now pick one data set and substitute its name into **<Data Set Name>** in the following code:

> ?<Data Set Name>

> class(<Data Set Name>)

> head(<Data Set Name>)

Notice that datasets is a base package, so there is no need to load it before using its data sets, you might need to call the package name instead if there is a name conflict but I’m assuming we start this chapter with a clear R environmen. If you feel afraid about name conflict or want to make explicit where this data come from replace **<Data Set Name>** for **datasets::<Data Set Name>** into each line.

First line gives the documentation on the set you picked, what kind of data the set brings and its origins are shown by documentation. Second line returns the class type of the object in the point of view of object oriented R programming, it’s important to check if the class type is really a **data.frame**, some sets are time series (**ts**) type, calling for some modification before used into **ggplot()**.

Last line might give you the first six observations if you not had altered the default second argument. You can set the number of observations shown by setting an integer as second argument.

Let’s give the cars dataset a try. Class function shows that cars is a data frame, ?class shows the data come from the distance to stop a car given the speed. Data was collected at the 1920s hold two variables – speed (mph) and stopping distance (ft). **Ggplot2** allows to save plots into objects, and calling them in order to display the plot itself.

**Recipe 1.3 Part 1/3 – Getting Ready to Plot**

> ?cars

> class(cars)

> head(cars)

> library(ggplot2)

> plot1 <- ggplot(cars, aes(x = speed,y = dist))

Notice that you could call **plot1** into your console but it would display an empty graph once there is nothing there to show yet. As the first argument of the **ggplot()** function we stated the data to be used – it has to be a data frame. The second argument it’s **aes()**, that stands for aesthetic mapping.

The second argument calls for arguments, the first one is the **x** variable, second one is the **y** variable. For these first arguments there is no need to name variables, it’s possible to just call them by the right order. Some geometries don’t need the **y** variable, as happens to the box plot that only calls for **x**. Different arguments may be required for specifics plots, in this case naming is mandatory.

Checking **?aes()** shows an “. . .” as argument, popular known as three-dots, is technically named ellipsis and allows the user to pass an arbitrary number and variety of arguments, also allows to pass arguments on to other functions. So as R does lazy-evaluation – only evaluate arguments as they are requested – you could make up imaginary arguments and pass into the **aes()** function with zero or only little trouble to the function. Perceive that:

> plot1 <- ggplot(cars, aes(x = speed,y = dist, gorillaTroubleShooter = T, sight = ‘Legolas’))

Would work as good as the last version, just don’t forget to name the arguments and you got yourself a good way to easter eaging your code. Both **aes()** and **ggplot()** plays a core role at building graphics with **ggplot2**.

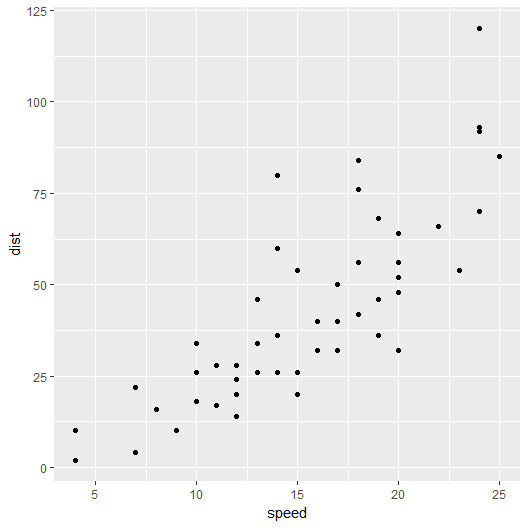
In order to get the graph plotable, a geometry is required, that means adding a new layer. As the data correspond to two continuous variables let’s set a scatterplot. **Ggplot2** deals in a very intuitive manner they layer adding operation it uses the **+.gg** operator. By simple using + after the **ggplot()** the user adds a new layer.

## How to do it…

**Recipe 1.3 Part 2/3 – Adding one layer**

> plot1 + geom\_point()

This code outputs a scatterplot with the two variables, speed and distance. Figure 1.1 displays a representation on the graphic displayed by Recipe 1.3 (2/3).



[B08014\_1\_1.png]

Figure 1.1 – Plotting points using ggplot2.

## How it works…

On behind scenes the **geom\_point()** function called the **layer()** function with a couple of default arguments that culminated on the creation of the scatterplot. One would like to modify axis labels and add a regression line by the plot. It can be done by simply adding more layers to the plot.

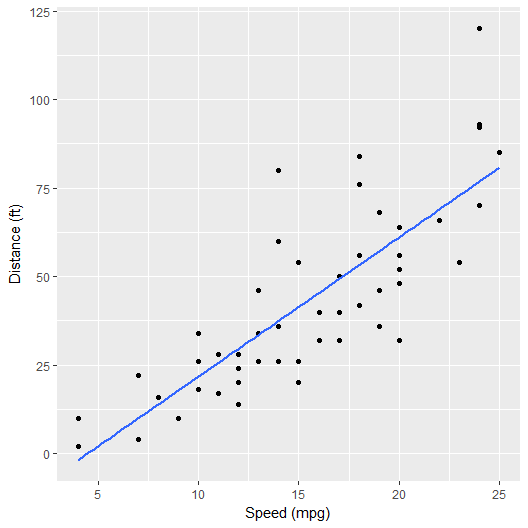
**Recipe 1.3 Part 3/3 – Adding three more layers**

> plot1 + geom\_point() +

> labs(x = "Speed (mpg)", y = "Distance (ft)") +

> geom\_smooth(method = "lm", se = F) +

> scale\_y\_continuous(breaks = seq(0, 125, 25))



[B08014\_1\_2.png]

Figure 1.2

Into part 3 of the **Recipe 1.3** the second line modify the axis labels. Second line adds a linear regression line while the third one force the Y axis scale to remain still as the addition regression line may change it.

## There is more

**Ggplot2**’s addition play a central role at the package. Graphics can be build or enhanced step by step. **Ggvis** also work in a very similar to **ggplot2**, relying on Grammar of Graphics, summoning one of them shall be easier if there is familiarity with another. **Ggplot2**’s **aes()** and **ggplot()** could be translated respectively into **props()** and **ggvis()** at **ggvis** package.

Although there are similarities it`s important to outline that are things that can be recalled as clear upsides or downsides linked to the usage of one over another. Other things are just different in both packages, neither better nor worse.

For instance, **ggplot2**`s addition operator may be very intuitive – user is adding layers – alternatively **ggvis** apply iterations by using the pipe operator (**%>%**). Pipe operator is very popular among R users, so many may be already familiar to it, it`s only a matter of preference – at the end, coding both will look very similar.

Other than that, **ggvis** is a lot younger than **ggplot2**, still lacking the maturity achieved by the last. So, older means better? Of course not, it’s only a matter of preferences again, younger means that many features are yet to come and becoming disruptive is also a possibility.

By the version 0.4.3, **ggvis** did not support facetting the way **ggplot2** did. When the task is to build interactive graphics, **ggvis** can deal with it by itself, bringing along the reactive framework of shiny package and the drawing web skills of vega. As standalone packages, one could elect **ggvis** as a more powerful tool for building interactive graphics and dashboards while **ggplot2** takes the heads at faceting. Also important to outline that **ggvis** does not stick with the same data type restrictions shown by **ggplot2**.

Each package may not be seen as an isolated planet at the R universe, but systems with several interdependencies and interactions among them. Such is that building interactive graphics is made possible at **ggplot2** by the **plotly** package. **Plotly** is named after its web application, and what the package does is to enable building interactive web graphics direct via R, translating **ggplot2** graphics into plotly`s interactive web graphics is specially easy.

Taking from where we stopped, the following routine gives an interactive version of the first graphic in this chapter.

**Recipe 1.4 – Translating ggplot into plotly**

> library(plotly)

> ggplotly(plot1 + geom\_point())

The plot displayed is interactive, try zooming in by dragging the mouse and zooming out by double clicking. Some malfunctions can be avoided by ensuring to have the developer version of **ggplot2** installed.

Calling **ggplotly()** with no arguments shall create an interactive version of the last **ggplot2**’s figure made, no matter if any other kind of plot was made right before – try calling **plot(cars$dist, cars$dist)** right before calling **ggplotly**.

The package brings a series of functions to brew figures by itself, interactions are made through pipe operator (**%>%**) and a series of examples and tutorials can be found at plotly website (plot.ly/r).

**Plotly** highly relies on Java to run, check your Java version, compatibility with system’s architecture is also an important issue. Using RStudio won’t require neither internet connection nor a registered plotly`s account, figures appear interactively into the graphics viewer, but having those two enables the user to upload graphics into plotly’s cloud – a registered account, internet connection and API key are needed. It’s also possible to sing up for plotly by using the **singup()** function. Important to outline that plotly works along with **shiny**, supporting dashboards.

Those three packages – ‘ggplot2’, ‘ggvis’ and ‘plotly’ – are well coded and powerful graphics packages. Right before tackling a task using them ever consider both what the package is able to do and what you are able to do with it. Acknowledge package’s limitations and strengths, account both what you already know and what you need to know. Being vigilante for improvements on your coding skills no matter how good you already are is a very good way to achieve success.

# Making plots using primitives

Above, a brief introduction on the framework of those three packages were conducted, driving through the framework of each one, highlighting pros, cons, limitations and strengths commonly linked to each one. Next we are getting started into ggplot2 graphical primitives, displaying them in a series of recipes with related examples made with ggvis and plotly.

There are a total of eight graphical primitives at ggplot2, one of them already used by this chapter. It’s important to know well the primitives, what they do and when to use. As fundamental building blocks they play an essential role at the drawing process.

The complete list is given by geom\_: blank(), path(), ribbon(), polygon(), segment(), rect(), text(), point(). Notice that every primitive starts with geom\_ but not every geom\_ is a primitive. In fact, the better chances stands for the opposite.

More or less, geom\_blank() seens to be the simpler of the primitives, calling for it right after setting ggplot() will display a blank plot, with axis already adjusted. It’s mostly used to check axes limits given by data itself, but that can be done by plotting it right away. Maybe you can find it useful for another task, suite yourself.

Other primitives may work in a similar way. That is the case for geom\_path(), geom\_ribbon() and geom\_polygon(). The first one draw lines between coordinates, second one looks like the first, but thicker requiring additional aes() arguments (ymin and ymax). For the last polygons are draw in somewhat similar way geom\_path(), but connecting starting and ending points plus filling the inside.

By setting only starting and ending point geom\_segment() adds a segment line. geom\_rect() adds a rectangle to the plot, requiring the four corners to do so (xmin, xmax, ymin, ymax). geom\_text() adds given text to given coordinates, some plots would display only texts for each observations instead of points.

The remaining primitive is geom\_point(). It’s the only primitive direct called so far, displaying points by coordinates given. Two important points must be highlighted. One: getting to know the primitives might give an idea on which function you will require the most and what the least but that is not all that ggplot2 is capable of, only the building blocks used by other functions.

For the second point notice that they can be combined. It’s important to get to know how the functions interact. **Recipe 1.4** displays the result given by the combination of two primitives.

## How to do it…

**Recipe 1.4 – Combining two ggplot2 primitives**

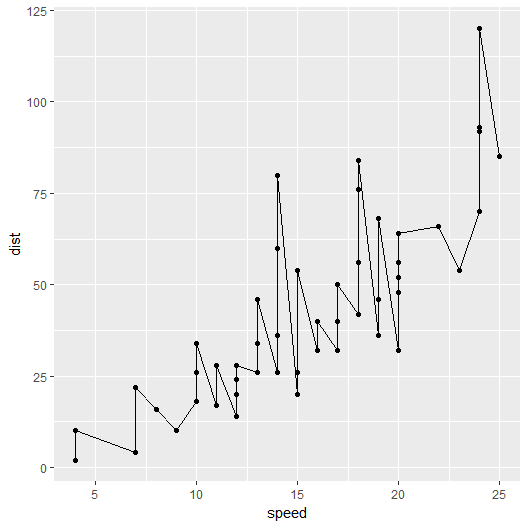
> library(ggplot2)

> plot1 <- ggplot( cars, aes(x = speed, y = dist))

> plot1 + geom\_point() + geom\_path()

## How it works…

As result Figure 1.3 is outputted. Note that there is both points and paths between theses points. One could try calling the same come with + geom\_blank() to its end, nothing changes.



[B08014\_1\_3.png]

Figure 1.3

## There is more

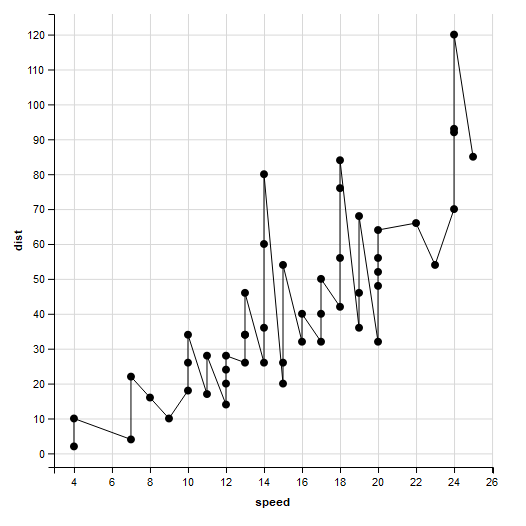
Two primitives were used, while geom\_point() took care of plotting the dots geom\_path() connect each one. In order to create a very similar plot using ggvis type the code below into your console:

**Recipe 1.5 – Related ggvis code**

> library(ggvis)

> ggvis(cars, x = ~speed, y = ~dist) %>% layer\_points() %>% layer\_paths()

Earlier section argued that ggvis is very similar to ggplot2 in the way to code graphics, this section actually showed that. First function get the data set, no aes() argument used. Pipe operators (%>%) are used instead of plus sign and layer\_ works in a very similar way geom\_ does. Figure 1.4 displays a representation of the resulting graphic.



[B08014\_1\_4.png]

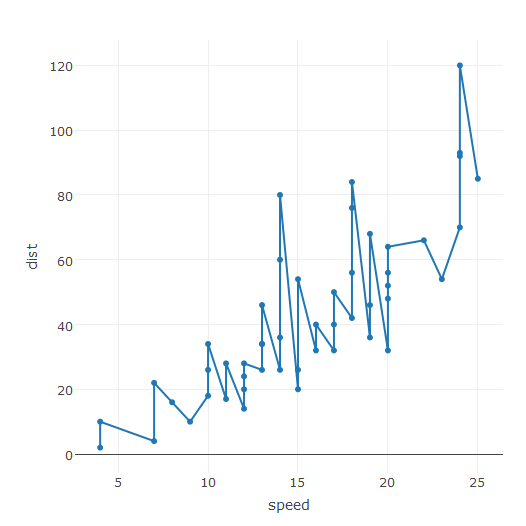
Figure 1.4

Recipe 1.5’s results is displayed by viewer tab once is interactive. It can be exported as a web page. Note that ggvis plot carry a different theming from ggplot2. Themes can be changed as we are seeing in details later in the book. Without using the translation function from plotly package, it’s also possible to code the same graphic from scratch.

Recipe 1.6 – Related plotly code

> plot\_ly(cars, x = ~speed, y = ~dist, type = ‘scatter’, mode = ‘lines+markers’)

Building this graphic into plotly is not so layer like does ggplot2 or ggvis. Although plotly does have layers integrated by pipe (%>%) it would be required only if we were willing for more complex graphics. Figure 1.5 is a snap shot of the graphic brewed by the Recipe 1.6.



[B08014\_1\_5.png]

Figure 1.5

There are some important points to remark such the theming and interactivity. Plotly carries a theme of its own. Translating ggplots into plotly’s would coerce the theme used by ggplot into plotly’s. Note that it’s only a representation of the output as the outputted graphic is interactive and it’s displayed into the viewer.

This section so far aimed to show how to construct plots using ggplot2’s primitives, and build similar graphs using other packages. A question you should ever ask yourself is if the geometry adopted goes along the data used. In other words, if the graphic tells the story that you are willing to tell.

Figures 3 to 5 five resembles much a time series, but it is not and may give the wrong intuition. These are two variables observations, check how for some speeds there are up to 4 different distances to stop. Notice that the cars data frame is ordered first by speed and secondly by distance, paths obey the order showed by data while for points it does not matter.

Adding the path may mislead the audience, geom\_point() were enough. For the next recipe let’s build fictional data but draw a graphic that tell the story the right way. Picture a small classroom, with only 7 students, the teacher built a data frame with studying hours and grades of each student.

First we are creating 3 vectors that stands for names, studying hours and grades of each student. Right after that we are coercing the vectors into a data frame.

**Recipe 1.7 – Creating fictional data frame**

> allnames <- c(‘Phill’,’Ross’,’Kate’,’Patrice’,’Peter’,’James’,’Monica’)

> classr <- data.frame(names = allnames)

> classr$hours <- c(4, 16, 8, 11, 6, 14, 8)

> classr$grades <- c(4, 9.5, 6, 4, 6, 9, 7.5)

First line creates a character vector with the name of all students. The second line coerce this single vector into a data frame called **classr**, note that argument name stands for column name. The following lines creates the hours and grades variables direct into the data frame, the $ sing separates the data frame name and the column name.

Coercing data into data frames is a ggplot2’s requirement. Storing ggplot() into an object it’s not a requirement but make the code more readable. Next steps do so and call a primitive right after. Pay attention to the new aes() argument demanded by geom\_text() primitive.

**Recipe 1.8 – ggplot2 classroom plot**

> library(ggplot2)

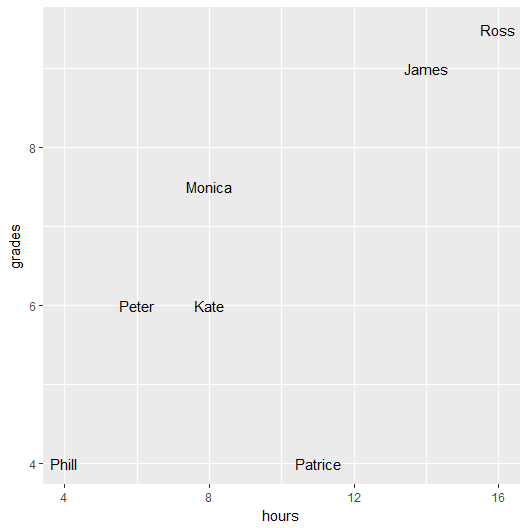
> plot2 <- ggplot( classr, aes(x = hours, y = grades)

> plot2 + geom\_text( aes( labels = names))

Make sure to run **Recipe 1.7** before running **1.8**. Instead of using points, the resulting figure displays the coordinates student’s names at the given coordinates. The story is told in a good manner such as the audience might understand it right away, with no confusing signals. The graphic displayed is similar to **Figure 1.6**.

You could try adding **+ geom\_path()** to the end of the last code of line. By checking the result you may get a better understanding on what order the paths obey (same as data input). Also you shall notice that adding paths to this graphic clearly messed up the understanding, this geometry holds no meaning to the data used.

Notice how well the data was represented by the plot. Worked because there are only few observations and no two share the same position. If there are a bunch of observations really close to each other you may consider using points instead, or using text to a few observations do you want to highlight. Even points can be misleading if there is over-plotting but there are solutions available – more on that later.



[B08014\_1\_6.png]

Figure 1.6

Achieving a very similar result is also possible by using either ggvis or plotly. Follow the code showed by Recipe 1.9 to get it.

**Recipe 1.9 – Similar ggvis and plotly graphics**

> library(ggvis)

> ggvis(classr, x = ~hours, y = ~grades, text := ~names) %>% layer\_text()

> library(plotly)

> plot\_ly(classr, x = ~hours, y = ~grades, type = 'scatter', mode = 'text', text = ~names)

Second line brew a ggvis plot as the last goes for plotly’s. The appearance of both could be enhancing declaring some additional parameters that set font size for example. This section’s goal was to introduce you to the graphical primitives of ggplot2 and draw simple graphics by using only them.

Additional goal was to draw related graphics using ggvis and plotly packages. Next chapters are going deeper, each one shall tackle specifics families of graphics, highlighting nuts and bolts in the way to building high quality graphics. As the book advances also does the complexity involved. At some point we are going to be plotting interactive middle earth maps, I find it pretty sicking cool, hope you enjoy it.