Berlin Exchange medicine

Inferential Statistics II – Linear Regression via the Linear Least Square Method in R

Absolute Beginner's Stat-o-Sphere

by Steffen Schwerdtfeger

Medical Student, Charité Universitätsmedizin, Berlin
BEM Editor – Data and Statistics / Medical Humanities

Berlin Exchange Medicine e. V.

URL: https://journal.medicine.berlinexchange.de/pub/linreg

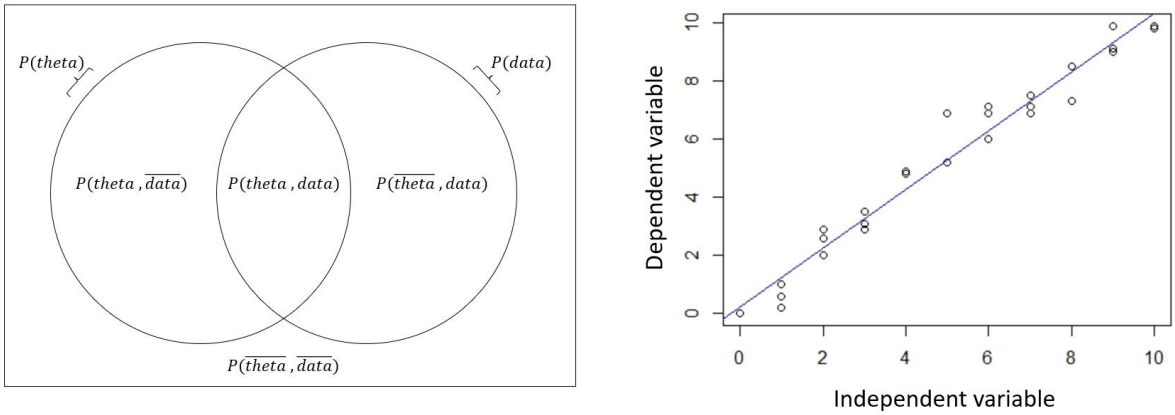
License: Creative Commons Attribution 4.0 International License (CC-BY 4.0)

Corresponding R script: https://assets.pubpub.org/i8ctxf3z/61660496136616.R

Introduction and Recap of Inferential Statistics I

In the *first* part of our tutorial series on inferential statistics we explored the general idea of science as iterative hypothesis testing in its conceptual and intuitive, as well as its most basic mathematical form: conditional probability / Bayes' theorem (essentially used as a 'learning algorithm' in science, representing the inference made and its course). We have also learned about the difference between the frequentist and the Bayesian interpretation or usage of Bayes' theorem — and that the p-value is essentially a likelihood $P(data|\overline{theta})$, but involves a little more than that, such as a t-test reflecting on the data (difference in the mean, given a bunch of assumptions on sample size etc.). Note that the *second* and *third* part of this series of tutorials will rather focus on a frequentist approach to statistics, but we will catch up with *Bayesian linear regression models* in the future, eventually comparing the two methods under previous and more specific considerations.

Before we fully get to linear models as the subject of *this* tutorial, **let us first recall** that hypothesis testing (t-test, Chi-squared…) in the mathematical sense not only refers to plain conditional probability, but implicitly also to methods for **obtaining a *probabilistic model* from a *non-probabilistic model***, such as a linear model — as the latter consists of measurements and *not* probability values by itself. Also recall that the term 'model' in general just means "relation". Therefore a "non-probabilistic" model can also be looked at as a "joint" in terms of a *relation* — a relation between an ***independent*** and a ***dependent variable***. A *linear relation* is the most basic form of such "non-probabilistic" models, mathematically expressed as a *linear function* (we will get there in detail below).



Probabilistic model *Linear model*

Fig. 1 A probabilistic and a linear model side by side. On the **left** we have conditional probability, a relation between theta and data in this case, visualized via a Venn diagram. On the **right** we have a linear regression model, represented as a simplified graph of the relation between an *independent* (x-axis) and a *dependent* variable (y-axis). We have learned what a probabilistic model is in the context of conditional probability; in this tutorial we are going to focus on linear regression models.

However, every model is typically used to do predictions (either via inferring on the probability of an event or in the sense of estimating values of variables). As you might see, the graph of our linear model indicates that these predictions will not be perfect, otherwise

every data point would be located on the graph itself (which ***is*** our model). We will see that the linear model above is a compromise, trying to do the "least number of mistakes when making predictions", done by cautiously drawing the line going through the data. Mathematics will eventually help us to decide where to *optimally* draw such a line.

So, this time we are essentially going to focus on step II (gathering data / formulating a model) of the three steps of hypothesis testing, but under a *non-probabilistic* perspective first. We will again use R to plot data and calculate the results. This time we will go in full cyber, beyond simple arithmetic exercises, and will also start with some plotting, whoop! We will also fully introduce R in this tutorial, so in case you missed it, it is not mandatory to have read Inferential statistics I for the R part.

In case you are worried: you have still entered the beginner´s sphere, so no relevant prior knowledge should be required and we will also work with a certain degree of redundancy in order to avoid turbulences on your journey through our tutorial.

Note that some chapters, e.g., on the decomposition of some formulas can also be considered optional side-quests, if you feel you don't want to go through all the details. As before, we have also provided a short summary at the end of every chapter — for the rather impatient reader or for those seeking orientation in the text. We still wanted to give you the chance to follow every step taken in the math without missing links and used the opportunity to introduce R as an abstraction of mathematics in general (at least for our first series of tutorials for beginner's).

Mathematical abstraction? Doesn't that sound kind of redundant, a little dizzy? N o w o r r i e s ! Mathematically we will mostly just recall and especially *rediscover* some basic math from our time in school, when going through the math of linear modelling (the essence is a plain linear function). We hope that this tutorial will leave you as surprised as we are, considering that we actually possess nearly every tool to solve our problem of drawing an optimal line through data already.

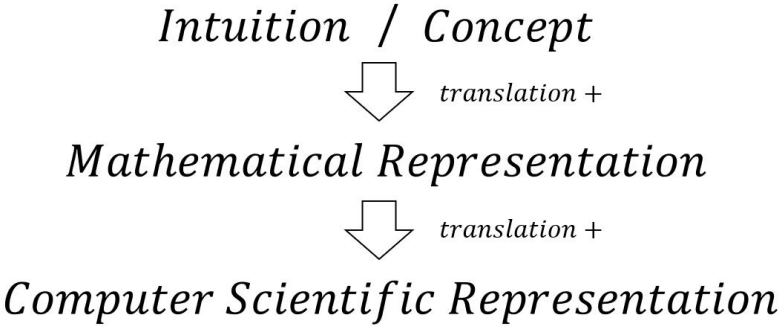


Fig. 2 We will again apply the upper triadic structure when introducing you to linear regression analysis, arguing that the intuition, the math and the code are more or less three languages for the same.

The structure of this tutorial will be as follows: As a first step into the sphere of linear modelling we will gain a stable grip on two already mentioned terms: the infamous, but mostly just misunderstood concept of *independent* and *dependent* variables. After that we

will look at simple linear functions, given idealized data points that are located on the linear function already. From there we will dig in deeper into the math and will apply the *linear least square method* on further non-idealized synthetic data sets. We will do so both mathematically, as well as computationally. In order to give you a general understanding on R as a programming language, we will also write our own `linear_least_square()` function doing the mathematics of a linear regression and will eventually compare the results of our function with the built in `lm()` function. All of the above will be the preparation needed for the *third* part of this tutorial series that will go through the math of the output of the `summary()` of the `lm()` function (as said, including F-statistics, p-values etc.).

1 Dependent and Independent Variables

The distinction and definition of dependent and independent variables is — even with some experience — not always a trivial task. Nevertheless, we are going to see that there are "algorithmic ways" to clearly orientate within both the terminology and the process of defining or evaluating the dependencies of actual variables.

To get some intuition, we will start with a simple example. Let us say our life is accompanied by **a doggo**. We want to please our dog and we have experienced that our dog had "health problems" recently, so we thought about checking if the dog *food* is causing the problems (influencing the dog in some way). We could now check on the weight, take a blood sample etc., but for simplicity we will just check, if the dog is showing signs of "*feeling good*", or "*feeling not good*" — his mood (whatever it may mean). In other words: let us not focus on the details of our chosen variables for now, just on the characteristic relation the variables may have to each other.

Fig. 3 Checking which kind of food is best for the dog. The dog food represents the independent variable, i.e., it is not influenced by another variable, but can influence another variable, such as the dog. The dog's mood itself is considered the dependent variable, dependent on the dog food so to speak (influenced by another variable, but does not influence the independent variable). When the fact that the food is considered independent appears too twisty for you, recall: The dog´s mood itself does not influence which type of food is served in this setting (it is the other way around). The food therefore remains independent. Original photos by Sebastian Coman and "Fatty Corgi".

The specification *dependent* and *independent* can intuitively be understood in terms of the influence that variables have *to each other* — either thought of as ***conditional or causal relation or 'course of inference'***:

- ***Conditional****:* If the dog food is changed, the mood of the dog changes.
- ***Causal***: Changing the dog's food (cause) results in a change in the dog's mood (consequence; similarly discussed here).

Fig. 4 She is right: conditional, as in conditional probability - just without the probability aspect. The relation between independent and dependent can be understood as a course of inference as well, such that the independent variable xx is the set (influential) condition for the dependent variable yy, the variable we observe and try to predict, such that $(Y \mid X)$(Y|X), spoken "the estimate of Y given X". What we are aiming for in this tutorial is also referred to as **conditional linearity**. Original photo by Kevin Mueller

Note that there are experimental settings in which terms or phrases such as "causal/conditional", "influenced by", "changed by", "in relation with" or "dependent on" may appear counterintuitive and / or rises philosophical questions, as time for example measured via a clock as well as the clock itself is *not* what influences an event *in the literal sense* of a cause (time considered as an object that acts upon things (reification of time)). The upper example concerning the dog food and the dog's mood makes more sense in the context of "literal" causal influence, as the dog food is "actual matter" that evidently influences the dog as an organism (in genereal not in a specific way by necessity).

The most neutral way to address a relation between variables is probably by just saying: y changes linearly in correspondence with changes in x. However, reflecting on the relation of variables in such a way is essentially picking up our discourse on gathering evident data and evaluating its evidence in the wider sense (checking on bias, validity etc.).

Apart from correctly describing what is meant by a "relation" of variables, note that the form of dependency is also something not always "objectively" clear, as there might be "*hidden"* variables influencing the situation (e.g., discussed as **proxy** or **confounder**, see

below), leading to further questions that may have to be evaluated in order to get further insight into possible dependencies influencing the results. **In other words: just because one thinks, defines or makes it look plausible by concept / belief that something is independent, it does not actually have to be the case — assuming so is essentially setting a hypothesis.** Note that a measurement that leads to data itself can also be understood as a small experimental design itself, usually evaluated and rated by its precision (of a clock for example).

The task of designing an experiment (gathering data) is therefore always to either find or create a situation, where dependencies of variables can be distinguished as being either independent or dependent to each other, so that no other variable influences the result (there are also multivariate models and other concepts, but we are going to stick with the simple scheme for now). This process again involves the goal of gathering *evident* data (see Inferential statistics I), in other words: to operate with evident variables in the form of data *that represent events in the world* in some way (can be cast as data in the first place). To cut it short: one's own confidence is not data, so we have to find ways to intersubjectively make those experience / data accesible vi an experiment for instance.

Questions concerning the evidence and the epistemics of inference ("what can be known?") are especially present in conducting psychological experiments: — does a certain behavior of a person (dep. var.) really represent the consequence of a *certain* cause (indep. var.)? **However, no matter how far reflections on "relationality" as such have dragged you away, it is always good to recall** that the two terms in question are actually *mathematical terms* and refer to the *characteristics* of a *relation* of variables defining, e.g., the probabilistic influence of one event on the probabilities of another event occurring, e.g., a certain type of dog food (indep.) makes the probability of encountering a dog feeling good (dep.) more likely (probabilistic model), or a rise in time corresponds with a rise in the number of cups of coffee, e.g., in a linear way (non-probabilistic model; conditional linearity).

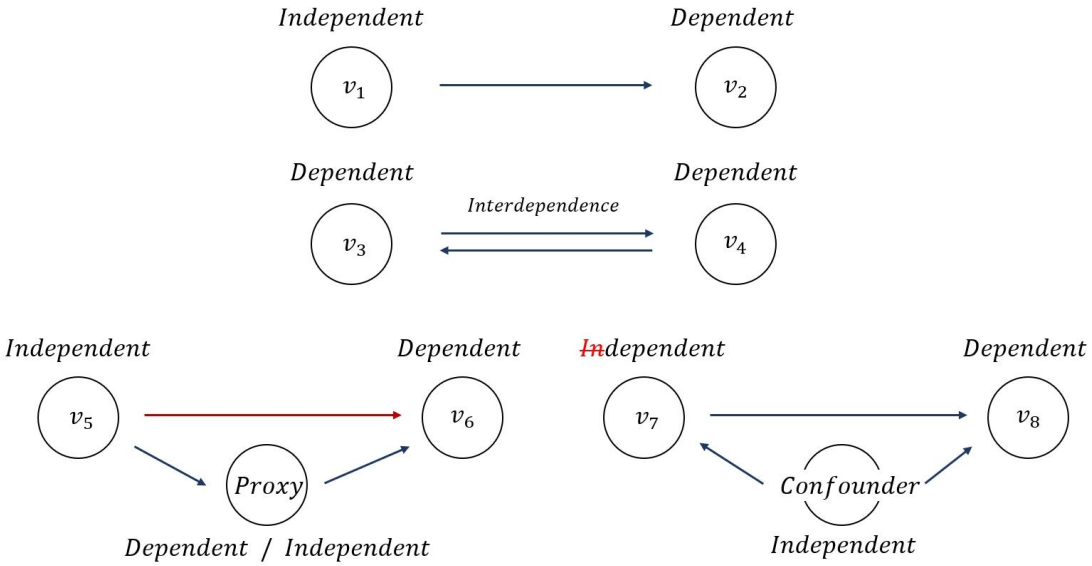


Fig. 5 Generalized graphical reflection on dependencies between variables of interest. **Upper part:** $v1$ is considered independent, as it is not influenced by other variables, such as $v2$, which is influenced by $v1$. The variables $v3$ and $v4$ on the other hand are both dependent, as they are

influenced by each other — and in this special case the relation to each other can be defined as *interdependent*.

Note that seemingly independent variables may be dependent on something unknown or overlooked variable in general, e.g., some bias of the participant, creating an interdependence of some kind, or **confounders.** On the **lower right** the confounder unknowingly influences v7 in a way that the relation between v7 and v8 and the relation between the confounder and v8 may even result in the same. v7 for itself though does not influence v8, but may appear as if, when confounders are unknown. A classic example is smoking as confounder: a study claims that drinking alcohol highly increases the risk of lung cancer, though it *may turn out* that those who drink usually also smoke (often also passively due to, e.g., the pub) and that consuming alcohol for itself does at least not come with a higher risk of developing lung cancer (but may come with a higher probability to be exposed with airborne cancerous substances by context).

Let us now get back to our dog example: Let´s say the dog *owner* appears to be rather happy serving a *certain* kind of food — e.g., due to the branding — then one might even create a ***by proxy effect* (lower left)**: the dog is behaving happy, as the owner appears to be happy when serving a certain kind of food, regardless his own "opinion" or "dependence" on the dogfood (therefore marked red). The owner then becomes the *actual* independent variable related to the dog's mood and also the actual dependent variable in relation to the dog's food. This eventually messes up the setting — *even though we will get mathematical results!* If the dog owner is the person that is conducting the experiment, then we could also interpret the whole setting as a self-fulfilling prophecy. Note that effects like the above are also drivers for any kind of pseudoscience and pseudomedicine.

Another way of reflecting all of the above difficulties in understanding the dependencies of variables is by becoming aware of an implicit twist in the definition of the dependency of variables, entailing *two concepts at once*:

- **I.** The terms dep./indep. address the *characteristics* of the dependencies of variables, e.g., within an experimental setting (as discussed above).
- **II.** The term variable and the *chosen* variables are pointedly defined/set *independently from the definition of the characteristic dependency of any variables.*

The latter also implies the need of an *evaluation* of such a model of the relation of variables in the first place, as we can set a pair of variables and hypothesize on a dependency, but the assumed dependency does not have to be supported by the data / experience. A dependency may be supported by previous observations or plausible for some other reason, but is at first always a mere hypothesis and therefore also set.

In other words: A major difficulty in getting a stable grip on the dependencies of variables relies on the fact that **the dependency of variables will never be something that is *necessarily* given in a variable *itself* (that's why we have to do research in the first place). This is again due to the fact that variables are in general something *we* set and their dependency something *we* hypothesize and evaluate. To give an example:** the variable "dog's mood" itself does not entail information on any possible or in general necessary dependency to another possible variable, apart from those *we* potentially associate the variable with (our hypotheses).

Evaluating the dependency of variables therefore involves a **twofold process**: **a)** *defining/setting* the variables of a model (or finding them in a study design, reading or

reviewing a paper) and **b)** *checking* on their dependencies, *given* a definition of both, variables and the various possible forms of dependency. This twofold process is involved in *both* designing (includes reflections on evidence as such), as well as conducting an experiment and evaluating its results (therefore this process is involved *before* and *after* gathering data within in our triadic scheme of hypothesis testing).

To wrap this up: Operating with the concept of the "dependencies of variables", either when reading a paper or designing and conducting an experiment oneself, should always involve an ***algorithm of some kind*** running through your mind**,** moving *from* the linguistic/mathematical definition in general *to* the set variables *and then to their* hypothesized and evaluated dependencies respectively (this is essentially a deductive process: from the "rule", to the "event"). Such an "algorithm" of evaluating the dependencies of set variables is therefore not "only" an abstract mathematical or mere definitional process, somewhat unintuitive or even robotic. It rather represents what makes up *creating* and *testing* an experimental design in the first place.

In case of philosophical vertigos: One should not overestimate philosophical perspectives on a relation of variables as not everything we experience is just a matter of interpreting a term or language in the first place (its a tool for communication) and it is also not the goal to represent as much detail as possible in the form of language in order to "speak evidently", or so. For science I'd say, it is not even important to find a stable definition for a term, due to the fact that language is not where we actually start when getting involved with 'evidence' (I find gravitation a very evident experience, especially when tired or exhausted, no matter how gravitation is further discussed in physics or what connotation it may have or what it is called in the first place). So finding an exclusion of a definitional rule, such as contemplating on the meaning of "causal relation", "influence", "conditional relation" et cetera is often not as witty as it may seem, especially when the concept of a variable allows for potentially anything (is contingent) to be in conditional relation with anything else, even though it may require a (physical) measurment to be an object of discussion in the first place. So the goal of this tutorial is to make people aware of different levels of abstraction and how they should be part of planning and evaluating an experiment / study, but we want to also clearly distinct and present the boundaries of different perspectives on science as such (e.g. philosophy).

We hope this chapter has given you a stable grip on all the diverse aspects involved when operating with the dependencies of variables. What's next? We will dig in deeper into the actual mathematical representation of the *relation* between our dependent and our independent variable**.** The easiest way such a mathematical relation can be drawn is of course via a *linear model* (conditional linearity), i.e., a linear function with $f(x)$ (y-axis) being the dependent and x (x-axis) the independent variable. Note that in the case of the dog's food and the dog's mood, both x and $f(x)$ are considered categorical (in this case representable as a binary 0 or 1 for each type of mood and food; we can still fit in a linear function in only two possible values of x and $f(x)$, we will do so in the third part of this series).

If this feels a lot already, no worries, the tutorial is intentionally structured in a redundant way and will soon provide you with some tools in the form of code that

empowers you to visualize and literally experience the mathematics behind the concept of statistical inference in the form of linear modelling.

> We have learned…
>
> … that a dependent variable is called dependent, as it is influenced by another variable. E.g., a change in the dog food results in a change in the mood of the dog, such that the dog's mood is *dependent* on the dog food, so to speak. The dog food on the other hand does *not* change just because the dog's mood changed and is therefore understood as an *independent* variable (otherwise the dependencies would be *tilted*).
>
> … that the dependency of variables is nothing that comes with the definition of what a variable is and is also not entailed in a set or chosen variable itself. The dependencies have to be evaluated in several ways, **either** intellectually by forming a hypothesis, developing a plausible experimental design, evaluating previous research, **or** via a hypothesis test on a model, i.e., after gathering data.
>
> … that there might always be a chance for some unknown variable influencing the results. Variables which are not represented in the model (**confounder**, **proxy** etc.). Unfortunately the model does *not* evaluate itself on that matter, so it is on us to reflect on various possible flaws of an experimental or study design.

2 Obtaining Linear Models via the Linear Least Square Method

As we know already: The main goal of a linear regression is to draw a relation between different data points in a way that one can literally *draw* a linear conclusion between the two variables represented as a line in a graph, such that, e.g., a *steady* rise in the amount of time results in a *steady* rise in the amount of, say, cups of coffee drank — or a change in dogfood (categorical "rise") resulting in a change in the dog's behavior.

As mentioned before, the purpose of such a model is in general to do **predictions on new data**! We intuitively know that a rock-solid linear relation may be rather unrealistic, as there might always be something slightly influencing the result (e.g., referred to as standard deviation). In other words: our predictions may never be exact — but maybe good enough! The goal is therefore to find a model that results in the *least amount of error* to actual data points (events we encounter). We are looking for a compromise, so to speak, given that our model is a simplification of reality.

If this sounds enigmatic, don´t worry, as the general idea of a linear regression model is actually really simple and intuitive: it is *literally* just about drawing a line into a graph with data points in a way that the linear model doesn´t deviate too much from the data points, when the data points are seen as a whole of some kind. Try yourself:

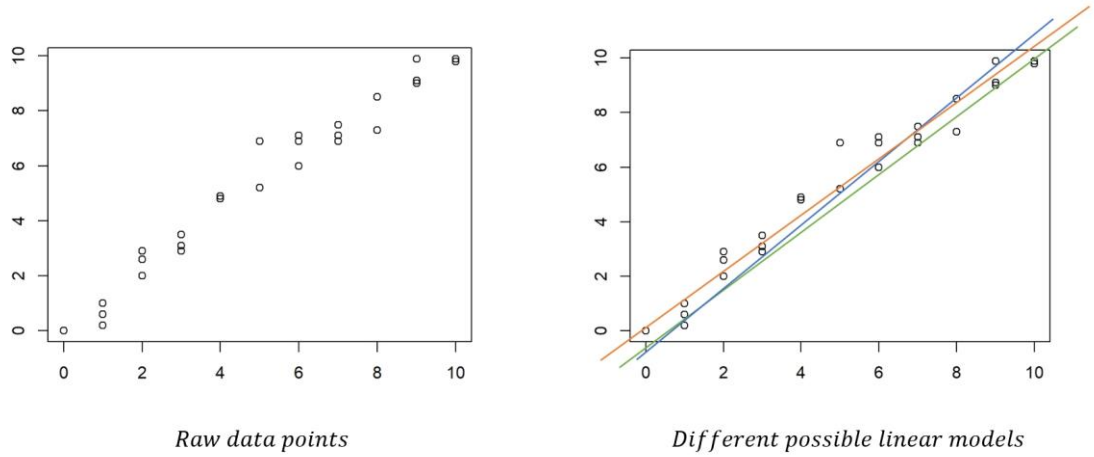


Fig. 6 On the **left-hand side** we see a plot of some data. On the **right-hand side** we see several attempts of drawing a line that may be optimal regarding the data points. Mathematics will eventually help us to make a reflected and transparent decision on that matter.

2.1 Linear Functions - Idealized Linear Relations between Variables

For our first idealized example below, we will *not* work with categorical variables this time. We will also change the species: This time we want to evaluate how many cups of tea **Lady Meow** drinks in a period of 10h.

Fig. 7 Lady Meow, happy about a fresh cup of tea. Original photo by Miguel Á Padriñán.

To get some orientation and in order to wrap our head around linear relations, we will plot the relation between cups of tea drunk by Lady Meow and the time passed as data points $P_i(x_i = time\ passed \mid y_i = cups\ of\ tea)$, where i refers to the i^{dth} element of X and Y. We will therefore start to represent the relation of our variables via *discrete* data points, i.e., single events in time (each hour), *not* a continuous relation between time and cups (values for "cups" for potentially infinitely small values of "time"). **Think of these data points more as snapshots of reality and our linear function as a *continuous solution* to the gaps in our discrete data** (1, Bärtl, 2021).

In our new furry example, the **time is considered the *independent variable***, which can be found intuitively as **time is something that is usually *not* influenced by any other relevant variable** apart from maybe failures in measurements, and therefore represents the independent variable in most of the cases. Exclusions are physics experiments on, e.g., the relativity of time and space in relation to mass, or measuring reading times of a text item, where the time passed, i.e., the time needed to read the item, may change (dependent), when changing the item (independent, as the item itself is not influenced by the time needed to read the item). The **amount of tea that Lady Meow drank represents the *dependent variable*** — dependent on the amount of time (passed).

Fig. 8 We will observe Lady Meow's tea drinking behavior over time. The **time** will be our **independent variable** and the **amount of tea** will be the **dependent variable**. In other words: the amount of tea Lady Meow drinks is dependent on the amount of time. The amount of time itself is independent from the amount of tea Lady Meow drinks, because when she drinks more tea, it will not speed up time or so. The time therefore represents a variable that is in our hands (our measurement), not in the hands of Lady Meow (she cannot influence time or our measurement of time with her drinking behavior — she just "passes through time", so to speak). Original photos by Miguel Á Padriñán and Malvestida.

Further below you will again find code for the programming language R that can be used to plot some **idealized data points**, i.e., where every point is located on a linear function already, as an example. **If you are new to R** you can either just read this tutorial and with it read the code (we will provide all the results), or you **just download R and RStudio**, **open a new script** and copy and paste the code into it. **You can also download and use the R script we provided at the beginning of this tutorial, which contains all of the code we are going to use**.

Mark the lines you want to execute and **press ALT+ENTER**. The result can be seen in the environment tab on the upper right side in RStudio. If you ever feel that your script is presenting a funny output (especially after a series of errors), **clear the environment via the brush tool** — there is another brush-icon placed in the console tab to clear the console. If this does not help close R without saving the workspace. **In case you are using any Apple hardware, use the command key instead of ALT.**

We also recommend using the keyboard when operating within the script: use **SHIFT+ARROW** to mark and demark letters (left/right) or whole lines (up/down).

Mark the **name** of an object **only** and press **ALT+ENTER again** to obtain the **results in the console** (below the script) — **you again won't need to know much more for this tutorial and all the rest will be introduced along the way.**

Note that R scripts are executed *from* top *to* bottom. The R script we provided can theoretically be executed as a whole (mark all and execute). However, it may be that a variable, e.g., with the name `test` , gets *redefined* in lower parts of a script with a different value (results in changes in the environment!). In other words: The content of the previous object with the same name `test` will be "overwritten" by the new content assigned to the name `test`**. Keep that in mind, when playing around with the script.** Just download the original script again if you have the feeling that you changed to much in the script and lost track or so.

We will represent our data points via a **vector** in R. Think of a column of a data table as a (column) vector. In R a vector is in general used as an abstraction of mathematics, where vectors can be seen as, e.g., a *list* of coordinate values `c(x,y)`. In R it can list anything that way, such as a set of values of measurments. The function `c()` is called the combine function, with which objects of any kind can be combined as a list of objects, or even a list of lists (note that there is also a function called `list()` as well, which is not structured in rows and columns, but sequentially numerates elements — also of various kinds of classes (vector, matrix, single values, whole data frames, all in one list)).

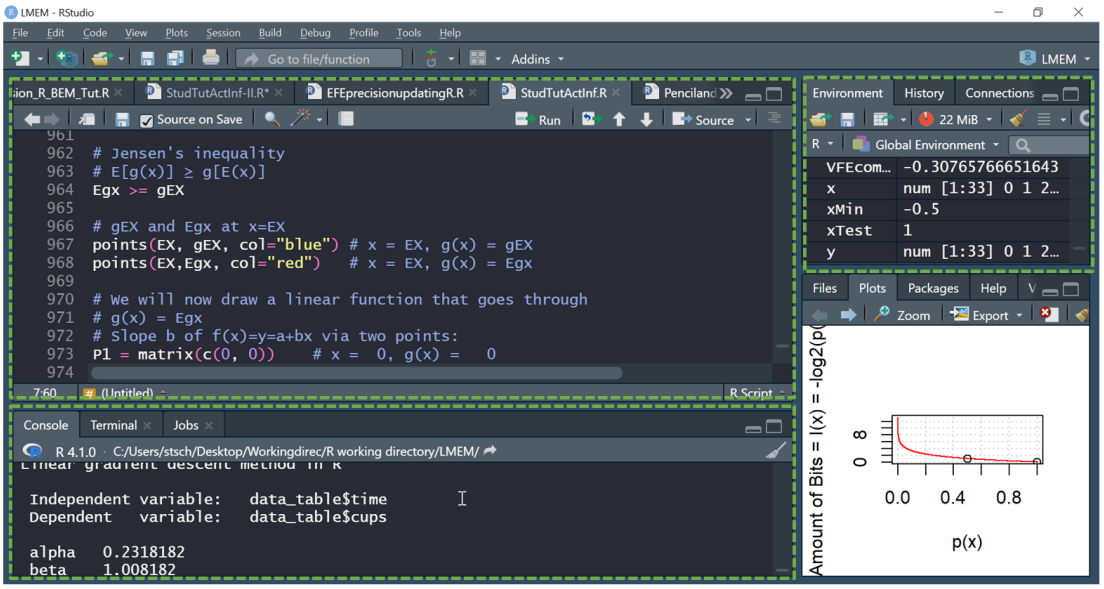


Fig. 9 RStudio is an IDE, i.e., an integrated development environment, in general used to ease the use of R. For this tutorial you won't need to understand much of RStudio in order to follow and execute the script. On the **upper left** you will find the window for the script — containing unrelated content. On the **upper right** side, you will find the *environment*. It keeps track of what you executed. You can also find the results in the console output, which is usually to be found on the **lower left** side. To also obtain the result of an object in the console, after you have executed the whole line, such as test = 2+5, mark the name of an object only (here "test") and execute (*again, the whole line has to be*

executed first). The appearance of RStudio can be changed via *Tools -> Global Options -> Appearance*. The upper theme I am using is called "Dracula".

Below you will find our first lines of code for plotting a set of synthetic data. Below x and y will always have equivalent values (idealized data set). Recall that # marks lines as *comments*, so they won't be interpreted as command code by the program (as plain text will result in various errors).

```
# Variables:

# time_passed = indep. = y-axis
time_passed = c(0:10)    # 0:10 = 0 to 10 = c(1,2,3,4 … 10)

# cups_of_tea = dep. = x-axis
cups_of_tea = c(0:10)

# Let us plot our data points
plot(x = time_passed, y = cups_of_tea)
```

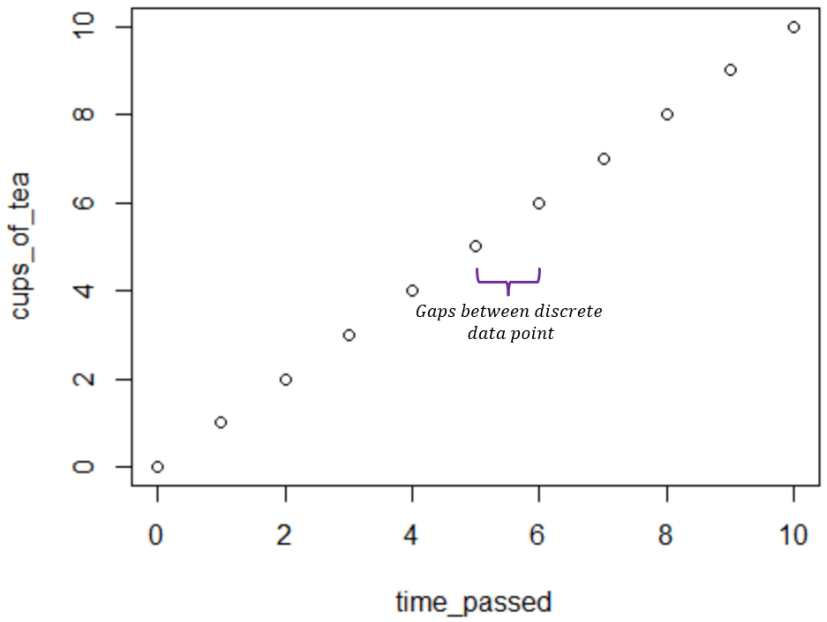


Fig. 10 (Slightly modified) plot of our synthetic data points — we can already "see" a line going through the discrete data points. Note that the y-axis is actually missing in the plot. The plot just entails the tick marks of the plot window, but that should not bother us any further.

You might remember that the information that is entailed in the formula of a linear function refers to **a)** the **slope** or "steepness" of a function and **b)** the **point where it crosses the y-axis**, when x is 0 (which can be read out of a simple linear function — or a graph: in the plot above its 0).

Before we look at the actual formula of a classic linear function as a whole, we will start with calculating the slope. First recall that a function is nothing else than a *relation of two variables* (here a linear relation). The slope is therefore also a relation, just a relation within a time-space in our case, or within a *difference* in time ($x = time\ passed$), more

general: a slope represents a *relation of the change of variables*: a relational difference. Considering our furry example, the slope reflects the difference between, e.g., the amount of tea Lady Meow drank after 1h compared to the amount after, say 10h (the ratio of the "change" of these to snapshots of reality so to speak).

We can mathematically evaluate the slope via two randomly chosen points of our data set, specifying and evaluating "change" as a "differing relation" (denoted $\Delta = delta$) of each y and x. Considering that x equals $time$, we can think of the below stating "a future data point $P_2(x_2|y_2)$ minus a past data point $P_1(x_1|y_1)$", in order to evaluate the difference between them (over time). Note that in other functions, such as a parabola, we can only determine the average difference between two points or the difference at *one point* in time (we will go through an example later below). Also note that the below formula is also defined as the quotient of differences or ***difference quotient*** (only *slightly* different to the differen*tial* quotient).

$$P_1(0|0), P_2(1|1) \quad => general\ structure: P_i(x_i|y_i)$$

$$m = b = \beta = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{1-0}{1-0} = 1$$

Below you will find a modified plot that will graphically explain the formula above. It also includes a drawn line representing the value of y or $f(x)$ for every possible x (continues line = continues values), our linear model function:

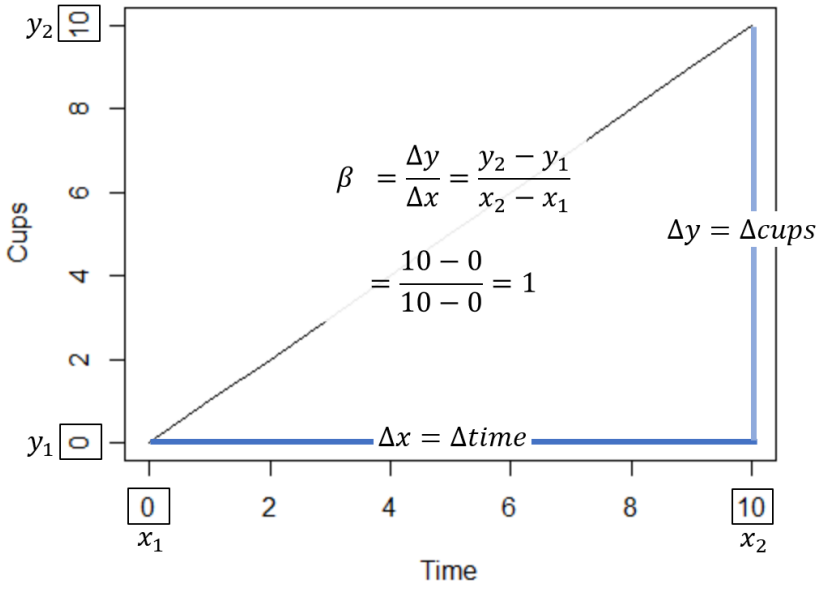


Fig. 11 Calculating the slope β of a function. In the modified plot the points P1(0|0), P2(10|10) were used. For different points doing the same calculation see code below and eventually choose points for yourself, if you wish. Any two ***deviating points*** on the function can be used. The formula is also called the differen*ce* quotient (only slightly different to the differenti*al* quotient — see later below).

However, the slope can also be calculated via one point only, via y/x. A special characteristic of a linear function makes it possible (will be revealed further below, so make a guess if you wish).

Let us now do the above calculation of the slope via R. To give you an example for the "logical flexibility" of R, we will use two different methods. There are often several possible ways to do a certain kind of operation. Try to fully figure out what is done in the code below, as the translation of the formula into reasonable code sometimes comes with the necessity of additional abstractions that have to be overlooked.

```
# Calculating the slope, directly via the objects of the variables
# that we used above already. The [] are used to call a certain value of
# the vector, i.e., element 1 of an object is object[1] etc. If this is
# too abstract, check on the whole object again and play around a little
# by adding [] with and without values to the object.
beta = (cups_of_tea[2]-cups_of_tea[1])/(time_passed[2]-time_passed[1])
# beta = 1

# Alternative via defining the points, represented as vectors P = c(x,y),
# first.               Console output: #     x y
P1 = c(time_passed[1],cups_of_tea[1])  # [1] 0 0
P2 = c(time_passed[2],cups_of_tea[2])  # [1] 1 1

# Note that here [] refers to the vector list element of P now, not the
# vector list of our data for x and y!
beta = (P2[2]-P1[2])/(P2[1]-P1[1])
# beta = 1
```

Now what about the point where our linear function crosses the y-axis? We can see in our graph further above (and by logic) that the minimum value of cups and the minimum *input value* of time measured (passed) will be set to 0 and therefore already represents the value of α in this special case, as the points of our idealized data set are located on the function already. We could start somewhere else, say with 5 cups of tea, so the value of y at $x = 0$ would be 5, such that $f(x) = x + 5$. However, our value of cups (y) at the beginning of our measurement ($x = time = 0$) will be 0, such that $n = a = \alpha = 0$, no matter what the value of β, i.e., the slope will be. To make sense of this in our example, let´s say that we want to evaluate how many cups of tea Lady Meow drinks when she is *awake during one day* — as she sleeps around 14h a day on average (^◦ ^). We will therefore start with zero and will not use a 24h scale for one day of measurements, just 1:10.

Now let's finally have a look at the complete **general formula of a linear function** itself (various common forms displayed, do not get confused!).

$$f(x) = y = mx + n = a + bx = \alpha + \beta x = \beta_0 + \beta_1 x = \ldots$$

$$Slope\ of\ a\ function := m = b = \beta = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

$$Point\ where\ our\ function\ is\ crossing\ the\ y-axis, i.e., where:$$

$$x = 0 \quad := \quad f(0) = \alpha + \beta * 0 = \alpha$$

It is also very important to understand (!!!!) that the concept of functions in programming is an abstraction of the mathematical concept of a function such as the one above:

$$input \longrightarrow \underset{function}{f(x)} \longrightarrow output$$

Fig. 12 Description of a function as an algorithmic process, specifically referring to the computational abstraction used in programs such as R: A mathematical function, say f(x) = x^2, has an input, say x = 2, and delivers an output for f(2), which in this case results in 2^2 = 4 and resembles the y-coordinate of a point on that function with the respective x-coordinate value of x = 2. The input represents the independent variable and the output the *change* of the dependent variable *after* it was influenced by or in general simply in *relation* to the independent variable, x. The algorithmic process performed within a function is arbitrary in programming. The input could be a list of names and the output a blank data table with the respective input names set as column names. We will learn more about functions and their possibilities later below, eventually writing functions for ourselves.

To get a better orientation on the variables α and β of a function, we will look at another modified plot. Below you will see how changes in the values of α and β change the graph, i.e., our *linear model*. The model is again the relation between $f(x)$ and x, and a *linear relation* means that the relation can be represented as having a constant numerical gradient in terms of a steady differential ratio throughout the whole graph (different to a pond like parabola function, as we will see later below), which just means that the slope of a linear function will be the same at every point of that function.

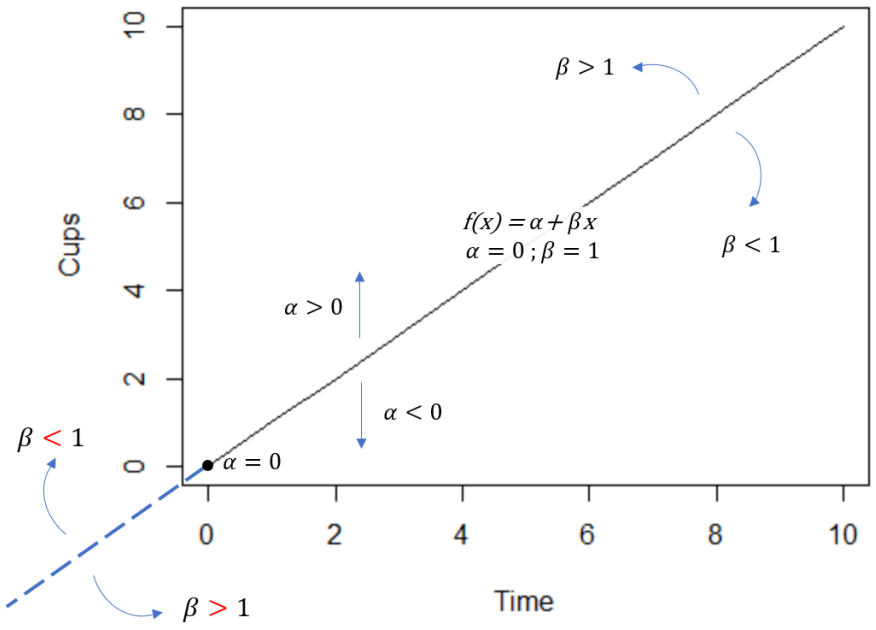


Fig. 13 Modified plot of f(x) = α + βx = 0 + 1*x = x. A change in α indicates that the function as a whole is moved either upwards or downwards. A change in β on the other hand changes the slope,

i.e., how "steep" the function is. A change in steepness, as indicated in the modified plot above, is not to be understood as indirectly changing α. Think of α more as an **axis-point**, P(0|α), where the function rotates on when β is changed.

Again, our function will relate the number of cups of tea that Lady Meow drank ($f(x)$) with the time passed (x). **In our case we already know that the function that fits our data best will also have the final form of:**

$$f(x) = 0 + x * 1$$

$$f(x) = x$$

With this information we can now add our function to the plot of our *idealized data set*. There are again several different ways to do so. We will mostly use the `abline()` function in R for such cases:

```
# We can add a line representing y = a + bx
# to our plot. The line crosses the y-axis
# at 0 (a = 0), as we are starting with y = 0 at x = 0.
# a(lpha) = 0, b(eta) = 1
abline(a = 0, b = 1)
```

Another way of representing the function via R would be as actual "R function", referring to what we have learned on the abstraction of functions above:

```
# Define function f(x) = 0 + x * 1 = x.
f = function(x)(x)
plot(y = f(0:10), x = 0:10, type = "l", ylab = "Cups", xlab = "Time")
points(0:10, 0:10) # add data points to plot
```

Note that the function above is spoken out "**f** is the name of a function with one input **x** and with that **x** some math is done, i.e., it remains **x**" — which may appear confusing, as it says, nothing is done with it. This is due to the fact that in the case of our function **f** the input value is equivalent to the output value. A parabola function, $g(x) = x^2$, in R would be coded as follows (we will also plot such a function later below in another chapter):

```
# Example parabola function:
g = function(x)(x^2)
g(2) # = 4
```

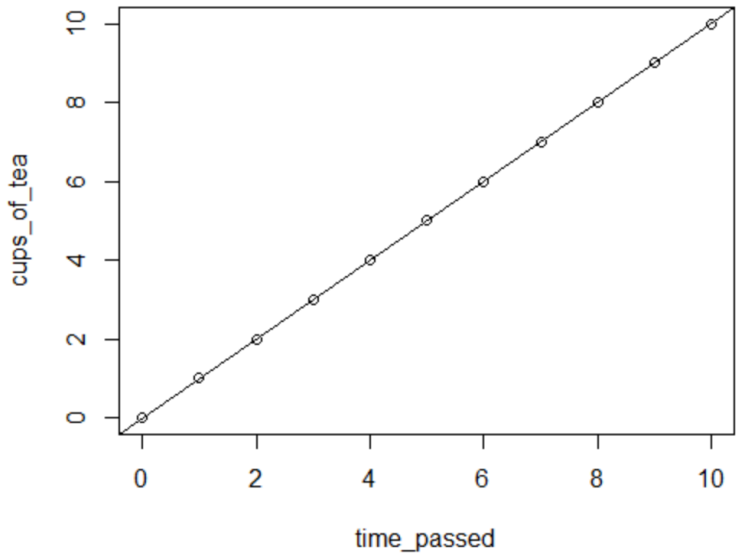


Fig. 14 No matter what code option you have chosen, the result of plotting our f(x) = x will look like the above plot. Some points appear to deviate, though this is just the sub-optimal pixel-plot preview in RStudio (newer versions come with higher preview resolution).

With this we have visually/graphically proven to ourselves that there is a linear relation between the time that had passed and the amount of tea measured in cups, without using any linear regression math or so. We can also use the `lm()` function, i.e., the **l**inear **m**odel function in R in order to evaluate the optimal α and β for our function regarding our synthetic data, actually applying the linear least square method we are after.

```
# Using lm() to evaluate alpha (intercept) and beta of x:
# Formula: lm(y~x), saying: a linear model of y (dep.) in
# conditional relation (~) to or dependent on x (indep.):
lm(cups_of_tea~time_passed)

#  The result in the console will look like this
#  CONSOLE OUTPUT (DO NOT EXECUTE):
#  Call:
#  lm(formula = cups_of_tea ~ time_passed)
#  Coefficients:
#  (Intercept) time_passed
#   1.071e-15   1.000e+00
##      α           β

# Note that one can now also add the line to a plot of data points via:
plot(x = cups_of_tea, y = time_passed)
abline(lm(cups_of_tea~time_passed)) # does not work with plot(f(0:10)...)
```

The *intercept* "cups_of_tea" (α) is here to be understood as $1.071^{-15} \sim 0$. The y-intercept represents the value of $y = f(x) = cups\ of\ tea$, exactly when $x = time\ passed = 0$, which in our case is also 0, since $f(0) = 0 * x = 0$. **The y-intercept is therefore equivalent to our** α. The value $1.000e + 00 = 1^0 = 1$ and represents the slope. **The slope essentially represents the value of y, when x is raised or changes** (Δ) **by the value of 1**

— **which for linear models is *always*** $\frac{\Delta y}{\Delta x} = \frac{y}{x} = \beta$. **In other words:** this tells us that Lady Meow drinks about 1 cup of tea per hour! **This is again the same as asking for the value of y under the condition of x, i.e., conditional linearity.**

You might have noticed that the slope appears much more intuitive than interpreting α. It makes sense in our example with time $x = 0$ in the sense of the start of our measurment. However, some measure start at higher values, so that α will appear rather abstract, as it it may not lay in the field of relevance / plausibility — just keep in mind.

Don't let yourself get confused by unknown terms such as *intercept*, used in the console output of the `lm()` function, as they do not add much to the story. **In case you get lost, try to always recall the basics of a simple linear function to reorientate within the math.**

We have learned…

… what the difference between discrete data points (snapshots) and a continuous function is.

… what the general structure of a linear function is and **a)** how the slope changes when β is changed, and **b)** how the crossing point with the y-axis changes, when the value of α is changed. We have also learned that α is an axis point, where the linear model rotates on when is β changed.

… how to use the functions **c()**, **plot()** and **abline()**; what vectors and objects are; and how a function in R can be understood as an abstraction of mathematical functions.

… how to use the **lm()** function on our first synthetic data set, where every data point was already located on a function $f(x) = x$. Subsequently the model that best fits the data is also $f(x) = x$. There is no compromise we had to make and we obtained equivalent results via the **lm()**, the **l**inear **m**odel function in R.

… from the data that Lady Meow had a steady pace when drinking her tea.

2.2 Obtaining a Linear Model from *Non-Idealized* Data Sets using lm() only

Our linear function in the previous chapter represented our so-called "model" function, i.e., a function that served as a model of the relation between variables, such as "time passed" and "cups of tea", in order to predict new observations (new data). Note that other functions such as logistic functions are also used (see further tutorials in the future). In the far most of the cases we will *not* obtain an idealized linear relation between variables, representing a perfect line of data points. Again, the goal is to find a model function with which we will make the *least* number of mistakes, or get the *least* possible deviation on average from "reality" (phenomenologically evident events), when trying to *predict* future events — new data. In general, minimizing an (prediction) *error* or *cost* in terms of "how much more information would I need to get to the right answer, i.e., how do I have to change the parameters of my model to better predict the future?" is considered an

optimization problem in mathematics — the *least* square method being a very simple representation for solving such problems.

Let us now work with data points that do not already perfectly align with a linear function. Below you will find some random data points I chose, this time slightly deviating from our linear sequence of 0:10, where the numerical gradient, the difference between each step of time, was always 1, i.e., linear by itself already. The data below will represent ***three measurements*** of $0h$ to $10h$ each. **No worries, you will soon get a proper overview over the data below in the form of a table**. Note that time is not deviating from our linearity or steady gradient, as it is the one variable we have control over — our steady independent variable:

```
# Cups of Tea:
cups = c(0, 1, 2, 3.5, 4.8, 5.2, 6, 6.9, 8.5, 9.1, 9.9,
         0, .6, 2.6, 3.1, 4.8, 6.9, 7.1, 7.5, 8.5, 9.9, 9.9,
         0, .2, 2.9, 2.9, 4.9, 5.2, 6.9, 7.1, 7.3, 9, 9.8)

# Time passed for each run of measurements (cups), abbreviated:
time = c(0:10,0:10,0:10)
```

The three runs can be understood as different participants or just the same person observed over $10h$ three times. At this point of the tutorial, it just does not matter if we model just the behavior of one person or a group of participants. Let's say it's our beloved Lady Meow drinking tea over and over again (Omnomnom). Later on, in the third part of this series, we will discuss and consider aspects such as number of participants in our statistical analysis (power analysis, α- and β-failures…).

With the `cbind()`, i.e., the column bind function, we can combine our two (column-) vectors, resulting in an object that is something more of a data table (class of that object is actually a combination of a matrix and an array; see the web or *future* R-Basic tutorials within this collection on that matter). We will then convert the result of `cbind()` and turn it into an object of the class `data frame`. With this conversion we are now able to make use of the function of the `$` sign in R, to call only a certain column (-vector), such as `data_table$time` within the `lm()` function. There are also other ways, such as again using `y = cups` and `x = time` within the function `lm()`. I chose the following, as a proper data table should be a more familiar and less abstract format to you and using the `$` sign is common when operating with .csv files for eyample. **It is also a good chance to learn about the circumstance that not every object class is compatible with every function, often resulting in enigmatic errors in the console. The function `lm()` for example cannot handle our "matrix, array" object for itself, therefore the conversion; you will definitely come across such issues in the future). By the way, you can evaluate the class of an object via `class(object)`. Give it a try!**

What we get below is a data table, where each row represents a possible relation between the number of cups and the amount of time that has passed. The numbers on the far-left side of the console output (see below) just represents the number of elements as row number. I did not include an extra column differentiating the different runs of measurement any further, so the values of all three runs are entailed in one column. In nuce, each row of the data table entails the y-value with its respective value of x.

```
data_table = as.data.frame(cbind(cups,time))

# Output of "data_table":
#
#    cups time
# 1   0.0    0
# 2   1.0    1
# 3   2.0    2
# 4   3.5    3
# 5   4.8    4
# 6   5.2    5
# ...
# ...

# Let us now plot our new set of data points. By using the $ sign
# a certain column or object in a list only will be called:
plot(x=data_table$time, y=data_table$cups,
     ylab = "Cups of Tea", xlab= "Hours passed")
```

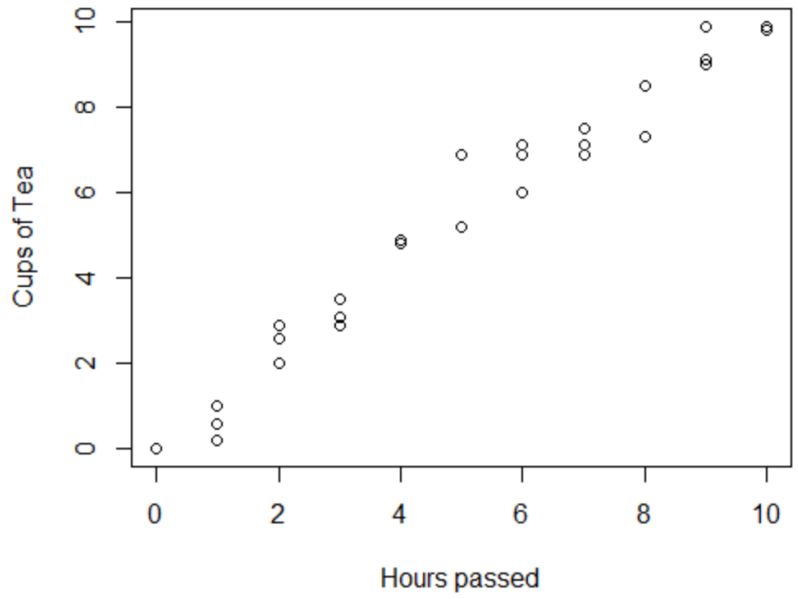


Fig. 15 Our second synthetic data set — again representing a relation between time passed and cups of tea (three for each x with their respective value of y). This time we *still* see a rather linear relation, but if we were to draw a linear function to it, not every point would be on that drawn function — we'd have to find a compromise as part of deciding where to draw a line. This trade-off though comes with a cost. If we were to predict data via the compromised linear function we evaluated via, e.g., **lm()** in R, our predictions would deviate from the actual (new) data in most of the cases. The compromise can therefore be looked at methodologically and epistemically as follows: if we know nothing for sure (epistemics), we can still heuristically use methods (say linear regression only) that lets us make predictions with a model that at least makes the *fewest* errors of all possible models — a linear function that *fits best* to the data. This can be done by choosing the model with the *smallest deviations* when predicting actual (new) data, in other words: via an optimization of the choice of the linear function we draw, as we will see below. Let us first take the easy path via the **lm()** function and then go through what it actually does mathematically.

We can now use the `lm()` function to again easily obtain a *fitted linear model* between the variables time and cups of tea:

```
# As mentioned, we could just use:
linear_model = lm(cups ~ time)
# or the following, using $ to refer to a
# name of a column in this case:
linear_model = lm(data_table$cups ~ data_table$time)

# OUTPUT CONSOLE – Do note execute
# lm(formula = cups ~ time)
# Coefficients:
# (Intercept) time
#  0.2318    1.0082
##    α         β
```

Comparing the result of our new model with the previous, we can see that α has slightly increased, indicating a crossing point with the y-axis at $P(0|0.2318)$. The slope has also slightly increased, indicating a "steeper" linear model function ($\beta = 1.0082$), compared to the model of our previous *idealized* data points, where the slope was 1. Let's have a look via the plot:

```
# Add linear model to plot:
abline(linear_model, col = "blue")

# or again via:
abline(a = 0.2318, b = 1.0082, col = "blue")
```

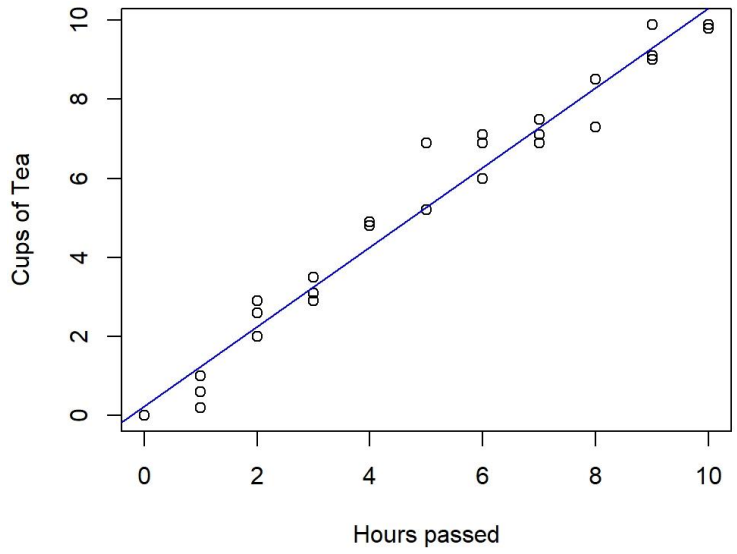


Fig. 16 The optimal linear model regarding our second synthetic data set, f(x) = 0.23 + 1.01*x.

We now know how to perform a simple linear regression via the `lm()` function in R. In our first synthetic data set all of the data points were perfectly in line with the function already. But how do we decide where to draw the line in the second case *mathematically*, without taking the easy path via R? **We will now start to go through the simplest mathematical**

method to make a compromising but optimal decision, in order to find a model, which will make the *fewest* errors of all possible models, when making a prediction: **the (linear) least square method**.

> We have learned…
>
> … how the result of **lm()** looks when applied to a non-idealized data set, where not every data point is located on our model function already.
>
> … how to transform vectors into the columns of a data table, using **cbind()** and **as.data.frame()** to do so. We have also learned a way to call a certain column or element of a list with a certain name by using the **$** sign: **data_table$column**.
>
> … that objects are assigned to have a certain class, such as "vector" or "data frame" and that not every class is compatible with every function (either completely or it results in faulty output).

2.3 Errors / Residuals

Let us first look at how these "errors" mentioned above are represented mathematically, before we start to figure out how to optimize them, eventually obtaining a model which makes the *fewest* errors of all models. **In general, solving our optimization problem involves setting up a "guess function" in order to optimize its parameters α and β with respect to the errors made when comparing the model's prediction with actual data points of a data set. Below we will just use our well known function $f(x) = x$ for a "guess function"**.

The error is a mathematical value that represents the deviation of the value of our model function, $f(x)$, from the respective data point y in relation to the same value of x. By subtracting $f(x)$ from our measured y, we will get a value that indicates how far off the value of our model function $f(x)$ is from the measured value y — again, both related to the same x (our independent variable).

$$Error = Residual := \ r \ = \ \varepsilon \ = \ y_{measurement} - y_{function} \ = \ y_{error} \ = \ y - f(x)$$

We will now look at some randomly chosen values $x \ = \ 1 \ to \ 5$ to get a simplified graphical overview on what residuals / errors are (our data set above has too many values to result in a nice visualization):

```
# Example residuals:
# x = indep. var.; y = dep. var.
x = c(0:5)
y = c(0, 1.5, 1.8, 3.4, 4.4, 5.9)
```

We can now evaluate the errors, defining the function `f` in R, representing $f(x) \ = \ x$ as our "***guess function" that we want to optimize*** to better fit the given data.

```
# Our guess function:
f = function(x)(x)

# Calculate errors (here x and y refer to the above data vectors!):
error = y-f(x)

# Console output of the object "error", entailing the elements 0-5:
# [1] 0.0 0.5 -0.2 0.4 0.4 0.9
```

Let us now plot our data points and our linear model $f(x)$, and add some lines to show what the residuals are graphically:

```
# Plot of our data points:
plot(x=x,y=y)

# Plot of our guessed function:
abline(a=0,b=1)

# Adds lines representing the residuals, i.e., the error between our
# "guessed function" and the actual data points, drawn along the y-axis.
# x0 is the x-coordinate of the start of the line and x1 the x-coordinate
# of the end of the dotted line (lty=3):
segments(x0=x,y0=f(x), x1= x, y1=y, lty =3, col = "blue")
```

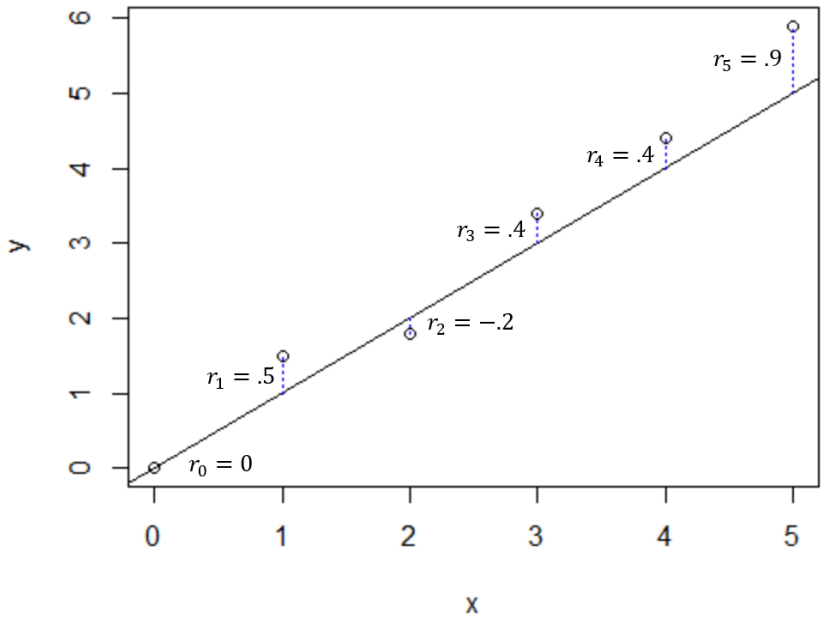


Fig. 17 Modified plot of some example values for x = indep. variable and y = dep. variable. Residuals are marked as vertical *blue* dotted line parallel to the y - axis, representing the deviation from y to our guessed model function f(x). It is parallel to the y-axis as both y and f(x) refer to the same value of x. In other words: the residual reflects the deviation between the y(-coordinate) and the model f(x) as *another* y-axis coordinate, given the same x(-coordinate) of some data.

Recall that we are after finding a function that produces a *minimum* of summed errors *compared* to the errors of all other possible functions. However, below we will also calculate

the *squared* error, r^2. We will figure out later, why it is called the linear least *square* method in detail, but think of it visually and linguistically and recall some prior knowledge: a minimum of something is usually graphed in a way that it mathematically at least looks something like a parabola function $g(x) = x^2$ — say a pond with a point that is lowest amongst all of the possible points in the pond. **If we were to represent our minimization problem as a *linear problem*, the minimum of a *linear* error function would be minus infinity for α — which is not what we want for an estimate for the crossing point with the y-axis. What we are definitely still looking for though is a *linear solution* in the sense that the result of our calculations leaves us with an optimal *linear* function (not a parabola function).**

In order to get a function that has at least something like a parabolic minimum as an attribute, we are mathematically just bluntly going to ***turn our linear problem, i.e., deciding for a linear function with the lowest errors (r), into a quadratic problem, deciding for a particular α and β that leads to the lowest values of errors squared (r²).*** We will get to this in detail in a bit, but definitely keep in mind that intuitively a linear function itself will not be something we are going to use to find a *minimum in general* — again, as the lowest plain (unsquared) error would result from a function with values in the area of infinity for its α and β (therefore, again, something at least parabolic is what we are looking for).

Now let us say, we are again working with our previous *idealized values* of y_i and our first guess for a model will be again our *idealized function* $f(x_i) = x_i$ (which we will also use as a "guess function" for any data set). Calculating the errors of some of our values **c(1:10)** would look like this (code will follow below):

$Time$ x_i	$Cups$ y_i	$f(x_i) = 0 + x_i$	$Error = y_i - f(x_i)$	$Error^2$
0	0	$f(0) = 0 + 0 = 0$	$0 - 0 = 0$	0
1	1	$f(1) = 0 + 1 = 1$	$1 - 1 = 0$	0
2	2	$f(2) = 0 + 2 = 2$	$2 - 2 = 0$	0
3	3	$f(3) = 0 + 3 = 3$	$3 - 3 = 0$	0
4	4	$f(4) = 0 + 4 = 4$	$4 - 4 = 0$	0
5	5	$f(5) = 0 + 5 = 5$	$5 - 5 = 0$	0
6	6	$f(6) = 0 + 6 = 6$	$6 - 6 = 0$	0
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
		$\sum_r$	0	$\sum_{r^2}$ 0

Fig. 18 Calculating the residuals and the squared residuals. As expected, we will not encounter any errors at all in this particular example. We will get into details why we mathematically want to evaluate Error^2 when discussing a way to minimize our errors — so far, we are at least conceptually informed why we do so.

We can do the same with our ***second synthetic dataset, using the same guess function again (theoretically any function can be optimized)***:

$Time$ x_i	$Cups$ y_i	$f(x_i) = 0 + x_i$	$Error = y_i - f(x_i)$	$Error^2$
0	0	$f(0) = 0 + 0 = 0$	$0 - 0 = 0$	0
1	.2	$f(1) = 0 + 1 = 1$	$.2 - 1 = -.8$	.64
2	2.6	$f(2) = 0 + 2 = 2$	$2.6 - 2 = .6$	.36
3	3.1	$f(3) = 0 + 3 = 3$	$3.1 - 3 = .1$	.01
4	4.8	$f(4) = 0 + 4 = 4$	$4.8 - 4 = .8$	.64
5	5.2	$f(5) = 0 + 5 = 5$	$5.2 - 5 = .2$	.04
6	6.9	$f(6) = 0 + 6 = 6$	$6.9 - 6 = .9$	.81
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
		$\sum_r$	9	$\sum_{r^2}$ 12.18

Fig. 19 Some random picked exemplary values from our second synthetic data set and the residual errors and r^2r2 of the whole data set. Note that R2 is confusingly not the same as our r^2 and is evaluated via a different formula, resulting in a value from 0 to 1 (discussed in the third part of this series).

Here is the respective code for the above and a possibility to plot the residuals:

```
# Following the plain math:
error = (cups_of_tea - time_passed) # here f(x) = x = time_passed itself
sumError = sum(error)
sumError2 = sum(error^2)

# Via lm() and residuals() of our idealized data set:
idealized = lm(cups_of_tea~time_passed)
residuals_fun = residuals(idealized) # all basically 0 values

# Checks for (near) equality of the result of the function
# residuals() with our object "error" that we obtained
# by applying the plain math (change object class of
# residuals_fun via is.numeric() to make it work):
all.equal(error, as.numeric(residuals_fun))
# [1] TRUE

# Let us plot our error / residuals
plot(error)
```

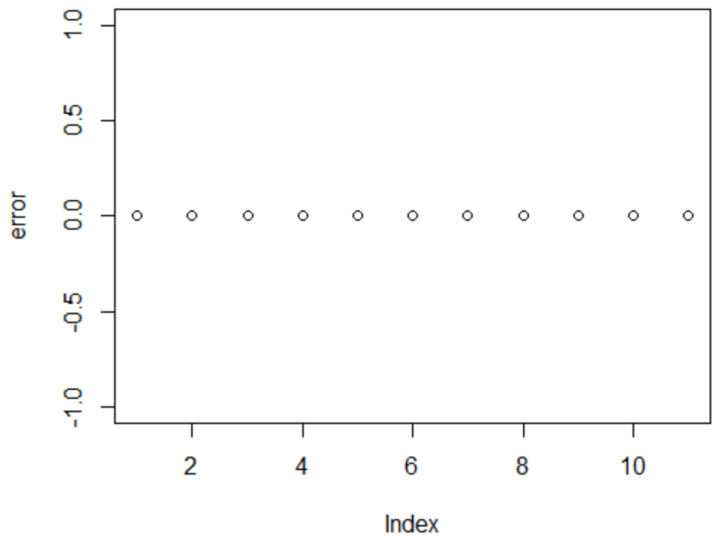


Fig. 20 Residuals of our idealized function f(x) = x. Without adding a horizontal line parallel to the x - axis : = error = 0 for our function f(x), we can already clearly see that there are no errors given in this model. In case it may be confusing: the dots do not represent our data, but the errors.

Let us now do the same with our second synthetic data set and **compare the residuals of our first guess with the residuals of our already fitted function** obtained via `lm()`. We will start with the errors of our guessed function.

```
# Guessed function
lmRandom = function(x) (x)

# These are the errors we still have to minimize
errorRandom = (data_table$cups-lmRandom(data_table$time))
sumErrorRan = sum(errorRandom)
sumErrorRan2 = sum(errorRandom^2)

# Plot errorRandom
plot(errorRandom)

# adds (h)orizontal line at a-axis = error = 0 (:= no error)
abline(h=0)
```

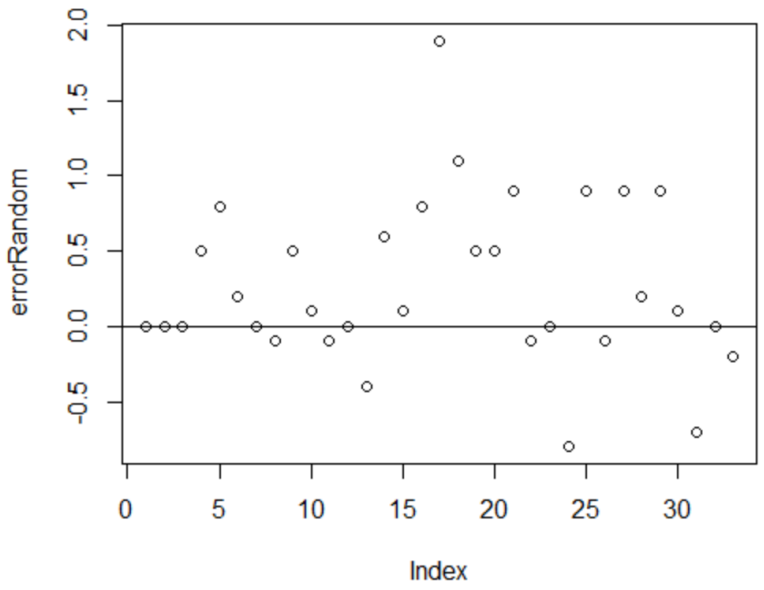


Fig. 21 Residuals of our *function* f(x) = x as guess for a model of our second synthetic data set. Looking at the errors of our *unfitted model function*, we can already see a tendency towards points being above 0 than below, visually indicating that our guessed function can be optimized. The first three points in the graph indicate zero error and refer to our starting value of zero in three runs of observing Lady Meow drinking her tea — our second synthetic data set.

```
# This is our fitted model function
lmFitted = function(x) (0.2318+1.0082*x)

# These are the errors / residuals that are already fitted
# In other words: this is where we want to get to.
errorFitted = (data_table$cups-lmFitted(data_table$time))

# The function residuals() does the same with our model
residualsFitted = residuals(lm(data_table$cups~data_table$time))

# Let us check if our method delivers same results as
# the function residual():
all.equal(errorFitted, residualsFitted)

# [1] "Mean relative difference: 0.0001735146" = neglectable difference

# Plot residuals:
plot(residualsFitted)
abline(h=0)
```

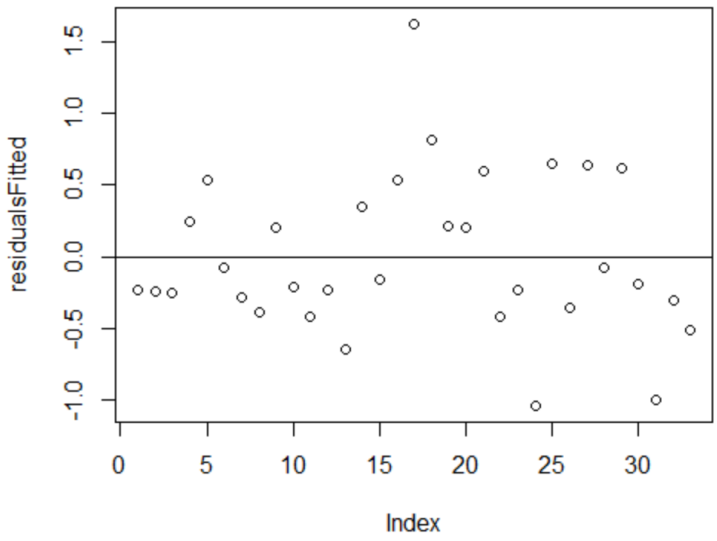


Fig. 22 Residuals of our ***fitted*** model function. y - axis = 0 represents a continuous horizontal zero error line for each value of x. The points represent our residuals, our errors. Note that the first three points on the left lost their attribute of having zero error; however, this is the compromise we decided to make, when getting involved with linear regression models.

As promised, we will now compare how the residuals have changed, after optimizing the errors:

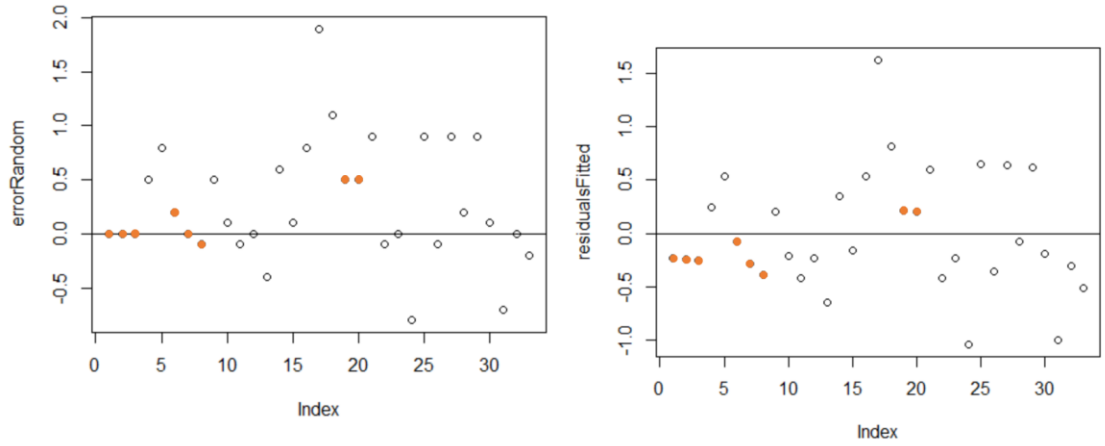


Fig. 23 Comparison of the residuals of our random (left) and our fitted model function (right). One can already see that all the points have moved downwards in some way (higher α) and that the overall plain, i.e., all of the points seem to slightly have lost slope (indicating slightly higher β of the fitted function). I have marked some of the points for comparison (*orange*). Both of the characteristics hold, since we already know that α = 0.2318 and β = 1.0082. We can also see that the number of points below and above zero error is more balanced in the fitted model function. What we cannot see at first is the fact that the points weren't moved with respect to a change in the frame looking on a stable plane of the graph (i.e., just moving the line on the plane), but the actual errors have changed diversly, just not that much in this case, so the relation between unfitted and fitted model is not that far of.

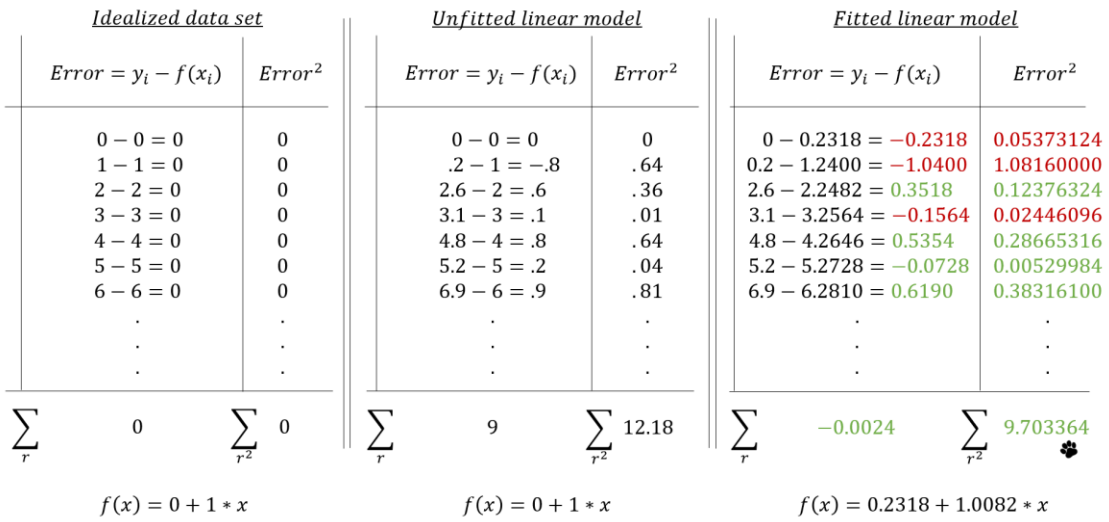


Fig. 24 Error and Error^2 of some of the values of our idealized synthetic data set, as well as of our *unfitted* and *fitted* linear model of our second synthetic data set. Below the table you will find the corresponding model function that is related to the actual data points. *Red* highlighting indicates a higher Error / Error^2 compared to the *unfitted* model. The errors of the fitted model result in the "diverse compromise" we have spoken about before, but still ends in a lower sum of errors^2. However, even with only some of the data values at hand in the tables, one can already see a tendency towards lower errors (green), which also reflects in the comparison of the sums or the errors between unfitted and fitted model (also green, if lower). Note that we could just also heuristically play around with possible values of α and β to see if we find a *better* model compared to previous guessed model in order to perform some kind of linear regression, given data.

We have learned…

… what errors / residuals are: $r = y - f(x)$, i.e., the deviation between the y-coordinate of our model function and the y-coordinate of our actual data points, both regarding the same value of x.

… that we are going to evaluate the optimal α and β that results in a model with a minimum of r^2 compared to all *other* possible models. Minimizing the plain errors of a *linear* function does not make sense, as we would end up with something +/- infinity for α and β. So far we are at least informed that we need to turn our linear problem into an quadratic problem to obtain functions, such as a pond like parabola function that entail minima that can be looked for in the first place.

… how to compared the residuals of our guessed and our fitted function and found out that the residuals diversly change, when assigning different values for α and β.

2.4 Finding a Model with a Minimum of *Error^2* via a Partial Differential Equation

In order to represent our minimization problem formally, we are going to use a partial differential equation. The field of partial differential equations (PDEs) can be complicated, due to a lack of solutions for *non*-linear PDEs. We are lucky though, as we are looking for a

linear solution only anyways, and our PDE can be solved, e.g., via a system of linear equations — the method we are after.

Note that there are other methods to find a linear function fitting our data, i.e., with the least amount of squared error. For example: in the third part of this series, we will obtain our optimal β just by dividing the so-called covariance of y and x by the variance of x (involves probabilitic perspective, which we will go into in part three of this series). One can even find a solution via gradient descent, typically applied in machine learning (we will eventually provide an advanced tutorial on that matter in the near future).

Excited Maneki-neko cat is drinking red tea and travels through time

Fig. 25 Three images generated by DALL-E 2, a program that uses neural networks to generate images from linguistic commands. The way these neural networks are trained is by minimizing a cost function, similar to our error in linear regressions. There is much more involved in machine learning methods. However, statistical modelling is still always about solving (mathematical) optimization problems in some form or another and methods applied in machine learning, such as *gradient descent,* can also be used to perform a linear regression. In fact, if we just go an and heuristically change the values for a and b and evaluate the respective errors in a way that we would follow a "direction" towards lower errors compared to previous errors, it would roughly represent what is called gradient descent: following the gradient of the descending r^2, when changing a and b. However, we see that the output of DALL-E 2 is also not fully in line with our expectations, as time is not always represented above, in other words: the inference performed by neural networks in the form of machine learning remains an abductive inference as well.

If any of this sounds too unfamiliar, no worries! This tutorial is not going to introduce the whole field of ordinary and partial differential equations (ODEs and PDEs) or any complex linear algebra — you will still be able to fully understand, follow and reproduce all of the calculations below (even by hand). Apart from that it is just important *to at least be aware* that different methods and formulas are used to evaluate a fitted function, *so that you will not become insecure when doing your own research, knowing only one of the methods*. (2) **However, solving optimization problems is essential for any kind of statistical modelling, so we believe it is more than helpful to at least know the poetics of a linear model in detail, to get a stable overview on statistical modelling.**

After we mastered the mathematics, we are also going to reproduce the essentials of the original `lm()` function to show you how easy it can be written by oneself and in order to demystify what the code of the original `lm()` function actually does.

Let us now finally look at our PDE:

$$E(a;b) = \sum_{i=1}^{n}(y_i - f(x_i))^2 = \sum_{i=1}^{n}(y_i - a - bx_i)^2$$

What we see on the far-right-hand side of the equation is essentially our squared error formula above ($\left(y_i - f(x_i)\right)^2$, the term $f(x)$ though is substituted by the full equation, $\sum_{i=1}^{n}(y_i - a - bx_i)^2$, and the squared error is now cast as a PDE of a and b of $f(x)$. Recall that the variables x_i and y_i are given. E stands for error.

So, essentially our PDE in the current form above is a function with respect to two variables, a **and** b, which correspond to the given values x_i and y_i, all together resulting in a value of the squared error (*not its minimum yet!!*). What we now want to use this formula for is to get the values of a and b corresponding to the *minimum value* of r^2 (error squared). Our optimization problem for itself, regardless the method, can mathematically be expressed as follows:

$$\min_{a,b} \sum_{i=1}^{n} r^2$$

Finding the minimum can easily be done by evaluating the *partial* derivatives of our PDE and setting each of them to zero, in order to evaluate the optimal a and b given minimum squared error – our compromise.

"Setting the partial derivatives of a PDE to zero" has a similar purpose as with regular linear functions, where setting $f(x) = x$ to zero, i.e., $f(x) = x = 0$, reveals the value of y, where x is zero, i.e., we obtained the value for a (in the case of $f(x) = x$ both values x and a are zero). *However,* we already know that we have to turn our linear problem into a *quadratic* problem, so we are looking for a minimum of a 'pond' as a result of setting something to zero (and we have to take the derivative, which "downgrades" a parabola function to a linear function). Before we start with setting the partial derivatives to zero, let us recall some basics on *quadratic functions* that you will probably remember from school, to slowly build up complexity. **Note that the next chapter can be considered optional.**

We have learned…

… that there are different methods for solving our optimization problem of finding an optimal a and b of a function that corresponds to the least squared error.

… that we can use the linear least square method, which is translating our optimization problem of drawing an optimal line into an error function E that corresponds to a and b respectively (a function with two variables, such as $f(x, y)$ and not just $f(x)$ so to speak). Setting its partial derivatives to zero eventually leads us to values of a and b that correspond to our desired *minimum* squared error (an optimal model function). We will see that this process is similar to finding the minimum in a parabola function.

… that solving optimization problems is an essential part of any kind of statistical modelling.

2.4.1 Brief Excurse on Evaluating the Minimum/Maximum in a Parabola Function (Optional)

In a parabola function $g(x) = x^2$ the minimum/maximum value of a function $g(x)$ can be obtained by taking the *first derivative*, then setting it to zero, subsequently solving the equation for $x_{min/max}$. We can then obtain the corresponding value of y of that $x_{min/max}$ via plugging in $x_{min/max}$ into $g(x)$.

In general, the derivative gives us the slope of a *tangent* at a certain point x of the function $g(x)$, i.e., the slope of a (linear) tangent at a point $P\big(x|g(x)\big)$ on the parabola function.

The derivative of $g(x) = x^2$ is $g'(x) = 2 * x$ (via the general rule $g'^{(x)} = n * x^{n-1} + \$ \sout{value}\$, where single *values* without *coefficient*, i.e., *without a corresponding factor variable*, such as α, are crossed out; note that α will become a coefficient within our PDE and is therefore often referred to as coefficient as well).

Note that the *derivative* $g'(x) = 2 * x$ is a linear function — and it crosses the $x - axis$ at the minimum/maximum of our parabola function $g(x)$. **Below you will find code to plot an example of a parabola function that does *not* have its minimum/maximum at the origin of the coordinate system.**

In case you may wonder or get confused, note that setting the derivative to zero, is *not* the same as setting x of the derivative to zero. The result of $g'(0)$ will give us information of the slope of the function at a point $x - axis = x$ and $y - axis = g(x)$. We are after a *particular* point with a *particular* slope: a slope of zero, indicating a minimum or maximum of a function — therefore we set to zero $g'(x) = 0$ and fill in the whole function for $g'(x)$, resulting in $2 * x = 0$ in the case of $g'(x) = 2 * x$.

Conceptually it is also important to keep in mind that x is our independent variable, i.e., the one that is not influenced or changed, and refers to both $g(x)$ and $g'(x)$ i.e., the input value for x does not change.

Below you will find some code to do some calculations and plotting:

```
# Slightly more complex parabola function:
# Defining our function g(x) = x^2 + x + 2
g = function(x) (x^2+x+2)

# Plot g(x)
plot(g, xlim = c(-5,5), ylim=c(-1,10), asp=1 )
abline(h=0, v=0)

# Derivative g'(x) of g(x)
gdev = function(x) (2*x + 1)

# Evaluate if minimum or maximum is given (none if below = 0)
# a) Input neg. and pos. test values
# b) Output: changes from negative to positive => maximum is given,
#            else a minimum is given (i.e., when input == output)
gdev(-Inf); gdev(+Inf) # Minimum is given

# Add derivative function (it crosses x at our xMin!)
# Here we graphically see that x refers to both g’(x) and g(x)
# g’(x)=2*x+1
abline(a=1,b=2) # adds our linear function to plot

# Setting g'(x) to zero, to get the x coordinate of that minimum
# which has a slope of zero
# g'(x) = 2*x + 1
#   0   = 2*x + 1
# -2x   =    1
xMin = solve(-2,1)
#  x    =  -.5

# Let's check if this is true, since g'(-.5) = 0
# In other words: the slope at g(-.5) is zero
gdev(xMin) == 0 # [1] TRUE

g(xMin) # y-coordinate of minimum
points(x = xMin, y = g(xMin))
# Result: P(-.5| 1.75)

# Add vertical line at x = -.5
segments(x0=-.5,y0=0, x1=-.5,y1=g(-.5), col="darkgreen", lty = 3)
```

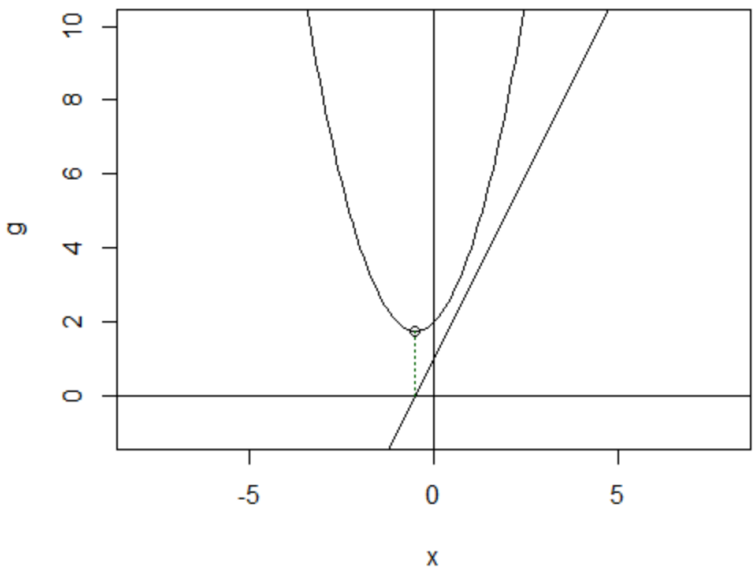


Fig. 26 Plot of our function g(x) and it's derivative g'(x). The vertical green dotted line shows the connection between the crossing of the derivative with the x-axis and the minimum of g(x) at the same value of x (using the **segments()** function in R).

Now that we know how to find the minimum in a parabola function, let's precede to PDEs. The relation in our function $E(a, b)$ is more complex this time, so the minimum has to be evaluated *partially*, i.e., first either for a and then b or vice versa (we could also use good old α and β, but I'll shuffle conventions a little, as it is good to keep in mind that these letters/symbols are never exclusively fixed to a method).

In other words: we cannot evaluate a and b at the same time, and what we will get when taking the derivative of a PDE is actually two equations that we have to solve – after doing some rearrangements. Both a and b will be related to the least squared errors, but it will be a rather abstract visualization compared to finding the minimum of a simple parabola function (so we will skip plotting on that part).

We have learned…

… that the first derivative of a parabola function is a linear function that crosses the $x - axis$ where the slope of that parabola function is 0 (and therefore marks a minimum / maximum of a parabola function).

…that the $x - coordinate$ of the minimum / maximum of a parabola function can be obtained by setting the first derivative to zero. The $y - coordinate$ of that point can be obtained by inserting $x_{min/max}$ into $f(x)$.

… that we can apply a similar method for our function $E(a, b)$, but we have to take the derivate partially, i.e., either with respect to a *or* b, and will end up with *two* equations that we have to solve.

2.4.2 How to Obtain the Partial Derivative and the Difference between ∂, d and Δ.

Most of the below will be more about formalism and will be conceptually familiar to you, promised. It is also enough do get an overview of this chapter. However, we again still want to give you an optional overview of the process with the least number of missing links (at least for simple linear regression models). Same goes for the decomposition in the next chapter, so don't be afraid of what's ahead.

The below is supposed to show that a PDE is not too different from an ordinary differential equation and from the concept of a difference quotient. We will therefore again rediscover some math basics from our time in school. By looking into the difference between ∂ (del), d and Δ we will slowly build up the necessary complexity we need in order to understand what "setting a partial derivative of a PDE to zero" means in relation to an optimal linear function with the lowest possible r^2. However, for a less heuristic and more detailed introduction into partial differential equations on the example of a heat or diffusion equation, we recommend the following video by the channel 3blue1brown.

So, let´s get to it. This is our PDE again:

$$E(a;b) = \sum_{i=1}^{n}(y_i - f(x_i))^2 = \sum_{i=1}^{n}(y_i - a - bx_i)^2$$

The derivatives are formulated as follows. The purple part refers to the outer derivative, and the blue to the inner derivative. Obtaining the outer derivate is simple: ^{2} becomes $2*$. The inner derivative only concerns a or b, such that $-a$ becomes (-1) and $-bx$ becomes $(-x)$ (we will not go any further into the rules for partial derivatives here). Just note that $-f(x)$ is substituted by $-a-bx$, as all of $a+bx$ is subtracted.

$$E'_a(a;b) = \frac{\partial E(a;b)}{\partial a} = \sum_{i=1}^{n} 2*(y_i - a - bx_i)*(-1)$$

$$E'_b(a;b) = \frac{\partial E(a;b)}{\partial b} = \sum_{i=1}^{n} 2*(y_i - a - bx_i)*(-x_i)$$

The fraction is spoken "partial derivative of $E(a;b)$, regarding a or b, i.e., in respect of either a or b", which just means that either a or b is set constant (does not change; no difference), when the partial derivative with respect to either a or b is evaluated.

The symbol ∂ (del) serves a similar purpose as d in ordinary differenti*al* equations, and as Δ (capital delta) when evaluating the slope of a linear function: they all denote a *difference*, just different kinds of differences, so to speak. In order to get a stable grip on the differences of differences, recall that the slope of a linear function is calculated by the difference of y divided by the difference of x. If a function is linear, it is actually enough to just calculate $\frac{y}{x}$,

when 0 difference is given (the reason as you might figure is that a linear function is its own tangent with the same slope not only on one point, but every point of the function).

However, the Greek capital delta, Δ, is used for ***finite differences*** only. We will see in a bit what that means.

$$\beta = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

A **differen*tial* equation** on the other hand can also be used to evaluate the slope of a function, but with a twist to it. Equivalent to the difference quotient, a differential equation evaluates the difference, but substitutes all y with $f(x)$, such that one can potentially fill in the whole equation of $f(x)$. A differential equation can therefore also be applied *on a lot of other types of functions*, such as a parabola function, not only linear functions. In differenti*al* equations the difference is denoted by the Latin letter d. Another important aspect is that it *also* holds for ***infinite differences*** as well. For the numerical example below, we will work with finite values, i.e., where the difference *approaches* 0, but is still *not* 0 and dose not approach an infinitely small difference (we will go through a numeric example in a second). The differen*tial* or limit is denoted *lim* for limit or limes.(3)

$$\beta = \frac{dy}{dx} = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} = f'(x_0)\,, where\ "\to"\ means\ "approaching"\ not\ "equal\ to"\ x_0.$$

Let us check on the slope at our previously evaluated parabola minimum $P_{\min}(-.5|1.75)$ using a differential equation of the ordinary kind this time. Intuitively the below does not work when the *difference* is actually *equal to* 0, as it always ends up in a situation where 0 is divided by 0 (x - x = 0) — so it is important to understand that this is essentially an approximation by *approaching 0 difference, but never being in such a state*. To make it work in R, we have to add a tiny term (1^{-7}) into our calculation as difference, to obtain the correct value via regular R functions. There are other ways, but we just want to get the concept, in order to get a grip on our mathematical minimization problem in full.

$$\beta = \frac{dy}{dx} = \lim_{x \to x_0} \frac{f(x = -0.5 + 1^{-7}) - f(x_{min})}{(x = -0.5 + 1^{-7}) - x_{min}} = 0$$

Here x_o is in this case equal to $x_{\min}$ (equivalent to x_1 in the difference quotient so to speak). The value x on the other hand is our approximate value that is ***approaching*** the value of x_0 and therefore has to be slightly different (x is here more or less the same the future x_2 in Δx).

```
# Output using different x  =>       x  = x +.1          x = x +.001
betaNumerator = (g(-.5+.1)-g(-.5)) #   0.01                1e-06
betaDenominator = ((-.5+.1)+.5)    #    0.1                1e-03 = 0.001
betaNumerator/betaDenominator      #    0.1                1e-03 = 0.001
```

The output in R could be coincidental, since $0/0$ is also 0, so let us check with some other value of x we choose:

```
# Choose a value for x
xTest = 1

# Using g(x) from above:
g = function(x) (x^2+x+2)
betaTest = (g(xTest+1e-7)-g(xTest))/((xTest+1e-7)-xTest)
# Result for xTest = 1 => beta = 3

# Some values come with a remaining difference
# (not the case with, e.g., xTest = 9):
gdev = function(x) (2*x + 1)
all.equal(betaTest,gdev(xTest))
```

With an added value of `1e-7` to our chosen x, R outputs a rounded value for the slope, which is mostly equivalent to `gedev(x)`. Using `1e-12` leads to false values and at some point, you will get a `NaN` (Not a Number) as output, as the precision is not high enough or due to some other reason within the function (there may be ways in R, but it is not important for us now). We are not going to need the calculations above for the actual linear regression, but in case a PDE ist still too abstract, always just recall the general idea from a conceptually downgraded perspective.

We have learned: In differential equations d denotes the difference, when dx *approaches* a difference of some limit, such as 0, in order to obtain the slope at the respective point x, of $f(x)$. There are more things that can be done with differential equations, but we will not go into details any further.

We are now going to move on to taking the derivative of our PDE, setting them to zero, in order to find the minimum of *partial* differential equations, in our case the minimum of the sum of our squared errors of our guessed function. The goal is again the same as with ordinary differential equation, just a little more abstract: **we will be handling two equations at once and will solve them via a *system of equations*, when using the derivatives to obtain the optimal values for a and b, given minimum residuals.**

We have learned…

… that a lot of the mathematical basics we obtained in school can be used to solve our optimization problem.

…. how partial derivatives – each with respect to a *or* b – are denoted.

… that the sign ∂ (del) serves a similar purpose as d and as Δ (capital delta), denoting a *difference (different kinds of difference)*, just that ∂ refers to a *partial differential equation*, i.e., a difference to some condition (such as only a or b) and d to an ordinary difference and therefore an ODE.

… that we can also use an ordinary differential equation to evaluate the slope of a function at a certain point.

… that the field of PDEs can be complicated, but as we are looking for a linear solution for our problem, we only have to add tiny bits of complexity to what we know about linear and quadratic functions.

2.4.3 Setting Partial Derivatives to Zero and how to Rearrange its Equations to Represent a System of Equations

The next steps of decomposing our derivatives are a preparation to fit our derivatives into an easily solvable system of equations. Note that the rearranging part is supposed to give you the opportunity to follow all of the steps by hand, but is not necessary to understand in detail. The important part will be our resulting system of equations that can be used for any set of x and y in the future.

In case you my wonder: **We will get into what a system of equation is in a bit** — we could have explained it now or before, but as it is rather simple, we decided to first rearrange the equations of our derivatives into its respective form. For now, just think of more than one equation, *each* looking something like $x - y = 15$. Both of them can be used *together* to solve for x and y using a small trick called substitution, as we will see.

Let us start decomposing our derivatives: As we are looking for a minimum value of r^2, we are first going to set each $E'_a(a;b)$ and $E'_b(a;b)$ to zero, such that:

$$\sum_{i=1}^{n} 2 * (y_i - a - bx_i) * (-1) = 0$$

$$\sum_{i=1}^{n} 2 * (y_i - a - bx_i) * (-x_i) = 0$$

Now we get the factor 2 in front of the sum sign, as it does not affect the sum (**the sum sign looks spiky, but as it turns out it is actually harmless**). Contemplate on this and convince yourself via R (double "=" checks for equality; mark only one side of the equation and execute to obtain the single results):

```
2*sum(1:6) == sum(2*1:6)
```

We can also get rid of the factor 2 all at once without affecting the essential equation just by dividing the equations by two!

$$2 * \sum_{i=1}^{n} (y_i - a - bx_i) * (-1) = 0 \quad => \div 2$$

$$2 * \sum_{i=1}^{n} (y_i - a - bx_i) * (-x_i) = 0 \quad => \div 2$$

Resulting in:

$$\sum_{i=1}^{n} (y_i - a - bx_i) * (-1) = 0$$

$$\sum_{i=1}^{n} (y_i - a - bx_i) * (-x_i) = 0$$

Now we get rid of the product, (-1) and $(-x_i)$, *within the sum*:

$$\sum_{i=1}^{n} (-1) * y_i - (-1) * a - (-1) * bx_i = 0$$

$$\sum_{i=1}^{n} (-x_i) * y_i - (-x_i) * a - (-x_i) * bx_i = 0$$

Resulting in:

$$\sum_{i=1}^{n} -y_i \quad + a \quad + bx_i \; = 0$$

$$\sum_{i=1}^{n} -x_i y_i + a x_i + b {x_i}^2 = 0$$

Next, we can split up the sum to get closer to the appearance of a system of equations:

$$-\sum_{i=1}^{n} y_i \quad + \sum_{i=1}^{n} a \quad + \sum_{i=1}^{n} bx_i \; = 0$$

$$-\sum_{i=1}^{n} x_i y_i \; + \sum_{i=1}^{n} a x_i + \sum_{i=1}^{n} b {x_i}^2 = 0$$

Now we bring the negative sum of y_i and $x_i y_i$ to the right side of the equation:

$$\sum_{i=1}^{n} a \quad + \sum_{i=1}^{n} bx_i \; = \sum_{i=1}^{n} y_i$$

$$\sum_{i=1}^{n} a x_i \; + \sum_{i=1}^{n} b {x_i}^2 \; = \sum_{i=1}^{n} x_i y_i$$

Next, we do the same as with the factor 2 above, this time bringing the variable factors a and b outside of the sum. This will make things much easier in a bit, as x_i and y_i are variables that *are given to us* and that can be summed out easily. In the special case of $\sum_{i=1}^{n} a$, we will get $a * \sum_{i=1}^{n} 1$, which is essentially outwritten: $1 + 1 + . \; . \; .$, or more correctly: $x_{i=1} + x_{i+1} + . \; . \; . + x_{i=n}$, which can be abbreviated to the simple multiplication of $\sum_{i=1}^{n} 1 = 1 * n = n$, such that we will end up with the following:

$$a * n \quad + \; b \sum_{i=1}^{n} x_i \; = \sum_{i=1}^{n} y_i$$

$$a \sum_{i=1}^{n} x_i \; + \; b \sum_{i=1}^{n} {x_i}^2 \; = \sum_{i=1}^{n} x_i y_i$$

From here everything becomes a lot easier, again as x_i and y_i are given values, so we can actually compute some numbers in R.

```
# The above just represented in R (DO NOT EXECUTE IN THIS FORM)
#   sum(a*length(x)) + b*sum(x)    = sum(y)
#      a*sum(x)      + b*sum(x^2)  = sum(x*y)
```

Numerically, the result of applying our second synthetic data set to the upper formulas will leave us with the following system of equations (code will follow in a bit):

$$a * 33 \quad + \quad b * 165 \quad = \quad 174$$

$$a * 165 \quad + \quad b * 1155 \quad = \quad 1202.7$$

We have learned…

… how to easily decompose the partial derivatives of our error function $E(a, b)$ leaving us with a system of equations that can be used for any set of x and y in the future. Understanding the upper decomposition is therefore a once in a lifetime opportunity, leaving you with a general intermediate solution for linear models, such that a lot of our previous steps can be abbreviated in the future.

… that sum signs look spiky, but as it turns out it they are actually harmless.

2.4.4 Approaching Results: Solving our System of Equations

With the above **we have obtained our desired structure of a system of linear equations and are confidently approaching higher spheres of inference**. We can see two equations, each with the variables a and b, kept on the left side of the equations, and potentially single numeric values as result of each equation on the right-hand side, as all the values for x and y are given.

One might already have an idea on how to get to a result by substitution… For those who don't know, or only roughly remember what a system of linear equations is, let us go through a quick example below.

In general, systems of equations are used to solve a wide range of very basic questions in science. We will start with a very simple example that most of us probably have formally encountered in riddles or so, without thinking of a system of linear equations as such:

FURRY EXERCISE: Two cats, Mr. Midnight and Jurassica Parker, are together 15 years old. Their age deviates by 7 years – Jurassica Parker being the older sister of Mr. Midnight. How old is each of them?

Fig. 27 Jurassica Parker and Mr. Midnight (don't be fooled by the young looks of Jurassica, when solving the riddle — it may be a trap). Original photos by Avel Chuklanov and Dominika Roseclay.

We can solve this equation by *substituting* $y = x + 7$ within $x + y = 15$.

$$x + (x + 7) = 15$$

$$x + x = 15 - 7$$

$$x = 4$$

$$y = 4 + 7 = 11$$

$$Mr.\,Midnight = 4\ years\ old, \qquad Jurassica\ Parker = 11\ years\ old$$

Another way of solving our exercise is via linear algebra: we can write the left side of the equations as a matrix and the right side as a vector, resulting in: $\begin{bmatrix} x_1 + y_1 \\ -x_2 + y_2 \end{bmatrix} = \begin{bmatrix} 15 \\ 7 \end{bmatrix}$, which can be abstracted in R to: $\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 15 \\ 7 \end{bmatrix}$. We will now use R to solve this system of equations:

```
# Set up two elements, so that we can solve an
# equation in the form: left = right
left = matrix(c(1,1,
                -1,1), ncol = 2, byrow = TRUE)

right = matrix(c(15,7), ncol = 1, byrow = TRUE)

# Use the solve function to solve the system of
# equations. It outputs a vector list = c(x,y)
ResultXY = solve(left,right)
```

Let us now finally solve the system of equations regarding our second synthetic data set **by hand**:

$$a * 33 = 174 - b * 165 \quad => \div 33$$

$$b * 1155 = 1202.7 - a * 165 \quad => \div 1155$$

Resulting in:

$$a = 5.272727 - b * 5$$

$$b = 1.041299 - a * 0.1428571$$

Now we can smoothly insert b into $E'_a(a; b)$:

$$a * 33 + (1.041299 - a * 0.1428571) * 165 = 174$$

Resulting in:

$$a * 33 + 171.8143 - a * 23.57142 = 174$$

$$\boldsymbol{\alpha = 0.2318182}$$

And now we can insert a into $E'_b(a; b)$, in order to obtain b:

$$(5.272727 - b * 5) * 165 + b * 1155 = 1202.7$$

$$b * 330 = 332.7$$

$$\boldsymbol{\beta = 1.008182}$$

Whoop-whoop! The values for α and β are equivalent to the values we obtained using our regular `lm()` function! **You have now mastered your first linear regression by hand!** Let us now replicate the above fully via R and write our own function to perform a linear regression.

We have learned…

… how old Jurassica Parker and Mr. Midnight are.

… that a system of equations can be easily solved via substitution and can be done by hand, but that we can also use the **solve()** function in R. To do so we translated that part of the equations that holds the variables into a matrix and the results of our two equations into a vector.

3 Replicating the Rest of the Math in R and how to Write your own Linear ModelFfunction

Next up we will replicate all of the math above using R and subsequently write our own `linear_least_square()` function. In other words: we will replicate the essential *mathematical* part of the `lm()` function ourselves. You may be surprised that only a few lines of code are needed to do so. Apart from putting together all the math from previous chapters, we will also include some fancy code for some nice-looking console output. ^^

First of all, we will again define our random "guess function" and our x and y (meow!):

```
regRandom = function(x) (x)
x = data_table$time; y = data_table$cups
```

Next up we will optimize our guessed function, by looking for the minimum of the values of a (α) in relation to b (β). The formula of our PDE was:

$$E(a;b) = \sum_{i=1}^{n}(y_i - f(x_i))^2 = \sum_{i=1}^{n}(y_i - a - bx_i)^2$$

We have been to the part where we rearranged the derivatives to resemble a system of equations. Our insights can now be used to *abbreviate* the code that performs the math dramatically. To do so, we will also define `a` and `b`, due to an abstraction of linear algebra in R, when using the function `solve()` — you'll see in a bit.

```
a = 1; b = 1   # ";" marks a new line (delimitter)
```

Below you will find our rearranged system of equations that can be looked at as forming of a 2x2 matrix (left side) and a 2x1 (column-) vector (right side of the equation). We will use these objects as input for our `solve(left,right)` function. The formula of the rearrenged equations of our partial derivatives was:

$$a * n \quad + \quad b\sum_{i=1}^{n} x_i \quad = \quad \sum_{i=1}^{n} y_i$$

$$a\sum_{i=1}^{n} x_i \quad + \quad b\sum_{i=1}^{n} {x_i}^2 \quad = \quad \sum_{i=1}^{n} x_i y_i$$

In terms of code, the above can be written as:

```
# Formally in code (DO NOT EXECUTE):
# sum(a*length(x)) + b*sum(x)    = sum(y)
#     a*sum(x)     + b*sum(x^2)  = sum(x*y)
```

To get the above set for solving a system of linear equations we will create objects of the parts of the above and then fit them into a *matrix* and a *vector* in R. The `solve()` function will treat each element separately, so something like $a + b$ will not result in 2, when each of them is defined to have the value 1. We could spare the whole process of defining a and b and get equivalent results, but with the below we will exactly represent the formula above in the form of code (the function `length()` evaluates n).

```
# Named in the form r = row, c = column
left_a_r1c1 = sum(a*length(x)); left_b_r1c2 = b*sum(x) ;
left_a_r2c1 = a*sum(x)        ; left_b_r2c2 = b*sum(x^2)

right_r1c1 = sum(y)
right_r2c1 = sum(x*y)
```

Now we will set up the above as actual matrix (left) and vector (right) objects:

```
# matrix
left = matrix(c(left_a_r1c1 , left_b_r1c2,
                left_a_r2c1 , left_b_r2c2),
                       ncol=2, nrow=2, byrow = TRUE)
# vector
right = c(right_r1c1, right_r2c1)
```

Now we can finally solve our system of equations via:

```
Result = solve(left,right)
# [1] 0.2318182
# [2] 1.008182

# Fitted function:
fitfun = function(x) (Result[1]+Result[2]*x)

SumError2fitted = sum((y-fitfun(x))^2)
# [1] 9.703364
```

In order to get a nice looking output for our function below, we will use the `cat()` function. The `cat()` function *concatenates* elements of any kind in the console (!), such as text and our result values. Use `"\n"` for a line break, and `"\t"` for the tab separator. We also have to apply a trick, to get the name of the input object into our console output using `deparse(substitute(x))`:

```
# DO NOT EXECUTE. Code is exclusively meant for our function below:
# cat("Linear least square method in R","\n","\n",
# "Independent variable:", "\t", deparse(substitute(x)),"\n",
# "Dependent variable:", "\t", deparse(substitute(y)),"\n","\n",
# "alpha", "\t",Result[1],"\n",
# "beta","\t",Result[2], "\t","SumR2", "\t", SumError2fitted)
```

Let us now also plot our results. We will also use `deparse(substitute(x))` for the labels of our plot axes.

```
plot(x=x,y=y, ylab = deparse(substitute(y)),
                          xlab = deparse(substitute(x)))
abline(a=Result[1], b=Result[2], col = "darkblue")
```

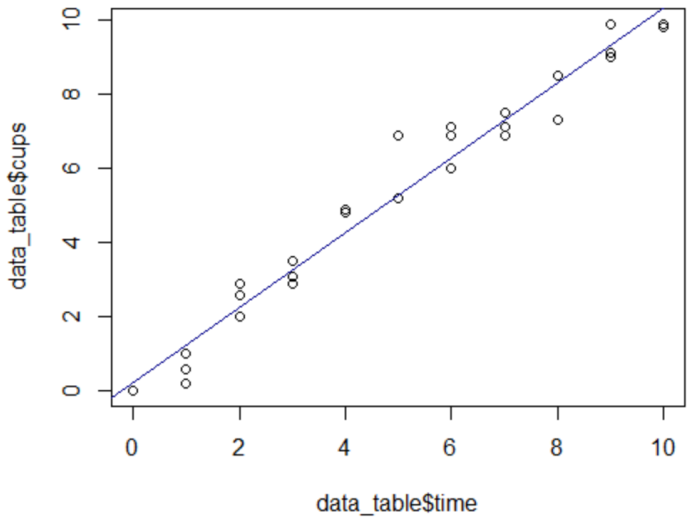


Fig. 28 Plot of our results — equivalent to the ones we have obtained using the **lm()** function.

Voilà, that was it! We can now put all of the above together, execute and subsequently use our own `linear_least_square()` function — **whoooooop!!!**

```
# Go-go-gadgeto linear_least_square!!!!
# Replication of the lm() function:

linear_least_square = function (x,y){ # Start of function
    # "Guess function"
    lmRandom = function(x) (x)

    # Setting up system of equations:
    left_a_r1c1 = sum(a*length(x)); left_b_r1c2 = b*sum(x)
    left_a_r2c1 = a*sum(x)        ; left_b_r2c2 = b*sum(x^2)
    right_r1c1 = sum(y)
    right_r2c1 = sum(x*y)

    # Now we will set up the above as matrix (left) and
    # vector (right) objects:
    left = matrix(c(left_a_r1c1 , left_b_r1c2,
                    left_a_r2c1 , left_b_r2c2),
                           ncol=2, nrow=2, byrow = TRUE)
    right = c(right_r1c1, right_r2c1)

    # Now we can solve our system of equations via:
    Result = solve(left,right)

    # Fitted function:
    lmFitted = function(x) (Result[1]+Result[2]*x)
    SumError2fitted = sum((y-lmFitted(x))^2)
```

```
    # Code for nice console output
    cat(" Linear least square method in R","\n","\n",
    "Independent variable:", "\t", deparse(substitute(x)),"\n",
    "Dependent variable:", "\t", deparse(substitute(y)),"\n","\n",
    "alpha", "\t",Result[1],"\n",
    "beta","\t",Result[2], "\t","SumR2", "\t", SumError2fitted)

    # Plot Results:
    plot(x=x,y=y, ylab = deparse(substitute(y)),
                  xlab = deparse(substitute(x)))
    abline(a=Result[1], b=Result[2], col = "darkblue")
} # End of function
```

We could actually also name our function "lm", though it would conflict with the built in `lm()` function, so we do not recommend to rename the above function in that way.

Let's see what our function can do:

```
# Our function using our second synthetic data set:
linear_least_square(data_table$time, data_table$cups)

# OUTPUT:
# Linear least square method in R
# Independent variable:   data_table$time
# Dependent   variable:   data_table$cups
# alpha  0.2318182
# beta   1.008182         SumR2 9.703364

# With our idelaized data set:
linear_least_square(time_passed, cups_of_tea)

# Compare with the regular old lm() function:
lm(data_table$cups~data_table$time)
```

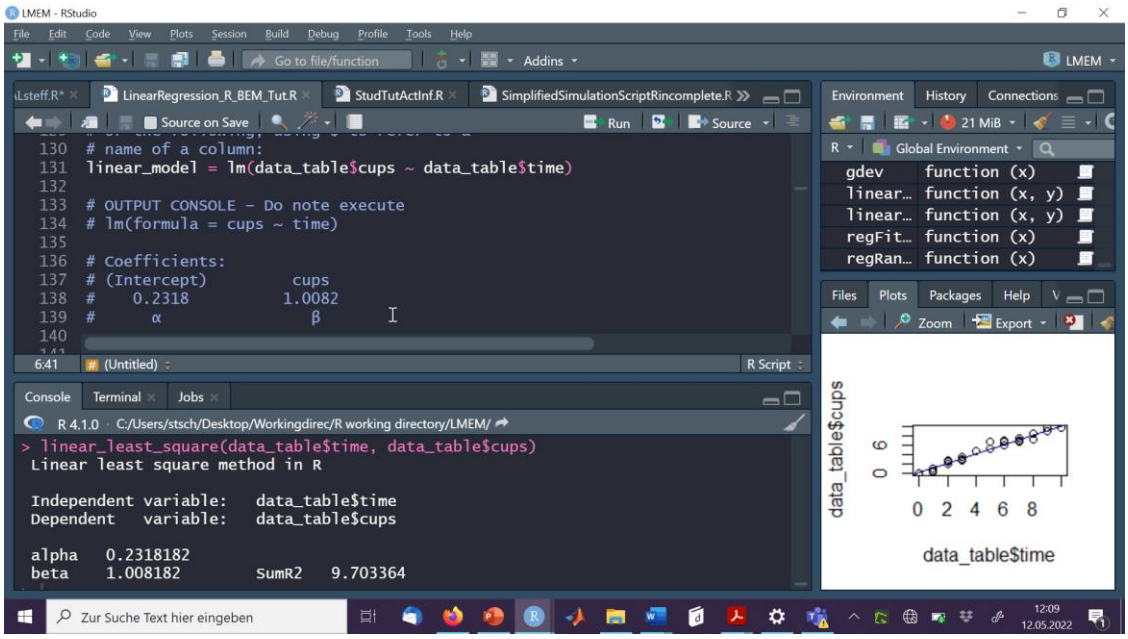


Fig. 29 Screenshot of RStudio. In the console you see the output of our function. Above in the script you see the output of the regular **`lm()`** function, marked as comment in the script. On the lower-right you see the plot of our function, equivalent to the output via **`plot(lm(y~x))`**.

Note that the actual `lm()` function includes much more features and the code will look quite different from the one we provided above, as it is, e.g., compatible with a wide range of input classes, and partially executes code in C (which runs faster). The function above is also not an official comparable function, such as `lm()` that can be cited (here you can find the original source code for the lm() function of the pre-installed library "stat": (4)**. It was written for didactic purposes only. However, the mathematics will be the same, when it comes to simple linear regression models.**

We hope this has given you a stable intuition and understanding of how a simple linear regression actually works. Multivariate models, linear mixed effect models and other forms of regression analysis are essentially based on the same principle. Importantly: the output of e.g. `glm()` or `lmer()` will look more or less the same. This article is therefore our basis for the following tutorials on that matter of regression analysis in general.

So far we have looked into the regular output of the `lm()` function. Next up we are going to look at the output of `summary(lm(y~x))` and will replicate and integrate its math into our own function.

We have learned…

… that finding a linear model that fits our data best involves the following steps:

a) Defining a "guess function" that we want to optimize, such as $f(x) = x$. We can use this function for any set of y and x.
b) Defining our optimization problem as an error or cost function $E(a, b)$, which can be minimized by setting its partial derivatives to zero. The decomposition of the resulting equations results in a system of equations that can be generalized and therefore used for any x and y (abbreviates processes).
c) Solving the obtained system of equations, e.g., via substitution, gaining the optimal a and b for our linear model function, corresponding to the least squared error (regarding y and x).

… that we can write a function in R, doing all of the above at once with just a few lines of code, such that we only have to define our variables and the function does the rest for us, equivalent to the built in **lm()** function in R.

… that we can generate a nice looking output using the **cat()** function in R.

… that linear modelling involves simple basics that we are actually quite familiar with, due to our time in school.

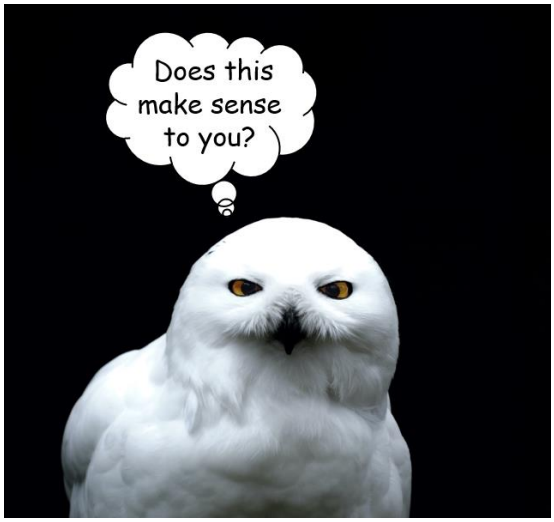


Fig. 30 Ms. Miranda, longing for feedback. Did any of this makes sense? We would like to hear from you! Similar to our open review process, **every of our articles can be commented by you**. Our journal still relies on the expertise of other student and professionals. However, the goal of our tutorial collection is especially to come in contact with you, helping us to create an open and transparent peer-teaching space within BEM.

References:

1. Bärtl, M. (2021). Kurzes Tutorium Statistik – Statistik-Videos auf YouTube. In *Konzepte und Studien zur Hochschuldidaktik und Lehrerbildung Mathematik* (pp. 29–

44). Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-33636-3_3

2. https://mathworld.wolfram.com/LeastSquaresFitting.html Some other approaches can be found here and served as another reference for this tutorial.

3. Here are some simple tutorials on differential quotients for the german readers:

   German: https://www.youtube.com/watch?v=52D8qCk1vGU

4. https://github.com/SurajGupta/r-source/blob/master/src/library/stats/R/lm.R

   Below you will find a link for the code of other functions such as wilcox.test(), t.test(). This is only if you are interested in how the original looks like. You have learned pretty much the most you need to know about writing and using functions for the future, so no worries when the source code looks a little dizzy.

   https://github.com/SurajGupta/r-source/tree/master/src/library/stats/R