In this chapter, we introduce the basics of Bayesian statistical modeling. Bayesian methods are not specific to ACT-R, or to cognitive modeling. They are a general framework for doing plausible inference based on data—both categorical (‘symbolic’) and numerical (‘subsymbolic’) data.

Introducing Bayesian methods at this point in a book about cognitive modeling for linguistics might seem like an unnecessary detour into a complex topic that is only tangentially related to our main goal here. Why would Bayesian methods be a necessary component of building fully formalized and integrated competence-performance/processing theories for generative linguistics in general, and formal semantics in particular?

They are not. However, we believe this detour to be necessary for a different reason. One of the two main goals of this book is to argue for integrated, fully formalized theories of competence and performance: theories that provide full mathematical formalizations that explicitly link the theoretical constructs generative linguists postulate in their analyses and the experimental data generated by widely used psycholinguistic methodologies.

But the second main goal of this book is to enable our readers to build these kinds of theories for themselves and use them in their own work. We want our readers to understand that linguistic theorizing can be more than what it is traditionally taken to be. But equally importantly, we want our readers to be able to _do_ more and build more encompassing, fully formalized and experimental-evidence based linguistic theories.

This is why we always provide all the gory details of our Python code throughout the book, although much of that is not immediately relevant to our main, linguistics-centered enterprise: we want our readers to be able to do everything we do, so that they can start using these ideas, frameworks and tools in their own work.

We are now at an important juncture in this book. In the next chapter, we will start introducing the subsymbolic components of ACT-R, which come with a good number of numerical parameters/‘knobs’. These parameters need to be dialed in to
specific settings based on (numerical) experimental data. For simplicity and clarity of exposition, we could choose to pull the correct settings and values out of thin air, and hand-wave in the general direction of statistical inference for the proper way to obtain these specific values. But this will satisfy only our first goal for this book. We would present a formalized theory of competence and performance, but our readers would not be able to reproduce our results, and would ultimately not be able to build better theories and ‘tune’ their numerical parameters on their own. Similarly, it would be impossible to compare several theories by tuning their parameters and finding out which theory better accounts for the data under consideration.

The Bayesian inference framework introduced in this chapter will enable us to: (i) learn the best settings for numerical parameters from the data, (ii) explicitly quantify our uncertainty about these settings, and (iii) do empirically-driven theory comparison. We will then be able to introduce the subsymbolic components of ACT-R, set the values of the numerical parameters associated with these components based on linguistic data, and numerically compare and evaluate different linguistic theories.

A final word for our readers with a largely non-technical background: it might seem that numbers, equations and code start flooding the pages at this point, and you might feel that the difficulties are insurmountable. They are not. You have to understand that the math we will use in this and the following chapters is actually pretty simple, and it is possible to follow the text by slowing down and googling around a bit. If you find things heavy going, just be enterprising and look online for the right kind of resource for you—the resource that will enable you to bridge the specific gaps you have in your background math knowledge. There are many such resources available now, from blog posts to videos of lectures/tutorials and fully-fledged online courses.

Another thing that you, dear reader, should keep in mind is that you do not have to fully understand everything we show you at this point. Understanding the main ideas and the general process of theory and model development is good enough for a first (or second) pass through the book. That’s why we want you to have the full code and be able to run it, modify it, play with it, and grasp how things work by seeing how the results change.

The approach that both of us (i.e., the authors) have gravitated towards more and more as teachers is: your daily activity as a budding (or mature, for that matter) researcher is tinkering within the horizon of a big question. The big question gives you a compass for where the research should be going, and how to change strategies when you get stuck. That’s where all the big words (‘developing integrated, fully formalized linguistic theories of competence and performance’) come in. The tinkering aspect is what makes research fun, doable and hands-on, and keeps it exciting and fundamentally open-minded and exploratory. That’s where the code and the instantaneous interaction with the Python interpreter come in.
5.1 The Python Libraries We Need

We first load the relevant Python libraries:

- **numpy** provides fast numerical and vectorial operations;
- **matplotlib** and **seaborn** provide plotting facilities; we adjust a variety of settings for matplotlib in [py1], but the default ones are very good too, and more useful if working interactively in the terminal, in which case you should comment out/ignore lines 4–17;
- **pandas** provides data frames, i.e., data structures well suited for data analysis, basically Excel sheets on steroids; similar to R data frames;
- **finally, pymc3** is the library for Bayesian modeling—Monte Carlo (MC) methods for Python3

```python
>>> import numpy as np
>>> import matplotlib as mpl
>>> mpl.use("pgf")
>>> pgf_with_pdflatex = {"text.usetex": True, "pgf.texsystem": "pdflatex", "pgf.preamble": [r"\usepackage{mathpazo}"
... r"\usepackage[utf8x]{inputenc}"
... r"\usepackage[T1]{fontenc}"
... r"\usepackage{amsmath}"], "axes.labelsize": 8,
... "font.family": "serif",
... "font.size": 8,
... "legend.fontsize": 8,
... "xtick.labelsize": 8,
... "ytick.labelsize": 8},
>>> mpl.rcParams.update(pgf_with_pdflatex)
>>> import matplotlib.pyplot as plt
>>> plt.style.use('seaborn')
>>> import seaborn as sns
>>> import pandas as pd
>>> import pymc3 as pm
```

5.2 The Data

We introduce Bayesian estimation methods by using a very simple data set from Brasoveanu and Dotlačil (2015c), which consists of reading times (RTs) associated with two quantifiers, namely *every* and *each*. The data is available in the pyactr-book github repo—see the ‘data’ folder.¹

We load the file using the read_csv method provided by pandas—see line 1 in [py2] below.² Line 2 in [py2] specifies that the "quant" (quantifier) variable should be considered categorical.

¹At this address: https://github.com/abrsvn/pyactr-book/blob/master/data/every_each.csv.
²If you are using Windows, you might have to modify the way you provide the path to the csv file. Similarly, you might need to modify the path if the relative path from your current working
We can look at the shape of our data (line 3 in [py2]) and we can list the first 3 rows of the data (line 5). We see that the data consists of 347 observations (rows) with respect to two variables (columns): "logRResid" (residualized log-transformed RTs) and "quant". We can also select several different rows/observations by using the iloc (integer-based location) method (line 10): we select rows [0, 8, 18, 31], and we display the values in all the columns (:).

```
>>> every_each = pd.read_csv("./data/every_each.csv")
>>> every_each["quant"] = every_each["quant"].astype('category')
>>> every_each.shape
(347, 2)
>>> every_each.head(n=3)
logRResid  quant
0  0.056128  each
1  0.241384  each
2  0.056128  every
>>> every_each.iloc[[0, 8, 18, 31], :]
logRResid  quant
0  0.056128  each
8  0.869077  every
18 -0.073706  every
31 -0.187536  each
```

This data is part of the results of Experiment 2 (a self-paced reading experiment) reported in Brasoveanu and Dotlačil (2015c). The experiment investigates the hypothesis formulated in Tunstall (1998) that the distinct scopal properties of each and every are, at least to some extent, the consequence of an event-differentiation requirement contributed by each. By scopal properties, we mean the preference of these quantifiers to take wide scope over another quantifier in the same sentence.

Consider the examples in (1) and (2) below:

1. A helper dyed every shirt without thinking about it.
2. A helper dyed each shirt without thinking about it.

The quantifier phrases every/each shirt can take wide or narrow scope relative to the indefinite a helper in subject position. On the wide scope reading, the sentences in (1)/(2) are taken to mean that every/each shirt was dyed by a possibly different helper. We also call this reading the inverse scope reading because the scope of the quantifiers is the inverse of their surface order. On the narrow scope reading, also known as the surface scope reading (for obvious reasons), the sentences in (1)/(2) are taken to mean that the same helper dyed every/each shirt.

On the face of it, both every and each have the same meaning: they contribute so-called universal quantificational force—as opposed to indefinites like a or some, which contribute existential quantificational force. However, Tunstall (1998) (see also references therein) notices that each, but not every, require a separate event for each element it quantifies over.

For example, the sentence Jake photographed every student in the class, but not separately is perfectly acceptable, while the minimally different sentence Jake...
photographed each student in the class, but not separately, where each is substituted for every, is definitely less acceptable.

Based on contrasts like this, Tunstall (1998, 100) proposes that each contributes a differentiation condition to the effect that “[e]ach individual object in the restrictor set of the quantified phrase must be associated with its own subevent, in which the predicate applies to that object, and which can be differentiated in some way from the other subevents.”

There are many ways in which events can be differentiated from one another, but one way, relevant for our sentences in (1)/(2) above, is for each to take inverse scope. In that case, each shirt is dyed by a (possibly) different helper, which ensures that each shirt-dyeing event is differentiated from all others because of the different person doing the dyeing.

Thus, if each contributes an event-differentiation requirement, unlike every, we expect it to have a higher preference for inverse scope than every. And since inverse scope is known to lead to processing difficulties (Kurtzman and MacDonald 1993; Tunstall 1998; Anderson 2004; Pyllkänen and McElree 2006 among many others), which manifest themselves as increased RTs, we expect to see higher RTs for sentence (2) relative to (1).

In their Experiment 2, Brasoveanu and Dotlačil (2015c) test this prediction using a moving-window self-paced reading task (Just et al. 1982). Because the experiment included a separate manipulation, the most important regions of interest (ROIs) were the spillover words immediately following the universal quantifier phrase. Specifically, in examples (1)/(2) above, these ROIs were the words without, thinking and about. The data set we have just loaded in Python and assigned to the every_each variable contains measurements collected for the third ROI about.

The RTs collected for the ROI about were transformed in a couple of ways. Raw reading times in self-paced reading experiments are roughly between 300 and 600 ms per word. These raw reading times were first log-transformed, which yields log RTs roughly between 5 and 7—see lines 1–4 in (py3) below. We discuss log transformation/log compression in more detail in the next chapter.

In addition, following Trueswell et al. (1994), Brasoveanu and Dotlačil (2015c) residualized the log RTs by factoring out the influence of word length and word position. This yields residualized log RTs that are roughly between −3 and 3. In fact, in this particular case, they fall just between −1 and 2, as we can see when we inspect the minimum and maximum of the residualized log RTs in our data—see lines 5–8 in [py3].

```
py3 >>> np.log(300) 1
5.703782474656201
2 >>> np.log(600) 3
6.396929655216146
4 >>> np.min(every_each["logRTresid"]) 5
-0.678407840683957
6 >>> np.max(every_each["logRTresid"]) 7
1.19278354190761
8```
The main question we want to ask of this data set is the following: are the reading times, specifically in the form of residualized log RTs, different for the two quantifiers every versus each?

That is, we will model RTs as a function of quantifier. One way to model RTs as a function of quantifier is to estimate the two means for the two quantifiers: we can estimate the means and our uncertainty about them, that is, we estimate two full probability distributions, one for each of the means.

But estimating the mean RTs for the two quantifiers will not give us a direct answer to our question: is there a difference in RTs between the two quantifiers? In a Bayesian framework, we could still answer the question given a two-mean model, but it is more straightforward (and closer to the way frequentist estimation would be done) to estimate the difference between the two quantifiers directly.

Thus, in our model, we estimate the mean RT for every (together with our uncertainty about it), and instead of estimating the mean RT for each, we estimate the mean difference between the every RTs and each RTs (together with our uncertainty about it). We can still obtain our mean RT for each by starting with the mean for every and adding to it the mean difference in RTs between the two quantifiers.

If we want to answer our question—are the RTs different for every versus each?—we basically look at our probability distribution for the difference in RTs and check if 95% of the probability mass is away from 0.3

It is important to pause at this point and realize that the reasoning we are going through here has a different structure than the kind of reasoning and arguments linguists are familiar with. From very early on in our linguistic training, we are presented with some data, we automatically assume there is a pattern in the data, and we try to identify the pattern and build a theory to capture it. In contrast, our first job as empirically-driven statistical modelers is to ask: is there really a pattern in the data? how sure are we that we’re not hallucinating regularities/signal in what is actually pure noise?

This kind of skepticism is actually familiar to linguists in other forms. For example, it is never clear at the outset whether a meaning-related phenomenon (e.g., licensing negative polarity items like any or ever) should receive a syntactic analysis (Klima 1964), which might seem the ‘obvious’ way to go, or a semantic one (Ladusaw 1979). As linguists, we know all too well that it is important to be skeptical about the assumptions we make as we build theories.

But it is equally important to be skeptical about the assumptions we make when we identify ‘obvious’ generalizations and patterns in the data. All data (even introspective data) is ultimately behavioral data, i.e., a product of a performance system, never a direct expression of the unobservable competence system hypothesized to be at the ‘core’ of the performance system. As such, we need to be reasonably skeptical about all the generalizations and patterns we think we see in the behavior of the system.

---

3This is an oversimplification. See, for example, Kruschke (2011), Nicenboim and Vaisishth (2016) and references therein for more discussion of posterior distributions, credible intervals, hypothesis testing and related issues in a Bayesian framework.
5.2 The Data

Therefore, our question about the quantifier data set is unpacked as follows: (i) can we actually show with enough credibility that the RTs actually differ between the two quantifiers (every and each)? Assuming we can, (ii) what is the magnitude of the difference and what is our uncertainty about that magnitude? The answer to question (ii) should be of the form: given both our prior knowledge about the phenomenon under discussion and the experimental data, the mean difference between the RTs of the two quantifiers is $x_{\text{mean}}$, and we’re 95% certain that the magnitude of the difference is somewhere in the interval $(x_{\text{lowerlimit}}, x_{\text{upperlimit}})$.

Let’s now turn to specifying the actual model, and more of this will start making sense. The model we are about to specify is called a t-test, or a linear regression with one binary categorical predictor.

5.3 Prior Beliefs and the Basics of pymc3, matplotlib and seaborn

The way Bayesian estimation works is basically as follows. We start with a prior belief about the quantities of interest, in our case: the RTs for every, and the difference in RTs between each and every. ‘Prior’ beliefs means that these are our beliefs before we see the data. Furthermore, these beliefs take the form of full probability distributions: we say what values are possible for the quantities of interest, and in addition, we say which of the possible values are plausible (before we see the data).

We then update these prior beliefs with the data—specifically, the data stored in the "logRTresid" and "quant" columns in our data set. Upon exposure to the data, we shift/update our prior beliefs in the direction of the data, and the result of this update consists of two posterior probability distributions, one for the mean RT for every, and the other for the difference in RTs between the two quantifiers.

Our posterior beliefs/posterior probability distributions are a weighted average of our prior beliefs, on one hand, and the data, on the other hand. If our prior beliefs are very strong (not the case here; see below), the posterior beliefs will reflect the data to a smaller extent. If we have a lot of data, and the data has low variability, the posterior beliefs will reflect the data to a larger, or even overwhelming, extent.

We have very weak prior beliefs about the quantities of interest. Let’s characterize them. Before even looking at the specific residualized log RTs associated with each and every, we know that self-paced reading RTs are usually between 300 and 600 ms, since participants read about 3 words per second. After log-transforming and residualizing these RTs with respect to word length and word position, we get very small values clustered around 0 and spanning the $(-3, 3)$ interval (actually, the $(-2, 2)$ interval) most of the time.

In sum, residualized log RTs for word-by-word self-paced reading are very rarely, if ever, larger than 3 in absolute value. Note that we derived these limits from considerations about word-by-word self-paced reading experiments in general, and the

---

4In residualized log ms, admittedly a non-intuitive temporal unit, which we will omit from now on.
various transformations we apply to the resulting raw RTs (namely, log + residualization).

Therefore, a very reasonable—in the sense of very weakly constraining/very low information—prior for the mean RT for every would be a normal (Gaussian) distribution centered at 0 and with a standard deviation (which is a measure of dispersion) of 10. This prior belief effectively says that the mean RT for every (after log transformation and residualization, as we already indicated) is, as far as we are concerned and before seeing the data, anywhere between about $-30$ and $30$ ($\pm 3$ standard deviations from the mean), most likely somewhere between $-20$ and $20$. Values below $-30$ or above $30$ are possible, but really unlikely.

We call this prior a very low information prior in the sense that it very weakly constrains the range of data we expect to see when we finally look at the experimental data. From prior considerations, we know that this data is very likely in the $(-3, 3)$ interval, and our prior very liberally allows for values in the $(-30, 30)$ interval.

Let’s plot a normal distribution with a standard deviation of 10. We’ll take this opportunity to introduce the basics of pymc3 models. In [py4] below, we first create a model `every_each_model` using the `pm.Model()` method (line 1).

Then, we start specifying the model components. In our case, we are simply interested in a normally distributed variable called ‘`normal_density’`. We create this variable with the corresponding `pymc3` probability density function (line 3). When we call this function, we need to specify the name of the variable (needed for `pymc3`-internal purposes), and then provide the parameters for the probability density function: the mean of the normal is set to 0 and the standard deviation is set to 10.

We then ask `pymc3` to sample 5000 draws from this distribution using some sampling magic that we won’t worry about,\(^5\) and save the results to a separate database `db` (lines 7–9 in [py4]). We specify the type of database that will store the draws to be `Text`. To save and load this type of database, we need to import two additional convenience functions from `pymc3` (lines 5–6 in [py4]).

Lines 7–9 in [py4] are commented out because we do not want to execute them. Executing them takes a while with the default sampling procedure for `pymc3`. Instead, we load the samples from a previous run of the model, as shown on lines 12–13. The samples we load here are available in the `pyactr-book` github repository, as are the samples (a.k.a. draws, or traces, or posterior estimates) for all the other Bayesian models in this book.\(^6\) But you can also generate your own samples; to do that, simply uncomment lines 7–9 and run them.

\(^5\)See Lynch (2007); Kruschke (2011); Lambert (2018) among others for detailed and clear discussions.

\(^6\)The traces/posterior samples for all the models in the book are available in the ‘data’ folder here: https://github.com/abrsvn/pyactr-book/tree/master/data. The samples for the model under discussion are available here: https://github.com/abrsvn/pyactr-book/tree/master/data/normal_trace.
Finally, we turn to plotting the results. We start by defining a plotting function on line 15 in [py4]. On line 16, we initialize the figure by indicating that we will have one plot, or ‘axis system’, which we call ax. This plot will be displayed in a grid consisting of 1 column and 1 row (n_cols=1, n_rows=1); this specification is superfluous here, but it is useful when we have more than one plot to display. The total figure size is set to 5.5 by 3.5 in. Lines 18–19 plot the histogram of the normal samples and label the plot accordingly. Lines 20–23 tighten up/suitably shrink all the blank margins in the figure and save it in various formats.

As we already mentioned, this entire procedure is assembled under a function `generate_normal_prior_figure`, which is called on line 25 of [py4]. The result is provided in Fig. 5.1. As expected for a normal density centered at 0 and with a standard deviation of 10, most of the probability mass (area under the curve) is spread over the \((-30, 30)\) interval, i.e., within \(\pm 3\) standard deviations of the mean.

We will use this normal density as our very weak, noncommittal prior for the mean RT for *every*.

We can now turn to specifying the prior for the difference in RTs between *every* and *each*. This difference could be positive (the mean RT for *each* is greater than the one for *every*), negative (the mean RT for *each* is less than the one for *every*), or 0 (the mean RTs for *each* and *every* are the same). Whether 0, negative or positive, this difference cannot be larger than 6 in absolute value.

To see this, recall that log RT residual values are roughly between \(-3\) and 3 (and are usually very close to 0). In the unlikely event that the mean RT for *each* happens to be 3 and the one for *every* happens to be \(-3\) (or the other way around), we’ll get a difference of 6 in absolute value. This is already very unlikely, and the a priori probability of even more extreme values for the difference in RTs is practically 0.

It is therefore reasonable to, once again, specify our prior for the difference in RTs as a normal distribution with a mean of 0 and a standard deviation of 10.

One final remark about the plot in Fig. 5.1. The shaded area under the dark-blue curve is probability mass and the curve itself plots probability density. That is, for any point on the x-axis, the height of the curve at that point (i.e., the corresponding value on the y-axis) does not plot the probability of that x-axis value: the height of
the curve does not provide the probability mass associated with that $x$-axis value.\footnote{There is 0 probability mass associated with any single point on the $x$-axis. Just as single points on the real line do not have length, i.e., they have 0 length, single points on the real line have 0 probability associated with them. Only intervals on the real line can have length. Similarly, only intervals can have a non-0 amount of probability mass associated with them.}

Since they are density curves, their values can exceed 1 (we will in fact see such a case when we estimate posterior probabilities for our every/each model). In contrast, probabilities, i.e., probability masses, can never exceed 1.

### 5.4 Our Function for Generating the Data (The Likelihood)

Now that we specified our priors, we can go ahead and specify how (we think) nature generated the data. For this purpose, we need to mathematically specify how RT is a function of quantifier.

What we conjecture as our ’data-generating’ function (technically, our likelihood function) is that we have two mean RTs for the two quantifiers every and each. For each of the two quantifiers, the RTs we observe are imperfect reflections of the mean RT for that quantifier, that is, they are somewhere around the mean for that quantifier. More specifically, the observed RTs for a quantifier are composed of the mean RT for every/each model. In contrast, probabilities, i.e., probability masses, can never exceed 1.

The height of the curve at an $x$ value does not indicate the probability mass at that value, but the probability density in an infinitesimal interval around that value. That is, we take an infinitesimal interval around the $x$ value, we measure the probability mass sitting on top of that interval and we divide that mass by the length of the interval. In the limit, i.e., as the length of the infinitesimal interval around the $x$ value goes to 0, this ratio, namely \( \frac{\text{probability mass}}{\text{interval length}} \), gives us the probability density at that point—and this is what is plotted by the dark-blue curves.
that quantifier plus some error, which is caused by our imperfect measurement, the fact that a participant was faster pressing the space bar on one occasion than another etc.

Plotting the RTs by quantifier will make this clearer. On line 1 in [py5] below, we check to see how many observations we have for each quantifier. We see that we have roughly the same number of observations for every and each—around 170, for a total of 347 observations. We define a function to generate a plot of these 347 observations by quantifier on lines 6–16, and call this function on line 18.

```python
>>> every_each["quant"].value_counts()
    Name: quant, dtype: int64
    every    173
    each    174

>>> def generate_RTs_by_quant_figure():
    fig, ax = plt.subplots(ncols=1, nrows=1)
    fig.set_size_inches(5.5, 3.5)
    g = sns.stripplot(x="quant", y="logRTresid", data=every_each,
                      jitter=True)
    g.set_xlabel("Quantifier")
    g.set_ylabel("Residualized log RTs")
    plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
    plt.savefig('./figures/RTs_by_quant.eps')
    plt.savefig('./figures/RTs_by_quant.png')
    plt.savefig('./figures/RTs_by_quant.pdf')
    >>> generate_RTs_by_quant_figure()
```

The plot, provided in Fig. 5.2, shows that the approx. 170 observations collected for each are centered somewhere around 0.05, while the approx. 170 observations collected for every are centered a little lower, around −0.05. The observations are jittered (jitter=True on line 10 in [py5]), that is, they are not plotted on a straight line so that we can distinguish overlapping points in the plot. The observations for each are generated from the mean RT for each (which is, say, around 0.05) plus some error/noise around that mean RT. The noise is pretty substantial, with observed RTs spread between about −0.75 and 0.75. Similarly, the observations for every are generated from the mean RT for every (which seems to be around −0.05) plus some error/noise around that mean RT. Once again, the noise is substantial, spreading the observed values mostly between −0.75 and 0.75.

Our job right now is to write this story up in a single formula that will describe how the 347 RTs depend on quantifier. Furthermore, recall that we are interested in the difference between the two quantifiers: we want to estimate it so that we can say whether this difference is likely different from 0, i.e., whether the mean RT for each is different from the mean RT for every, as Tunstall’s differentiation condition would predict. To this end, we will estimate two quantities:

- the mean RT for every: $RT_{\text{every}}$
- the mean difference in RT between each and every: $RT_{\text{each}−\text{every}}$

With these two quantities in hand, we can obtain the mean RT for each in the obvious way: $RT_{\text{each}} = RT_{\text{every}} + RT_{\text{each}−\text{every}}$.

We will now use a simple reformulation of the quant variable (called ‘dummy coding’ of the categorical predictor variable every_each["quant"]) to be able
Fig. 5.2 Plot of residualized log RTs by quantifier

to write a single formula describing how all 347 RTs are a function of the quantifier they are associated with.

The idea is to rewrite the quant variable as taking either a value of 0 or a value of 1, depending on whether the RTs are associated with every or each. We then multiply this rewritten/dummy-coded quant variable with the $RT_{each-every}$ difference as follows:

(3) Formula for RT as a function of quantifier:
\[
RT = RT_{every} + quant \cdot RT_{each-every} + noise
\]

a. If $RT$ is associated with every, our dummy-coding for quant says that \( quant = 0 \). Therefore, the RT is generated from the mean RT for every plus some noise:
\[
RT = RT_{every} + 0 \cdot RT_{each-every} + noise = RT_{every} + noise
\]

b. If $RT$ is associated with each, our dummy-coding for quant says that \( quant = 1 \). Therefore, the RT is generated from the mean RT for each plus some noise:
\[
RT = RT_{every} + 1 \cdot RT_{each-every} + noise = RT_{each} + noise
\]

The code for the dummy coding of the quant variable is a one-liner (line 1 in [py6] below). This takes advantage of the vectorial nature of both data and operations in numpy/pandas. The resulting "dummy_quant" variable, as expected, recodes each as 1 and every as 0:
5.4 Our Function for Generating the Data (The Likelihood)

We can now use the variable `every_each["dummy_quant"]` and the likelihood function in (3) to generate synthetic datasets. In [py7] below, we set our mean RT for `every` to $-0.05$ and our mean difference in RT to $0.1$ (lines 1–2). This will result in a mean RT of $0.05$ for `each`. For convenience, we extract the dummy-coded `dummy_quant` variable and store it separately (line 3 in [py7]).

We then assemble the means for the 347 synthetic observations we want to generate: line 4 in [py7] directly implements the likelihood function in (3). We can look at the first 25 means thus assembled (lines 5–8 in [py7] below). For example, the first two are mean RTs associated with `each`, since the first two observations in our original data set are associated with `each` (see lines 5–6 in [py6] above). Their mean RT is therefore $0.05$. The third observation is associated with `every` since the third observation in our original data set was associated with `every` (see line 7 in [py6] above). Its mean RT is therefore $-0.05$. And so on.

```python
>>> mean_every = -0.05
>>> mean_difference = 0.1
>>> quant = np.array(every_each["dummy_quant"])
>>> synthetic_RT_means = mean_every + quant * mean_difference
>>> synthetic_RT_means[:25]
array([ 0.05, 0.05, -0.05, 0.05, -0.05, -0.05, 0.05, 0.05, -0.05, 0.05, 0.05, -0.05, 0.05, -0.05, -0.05, 0.05, -0.05, 0.05, -0.05, 0.05, -0.05, 0.05, -0.05, 0.05, -0.05, 0.05, -0.05, 0.05, -0.05, 0.05, -0.05, 0.05])
>>> sigma = 0.25
>>> synthetic_RTs = np.random.normal(synthetic_RT_means, sigma)
>>> synthetic_RTs.round(2)[:25]
array([-0.21, -0.01, 0.26, 0.24, -0.15, 0.08, 0.04, -0.11, -0.04, 0.02, 0.87, 0.04, 0.09, -0.02, 0.18, -0.49, -0.04, 0.17, -0.28, -0.16, -0.07, -0.18, -0.13, -0.27, 0.14, -0.34, 0.08])
```

The likelihood function in (3) has one final component, namely the noise: RTs from a specific quantifier are only imperfect, noisy reflections of the mean RT for that quantifier. The noise comes from variations in the measuring equipment (keyboard etc.), or variations in the way the participants press the space bar at different times, or any other factor that we are not controlling for.

We generate noisy observations by drawing random numbers from a normal distribution: we use the `numpy` function `random.normal` for this purpose (line 10 in...
The mean of the normal distribution is the mean RT for one quantifier or the other, and the standard deviation is set to 0.25, which generates noise of about \pm 0.75 (lines 9–10 in [py7]). The resulting RTs are randomly generated real numbers (lines 11–14). We can compare them to the actual RTs, extracted and stored in an independent variable \( \text{RTs} \) (lines 16–20). We see that the range of variation in the synthetic data is pretty similar to the actual data.

Finally, if we want to synthesize more RT datasets that are similar to our actual dataset, we can simply do another set of draws from a normal distribution centered at \(-0.05\) or \(0.05\) (depending on the quantifier) with a standard deviation of 0.25 (lines 22–26 in [py7]).

Now that we understand that the likelihood function has to incorporate some noise, which needs to be estimated from the data, we need to set up a prior for this noise. Reasoning again from our prior knowledge about residualized log RTs, we know that this noise/variability in data cannot really be larger than maybe about 3.

This can be justified as follows. We know that residualized log RTs are between about \(-3\) and 3. Now, if we think of them as being generated from a normal distribution centered somewhere in the interval \((-3, 3)\), a standard deviation (i.e., a noise setting) of about 3 for this normal distribution would very easily cover the interval \((-3, 3)\). This is because a normal distribution spreads 99% of its probability mass within \pm 3 standard deviations from its mean.

So, if the normal distribution that generates noisy RTs is centered at 3 and has a standard deviation (noise) of 3, it will spread most of its probability mass between about \(-6\) and 12. This will easily cover the interval \((-3, 3)\). A similar conclusion is reached if we assume that the normal distribution that generates noisy RTs is centered at the other extreme of the interval, namely \(-3\). With a standard deviation of 3, it will spread its probability mass between about \(-12\) and 6, once again easily covering the interval \((-3, 3)\), within which we know from prior considerations that most residualized log RTs live.

Therefore, a very weak and non-committal prior for residualized log RT noise would be a half-normal distribution centered at 0 and with a standard deviation of 10. A half-normal distribution is a normal (Gaussian) distribution centered at 0 and ‘folded over’ so that all the probability mass over negative values gets transferred to the corresponding positive values. Half-normal distributions correctly require noise/dispersion to be positive.

If we set the standard deviation of this half-normal prior for noise to 10, we place practically no constraints on the actual value of the noise before we see the data: as far as we a priori expect, the noise can be anywhere between 0 and about 30, a very diffuse interval that allows for much larger values than 3, which we already argued would be good enough.

However, since this prior assigns higher probabilities to lower values than to larger values, as we can see in Fig. 5.3, we do expect the noise to be smaller rather than larger. This makes sense: even though values larger than 3 for residualized log RT noise are possible, such values are unlikely and they are more and more unlikely as they get bigger and bigger.
5.4 Our Function for Generating the Data (The Likelihood)

To plot a half-normal prior, we can simulate draws from it in the same way we did for the normal priors for the mean RT for *every* and the mean RT difference between *each* and *every*. We do this in [py8] below, and plot the results in Fig. 5.3.

```python
>>> every_each_model = pm.Model()
>>> with every_each_model:
...     half_normal_density = pm.HalfNormal('half_normal_density', sd=10)
... 
>>> # we load the results / trace of a previous run
>>> with every_each_model:
...     trace = load('./data/half_normal_trace')

>>> def generate_half_normal_prior_figure():
...     fig, ax = plt.subplots(ncols=1, nrows=1)
...     fig.set_size_inches(5.5, 3.5)
...     sns.distplot(trace['half_normal_density'], hist=True, ax=ax)
...     ax.set_xlabel('Half-normal density, standard deviation = 10')
...     plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
...     plt.savefig('./figures/half_normal_prior.eps')
...     plt.savefig('./figures/half_normal_prior.png')
...     plt.savefig('./figures/half_normal_prior.pdf')

>>> generate_half_normal_prior_figure()
```

**Fig. 5.3** A half-normal probability density

![Half-normal density, standard deviation = 10](image)
5.5 Posterior Beliefs: Estimating the Model Parameters and Answering the Theoretical Question

With the priors and likelihood in hand, we can finally ask `pymc3` to give us the posterior distributions for the quantities of interest, namely `mean_every` and `mean_difference`. We specify the full model as shown in [py9] below:

```python
>>> every_each_model = pm.Model()
>>> with every_each_model:
...    # priors
...    mean_every = pm.Normal('mean_every', mu=0, sd=10)
...    mean_difference = pm.Normal('mean_difference', mu=0, sd=10)
...    sigma = pm.HalfNormal('sigma', sd=10)
...    # likelihood
...    observed_RTs = pm.Normal('observed_RTs', mu=mean_every + quant*mean_difference, sd=sigma, observed=RTs)
```

Our assumptions for this model—and for Bayesian models in general—fall into two classes: (i) the assumptions we need to specify the priors, (ii) the assumptions we need to specify the likelihood (how the data is generated). We discuss them in turn.

The priors for our model are specified on lines 4–6 in [py9].

Our prior for the mean RT for `every` is very weak/low information, as already discussed. Recall that in normal and half-normal distributions, 99% of observations fall within 3 standard deviations from the mean, so the prior on line 4 allows for mean (residualized log) RTs anywhere between $-30$ and $30$, an extremely wide range given that log RT residuals are very close to 0 and usually fall within the $(-3, 3)$ interval.

Our prior for the difference in RT between `each` and `every` is similarly weak/low information (line 5): the difference can be positive (higher mean RT for `each` than `every`), negative (lower mean RT for `each` than `every`), or 0 (same mean RT for `each` and `every`). The differences this prior allows for fall anywhere between $-30$ and $30$, again an extremely wide range given that the maximum difference between two residualized log RTs is usually at most 6 in absolute value, and commonly much smaller.

Finally, the prior for our noise distribution on line 6 allows for values anywhere between 0 and 30. This is a very non-committal range since the standard deviation of residualized log RTs is usually around 1, generating a range of residualized log RTs between $-3$ and $3$ if these RTs are centered around 0 and approximately normally distributed.

The likelihood (the data generation part of the model) is specified on lines 8–11 in [py9]. We indicate that we model the observed data by explicitly specifying on line 11 that these values are observed, and providing the variable that stores these
observed values. In this case, the observed values are stored in the variable RTs, which we introduced on line 16 of [py7].

On line 9 of our model specification in [py9], we say that each RT is somewhere near the mean RT for the corresponding quantifier. If the RT is associated with the quantifier every, our dummy variable quant takes the value 0, so line 9 reduces to \( \mu = \text{mean}_{\text{every}} \). If the RT is associated with the quantifier each, the dummy variable quant takes the value 1, so line 9 reduces to \( \mu = \text{mean}_{\text{every}} + \text{mean}_{\text{difference}} \), that is, the mean RT for each.

As we already discussed above, each RT is an imperfect, noisy reflection of the mean RT for the corresponding quantifier, so we add some normally distributed noise to that mean RT to obtain the actual RT. This normally distributed noise has a standard deviation \( \sigma \) (line 10). We do not know how large the noise is, so this will be the third parameter we estimate from the data, in addition to our parameters of primary interest \( \text{mean}_{\text{every}} \) and \( \text{mean}_{\text{difference}} \).

At the end of the day, our assumptions about the priors and the likelihood lead us to using two types of probability distributions: (i) normal distributions, which come with two parameters (mean and standard deviation) that we can tweak to rearrange the way the probability mass is spread over the entire real line; and (ii) half-normal distributions, which are by default assumed to cover the entire positive part of the real line, and which come with only one parameter (standard deviation) that manipulates the spread of probability mass over the positive real numbers.8

To summarize, we have three parameters we want to learn about from the data:

i. the mean RT for every (mean_every),
ii. the mean difference in RT between each and every (mean_difference),

and

iii. the magnitude of the noise/dispersion of the actual RTs around the mean RT for the corresponding quantifier (\( \sigma \)).

All these three parameters contribute in essential ways to the likelihood, i.e., to the way we think the observed RT data was generated (lines 8–11 in [py9]). And we need prior distributions for each of these three parameters (lines 4–6 in [py9]), so that we can update the prior distributions with our observed RT data. These prior distributions have a total of five hyperparameters (two means and three standard deviations) that we need to set, and we have set them all to values that are very non-committal. This is why we called them weak, low information priors.

We finally run the model on lines 13–15 in [py9], which means that we ask pymc3 to compute for us the posterior distributions for the two quantities that are of primary interest to us, namely mean_every and mean_difference. As before, we do not actually run the model interactively here, but load the samples from a previous run of the model (lines 17–18). And as before, these samples are available in the pyactr-book github repo.9

---

8We assume that half-normal distributions are always obtained by ‘folding over’ normal distributions centered at 0, so we do not specify a location parameter for them.

9See here: https://github.com/abrsvn/pyactr-book/tree/master/data/every_each_model_trace.
We then plot the resulting posterior distributions. The plotting code is provided in [py10] below and the resulting plots are shown in Fig. 5.4. The posterior distributions plotted in Fig. 5.4 are much more constrained than the very ‘loose’ prior distributions we specified, which indicates that we learned a lot from the data. That is, these posterior distributions mostly reflect the data and not our priors (which were very weak).

Specifically, the posterior mean for the *every* mean RT is around $-0.05$ and, after seeing the data, we know that this mean RT is roughly between $-0.1$ and $0$. This is much stronger than our prior assumptions, which stated that the mean RT for *every* could be anywhere between $-30$ and $30$.

Similarly, after seeing the data, we know that the RT difference between *each* and *every* is probably around $0.07$, and we’re very confident this difference is between about $0$ and $0.15$. That is, we are fairly confident that the mean RT for *each* is larger...
than the mean RT for *every*. Once again, this is much more constrained than our prior, which countenances differences as small as $-30$ and as large as $30$.

We can ask *pymc3* to examine the posterior distribution of the mean RT difference and tell us precisely where the central 95% portion of the probability mass is$^{10}$:

```python
>>> mean_difference = trace['mean_difference']
>>> pm.hpd(mean_difference).round(2)
array([0.01, 0.12])
```

The resulting interval is our 95% credible interval: after seeing the data, we are 95% confident that the difference in (residualized log) RTs between *each* and *every* is positive and is somewhere between 0.01 and 0.12.

We can also compute the means and standard deviations (a.k.a. standard errors) of these posterior distributions:

```python
>>> mean_difference.mean().round(2)
0.07
>>> mean_difference.std().round(2)
0.04
>>> mean_every = trace['mean_every']
>>> mean_every.mean().round(2)
-0.05
>>> mean_every.std().round(2)
0.03
```

We see that these numbers are close to the frequentist ones we would get using a standard function in a standard statistical software environment, e.g., the function `lm()` in R:

```
(4) Output from R’s `lm` function for comparison with our Bayesian estimates:

every_each = read.csv("./data/every_each.csv")
every_each$quant = relevel(every_each$quant, ref="every")
print(summary(lm(logRTresid ~ quant, data=every_each)), digits=2)
...]
[mean RT for every] -0.055 0.019 [...]
[mean RT difference] 0.070 0.028 [...]
...]
```

The Bayesian and frequentist point estimates are very close, while the standard errors (our uncertainty about these estimates) are smaller, i.e., underestimated, on the frequentist side.

---

$^{10}$Technically speaking, the *pm.hpd* function returns the 95% highest posterior density (HPD) interval, not the central one. These intervals are very similar for posterior distribution approximations that are basically symmetric and unimodal, like the one under consideration here. For a discussion of the differences between these two types of posterior credible intervals (CRIs), see, for example, Kruschke (2011).
5.6 Conclusion

Bayesian methods for data analysis and cognitive modeling have two advantages, one theoretical and one computational. The theoretical one is that Bayesian methods are very intuitive: they mathematically encode the common-sense idea that we have beliefs about what is plausible and (un)likely to happen in the world, and that we learn from experience, that is, experience/data updates these prior beliefs. To do statistical inference or compute predictions boils down to computing posterior beliefs (i.e., prior beliefs updated with data) and examining these posterior beliefs in various ways.

Computationally, Bayesian methods in general and `pymc3` in particular give us access to a very powerful and flexible way of empirically evaluating linguistic theories. These theories can be faithfully and fairly directly encoded in specific structures for the priors and for the way we think the data is generated (the likelihood). We are not required to take our independently motivated linguistic theories and force them (or parts of them) into a pre-specified statistical inference mold. The opposite actually happens: we take our theory in all its complex and articulated glory and embed it in a fairly direct fashion in a Bayesian model. This, in turn, enables us to empirically evaluate it and do statistical inference about the parameters of the theory in a straightforward way.

Being able to take mathematically specified cognitive models and embed them in Bayesian models to empirically evaluate them will become essential in the next chapter, when we start introducing the subsymbolic components of ACT-R. These subsymbolic components come with a good number of real-valued parameters, and the Bayesian methods introduced in this chapter will enable us to learn the best settings for these parameters from the data, rather than relying on default values that seem to be pulled out of thin air.

Equally importantly, embedding rich cognitive theories in Bayesian models also enables us to do empirically-driven theory comparison. We can take two competing theories for the same phenomenon, collect experimental data, identify the best parameter settings for the two theories, as well as our uncertainty about these parameter settings, and then compare how well the predictions made by these parameter settings fit the data. We will in fact do this at the beginning of the next chapter, where we compare an exponential and a power-law model of forgetting.
Appendix

All the code discussed in this chapter is available on GitHub as part of the repository https://github.com/abrsvn/pyactr-book. If you want to examine it and run it, install pyactr (see Chap. 1), download the files and run them the same way as any other Python script.

File ch5_code.py:

☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch5_code.py.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.