In Chap. 4, we introduced a simple lexical decision task and a simple left-corner parser. The models we introduced in that chapter might be sufficient with respect to the way they simulate interactions with the environment, but they are too simplistic in their assumptions about memory, since memory retrievals are not dependent on any parameters of the retrieved word. In this chapter, we will improve on both models by incorporating the ACT-R model of declarative memory we just introduced in the previous chapter.

We start with a discussion of word frequency and the way it modulates lexical decision (Sect. 7.1). We then build several ACT-R models for lexical-decision tasks that incorporate the subsymbolic declarative memory components introduced in the previous chapter and take into account word frequency in a theoretically motivated way (Sects. 7.2–7.4). In Sect. 7.5, we do the same for a left-corner parser.

### 7.1 The Log-Frequency Model of Lexical Decision

One very robust parameter affecting latencies and accuracies in lexical decision tasks is frequency (Whaley 1978). In fact, frequency effects have been found not just in lexical decision tasks, but in many if not all tasks that involve some kind of lexical processing (Forster 1990b; Monsell 1991). These frequency effects have a specific functional form: since Howes and Solomon (1951), it is accepted that lexical access can be well approximated as a log-function of frequency.

Modeling lexical access in terms of log-frequency provides a good, but not perfect, fit to the data. Murray and Forster (2004) studied the role of frequency in detail and identified various issues with the log-frequency model. The data consisted of
Table 7.1 Frequency bands of words used in Murray and Forster (2004) (Exp. 1); frequency reported in number of tokens per 1 million words

| Group | Frequency range | Mean frequency | Latency (ms) | Accuracy (%) | Example word |
|-------|-----------------|----------------|--------------|--------------|--------------|
| 1     | 315–197         | 242.0          | 542          | 97.22        | Guy          |
| 2     | 100–85          | 92.8           | 555          | 95.56        | Somebody     |
| 3     | 60–55           | 57.7           | 566          | 95.56        | Extend       |
| 4     | 42–39           | 40.5           | 562          | 96.3         | Dance        |
| 5     | 32–30           | 30.6           | 570          | 96.11        | Shape        |
| 6     | 24–23           | 23.4           | 569          | 94.26        | Besides      |
| 7     | 19              | 19.0           | 577          | 95           | Fit          |
| 8     | 16              | 16.0           | 587          | 92.41        | Dedicate     |
| 9     | 14–13           | 13.4           | 592          | 91.67        | Robot        |
| 10    | 12–11           | 11.5           | 605          | 93.52        | Tile         |
| 11    | 10              | 10.0           | 603          | 91.85        | Between      |
| 12    | 9               | 9.0            | 575          | 93.52        | Precedent    |
| 13    | 7               | 7.0            | 620          | 91.48        | Wrestle      |
| 14    | 5               | 5.0            | 607          | 90.93        | Resonate     |
| 15    | 3               | 3.0            | 622          | 84.44        | Seated       |
| 16    | 1               | 1.0            | 674          | 74.63        | Habitually   |

collected responses and response times in a lexical decision task using words from 16 frequency bands, summarized in Table 7.1.1

Using the RT latencies from Murray and Forster (2004), let us build a log-frequency model and evaluate the discrepancies between the predictions of the model and the data. We first store the data in two variables \( \text{freq} \) (mean frequency) and \( \text{rt} \) (reaction time/latency; measured in s) (Fig. 7.1).

[py25] >>> 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.serif": ["Palatino"], "font.size": 8,
    ... "legend.fontsize": 8, "xtick.labelsize": 8, "ytick.labelsize": 8}

1Example words in Table 7.1 are based on the Corpus of Contemporary American English (COCA; http://corpus.byu.edu/coca/), specifically the list available at http://www.wordfrequency.info/files/entriesWithoutCollocates.txt, which lists frequencies of words of 450 million words total (as of March 7, 2017). The chosen example word was one of the closest one to the mean frequency listed in the same row. These were not the words used in the actual experiment—Murray and Forster (2004) controlled for other parameters, e.g., word length while manipulating word frequency.
We can now build a Bayesian model. We are thoroughly familiar with this kind of code, so we include it in [py26] below without any further comments:

```python
>>> log_freq_model = pm.Model()
>>> with log_freq_model:
...    # priors
```

[py26]
... intercept = pm.Normal('intercept', mu=0, sd=300) 4
... slope = pm.Normal('slope', mu=0, sd=300) 5
... sigma = pm.HalfNormal('sigma', sd=300) 6
... # likelihood 7
... mu = pm.Deterministic('mu', intercept + slope*np.log(freq)) 8
... observed_rt = pm.Normal('observed_rt', mu=mu, sd=sigma, 9
... observed=rt) 10
... 11
>>> with log_freq_model: 12
>>> # with log_freq_model:
>>> # db = Text('./data/log_freq_model_trace') 13
>>> # trace = pm.sample(draws=5000, trace=db, tune=15000, 14
>>> # n_init=200000, njobs=4) 15
>>> with log_freq_model: 16
... trace = load('./data/log_freq_model_trace') 17
... 18
We can now plot the estimates of the log-frequency model:

[py27] >>> mu = trace['mu']
>>> def generate_log_freq_figure():
... fig, (ax1, ax2) = plt.subplots(ncols=1, nrows=2)
... fig.set_size_inches(5.5, 5.5)
... # plot 1
... ax1.plot(freq, rt, marker='o', linestyle='')
... ax1.plot(freq, mu.mean(axis=0), color='red', linestyle='-')
... ax1.set_title('Observed (blue) \& predicted (red) RTs\ against log frequency')
... ax1.set_xlabel('Log frequency (log of \# tokens/1 million words)')
... ax1.set_xscale('log', basex=10)
... ax1.set_ylabel('RTs (s)')
... ax1.grid(b=True, which='minor', color='w', linewidth=1.0)
... # plot 2
... yerr=[mu.mean(axis=0)-pm.hpd(mu)[:,0],
... pm.hpd(mu)[:,1]-mu.mean(axis=0)]
... ax2.errorbar(rt, mu.mean(axis=0), yerr=yerr,
... marker='o', linestyle='')
... ax2.plot(np.linspace(0.5, 0.7, 10), np.linspace(0.5, 0.7, 10),
... color='red', linestyle=':')
... ax2.set_title('Log frequency model: Observed vs. predicted RTs')
... ax2.set_xlabel('Observed RTs (s)')
... ax2.grid(b=True, which='minor', color='w', linewidth=1.0)
... # clean up and save
... plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=1.9)
... plt.savefig('./figures/log_freq_model_figure.eps')
... plt.savefig('./figures/log_freq_model_figure.png')
... plt.savefig('./figures/log_freq_model_figure.pdf')
... >>> generate_log_freq_figure() 31

The plots show that the log-frequency model gets the middle values right, but it tends to underestimate the amount of time needed to access words in the extreme frequency bands—both low frequency (associated with high RTs) and high frequency (associated with low RTs). Murray and Forster (2004) take this as an argument for a specific information retrieval mechanism, the Rank Hypothesis (see Forster 1976, 1992), but as they note, other models of retrieval could similarly improve data fit. One such model treats frequency effects as practiced memory retrieval, which is commonly assumed to be a power function of time in the same way that memory performance is (Newell and Rosenbloom 1981; Anderson 1982; Logan 1990).
Practiced memory retrieval in ACT-R crucially relies on the power-function model of declarative memory. The power function is used to compute (base) activation based on the number of practice trials/‘rehearsals’ of a word (see (5) in Chap. 6), which in turn is used to compute latency and accuracy for retrieval processes (see (25) and (24) in Chap. 6).

For any word, the number of rehearsals that contribute to its base activation are crucially determined by its frequency. There are other factors that determine the number and timing of the rehearsals, but we will assume a simple model here: the number of rehearsals is exclusively determined by frequency. We will also assume, for simplicity, that presentations of a word are linearly spaced in time.

To be specific, let’s consider a 15-year old speaker. How can we estimate the time points at which a word was used in language interactions that the speaker participated in? Once we know these time points, we can compute the base activation for that word, which in turn will make predictions about retrieval latency and retrieval accuracy that we can check against the Murray and Forster (2004) data in Table 7.1.

We know the lifetime of the speaker (15 years), so if we know the total number of words an average 15-year old speaker has been exposed to, we can easily calculate how many times a particular word was used on average, based on its frequency. Once we find out how many times a word with a specific frequency was presented to our speaker during their lifetime, we can then present the word at linearly spaced intervals during the life span of the speaker (we use linearly spaced intervals for simplicity).

A good approximation of the number of words a speaker is exposed to per year can be found in Hart and Risley (1995). Based on recordings of 42 families, Hart and Risley estimate that children comprehend between 10 million and 35 million words a year, depending to a large extent on the social class of the family. This amount increases linearly with age.

According to the Hart and Risley (1995) study, a 15-year old has been exposed to anywhere between 50 and 175 million words total. For simplicity, let’s use the mean of 112.5 million words as the total amount of words a 15-year old speaker has been exposed to. This is a very conservative estimate because we ignore production, as well as the linguistic exposure associated with mass media.

The roughness of our estimate is not an issue for our purposes since we are interested in the relative effect of frequency, not its absolute effect. We do not want to predict how much time the retrieval of a word from one frequency band requires, but how much time a word requires compared to a word from another frequency band.

In [py28] below, we first compute the number of seconds in a year, and then the total number of seconds in the life span of the 15-year old speaker we’re modeling (lines 1–2). The function `time_freq` defined on lines 3–9 takes the mean frequency vector `freq` in [py25] above and generates a schedule of linearly spaced word rehearsals/presentations for words from the 16 frequency bands studied by Murray.
and Forster (2004). The schedule of rehearsals covers the entire life span of our 15-year old speaker.

```python
[py28] >>> SEC_IN_YEAR = 365*24*3600
>>> SEC_IN_TIME = 15*SEC_IN_YEAR
>>> def time_freq(freq):
...     max_idx = np.int(np.max(freq) * 112.5)
...     rehearsals = np.zeros((max_idx, len(freq)))
...     for i in np.arange(len(freq)):
...         temp = np.arange(np.int((freq[i]*112.5)))
...         temp = temp * np.int(SEC_IN_TIME/(freq[i]*112.5))
...         rehearsals[:len(temp),i] = temp
...     return(rehearsals.T)
```

On line 4 in [py28], we initialize our rehearsal schedule in the matrix `rehearsals`. This matrix has as many rows as the number of rehearsals for the most frequent word band: `np.max(freq)` gives us the maximum frequency in words per million, which we multiply by 112.5 million words (the total number of words our 15-year old speaker has been exposed to). The `rehearsals` matrix has 16 columns: as many columns as the frequency bands we are interested in.

The `for` loop on lines 6–9 in [py28] iterates over the 16 frequency bands and, for each frequency band, it does the following. On line 7, we identify the total number of rehearsals for frequency band `i` throughout the life span of the speaker (`freq[i]*112.5`) and generate a vector with as many positions as there are rehearsals. On line 8, at each position in that vector, we store the time of a rehearsal in seconds. The result is a vector `temp` of linearly spaced rehearsal times that we store in our full `rehearsals` matrix (line 9).

These rehearsal times can also be viewed as time periods since rehearsals if we reverse the vector (recall that we need time periods since rehearsals when we compute base activation). But we don’t need to actually reverse the vector since we will have to sum the time periods to compute activation, and summation is commutative.

Finally, we return the full `rehearsals` matrix on line 10, in transposed form because of the way we will use it to compute base activation (see below).

With this function in hand, we compute a rehearsal schedule for all 16 frequency bands on line 3 of [py29] below. We store the matrix in a `theano` variable called `time`. The `theano` library, which we import on lines 1–2, enables us to do computations with multi-dimensional arrays efficiently, and provides the computational backbone for the Bayesian modeling library `pymc3`. We need to access it directly to be able to compute activations from the rehearsal schedule stored in the `time` variable.

```python
[py29] >>> import theano
>>> import theano.tensor as tt
>>> time = theano.shared(time_freq(freq), ‘time’)
```

Now that we have the rehearsal schedule, we can implement the lexical decision model:

```python
[py30] >>> lex_dec_model = pm.Model()
>>> with lex_dec_model:
...     # prior for base activation
...     decay = pm.Uniform(‘decay’, lower=0, upper=1)
```
7.2 The Simplest ACT-R Model of Lexical Decision

... # priors for latency
... intercept = pm.Uniform('intercept', lower=0, upper=2)
... latency_factor = pm.Uniform('latency_factor', lower=0, upper=5)
... # priors for accuracy
... noise = pm.Uniform('noise', lower=0, upper=5)
... threshold = pm.Normal('threshold', mu=0, sd=10)
... # compute activation
... scaled_time = time ** (-decay)
... def compute_activation(scaled_time_vector):
... compare = tt.isinf(scaled_time_vector)
... subvector = scaled_time_vector[(1-compare).nonzero()]
... activation_from_time = tt.log(subvector.sum())
... return activation_from_time
... activation_from_time, _ = theano.scan(fn=compute_activation, sequences=scaled_time)
... # latency likelihood
... mu_rt = pm.Deterministic('mu_rt', intercept +
... latency_factor*tt.exp(-activation_from_time))
... rt_observed = pm.Normal('rt_observed', mu=mu_rt, sd=0.01, observed=rt)
... # accuracy likelihood
... odds_reciprocal = tt.exp(-(activation_from_time - threshold)/noise)
... mu_prob = pm.Deterministic('mu_prob', 1/(1 + odds_reciprocal))
... prob_observed = pm.Normal('prob_observed', mu=mu_prob, sd=0.01, observed=accuracy)

>>> with lex_dec_model:
... trace = load('./data/lex_dec_model_trace')

Computing activations from the rehearsal schedule requires us to define a separate function `compute_activation`—see lines 13–17 in [py30]. This function assumes that the matrix `scaled_time` has been computed (line 12): to compute this matrix, we take our rehearsal schedule stored in the `time` matrix (time periods since rehearsals for all frequency bands) and raise these time periods to the \(-\text{decay}\) power. The result is a matrix that stores scaled times, i.e., the base activation boost contributed by each individual word rehearsal for all frequency bands.

Some of the values in the `time` matrix were 0. In the `scaled_time` matrix, they become infinity. When we compute final activations, we want to discard all these infinity values, which is what the `compute_activation` function does. It takes the 16 vectors of scaled times (for the 16 frequency bands) as inputs one at a time. Then, it identifies all the infinity values (line 14). Then, it extracts the subvector of the input vector that contains only non-infinity values (line 15). With this subvector in hand, we can sum the scaled times and take the log of the result to obtain our final activation value (line 16), which the function returns to us.

With the function `compute_activation` in hand, we need to iterate over the 16 vectors of scaled times for our 16 frequency bands and compute the 16 resulting activations by applying the `compute_activation` function to each of these vectors of scaled times. However, theano-based programming is purely functional, which means there are no for loops. We therefore use the `theano.scan` method.
on lines 18–19 of [py30] to iteratively apply the `compute_activation` function to each of the 16 vectors in the `scaled_time` matrix.²

The likelihoods of the lexical decision model in [py30] (lines 21–24 and 26–29 in [py30]) are direct implementations of the retrieval latency and retrieval probability equations in (25) and (24). We omit the latency exponent in the latency likelihood (see `mu_rt` on lines 21–22) because we assume it is set to its default value of 1. We will see that this value is not appropriate, so we will have to move to a model in which the latency exponent is also fully modeled.

Note that the dispersions around the mean RTs and mean probabilities are very minimal—we set the standard deviations on lines 23 and 28 to 0.01. The reason is that our observed values for both RTs and accuracy are not raw values—they are already means, namely, the empirical means for the 16 frequency bands reported in Murray and Forster (2004). As such, we assume these means are very precise reflections of the underlying parameter values.

We could model these standard deviations explicitly, but we decided not to since we have only 32 observations here (16 for RTs, 16 for accuracies), and we are trying to estimate a fairly large number of parameters already: `decay`, `intercept`, `latency_factor`, `noise` and `threshold`. Low information priors for these parameters are specified on lines 4, 6–7 and 9–10 in [py30].³

The only new parameter in this model relative to the ACT-R probability and latency equations in (24) and (25) is the `intercept` parameter we use in the latency likelihood (line 21 in [py30]). The intercept is supposed to absorb the time in the lexical decision task associated with operations other than memory retrieval: focusing visual attention, motor planning etc.

With the model fully specified, we sample from the posterior distributions of the parameters. Once we obtain the samples, we are ready to plot them to evaluate how well the model fits the data (we take the first 2000 samples to be the burn-in and drop them; see, for example, Kruschke (2011) for more discussion of burn-in, thinning etc.). The code for the plots is provided in [py31] and the resulting plots are shown in Fig. 7.2.

```
>>> trace = trace[2000:]
>>> mu_rt = pd.DataFrame(trace['mu_rt'])
>>> yerr_rt = [(mu_rt.mean()-mu_rt.quantile(0.025))*1000,
...           (mu_rt.quantile(0.975)-mu_rt.mean())*1000]
>>> mu_prob = pd.DataFrame(trace['mu_prob'])
>>> yerr_prob = [(mu_prob.mean()-mu_prob.quantile(0.025)),
...           (mu_prob.quantile(0.975)-mu_prob.mean())]
>>> def generate_lex_dec_model_figure():
```

²For more information about functional programming, see, for example, https://en.wikipedia.org/wiki/Functional_programming and Abelson et al. (1996). Purely functional programming languages (Haskell is probably the most well-known example nowadays) should be easy to understand for formal semanticists, given their familiarity with λ-calculus.

³By low information priors, we mean priors that do not assign high probability to particular narrow regions in the parameter space. If some regions are less probable than others, that is because their lower probability can be determined from considerations that are independent of experimental data we are trying to model.
Fig. 7.2 Lexical decision model: estimated and observed RTs and probabilities

```python
fig, (ax1, ax2) = plt.subplots(ncols=1, nrows=2)
fig.set_size_inches(5.5, 5.5)

# plot 1: RTs
ax1.errorbar(rt*1000, mu_rt.mean()*1000, yerr=yerr_rt, marker='o', linestyle='')
ax1.plot(np.linspace(500, 800, 10), np.linspace(500, 800, 10), color='red', linestyle=':)
ax1.set_title('Lexical decision model:
Observed vs. predicted RTs')
ax1.set_xlabel('Observed RTs (ms)')
ax1.set_ylabel('Predicted RTs (ms)')
ax1.grid(b=True, which='minor', color='w', linewidth=1.0)

# plot 2: probabilities
ax2.errorbar(accuracy, mu_prob.mean(), yerr=yerr_prob, marker='o', linestyle='')
ax2.plot(np.linspace(50, 100, 10)/100, np.linspace(50, 100, 10)/100, color='red', linestyle=':)
ax2.set_title('Lexical decision model:
Observed vs. predicted probabilities')
ax2.set_xlabel('Observed probabilities')
ax2.set_ylabel('Predicted probabilities')
ax2.grid(b=True, which='minor', color='w', linewidth=1.0)

# clean up and save
plt.tight_layout(pad=0.5, w_pad=0.2, h_pad=0.7)
plt.savefig('./figures/lex_dec_model_figure.eps')
```
An important thing to note about the ACT-R lexical decision model is that predictions about latencies and probabilities are theoretically connected: base activation is an essential ingredient in predicting both of them. Thus, we are not proceeding purely in an inductive fashion here by looking at the RT data on one hand, the accuracy data on the other hand, and then drawing theoretical conclusions from the data in an informal way, i.e., in a way that is suggestive and possibly productive, but ultimately vague and incapable of making precise predictions.

Instead, our mathematical model takes a core theoretical concept (base activation as a function of word frequency) and connects it in a mathematically explicit way to latency and accuracy. Furthermore, our computational model directly implements the mathematical model, and enables us to fit it to the experimentally obtained latency and accuracy data.

In addition to connecting distinct kinds of observable behavior via the same unobservable theoretical construct(s), a hallmark of a good scientific theory is that it is falsifiable. And the plots in Fig. 7.2 show that an ACT-R model of lexical decision that sets the latency exponent to its default value of 1 (in effect omitting it) is empirically inadequate.

The bottom plot in Fig. 7.2 shows that our lexical decision model does a good job of modeling retrieval probabilities. The predicted probabilities are very close to the observed ones, and they are precisely estimated (there are very few visible error bars protruding out of the plotted points).

In contrast, latencies are poorly modeled, as the top plot in Fig. 7.2 shows. The predicted RTs are not very close to the observed RTs, and our model is very confident in its incorrect predictions (error bars are barely visible for most predicted RTs).

7.3 The Second ACT-R Model of Lexical Decision: Adding the Latency Exponent

Our ACT-R lexical decision model without a latency exponent does not provide a satisfactory fit to the Murray and Forster (2004) latency data. In fact, the log-frequency model is both simpler (although less theoretically motivated) and empirically more adequate.

We therefore move to a model that is minimally enriched by explicitly modeling the latency exponent. The usefulness of the latency exponent in modeling reaction time data has been independently noted in the recent literature—see, for example, West et al. (2010).

The code for the model is provided in [py32] below. The only additions are (i) the half-normal prior for the latency exponent on line 10 and (ii) its corresponding
addition to the latency likelihood on line 25. Note that we use a different method to sample the posterior for this model (Sequential Monte Carlo/SMC, line 37).4

```python
>>> from pymc3.backends.text import dump

>>> lex_dec_model_lat_exp = pm.Model()
>>> with lex_dec_model_lat_exp:
...     # prior for base activation
...     decay = pm.Uniform('decay', lower=0, upper=1)
...     # priors for latency
...     intercept = pm.Uniform('intercept', lower=0, upper=2)
...     latency_factor = pm.Uniform('latency_factor', lower=0, upper=5)
...     latency_exponent = pm.HalfNormal('latency_exponent', sd=3)
...     # priors for accuracy
...     noise = pm.Uniform('noise', lower=0, upper=5)
...     threshold = pm.Normal('threshold', mu=0, sd=10)
...     # compute activation
...     scaled_time = time ** (-decay)
...     def compute_activation(scaled_time_vector):
...         compare = tt.isinf(scaled_time_vector)
...         subvector = scaled_time_vector[(1-compare).nonzero()]
...         activation_from_time = tt.log(subvector.sum())
...         return activation_from_time
...     # latency likelihood
...     mu_rt = pm.Deterministic('mu_rt', intercept +
...         latency_factor*tt.exp(-latency_exponent*
...         activation_from_time))
...     rt_observed = pm.Normal('rt_observed', mu=mu_rt, sd=0.01,  
...         observed=rt)
...     # accuracy likelihood
...     odds_reciprocal = tt.exp(-(activation_from_time - threshold)/noise)
...     mu_prob = pm.Deterministic('mu_prob', 1/(1 + odds_reciprocal))
...     prob_observed = pm.Normal('prob_observed', mu=mu_prob, sd=0.01,  
...         observed=accuracy)

>>> # run 4 times to obtain 4 chains
>>> #with lex_dec_model_lat_exp:
>>> #step = pm.SMC()  
>>> #trace = pm.sample(draws=20000, step=step, njobs=1)
>>> #dump = Text('./data/lex_dec_model_lat_exp_trace', trace)

>>> with lex_dec_model_lat_exp:
...    trace = load('./data/lex_dec_model_lat_exp_trace')
```

We plot the results of this enriched lexical decision model: the code for the plots is provided in [py33] and the resulting plots are shown in Fig. 7.3.

```
>>> mu_rt = pd.DataFrame(trace['mu_rt'])
>>> yerr_rt = [(mu_rt.mean()-mu_rt.quantile(0.025))*1000,
...       (mu_rt.quantile(0.975)-mu_rt.mean())*1000]

>>> mu_prob = pd.DataFrame(trace['mu_prob'])
>>> yerr_prob = [(mu_prob.mean()-mu_prob.quantile(0.025)),
...       (mu_prob.quantile(0.975)-mu_prob.mean())]

>>> def generate_lex_dec_model_lat_exp_figure():
```

4Discussing different kinds of MCMC sampling schemes, chain convergence diagnostics etc. is beyond the scope of this book. For a fairly recent survey of SMC methods, see, for example, Creal (2012). For a brief comparison of BUGS versus NUTS, see, for example, this blog post: http://elevanth.org/blog/2017/11/28/build-a-better-markov-chain/. For more discussion of MCMC diagnostics etc., see Gelman and Hill (2007), Kruschke (2011) and references therein, among others.
Fig. 7.3 Lex. dec. model with latency exp.: estimated and observed RTs and probabilities
We see that the lexical decision model that explicitly models the latency exponent fits both latencies and probabilities very well. The latencies, in particular, are modeled better than both the lexical decision model without a latency exponent, and the log-frequency model, which did not have a very good fit to the data at the extreme frequency bands (low or high frequencies).

We list below the estimated posterior mean and 95% credible interval (CRI) for the latency exponent: the posterior mean value and the CRI are pretty far away from the default value of 1 we assumed in the previous model.

```python
>>> latency_exponent_posterior = trace['latency_exponent']
>>> latency_exponent_posterior.mean()
0.26874855202916675
>>> pm.hpd(latency_exponent_posterior)
array([0.05284098, 0.46545318])
```

The posterior estimates for the other parameters (means and 95% CRIs) are provided below for reference:

```python
>>> decay_posterior = trace['decay']
>>> decay_posterior.mean()
0.20922545157863634
>>> pm.hpd(decay_posterior)
array([6.48482310e-04, 7.12426351e-01])
>>> intercept_posterior = trace['intercept']
>>> intercept_posterior.mean()
0.48309734409006416
>>> pm.hpd(intercept_posterior)
array([0.37181586, 0.55819152])
>>> latency_factor_posterior = trace['latency_factor']
>>> latency_factor_posterior.mean()
0.35641063490136776
>>> pm.hpd(latency_factor_posterior)
array([0.00476266, 0.77476657])
>>> threshold_posterior = trace['threshold']
>>> threshold_posterior.mean()
-1.2335980386761345
>>> pm.hpd(threshold_posterior)
array([-10.80783392, 2.99617913])
>>> noise_posterior = trace['noise']
>>> noise_posterior.mean()
1.780720441227118
>>> pm.hpd(noise_posterior)
array([1.5927438 , 1.96868986])
```
7.4 Bayes+ACT-R: Quantitative Comparison for Qualitative Theories

In this subsection, we show how we can embed ACT-R models implemented in \texttt{pyactr} into Bayesian models implemented in \texttt{pymc3}. This embedding opens the way towards doing quantitative comparison based on experimental data for both subsymbolic and symbolic theories.

That is, this Bayes+ACT-R combination enables us to do \textit{quantitative} theory comparison. We will be able to take our symbolic theories that make claims about competence, for example, Discourse Representation Theory (DRT; Kamp 1981; Kamp and Reyle 1993), as we will see in Chaps. 8 and 9, embed them in larger performance theories that have a detailed processing component, and then compare different theories quantitatively based on behavioral data of the kind commonly collected in psycholinguistics.

In this section, we introduce the basics of our Bayes+ACT-R framework by considering and comparing three models for lexical decision tasks:

i. the first model builds on the simple ACT-R/\texttt{pyactr} lexical decision model introduced in Chap. 4; we will show how that model can be used as part of the likelihood function of a larger Bayesian model;

ii. the second model is cognitively more realistic than the first one: it makes use of the imaginal buffer as an intermediary between the visual module and the declarative memory module; we set the delay for imaginal-buffer encoding to its default value of 200 ms; once again, this ACT-R/\texttt{pyactr} model will provide part of the likelihood component of a larger Bayesian model;

iii. the third and final model is cognitively more realistic since it makes use of the imaginal buffer, just as the second model, but we set the encoding delay of the imaginal buffer to the non-default value of 0 ms\(^5\); this is the imaginal-buffer delay we needed when we implemented our left-corner parser in Chap. 4; just as before, the ACT-R/\texttt{pyactr} model is embedded in a larger Bayesian model, for which it provides part of the likelihood function.

The first model (i) without the imaginal buffer and the other two models (ii) and (iii) with imaginal buffers differ with respect to a symbolic (qualitative) component. In this particular case, the symbolic component (imaginal-buffer usage or lack thereof) belongs to the processing part of the symbolic (non-quantitative) theory, but theoretical differences at the ‘core’ competence level can (and will) be similarly compared.

In contrast, the last two models (ii) and (iii) differ with respect to specific conjectures about a subsymbolic (quantitative) component, namely the average time to encode chunks in the imaginal buffer.

\(^5\)In computational models of psychological experiments, the encoding delay in the imaginal buffer is usually set to 200 ms. See Chap. 3, and also Taatgen et al. (2009).
Our Bayes+ACT-R framework enables us to compare all these models. This comparison is not only qualitative. The models can be quantitatively evaluated and compared relative to specific experimental data. Since pyactr enables us to embed ACT-R models as the likelihood component of larger Bayesian models built with pymc3, we can do statistical inference over the subsymbolic parameters of our ACT-R model in the standard Bayesian way, rather than trying different values one at a time and manually identifying the best fitting ones.

We’ll therefore be able to identify the standard measures of central tendency (posterior means, but also medians or modes if needed), as well as compute credible intervals for every parameter of interest. The Bayesian framework will furthermore enable us to conduct unrestricted model comparison (using Bayes factors or WAIC, for example), unlike maximum likelihood methods—see the discussion at the end of this section and in Sect. 7.5.

Throughout this book, whenever we embed an ACT-R model in a Bayesian model, we turn off all the non-deterministic (stochastic) components of the ACT-R model other than the ones for which we are estimating parameters. This effectively turns the ACT-R model into a complex, but deterministic, function of the parameters, which we can straightforwardly incorporate as a component of the likelihood function of the Bayesian model.

For more realistic simulations, we would have to turn on various non-deterministic components of the ACT-R model (e.g., noise associated with visual module), in which case we would have to resort to Approximate Bayesian Computation (ABC; see Sisson et al. (2019) and references therein) to incorporate an approximation of the ACT-R induced likelihood into our Bayesian model. ABC is beyond the scope of this book, but it is a very promising direction for future research, and a central issue to be addressed as more linguistically sophisticated cognitive models are developed.

7.4.1 The Bayes+ACT-R Lexical Decision Model Without the Imaginal Buffer

The link to the full code for this model is provided in the appendix to this chapter—see Sect. 7.7.1. We will only discuss here the most important and novel aspects of this Bayes+ACT-R model combination. We first initialize the model under the variable lex_decision and declare its goal buffer to be g.

```python
>>> import pyactr as actr
>>> environment = actr.Environment(focus_position=(320, 180))
>>> lex_decision = actr.ACTRModel(environment=environment,
... subsymbolic=True,
... automatic_visual_search=True,
... activation_trace=False,
... retrieval_threshold=-80,
... motor_prepared=True,
... eye_mvt_scaling_parameter=0.18,
... emma_noise=False)
>>> lex_decision.goals = {}
>>> lex_decision.set_goal("g")
set()
```
We set up the data: see the `FREQ`, `RT` and `ACCURACY` variables in [py37] below (recall that `pyactr` measures time in s, not ms, so we divide the RTs by 1000). We then generate the presentations times for the 16 word-frequency bands considered in Murray and Forster (2004): see `FREQ_DICT` and the `theano` variable time.

[py37] >>> FREQ = np.array([242, 92.8, 57.7, 40.5, 30.6, 23.4, 19, 16, 13.4, 11.5, 10, 9, 7.5, 3, 1]) 1 ... >>> RT = np.array([542, 555, 566, 562, 570, 569, 577, 587, 3592, 605, 603, 575, 620, 607, 622, 674])/1000 2 ... >>> ACCURACY = np.array([97.22, 95.56, 95.56, 96.3, 96.11, 94.26, 95, 92.41, 91.67, 93.52, 91.85, 93.52, 91.48, 90.93, 84.44, 74.63])/100 3 ... >>> # on average, 15 years of exposure is 112.5 million words 4 ... >>> FREQ_DICT = {} 5 ... >>> FREQ_DICT['guy'] = 242*112.5 6 ... >>> FREQ_DICT['somebody'] = 92*112.5 7 ... >>> FREQ_DICT['extend'] = 58*112.5 8 ... >>> FREQ_DICT['dance'] = 40.5*112.5 9 ... >>> FREQ_DICT['shape'] = 30.6*112.5 10 ... >>> FREQ_DICT['besides'] = 23.4*112.5 11 ... >>> FREQ_DICT['fit'] = 19*112.5 12 ... >>> FREQ_DICT['dedicate'] = 16*112.5 13 ... >>> FREQ_DICT['robot'] = 13.4*112.5 14 ... >>> FREQ_DICT['tile'] = 11.5*112.5 15 ... >>> FREQ_DICT['between'] = 10*112.5 16 ... >>> FREQ_DICT['precedent'] = 9*112.5 17 ... >>> FREQ_DICT['wrestle'] = 7*112.5 18 ... >>> FREQ_DICT['resonate'] = 6*112.5 19 ... >>> FREQ_DICT['seated'] = 3*112.5 20 ... >>> FREQ_DICT['habitually'] = 1*112.5 21 ... >>> ORDERED_FREQ = sorted(list(FREQ_DICT), key=lambda x:FREQ_DICT[x], reverse=True) 22 ... >>> def time_freq(freq): 23 ... rehearsals = np.zeros((np.int(np.max(freq) * 113), len(freq))) 24 ... for i in np.arange(len(freq)): 25 ... temp = np.arange(np.int((freq[i]*112.5))) 26 ... temp = temp * np.int(SEC_IN_TIME/(freq[i]*112.5)) 27 ... rehearsals[:len(temp),i] = temp 28 ... return(rehearsals.T) 29 ... >>> time = theano.shared(time_freq(FREQ), 'time') 30 ... >>> LEMMA_CHUNKS = [(actr.makechunk('', typename='word', form=word)) for word in ORDERED_FREQ] 31 ...

We are now ready to build the procedural core of our model. The production rules are the same as the ones we introduced and discussed in Chap. 4, listed for ease of reference in [py38] below:

- the "attend word" rule takes a visual location encoded in the visual *where* buffer and issues a command to the visual *what* buffer to move attention to that visual location;
- the "retrieving" rule takes the visual value/content discovered at that visual location, which is a potential word form, and places a declarative memory request to retrieve a word with that form;
- finally, the "lexeme retrieved" and "no lexeme found" rules take care of the two possible outcomes of the memory retrieval request: if a word with that form is retrieved from memory ("lexeme retrieved"), a command is issued to the motor module to press the ‘J’ key; if no word is retrieved ("no
lexeme found"), a command is issued to the motor module to press the 'F' key.

```python
[py38] >>> lex_decision.productionstring(name="attend word", string=""
...
...
... =g> 
...
...
... isa _visuallocation
...
...
... state attend
...
...
... =visual_location>
...
...
... isa _visuallocation
...
...
... state free
...
...
... =g> 
...
...
... isa goal
...
...
... state retrieving
...
...
... isa _visual
...
...
... cmd move_attention
...
...
... screen_pos =visual_location
...
...
... "visual_location>"
...
...
"(**="
... =g': goal(state= attend), '=?visual_location': _visuallocation(color= ,
... screen_x= , screen_y= , value= ), '=?visual': ('state': 'free'))

==>
... =g': goal(state= retrieving), '=?visual': _visual(cmd= move_attention,
... color= , screen_pos= =visual_location, value= ), '=?visual_location': None)

,...
...
...
... =g> 
...
...
... isa goal
...
...
... state retrieving
...
...
... =visual>
...
...
... isa _visual
...
...
... value =val
...
...
... =g> 
...
...
... isa goal
...
...
... state retrieval_done
...
...
... +retrieval>
...
...
... isa word
...
...
... form =val
...
...
... **")

("=?g': goal(state= retrieving), 
... '=?visual': _visual(cmd= , color= , screen_pos= , value= =val))

==>
... =g': goal(state= retrieval_done), '=?retrieval': word(form= =val))

,...
...
...
... =g> 
...
...
... isa goal
...
...
... state retrieval_done
...
...
... ?retrieval>
...
...
... buffer full
...
...
... state free
...
...
... =g> 
...
...
... isa goal
...
...
... state done
...
...
... +manual>
...
...
... isa _manual
...
...
... cmd press_key
...
...
... key 'J'
...
...
... **")

("=?g': goal(state= retrieval_done),
... '=?retrieval': ('?buffer': 'full', 'state': 'free'))

==>
... =g': goal(state= done), '=?manual': _manual(cmd= press_key, key= J))

,...
...
...
... =g> 
...
...
... isa goal
...
...
... state done
...
...
... +manual>
...
...
... isa _manual
...
...
... cmd press_key
...
...
... key 'J'
...
...
... **")

("=?g': goal(state= done), 
... '=?manual': _manual(cmd= press_key, key= J))

,...
...
...
... =g> 
...
...
... isa goal
...
...
... state done
...
...
... +manual>
...
...
... isa _manual
...
...
... cmd press_key
...
...
... key 'J'
...
...
... **")

("=?g': goal(state= done),
... '=?manual': _manual(cmd= press_key, key= J))

,...
...
...
... =g> 
...
...
... isa goal
...
...
... state done
...
...
... +manual>
...
...
... isa _manual
...
...
... cmd press_key
...
...
... key 'J'
...
...
... **")

("=?g': goal(state= done),
... '=?manual': _manual(cmd= press_key, key= J))
```
With the production rules in place, we can start preparing the way towards embedding the ACT-R model into a Bayesian model. The main idea is that we will use the ACT-R model as the likelihood component of the Bayesian model for lexical-decision latencies/RTs. Specifically, we will feed parameter values for the latency factor $lf$, latency exponent $le$ and decay into the ACT-R model, run the model with these parameters for words from all 16 frequency bands, and collect the resulting RTs. The Bayesian model will then use these RTs to sample new values for the $lf$, $le$ and decay parameters in proportion to how well the RTs generated by the ACT-R model agree with the experimentally collected RTs (and the diffuse priors over these parameters).

The first function we need is `run_stimulus(word)` in [py39] below. This function takes a word from one of the 16 frequency bands as its argument and runs one instance of the ACT-R lexical decision model for that word. To do that, we first reset the model to its initial state: we flush buffers without moving their contents to declarative memory (lines 2–9 in [py39]), we set the word argument as the new stimulus (line 10), we initialize the goal buffer $g$ with the "start" chunk (lines 11–13), and we initialize the lexical decision simulation (lines 14–19).

At this point, we run a `while` loop that steps through the simulation until a lexical decision is made by pressing the ‘J’ or ‘F’ key, at which point we record the time of the decision in the variable `estimated_time` (set to $-1$ if the word was not retrieved),6 exit the `while` loop and return the estimated RT (lines 20–28).

The second function `run_lex_decision_task()` in [py39] runs a full lexical decision task by calling the `run_stimulus(word)` function for words from all 16 frequency bands (lines 31–33). The function returns the vector of estimated lexical decision RTs for all these words (line 34).

---

6In this Bayes+ACT-R estimation, we only model latencies for successful retrievals. In other words, we want the model to always retrieve successfully, and if it does not, we penalize the result by choosing a latency that is markedly off. To ensure that the model performs a successful retrieval, the retrieval threshold in the ACT-R model is set to a very low value ($\tau = -80$, see line 7 in [py36]).
With the \texttt{run\_lex\_decision\_task()} function in hand, we only need to be able to interface the ACT-R model implemented in \textit{pyactr} with a Bayesian model implemented in \textit{pymc3} (and \textit{theano}). This is what the function \texttt{actrmodel\_latency} in [py40] below does. This function runs the entire lexical decision task for specific values of the latency factor $lf$, latency exponent $le$ and decay parameters provided by the Bayesian model (which will be discussed below). The activation computed by \textit{theano} with the same value of the decay argument is also passed as a separate argument \texttt{activation\_from\_time} to save (a significant amount of) computation time.

The \texttt{actrmodel\_latency} function takes these four parameter values as arguments (line 3 in [py40]), initializes the lexical decision model with them (lines 4–9), runs the lexical decision task with these model parameters (line 10) and returns the resulting vector of RTs (line 11). The entire function is wrapped inside the \textit{theano}-provided decorator \texttt{@as\_op} (lines 1–2),\footnote{Python3 decorators provide a specific way to define and call higher-order functions, that is, functions that take other functions as arguments and/or return functions as values. Such functions are, of course, very familiar to formal semanticists, who follow Montague in extensively using such functions (usually represented in a simply-typed $\lambda$-calculus) to model the semantics of natural languages.} which enables \textit{theano} and \textit{pymc3} to use the \texttt{actrmodel\_latency} function as if it was a native \textit{theano/pymc3} function. The only thing the \texttt{@as\_op} decorator needs is data-type declarations for the arguments of the \texttt{actrmodel\_latency} function ($lf$, $le$ and decay are scalars,}
while `activation_from_time` is a vector—line 1) and for its value (which is a vector—line 2).

```python
>>> @theano.as_op(itypes=[tt.dscalar, tt.dscalar, tt.dscalar, tt.dvector], 1
    otypes=[tt.dvector]) 2
    def actrmodel_latency(lf, le, decay, activation_from_time): 3
        lex_decision.model_parameters['latency_factor'] = lf 4
        lex_decision.model_parameters['latency_exponent'] = le 5
        lex_decision.model_parameters['decay'] = decay 6
        activation_dict = {x[0]: x[1] for x in \ 7
            zip(LEMMA_CHUNKS, activation_from_time)} 8
        lex_decision.decmem.activations.update(activation_dict) 9
        sample = run_lex_decision_task() 10
        return np.array(sample) 11

We are now ready to use the `actrmodel_latency` function as the likelihood function for latencies in a Bayesian model very similar to the ones we already discussed in this chapter. The model is specified in [py41] below. The prior for the `decay` parameter is uniform (line 3), the priors for the lexical-decision accuracy parameters `noise` and `threshold` are uniform and normal (lines 4–5), and the priors for the lexical-decision latency parameters `lf` and `le` are both half-normal (lines 6–7).

We then compute activation based on word frequency in the same way we did before (lines 8–15), after which we specify the likelihood function for lexical-decision latency (lines 16–19), which crucially uses the ACT-R model via the `actrmodel_latency` function (line 16), and the likelihood function for lexical-decision accuracy (lines 20–23). Note that the accuracy is computed independently of the latency, which simplifies the workings of the `pyactr` model (as we already indicated, we can assume that the `pyactr` model recalls all words successfully).

```python
>>> lex_decision_with_bayes = pm.Model() 1
    >>> with lex_decision_with_bayes: 2
        decay = pm.Uniform('decay', lower=0, upper=1) 3
        threshold = pm.Normal('threshold', mu=0, sd=10) 4
        noise = pm.Uniform('noise', lower=0, upper=5) 5
        lf = pm.HalfNormal('lf', sd=1) 6
        le = pm.HalfNormal('le', sd=1) 7
        scaled_time = time ** (-decay) 8
        def compute_activation(scaled_time_vector): 9
            compare = tt.isinf(scaled_time_vector) 10
            subvector = scaled_time_vector[(1-compare).nonzero()] 11
            activation_from_time = tt.log(subvector.sum()) 12
            return activation_from_time 13
        activation_from_time, _ = theano.scan(fn=compute_activation, 14
            sequences=scaled_time) 15
        pyactr_rt = actrmodel_latency(lf, le, decay, activation_from_time) 16
        mu_rt = pm.Deterministic('mu_rt', pyactr_rt) 17
        rt_observed = pm.Normal('rt_observed', mu=mu_rt, sd=0.01, 18
            observed=RT) 19
        odds_reciprocal = tt.exp(-(activation_from_time - threshold)/noise) 20
        mu_prob = pm.Deterministic('mu_prob', 1/(1 + odds_reciprocal)) 21
        prob_observed = pm.Normal('prob_observed', mu=mu_prob, sd=0.01, 22
            observed=ACCURACY) 23

The Bayesian model is schematically represented in Fig. 7.4 (following the type of figures introduced in Kruschke 2011).

The plots in Fig. 7.5 show that the Bayes+ACT-R model without any imaginal-buffer involvement has a very good fit to both the latency and the accuracy data. The
Fig. 7.4 The structure of the Bayesian model in [py41]

Lex. dec. model (pyactr, no imaginal): RTs

Lex. dec. model (pyactr, no imaginal): Prob.s

Fig. 7.5 Lex. dec. model, Bayes+ACT-R, no imaginal buffer: estimated and observed RTs and probabilities
top panel plots observed RTs against predicted RTs, while the bottom panel plots observed probabilities against predicted probabilities. The closer the points are to the red diagonal line, the better the model predictions—and we see that the model can fit the observed latency and accuracy data very well.

For reference, we provide the Gelman-Rubin diagnostic (a.k.a. $\hat{R}$) for this model in [py42] below. As Gelman and Hill (2007, 352) put it, “$\hat{R}$ gives information about convergence of the algorithm. At convergence, the numbers [...] should equal 1 [...]. If $\hat{R}$ is less than 1.1 for all parameters, then we judge the algorithm to have approximately converged, in the sense that parallel chains have mixed well.” We see that the $\hat{R}$ values for the model are below 1.1, which is reassuring.8

[py42] >>> with lex_decision_with_bayes:
...     trace = load('./data/lex_dec_pyactr_no_imaginal')
...     ...
... >>> pm.diagnostics.gelman_rubin(trace)
{'threshold': 1.0054899750513782,
 'decay': 1.0062475153014665,
 'noise': 1.009517464648299,
 'lf': 1.0083217341786315,
 'le': 1.0106803162613,
 'mu_rt': array([1.01458128, 1.01275183, 1.01129197, 1.00973846,
  1.00822621, 1.00694688, 1.00527242, 1.00366908, 1.00211617,
  1.00067242, 1.00027242, 0.99997675, 1.00019667, 1.00228782,
  1.00666908, 1.01181443]),
 'mu_prob': array([1.0111905 , 1.01148533, 1.01152191, 1.01144346,
  1.01127359, 1.01096758, 1.01058891, 1.01014176, 1.00953423,
  1.00884847, 1.00804849, 1.00736052, 1.00533042, 1.00216177, 0.9999311, 1.00019667])}

However, this model oversimplifies the process of encoding visually retrieved data. We assume that the visual value found at a particular visual location is immediately shuttled to the retrieval buffer to place a declarative memory request – see the productions "attend word" and "retrieving" in [py38] above.

This disregards the cognitively-motivated ACT-R assumption that transfers between the visual what buffer and the retrieval buffer are mediated by the goal or imaginal buffers. Cognition in ACT-R is goal-driven, so any important step in a cognitive process should be crucially driven by the chunks stored in the goal and/or imaginal buffers.

### 7.4.2 Bayes+ACT-R Lexical Decision with Imaginal-Buffer Involvement and Default Encoding Delay for the Imaginal Buffer

We now turn to the examination of the first of two alternative Bayes+ACT-R models, both of which crucially involve the imaginal buffer as an intermediary between

---

8For more information about $\hat{R}$, see Gelman et al. (2013, 284–286), and also pymc3’s implementation of and help file for the Gelman-Rubin diagnostic. Plots of the chains for the models in this book are available in the pyactr-book github repository (see the ‘figures’ folder, specifically file names that end with ‘…_trace’), and so are the chains themselves (see the ‘data’ folder).
the visual what and retrieval buffers. The full code for the model discussed in this subsection is available at the link provided in the appendix to this chapter—see Sect. 7.7.1.

The Bayesian model remains the same, the only part we change is the ACT-R likelihood for latencies. Specifically, we modify the procedural core of the ACT-R model as shown in [py43] below. We first add the imaginal buffer to the model (line 1 in [py43]), and then replace the "attend word" and "retrieving" rules with three rules "attend word" (lines 4–21), "encoding word" (lines 28–42) and "retrieving" (48–62).

The new rule "encoding word" mediates between "attend word" and "retrieving". The visual value retrieved by the "attend word" rule is shifted into the imaginal buffer by the "encoding word" rule. Then, the "retrieving" rule takes that value, i.e., word form, from the imaginal buffer and places it into the retrieval buffer.

```
[py43] >>> lex_decision.set_goal("imaginal")
set()

>>> lex_decision.productionstring(name="attend word", string="**
... =g> 4
... isa  goal
... state  attend
... =visual_location>
... isa  _visuallocation
... ?visual>
... state  free
... =>
... =g>
... isa  goal
... state  encoding
... +visual>
... isa  _visual
... cmd  move_attention
... screen_pos =visual_location
... "visual_location>
... **")

('=g': goal(state= attend), '='visual_location': _visuallocation(color= ,
 screen_x= , screen_y= , value= ), '?visual': ('state': 'free'))

>>> lex_decision.productionstring(name="encoding word", string="**
... =g> 28
... isa  goal
... state  encoding
... =visual>
... isa  _visual
... value  =val
... =>
... =g>
... isa  goal
... state  retrieving
... +imaginal>
... isa  word
... form  =val
... **")

('=g': goal(state= encoding), '='visual': _visual(cmd= move_attention,
 color= , screen_pos= =visual_location, value= ), 'visual_location': None)

>>> lex_decision.productionstring(name="retrieving", string="**
... =g> 47
... isa  goal
... state  retrieving
... +imaginal': word(form= =val))

```
These modifications are all symbolic (discrete, non-quantitative) modifications. We will however be able to fit the new model to the same data and quantitatively compare it with the no-imaginal-buffer model discussed in the previous subsection.

The top plot in Fig. 7.6 shows that the model has a very poor fit to the latency data. Adding the imaginal-buffer mediated encoding step adds 200 ms to every lexical decision simulation, since 200 ms is the default ACT-R delay for chunk-encoding into the imaginal buffer.

We therefore see that the predicted latencies for all 16 word-frequency bands are greatly overestimated (they are far above the red diagonal line). The model with the imaginal buffer cannot run faster than about 725 ms, at least not when the imaginal-buffer encoding delay is left at its default 200 ms value.

We can think of the 200 ms imaginal delay as part of the baseline intercept for our ACT-R model. The intercept is simply too high to fit high-frequency words, for which the lexical decision task should take 100 to 200 ms less than this intercept.

The $R_{\text{hat}}$ values for this model are once again below 1.1 (in fact, they are very close to 1):

$R_{\text{hat}} = \{\begin{array}{l}
\cdot\text{threshold}: 1.0066039259999719, \\
\cdot\text{decay}: 1.0064848072431016, \\
\cdot\text{noise}: 0.9999915488454998, \\
\cdot\text{IF}: 1.0020696744920665, \\
\cdot\text{le}: 1.003927035222827, \\
\cdot\text{mu}_\text{rt}: \text{array}([1.00000000, 1.00000000, 0.99995000, 1.00000334, 1.00000562, 1.00001281, 1.00001532, 1.00002131, 1.00002576, 1.00003001, 1.00003510, 1.00004000, 1.00004551, 1.00005222, 1.00005983, 1.00006826, 1.00007439]), \\
\cdot\text{mu}_\text{prob}: \text{array}([1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000, 1.00000000])
\end{array}\}$

We see here one of the main benefits of our Bayes+ACT-R framework. We are able to fit any model to experimental data, and we are able to compute quantitative predictions (means and credible intervals) for any model. We are therefore able to quantitatively compare our qualitative theories.

In this particular case, we see that a model that is cognitively more plausible fares more poorly than a simpler, less realistic model.
7.4.3 Bayes+ACT-R Lexical Decision with Imaginal Buffer and 0 Delay

We will now improve the imaginal-buffer model introduced in the previous subsection by setting the imaginal delay to 0 ms, instead of its default 200 ms value. When we built our left-corner parser in Chap. 4, we already saw that the imaginal delay might need to be decreased if we want to have empirically-adequate models of linguistic phenomena. This is because natural language interpretation involves incremental construction of rich hierarchical representations that seriously exceed in complexity the representations needed to model other high-level cognitive processes in ACT-R.

The full code for the model discussed in this subsection is once again available at the link provided in the appendix to this chapter—see Sect. 7.7.1. The only change relative to the model in the previous subsection is setting the delay for the imaginal buffer to 0 when the model is reset to its initial state in the run_stimulus(word) function.
The resulting predictions are plotted against the observed data in Fig. 7.7. We see here that, once the latency ‘intercept’ of the ACT-R model is suitably lowered by removing the imaginal-encoding delay, a cognitively plausible model that makes crucial use of the imaginal buffer can fit the data very well.

The Rhat values for this model are also below 1.1:

\[
\begin{align*}
\text{threshold:} & \quad 1.0239260960818166, \\
\text{decay:} & \quad 1.0240525803780482, \\
\text{noise:} & \quad 1.0002178227957363, \\
\text{lf:} & \quad 1.014459091108735, \\
\text{le:} & \quad 1.0015428790086978, \\
\text{mu_rt:} & \quad [1.00355696, 1.00402281, 1.00437915, 1.00471154, 1.00506452, 1.00545713, 1.00575474, 1.00605512, 1.00633888, 1.00655243, 1.00678295, 1.00685419, 1.00690249, 1.00638288, 1.00460097, 1.00171009], \\
\text{mu_prob:} & \quad [1.00049237, 1.00062568, 1.00073162, 1.00083857, 1.00094761, 1.00107675, 1.00119749, 1.00130761, 1.00144432, 1.00157235, 1.00167635, 1.00177533, 1.00196299, 1.00205086, 1.00156458, 1.00025179].
\end{align*}
\]
We now have a formally explicit way to connect competence-level theories to experimental data via explicit processing models. That is, we can formally, explicitly connect qualitative (symbolic, competence-level) theory construction—the main business of the generative grammarian—and quantitative (subsymbolic, performance-level) statistical inference and model comparison based on experimentally collected behavioral data—the main business of the experimental linguist.

Traditionally, these are separate activities that are only connected informally. The fundamental vagueness of this informal connection is intrinsically unsatisfactory. But, in addition to that, this vagueness encourages the generative grammarian and the experimental linguist to work in separate spheres, with the generative grammarian developing sophisticated theories with a relatively weak empirical basis, and the experimental linguist often using an informal, overly simplified theory that can fit in the Procrustean bed of a multi-way ANOVA (or similar linear models).

There are several reasons for embedding ACT-R models in Bayesian models for statistical inference, rather than just using maximum likelihood estimation. These reasons are not specific to ACT-R models, but are brought into sharper relief by the complexity of these models relative to the generalized linear models standardly used in (psycho)linguistics.

The first reason is that we can put information from previous ACT-R work into the priors. Most importantly, however, the Bayesian framework enable us to perform generalized model comparison (via Bayes factors, or using other criteria). In contrast, maximum likelihood model comparison fails for models for which we cannot estimate the number of parameters in the model. Estimating the number of parameters is already difficult for models with random effects. For hybrid symbolic-subsymbolic models like the ACT-R ones we have been constructing, the question of identifying the “number” of parameters is not even well-formed.

For a distinct line of argumentation that the integration of ACT-R and Bayesian models is a worthwhile endeavor, see Weaver (2008).

### 7.5 Modeling Self-paced Reading with a Left-Corner Parser

Apart from the lexical decision model, Chap. 4 also showed how to implement a left-corner parser model in ACT-R/pyactr. We noted in that chapter that the parsing model was not realistic due to, among other things, its simplifying assumption that memory retrievals of lexical information always take a fixed amount of time, irrespective of the specific state and properties of the components of the recall process (the specific recall cue, the state of declarative memory, the contents of the other buffers etc.). Since we now have a more realistic model of lexical access at our disposal, we might want to investigate whether this model could also be used to improve our parsing model.
We can go even further than that: one interesting property of ACT-R is that it assumes one model of retrieval irrespective of the cognitive sub-domain under consideration. We can therefore ask how this model of retrieval fares with respect to language: can we model both syntactic and lexical retrieval using the same mechanisms and the same parameter values within one ACT-R model? ACT-R/pyactr left-corner parsing models addressing these questions were introduced and discussed in Brasoveanu and Dotlačil (2018). In this section, we will summarize the main points of that work.

Brasoveanu and Dotlačil (2018) studied the fit of the parser to human data by simulating Experiment 1 in Grodner and Gibson (2005). This is a self-paced reading experiment (non-cumulative moving-window; Just et al. 1982): participants read sentences that do not appear as a whole on the screen. Rather, they have to press a key to reveal the first word, and with every key press, the previous word disappears and the next word appears. What is measured (and modeled) is how much time people spend on each word.

The modeled self-paced reading experiment has two conditions. In one condition, the subject noun phrase is modified by a subject-gap relative clause—see (3) below. In the second condition, the subject noun phrase is modified by an object-gap relative clause—see (4) below.

Using relative clauses is crucial, since this allows us to study the properties of syntactic retrieval. At the gap site, indicated as \( t_i \) in (3/4) below, the parser has to retrieve the wh-word from declarative memory to correctly interpret the relative clause. Studying the reading-time profiles of these sentences can therefore help us understand the latencies of both lexical and syntactic recall.

(3) The **reporter who**; \( t_i \) **sent the photographer to the editor** hoped for a story.

(4) The **reporter who**; the **photographer sent** \( t_i \) **to the editor** hoped for a story.

Brasoveanu and Dotlačil (2018) modeled 9 regions of interest (ROIs), boldfaced in (3/4) above. These are word 2 (the matrix noun in subject position) through word 10 (the matrix verb).

Just as when we modeled lexical decision, Brasoveanu and Dotlačil (2018) built more than one model and quantitatively compared them. This comparison is a necessary part of developing good ACT-R models, and cognitive models in general. But more importantly, it enables us to gain insight into underlying (unobservable) cognitive mechanisms by identifying the better fitting model(s) in specific ROIs.

In total, three models were created to simulate self-paced reading and parsing. All three models were extensions of the eager left-corner parser described in Sect. 4.4. The two main modifications were: (i) the parser was extended with a more realistic model of lexical access, the same as the one used in the second ACT-R model for lexical decision in this chapter (see Sect. 7.3), and (ii) the parser had to recall the

---

9 A small subset of this data was studied in a different ACT-R model of parsing, the left-corner parser discussed in Lewis and Vaishth (2005).

10 As well as statistical models, machine learning models etc.
wh-word in the relative clause to correctly parse it. The parser incorporated visual and motor modules, just like the one in Chap. 4.

The three models differ from each other in two respects. First, Models 1 and 2 assume a slightly different order of information processing than Model 3. Models 1 and 2 are designed in a strongly serial fashion:

- first, a word $w$ is attended visually;
- after that, its lexical information is retrieved, and syntactic retrieval also takes place (if applicable, e.g., when we need to retrieve the relativizer $who$);
- the parse tree is then created and, finally,
- visual attention is moved to the next word $w+1$ at the same time as the motor module is instructed to press a key to reveal that word;
- then the whole process is repeated for word $w+1$.

The processes in Model 3 were staged in a more parallel fashion: after lexical retrieval, syntactic retrieval (if applicable) and syntactic parsing happened at the same time as visual-attention and motor commands were prepared and executed. This difference is schematically shown in Figs. 7.8 and 7.9.

The second way the models differ is with respect to the analysis of subject gaps. Models 1 and 3 assume that the parser predictively postulates the subject gap immediately after reading the wh-word (word 3 in (3) and (4)). This strategy should slow
down the parser on the wh-word itself, since it has to postulate the upcoming gap when reading it. But the strategy predicts that the parser will speed up when reading the following word in the subject-relative clause sentence (3), since the parser has already postulated the gap and nothing further needs to be done to parse the gap.

In contrast, Model 2 assumes that no subject gap is predictively postulated when reading the wh-word: the gap is parsed bottom-up. This strategy predicts faster reading times on the wh-word compared to Models 1 and 3. But it also predicts a slowdown on the next word in the subject-relative clause sentence (3), since it is at this point that the subject gap is parsed/postulated.

Why would we compare these three models? The main reason is to test two distinct hypotheses (qualitative/symbolic theories) about the human processor. These hypotheses are commonly entertained in psycholinguistics, but are not usually fully formalized and computationally implemented.

And it is important to realize that we can never really establish at an informal level if hypotheses embedded in complex competence-performance theories like the ones we’re entertaining here make correct predictions. To really test hypotheses and theories at this level of complexity, we need to fully formalize and computationally implement them, and then attempt to fit them to experimental data. We don’t really know what a complex model does until we run it.

The two hypotheses we test are the following. First, we implement and test the standard assumption that the parser is predictive and fills in gap positions before they appear (cf. Stowe 1986; Traxler and Pickering 1996). Given that hypothesis, we expect that Models 1 and 3 fit the reading data better than Model 2.

Second, it is commonly assumed that processing is to some degree parallel. In particular, a standard assumption of one of the leading models of eye movement (E-Z Reader, Warren et al. 2009) is that moving visual attention to word $n+1$ happens alongside with the syntactic integration of word $n$. Under this hypothesis, we expect Model 3 to have a better fit than Models 1 and 2.

Both predictions turn out to be correct, supporting previous claims and showing that these claims hold under the careful scrutiny of fully formalized and computationally implemented end-to-end models that are quantitatively fit to experimental data.

Equally importantly, this shows that our Bayes+ACT-R framework can be used to quantitatively test and compare qualitative (symbolic) hypotheses about cognitive processes underlying syntactic processing.

The code for Model 3 is linked to in the appendix to this chapter (see Sect. 7.7.2). The three models were fit to experimental data from Grodner and Gibson (2005) (their Experiment 1) using the Bayesian methods described previously in this chapter.

Four parameters were estimated: the $k$ parameter, which scales the effect of visual distance, the rule firing parameter $r$, the latency factor $lf$ and the latency exponent $le$. Of these, only the first two have not been discussed in this chapter.

The rule firing parameter specifies how much time any rule should take to fire and has been (tacitly) used throughout the whole book. The default value of this parameter, which we always used up to this point, is 50 ms.
The \( k \) parameter is used to modulate the amount of time that visual encoding \( T_{enc} \) takes, and it has been discussed in Chap. 4, Sect. 4.3.1.\(^\text{11}\) We fit the \( k \) parameter mainly to show that parameters associated with peripherals (e.g., the visual and motor modules) can be jointly estimated with procedural and declarative memory parameters when fitting full models to data.

After we fit (the parameters of) the three models to data, we collect the posterior predictions for the 9 ROIs in the two (subject-gap and object-gap) conditions, plotted in Figs. 7.10, 7.11 and 7.12. The diamonds in these graphs indicate the actual, observed mean RTs for each word from Grodner and Gibson (2005). The bars provide the 95% CRIs for the posterior mean RTs, which are plotted as filled dots.

It is important to note that what we estimate here are parameters for the full process of reading the 9 ROIs. We do not estimate means and CRIs region by region (which is the current standard in the field), falsely assuming independence and leaving the underlying dependency structure, i.e., the parsing process, largely implicit and unexamined.

Figure 7.10 shows that Model 1 captures wh-gap retrieval well: the observed mean reading times on the 3rd word (sent) in the top panel (subj-gap) and the 5th word (also sent) in the bottom panel (obj-gap) fall within the CRI. However, the spillover effect on the word after the object gap—the 6th word (to) in the bottom panel—is not captured: the model underestimates it pretty severely.

\(^\text{11}\)Recall that \( T_{enc} \) is specified by the function \( K \cdot (−\log f) \cdot e^{kd} \), where \( k \) is the parameter that scales the effect of visual distance \( d \) measured in degrees of visual angle, and \( f \) is the (normalized) frequency of the object (word) being encoded. However, since word frequency affects lexical retrieval, we do not need to use it in visual encoding, so we substitute word length (a straightforward visual property of a word) for \( −\log f \). \( K \) is another parameter, set to its default value of 0.01.
The posterior predictions of Model 2, provided in Fig. 7.11, are clearly worse: the 95% CRI s are completely below the observed mean RTs for the wh-word in both conditions, and also for the word immediately following the wh-word in the object-gap condition. This indicates that the model underestimates the parsing work triggered by the wh-word, and it also underestimates the reanalysis work that needs to be done on the word immediately following the wh-word in the object-gap condition.

Finally, Model 3 is the best among these three models. It captures the spillover effect for object gaps and increases the precision of the estimates (note the smaller CRI s). At the same time, Model 3 maintains the good fit exhibited by Model 1 (but not Model 2) for the wh-word and the following word in both conditions. This is shown in Fig. 7.12. As we already mentioned, the code for this final and most successful model is linked to in the appendix to this chapter (Sect. 7.7.2).

This relatively informal quantitative comparison between models can be made more precise by using WAIC measures for model comparison. For example, if we use WAIC$_2$, which is variance based, we can clearly see that Model 3 has the most precise posterior estimates for the Grodner and Gibson (2005) data; see Brasoveanu and Dotlačil (2018) for more details.

Thus, we see that the left-corner parser, first introduced in Chap. 4, can be extended with a detailed, independently motivated theory of lexical and syntactic retrieval. The resulting model can successfully simulate reading time data from a self-paced reading experiment.

The fact that the three models we considered help us distinguish between several theoretical assumptions, and that the model with the best fit implements hypotheses

---

$^{12}$See Gelman et al. (2013) for more discussion of WAIC$_1$ and WAIC$_2$. 

Fig. 7.11 Model 2: no postulated subject gaps
that we expect to be correct for independent reasons, is encouraging for the whole enterprise of computational cognitive modeling pursued in this book.

Finally, we see that Model 3 does not show any clear deficiencies in simulating the mean RTs of self-paced reading tasks, even though it presupposes one and the same framework for both lexical and syntactic retrieval. This supports the ACT-R position of general recall mechanisms across various cognitive sub-domains, including linguistic sub-domains such as lexical and syntactic memory.

Before concluding, we have to point out that, even though the investigation presented in Brasoveanu and Dotlačil (2018) and summarized in this section is very promising, it is rather preliminary, particularly when compared to the models in the rest of this chapter and the rest of this book. Furthermore, the estimates of the three models we just discussed were obtained using different sampling methods than the ones we use for the Bayes+ACT-R models throughout this book.

Improving on these preliminary results and models, and investigating if the sampling methods used in this book would substantially benefit the ACT-R models in Brasoveanu and Dotlačil (2018) is left for a future occasion.

### 7.6 Conclusion

The models discussed in this chapter show that the present computational implementation of ACT-R can be used to successfully fit data from various linguistic experiments, as well as compare and evaluate assumptions about underlying linguistic representations and parsing processes. While one could investigate many experiments using the presented methodology, we opted for a different approach here,
focusing only on a handful of studies and dissecting modeling assumptions and the way computational modeling can be done in our Bayes+ACT-R/pymc3+pyactr framework.

We take the results to be encouraging. We believe they provide clear evidence that developing precise and quantitatively accurate computational models of lexical access and syntactic processing is possible in the proposed Bayes+ACT-R framework, and a fruitful way to pursue linguistic theory development.

Unfortunately, the models are clearly incomplete with respect to many aspects of natural language use in daily conversation. An important aspect that is completely missing in these models stands out: natural language meaning and interpretation.

Our goal in conversation is to understand what others tell us, not (just) recall lexical information and meticulously parse others’ messages into syntactic trees. Ultimately, we construct meaning representations, and computational cognitive models should have something to say about that. This is precisely what the next two chapters of this book will address.

7.7 Appendix: The Bayes and Bayes+ACT-R Models

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.

7.7.1 Lexical Decision Models

File ch7_first_three_models.py:
☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_first_three_models.py.

File ch7_lexical_decision_pyactr_no_imaginal.py:
☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_lexical_decision_pyactr_no_imaginal.py.

File ch7_lexical_decision_pyactr_with_imaginal.py:
☞ https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_lexical_decision_pyactr_with_imaginal.py.
7.7 Appendix: The Bayes and Bayes+ACT-R Models

File `ch7_lexical_decision_pyactr_with_imaginal_delay_0.py`:
☞ [GitHub link](https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_lexical_decision_pyactr_with_imaginal_delay_0.py).

7.7.2 Left-Corner Parser Models

File `readme.txt` explains how to run the model:
☞ [GitHub link](https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_grodner_gibson/readme.txt).

File `parser_rules.py`:
☞ [GitHub link](https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_grodner_gibson/parser_rules.py).

File `run_parser.py`:
☞ [GitHub link](https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_grodner_gibson/run_parser.py).

File `estimation_subj_obj_extraction.py`:
☞ [GitHub link](https://github.com/abrsvn/pyactr-book/blob/master/book-code/ch7_grodner_gibson/estimation_subj_obj_extraction.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.