Phonological Data and Analysis Volume 6, Article 3: 1–44 (2024)
https://doi.org/10.3765/pda.v6art4.88

Introducing `maxent.ot`: an R package for Maximum Entropy constraint grammars

Connor Mayer[a] & Adeline Tan[b] & Kie Zuraw[c]*

[a]University of California, Irvine – cjmayer@uci.edu,
[b]University of California, Los Angeles – aderaetan@ucla.edu,
[c]University of California, Los Angeles – kie@ucla.edu

This paper presents `maxent.ot`, a package for doing phonological analysis using Maximum Entropy Optimality Theory written in the statistical programming language R. R has become the de facto standard for doing statistical analysis in linguistic research, and this package allows phonologists to create and disseminate MaxEnt OT analyses in R. A central goal of the package is to support reproducible research and to allow the crucial components of a MaxEnt analysis to be performed conveniently and with only a basic knowledge of R programming. The paper first presents a tutorial on MaxEnt constraint grammars and how to use `maxent.ot` to perform a simple analysis. We then turn to more advanced features of the package, including model comparison, regularization, and cross-validation.

Keywords: reproducible research; Maximum Entropy; constraint grammar; R; software

1 Introduction: Why this package?

1.1 The problem of switching out of an analysis script

If your research includes any kind of quantitative, computational, or statistical component, you have probably experienced the situation of returning to a project after a break and being confronted with a tangle of files. Perhaps you have multiple versions of a spreadsheet, with names like `version4_corrected_-USETHISONE_real.csv`. If you're lucky, you've left yourself a text file with notes on the series of steps needed to clean and process spreadsheet data, and a statistics script with comments indicating how to make the input files it needs. But in trying to implement the steps, you likely find that not all of the steps are as unambiguously recorded as you'd thought when you wrote them. The frustrations of working this way have led many researchers to move towards integrating as much as possible of an analysis into a single script that can easily be modified and re-run, in the spirit of *reproducible research* (see Section 2).

The programming language R has become a standard tool for such analyses due to its robust array of packages for statistical analysis and data visualization (R Core Team 2022). For researchers who use Maximum Entropy constraint grammars, however, there has not been a good way to carry out constraint-weight fitting within R. Instead, it was necessary to leave an R script and fit weights in external software, such as the MaxEnt Grammar Tool (Hayes et al. 2009) or the Microsoft Excel Solver (Fylstra et al. 1998).

*Thanks to the audience at the 2022 Annual Meeting on Phonology at UCLA for their useful feedback and discussion. Thanks also to the valuable feedback from two anonymous reviewers.

1.2 Goals of this software

We created this package to make life easier for phonologists and others who use Maximum Entropy constraint grammars—including those who argue against them. We also want to make it easier for analysts to evaluate and build on each other's work. Specifically, this package makes it easy to carry out an end-to-end MaxEnt analyses within a a single R script. Our package can be used alongside existing downloadable R code for carrying out various functions in Harmonic Grammar generally and MaxEnt specifically (Staubs 2011).

1.3 Overview

In the following sections, we discuss the concept of reproducible research (Section 2), and then step through the functionality of the `maxent.ot` package, including fitting, generating predictions, model comparison, and the use of *prior* (or *bias*) terms (Section 3). Finally, we present the machine learning technique of cross-validation as an approach for avoiding over-fitting and setting parameters of the prior term (Section 4). Appendices present the OTSoft input file format, which our package is compatible with, and the use of a temperature parameter when generating predictions.

Different readers will want to read different parts of this paper. Readers already familiar with how Maximum Entropy constraint grammars work in phonology may skip the sections that contain background information, which are labelled "MaxEnt background": Sections 3.4, 3.5, 3.6, and 3.10.1.

2 Reproducible research

Reproducibility in research (Schwab et al. 2000; Peng 2011; Stodden et al. 2014) means making it easier for someone to re-run an analysis. *Reproduction* is not the same as *replication*: replicating a result requires new data, such as from a new experiment; reproducing a result uses the same data as the original study. Reproducibility supports replicability, though, by making it easier to ensure that only the data have changed and that the method of analysis remains the same.

To make research reproducible, as much as possible the analysis should be run from a single script. The script should take as its input raw, unprocessed data, such as a public database, or a results file written by experimental software. The script should do all data cleaning and organization, and carry out all statistical analyses. Ideally, the script produces a document combining explanatory text, easy-to-read code chunks, statistical results, tables, and figures. (See Section 2.1 for the mechanics of how this works.) The movement for reproducible research has roots in the movement for literate programming (Knuth 1992).

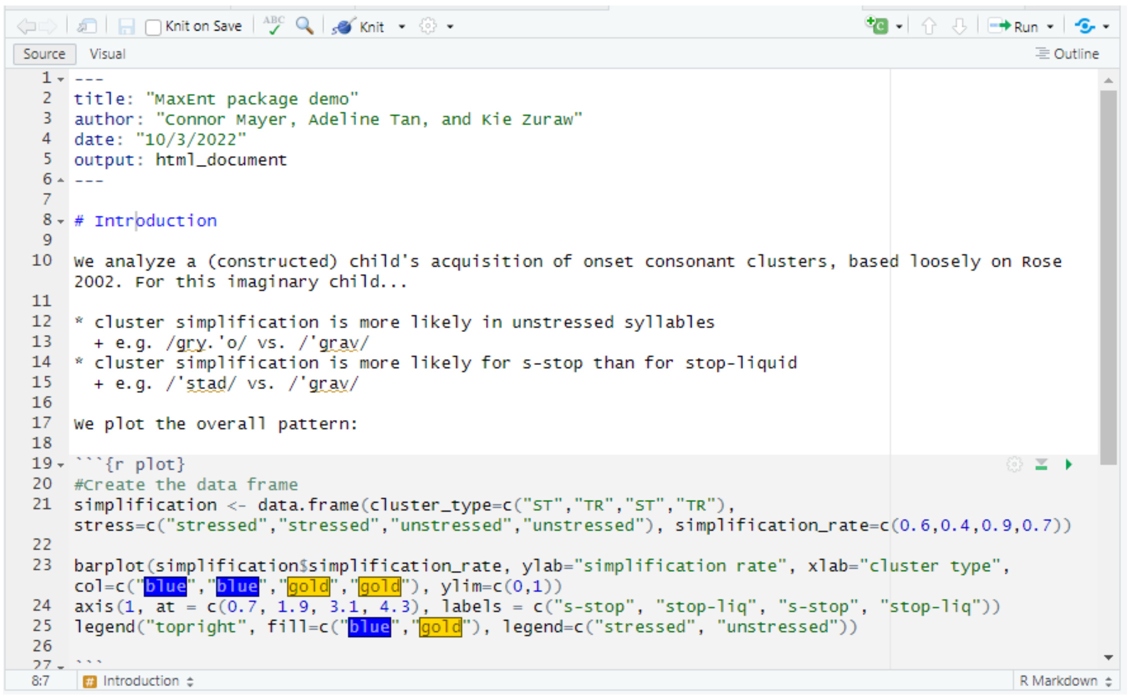
All this is in contrast to the approach that some of us have employed in the past, where we carry out a series of data-cleaning operations using spreadsheet software—ideally keeping notes on how this is done, but often finding those notes to be confusing or inadequate—then perhaps run a Python program, then an R script, then copy and paste results from our R console and write figures to separate files.

2.1 R Markdown for reproducible research

Commonly used tools for reproducible research are R Markdown files (Allaire et al. 2021) for R, and Jupyter notebooks (Kluyver et al. 2016) or Google Colab (Bisong 2019) for Python (Van Rossum & Drake 2009). We will focus here on R Markdown, since our package is for use in R.

R (R Core Team 2022) is a widely-used statistical programming language and software environment. One reason for R's popularity is the existence of thousands of user-contributed *packages*. There are packages for plotting vowel acoustics (`visvow;` Heeringa & Van de Velde 2018), working with basketball play-by-play data (`wehoop`; Gilani & Hutchinson 2021), detecting blood doping (`ABPS`; Schütz & Zollinger 2018), and making maps of Brazil (`geobr`; Pereira et al. 2019), among many, many others.

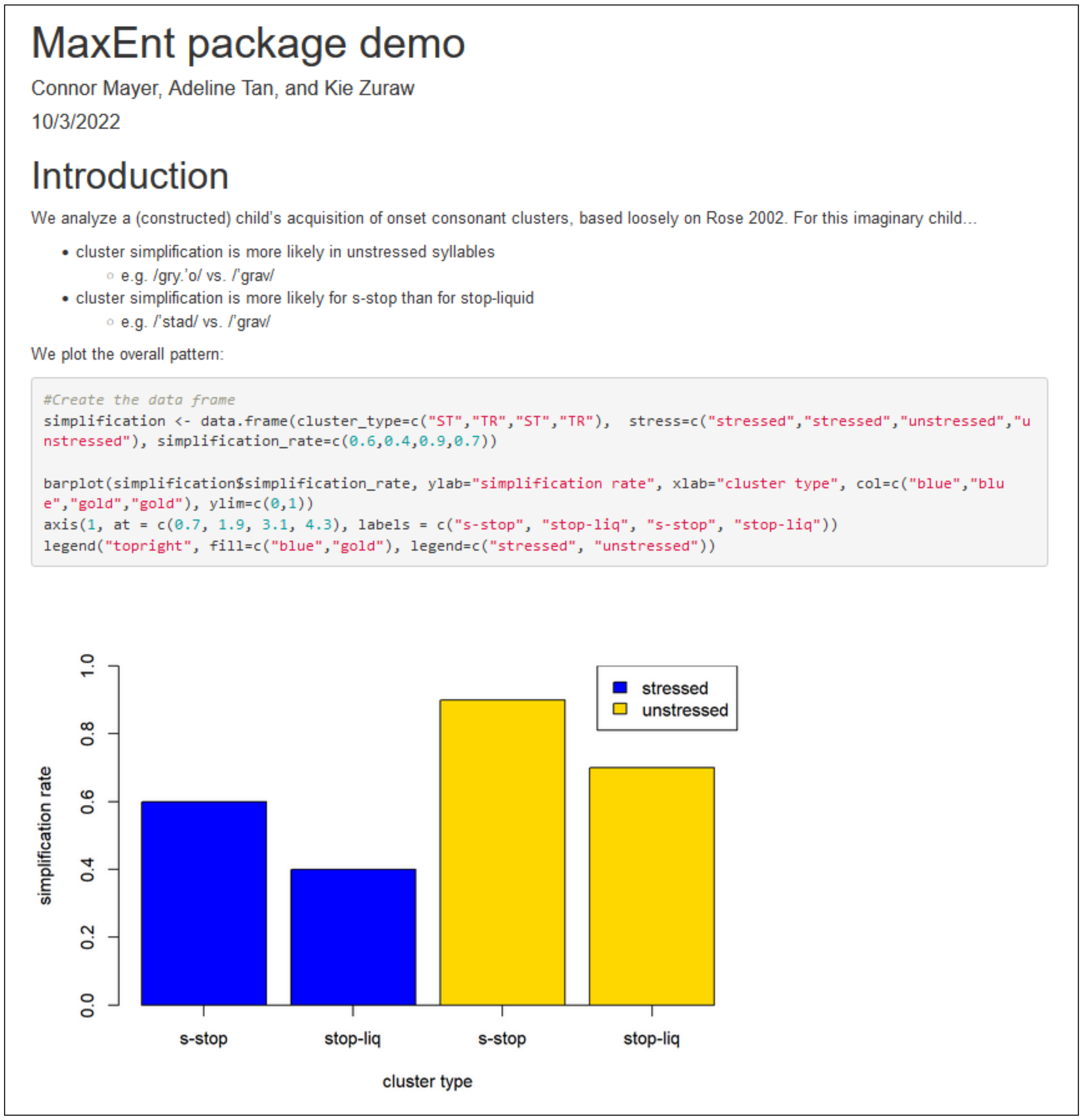
It is possible to type R commands into the R command line interpreter, one at a time. In an analysis of any complexity, however, it is generally easier to assemble a series of commands into a script, with the possibility of including comments. But a much bigger step towards reproducibility is to use *R Markdown*.



~~~
---
title: "MaxEnt package demo"
author: "Connor Mayer, Adeline Tan, and Kie Zuraw"
date: "10/3/2022"
output: html_document
---

# Introduction

We analyze a (constructed) child's acquisition of onset consonant clusters, based loosely on Rose
2002. For this imaginary child...

* cluster simplification is more likely in unstressed syllables
  + e.g. /gry.'o/ vs. /'grav/
* cluster simplification is more likely for s-stop than for stop-liquid
  + e.g. /'stad/ vs. /'grav/

We plot the overall pattern:

```{r plot}
#Create the data frame
simplification <- data.frame(cluster_type=c("ST","TR","ST","TR"),
stress=c("stressed","stressed","unstressed","unstressed"), simplification_rate=c(0.6,0.4,0.9,0.7))

barplot(simplification$simplification_rate, ylab="simplification rate", xlab="cluster type",
col=c("blue","blue","gold","gold"), ylim=c(0,1))
axis(1, at = c(0.7, 1.9, 3.1, 4.3), labels = c("s-stop", "stop-liq", "s-stop", "stop-liq"))
legend("topright", fill=c("blue","gold"), legend=c("stressed", "unstressed"))

```
~~~

Figure 1: A sample of R Markdown in RStudio.

Figure 1 shows what an R Markdown file looks like when open in RStudio (RStudio Team 2020), which is a integrated development environment for R—that is, software for writing and running R code. There is a light sprinkling of markup tags, such as the block at the top with the file's metadata (title, author, date, output type), tags for section and sub-section headers (#, ##, etc.), bulleted lists (*, +), and more. Most important is the code chunk marked off with ``` and automatically shaded in grey by RStudio. This is where the analyst types R code, just as they would in an R script or on the command line. For more information about R Markdown, see Xie et al. (2018, 2020).



MaxEnt package demo

Connor Mayer, Adeline Tan, and Kie Zuraw

10/3/2022

Introduction

We analyze a (constructed) child's acquisition of onset consonant clusters, based loosely on Rose 2002. For this imaginary child...

- cluster simplification is more likely in unstressed syllables
  - e.g. /gry.'o/ vs. /'grav/
- cluster simplification is more likely for s-stop than for stop-liquid
  - e.g. /'stad/ vs. /'grav/

We plot the overall pattern:

```
#Create the data frame
simplification <- data.frame(cluster_type=c("ST","TR","ST","TR"),  stress=c("stressed","stressed","unstressed","u
nstressed"), simplification_rate=c(0.6,0.4,0.9,0.7))

barplot(simplification$simplification_rate, ylab="simplification rate", xlab="cluster type", col=c("blue","blu
e","gold","gold"), ylim=c(0,1))
axis(1, at = c(0.7, 1.9, 3.1, 4.3), labels = c("s-stop", "stop-liq", "s-stop", "stop-liq"))
legend("topright", fill=c("blue","gold"), legend=c("stressed", "unstressed"))
```

Figure 2: The knitted output of the R Markdown file in Figure 1.

When the analyst presses the Knit button, the knitr package (Xie 2014) creates a output in HTML, PDF (via a LaTeX template), Microsoft Word, Beamer, or various other formats. Figure 2 shows an example. The text from the markdown file is now formatted, the code chunk is printed (there is an option to prevent some or all code chunks from appearing in the output), and the figure produced by the code is displayed. A figure can also be written to a file.

2.2 Benefits of reproducible research

An immediate advantage of the reproducible approach that whenever a researcher wants to correct an error or add some more analysis, they only need to edit the script and then press a button to re-run everything—they don't need to repeat a whole series of steps.

It also becomes easier to return to a project after a break, such as after getting journal reviews back. There is no need to hunt for multiple files or remember procedures for analysis, because everything is organized in one file.

Preventing, catching, and correcting errors becomes easier in the reproducible framework, because all steps of the analysis are written down for the researcher's inspection.

In addition to these benefits to the researcher over the course of a project, reproducibility makes it easier for others to understand, check, and build on our work. At the most basic level, another researcher—or the original researcher's own future self—can run the same script on the same input data, step by step, to understand the analysis thoroughly and perhaps check for mistakes. Or a reader can copy chunks of code to adapt for their own use. But a reproducible script can also be used to run the same analysis on a different set of data—for instance, to compare two corpora, or to apply the same analysis to a replication of an experiment—for maximum comparability. A reproducible script can be used to try out variations on the original analysis, for example to see whether a statistical significance result holds up if random slopes are added.

3 Carrying out a MaxEnt analysis using `maxent.ot`

In this section, which makes up the bulk of this article, we give a tutorial introduction to Maximum Entropy Optimality Theory (henceforth MaxEnt) and the `maxent.ot` package. The analysis will illustrate the main functions of our package using a small, fabricated data set. The reader can download tutorial material from https://github.com/connormayer/maxent.ot_pda_material to follow along.

As mentioned earlier, readers familiar with MaxEnt models may want to skip the sub-sections labelled "MaxEnt background".

3.1 Installing the package

In order to use `maxent.ot`, you will need to have the programming language R installed on your computer (R Core Team 2022: https://www.r-project.org/). You should also install RStudio (https://posit.co/download/rstudio-desktop/), an integrated development environment for R.

There are two ways to install the `maxent.ot` package. You can install it from the CRAN repository using the following command:

```r
install.packages("maxent.ot")
```

You can also install it from the development GitHub repository using the following commands, which will install the package only if it is not already installed:

```r
# This installs the devtools package if necessary
if (!require(devtools)) {
  install.packages("devtools", repos = "http://cran.us.r-project.org")
}
if (!require(maxent.ot)) {
  devtools::install_github("connormayer/maxent.ot")
}
```

If the package is installed, but you want to re-install it—for example, in case there is a more recent version—then use the following commands instead:

```r
if (!require(devtools)) {
  install.packages("devtools", repos = "http://cran.us.r-project.org")
}
devtools::install_github("connormayer/maxent.ot")
```

If the package has been installed successfully, you should be able to run the following command without error.

```
library(maxent.ot)
```

3.2 A running example: simplification of onset clusters by French-acquiring children

Throughout this paper we'll use a simple, fabricated data set that is loosely based on Rose (2002). This data set consists four types of French words with complex onsets, and the rate at which each word type's onset is realized faithfully or simplified by an imaginary French-acquiring child. The dataset is shown in Table 1.

Underlying	Surface	Observed count
/ˈstad/	[ˈstad]	40
	[ˈtad]	60
/ˈgʁo/	[ˈgʁo]	60
	[ˈgo]	40
/spa.gɛ.ˈti/	[spa.gɛ.ˈti]	10
	[pa.gɛ.ˈti]	90
/gʁy.ˈjo/	[gʁy.ˈjo]	30
	[gy.ˈjo]	70

Table 1: A fabricated data set based on Rose (2002).

The frequencies in this data set reflect two specific properties:

1. Cluster simplification is more likely to occur in unstressed syllables (e.g., /gʁy.ˈjo/ vs. /ˈgʁo/). This is a real aspect of Rose's data, although the frequencies are invented;

2. Cluster simplification is more likely in onsets that violate the Sonority Sequencing Principle (e.g, Steriade 1982; Selkirk 1984): that is, it is more likely for fricative-stop sequences than for stop-liquid sequences (e.g., /ˈstad/ vs. /ˈgʁo/). This is not part of Rose's data, but it has been observed in both L1 (e.g., Yavaş et al. 2009; Jarosz 2017) and L2 (e.g., Carlisle 1991, 2001) learners that complex onsets with greater sonority rises tend to be produced more accurately.

These patterns are visualized in Figure 3.

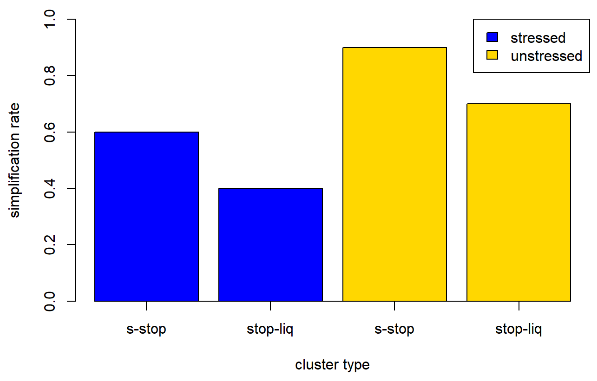


Figure 3: Rates of simplification of different types of onset cluster.

3.3 Formatting data for `maxent.ot`

The primary input data to `maxent.ot` are tableaux. These consist of input/output pairs with corresponding frequency counts and violation profiles. Our input files will use the input/output pairs and counts from Table 1 and the constraints introduced above. For the time being we'll consider only the two constraints *COMPLEX and MAX, which will be defined in the following section. Note that these input files do not include constraint weights: we'll talk below about how weights can be specified by the analyst or learned from the data.

The functions in the `maxent.ot` package expect the input data to be in the format of either a data frame, which is a table-like structure that is built into R, or a tibble, which is a similar structure used by the popular `tidyverse` package (Wickham et al. 2019).

An example of this format is shown in Table 2, and the corresponding file can be found at `data/simple_input_dataframe.csv` in the tutorial materials.

Input	Output	Frequency	StarComplex	Max
ˈstad	ˈstad	40	1	
ˈstad	ˈtad	60		1
ˈgʁav	ˈgʁav	60	1	
ˈgʁav	ˈgav	40		1
spa.gɛ.ˈti	spa.gɛ.ˈti	10	1	
spa.gɛ.ˈti	pa.gɛ.ˈti	90		1
gʁy.ˈjo	gʁy.ˈjo	30	1	
gʁy.ˈjo	gy.ˈjo	70		1

Table 2: `maxent.ot` formatted input. Note that the actual input files are comma separated, but we present the data in a tabular fashion here for clarity.

The first step to using the `maxent.ot` package is to either use R code to create a `dataframe` or `tibble` containing the input data, or, as we'll do here, to load the input from a file into a `tibble`, which we'll call `simple_input`. We'll use the `tidyverse` packages throughout this tutorial.[1]

```
if (!require(tidyverse)) {
```

[1]Although we use the `tiydverse` packages here, they are not a requirement for using `maxent.ot`.

```
  install.packages("tidyverse", repos = "http://cran.us.r-project.org")
}
library(tidyverse)
simple_input <- read_csv("data/simple_input_dataframe.csv")
```

`maxent.ot` can also read files in OTSoft format (Hayes et al. 2013) for backwards compatibility with existing research projects. See Appendix A for more detail.

3.4 MaxEnt background: Constraint grammars

Maximum Entropy constraint grammars (MaxEnt; Smolensky 1986; Goldwater & Johnson 2003) are a tool for modeling variation, and are a type of *harmonic grammar* (HG; Legendre et al. 1990; Pater 2009). Like Optimality Theory (OT; Prince & Smolensky 1993/2004), HG models the mapping from input forms to output candidates as a function of the relative strength of a set of constraints. These constraints represent violable restrictions on either properties of the output form (*markedness constraints*) or ways in which the input and output forms differ (*faithfulness constraints*). HG differs from OT in how it represents relative constraint strength: OT uses a rank ordering of the set of constraints from strongest to weakest, while HG assigns numeric weights, with larger values indicating greater strength.

We'll use the following set of simplified constraints to model the data:

(1) *COMPLEX: No complex syllable onsets (Prince & Smolensky 1993/2004).

(2) MAX: Don't delete segments (McCarthy & Prince 1994).

(3) MAX-STRESSED: Don't delete segments from stressed syllables (Beckman 1998).

(4) SSP: Don't violate the Sonority Sequencing Principle (e.g., Cho & King 2003).

Let's take a single observation from our French data: one where /ˈstad/ is realized as [ˈtad], with simplification of the initial cluster. For simplicity, we'll initially consider only two of the four constraints presented above: the markedness constraint *COMPLEX, which drives cluster simplification, and the faithfulness constraint MAX, which penalizes deletion.

Tableaux 3 and 4 provide examples of these constraints in OT and HG models, respectively.

/ˈstad/	*COMPLEX	MAX
a. ˈstad	*!	
☞ b. ˈtad		*

Table 3: Using ranked constraints to model onset cluster simplification.

/ˈstad/	*H*	*COMPLEX $w = 2$	MAX $w = 1$
ˈstad	2	1	0
☞ ˈtad	1	0	1

Table 4: Using weighted constraints to model onset cluster simplification.

In both tableaux, [ˈtad] is the optimal candidate because it undergoes onset cluster simplification. Deleting /s/ incurs a violation of MAX, but prevents a violation of the stronger constraint *COMPLEX. In the OT model in Tableau 3, the relative strength of the two constraints is represented by their rank ordering in the tableau: even though [ˈstad] and [ˈtad] each violate only one constraint, [ˈtad] is preferred because it violates the lower-ranked constraint MAX. In the HG model in Tableau 4, the relative strength of the two constraints is represented by their associated weights. *COMPLEX has a weight of 2, and so its violations are penalized more strongly than those of MAX.

In the OT model, the optimal candidate is chosen by going through the constraints from highest to lowest ranked and eliminating all remaining candidates that aren't the best or tied for being the best according to that constraint, until only a single candidate remains (or there are no more constraints, in which case the remaining candidates tie for being optimal). In the HG model, the optimal candidate is the one with the lowest *harmony score*, which we'll notate as H. H is the sum of the weights of each constraint multiplied by the number of times the candidate violates it. Expressed more precisely, let x be an input form and y be a candidate output form for x. Then the harmony score associated with mapping from x to y is:

(1)

$$H(x,y) = \sum_{i=1}^{K} w_i C_i(x,y),$$

where K is the number of constraints, w_i is the weight of the i^{th} constraint, and $C_i(x,y)$ is the number of times the input-output mapping (x,y) violates the i^{th} constraint.

We can use more compact notation if we represent the weights and constraint violations as vectors. This corresponds more closely to the implementation of weights in `maxent.ot` (see Section 3.7). Let w be the vector of constraint weights and $C(x,y)$ be the vector of constraint violations (or the *violation profile*) of (x,y). Then:

(2)

$$H(x,y) = w \cdot C(x,y),$$

where $\cdot$ is the dot product.

In the example in Tableau 4, $w = \langle 2,1 \rangle$, $C(\text{ˈstad}, \text{ˈtad}) = \langle 0,1 \rangle$ and:

(3)

$$\begin{aligned} H(\text{ˈstad}, \text{ˈtad}) &= w \cdot C(\text{ˈstad}, \text{ˈtad}) \\ &= \langle 2,1 \rangle \cdot \langle 0,1 \rangle \\ &= 2 \times 0 + 1 \times 1 = 1 \end{aligned}$$

HG predicts some quantitative patterns that cannot be represented in OT: for example, HG can encode *constraint cumulativity effects* (e.g., Jäger & Rosenbach 2006; Breiss 2020) where multiple violations of low-ranked constraints can outweigh fewer violations of high-ranked constraints.

3.5 MaxEnt background: How MaxEnt relates harmony to probability

HG models assign harmony scores to input-output pairs. We can consider these scores to be the output of the phonological grammar that evaluates proposed input-output mappings (see, e.g., Daland 2015). In Tableau 4, we selected the optimal candidate by choosing the one with the lowest score.

Although this aligns with the behavior of OT models, it is not the only way to link harmony scores with empirical data. In particular, selecting a single, optimal candidate ignores the *gradience* in harmony scores:

harmony scores allow us not only to identify the best output candidate under some grammar, but also to evaluate the *relative* goodness of each candidate. Consider a mapping like /ˈstad/ → [ˈsta], which deletes a consonant, violating MAX, while retaining the violation of *COMPLEX. Under the constraint weights in Tableau 5, $H(\text{ˈstad}, \text{ˈsta}) = 3$. Thus although [ˈstad] and [ˈsta] are both sub-optimal candidates compared to [ˈtad], [ˈstad] receives a ‘better’ (lower) harmony score than [ˈsta].

/ˈstad/	H	*COMPLEX $w = 2$	MAX $w = 1$
ˈstad	2	1	0
☞ ˈtad	1	0	1
ˈsta	3	1	1

Table 5: A range of H scores.

MaxEnt models implement a specific method for linking gradience in harmony scores to *probability distributions* over candidates. MaxEnt is not the only HG model that relates harmony scores to probabilities (see, e.g., *noisy harmonic grammar*; Boersma & Pater 2016), but it has well-defined learning procedures, is analytically tractable, and lends itself to the application of standard statistical inference procedures (Daland 2015; Flemming 2021). Furthermore, its predictions have been shown to align well with empirical data (Flemming 2021).There are also a number of critiques of MaxEnt as a framework for modeling the phonological grammar (e.g., Anttila et al. 2019; Magri & Anttila 2022; Tilsen 2023).

MaxEnt models define the probability of the mapping (x, y) as follows. Let $\mathcal{Y}(x)$ be the set of candidate outputs for x (the GEN function applied to x). Then the conditional probability of an output y given input x is

(4)
$$P(y|x) = \frac{\exp(-H(x,y))}{\sum_{y' \in \mathcal{Y}(x)} \exp(-H(x,y'))}.$$

The expression on the right side of this equation has a special name: the *softmax function* (Luce 1959; Bridle 1989). This function is commonly used to transform a vector of numbers into a vector representing a categorical probability distribution, such that larger values are associated with higher probabilities.

Using the softmax function in MaxEnt means that a candidate’s probability is proportional to its (negative) harmony score, exponentiated. The best possible harmony score, zero (no violations of any non-zero-weighted constraints), yields the largest possible numerator, $e^0 = 1$. A large harmony score (caused by violations of high-weighted constraints), once negated, yields a smaller numerator.[2] The sum in the denominator, which is the same for every candidate output of a given input, ensures that the probabilities across all candidates sum to 1 within a tableau.

/ˈstad/	H	e^{-H}	P	*COMPLEX $w = 2$	MAX $w = 1$
ˈstad	2	0.14	$0.14 \div (0.14 + 0.37) = 0.27$	1	0
ˈtad	1	0.37	$0.37 \div (0.14 + 0.37) = 0.73$	0	1

Table 6: A tableau with a probability distribution over output candidates.

[2]It is also common for constraint weights or violation counts (but not both) to be represented as negative numbers, which removes the need for negation at this step.

Tableau 6 contains the same weights and violation profiles as Tableau 4, but shows how each candidate's harmony H is exponentiated and then converted into a probability.

3.6 MaxEnt background: Quantifying how well a model fits the data

A common task of the analyst is to find the model that best describes the observed data, which includes choosing which constraints to include and how they should be weighted. In this section, we compare models that have different weights for the same constraints. Section 3.8 will introduce the `maxent.ot` function `optimize_weights`, which automates this process. We delay comparing models that differ in *which* constraints they include to Section 3.9.

Consider the two models in Table 7. In Model 1, *COMPLEX has a weight of 2 and MAX has a weight of 1. In Model 2, the same constraints now have different weights, 5 and 2.

	*COMPLEX	MAX
w_1	2	1
w_2	5	2

Table 7: Constraint weights of two toy models.

MaxEnt models assign predicted probabilities to input/output mappings. For example, Model 1, as shown in Tableau 6, assigns the mapping /ˈstad/ → [ˈstad] a probability of $e^{-2}/(e^{-2}+e^{-1}) = 0.27$, and the mapping /ˈstad/ → [ˈtad] a probability of $e^{-1}/(e^{-2}+e^{-1}) = 0.73$. That is, we predict our hypothetical child to produce [ˈstad] about a quarter of the time, and [ˈtad] about three quarters of the time. Doing the same calculations, we get the probabilities that Model 2 predicts for the input/output mappings. These predicted probabilities are summarized in Table 8.

			Model 1	*Model 2*
/ˈstad/	→	[ˈstad]	.27	.12
/ˈstad/	→	[ˈtad]	.73	.88

Table 8: Input/output probabilities predicted by the two toy models.

Suppose we have collected the data in Table 9, observing [ˈtad] twice and [ˈstad] once. The relative frequency column is included to make it easier to compare the observed data (Table 9) with the predicted probabilities in Table 8.

			Frequency	*Relative frequency*
/ˈstad/	→	[ˈstad]	1	.33
/ˈstad/	→	[ˈtad]	2	.67

Table 9: Toy data from our hypothetical child.

For a small data set like this, it is easy to eyeball the model predictions and conclude that Model 1 fits the observed data better. However, real data sets are typically much larger. It is useful to have a single metric to evaluate how well a model performs on the observed data. Here we introduce the concept of *conditional likelihood*, which is a measure of how well a model fits the observed data. A higher conditional likelihood indicates a better model fit to the observed data. (The term *conditional* is inherited from the conditional probabilities that go into the computation of the conditional likelihood.)

The conditional likelihood of a data set *d* is computed from individual probabilities by taking their product, as below:

(5)

$$L(w,d) = \prod_{i=1}^{N} P(y_i|x_i;w)$$

where N is the number of data points in d and (x_i, y_i) is the ith observed input/output pair. For example, suppose we had observed 1 token of (/ˈstad/, [ˈstad]) and 2 tokens of (/ˈstad/, [ˈtad]). Under the weights in Model 1, $P(\text{ˈstad}|\text{ˈstad}) = 0.27$ and $P(\text{ˈtad}|\text{ˈstad}) = 0.73$. Given the observed data, the conditional likelihood under the weights of Model 1 is:

(6)

$$\begin{aligned}
&L(w_1,\ \{(\text{/ˈstad/, [ˈstad]}), (\text{/ˈstad/, [ˈtad]}), (\text{/ˈstad/, [ˈtad]})\}) \\
&\quad = P([\text{ˈstad}] \mid \text{/ˈstad/}; w) \times P([\text{ˈtad}] \mid \text{/ˈstad/}; w) \times P([\text{ˈtad}] \mid \text{/ˈstad/}; w) \\
&\quad = 0.27 \times 0.73 \times 0.73 \\
&\quad = 0.14,
\end{aligned}$$

and the conditional likelihood under the weights of Model 2 is worse (lower):

(7)

$$\begin{aligned}
&L(w_2,\ \{(\text{/ˈstad/, [ˈstad]}), (\text{/ˈstad/, [ˈtad]}), (\text{/ˈstad/, [ˈtad]})\}) \\
&\quad = P([\text{ˈstad}] \mid \text{/ˈstad/}; w) \times P([\text{ˈtad}] \mid \text{/ˈstad/}; w) \times P([\text{ˈtad}] \mid \text{/ˈstad/}; w) \\
&\quad = 0.12 \times 0.88 \times 0.88 \\
&\quad = 0.09,
\end{aligned}$$

which indicates that Model 1 fits the data better than Model 2 does.

In a more realistic dataset, we will be multiplying many more probabilities—perhaps hundreds, or thousands. Multiplying probabilities together quickly produces very small numbers, and the conditional likelihood of large data sets can become small enough to cause numerical issues on computers. Because of this, it is more common to work with conditional *log* likelihoods instead. It is typical to use the natural log (log base e), but the choice of base is not particularly important. Log probabilities range from $-\infty$ (corresponding to a probability of 0) to 0 (corresponding to a probability of 1). Because multiplying two numbers corresponds to adding their logarithms, the conditional log likelihood of a data set can be calculated as follows:

(8)

$$LL(w,d) = \sum_{i=1}^{N} \ln P(y_i|x_i;w).$$

As with conditional likelihood, a higher conditional log likelihood (i.e., less negative, closer to zero) indicates a better model fit to the observed data.

Consider a subset of the data in Table 1 for just the input /ˈstad/, consisting of 40 tokens of [ˈstad] and 60 tokens of [ˈtad], and assuming the same weights as above. This means that under Model 1, the conditional log likelihood of this subset of the data is:

(9)

$$\begin{aligned} LL(w_1, d) &= 40\ln(P(\text{ ['stad] } | \text{ /'stad/ })) + 60\ln(P(\text{ ['tad] } | \text{ /'stad/ })) \\ &= 40(ln(0.27)) + 60(ln(0.73)) \\ &= 40(-1.31) + 60(-0.31) \\ &= -71.26 \end{aligned}$$

As expected, Model 2 has a worse (lower) conditional log likelihood than Model 1:

(10)

$$\begin{aligned} LL(w_2, d) &= 40\ln(P(\text{ ['stad] } | \text{ /'stad/ })) + 60\ln(P(\text{ ['tad] } | \text{ /'stad/ })) \\ &= 40(ln(0.12)) + 60(ln(0.88)) \\ &= 40(-2.12) + 60(-0.13) \\ &= -92.6, \end{aligned}$$

which indicates that the weights of Model 2 provide a poorer fit to the observed data.

The conditional likelihood of a data set is always maximized by choosing probabilities that exactly match the observed frequencies in the data set. In the context of a MaxEnt model, this means choosing weights that generate such probabilities. For example, consider the same data set as above, but with the probabilities set exactly to the observed frequencies of each output:

(11)

$$\begin{aligned} LL_w(d) &= 40\ln(0.4) + 60\ln(0.6) \\ &= 40(-0.92) + 60(-0.51) \\ &= -67.3 \end{aligned}$$

The conditional log likelihood under these probabilities, -67.3, is the best conditional log likelihood possible for this data set, and is greater than the -71.3 we obtained using the weights in Model 1 or the -92.6 we obtained using the weights in Model 2.

The conditional log likelihood of the data is an important metric that is used for both training a model (Section 3.8) and evaluating the success of trained models that have different constraints (Section 3.9). In training a model—i.e., learning constraint weights—we aim to maximize the conditional log likelihood. Essentially, we are choosing the best model (highest conditional log likelihood) among models that have the same constraints, but different weights.

3.7 Calculating candidate distributions and conditional likelihoods

Our first application of the `maxent.ot` library will be to calculate probability distributions over candidates and the conditional likelihood of a data set given a vector of weights. We'll do this using the `predict_probabilities` function. This function takes a data set and a vector of weights, and returns the probabilities for each input-output pair and the conditional log likelihood of the entire data set.

Assuming you've created the `simple_input` variable as in Section 3.3, you can call `predict_probabilities` as follows:

```
result <- predict_probabilities(simple_input, c(2,1))
result$loglik
# -265.3047
result$predictions
# See Table 10
```

Here we've passed in the data set as the first argument, and a vector of constraint weights, `c(2,1)`, as the second. (The `c` function is R's way of creating a vector.) The function returns an object with two attributes. The attribute `loglik` is the conditional log likelihood of the data set under the provided weights—it is smaller (more negative) than the log likelihoods we previously calculated, because now we are looking at a larger data set consisting of all four words rather than just one. The attribute `predictions` is a data frame; its contents in this case are shown in Table 10.

	Input	Output	Freq	StarComplex	Max	Predicted	Observed	Error
1	ˈstad	ˈstad	40	1	0	0.27	0.40	-0.13
2	ˈstad	ˈtad	60	0	1	0.73	0.60	0.13
3	ˈgʁav	ˈgʁav	60	1	0	0.27	0.60	-0.33
4	ˈgrav	ˈgav	40	0	1	0.73	0.40	0.33
5	spa.gɛ.ˈti	spa.gɛ.ˈti	10	1	0	0.27	0.10	0.17
6	spa.gɛ.ˈti	pa.gɛ.ˈti	90	0	1	0.73	0.90	-0.17
7	gʁy.ˈjo	gʁy.ˈjo	30	1	0	0.27	0.30	-0.03
8	gʁy.ˈjo	gy.ˈjo	70	0	1	0.73	0.70	0.03

Table 10: The output of the `predict_probabilities` function.

The first five columns are the same as the training data, `simple_input`. The `Predicted` column contains the probabilities calculated by the MaxEnt model, and the `Observed` column contains the observed probabilities in the data set. The `Error` column is the difference between the predicted and observed probabilities. This column is helpful to identify what kinds of errors a model makes. In this case, we can see the model systematically over-predicts the application of cluster simplification, which suggests the relative weight of *COMPLEX should be lowered.

To better understand how these grammars work, the reader might find it interesting to try to achieve the highest conditional log likelihood possible by calling `predict_probabilities` with a variety of different weights.

3.8 Learning weights in a MaxEnt grammar

Finding a near-optimal vector of weights by hand can be tedious, but fortunately, the process can be automated. Given a data set d, which for us consists of counts of input-output pairs and their corresponding violation profiles, it is always possible to find the constraint weights that maximize the conditional log likelihood of the data.[3]

The goal is to find weights w that maximize the objective function $LL_w(d)$:

[3]It may be the case that there is more than one set of weights that correspond to the same optimal distribution.

(12)

$$LL_w(d) = \sum_{i=1}^{N} \ln P(y_i|x_i; w)$$

That is, we want to find the weights w such that the conditional log likelihood of the data is as high as possible. The properties of MaxEnt models are such that it's always possible to find weights that maximize this value (or come arbitrarily close to doing so). For reasons of space, we refrain from going into detail about this process here, but there are a number of references that describe the process in more detail (Della Pietra et al. 1997; Goldwater & Johnson 2003; Hayes & Wilson 2008). `maxent.ot` uses the L-BFGS optimizer implemented by the R statistics library's `optim` function. Following standard practice, we assume that weights are required to be non-negative. This corresponds to a theoretical assumption that violating constraints should only ever penalize candidates, not reward them.[4]

The function in `maxent.ot` to find the optimal weights for a data set is `optimize_weights`. The simplest application of this function takes a single data frame or tibble as an argument, in the same format as `predict_probabilities`.

```
simple_model <- optimize_weights(simple_input)
simple_model$weights
# StarComplex  Max
# 0.6190392    0.0000000
simple_model$loglik
# -258.9787
simple_model$k
# 2
simple_model$n
# 400
simple_model$bias_params
# NA
```

The object returned by `optimize_weights` contains five attributes:

1. `weights`: A named vector containing the optimal constraint weights (here, 0.62 for *COMPLEX and 0.00 for MAX).
2. `loglik`: The conditional log likelihood of the data set under these weights.
3. `k`: The number of constraints.
4. `n`: The number of data points.
5. `bias_params`: This will be discussed in Section 3.10.

The `k` and `n` parameters will become relevant in Section 3.9 when we turn to model comparison.

The data set and the weights calculated by `optimize_weights` can be used as input for `predict_probabilities` to get the predicted probabilities for individual input-output pairs.

[4]There is work that has proposed the use of constraints that reward violations (e.g. Kimper 2011; Kaplan 2018). To allow constraint weights to be negative, the `optimize_weights` and `cross_validate` functions can take the optional argument `allow_negative_weights = TRUE`. (By default, this argument is set to `FALSE`.)

```
result <- predict_probabilities(simple_input, simple_model$weights)
result$loglik
# -258.9787
result$predictions
# See Table 11
```

	UR	SR	Freq	StarComplex	Max	Predicted	Observed	Error
1:	stad	stad	40	1	0	0.35	0.4	-0.05000001
2:	stad	tad	60	0	1	0.65	0.6	0.05000001
3:	grav	grav	60	1	0	0.35	0.6	-0.25000001
4:	grav	gav	40	0	1	0.65	0.4	0.25000001
5:	spagɛti	spagɛti	10	1	0	0.35	0.1	0.24999999
6:	spagɛti	pagɛti	90	0	1	0.65	0.9	-0.24999999
7:	gryjo	gryjo	30	1	0	0.35	0.3	0.04999999
8:	gryjo	gyjo	70	0	1	0.65	0.7	-0.04999999

Table 11: Output of the `predict_probabilities` function for the model fitted to the data.

You can save the output of `optimize_weights` to a file by using `write_csv` or a similar function on the `predictions` attribute.

3.8.1 *Testing against new data*

Given a model that has been fitted by `optimize_weights`, we can see how it generalizes to new data using `predict_probabilities`. For example, we might wish to train a model of English irregular past tense forms on real English verbs—(/hajd/, [hɪd]), (/fajnd/, [fawnd]), etc.—and then test that model against some novel verbs—such as /splɪŋ/—and compare the model's performance on the novel verbs to real humans' performance on a wug test (Berko 1958; Albright & Hayes 2003).

We will stick with our toy example of acquiring French, and test the model just fitted against the hypothetical data shown in Table 12, representing how many times a child would produce hypothetical /ˈkʁob/ faithfully or with onset simplification (the corresponding file can be found at `data/simple_wug.csv` in the tutorial materials).

Input	Output	Frequency	StarComplex	Max
ˈkʁob	ˈkʁob	4	1	
ˈkʁob	ˈkob	3		1

Table 12: Wug data to test fitted grammar against.

To test the grammar against these new data, we simply use the `predict_probabilities` function again, specifying the wug data now as input:

```
wug_input <- read_csv("data/simple_wug.csv")
result_wug <- predict_probabilities(wug_input, simple_model$weights)
result_wug$loglik
# -5.491637
```

```
result_wug$predictions
# See Table 13
```

	Input	Output	Freq	StarComplex	Max	Predicted	Observed	Error
1	ˈkʁob	ˈkʁob	4	1	0	0.35	0.5714286	-0.2214286
2	ˈkʁob	ˈkob	3	0	1	0.65	0.4285714	0.2214286

Table 13: Output of `predict_probabilities` for simple wug data.

We can see that the grammar trained on the original data doesn't fit the wug data very well, mainly because there is already a word in the training data, /ˈgʁav/, that has the very same violation profile as /ˈkʁob/, but a higher rate of onset cluster simplification (40%). Because the model only saw /ˈgʁav/ as the weights were being selected, it incorrectly predicts a similar simplification rate for /ˈkʁob/.

Besides modeling a human's encounter with new items, being able to test the grammar's predictions on unseen items is also useful for model-fitting itself, as we'll see in Section 4.

3.9 Comparing how well different models fit the data

One of the analyst's main tasks is to compare different models of the same data, and perhaps to select one model as preferred. Often, a phonologist wants to determine whether the data justify including a certain constraint in the grammar (e.g., Hayes et al. 2012; Shih 2017).

We first consider adding the two additional constraints listed earlier, MAXSTRESSED and SSP. The faithfulness constraint MAXSTRESSED penalizes deleting from a stressed syllable, as in /ˈstad/ → [ˈtad] and /ˈgʁav/ → [ˈgav]. This was a real effect noted by Rose (2002).

The markedness constraint SSP, short for SONORITYSEQUENCINGPRINCIPLE (e.g., Cho & King 2003), penalizes onsets that decrease in sonority, such as [st, sp, sk]; this constraint therefore gives an extra reason to simplify those onsets.

The new input data are shown in Table 14.

Input	Output	Frequency	StarComplex	Max	MaxStressed	SSP
ˈstad	ˈstad	40	1			1
ˈstad	ˈtad	60		1	1	
ˈgʁav	ˈgʁav	60	1			
ˈgʁav	ˈgav	40		1	1	
spa.gɛ.ˈti	spa.gɛ.ˈti	10	1			1
spa.gɛ.ˈti	pa.gɛ.ˈti	90		1		
gʁy.ˈjo	gʁy.ˈjo	30	1			
gʁy.ˈjo	gy.ˈjo	70		1		

Table 14: Input data, now with two more constraints.

We could create another `.csv` input file with the two new columns, but instead here we use R code to add two new columns to the existing `tibble`, and while we're at it we also create an input `tibble` that adds only MAXSTRESSED and one that adds only SSP:

```
# copy the original data and add new constraint violations
max_stressed_ssp_input <- simple_input %>%
  mutate(MaxStressed = c(0,1,0,1,0,0,0,0),
         SSP = c(1,0,0,0,1,0,0,0))

# Remove SSP column for model without SSP
max_stressed_input <- max_stressed_ssp_input %>%
  select(-SSP)

# Remove MaxStressed column for model without MaxStressed
ssp_input <- max_stressed_ssp_input %>%
  select(-MaxStressed)
```

Now, we can fit constraint weights to each of the three new scenarios:

```
model_max_stressed <- optimize_weights(max_stressed_input)
model_ssp <- optimize_weights(ssp_input)
model_max_stressed_ssp <- optimize_weights(max_stressed_ssp_input)
```

We now have four models: `simple_model` with just two constraints, `model_max_stressed_ssp` with all four constraints, and `model_max_stressed` and `model_ssp` each with three constraints.

Using the `predict_probabilities` function, we can derive the predictions of each model for the same data. The code below also collates the predictions into a single table for easier comparison.

```
# Apply predict_probability to each model
result_basic <- predict_probabilities(simple_input, simple_model$weights)

result_max_stressed <- predict_probabilities(
  max_stressed_input, model_max_stressed$weights
)

result_ssp <- predict_probabilities(ssp_input, model_ssp$weights)

result_max_stressed_ssp <- predict_probabilities(
  max_stressed_ssp_input, model_max_stressed_ssp$weights
)

# Consolidate model predictions into single tibble
# Round and format for readability
results_compiled <- result_max_stressed_ssp$predictions %>%
  select(-Error, -Predicted) %>%
  mutate(`Pred. basic` = result_basic$predictions$Predicted,
         `Pred. w/ MaxStr` = result_max_stressed$predictions$Predicted,
         `Pred. w/ SSP` = result_ssp$predictions$Predicted,
         `Pred. w/ both` = result_max_stressed_ssp$predictions$Predicted) %>%
  mutate_at(vars(starts_with("Pred")), list(~ round(., 2)))

results_compiled
# See Table 15, which is slightly modified from the R output for better display
```

Looking at the predictions of each model in Table 15, it is clear that the model with both of the new constraints (“Pred. w/ both” column) achieves the closest fit to the data. The stress and sonority effects are both well captured, and the predicted probabilities are close to the observed probabilities that the model was trained on (“Obs.” column). But, does the full model’s improved fit justify its greater complexity? The `compare_models` function answers this question, under various measures. The function takes as its arguments the models that are being compared and the desired method of comparison. We will briefly describe the supported methods below.

UR	SR	*Compl	Max	MaxStr	SSP	Obs.	Pred. basic	Pred. w/ MaxStr	Pred. w/ SSP	Pred. w/ both
ˈstad	ˈstad	1	0	0	1	0.40	0.35	0.50	0.25	0.38
ˈstad	ˈtad	0	1	1	0	0.60	0.65	0.50	0.75	0.62
ˈgʁav	ˈgʁav	1	0	0	0	0.60	0.35	0.50	0.45	0.62
ˈgʁav	ˈgav	0	1	1	0	0.40	0.65	0.50	0.55	0.38
spagɛˈti	spagɛˈti	1	0	0	1	0.10	0.35	0.20	0.25	0.12
spagɛˈti	pagɛˈti	0	1	0	0	0.90	0.65	0.80	0.75	0.88
gʁyˈjo	gʁyˈjo	1	0	0	0	0.30	0.35	0.20	0.45	0.28
gʁyˈjo	gyˈjo	0	1	0	0	0.70	0.65	0.80	0.55	0.72

Table 15: Predictions of four models.

3.9.1 BIC

The BIC, or Bayesian Information Criterion (Schwarz 1978), combines a measure of how well the model fits the data and a measure of how complex the model is. The smaller this number, the better the model. The measure of model fit is $-2LL_w(d)$, where $LL_w(d)$ is the log likelihood of the data d given the weights w, as defined and explained in Section 3.6. The closer to zero this quantity is, the better the model fit. The measure of model complexity is $k \ln(N)$, where k is the number of constraints (in the case of a constraint grammar—more generally, it is the degrees of freedom in the model) and N is the number of data points, which in our running example is 400. Thus, the BIC becomes larger (worse) when there are more constraints, and the penalty per constraint is greater the more data has been observed; in other words, the more data there is, the more a constraint needs to explain in order to earn its place in the grammar. The full expression is shown in Equation 13.

(13)
$$BIC = -2LL_w(d) + k\ln(N)$$

Let us calculate the BIC for each of the grammars we have fit so far. We can do this by hand, as we demonstrate using the basic, two-constraint model below: we simply retrieve $LL_w(d)$, k, and N, and plug them into Equation 13, yielding.

```
-2 * simple_model$loglik + simple_model$k * log(simple_model$n)
# 529.9402
```

But we can also use the function `compare_models` to automatically obtain the BIC for two or more models. The function takes as its arguments two or more objects in the format returned by `optimize_weights`, plus the method of comparison to use. In this case, we specify `method = "bic"` (to prevent numbers from displaying in scientific notation, we set R’s global option `scipen` to a high value to turn scientific notation off, and `digits` to 5 to limit the number of decimal places shown. We later return `digits` to its default value of 7.)

```
options(scipen = 999)
options(digits = 5)
compare_models(
    simple_model, model_max_stressed, model_ssp, model_max_stressed_ssp,
    method = "bic"
)
options(digits = 7)
# See Table 16
```

model	k	n	bic	bic.delta	bic.wt	cu.wt	ll
model_max_stressed_ssp	4	400	481.59	0.000	0.998996393522926	0.999	-228.81
model_max_stressed	3	400	495.39	13.806	0.001003595038440	1.000	-238.71
model_ssp	3	400	518.16	36.576	0.000000011407013	1.000	-250.09
simple_model	2	400	529.94	48.352	0.000000000031622	1.000	-258.98

Table 16: BIC values for four models.

We can see in Table 16 that, as we would always expect, adding more constraints brings the log likelihood (the column `ll`) closer to zero. We can also see that even though the model with both MAXSTRESSED and SSP is the most complex ($k = 4$), the added complexity is worth it, because this model has the lowest BIC (481.5879). The `bic.delta` column shows how much worse each model is than the best model—you can verify that each model's `bic.delta` value is obtained by subtracting 481.5879 from that model's `bic` value. Raftery (1995) proposes that a difference of 0 to 2 be taken as weak evidence for the "better" model over the "worse" model; a difference of 2 to 6 as positive evidence; a difference of 6 to 10 as strong evidence, and over 10 as very strong evidence. In our case, there is, in Raftery's terms, very strong evidence that the full model is better than any of the three smaller models.[5]

But of course, more complex is not always better. For comparison, we now introduce an even more complex model, with the constraint DOTHERIGHTTHING. This constraint is simply defined to penalize, in each tableau, the candidate that the four-constraint model was (slightly) over-predicting. For example, the mapping (/ˈstad/, [ˈtad]) incurs a violation of DOTHERIGHTTHING because the observed probability is 0.60 but the four-constraint model predicts 0.62.

[5]The `bic.wt` column reports the "Schwartz weights", which are each model's BIC score divided by the sum of the BIC of all models. The `cu.wt` column reports the "cumulative Schwarz weights", which are just the cumulative sum of the Schwarz weights starting from the model with the smallest BIC, and moving down the list to larger-BIC models. For information on how to interpret these quantities, see, e.g., Burnham & Anderson (2004); Wagenmakers & Farrell (2004).

```
# Make the new data tableau
# Copy the four-constraint data
# Add DoTheRightThing column
dtrt_input <- max_stressed_ssp_input %>%
  mutate(DoTheRightThing = c(0,1,1,0,1,0,0,1))

# Fit constraint weights
model_dtrt <- optimize_weights(dtrt_input)

# Retrieve model predictions
result_dtrt <- predict_probabilities(dtrt_input, model_dtrt$weights)

# Get log likelihood
result_dtrt$loglik
# -228.1971
```

To save space, we don't show here `result_dtrt`, the table of the new model's predictions, but they are essentially identical to the training data, with the biggest error value being 0.0000047. This is reflected in the log likelihood of the training data under this model, -228.1971—identical to the best possible log likelihood that any model could obtain for this data set, calculated below using the expression we introduced above in Equation 8.

```
sum(simple_input$Frequency * log(simple_input$Frequency/100))
#-228.1971
```

But, `compare_models` shows that the improved fit does not justify the increase from four constraints to five, and the five-constraint model's BIC is bigger (worse) than the four-constraint model's.

```
options(digits = 5)
compare_models(model_max_stressed_ssp, model_dtrt, method = "bic")
#See Table 17
```

In Table 17, the `bic.delta` value indicates positive evidence favoring the four-constraint model over the four-constraint model plus DoTheRightThing.

model	k	n	bic	bic.delta	bic.wt	cu.wt	ll
model_max_stressed_ssp	4	400	481.59	0.0000	0.915426	0.91543	-228.81
model_dtrt	5	400	486.35	4.7635	0.084574	1.00000	-228.20

Table 17: BIC values for four-constraint model vs. five-constraint model.

Thus, out of all the models we considered, BIC finds the model with *Complex, Max, MaxStressed, and SSP to be the sweet spot between under-fitting, with too few constraints, and over-fitting, with too many (see Section 4 for more on under- and over-fitting).

BIC is not the only measure that `maxent.ot` implements. The remainder of this subsection introduces the other measures available and shows how the various models perform on them.

3.9.2 AIC and AICc

The AIC, or Akaike Information Criterion (Akaike 1974), is another measure that balances model fit against model complexity. The difference is that the penalty for model parameters does not make reference to the number of data observations:

(14)

$$AIC = -2LL_w(d) + 2k$$

Although the equations for these metrics are superficially similar, they are derived from different starting assumptions. It is beyond the scope of this article to examine the differences between BIC and AIC—both theoretical, when n is infinite, and practical, with real data sets. For some comparisons, see, e.g., Burnham & Anderson (2004); Wagenmakers & Farrell (2004); Vrieze (2012). In our toy example, when `compare_models` is told to use AIC rather than BIC, it again chooses the four-constraint model as the best trade-off between fit and complexity, although the over-fit, five-constraint model is close behind.

```
compare_models(
  simple_model, model_max_stressed, model_ssp, model_max_stressed_ssp,
  model_dtrt, method = "aic"
)
# See Table 18
```

model	k	aic	aic.delta	aic.wt	cum.wt	ll
model_max_stressed_ssp	4	465.62	0.00000	0.59527945074509503	0.59528	-228.81
model_dtrt	5	466.39	0.77207	0.40463926902930786	0.99992	-228.20
model_max_stressed	3	483.42	17.79779	0.00008127930141631	1.00000	-238.71
model_ssp	3	506.19	40.56753	0.00000000092383280	1.00000	-250.09
simple_model	2	521.96	56.33526	0.00000000000034808	1.00000	-258.98

Table 18: AIC values for five models.

The AICc, or AIC-corrected, is a variant of the AIC designed to avoid overfitting when the amount of observed data is small. The correction is performed by adding a term that further penalizes a model for its number of parameters k, but less so as the number of observations, n, exceeds the number of parameters k:

(15)

$$AICc = AIC + \frac{2k^2 + 2k}{n - k - 1}$$

Once again, even with this correction, the four-constraint model is rated best:

```
compare_models(
  simple_model, model_max_stressed, model_ssp, model_max_stressed_ssp,
  model_dtrt, method = "aic_c"
)
# See Table 19
```

model	k	n	aicc	aicc.delta	aicc.wt	cum.wt	ll
model_max_stressed_ssp	4	400	465.72	0.00000	0.60140768797465816	0.60141	-228.81
model_dtrt	5	400	466.55	0.82309	0.39850850852858155	0.99992	-228.20
model_max_stressed	3	400	483.48	17.75713	0.00008380254388359	1.00000	-238.71
model_ssp	3	400	506.25	40.52687	0.00000000095251235	1.00000	-250.09
simple_model	2	400	521.99	56.26422	0.00000000000036438	1.00000	-258.98

Table 19: AICc values for five models.

3.9.3 *Likelihood ratio test*

`compare_models` implements one more measure, the likelihood ratio test. This measure is more restricted in its application: it can only be used to compare pairs of models, and it requires that the models be *nested*: in this case, that one model's set of constraints be a subset of the other's. For example, we can compare the simple two-constraint model to the three-constraint model that adds SSP, but this measure does *not* allow us to compare the two three-constraint models. To perform the likelihood ratio test, the code first calculates the likelihood ratio, which is defined as follows, where LL_2 is the conditional log likelihood, which we wrote above as $LL_w(d)$, for the model with more parameters, and LL_1 is the conditional log likelihood for the model with fewer parameters:

(16)
$$LR = 2(LL_2 - LL_1)$$

To calculate a p-value, the code then performs a chi-squared test, with $\chi^2 = LR$ and k, the degrees of freedom, set to the difference in number of constraints between the two models. We illustrate here just one comparison, between the four-constraint model and the five-constraint model. The `chi_sq` column shows LR, the likelihood ratio between the two models, the `k_delta` column shows that the difference in number of constraints between the two models is 1 ($5 - 4$), and the `p_value` column shows that the additional constraint in the more-complex model (DoTheRightThing) does not significantly increase model fit.

```
compare_models(model_max_stressed_ssp, model_dtrt, method = "lrt")
options(digits = 7)
# See Table 20
```

description	chi_sq	k_delta	p_value
model_dtrt ∼ model_max_stressed_ssp	1.2279	1	0.26781

Table 20: Comparison of two models.

3.10 Using prior terms to encode bias or avoid over-fitting

So far, we have been simply fitting model weights as closely as possible to the data. But, in both phonology and machine learning, it is common to build in a *bias* of some kind to improve generalization to new data. In its simplest form, this is a bias against *over-fitting*: by penalizing weights for diverging from a default of zero, we require each weight in the model to be strongly justified by the data. As we'll see below, we can also use more content-ful biases, such as a bias for a certain constraint to have a large weight.

3.10.1 MaxEnt background: The Gaussian prior

In MaxEnt, these biases are most often implemented as a *Gaussian prior* (Chen & Rosenfeld 1999; Goldwater & Johnson 2003). Rather than finding the weights that simply maximize the predicted probability of the data (maximize $LL_w(d)$, as above in Equation 12), we find weights that maximize the expression in Equation 17: the predicted probability of the data minus a penalty for deviations from a prior distribution over the weight values. This penalty is known variously as a *prior*, *regularization term*, *smoothing term*, or *bias term*. In the case of a Gaussian prior, it has two parameters: μ, a vector of means for the prior on each constraint weight, and σ, a vector of standard deviations for each prior. All else being equal, smaller values of σ will penalize deviations from μ more severely than larger ones.

As shown in Equation 17, for each of the M constraints we square the difference between the j^{th} constraint's weight w_j and the mean weight that we have specified for that constraint's prior, μ_j, and divide by the square of the corresponding standard deviation that we have specified for that constraint, σ_j^2. The penalty term is thus the log of a Gaussian distribution, ignoring a constant.

(17)

$$LL_w(d) - \sum_{j=1}^{M} \frac{(w_j - \mu_j)^2}{2\sigma_j^2}$$

A common agnostic default sets μ to 0 for every constraint and σ to the same value for every constraint (see Section 4 for selecting good values of σ). Under this parameterization, the larger the constraint weight in the model, the greater the penalty. The term σ modulates the severity of the penalty. If a constraint C_j has a large σ_j, then constraint C_j is tolerant of departures from its default μ_j—not much data is required to justify such a departure. If σ_j is small, then the penalty for departure is greater.

Penalty terms like this are often used in machine learning to avoid over-fitting. Intuitively, if the model is fit too closely to the training data, it will tend to encode accidental peculiarities of the training data, and therefore perform worse on new data (see Section 4 for an extended example).

The agnostic default, with $\mu_j = 0$ for all constraints C_j, has also been used to explain linguistic phenomena. Martin (2007, 2011) exploits the nature of a Gaussian prior: because the penalty for increasing a constraint weight grows with the square of the weight, it is preferable—in terms of maximizing Equation 17—to spread the responsibility for a data pattern out over several moderately-weighted constraints, rather than piling it up onto one high-weighted constraint. Martin shows that this tendency to spread out weight over multiple constraints can account for how, over generations, a phonotactic restriction that holds within morphemes, such as the English ban on geminate consonants, can "leak" onto affixed words and compounds, so that words like *give**n-n**ess* and *boo**k-c**ase* are fewer than would be expected.

Because of the flexibility in assigning constraint-specific μ and σ values, it is also possible to build in a default that is phonetically substantive. Wilson (2006), for example, derives constraint-specific σ values from auditory confusion matrices, to account for typological and experimental biases in velar palatalization. White (2013) also starts from confusion data, but derives constraint-specific μ values, to account for the typological and experimental bias against "saltation" patterns.

The Gaussian prior is not the only one possible, although it is the only one currently supported by `maxent.ot`. For example, rather than penalizing the square of a constraint weight's deviation from its default ("L2 regularization"), we can penalize the absolute values of the weight deviations ("L1 regularization"), or even the square roots of the weight deviations, which will have the effect that responsibility for a data pattern is piled up onto a single constraint as much as possible (for discussion of different regularization techniques in MaxEnt, see Hughto et al. 2019). Our code is open-source, and users comfortable with R should find it feasible to alter the `optimize_weights` function to use their preferred prior.

3.10.2 Prior terms in `maxent.ot`

The `optimize_weights` function in `maxent.ot` has optional arguments for implementing a Gaussian prior. To use the same value for all constraints, pass in a scalar to the arguments `mu` and `sigma`.[6]

```
# all constraints have mu of 0, sigma of 1
model_max_stressed_ssp_bias1 <- optimize_weights(
  max_stressed_ssp_input, mu = 0, sigma = 1
)

rbind(
  unbiased_weights = model_max_stressed_ssp$weights,
  biased_weights = model_max_stressed_ssp_bias1$weights
)
# See Table 21
```

The code above also creates a table to compare the weights in our previous, maximally fitted model to those in the new model that was fitted with bias to keep all weights small, shown in Table 21. All of the weights get slightly smaller.

	StarComplex	Max	MaxStressed	SSP
unbiased_weights	1.0377363	0.07827311	1.461826	1.0047257
biased_weights	0.9033141	0.00000000	1.355061	0.9608867

Table 21: Weights resulting from unconstrained fitting vs. fitting with a Gaussian bias for small weights.

We can see why, with the bias in place, these smaller weights were preferred, by calculating the two models' log likelihoods and values for the prior:

```
# log likelihoods
model_max_stressed_ssp$loglik
# -228.811
model_max_stressed_ssp_bias1$loglik
# -228.9289

# priors
sum(((model_max_stressed_ssp$weights - 0)^2)/(2 * 1^2))
# 2.114716
sum(((model_max_stressed_ssp_bias1$weights - 0)^2)/(2 * 1^2))
# 1.787735

# values of objective function
model_max_stressed_ssp$loglik - sum(
        ((model_max_stressed_ssp$weights - 0)^2)/(2 * 1^2)
```

[6]Note that in MaxEnt Grammar Tool (Hayes et al. 2009) the value "sigma" that the program takes as input is actually the full denominator of the bias term, $2\sigma^2$, whereas in `maxent.ot` it is just σ. If you are trying to replicate an analysis done in the MaxEnt Grammar Tool using `maxent.ot`, you will need to reduce your values of σ accordingly.

```
)
# -230.9257

model_max_stressed_ssp_bias1$loglik - sum(
        ((model_max_stressed_ssp_bias1$weights - 0)^2)/(2 * 1^2)
)
# -230.7166
```

The fit of the new model is slightly less good, with a conditional log likelihood of -228.93 instead of -228.81. But it compensates by having a smaller value for the Gaussian prior, 1.78 instead of 2.11. Once the priors are subtracted from the log likelihoods, the biased model comes out slightly ahead, at -230.72 instead of -230.93.

Instead of single values of μ and σ, it is also possible to pass in a vector listing the desired μ and σ for each constraint. For example, we could put a small σ on the MAXSTRESSED constraint, indicating that more evidence is required in order for it to depart from its default weights. We do this by setting the argument `sigma` to the vector `c(1, 1, 0.1, 1)`, reflecting the order of the constraints in the `max_stressed_ssp_input` tibble, where MAXSTRESSED is the third constraint. In this example we will also put non-zero default weights on the two markedness constraints, using the `mu` argument.

```
model_max_stressed_ssp_bias2 <- optimize_weights(
  max_stressed_ssp_input, mu = c(2, 0, 0, 1), sigma = c(1, 1, 0.1, 1)
)

rbind(
  unbiased_weights = model_max_stressed_ssp$weights,
  biased_weights = model_max_stressed_ssp_bias1$weights,
  individually_biased_weights = model_max_stressed_ssp_bias2$weights
)
# See Table 22
```

We can see in Table 22 that MAXSTRESSED now gets a much smaller weight—and MAX's weight increases to partially compensate. The two markedness constraints are not much changed, but we can see that *COMPLEX, which had the highest default weight, $\mu_{*\text{COMPLEX}} = 2$, does have an even higher weight than it did in the maximally-fitted model (`unbiased_weights`). This newest model does a poor job of capturing the stress effect, reflecting the bias we built in against the constraint that encodes that effect (MAXSTRESSED).

	StarComplex	Max	MaxStressed	SSP
unbiased_weights	1.0377363	0.07827311	1.461826	1.0047257
biased_weights	0.9033141	0.00000000	1.355061	0.9608867
individually_biased_weights	1.1705216	0.82951198	0.2499482	0.8893151

Table 22: Weights resulting from maximally fitted vs. fitting with bias for small weights.

Finally, these values can be read from a file, data frame, or tibble, using the `bias_input` argument. In this example, we first create a data frame, using the very same μ and σ values as before, then use that data frame as the `bias_input` argument in `optimize_weights`. The resulting weights are the same, as expected:

```
bias_table <- tibble(
  Constraint = c("StarComplex", "Max", "MaxStressed", "SSP"),
  Mu = c(2, 0, 0, 1),
  Sigma = c(1, 1, 0.1, 1)
)

model_max_stressed_ssp_bias3 <- optimize_weights(
  max_stressed_ssp_input, bias_input = bias_table
)

model_max_stressed_ssp_bias3$weights
# StarComplex Max MaxStressed SSP
# 1.1705216 0.8295120 0.2499482 0.8893151
```

4 Cross-validation

Machine learning seeks to find a balance between the extremes of under-fitted models, which do not learn adequately from the data, and over-fitted models, which learn too well from the data, in the sense that they learn accidental details, and accordingly fail to generalize effectively. We will introduce a new and more realistic example to illustrate these extremes, and to illustrate *cross-validation*, which provides a process for finding the best compromise between the two extremes.

4.1 Example from the phonology of Shona

The example we will use comes from Shona, which is the majority language of Zimbabwe, and also spoken in neighboring countries. The phonology of Shona requires every consonant to be followed by a vowel (or sometimes [w]), leading to extensive vowel insertion in loanwords, such as [tim**u**] from English [tim] 'team'. Using various published and unpublished sources, Uffmann (2007) compiles a corpus of 1709 loans in Shona with inserted vowels, and uses cluster analysis to identify the factors that influence which of Shona's five vowels [i, e, a, o, u], is inserted. For word-final inserted vowels, as shown in Table 23, Uffmann finds that both the preceding consonant and the vowel before matter. Labials tend to be followed by [u], as in [tim**u**][7] 'team'; coronals and dorsals tend to be followed by [i], as in [ejit**i**] 'eight'. If the preceding consonant is a liquid, there is a strong tendency to copy the preceding vowel ([hor**o**] 'hall'). The tendency for vowel copying is also seen, more weakly, for non-liquids; for example, [o] is the most likely vowel to be inserted following [o] and a dorsal consonant ([kok**o**] 'cork').[8]

[7]The Uffmann (2007) transcriptions do not indicate tone.

[8]For expository simplicity, we ignore two further distinctions that Uffmann's modeling identified. For example, a sequence of i + labial obstruent is usually followed by inserted [i], as in [tʃipi] 'sale', from English [tʃip] 'cheap', but a sequence of i + labial *sonorant* is usually followed by inserted [u], as in [timu]. Similarly, consonant manner matters for sequences of u + coronal.

		Number inserted							
Preceding V	Preceding C	i	u	e	o	a	total	example	gloss
i	labial	40	13	0	4	4	61	tim**u**	‘team’
e,a,o,u	labial	17	134	1	14	14	180	tʃitof**u**	‘stove’
u	coronal	52	25	0	0	0	77	b̤uʃ**i**	‘bush’
i,e,a,o	coronal	895	2	25	8	27	957	ejit**i**	‘eight’
i,e	dorsal	92	0	6	2	8	108	hwik**i**	‘wick’
a	dorsal	30	2	0	0	7	39	mag̤**i**	‘mug’
o	dorsal	3	4	0	23	1	31	kok**o**	‘cork’
u	dorsal	1	7	0	0	1	9	b̤uuk**u**	‘book’
i	liquid	22	2	0	5	6	35	v̤ir**i**	‘wheel’
e	liquid	15	0	12	19	22	68	v̤er**i**	‘veil’
a	liquid	21	8	4	0	8	41	minarar**i**	‘mineral’
o	liquid	1	0	0	44	4	49	hor**o**	‘hall’
u	liquid	1	29	1	21	4	56	fur**u**	‘fool’

Table 23: Word-final vowel insertion rates in Shona loanwords (Uffmann 2007: pp. 52–57).

Uffmann finds that similar factors are at work in word-internal inserted vowels, as shown in Table 24. If the following consonant is an obstruent or nasal, then [i] is always inserted ([s**i**peja] ‘spare’), but if the following consonant is a liquid then both the preceding consonant’s place and the following vowel’s quality play roles.[9]

			Number inserted							
preceding C	following C	foll. V	i	u	e	o	a	total	example	gloss
anything	obstruent or nasal	anything	129	0	0	0	0	129	s**i**peja	‘spare’
labial	liquid	i	20	13	0	0	1	34	f**i**ridʒ̤i	‘fridge’
labial	liquid	o	0	12	0	6	0	18	p**o**rofiti ∼ p**u**rofiti	‘profit’
labial	liquid	e,a,u	1	86	0	0	0	87	pureʃa	‘pressure’
coronal	liquid	anything	43	11	1	1	1	57	d̤**i**riŋi	‘drink’
dorsal	liquid	i,e,a	51	0	0	0	1	52	g̤**i**rini	‘green’
dorsal	liquid	o	9	0	0	6	0	15	g̤**i**rovu ∼ g̤**o**rovu	‘glove’
dorsal	liquid	u	0	3	0	0	0	3	g̤**u**ruu	‘glue’

Table 24: Word-internal vowel insertion rates in Shona loanwords (Uffmann 2007: pp. 70–72).

For this illustration, we consider just five candidates for each word type—one for each possible inserted vowel—and a set of 55 DEP constraints (McCarthy & Prince 1994), of varying degrees of specificity. The constraints are listed in Table 25. For each vowel X, there is a simple DEP-X constraint against inserting it; a DEP-X/C__T constraint against inserting it between any consonant and an obstruent; four DEP-X/W__ constraints against inserting it after each consonant type W; and five DEP-X/NEAR Y constraints against inserting it when the nearest vowel is Y, where “nearest” means preceding vowel for word-final inserted vowels, and following vowel for word-internal inserted vowels.

[9] We again ignore some complexities in Uffmann’s findings, by omitting the 18 words that show insertion between a consonant and a glide.

Basic DEP-X:	DEP-I, DEP-U, DEP-E, DEP-O, DEP-A
DEP-X/C_T:	DEP-I/C_T, DEP-U/C_T, DEP-E/C_T, DEP-O/C_T, DEP-A/C_T
DEP-X/W_:	DEP-I/LAB_, DEP-I/COR_, DEP-I/DORS_, DEP-I/LIQ_, DEP-U/LAB_, etc. (20 constraints)
DEP-X/NEAR Y:	DEP-I/NEAR I, DEP-I/NEAR U, DEP-I/NEAR E, DEP-I/NEAR O, DEP-I/NEAR A, DEP-U/NEAR I, etc. (25 constraints)

Table 25: Constraints for Shona epenthesis.

Table 26 shows the upper left corner of the input file, `Shona_tableaux.csv`. There are 21 tableaux and 55 constraint columns in the full file. Each tableau is labeled with one word input, such as /tim/, for concreteness, but represents the set of all words that give rise to the same set of violation profiles. Thus, /tim/ represents all 50 words that end with a labial consonant preceded by /i/.

Input	Output	Frequency	Dep-i	Dep-u	Dep-e	Dep-o	Dep-a	Dep-i/i	Dep-u/i	...
tim	timi	31	1					1		...
tim	timu	11		1					1	...
tim	time	0			1					...
tim	timo	4				1				...
tim	tima	4					1			...
tʃitof	tʃitofi	14	1							...
tʃitof	tʃitofu	106		1						...
tʃitof	tʃitofe	1			1					...
tʃitof	tʃitofo	13				1				...
tʃitof	tʃitofa	8					1			...
...										

Table 26: Constraints for Shona epenthesis.

The R code below reads in the data, and adds an extra column with the number of words each tableau represents, for use in plots later on.

```r
shona_input <- read_csv("data/Shona_tableaux.csv")
# display top left corner of tableaux
shona_input[1:10,1:10]

# make a version with a column for total frequency of each tableau
# (will be useful for plots below)
# This works because every tableau in this data set has a unique input
shona_input_with_total <- shona_input %>%
    group_by(Input) %>%
    mutate(tableau_freq = sum(Frequency))
```

With the data set in place, let us now turn to the question of over- versus under-fitting.

4.2 Using cross-validation to explore different values of sigma

This section steps through cross-validation results for different values of σ. Readers already familiar with cross-validation for setting hyper-parameters may wish to skip ahead to Section 4.3, where the steps are telescoped together into one R function.

If our goal is to achieve the closest possible fit to the data with these constraints, we should set σ to be very large, say 1000, so that each constraint is very free to depart from its prior mean μ (which we will set to 0, although it won't make much difference, because of the large σ). The R code below does this, and then plots the observed probabilities of the training data against their predicted probabilities under the fitted MaxEnt model (Figure 4).[10] To achieve convergence for this input file, it proved necessary to use `optimize_weights`'s `upper_bound` parameter, which tells the optimizer not to consider weights greater than, in this case, 10.[11]

```
shona_1000 <- optimize_weights(
  shona_input, mu = 0, sigma = 1000, upper_bound = 10
)

shona_1000_predictions <- predict_probabilities(
  shona_input, constraint_weights = shona_1000$weights
)

shona_1000_predictions$loglik
# -874.2246

shona_1000_predictions$predictions %>%
  ggplot(aes(x=Observed, y=Predicted)) +
  geom_point(
    shape=21, fill=alpha("blue", 0.2), stroke=1,
    size=log(shona_input_with_total$tableau_freq)) +
  geom_abline(slope=1, intercept=0, color="grey") +
  xlab("observed probability of each candidate") +
  ylab("predicted probabilities") +
  theme_classic(base_size=16)
```

[10]There is some redundancy in the plot. Each point represents one candidate, and when the probabilities of four candidates in a tableau are known, the probability of the fifth follows.

[11]Without an upper bound on constraint weights, depending on the user's particular R environment they may receive an error message from the `stats::optim` function. When this happens, our package also supplies the following message, to help the user solve the problem: "This error indicates that the likelihood function has returned a non-finite value because the constraint weights have become too large. You can resolve this by passing in a lower upper bound on maximum weights using the `upper_bound` argument, or by introducing a stronger bias towards lower weights by manipulating the `mu` and `sigma` parameters."

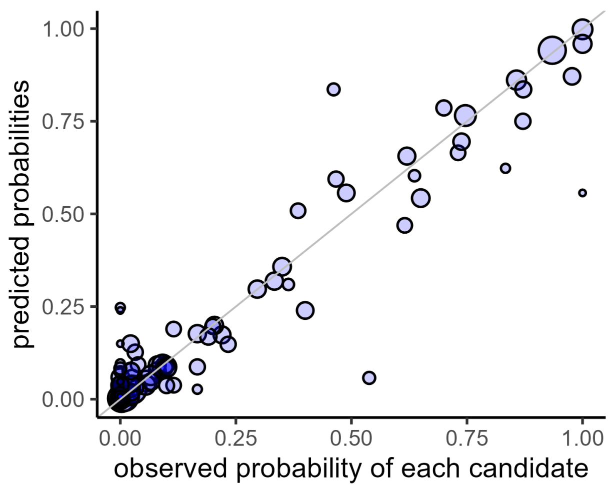


Figure 4: Observed vs. predicted probability, model fitted with σ = 1000. Each point is a candidate; the size of the point is proportional to the log of number of words that the candidate's tableau is based on. Candidates on the diagonal line are fitted perfectly.

But have we over-fitted the data? In machine learning, a popular technique to answer the question is *cross-validation* (Stone 1974; Geisser 1975; Browne 2000; for a linguistic example, see pp. 14-15 of Bresnan et al. 2007). Later in this section we will show how to use the `maxent.ot` package's `cross_validate` function to automate the process, but first, for expository purposes, we will build up the process of cross-validation step by step.

Instead of fitting the grammar to all the data, we will randomly divide the data into 80% to be used for training, and then see how well that grammar fits the held-out 20%. If the fit is very poor, then it is possible that the grammar is *over-fitted*, and is modeling peculiarities of the training data that, because they are not systematic, do not hold of the held-out data. It is also possible that the grammar is *under-fitted*, and is ignoring systematic properties of the training data that are also present in the held-out data. We will diagnose over- vs. under-fitting below by exploring different values of σ.

First, we use the `partition_data` function to split the data into five random slices, then we choose one slice to hold out; we will train on the other four slices. Making the random division is slightly complicated, because in this case we don't want to select 80% of *tableaux* to train on, but rather 80% of *words*. The `partition_data` and `populate_tableau` functions take care of this. Later, we will see that we don't even need to call these two functions directly, because the `cross_validate` function automates the work that we are breaking down step by step here for expository purposes.

```
# Housekeeping needed to format data the way partition_data() is expecting
# We're using internal functions from maxent.ot using the ::: operator
processed_input <- maxent.ot:::load_input(shona_input)
data <- processed_input$data

# slice data into 5 random parts
partitions <- maxent.ot:::partition_data(data, k=5)

# slice number 1 will be held out
hold_out <- 1
```

```r
# the training slices
training_part <- partitions %>%
  filter(partition != hold_out)

# the held-out slice
test_part <- partitions %>%
  filter(partition == hold_out)

# housekeeping to set up data the way populate_tableau() expects
training_data <- data %>%
  mutate(Frequency = 0)
test_data <- training_data

# format the training and held-out data into tableaux
training_tableau <- maxent.ot:::populate_tableau(training_data, training_part)
test_tableau <- maxent.ot:::populate_tableau(test_data, test_part)
```

Now we have a set of training tableaux and a set of testing tableaux. We fit a model to the 80% of data that were selected for training, again using a very permissive σ of 1000:

```r
# Fit model to training data
shona_1000_crossval_model <- optimize_weights(
  training_tableau, mu = 0, sigma = 1000, upper_bound = 10
)
shona_1000_crossval_model$loglik
# Model's loglik = -698.4757
```

We can use `predict_probabilities` to see how well the resulting model fits both the 80% of data it was trained on, and the 20% of data that were held out. Raw log likelihood is not informative, because log likelihood is dependent on the number of observations (each observation adds to the sum). Therefore, we divide log likelihood by the number of observations, so that we can directly compare the fit to the training data and the fit to the held-out data. The average log likelihood shown in the R code below is, as expected, better for the training data, at -0.520, and worse (more negative) for the held-out data, at -0.538 (if you run the code yourself, you will get different numbers, because of the randomness in dividing up the data into training vs. held-out sets). The plots in the left column of Figure 5 show the model's tight fit to the training data, and more scattered fit to the held-out data.

```r
# How does it do on the training data?
predictions_training <- predict_probabilities(
    training_tableau, shona_1000_crossval_model$weights
)
predictions_training$loglik / sum(predictions_training$predictions$Freq)
# average log likelihood: -0.5204737

# How does it do on the held-out data?
predictions_test <- predict_probabilities(
    test_tableau, shona_1000_crossval_model$weights
)
predictions_test$loglik / sum(predictions_test$predictions$Freq)
# average log likelihood: -0.5379555
```

We repeat the procedure here for $\sigma = 0.1$ and $\sigma = 3$ (the right and middle columns of Figure 5). For the very small $\sigma = 0.1$, the fit to both training and held-out data is poor; the model has failed to capture important patterns in the data, and thus does a poor job of predicting both the data it was trained on and new data. But for the intermediate σ value of 3, while fit to the training data is slightly poorer (average log likelihood of -0.524, vs. -0.520 when $\sigma = 1000$), the fit to the held-out data has actually improved somewhat, to an average log likelihood of -0.532 (vs. -0.538 for $\sigma = 1000$). This suggests that the model with $\sigma = 1000$ was fitting the training data too closely, capturing aspects of it that were mere artefacts of sampling, and thus, incorrectly, predicting those same artefacts to occur new data. An intermediate σ of 3 seems to smooth over more of those artefacts and do a slightly better job on the held-out data.

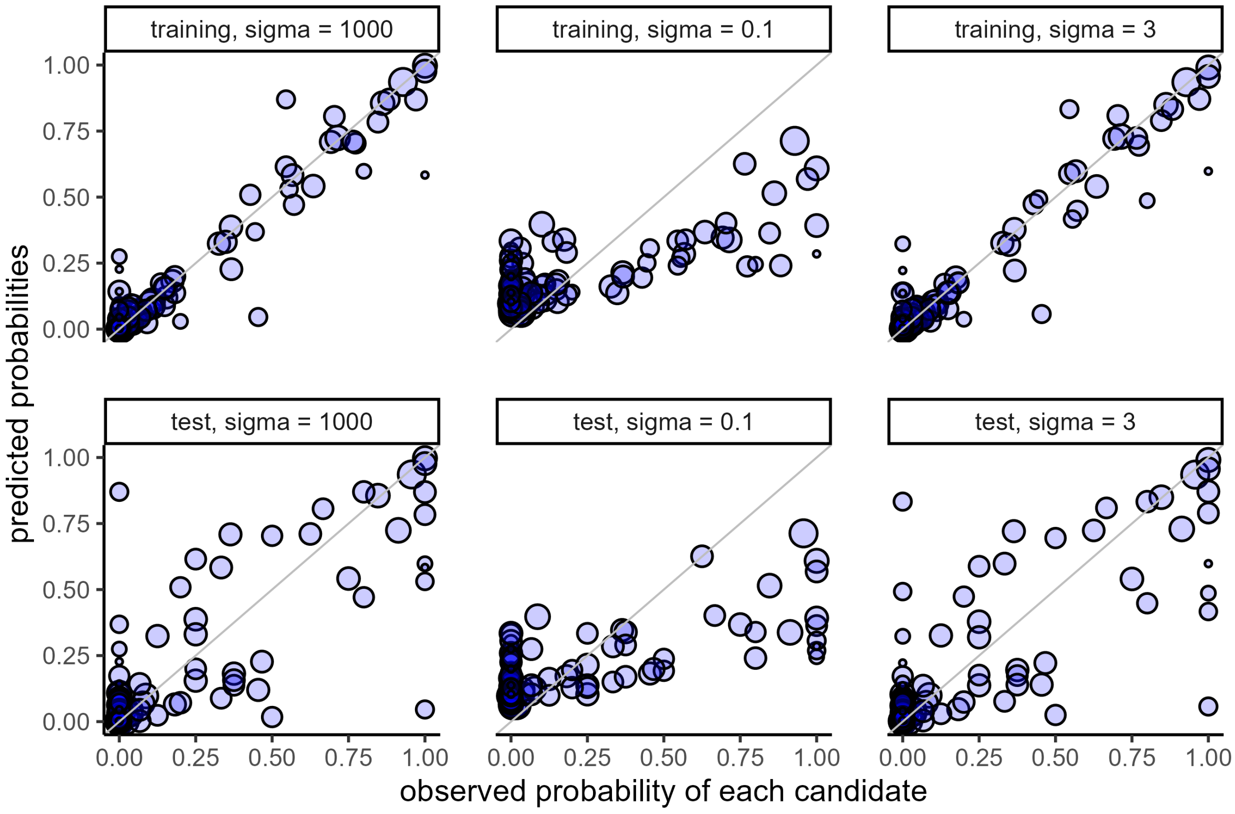


Figure 5: The predictions of the model fitted with $\sigma = 1000$ (training average LL = -0.520, test average LL = -0.538), $\sigma = 0.1$ (training average LL = -0.834, test average LL = -0.829), and $\sigma = 3$ (training average LL = -0.524, test average LL = -0.532).

All of the results in this subsection were just for one random partition of the data into 80% vs. 20%, and may not be terribly representative. The following subsection uses a built-in function from the `maxent.ot` package to automate and improve this randomization, which allows us to easily explore a range of sigma values.

4.3 Choosing the best value for sigma

In the previous subsection, we trained the grammar on random slices 1 through 4, holding out slice 5. To perform a *five-fold cross-validation*, we would next train on slices 1, 2, 3, and 5, and test on held-out slice 4; then train on slices 1, 2, 4, and 5, and test on held-out slice 3; etc. We can then obtain the mean log likelihood from the trained model's predictions on the held-out data, over these five runs. This will give us a stabler picture (though still subject to some random noise) of how well a grammar fitted to a random 80% of the data fares on the remaining 20%.

The `cross_validate` function allows us to automate this k-fold cross-validation, not only for $k = 5$ but for any choice of $k > 1$. Furthermore, we can ask the function to explore a range of μ and σ values. In this example `mu_values` is set to be 0 since we are not interested in exploring different values of μ (though if we were we could pass in a similar vector of values; see the documentation of `cross_validate` in the R package). By default the length of the μ and σ vectors needs to be the same, but specifying `grid_search = TRUE` runs the pairwise combination of all the values of μ with all the values of σ.

```
sigmas_to_try <- c(
    100, 75, 50, 40, 30, 20, 15, 10, 9, 8, 7, 6, 5.5,
    5, 4.5,4, 3.5, 3, 2.5, 2, 1.5, 1, 0.8, 0.7, 0.6, 0.5
)
# This might take some time
shona_crossval_5 <- cross_validate(
    shona_input,
    k = 5,
    mu_values = 0,
    sigma_values = sigmas_to_try,
    upper_bound = 10,
    grid_search = TRUE
)
shona_crossval_5
# Looks like
#      model_name mu sigma folds mean_ll_test mean_ll_training
# 1   shona_input  0   100     5    -183.3994        -696.2001
# 2   shona_input  0    75     5    -183.7278        -696.0800
# 3   shona_input  0    50     5    -183.7573        -696.0632
# ...
```

The resulting data frame, the top of which is shown in the R code above, has one row for each value of σ, showing the mean log likelihoods for the held-out ("test") data (not the mean per data point, but the mean over all $k = 5$ folds) and the training data. We can do the same again for $k = 10$: the data are divided into 10 slices, then trained on 1 through 9 and tested on 10, etc.

```
# This might take some time
shona_crossval_10 <- cross_validate(
  shona_input, k = 10, mu_values = 0,
  sigma_values = sigmas_to_try, upper_bound = 10, grid_search = TRUE
)
```

The code below plots the fits to training and held-out data for each value of σ for both $k = 5$ and $k = 10$, averaging over the number of data points for comparability:

```
# Format k=5 set for plotting

# get approximate number of data points for training and held-out for k=5
approx_num_of_train_5 <- sum(shona_input$Frequency) * 0.8
approx_num_of_testing_5 <- sum(shona_input$Frequency) * 0.2

# Get row corresponding to best sigma
best_row_5 <- shona_crossval_5 %>%
  filter(mean_ll_test == max(mean_ll_test))

# find best sigma value, for placing vertical line
best_sigma_5 <- best_row_5$sigma

# find best fit, for placing horizontal line
best_fit_to_test_5 <- best_row_5$mean_ll_test / approx_num_of_testing_5

# Convert to format used in plotting, normalize LL by count
shona_crossval_5_df <- shona_crossval_5  %>%
  pivot_longer(cols=c(mean_ll_training, mean_ll_test),
               names_to='type', values_to='ll') %>%
  mutate(type=ifelse(type == 'mean_ll_training',
                     'training data',
                     'held-out data'),
         ll=ifelse(type=='training data',
                   ll/approx_num_of_train_5,
                   ll/approx_num_of_testing_5),
         best_sigma=best_sigma_5,
         best_fit=best_fit_to_test_5)

# Do the same for k=10
approx_num_of_train_10 <- sum(shona_input$Frequency) * 9/10
approx_num_of_testing_10 <- sum(shona_input$Frequency) * 1/10
# Get row corresponding to best sigma
best_row_10 <- shona_crossval_10[which.max(shona_crossval_10$mean_ll_test),]
# find best sigma value, for placing vertical line
best_sigma_10 <- best_row_10$sigma
# find best fit, for placing horizontal line
best_fit_to_test_10 <- best_row_10$mean_ll_test / approx_num_of_testing_10

shona_crossval_10_df <- shona_crossval_10 %>%
  pivot_longer(cols=c(mean_ll_training, mean_ll_test),
               names_to='type', values_to='ll') %>%
  mutate(type=ifelse(type == 'mean_ll_training',
                     'training data',
                     'held-out data'),
         ll=ifelse(type=='training data',
                   ll/approx_num_of_train_10,
                   ll/approx_num_of_testing_10),
         best_sigma=best_sigma_10,
         best_fit=best_fit_to_test_10)

shona_full_crossval <- rbind(shona_crossval_tmp, shona_crossval_10_tmp) %>%
  mutate(folds=fct_relevel(ifelse(folds == 5, 'k=5', 'k=10'), 'k=5'))

# Make plot
shona_full_crossval %>%
  ggplot(aes(as.numeric(sigma), ll, fill=type, color=type, shape=type, lty=type)) +
  geom_point() +
  geom_line() +
  xlab("sigma value used in training") +
  ylab("average log likelihood / N") +
  scale_x_continuous(trans='log10', breaks=c(0.5, 2, 5, 20, 100)) +
  geom_abline(aes(slope=0, intercept=best_fit), color="grey", lty=2) +
  geom_vline(aes(xintercept=as.numeric(best_sigma)), col="grey", lty=2) +
  theme_classic(base_size=16) +
  theme(legend.title= element_blank(), panel.spacing = unit(1, "lines")) +
  facet_grid(~ folds)
```

Figure 6 shows the resulting plot. In each subplot, the fit to the training data (blue triangles on dashed line) increases asymptotically as σ increases. This is to be expected: the higher the σ, the more closely the model's weights can fit the training data. But the fit to the held-out data (red circles on solid line) reaches a maximum somewhere between σ values of 2 and 5, and then declines somewhat for higher values of σ, because these higher values produce over-fitting. Taking the two plots together, we see that the best σ for this data set is somewhere around 3 to 5, with little difference within that range.

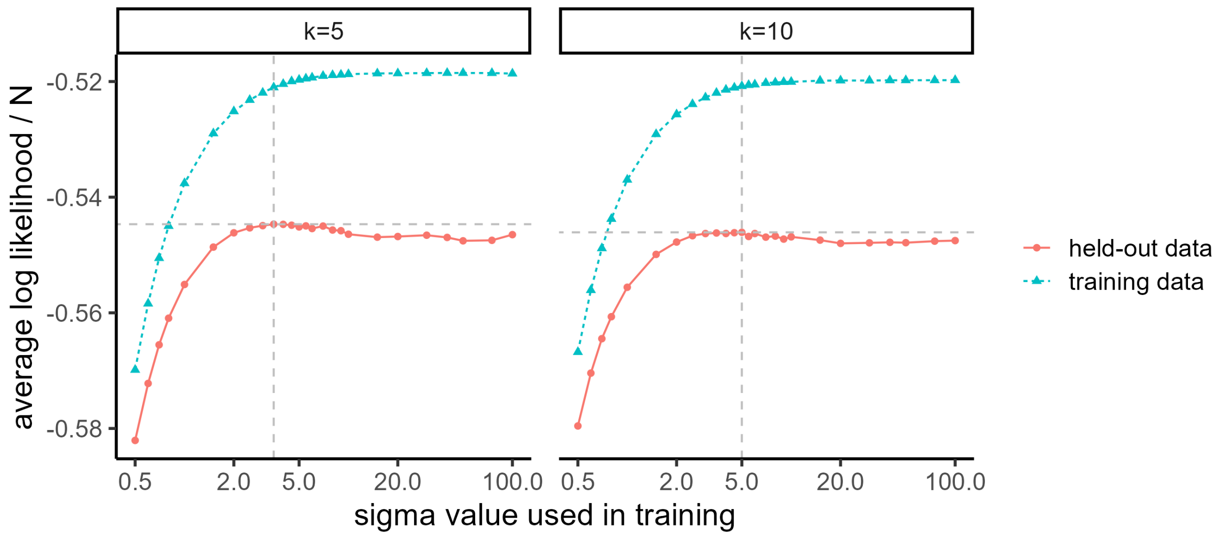


Figure 6: Mean normalized log-likelihood for training data and held-out data during k-fold cross validation as a function of σ.

The `cross_validate` function thus allows the researcher to explore values of the bias parameters μ and σ to find those that are suitable for their data set. It would be ideal if there were a single "find optimal sigma" or "find optimal mu" function one could call, but the curve for the fit to held-out data as a function of σ is not guaranteed to be convex–visual inspection shows that it is not completely convex in Figure 6, though this could be an artefact of the random division of the data into slices–so until the relationship of bias terms to fit-to-held-out is better understood, we leave this an area for researchers to explore using `cross_validate`.

Henriksson (2022) demonstrates a different use of cross-validation. As in our Shona example, Henriksson uses cross-validation to assess how well models perform on untrained data, but instead of comparing models with different prior terms, Henriksson compares models with different sets of constraints, and also looks at how much individual constraints' weights vary across different slices of training data. Our `cross_validate` function can be used for these purposes, except that instead of a single call to `cross_validate` with multiple μ and σ values, the user could call `cross_validate` once for each model of interest, giving a single value for μ and a single value of σ each time.

5 Conclusion and further resources

This paper has presented a gentle (we hope) introduction to Maximum Entropy Optimality analyses, illustrated using the `maxent.ot` R package. We showed that `maxent.ot` supplies a variety of useful functionality for MaxEnt analyses, including

- Fitting constraint weights to data consisting of constraint violation profiles, with and without a Gaussian bias term.
- Predicting probability distributions over candidates given a set of violation profiles and a vector of constraint weights.
- Model comparison using AIC, AICc, BIC, and the likelihood ratio test.
- k-fold cross-validation to determine optimal bias parameters.

We summarize here a typical workflow:

1. Create an input `dataframe` or `tibble`, either directly or by reading from a file, using the format shown in Section 3.3 to specify underlying and surface forms, frequencies, and constraint violations.
2. Using an input `dataframe` or `tibble`, which we'll refer to as `my_input`, fit a MaxEnt model to it using `my_model <- optimize_weights(my_input)` function. This returns a fitted model with attributes including weights.
   - To impose a Gaussian prior on constraint weights, add the arguments `mu` and `sigma`, as in
     ```
     my_model <- optimize_weights(my_input,
                                  mu = 0,
                                  sigma = c(1, 1, 0.1, 1))
     ```
   - If this produces an error message, impose a strict upper bound on constraint weights, by adding the argument `upper_bound`, as in
     `my_model <- optimize_weights(my_input, upper_bound = 10)`
3. Find out what probabilities the fitted model predicts, using
   `my_predictions <- predict_probabilities(my_input, my_model$weights)`.
   Or, find out what a different vector of constraint weights predicts by replacing
   `my_model$weights` with a different weight vector.

4. Compare models using BIC by calling `compare_models(my_model, my_model2, my_model3, method="bic")`. Or, replace `"bic"` with `"aic"` to use AIC, `"aic_c"` to use AICc, or `"lrt"` to use the likelihood ratio test.

5. To use cross-validation to find good values of μ and σ, use the `cross_validate` function, as in

```
my_crossval <- cross_validate(my_input,
                              k = 10,
                              mu_values = c(0,1),
                              sigma_values = c(1, 2, 4, 8, 16),
                              grid_search = TRUE)
```

   In this case, we have specified 10-fold cross-validation (`k = 10`), and a "grid search" trying all combinations of μ and σ given.

   - To make plots of fit to held-out and training data over different values of bias terms, see our code in Section 4.3.

You can get more information about `maxent.ot` in the following ways:

- `maxent.ot` is free and open-source software released under the GNU General Public License v3.0. The full source code can be found at https://github.com/connormayer/maxent.ot.

- More detailed documentation for each function can be found by typing `?` followed by its name in the R console (e.g., `?optimize_weights` will bring up documentation about the `optimize_weights` function). This is particularly useful for viewing the full set of arguments a function accepts.

- You can view the documentation and vignettes for `maxent.ot` on the R CRAN repository, which will be linked to in the README on the `maxent.ot` GitHub page. The documentation contains similar information to what you can view using the `?` operator as in the previous bullet point, and the vignettes demonstrate several use cases that go into more detail than what has been presented in this paper.

Finally, because this is free and open-source software, we welcome contributions! If you find bugs or have additional features you would like to see added, please contact cjmayer@uci.edu. We hope that this package will be a useful tool in future research for both proponents and critics of MaxEnt models of phonology.

Appendix A OTSoft format

`maxent.ot` also supports the OTSoft file format (Hayes et al. 2013). An example of this formatting is shown in Table 27, and the corresponding text file can be found at `data/simple_input_otsoft.txt` in the tutorial materials.

			StarComplex	Max
			StarComplex	Max
ˈstad	ˈstad	40	1	
	ˈtad	60		1
ˈgʁav	ˈgʁav	60	1	
	ˈgav	40		1
spa.gɛ.ˈti	spa.gɛ.ˈti	10	1	
	spa.gɛ.ˈti	90		1
gʁy.ˈo	gʁy.ˈo	30	1	
	gy.ˈo	70		1

Table 27: OTSoft formatted input.

A few things to note about this format:

- Columns are tab-delimited.
- The first two rows contain constraint names. The first four columns are not named. The first row contains the 'long names' and the second row contains the 'short names'. The 'short names' row is maintained for backwards compatibility with the OTSoft format, and is not used by `maxent.ot`.
- Each row following the constraint names consists of the following columns:
  - The underlying representation. This is only specified for the first occurrence of each input in the file. Rows with no values in this column are assumed to have the same input as closest non-empty value above.
  - The surface representation.
  - The observed frequency counts.
  - The violation profile of this input-output pair. This consists of one column for each constraint. Values must be numeric. Empty cells and cells with 0 violations are treated equivalently.

`maxent.ot` provides the `otsoft_tableaux_to_df` function that converts files in OTSoft format to the `dataframe` format described in Section 3.3. If using the OTSoft format for the tutorial, you can replace the code in Section 3.3 with the line below.

```
simple_input <- otsoft_tableaux_to_df("data/simple_input_otsoft.txt")
```

There is no need to use the OTSoft format aside from personal preference or compatibility with existing data.

Appendix B The temperature parameter

Previous work (e.g., Hayes & Wilson 2008; Hayes et al. 2009; Mayer 2021) has used a *temperature parameter* when modeling forced-choice wug test data (Ackley et al. 1985:p. 150–152). The motivation behind this parameter is that such wug tests typically display responses that are less categorical—closer to 50/50—than the statistics seen in the lexicon. This behavior is taken to be driven by task factors rather than grammatical factors: e.g., because participants are explicitly considering the choice between two (or more) different forms. Thus, rather than adjust the grammar itself to model wug data, an adjustment is made to the way probabilities are derived from the grammar.

Equation 18 shows the equation for deriving the probability of a candidate given the corresponding weights and violation profiles with a temperature term T included. Increasing T has the effect of making the probabilities of all candidates closer to equal.

(18)

$$P(y|x) = \frac{\exp(-H(x,y)/T)}{\sum_{y' \in \mathcal{Y}(x)} \exp(-H(x,y')/T)}.$$

Equation 4 in the body of the paper can be considered a special case of this equation where $T = 1$.

Although the linguistic applications are less clear, T can also be set to a value less than 1. As T is decreased, probability shifts away from the low-harmony candidates and onto the high-harmony candidate, until, when T is near 0, the highest-harmony candidate (or candidates, if there is a harmony tie) is predicted to occur nearly all the time. Negative values of T are even possible; they cause more probability to be assigned to *lower*-harmony candidates (it is even harder to imagine linguistic applications of negative T).

The temperature parameter can be set using the `temperature` argument of the `predict_probabilities` function, as below, where the four-constraint grammar for child French generates predictions. The result, shown in Table 28 is that candidates still lean in the direction of the observed probabilities, but they are closer to 50-50 than they were before, in the rightmost column of Table 15, when a temperature of 1 was being implicitly used for generating predictions from the same grammar.

```
# Call predict_probabilities with the sample parameters used to generate
# predictions in last column of Table 15, but with temperature = 2.
result <- predict_probabilities(
  max_stressed_ssp_input, model_max_stressed_ssp$weights, temperature=2
)

result$loglik
# -239.3805

result$predictions
# Table 28
```

UR	SR	Freq	StarComplex	Max	MaxStressed	SSP	Predicted	Observed	Error
ˈstad	ˈstad	40	1	0	0	1	0.4375327	0.4	0.03753267
ˈstad	ˈtad	60	0	1	1	0	0.5624673	0.6	-0.03753267
ˈgrav	ˈgrav	60	1	0	0	0	0.5624672	0.6	-0.03753279
ˈgrav	ˈgav	40	0	1	1	0	0.4375328	0.4	0.03753279
spagɛˈti	spagɛˈti	10	1	0	0	1	0.2724764	0.1	0.17247640
spagɛˈti	pagɛˈti	90	0	1	0	0	0.7275236	0.9	-0.17247640
gryˈjo	gryˈjo	30	1	0	0	0	0.3823155	0.3	0.08231551
gryˈjo	gyˈjo	70	0	1	0	0	0.6176845	0.7	-0.08231551

Table 28: The output of the `predict_probabilities` function with `temperature=2`.

For example, in Table 15, the probabilities of a faithful vs. repaired realization of the onset cluster in /stad/ are 0.38 vs. 0.62. Here they are 0.44 vs. 0.56. The basic preference for repair remains, but the probabilities become closer to 50/50.

References

Ackley, D., G. Hinton & T. Sejnowski. 1985. A learning algorithm for Boltzmann machines. *Cognitive Science* 9. 147–169.

Akaike, Hirotugu. 1974. A new look at the statistical model identification. *IEEE Transactions on Automatic Control* 19. 716–723. https://doi.org/10.1007/978-1-4612-1694-0_16.

Albright, Adam & Bruce Hayes. 2003. Rules vs. analogy in English past tenses: a computational/experimental study. *Cognition* 90(2). 119–161. https://doi.org/10.1016/S0010-0277(03)00146-X.

Allaire, JJ, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, Winston Chang & Richard Iannone. 2021. rmarkdown: Dynamic Documents for R. R package version 2.7. https://rmarkdown.rstudio.com.

Anttila, Arto, Scott Borgeson & Giorgio Magri. 2019. Equiprobable mappings in weighted constraint grammars. In Garrett Nicolai & Ryan Cotterell (eds.), *SIGMORPHON 2019: Proceedings of the 16th ACL Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 125–134. Association for Computational Linguistics. https://doi.org/10.18653/v1/W19-4215.

Beckman, Jill N. 1998. *Positional faithfulness*: University of Massachusetts Amherst dissertation. https://doi.org/10.1002/9780470756171.ch16.

Berko, Jean. 1958. The child's learning of English morphology. *Word* 14(2-3). 150–177. https://doi.org/10.1080/00437956.1958.11659661.

Bisong, Ekaba. 2019. *Google colaboratory* 59–64. Berkeley, CA: Apress. https://doi.org/10.1007/978-1-4842-4470-8_7.

Boersma, Paul & Joe Pater. 2016. Convergence properties of a gradual learning algorithm for Harmonic Grammar. In John J. McCarthy & Joe Pater (eds.), *Harmonic Serialism and Harmonic Grammar*, 389–434. Sheffield: Equinox.

Breiss, Canaan. 2020. Constraint cumulativity in phonotactics: evidence from Artificial Grammar Learning studies. *Phonology* 37(4). 551–576. https://doi.org/10.1017/S0952675720000275.

Bresnan, Joan, Anna Cueni, Tatiana Nikitina & R Harald Baayen. 2007. Predicting the dative alternation. In *Cognitive foundations of interpretation*, 69–94. Amsterdam: KNAW.

Bridle, John. 1989. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in Neural Information Processing Systems* 2.

Browne, Michael W. 2000. Cross-validation methods. *Journal of Mathematical Psychology* 44(1). 108–132. https://doi.org/10.1006/jmps.1999.1279.

Burnham, Kenneth P. & David R. Anderson. 2004. Multimodal inference: Understanding AIC and BIC in model selection. *Sociological Methods & Research* 33(2). 261–304. https://doi.org/10.1177/0049124104268644.

Carlisle, Robert. 1991. The influence of environment on vowel epenthesis in Spanish/English interphonology. *Applied Linguistics* 12(76–95). https://doi.org/10.1093/applin/12.1.76.

Carlisle, Robert S. 2001. Syllable structure universals and second language acquisition. *International Journal of English Studies* 1(1). 1–19. https://doi.org/20.500.12680/mk61rk413.

Chen, Stanley F. & Ronald Rosenfeld. 1999. A Gaussian prior forsmoothing maximum entropy models. Tech. rep. Canergie Mellon University.

Cho, Young-mee Yu & Tracy Holloway King. 2003. Semisyllables and universal syllabification. In Caroline Féry & Ruben van de Vijver (eds.), *The syllable in Optimality Theory*, 183–212. New York: Cambridge University Press. https://doi.org/10.1017/CBO9780511497926.008.

Daland, Robert. 2015. Long words in maximum entropy phonotactic grammars. *Phonology* 32(3). 353–383. https://doi.org/10.1017/S0952675715000251.

Della Pietra, Stephen A., Vincent J. Della Pietra & John Lafferty. 1997. Inducing features of random fields. *IEEE Transactions: Pattern Analysis and Machine Intelligence* 19(4). 380–393.

Flemming, Edward. 2021. Comparing MaxEnt and Noisy Harmonic Grammar. *Glossa* 6(1). 1–42. https://doi.org/10.16995/glossa.5775.

Fylstra, Daniel, Leon Lasdon, John Watson & Allan Waren. 1998. Design and Use of the Microsoft Excel Solver. *INTERFACES* 28(5). 29–55. https://doi.org/10.1287/inte.28.5.29.

Geisser, Seymour. 1975. The predictive sample reuse method with applications. *Journal of the American statistical Association* 70(350). 320–328. https://doi.org/10.1080/01621459.1975.10479865.

Gilani, Saiem & Geoff Hutchinson. 2021. wehoop: The sportsdataverse's r package for women's basketball data. https://wehoop.sportsdataverse.org.

Goldwater, Sharon & Mark Johnson. 2003. Learning OT constraint rankings using a maximum entropy model. In Jennifer Spenader, Anders Eriksson & Östen Dahl (eds.), *Proceedings of the Stockholm Workshop on Variation within Optimality Theory*, 111–120. Stockholm: Stockholm University, Department of Linguistics.

Hayes, Bruce, Bruce Tesar & Kie Zuraw. 2013. OTSoft.

Hayes, Bruce & Colin Wilson. 2008. A maximum entropy model of phonotactics and phonotactic learning. *Linguistic Inquiry* 39(3). 379–440. https://doi.org/10.1162/ling.2008.39.3.379.

Hayes, Bruce, Colin Wilson & Ben George. 2009. Maxent Grammar Tool. http://www.linguistics.ucla.edu/people/hayes/MaxentGrammarTool/.

Hayes, Bruce, Colin Wilson & Anne Shisko. 2012. Maxent grammars for the metrics of Shakespeare and Milton. *Language* 88(4). 691–731. https://doi.org/10.1353/lan.2012.0089.

Heeringa, Wilbert & Hans Van de Velde. 2018. Visible vowels: a tool for the visualization of vowel variation. In *Proceedings of CLARIN Annual Conference 2018, 8 - 10 October, Pisa, Italy*, 120 – 123. CLARIN ERIC. https://office.clarin.eu/v/CE-2018-1292-CLARIN2018_ConferenceProceedings.pdf.

Henriksson, Erik. 2022. *Greek meter: An approach using metrical grids and maxent*: University of Helsinki dissertation.

Hughto, Coral, Andrew Lamont, Brandon Prickett & Gaja Jarosz. 2019. Learning exceptionality and variation with lexically scaled MaxEnt. In *Proceedings of the Society for Computation in Linguistics 2019*, 91–101. https://doi.org/https://doi.org/10.7275/y68s-kh12.

Jäger, G. & A. Rosenbach. 2006. The winner takes all – almost: Cumulativity in grammatical variation. *Linguistics* 44(5). 937–971. https://doi.org/10.1515/LING.2006.031.

Jarosz, Gaja. 2017. Defying the stimulus: Acquisition of complex onsets in Polish. *Phonology* 34(2). 269–298. https://doi.org/10.1017/S0952675717000148.

Kaplan, Aaron. 2018. Positional licensing, asymmetric trade-offs and gradient constraints in Harmonic Grammar. *Phonology* 35. 247–286. https://doi.org/10.1017/S0952675718000040.

Kimper, Wendell. 2011. *Competing triggers: transparency and opacity in vowel harmony*: University of Massachusetts Amherst dissertation.

Kluyver, Thomas, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla & Carol Willing. 2016. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides & B. Schmidt (eds.), *Positioning and power in academic publishing: Players, agents and agendas*, 87 – 90. IOS Press.

Knuth, Donald E. 1992. *Literate programming*. Standford, CA: Center for the Study of Language and Information. https://doi.org/10.1093/comjnl/27.2.97.

Legendre, Géraldine, Yoshiro Miyata & Paul Smolensky. 1990. Harmonic grammar - a formal multi-level connectionist theory of linguistic well-formedness: theoretical foundations. Tech. Rep. 90-5 Institute of Cognitive Science, University of Colorado at Boulder.

Luce, R Duncan. 1959. Individual choice behavior. .

Magri, Giorgio & Arto Anttila. 2022. Paradoxes of MaxEnt markedness. In Noah Elkins, Bruce Hayes, Jinyoung Jo & Jian-Leat Siah (eds.), *AMP 2022: Supplemental Proceedings of the 2022 Annual Meeting on Phonology*, Washington, DC: Linguistic Society of America. https://doi.org/10.3765/amp.v10i0.5445.

Martin, Andrew. 2007. *The evolving lexicon*: University of California, Los Angeles dissertation.

Martin, Andrew. 2011. Grammars leak: modeling how phonotactic generalizations interact with the grammar. *Language* 87(4). 751–770. https://doi.org/10.1353/lan.2011.0096.

Mayer, Connor. 2021. *Issues in Uyghur backness harmony: Corpus, experimental and computational studies*: University of California, Los Angeles dissertation.

McCarthy, John J. & Alan Prince. 1994. The emergence of the unmarked: Optimality in prosodic morphology. In *Proceedings of the North East Linguistics Society 24*, 18.

Pater, Joe. 2009. Weighted constraints in generative linguistics. *Cognitive Science* 33(6). 999–1035. https://doi.org/10.1111/j.1551-6709.2009.01047.x.

Peng, Roger D. 2011. Reproducible research in computational science. *Science* 334(6060). 1226–1227. https://doi.org/10.1126/science.1213847.

Pereira, R.H.M., C.N. Gonçalves et al. 2019. geobr: Loads shapefiles of official spatial data sets of Brazil. https://github.com/ipeaGIT/geobr.

Prince, Alan & Paul Smolensky. 1993/2004. *Optimality theory: Constraint interaction in generative grammar*. Cambridge, MA: Blackwell. https://doi.org/10.1002/9780470759400.

R Core Team. 2022. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing Vienna, Austria. https://www.R-project.org/.

Raftery, Adrian E. 1995. Bayesian model selection in social research. *Sociological Methodology* 25. 111–163. https://doi.org/10.2307/271063.

Rose, Yvan. 2002. Relations between segmental and prosodic structure in first language acquisition. *The Annual Review of Language Acquisition* 2(1). 117–155. https://doi.org/10.1075/arla.2.06ros.

RStudio Team. 2020. *Rstudio: Integrated development environment for r*. RStudio, PBC. Boston, MA. http://www.rstudio.com/.

Schütz, Frédéric & Alix Zollinger. 2018. ABPS: An R Package for Calculating the Abnormal Blood Profile Score. *Frontiers in Physiology* 9(1638). https://doi.org/10.3389/fphys.2018.01638.

Schwab, Matthias, N Karrenbach & Jon Claerbout. 2000. Making scientific computations reproducible. *Computing in Science & Engineering* 2(6). 61–67. https://doi.org/10.1109/5992.881708.

Schwarz, Gideon. 1978. Estimating the dimension of a model. *Annals of Statistics* 6(2). 461–464. https://doi.org/10.1214/aos/1176344136.

Selkirk, Elisabeth. 1984. On the major class features and syllable theory. In Mark Aronoff & Richard T. Oehrle (eds.), *Language sound structure*, 107–136. Cambridge, MA: MIT Press.

Shih, Stephanie. 2017. Constraint conjunction in weighted probabilistic grammar. *Phonology* 34(2). 243–268. https://doi.org/10.1017/S0952675717000136.

Smolensky, Paul. 1986. Information processing in dynamical systems: Foundations of harmony theory. In David E. Rumelhart, John L. McClelland & The PDP Research Group (eds.), *Parallel distributed processing: Explorations in the microstructure of cognition*, 194–281. Cambridge, MA: MIT Press/Bradford Books.

Staubs, Robert. 2011. Harmonic Grammar in R (hgR). https://websites.umass.edu/hgr/.

Steriade, Donca. 1982. *Greek prosodies and the nature of syllabification*: Massachusetts Institute of Technology dissertation.

Stodden, Victoria, Friedrich Leisch & Roger D. Peng. 2014. *Implementing reproducible research*. New York: CRC Press. https://doi.org/10.1201/9781315373461.

Stone, Mervyn. 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society: Series B (Methodological)* 36(2). 111–133. https://doi.org/10.1111/j.2517-6161.1974.tb00994.x.

Tilsen, Sam. 2023. Probability and randomness in phonology: Deep vs. shallow stochasticity. *Studies in Phonetics, Phonology, and Morphology* 29(2). https://doi.org/10.17959/sppm.2023.29.2.247.

Uffmann, Christian. 2007. *Vowel epenthesis in loanword adaptation*. Tübingen: Max Niemeyer Verlag. https://doi.org/10.1515/9783110934823.

Van Rossum, Guido & Fred L. Drake. 2009. *Python 3 reference manual*. Scotts Valley, CA: CreateSpace.

Vrieze, Scott I. 2012. Model selection and psychological theory: A discussion of the differences between the Akaike information criterion (AIC) and the Bayesian information criterion (BIC). *Psychological Methods* 17(2). 228–243. https://doi.org/10.1037/a0027127.

Wagenmakers, Eric-Jan & Simon Farrell. 2004. AIC model selection using Akaike weights. *Psychonomic Bulletin & Review* 11(1). 192–196. https://doi.org/10.3758/BF03206482.

White, Jamie. 2013. *Bias in phonological learning: Evidence from saltation*: University of California, Los Angeles dissertation.

Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain Franois, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, Max Kuhn, Thomas Lin Pedersen, Evan Miller, Stephan Milton Bache, Kirill Mller, Jeroen Ooms, David Robinson, Dana Paige Seidel, Vitalie Spinu, Kohske Takahashi, Davis Vaughan, Claus Wilke, Kara Woo & Hiroaki Yutani. 2019. Welcome to the tidyverse. *Journal of Open Source Software* 4(43). 1686. https://doi.org/10.21105/joss.01686.

Wilson, Colin. 2006. Learning phonology with substantive bias: An experimental and computational study of velar palatalization. *Cognitive Science* 30(5). 945–982. https://doi.org/10.1207/s15516709cog0000_89.

Xie, Yihui. 2014. knitr: A comprehensive tool for reproducible research in R. In Victoria Stodden, Friedrich Leisch & Roger D. Peng (eds.), *Implementing reproducible computational research*, Boca Raton, Florida: Chapman and Hall. https://doi.org/10.1201/9781315373461-1. ISBN 978-1466561595.

Xie, Yihui, J. J. Allaire & Garrett Grolemund. 2018. *R markdown: The definitive guide*. Boca Raton, Florida: Chapman & Hall. https://doi.org/10.1201/9781138359444.

Xie, Yihui, Christophe Dervieux & Emily Riederer. 2020. *R markdown cookbook*. Boca Raton, Florida: Chapman & Hall. https://doi.org/10.1201/9781003097471.

Yavaş, Mehmet, Avivit Ben-David, Ellen Gerrits, Kristian E. Kristoffersen & Hanne G. Simonsen. 2009. Sonority and cross-linguistic acquisition of initial s-clusters. *Clinical Linguistics and Phonetics* 22(6). 421–441. https://doi.org/10.1080/02699200701875864.