Manual: Phonotactic learning program[1]

Version 1.1, June 2008

Manual revised November 2009

Frank Capodieci
Bruce Hayes
Colin Wilson

UCLA

[1] This manual was written by Bruce Hayes, reflecting input from Colin Wilson and Frank Capodieci.

Contents

1. Authorship

This program embodies the phonotactic learning algorithm described in the following article: Bruce Hayes and Colin Wilson (2008) “A Maximum Entropy Model of Phonotactics and Phonotactic Learning,” *Linguistic Inquiry* 39: 379-440.[2] The program was written in Java by Colin Wilson, with a user interface by Frank Capodieci.

2. What the program does

The focus of the program is the **phonotactics** of a language; that is, the principles that determine what is a phonologically legal word. For example, in English “blick” [blɪk] is legal, “bnick” [bnɪk] is illegal, and “bwick” [bwɪk] has an intermediate status.

When provided with a corpus of language data consisting of words in phonetic transcription, the program attempts to learn the phonotactic constraints implicit in the corpus. These constraints can then be used to evaluate novel forms, assigning them numerical values corresponding to their predicted phonotactic well-formedness.

The learning data consist of a list of forms in a user-chosen transcription, each datum accompanied by a integer frequency value. For example, the file of learning data from the Shona simulation reported in Hayes and Wilson (2008) begins like this:

```
a dh a           1
a k i t a        1
a m b u r a      1
a m w a          1
a N g a r a      1
a N g a r a r a  1
a n zv a         1
a sh a n u dz a  1
a t s a m a      1
a u d y u dz a   1
. . .
```

The program also needs a **feature chart**, which defines the symbols according to their phonetic (featural) properties. The format is very simple: the top row labels are feature names, the left side labels are the speech sounds, and the values are +, -, or 0 (unspecified). The separator for the columns is a tab. For example, the upper left hand corner of the feature chart for Shona looks like this:

[2] Preprint version downloadable from http://www.linguistics.ucla.edu/people/hayes/Papers/HayesWilsonMaximumEntropyPhonotacticsAugust2007.pdf.

	syllabic	consonantal	approximant	sonorant
w	-	-	+	+
y	-	-	+	+
m	-	+	-	+
v	-	+	+	+
p	-	+	-	-
pf	-	+	-	-
b	-	+	-	-
bh	-	+	-	-

Lastly, users of the program normally include a set of **testing data**, to see if the grammar learned by the program is adequate. Testing data consist of single words, in the same transcription system as the learning data, and optionally annotated with one or more classifying headings (tab-separated). Here is the start of the testing data for the Shona example; the right two columns serve purely for the convenience of the users: [3]

```
m e m i m a   SingleC_bad            3
m e m o m a   SingleC_bad            0
m u m e m a   SingleC_bad            4
m a m e m a   SingleC_bad            3
m u m o m a   SingleC_bad            1
m i m e m a   SingleC_bad            0
m a m o m a   SingleC_bad            1
m i m o m a   SingleC_bad            0
m o m i m a   SingleC_bMarginal     23
m o m u m a   SingleC_bMarginal     20
m o m o m a   SingleC_good         705
m e m e m a   SingleC_good         601
m i m i m a   SingleC_good         507
```

The program will read the learning data and the feature chart, will attempt to find the best phonotactic grammar it can, and will print out output files listing the grammar and what it predicts for the test forms.

2.1 Output files created

The program will create these files in a new folder called **output**, which will be the daughter of the folder that contains the input files.

- **Grammar.txt**. This has the constraints that the system discovered, along with their weights. For example, in the Shona simulation, the first four constraints are:

```
*[+word_boundary][+word_boundary]  (tier=Vowel)     1.756
*[-syllabic][+word_boundary]       (tier=default)   7.529
*[-low][+word_boundary]            (tier=Vowel)     7.347
*[-voice][-continuant]             (tier=default)   4.105
```

[3] As it happens, “singleC” means one consonant between the first and second vowels, and the integers denote the frequency of the pair consisting of the first two vowels in the CBOLD database; see Hayes and Wilson (in press) for details. The point is that any number of tab-separated fields that might be useful to the program user (for example, in sorting the output file in a spreadsheet program) may be included.

The “tiers” are discussed below in section 5.2.

- **blickTestResults.txt**. This file includes all the forms that you included in your testing data, and gives their constraint violations and the computed weighted sum for these violations (summed products of violations times constraint weights), indicating the degree of penalty assigned.[4] For the Shona simulation, the output file (truncated to make it visible) looks like this:

word	score	*[+word_boundary][+word_boundary]	*[-syllabic][+word_boundary]	*[-low][+word_boundary]	(other constraints omitted)	annotation
		Vowel	default	Vowel		
		1.832	7.527	7.348		
m e m i m a	0	0	0	0	…	AsingleC_bad
m e m o m a	3.599	0	0	0	…	AsingleC_bad
m u m e m a	3.863	0	0	0	…	AsingleC_bad
m a m e m a	0	0	0	0	…	AsingleC_bad
m u m o m a	3.863	0	0	0	…	AsingleC_bad

For how to convert the scores into probabilities, see section 8 below.

Other files, less crucial but possibly useful, are as follows.

- **sampleSalad.txt**. A set of randomly-created forms, which match the probability distribution defined by the grammar.[5]
- **NatClassesFile.txt**. The set of natural classes defined by your feature system.

[4] The name “blick test” comes from the example given by Chomsky and Halle (1965, *Journal of Linguistics*), who offer “blick” as an example of a non-existing but possible word of English, and “bnick” as a non-existing and impossible word. A “blick test” for English might consist of obtaining native speakers well-formedness judgments for words of the “blick” type and words of the “bnick” type. It can be illuminating to give the same blick test to both native speakers and proposed computational models; hence the name of the output file.

[5] The file name is whimsical: if for whatever reason the program does a bad job, the outputs will look like phoneme salad. The inspiration is “word salad”, used by the linguist Haj Ross for an utterly ungrammatical sentence.

- **LearnerSettings.txt**. A file that records the settings used in running the learning simulation.
- **ProgramTrace.txt**. The program's report of its own behavior; not really for public consumption but possibly useful for various purposes.

3. Installing the program

You need a computer that runs Java. This can include several different operating systems (Windows, Mac, Linux).

To enable your computer to run Java, you need to download the "Java runtime environment." Most computers already have it; you can check yours by searching for the file "java.exe". If you don't have the Java runtime environment, first download it (for free) from http://www.java.com/en/download/index.jsp.

The program uses a version of Java that is quite up-to-date (as of late 2007), at least version 1.6. You'll need to update your Java if your own copy is older than this.[6]

The program is downloaded in the form of a zip file. Put the file in a new folder and use unzipping software (readily obtained on the Internet) to unzip it.

The crucial materials are:

- **uclaplui.jar** — this is the program itself, click on it to run
- A folder called **lib**. This is a library of software, obtained under various open-source software packages,[7] that the main program uses when it runs. The materials in this library are licensed under various free public licensing agreements. You need to have the **lib** folder sitting in the same folder at **uclaplui.jar**.

3.1 A caution regarding running the program in Windows

Java and Windows are run by different companies and suffer from certain (manageable) compatibility problems. If you are using Windows to run this program, you can avoid trouble if you **never put spaces** into file names or folder names involving the program. This will let Java find the files that it needs.

[6] This will be harder if you have an older Mac, which may not run the newer Java. Try the SoyLatte Java, at http://landonf.bikemonkey.org/static/soylatte/.

[7] For the licenses, consult the following. commons-math1.0.jar, commons-cli-1.0.jar: http://jakarta.apache.org, colt.jar: http://dsd.lbl.gov/~hoschek/colt/, jas.jar: http://java.freehep.org/, trove.jar: http://trove4j.sourceforge.net/, datafile.jar: http://sourceforge.net/projects/datafile/, pal-1.5.jar: http://www.eso.org/science/scisoft7/, commons-io-1.2.jar: http://commons.apache.org/, swing-layout-1.0.jar: https://designgridlayout.dev.java.net/. If you download and use this software, you are agreeing to abide by the licenses for all of these packages (merely using the software will satisfy this requirement).

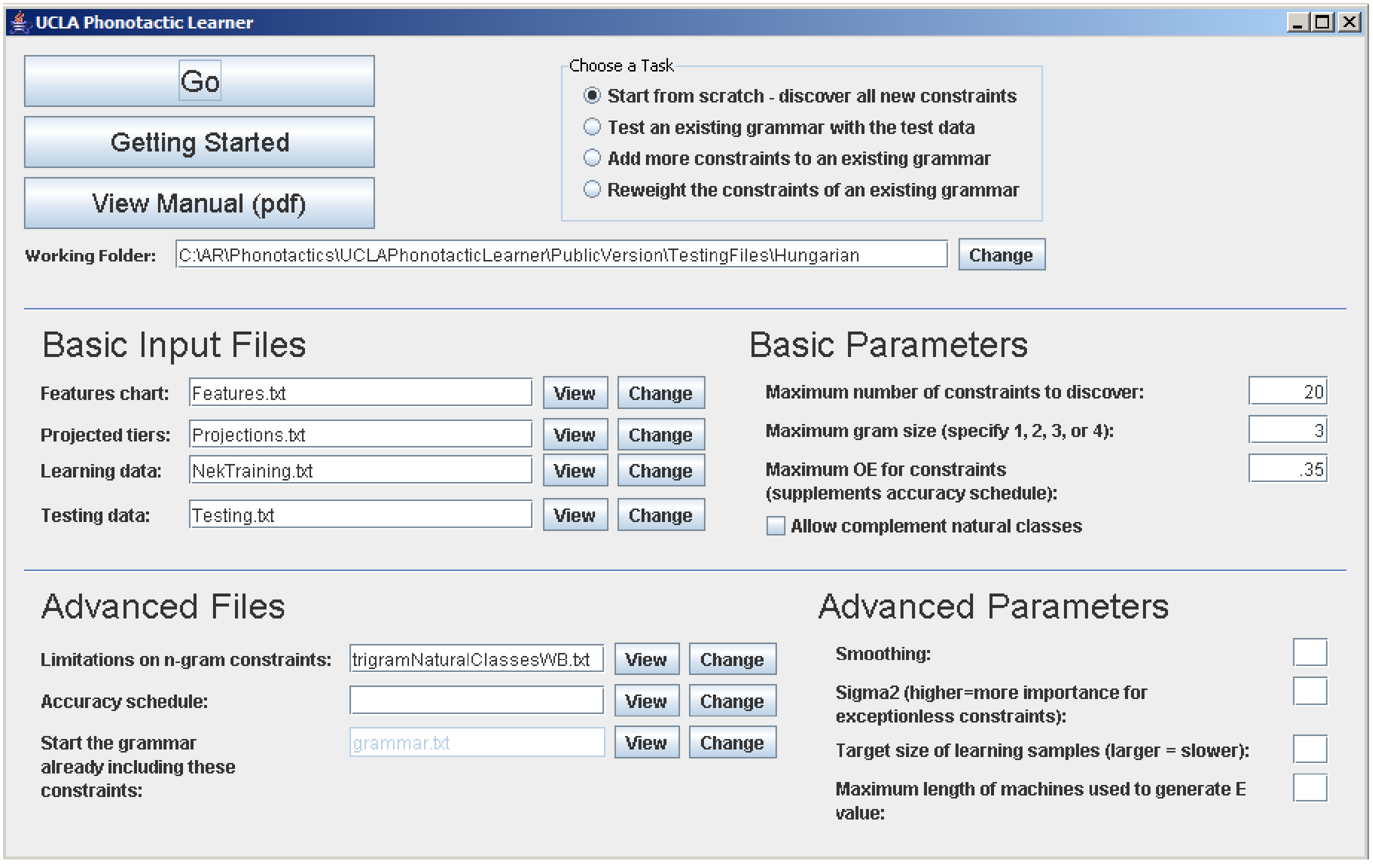
This program itself is hereby licensed under the GNU General Public License, described at http://www.gnu.org/copyleft/gpl.html.

You can still make your file and folder names reasonably legible by using traditional devices, for example, you can name a file *CapitalizeAllWords.txt* or *put_in_underscores.txt*.

4. Launching the program

The program is in a single file called **UCLA_Phonotactic_Learner.jar**. Click on this file to launch the program.

Once it launches, you will see an interface like this:



Choose a Task
[✓ Start from scratch - discover all new constraints]
Test an existing grammar with the test data
Add more constraints to an existing grammar
Reweight the constraints of an existing grammar

Working Folder: C:\AR\Phonotactics\UCLAPhonotacticLearner\PublicVersion\TestingFiles\Hungarian

Features chart: Features.txt
Projected tiers: Projections.txt
Learning data: NekTraining.txt
Testing data: Testing.txt

Maximum number of constraints to discover: 20
Maximum gram size (specify 1, 2, 3, or 4): 3
Maximum OE for constraints (supplements accuracy schedule): .35
Allow complement natural classes

Limitations on n-gram constraints: trigramNaturalClassesWB.txt
Accuracy schedule:
Start the grammar already including these constraints: grammar.txt

Smoothing:
Sigma2 (higher=more importance for exceptionless constraints):
Target size of learning samples (larger = slower):
Maximum length of machines used to generate E value:

To run the program, you will need to do the following.

4.1 Construct your input files

The minimum you need is the following, each of which has a box and a **Change** button on the interface to permit you to fill it in. Set up these files and put them into the folder for your simulation.

- **Learning data**: file containing learning data
- **Features chart**: file containing features and segments

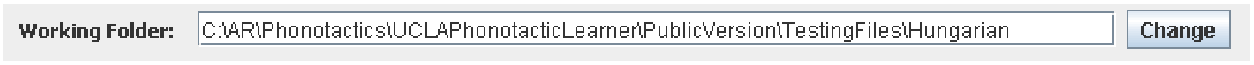
You can give them any name you like, though it would be normal to make the file suffix be **.txt**, to indicate that they are plain text files. The file specifications are given above (section 2), and you can see examples by looking at the sample files for Shona phonotactics (**Features.txt**, **LearningData.txt**) that come with the program.

When you construct your learning data file, it is wise to make sure that the frequencies total above 3000. You can simply edit your file (e.g. with a spreadsheet program) to insure this—note that if the items differ in their frequency, it's only the *proportional* difference that matters, so you can freely multiply all frequencies by a constant.

The program will warn you if your data total less than 3000.[8]

4.2 Tell the program where your files are

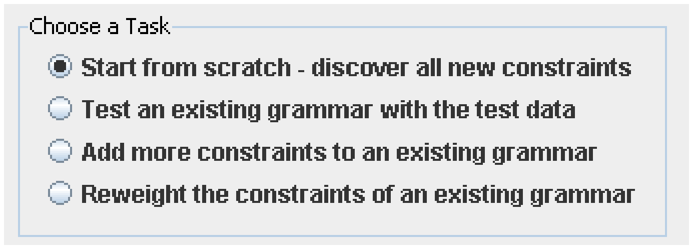
Fill in the box labeled **Working folder** with the folder where you're keeping the files for this particular simulation (you can use the **Change** button to do this easily).


Working Folder: C:\AR\Phonotactics\UCLAPhonotacticLearner\PublicVersion\TestingFiles\Hungarian

You must keep *all* of your input files in this folder, and not scattered across your hard disk, so if you want to use (say) a particular features file for more than one simulation, you must put a copy in each folder that uses the file.

4.3 Specify a task

The default task is simply to examine the learning data and construct a grammar from scratch. To do this, check the default box at the top of the interface.


Choose a Task
- [●] Start from scratch - discover all new constraints
- [ ] Test an existing grammar with the test data
- [ ] Add more constraints to an existing grammar
- [ ] Reweight the constraints of an existing grammar

There are three other possible tasks, discussed section 6 below.

4.4 Start

Click the "Go" button.

A text window reporting the program's progress will pop up. Its content is somewhat technical, but you will be able to discern first some preliminary work, then the learning of the constraints (note: they are numbered starting with zero), then the final precision calculation of

[8] It will issue a really dire warning if the total is 100 or less; the behavior of the program with very short learning-data files is rather unpredictable.

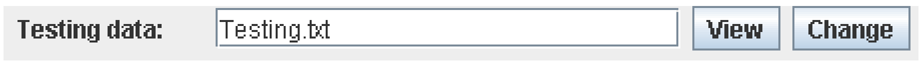
the weights (see Hayes and Wilson (in press), section 3.3.2). This material is kept for later inspection in the output file **ProgramTrace.txt**.

5. Controlling the program: optional elements

5.1 Testing data

You will probably want to include a file of **testing data**, to which the program will assign predicted scores, making it possible to evaluate the grammar. The file specification is given in section 2 above, and you can see an example by looking at the sample Shona file **Testing.txt** that comes with the program.

Enter the name of your testing data file in the appropriate box. You can do this with the **Change** button.



If you don't include a testing data file, then the program will automatically make up its own: it will test all of your training data.[9]

5.2 Projections

The system has more power to discover generalizations if you create *projections*, which are substrings selected from the full string of phonemes for each learning form. For example, if you select the substring consisting solely of vowels, it will make possible the learning of generalizations about vowel harmony more feasible. For full discussion of projections, see Hayes and Wilson (in press), sections 6 and 7.

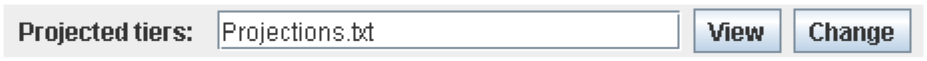
To create a projection, create a text-format file that looks like this (example is from the Hayes/Wilson simulation for Shona vowel harmony):

```
Vowel +syllabic : high, low, back, word_boundary  Grams: 3
```

- "Vowel" is the name of the projection, used to identify it in the learned grammar. Projection names should consist solely of letters, and should include no spaces.
- "+syllabic" is the feature that defines the projection; that is, the projection consists of just the [+syllabic] sounds. To more than one feature, separate the feature values with commas; so, for example, "+syllabic,+high" would create a projection of high vowels.
- "high, low, back" are the projected features; i.e. the projection consists of sequences of underspecified matrices that include just these three features.
- "Grams 3" is explained below in section 5.3.

[9] This is less trivial than it might sound: rare phonological sequences in the learning data often result in penalty scores, reflecting their underrepresented status.

The program will make use of your projections file if you enter the location of the file in the box labeled **Projected tiers**. You can leave this box blank, in which case the program will work only with complete phonological strings.


Projected tiers: Projections.txt

5.3 Specifying gram size

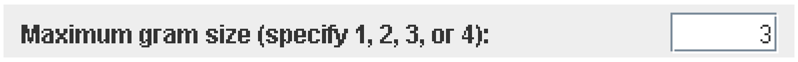
"Gram size" means the number of feature matrices that appears in a constraint. For example, the gram size of the constraint *[+nasal][–voice] is two.

The maximum gram size of constraints will be 3 unless you specify otherwise. (However, it's quite possible that none of the constraints learned will actually be this long—the system is set up that if it can explain the data using shorter constraints, it will.)

Generally, if you specify a high value for this parameter, the system will take longer—perhaps even unfeasibly long (see Hayes and Wilson, section 6.2) to finish learning.

There are several ways to specify maximum gram size.

- If you do nothing, it will be set at 3.
- Or you can fill in a value (1, 2, 3, or 4) on the interface, which will override the default value.

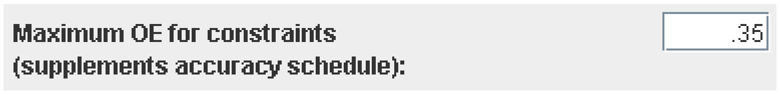

Maximum gram size (specify 1, 2, 3, or 4): 3

- Or, if you are using projections, you can separately specify the gram size separately on each projection, using the format given in the previous section.

5.4 Specifying maximum O/E

The value "O/E" ("observed over expected") is a measure of constraint effectiveness, the ratio of the number of times a constraint is violated in the learning data, to the number of times it would be *expected* to be violated, based on the grammar as learned so far. For discussion, see Hayes and Wilson, section 4.2.1.

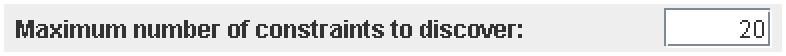
Powerful, important phonotactic constraints tend to have low O/E values. The program searches for these constraints first. But, unless you specify otherwise, it will continue to search for constraints, even rather nonuseful ones, essentially to the bitter end—i.e. up to the value of 1 for O/E. However, you can tell it to stop earlier by filling in a lower maximum O/E value in the appropriate window:


Maximum OE for constraints (supplements accuracy schedule): .35

In this case, the program will give up after having explored only the constraints that do better than this O/E criterion. Most of the simulations in Hayes and Wilson's paper were set up to stop at 0.3.

5.5 Maximum number of constraints to discover

This is a different way of making sure your simulation will terminate in reasonable amount of time. Note that it is not always necessarily to impose such a maximum—often, there are only a limited number of constraints that meet the specified O/E threshold, and the program will halt after it has discovered them.


Maximum number of constraints to discover: 20

5.6 Allowing complement natural classes

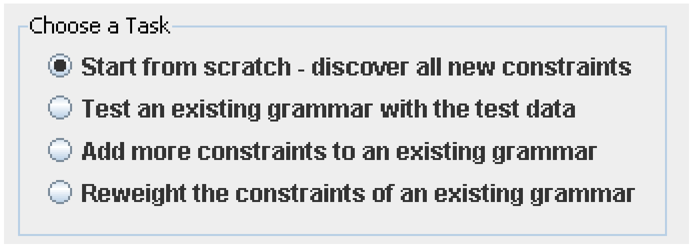
When you check this box, the program will (if appropriate) learn "implicational" constraints, of the type: "any segment occurring in (some environment) must belong to some particular natural class." Formally, this means that the constraint includes a matrix, designated with "^", meaning "any segment which is not this". Example: *[^–voice,+ant,+strid][+nasal], as applied to English, means "assess a violation for any nasal preceded by a segment other than [s]." For further discussion, see Hayes and Wilson, section 4.1.1.

5.7 Long words

If your words are particularly long, you'll need to adjust a parameter to enable it to run. Go to the box labeled "Maximum length of machines used to generate E value", and enter the length in segments of your longest word.

6. Using the program with preexisting results

The basic task menu at the top of the interface:


Choose a Task

- [✓ Start from scratch - discover all new constraints]
- Test an existing grammar with the test data
- Add more constraints to an existing grammar
- Reweight the constraints of an existing grammar

allows you to operate the program with some information already in place.

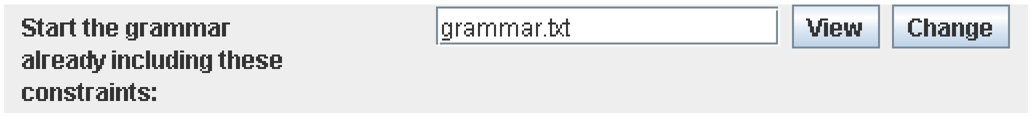
6.1 Starting from scratch: learn the constraints and weights

This is covered above.

6.2 Testing an existing grammar with the test data

Suppose you have already learned a complete grammar, including weights, and want to find out what it predicts for a new data set. To do this:

- Make sure your grammar is in the input folder for the simulation as a whole. (You will usually have to copy it "upward" from the **output** folder, where it was created, to the mother folder.)
- Include the file name for your grammar in this box:



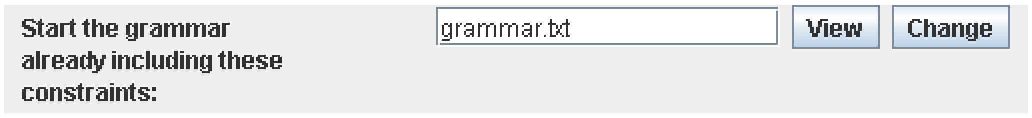
- Include the file name for your testing data in the testing data box.

Note: if you want to test a grammar, but it doesn't already have weights, you must select a different option, namely **Reweight the constraints of an existing grammar**, described under §6.4 below.

6.3 Add more constraints to an existing grammar

Suppose you've learned a grammar containing, say, 50 constraints, and want to add more. To do this:

- Make sure your grammar is in the input folder for the simulation as a whole. This means you'll have to take it from the output folder and put a copy in the input folder.
- Include the file name for your grammar so far in this box:



- Make sure that the current settings won't halt learning where it already is. This means that **Maximum number of constraints to discover** has a number higher than what is already in your grammar, and that your O/E threshold (see section 5.4 above) is generous enough to allow more constraints to be found.

The program will learn new constraints and add them to what you specified. Note that if the initial batch of constraints included weights, they will be ignored; a complete set of new weights will be assigned to the expanded grammar as a whole.

6.4 Reweight the constraints of an existing grammar

Check this box in the following situation: you have *n* predetermined constraints, and want the program simply to find the best weights for them and conduct a blick test. The weights already in the grammar file will be overwritten.

It is not necessary for your constraints to have been learned by the program—provided you format them with care, the constraints can be ones that you made up yourself. You can find a sample format in the file **grammar.txt**, which comes with software, part of the Shona simulation. You may find it helpful if you are creating a grammar to first run the program in conventional model, creating the file **NatClassesFile.txt**. This will tell you what natural classes you can use in making up constraints, and the particular feature values you should use in formulating them.

7. Other control parameters

7.1 Advanced files

7.1.1 Limitation on n-gram constraints.

Specify any list of feature values, one value (e.g. +syllabic) per line. Then, for constraints with 3 or 4 matrices, there will be a complexity limit: all but two of the matrices will be limited to containing feature values from your list. This is useful in guiding the program towards more sensible hypotheses for the longer constraints. See Hayes and Wilson, sections 4.1 and 5.1.

7.1.2 Accuracy schedule

Specify your own ascending sequence of O/E threshold values. One number, in the interval $0 < x <= 1$, per line. The default in the program is .001, .01, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1. For discussion, see Hayes and Wilson, section 4.2.1.

7.2 Advanced parameters

7.2.1 Smoothing

Added to O in computing O/E. Default is 0.5.

7.2.2 Sigma

Defines the Gaussian prior that prevents overfitting (Hayes and Wilson, section 3.1.1). Default is 1; you can use a larger value to get a closer (and perhaps: dangerously, overfittingly closer) fit to the learning data.

7.2.3 Target size of learning samples

Increasing this (above about 3000) will slow learning but make it more accurate.

7.2.4 Maximum length of machines used to generate E(xpected) value

Make this longer if you learning data has longer words; it should roughly match the longest words length as measured in segments.

8. More on phonetic symbols and features

8.1 *Legal symbols*

Phonetic symbols for the program may have more than one character (e.g., you can use “ch” or “CH” for [tʃ]). They may not begin with digits, nor may they contain apostrophes, colons, or the symbol @. There may be other restrictions as yet undiscovered! It is probably wisest to use ordinary letters for your symbols.

8.2 *The natural classes file*

When you run the program, it will create an output file (in the output folder) called **NatClassesFile.txt**. For example, the natural classes file for the sample Shona simulation begins like this:

```
Novel of size 1: +word_boundary         #
Novel of size 1: -word_boundary         N,a,b,bh,bv,ch,d,dh,dz,dzv,e,
                                        f,g,h,i,j,k,m,n,o,p,pf,r,s,sh,sv,t
                                        ,tsv,u,v,vh,w,y,z,zh,zv
Novel of size 3: +cont,-voice,+distr    sh
Novel of size 2: -continuant,-anterior  ch,dzv,j,tsv
Novel of size 2: -del_rel,+anterior     d,dh,t
Novel of size 2: +voice,+coronal        d,dh,dz,dzv,j,z,zh,zv
Novel of size 2: -continuant,+labial    b,bh,bv,p,pf
Novel of size 1: +back                  a,o,u,y
Novel of size 2: +del_rel,+voice        bv,dz,dzv
Novel of size 2: +approximant,+labial   v
```

The featural descriptions can be useful if you wish to try constructing your own constraints (to include in a handcrafted grammar file). Also, for any sort of work using feature systems it seems a good idea to know what natural classes the system defines.

Note that under a particular segment inventory, many of natural classes will have multiple, synonymous featural expressions in handcrafting a constraint you would want to use the same featural expressions used by the program.

8.3 *Adequacy in feature systems: specifying each segment uniquely*

Another important aspect of feature systems for purposes of this program is their *ability to designate each segment uniquely*. When one is designing a feature system, it is surprisingly easy to omit the values needed to do this. For instance, the following schematic system provides no unique designation for either labial /p/ or velar /k/, as each one has a subset of feature values of labial-velar [k͡p].

	[p]	[k͡p]	[k]
[labial]	+	+	0
[dorsal]	0	+	+

This means that, if the program were to use these features, it would be unable to formulate constraints applying specifically to just [p] or just [k]. The alternative is to provide a richer specification, such as the following:

	[p]	[k͡p]	[k]
[labial]	+	+	–
[dorsal]	–	+	+

The program is designed to watch for such cases and give you a warning where they occur.[10] Should you wish to do phonotactic learning with an incomplete feature system, this is still possible (though perhaps not advisable).

9. Mapping from model predictions to experimental data

For purposes of matching up model predictions with experimental data, Hayes and Wilson (2008) give the following equations:

- predicted-rating(x) = P*(x)$^{1/T}$, where
- P*(x) = exp(– h(x)) and
- h(x) is the "score" output by the model

They use best-fit values for *T*. Finding this value is best done with a sophisticated statistics package such as *R* (http://www.r-project.org/), but a quick alternative is to make a spreadsheet whose entries look like this:

	A	B	C	D
1	scores (from model output)	Native speaker preferences (scaled 0-1)	7.3	=CORREL(B2:B11,C4:C11)
2	0	0.818	=EXP(A2/-C$3)	
3	4.844	0.667	=EXP(A3/-C$3)	
4	0	0.606	=EXP(A4/-C$3)	
5	4.843	0.576	=EXP(A5/-C$3)	
6	4.898	0.455	=EXP(A6/-C$3)	
7	4.843	0.424	=EXP(A7/-C$3)	
8	6.661	0.394	=EXP(A8/-C$3)	
9	10.868	0.394	=EXP(A9/-C$3)	
10	10.754	0.394	=EXP(A10/-C$3)	
11	12.913	0.394	=EXP(A11/-C$3)	

[10] Specifically, it checks the list of natural classes to see if every single segment forms a (degenerate) natural class.

You can then hand-adjust the value in C1 (which is *T*) until the correlation value shown in D1 reaches a maximum.

10. Troubleshooting and queries

In pre-testing, we have succeeded in running this program on Windows machines (both the XP and Vista versions of Windows), as well as on a Macintosh running OS X, version 10.4.10. The program has bugs that hopefully will be addressed when the original programmer returns to service.

- If you **can't get the program to start up**, a likely problem is that your copy of Java is out of date—it needs a fairly recent (1.6 or later) version of Java. Download a new Java (free from http://www.java.com/en/download/index.jsp) and install it. Older Macs may have trouble running a new enough Java to work; see fn. 6 above.

- Possible **error on startup**: the program pops up a message saying "There's a problem writing the following file…". This is due to a bug in the program, but there is a workaround. Go into the folder where your input files are located and make a new folder, called **output**, inside it. (The bug evidently arises because Java is unable to make this new folder itself.)

- Program starts, but complains, "**The system cannot find the file specified**." The probable cause is an incompatibility between Java and Windows: Windows allows spaces in file names and folders, but Java does not. Rename all associated folders and file names, then try again.

- Program **runs, but doesn't produce an output**. The most likely problem is that the input files aren't in the exact form the program expects (we work on auto-detecting these, but this is an ongoing process).

  One way to detect errors is to look at the program output file called ProgramTrace.txt. Look at the very end; this often gives a constraint that the program couldn't deal with because it it misformatted.

  Three common file-format errors:

  - A particularly common error is using upper case for lower. Java, alas, was made case-sensitive.
  - If you are making up your own constraints, it is essential that they use natural classes taken from the file **NaturalClassesFile.txt**, which you can generate by running the program in its "generate constraints" mode. It is located in the **temp** folder, which will be a daughter folder of the folder in which you keep your input files.
  - Also, if you are making up your own constraints, it is essential that the last constraint be on the last line of the file. This must be a complete line (hit Enter), and there must be no blank lines after it.

- Program runs, generates an output, and gives every constraint a weight of one. Most likely, you've asked it to test a preexisting grammar that doesn't have any weights in it. Rerun the program with the "reweight" option.

- Program tries to run a big simulation and hangs up. You may need more memory, which can be done by launching the program with a command line (in Windows: open a "DOS window" and paste the command line in, or put the command line in a file with the name RunMe.bat). The command line should read:

```
java -Xms650m -Xmx650m -jar UCLA_Phonotactic_Learner.jar
```

For queries about the program, or for source code, please contact Bruce Hayes at bhayes@humnet.ucla.edu. If your simulation won't run, it may be helpful to send the input files.