Manual: The Gradient Lexicon and Phonology Learner

Claire Moore-Cantwell

February 6, 2023

1 Quick Start

To begin using GLaPL, you will need:

- A python terminal
- `learner.py`
- `config.gl` (more in **??**)
- `features.txt`
- an input file

Put all of these files in the same folder on your computer, and point your python terminal to that folder. Then, run the following commands:

```
import learner as l
g = l.Grammar()
```

You should see output confirming that your file was read in, and listing your constraints and starting weights (generally all zeros).

```
In [5]: g = l.Grammar()

Your input file contains candidates, therefore candidates will not be generated for you.

Your input file contains input frequency information (it's the column labelled, somewhat
opaquely, 'tab.prob')  Learning will therefore proceed according to this frequency-weighting.
 Note that you can turn off frequency weighting by...
Training from file: dutchinput
Feature set: features.txt
Constraint violations come only from input file
Candidates will not be generated

Your constraints and starting weights:

Id-Length STW *VVC
    0      0   0

In [6]:
```

If you see this, you are ready to run `g.learn()`. This function takes two numbers as input. The first is the number of learning iterations per epoch, and the second is the number of epochs. So, if I use 100 for both parameters, I'll learn a total of 10,000 (100*100) times. After each epoch, you'll get an error rate printed to the command line, and various information printed to output files. Try it:

```
1 g.learn(100,100)
```

If you want the learner to save information often during learning, increase the number of epochs and decrease the number of iterations per epoch, perhaps like `g.learn(10,1000)` or even `g.learn(1,10000)`. If you want the learner to run, faster decrease the number of epochs and increase the number of iterations: `g.learn(1000,10)` or `g.learn(10000,1)`.

As learning progresses, you will see a printout at each Epoch of an error rate. When learning is finished you will see the final constraint weights

```
Epoch 92: 16.0 % errors
Epoch 93: 19.0 % errors
Epoch 94: 23.0 % errors
Epoch 95: 22.0 % errors
Epoch 96: 22.0 % errors
Epoch 97: 18.0 % errors
Epoch 98: 20.0 % errors
Epoch 99: 17.0 % errors
Epoch 100: 22.0 % errors

 Final constraint weights:

 Id-Length    STW    *VVC
    2.69      2.63   2.51
predicting
Saving output predictions to output.txt

In [4]:
```

If you want to examine either the Sum Squared Error (SSE) or log likelihood of your learned grammar's predictions, you can run:

```
g.SSE()
g.logLikelihood()
```

After learning, you will see output files appear in the folder your python terminal is pointed to.

- `output.txt` - This is a plain text file, which contains predicted tableaux for every datum in your input file. You should copy-paste it into Excel or another spreadsheet program to read it easily.

  It should look something like this:

Using lexemes	glas							
input	lexeme(s)_used	candidate	obs.prob	pred.prob	H	Id-Length	STW	*VVC
						0	0.95	0.73
glas		glas	1	0.932289366	-7.4258	0	1	0
glas		glaas	0	0.067710634	-10.0482	1	0	1
Using lexemes	glas	en						
input	lexeme(s)_used	candidate	obs.prob	pred.prob	H	Id-Length	STW	*VVC
						0	0.95	0.73
glas_en		glas_en	0	0.0311197	-7.4258	0	1	0
glas_en		glaas_en	1	0.9688803	-3.9875	1	0	0
Using lexemes	slot							
input	lexeme(s)_used	candidate	obs.prob	pred.prob	H	Id-Length	STW	*VVC
						0	0.95	0.73
slot		slot	1	0.931960372	-7.4258	0	1	0
slot		sloot	0	0.068039628	-10.043	1	0	1
Using lexemes	slot	en						
input	lexeme(s)_used	candidate	obs.prob	pred.prob	H	Id-Length	STW	*VVC
						0	0.95	0.73
slot_en		slot_en	0	0.030963296	-7.4258	0	1	0
slot_en		sloot_en	1	0.969036704	-3.9823	1	0	0

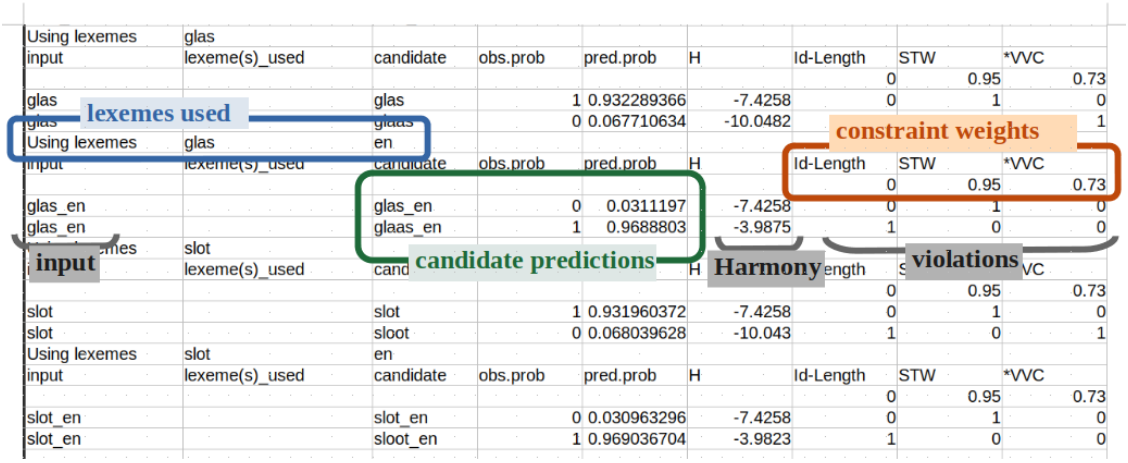
  `output.txt` contains one tableau for each input in your input file.

  IMPORTANT These may not be the only options for generating each input's tableau! To re-generate predictions, run

```
g.predict()
```

-

More details:



Using lexemes	glas							
input	lexeme(s)_used	candidate	obs.prob	pred.prob	H	Id-Length	STW	*VVC
						0	0.95	0.73
glas		glas	1	0.932289366	-7.4258	0	1	0
glas		glaas	0	0.067710634	-10.0482	1	0	1
Using lexemes	glas	en						
input	lexeme(s)_used	candidate	obs.prob	pred.prob	H	Id-Length	STW	*VVC
						0	0.95	0.73
glas_en		glas_en	0	0.0311197	-7.4258	0	1	0
glas_en		glaas_en	1	0.9688803	-3.9875	1	0	0
	slot							
	lexeme(s)_used	cand			H	ength	S	VC
						0	0.95	0.73
slot		slot	1	0.931960372	-7.4258	0	1	0
slot		sloot	0	0.068039628	-10.043	1	0	1
Using lexemes	slot	en						
input	lexeme(s)_used	candidate	obs.prob	pred.prob	H	Id-Length	STW	*VVC
						0	0.95	0.73
slot_en		slot_en	0	0.030963296	-7.4258	0	1	0
slot_en		sloot_en	1	0.969036704	-3.9823	1	0	0

lexemes used The same input and candidates can be generated from different combinations of lexemes (listed vs. composed for example). This row tells you which lexemes were used. It's one per cell

candidate predictions Candidates will either match what you put in your input file, or be generated for you. The next two columns are predicted probability (obs.prob) and predicted probability in this tableau (pred.prob)

constraint weights The constraints and weights that were used to generate this particular tableau - they might be different across inputs, if you used lexically indexed constraints

- `weights.txt` Each column is a constraint, each row is that constraint's weight at the end of each epoch
- `lexCs.txt` Lists each clone of each constraint, and which lexical items are indexed to each clone
- `listing.txt` A record of each time a new lexical item was listed. Columns are (1) which lexemes were combined (2) the segments in the listed form (3) the learning timestep (not Epoch) at which the new lexical item was created
- `PFCs.txt` (ignore for now)

2 Details of Theories Implemented

2.1 Preliminaries

2.1.1 Theories of Eval

2.1.2 Perceptron Learning

- basic input to get frequency matching - within-word variation - across-word variation

Getting the observed output candidate:

- threshold vs. sampling

2.2 UseListed

UseListed is a theory that many researchers implicitly or explicitly use as a default approach to exceptionality in phonology, but as far as I know no learning model has been developed. The theory can easily be summarized as "We memorize exceptions."

This theory assumes two things:

1. We memorize morphologically complex words, at least sometimes
2. We can therefore produce morphologically complex forms in two ways:
    - **Composed** forms are created by accessing multiple morphemes and realizing them together according to the morphological and phonological grammar
    - **Listed** forms are accessed whole, and realized according to their lexical entry and the phonological grammar

A couple of examples:

In Tagalog, morphemes often undergo 'nasal substitution' in which the final nasal of a prefix coalesces with the initial consonant of the root, forming a single sound.

(1) a. **d**iníg /pa**ŋ**+**d**iníg/ → pa**n-d**iníg ASSIMILATION
audible *sense of hearing*

b. **d**aláŋin /i+pa**ŋ**+**d**aláŋin+in/ → ʔi-pa-**n**aláŋin-in SUBSTITUTION
prayer *to pray*

These examples come from **?**. In (1a), the underlying velar nasal assimilates in place to the following stop, but in (1b), the two sounds have completely merged, leaving an [n] with the nasality of the [ŋ] and the place of the [d].

As the example suggests, the phonological shape of the individual words cannot completely predict whether the nasal will substitute or just assimilate. There are lexical trends, but no categorical rules. Importantly, individual words do not vary, the variation is entirely from word to word.

A UseListed approach to this pattern would say that both pa**n-d**iníg and ʔi-pa-**n**aláŋin-in are memorized as their own lexical entries. When speakers want to say the meaning *'sense of hearing'*, or *'to pray'*, they access those meanings directly in their lexicon, and produce them according to the idiosyncratic phonological form that is listed - one with substitution, and one without.

A second example:

English comparatives come in two forms, the 'periphrastic', using *more*, and the 'morphological', using *-er*. Both are available for most adjectives: *fouler* and *more foul* are about equally attested in a corpus, for example. However, higher-frequency adjectives exhibit idiosyncratic preferences **?**.

(2) a. simpler (96%) ≫ more simple (4%)

b. more stable (98%) ≫ stabler (2%)

While a variety of phonological factors condition the choice between these two versions of the comparative, *simple* and *stable* are similar on all relevant dimensions[1]. English speakers therefore must memorize that the meaning *simple* + COMPARATIVE is pronounced with *-er*, while *stable* + COMPARATIVE is pronounced with *more*.

Unlike in the case of Tagalog nasal substitution, many individual words do vary, and low-frequency words seem to follow the predictions of a probabilistic grammar. Even extreme cases like those in (2) still exhibit the minority form occasionally. One was to imagine what is happening here is that speakers have memorized the majority form for both words, but every so often they fail to use that memorized form and compose the comparative on the fly instead.

If we have both a **Composed** form and a **Listed** form available, there are many different ways we could decide between them. GLaPL can learn using any of these.

Option 1: Always use the listed form, if available. OR use the listed form with some static probability

Option 2: Do whatever is easier. If it is easy to find the listed form, use it, but if it is easier to compose, use the composed form.

Option 3: Directly compare the Composed and Listed derivations in the same tableau.

To do **Option 1**, set `p_useListed` to 1 or less. That will be the probability that a PREDICT step will use the listed form, if available.

To do **Option 2**, set `p_useListed` to 2. There are many possible ways to implement 'easier' here, but for now the implementation is based on lexical frequency. The probability that the listed form will be chosen is related to both the frequency of the listed form itself and the frequency of the lowest-frequency morpheme in the composed form.

[1]They are 2-syllable words with initial stress, ending in [l], and are about the same lexical frequency.

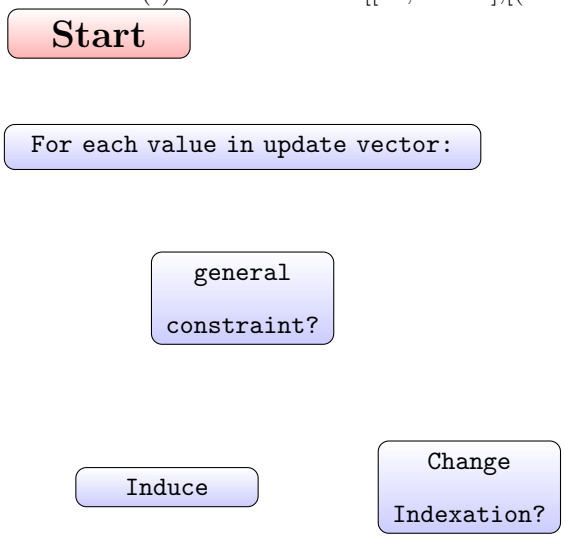
$$P(listed) = \frac{freq_{listed}}{freq_{listed} + min(\{freq_{composed1}, freq_{composed2} \dots freq_{composedx}\})}$$

Hay 2003, Forster and Chambers 1973, Whaley 1987

Note: A lexeme name cannot start with a digit. Can have a digit inside tho.

2.3 Lexically Indexed Constraints

In order to do actual locality restrictions, you need to write your constraint functions to return the indices of the violation(s). make it a list: [[v1,v2...vn],[(start1,end1),(start2,end2)...(startn, endn)]]



2.4 UR-constraints

2.5 Representational Strength Theory

2.6 Gradient Symbolic Representations

3 Supplementary details

3.1 How Tableaux are constructed

Tableaux are constructed in real time during learning. At each learning iteration, a tableau is constructed for that input-output pair. Here is a breakdown of how the `Grammar.makeTableau()` function operates.

`Grammar.makeTableau()`'s first argument is a **datum**, which is an entry in `Grammar.trainingData.learnData`, a list.

(3) **datum:** [[`lexeme`$_1$, `lexeme`$_2$, ... `lexeme`$_n$], surface string, input]

The surface string and input (also a string) both come directly from the input file. The surface string comes either from your 'candidate' column, or from your 'surface' column, if you have one. The input string comes from your 'input' column.

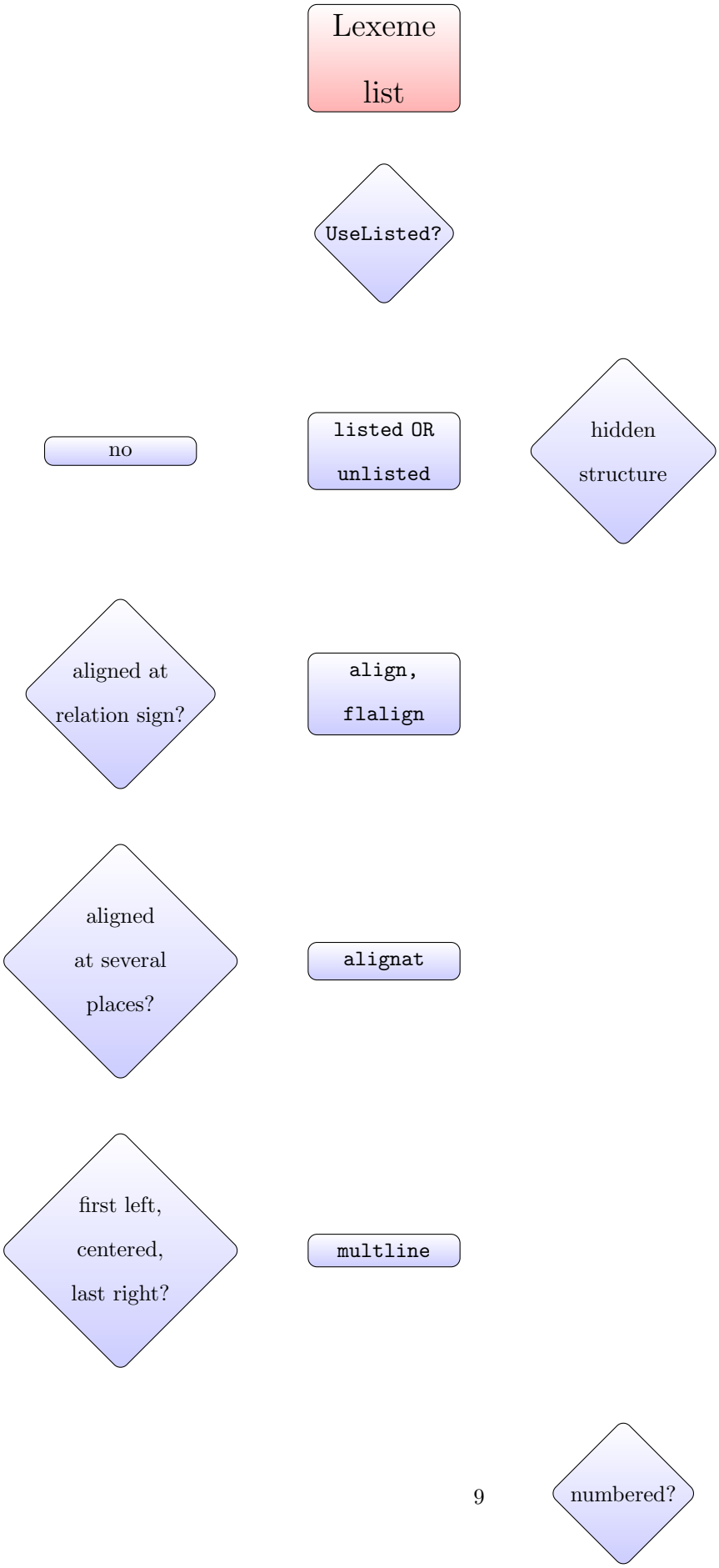
datum's first entry is a list of `lexeme` objects involved in this learning datum. Note that if you are using UseListed, these won't always be the actual lexemes used to generate the tableau - the function may try to use the listed form instead.

The first step in creating a tableau is to generate faithful candidates from the input lexemes. In most cases, this will be a single faithful candidate, but not always. The inner function `lexemesToFaithCands()` accomplishes this.

In order to directly test `lexemesToFaithCands()` behaviour on a list of lexemes, run `makeTableau()` with the option `textFcs=True`.

```
import learner as l
g = l.Grammar("lexemes_sampler.txt", l.Features("features.txt"))
g.makeTableau([lexemes],testFcs=True)
g.makeTableau([l.exlex_petit(),l.exlex_ami()],rich=True,testFcs=True)
```

You can find "lexemes_sampler.txt" in the manual_examples folder


Lexeme list
UseListed?
no
listed OR unlisted
hidden structure
aligned at relation sign?
align, flalign
aligned at several places?
alignat
first left, centered, last right?
multline
numbered?

3.2 Dealing with hidden Structure

Hidden structure problems can arise in any theory you are using, but UR-constraint models, UseListed with the UseListed constraint, and GSR models with segment blending all require hidden structure to operate. GLaPL follows the principles of Jarosz (2013)'s Expectation-Maximization algorithm ot learn hidden structure.

Let's look at an example using UseListed. UR-constraints would function similarly.

(4) Tableau for made-up multi-morphemic word, based on Pater et al, 2012.

				*VTV	Ident-Voice	UseListed		
			weights:	1	2	1	H	p
Composed	a.	/mot/+/a/	→ mota	1	0	1	-2	0.11
	b.	/mot/+/a/	→ moda	0	1	1	-3	0.04
Listed	c.	/moda/	→ mota	1	1	0	-3	0.04
	d.	/moda/	→ moda	0	0	0	0	0.81

In (**??**), candidates a. and b. come from a UR that concatenates two morphemes' URs, while c. and d. come from a single UR, the listed lexical item that combines the meanings of Mot and -a. In this example, the combined form of the two has a voiced stop instead of a voiceless one. Candidates a. and c. have different violation profiles: The [d] is a faithfulness violation for the composed form, but the [t] is a faithfulness violation in the listed form. UseListed also assigns violations differently. The same is true for candidates b. and d. Each of these pairs sounds identical, however. Suppose the correct form is [moda]. If on a given learning iteration, we sample either candidate b. or candidate d., this prediction would count as correct. This would not cause any problem, the learner simply would not update. However, if candidate a. or c. were sampled, this would count as an error, and we would need to update constraint weights.

Suppose we sampled candidate a. That candidate would be the **predicted** candidate for perceptron update, but in order to update we need to compare it to an **observed** candidate. b. and d. both correspond to the actual observed output, so either one could plausibly be used as the **observed** candidate. How do we choose between them?

One option, of course, is simply to designate one of them correct. Perhaps we could decide that we want the learner to always learn the Composed form, so a. is always the **observed** candidate. However, in this case we are determining the hidden structure in advance instead of allowing the learner to choose the most optimal hidden structure. In the UseListed case, we would be forcing the learner not to use listed forms. Note: if you do want to run a simulation like this with GLaPL, you can just turn lexical listing off to achieve

the same effect.

We could also sample at random between the two correct candidates. The trouble with this approach is that it fails to take into account what the learner already knows, forcing it to essentially reconsider hypotheses that it has already rejected. Much better would be to take into account the weights the learner has already arrived at, which predict that d. should be much more likely than b.

To do this, we create a mini-tableau considering only candidates whose surface form corresponds to the observed correct output form in our data.

(5) To get the observed form, consider only candidates whose surface form corresponds to the correct output.

			*VTV	Ident-Voice	UseListed		
		weights:	1	2	1	H	p
b.	/mot/+/a/	→ moda	0	1	1	-3	0.05
d.	/moda/	→ moda	0	0	0	0	0.95

In (**??**), candidate d. is vastly more likely to be sampled than b. 95% of the time, we will use that one as the **observed** candidate.

To see why it is so important to sample according to the current state of the grammar, consider another type of hidden structure: foot structure in a stress system.

(6) Tableau for made-up multi-morphemic word, based on Pater et al, 2012.

		Align-R	Align-L	Iamb	Trochee		
	weights:	0.1	0.2	0.2	3	H	p
a.	(patá)ka	1	0	0	1	3.1	0.03
b.	pa(táka)	0	1	1	0	0.4	0.41
c.	(páta)ka	1	0	1	0	0.1	0.55
d.	pa(taká)	0	1	0	1	3.4	0.20

In (**??**), candidates a. and b. have the same surface form: [patáka]. They differ only on whether that stress pattern is the result of a right-aligned trochee, or a left-aligned iamb. Other words in the language would not have such an ambiguity. A two-syllable word, /pata/, would either be stressed on the first syllable, (páta), which could only be a trochee, or the second (patá), which could only be an iamb (assuming two syllable feet).

So, assuming that the relatively high weight on Trochee comes from learning on forms like this in the language, we want the learner to take this into account in choosing a foot structure for patáka. Suppose the

learner predicted candidate c. to be the winner, and chose candidate a. as the **observed** candidate. The update would then look like this:

(7) Update vector for (páta)ka > (patá)ka. (Learning rate 0.01)

		Align-R	Align-L	Iamb	Trochee
	weights:	0.1	0.2	0.2	3
predicted	(patá)ka	1	0	0	1
observed	(páta)ka	1	0	1	0
	new weights:	-	-	0.21	0.29

If this update happens, both Trochee and Iamb will move in the wrong direction. Compare to the update that would happen if we chose candidate b. as the **observed** form:

(8) Update vector for (páta)ka > pa(táka). (Learning rate 0.01)

		Align-R	Align-L	Iamb	Trochee
	weights:	0.1	0.2	0.2	3
predicted	pa(táka)	0	1	1	0
observed	(páta)ka	1	0	1	0
	new weights:	0.11	0.19	-	-

In (**??**), Align-R goes up a little, Align-L goes down a little. This update is preferable to the one in (**??**), because it doesn't backtrack on what the learner has already figured out from other forms, and it *does* get information about the Align constraints from the error, learning that it should prefer right-alignment over left-alignment. Two-syllable forms which would be informative about Iamb vs. Trochee would be uninformative about alignment.

In summary, if we chose the candidate already preferred by the grammar, candidate b., we will get an update that is more informative, and pushes the grammar in a direction compatible with previously encountered data. For this reason, Jarosz (2013) recommends a procedure like that demonstrated in (**??**), and that is what is implemented in GLaPL.

One last thing to note is that hidden structure can arise in a lot of different ways. We've looked at foot structure, and underlying form here, but most Gradient Symbolic Representation analyses will also have a hidden structure component. Here's an example to illustrate:

(9) Sample GSR tableau showing hidden structure. Activity on t_a is 0.7, and on t_b is 0.2

	Max	Dep	Uniformity	NoCoda	*Hiatus	H	p
/petit/$_a$ + /t,z,nami/$_b$	2	2	3	5	5		
a. petiami	0	0	0	0	1	-5	0.57
b. petit$_a$ami	-0.7	0.8	0	0	0	0.1	0.21
c. petit$_b$ami	-0.2	0.3	0	0	0	0.1	0.01
d. petit$_{a,b}$ami	0	0.1	1	0	0	00	0.21

3.3 Simulating Gen

3.4 Applying constraints

4 Input file details

4.1 The input file

Your input file must have at least three columns (tab-delimited)

	input	candidate	obs.prob
input 1	pat	pat	1
	pat	pad	0
input 2	pat_ka	pat_ka	0.7
	pat_ka	pad_ka	0
	pat_ka	pad_ga	0.3
input 3	taka	taka	0.3
	taka	taga	0.7
input 4	taka_ka	taka_ka	0.3
	taka_ka	taga_ka	0.7
input 2	pat_ka	pa_ka	0

- `input` One unique input for each phonological input you want the learner to experience. Inputs may be one or more morphemes, separated by '_'. All rows with a particular input will be treated as the same learning datum, regardless of order

- `candidate` Candidates also contain morpheme boundaries, indicated by the '_'. Candidates should only contain characters (other than the '_') that match what's in your `features.txt` file, if you're using that. If you're assigning constraint violations by function, these strings are what the functions will refer to.

- `obs.prob` This column indicates how often a particular candidate appears for that input. A simple way to use this is as above, where each input's entries all sum to 1. You can also bake in token frequency by putting raw counts into this column.

This input file contains no constraints or violations. If we just used this input file, we would have to assign constraint violations by function. To include constraint violations for each candidate, simply add a column per constraint, with violations for each candidate:

	input	candidate	obs.prob	*Voice	*VTV	Ident-Voice
input 1	pat	pat	1	0	0	0
	pat	pad	0	1	0	1
input 2	pat_ka	pat_ka	0.7			
	pat_ka	pad_ka	0			
	pat_ka	pad_ga	0.3			
input 3	taka	taka	0.3			
	taka	taga	0.7			
input 4	taka_ka	taka_ka	0.3			
	taka_ka	taga_ka	0.7			
input 2	pat_ka	pa_ka	0			

4.2 The `config.gl` file

`config.gl` is where you will set almost all the parameters for learning. You can create multiple versions of this file to save different parameter settings, and you can name them whatever you want. You just then need to call `learner.Grammar()` with the config file you want to use as an argument:

```
g = l.Grammar("myURCsConfig.gl")
```

If no config file is specified, the learner will look for a file called `config.gl` in the same directory your console is pointed to (most likely the directory where the learner is).

Structure of the config file:

1. Each line is a parameter setting of the form `parameterName:  value`
2. Lines that begin with '#' will be ignored

Whitespace is ignored.

Parameters and their values:

`trainingData` A string, the name of your input file. No default value.

`weights` A list of numbers, separated by commas. The weights you would like your constraints to have at the beginning of learning.

If you write a single 0, all weights will be initialized to 0.

`featureSet` A string, the filename of your features file.

For many use cases, this file is irrelevant, but you still have to specify something. I recommend to just use the sample 'features.txt' file that came with this manual.

`addViolations` `True` or `False`. Make sure you spell them the way Python likes, capital first letter and the rest lower case.

If `True`, violations will be added to your tableaux during learning using the constraint functions that your provide.

`constraints` A string, the name of your constraints file. It should be a Python file, ending in .py, but you should leave off the .py here.

`generateCandidates` `True` or `False`. Make sure you spell them the way Python likes, capital first letter and the rest lower case.

If `True`, candidates will be generated for you, using the operations you defined in your constraint file.

`operations` `True` or `False`. Make sure you spell them the way Python likes, capital first letter and the rest lower case.

If `True`, ... COLLAPSE THESE

`learningRate` Float, indicating how much constraint weights should change on update. Default value is 0.01

`decayRate` Float, indicating how much to decay an object at each timestep.

This value will be used to decay PFCs if you are using them, UR-constraints if you use them and decay them, Lexically-indexed constraints if you use them and decay them, etc.

Default value is 0.0001

`threshold` A float between 0 and 1, indicating how much observed probability a candidate must have to count as correct.

Example: Suppose candidates A, B, C with observed output probabilities of 0.6, 0.38, and 0.02, respectively. If `threshold` is set to 0.1, then a predicted value of A or B would count as correct, but not C.

If `threshold` is set to 1, an observed output will be sampled. So, in the above example the observed output would be A 60% of the time, B 38% of the time, and C 2% of the time.

Note that it doesn't make sense to set `threshold` to 0, since this will result in no errors, and therefore no learning.

`noisy` String, "yes" or "no".

If "yes", learning will print out lots of updates to the console as it goes.

`useListedType` String, 4 options:

"hidden_structure" If chosen, the constraint UseListed will be added to the grammar, and tableaux will be constructed with listed and composed forms competing.

"sample_using_frequency" If chosen, either the listed form or the composed form will be chosen, sampled according to the relative frequency of the listed form and the *least frequent* morpheme in the composed form (usually the root if it's a root plus affixes)

"sample_flat_rate" If chosen, the listed form (if available) will be chosen at a flat rate, specified by you in the next parameter.

"none" No lexical listing will be used in learning. (No new lexemes created)

`useListedRate` Float, between 0 and 1. Only used if you chose "sample_flat_rate" above. The chance that the listed form will be used, if available.

Defaults to 1, always use the listed form if you can.

`flip` `True` or `False`. Make sure you spell them the way Python likes, capital first letter and the rest lower case.

Only relevant when inputs exhibit within-item variation in your data.

Example: Suppose you have an input "Cat+Pl", with two observed outputs, [moda] (30%) and [mota] (70%). If `flip` is `True`, then the lexical entry for the listed form "Cat+Pl" will be re-listed each time an error is made. Its value will keep "flipping" back and forth between /moda/ and /mota/.

`simpleListing` `True` or `False`. Make sure you spell them the way Python likes, capital first letter and the rest lower case.

Only relevant when inputs exhibit within-item variation in your data.

If `True`, then a listed lexeme will only be created once, and never changed. With the example from `flip` above, if [moda] happens to be sampled first, then the UR for "Cat+Pl" will be encoded as /moda/. If [mota] happens to be sampled first, then it will be encoded as /mota/.

`pToList` Float, between 0 and 1. If lexical listing is enabled (`useListedType` is not "none"), this is the probability that a complex form will be listed on error. Defaults to 0.75

`nLexCs` 0, any positive integer, or the string "inf".

If not 0, lexically indexed constraints will be learned.

If set to a positive integer, that integer is the maximum number of copies of each constraint that can be learned.

If set to "inf", there is no maximum number of copies.

`pChangeIndexation` Float between 0 or 1. The probability that on error, a lexical item will switch which copy of a constraint it is indexed to in order to improve performance.

`lexCStartW` Float greater than 0. The starting weight of any newly-induced lexically indexed constraints.

`PFC_type` String, 3 options:

"full" PFCs will be induced based on generated candidates.

"partial" PFCs will be induced based on given candidates. They will not use features, but will simply prefer a specific candidate over all the others. Refer to the Representational Strength Theory section to see how to set up your input file so this makes sense.

"none" No PFCs will be induced.

`PFC_lrate` Positive float. The learning rate for PFCs - how much their weight will change on error.

`PFC_startW` Float greater than 0. The starting weight of any new PFC induced.

`activityUpdateRate` Positive float. If using GSRs, the amount by which a segment's activity value will be updated on error.

5 Classes and Methods

6 Files that should come with this manual

features.txt, a sample features file for MAE

config.gl with default values