Manual for the GSR/RST learner

last edited: 10 February, 2022

highlighted things are statements that may/will become untrue as the program fills out

flagged things may change later, and need to be double-checked

TO DO's are things that need to be implemented. Some are essential but most are nonessential, or extensions of functionality

The **Features** class:

to use the **Features** class, you will first need to create a features text document. It should be tab-delimited, and formatted as follows:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| segment names | Feature names | | | | | | |
|  | consonantal | sonorant | continuant | high | low | voice | nasal |
| t | 1 | 0 | 0 | x | x | 0 | 0 |
| n | 1 | 1 | 0 | x | x | 1 | 1 |
| a | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| u | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| ... | ... | | | | | | |

Feature values can be any character at all. Here, we are using 1's for [+ feature] and 0's for [- feature]. You can also use '+' and '-', or real number values. Features will be considered a match only if they **match exactly**. TO DO: 'x' values are special and will be interpreted as a segment is unspecified for that feature. (Treated as a wildcard in some cases, and skipped in others)

Provided sample files:

features.xlsx

features.txt

These provide features for all the phonemes of English. Column A of the spreadsheet is IPA, while column B is single-character Arpabet (<https://en.wikipedia.org/wiki/ARPABET>). The text file was made by copying columns B-AD of the spreadsheet.

**Initializing the Features() class:**

You must use a features file to initialize an object of the Features class.

import learner

f = learner.Features('features.txt')

This class now has attributes:

featureNames – a list containing the feature names from the first row of the text file

featureValues – a dictionary with segment names as keys, and lists of feature values as values

use:

f.featureNames

f.featureValues

to examine them.

You can use this object to convert back and forth between strings of segments and feature dictionaries.

Features.stringToF(string)

This function takes a simple string, and interprets each character as a separate segment. It returns two items: a dictionary containing the segments and their feature values, and a list containing the order of the segments.

added seglabels argument

so, running

f.stringToF('ta')

should return this:

﻿({'seg1': [('0', 'syllabic'), ('0', 'stress'),

('0', 'long'), ('1', 'consonantal'), ('0', 'sonorant'), ('0', 'continuous'), ('0', 'delayedRelease'), ('0', 'approximant'), ('0', 'tap'), ('0', 'trill'), ('0', 'nasal'), ('0', 'voice'), ('0', 'spreadGlottis'), ('0', 'constrictedGlottis'), ('0', 'labial'), ('0', 'round'), ('0', 'labioDental'), ('1', 'coronal'), ('1', 'anterior'), ('0', 'distributed'), ('0', 'strident'), ('0', 'lateral'), ('0', 'dorsal'), ('x', 'high'), ('x', 'low'), ('x', 'front'), ('x', 'back'), ('x', 'tense')],

'seg2': [('1', 'syllabic'), ('0', 'stress'), ('0', 'long'), ('0', 'consonantal'), ('1', 'sonorant'), ('1', 'continuous'), ('x', 'delayedRelease'), ('1', 'approximant'), ('0', 'tap'), ('0', 'trill'), ('0', 'nasal'), ('1', 'voice'), ('0', 'spreadGlottis'), ('0', 'constrictedGlottis'), ('0', 'labial'), ('0', 'round'), ('0', 'labioDental'), ('0', 'coronal'), ('x', 'anterior'), ('x', 'distributed'), ('x', 'strident'), ('0', 'lateral'), ('1', 'dorsal'), ('0', 'high'), ('1', 'low'), ('0', 'front'), ('1', 'back'), ('x', 'tense')]},

['seg1', 'seg2'])

Converting a feature set to a string:

Features.featureToS(segs,order)

This function does the reverse. Given a dictionary object, and a list defining the order, it will return a string of the segments in the Features object. To run it, you first need to set up a dictionary for the 'segs' object. Here's an example:

segs = ﻿{'s1':[('1','lateral')],

's2':[('1','high'),('1','back'),('1','tense'),('1','syllabic'),('0','long')]}

f.stringToF(segs,['s1','s2'])

Should return:

'lu'

If you give the function a 'segs' object where one of the segments is compatible with more than one segment in the Features object, it will ask you to manually pick the one you want.

If you give it a set of features that are incompatible with any entry in the Features object, then it will ask you to make up a character, and it will then add that character to it's featureNames object, with featureValues being set to the ones you gave. It will fill in 'x' for all the features not mentioned in your dictionary.

**Note:** You must avoid specifying features that are completely complementary: For example, if I specify every vowel as +back -front OR +front -back, then morphing back-to-front will not work well. Instead, if they are completely predictable from each other, choose one.

**To Do:**

- Check that both functions work for a variety of input types

- Create the 'wildcard' functionality so that 'x' is ignored in Features.featureToS()

- In Features.featureToS(), add a check to make sure that the character the user gives is not already in the object

- Create error message for when the user gives a feature that's not part of the object in Features.featureToS(segs,order)

﻿# Tableau

Call print() on a `Tableau` object to see a printout to the console

\*\*tag\*\* The human-readable name of the tableau, generally indicating what the input to the tableau was

\*\*prob\*\* The tableau probability, corresponding to how likely the learner is to use this tableau for learning. Should correspond to token frequency of the input. Can be set using the 'tab.prob' column in an input file.

\*\*candidates\*\* A list of candidate objects that make up the tableau

\*\*surfaceCands\*\* A list of unique surface forms (strings), which can be set using the 'surface' column in an input file. If the tableau has hidden structure (if more than one candidate corresponds to the same surfaceCand), this list will be shorter than the list of candidates.

\*\*obsProbsList\*\* A list of the observed probabilities for surface candidates in `surfaceCands`. Length corresponds to the length of `surfaceCands`.

\*\*predProbsList\*\* A list of each candidate's predicted probability based on the last time `getPredWinner()` was called.

\*\*HList\*\* A list of each candidate's Harmony score, based on the last time `getPredWinner()` was called.

\*\*winner\*\*

\*\*constraintNames\*\*

\*\*hiddenStructure\*\*

\*\*probDenom\*\*

`addCandidate(cand)` Adds `candidate` `cand` to the tableau

`rect()` Checks the tableau for candidate violation length and violation sign. Returns 1 if the tableau is well-formed, 0 otherwise

`getPredWinner(w,theory=MaxEnt)` Returns a single candidate object that wins under `theory`, with constraint weights `w`. If multiple candidates win, one is sampled. In MaxEnt, sampling is done according to the predicted probability distribution. In HG or Noisy HG, sampling is done from a uniform distribution. Options for specifying `theory` are 'MaxEnt', 'HG', and 'NoisyHG'.

`getObsCandidate(w,theory='MaxEnt')` Returns a single candidate object that is the 'observed' candidate for learning. If more than one candidate has observed probability, then an observed candidate is sampled from the observed distribution. If the tableau has hidden structure, then an observed candidate is calculated using Expected Interpretive Parsing (EIP, Jaroscz, 2013).

`compareObsPred(w,theory='MaxEnt')` Generates an observed candidate and a predicted candidate, and compares their surface forms. Returns a tuple containing the error value, the observed candidate, and the predicted candidate, in that order. The error value is 1 if there was a mismatch between the two surface forms, and 0 otherwise.

**Candidate:**

Simply holds all the info for one candidate. Must be initialized with:

c: a string representation of the candidate. Something that can interface correctly with a Features object you are using, or it can be human-readable any way you want if you're not using a Features object.

violations: a list of violations for the candidate. Must be numbers. The Tableau.rect() function will yell at you though if they are negative, or all zero, or something.

observedProb: the observed probability that this candidate surfaces in its tableau

surfaceForm: the surface form if using hidden structure. If you're not using hidden structure, this does not need to be initialized, it will be set to the same string as c.

After initializing, the candidate has these 4 attributes, plus:

harmony

predictedProb

^ these are both set to zero initially, and filled out during tableau evaluation.

**richCand()**

Inherits from class **candidate**, which means it has all of **candidate**'s attributes and methods/functions. It includes additional information about segment features, ordering, and activity levels. Can interface better with a **Features** object.

Initialize with:

c, violations, observedProb .... (not done)

The **lexeme()** class:

This class is meant to hold a single lexical entry. You can initialize it with a 'tag' and a list of segments, or with a simple string:

l = learner.lexeme('ami', ['t','z','n','a','m','i'])

Here, 'ami' is just a human-readable label, attached to the lexeme but ignored by the program.

l = learner.lexeme('ami')

Here, the class will immediately create a segment list using the string 'ami': ['a', 'm', 'i']

(one entry per segment)

You may also specify a 'kind', such as 'root', 'prefix', 'suffix', or something else. Currently this has no consequences however.

l = learner.lexeme('ami', ['t','z','n','a','m','i'], kind='N')

Once initialized, **lexeme()** has the following attributes:

tag – A human-readable label for the lexeme, not used by the program

segmentList – a list of segments, either given or generated from characters in the tag

segLabels – a list of unique segment names, generated from segmentList.

If segmentList is ['p','a','t','a','k','a','t','a'], segLabels will be ['p', 'a','t','a3','k','a5','t6','a7']. Numbers are added to second, third, etc. copies of segments based on their position in the original list

activitys – (note the unusual spelling) a list of float value activity levels for segmentList Set

to all 1 at initialization

linearSegOrder – a list of integers specifying the linear order of the segments in segmentList

* Starts at 1
* By default, specifies the same order that segments appear in the list: if segmentList is ['a','m','i'], linearSegOrder will be [1,2,3]
* Can be used to specify multiple 'options' for a position: ['t','z','n','a','m','i'] -> [1,1,1,2,3,4]
* Can in principle be used to express epenthesis, deletion, and metathesis, though unclear what use that would be

kind – A label for humans, specifying 'root', 'suffix', etc. Not used by the rest of the program

freq – the frequency of the lexeme, initialized to 0. Intended to increase during learning.

PFCs – A placeholder to contain a list of PFC objects. Set to None on initialization

Use print() to view a handy readout of the lexeme's contents

print(l)

**TO DO**: fix print function to print prettily when there are decimals in the activity values

A set of example lexemes is provided:

exlex\_joli()

exlex\_petit() - partial activation on the last segment

exlex\_ami() - competing segments in the first position, with partial activation

exlex\_hero()

**lexeme.toRichCand(featureSet)**

This function is used in tableau creation, when candidates are not given by the user. It will return a richCand() object representing the 'faithful' candidate for the lexeme. If there are multiple equally 'faithful' candidates possible, it will return a list of all of them.

The argument featureSet must be a Features() object in which all the segments in the lexeme's segmentList are represented.

The **constraint()** class:

To use this class, you must first define a function to assign violations to a candidate. You can define constraints a number of ways, from simple lambda functions to more complex functions.

All constraints must be formulated in such a way that they take as input some subset of these four classes:

string – a string of characters, like ‘pataka’ to which the function will directly refer. Example:

lambda string: re.search([aeiou][aeiou], string) <- one way to define \*Hiatus

def hiatus(string):

return bool(re.search([aeiou][aeiou], string)

richSR – a richCand() object which is the ‘target’ candidate being evaluated

stringFaith – a string of characters which can be compared to string to evaluate a correspondence constraint

richFaith – a richCand() object representing the ‘faithful’ candidate, to which richSR can be compared to evaluate a faithfulness constraint.

Cost of transposition??

can't just be '1'

must be less than deletion plus epenthesis

**Parsing liason consonants:**

- I am not writing a morphological parser, that is a big job, which is not the purpose of this here learner.

- Therefore, the user must specify in advance segments which they would like to designate as ambiguous (liason consonants)

When the model is initialized, these will be stored on both morphemes, with an activity value of 0

﻿input lexeme surface obs.prob tab.prob

1syll I I 1 1

2syll IO IO 1 1

3syll IOO IOO 1 1

3syll2 OIO OIO 1 1

Types of input file:

regular tableau without hidden structure – constraint violations prespecified

regular tableau *with* hidden structure – constraint violations prespecified

regular tableau with the ability to change activity values, and therefore max/dep violation values, but all other constraint violations prespecified

regular tableau with the ability to add PFCs, but markedness violations prespecified

simple training data, candidates to be generated on the fly

**Input file structure:**

There are two types of basic input file:

Tableaux file:

The Tableaux file is similar to what you would use in other phonological learning software, such as OTSoft, OT-Help, or hgr.

Lexeme file:

|  |  |  |
| --- | --- | --- |
| input | surface | obs.prob |
| xn\_@pl | xn@pl | 1 |
| xn\_tri | xtri | 1 |
| xn | xn | 1 |
| @pl | Q@apl | 1 |

input: the lexemes, separated by '\_'

* Each string between '\_' will be interpred as a separate lexeme
* Each character will be interpreted as a phone in the UR of the lexeme (if using UR's)
* All the characters should be phones in your **features** file, or you'll get an error at this stage

surface: the string of phones that should surface when the lexemes in input are concatenated.

* these must all be phones in your **features** file

obs.prob: How often this surface form is observed.

* In a simple file like this one, it encompasses both how often the input is observed, and how often the surface form in **surface** is a result of the lexemes in **input**

An input file like this:

|  |  |  |
| --- | --- | --- |
| input | surface | obs.prob |
| xn\_@pl | xn@pl | 1 |
| xn\_tri | xtri | 2 |
| xn | xn | 3 |
| @pl | Q@apl | 4 |

would learn on 'apple' in isolation four times as often as the input 'an apple'. However, 'an apple' will always surface with 'an' and, 'apple' will always surface with glottal stop. (Q) etc.

Suppose 'an apple' surfaced most but not all of the time with the 'n'. Maybe 10% of the time, you get glottal stop insertion:

|  |  |  |
| --- | --- | --- |
| input | surface | obs.prob |
| xn\_@pl | xn@pl | .9 |
| xn\_@pl | x@pl | .1 |
| xn\_tri | xtri | 1 |
| xn | xn | 1 |
| @pl | Q@apl | 1 |

This input file will feed the learner 'xn@pl' as the correct output for 'xn + @pl' 9 times as often as it will feed 'x@pl' as the correct output. It will feed all the inputs equally often though.

Note that this will be treated totally equivalently to:

|  |  |  |
| --- | --- | --- |
| input | surface | obs.prob |
| xn\_@pl | xn@pl | 90 |
| xn\_@pl | x@pl | 10 |
| xn\_tri | xtri | 100 |
| xn | xn | 100 |
| @pl | Q@apl | 100 |

You don't have to choose between modeling input frequency and modeling output probability. Here's an input file that incorporates both:

|  |  |  |
| --- | --- | --- |
| input | surface | obs.prob |
| xn\_@pl | xn@pl | 90 |
| xn\_@pl | x@pl | 10 |
| xn\_tri | xtri | 200 |
| xn | xn | 300 |
| @pl | Q@apl | 360 |
| @pl | @apl | 40 |

'apple' will be trained on 4 times as often as 'an apple', but for both, onsets appear only 10% of the time.

**The lexeme column:**

The 'lexeme' column can be used for two purposes:

1. To specify names for your lexemes that are not identical to the strings that define the phones of your lexeme
2. To remain agnostic about the parsing of certain phones (for example, peti**t**ami, you may want to specify the **t** as ambiguous, belonging to either petit or to ami.)

Here is an example of the first use:

|  |  |  |  |
| --- | --- | --- | --- |
| input | lexeme | surface | obs.prob |
| an\_apple | xn\_@pl | xn@pl | 90 |
| an\_apple | xn\_@pl | x@pl | 10 |
| an\_tree | xn\_tri | xtri | 200 |
| an | xn | xn | 300 |
| apple | @pl | Q@apl | 360 |
| apple | @pl | @apl | 40 |

In this case, the tags of the lexemes will be the human-readable 'an', 'apple', and 'tree', while their phonetic specification will be taken from the 'lexeme' column, so the phonetic sequence will still be @pl, for 'apple' etc.

The following columns can appear in any order:

input, lexeme, tab.prob, surface, candidate, obs.prob (input and obs.prob are obligatory)

after these columns are input, any subsequent columns will be interpreted as constraints

**Ambiguous segments:**

lex: petit\_ami

splex: peti-t-ami

lex: faqa\_alo\_ia

splex: faqa-t-alo-f-ia

^ loop through lex:

check if correspondent contains ambiguous material

faqa\_-alofia-

**Examples here:**

\_\_\_\_\_\_\_\_\_\_\_\_\_

Note that if you provide a tableau with candidates, some of your sampler entries may be zero (perhaps a lot of them)

Candidate Generation function:

**NOTE:** there are some odd dependencies with the morphFeature operation – as defined now, you won’t ever get a candidate that changes, say, backness AND rounding unless there’s a segment in the language that just does one of those. Need to fix!

In order to generate candidates, we traverse a tree, that begins with the faithful candidate. (Note that in GSR’s there are multiple faithful candidates)

Daughters of each node are computed via operations, which must be defined by the user of the program. Here is an example (part of) a tree, with the following operations;

Delete a single segment

Add a glottal stop [ʔ]

Change a single feature

pʌdəl

Add

Change-feature

Delete

... ʌdəl

pʌdəɹ

pʌdəlʔ

pʌʔdəl

...

ʔpʌdəl

pʌɾəl

pədəl

phʌdəl

pʌdl

pʌəl

pdəl

pdʌl

pɾəl

phdəl

...

...

dəl

pʔdəl

ʔənæpl̩

ʔənæp

ʔənæp

ʔnæpl̩

How tree search proceeds:

Beginning with the root node, the faithful candidate, operations are considered with probabili

Rules about tree search:

1. If a candidate is already in the candidate set, it is never added again
   1. This means its daughters won’t be generated again
2. Operations are considered as a wgo

Calculating the Gradient:

Need the softmax function. Did you know that the MaxEnt function is a softmax function???

- explanation and history here: <https://deepai.org/machine-learning-glossary-and-terms/softmax-layer> -- Boltzmann or Gibbs distribution!!!

<https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1>

log likelihood:

Given any probability model, we can calculate the likelihood of our data given our model. Let's take a simple example with dice. Say, I have a fair 6-sided die. In this case, my probability model would be that each side of the die has an equal chance of being rolled. So, we expect to roll a 1 with probability of 1/6. Likewise, we expect to roll a 2 with a probability of 1/6, and a 3 a 4 and so on.

So, suppose our data are that we rolled the die 3 times and got: 1, 2, 1

The likelihood of rolling the first 1 is 1/6. So is the likelihood of rolling the 2, and the second 1. So, since P(A and B) = P(A)\*P(B), the likelihood of the data here is 1/6^3 = 0.00463

Just to illustrate what this would look like if we switched out the theory, which is what we'll be doing when we optimize weights: suppose our probability model was instead that the die only ever rolls 1's. So, the probability of a 1 is 1, and the probability of a 2 or any other number is 0. Now, our Likelihood is 1\*0\*1 = 0

One more: suppose our probability model was that the die was heavily biased towards 1's: p(1)= 0.9, prob of any other number is 0.016. Now, the likelihood of our data is .9\*.016\*.9 = 0.013]

So, according to our likelihoods, that third model is the best, since it yields the highest likelihood.

So, how do we scale up this likelihood calculation? Let's assume we've got a set of tableaux, with a bunch of observed counts, and predicted probabilities.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ℂ1 | ℂ2 | ℂ3 | 𝓗 | *predicted* | *observed* |
|  | 1.1 | 2.3 | 8.7 |  |  |  |
| /Input/1 |  |  |  |  |  |  |
| *cand1,1* | 1 | 1 | 0 | -3.4 | *0.25* | *415* |
| *cand1,2* | 1 | 0 | 1 | -9.8 | *0* | *12* |
| *cand1,3* | 0 | 1 | 0 | -2.3 | *0.75* | *891* |
| /Input/2 |  |  |  |  |  |  |
| *cand2,1* | 0 | 0 | 1 | -8.7 | *0* | *1* |
| *cand2,2* | 1 | 0 | 0 | -1.1 | *1* | *345* |

Now, for each candidate, we have a number of observations of that candidate, perhaps from a corpus or an experiment, and those observations together are our data. Each observation of each candidate has a likelihood of appearing equal to its predicted probability. So, *cand1,1* has a 0.24 chance of occurring. If it occurred twice, the chances of that happening would just be 0.24\*0.24. If it occurred *n* times, the chances of that are 0.24*n*. The same is true for each candidate:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | ℂ1 | ℂ2 | ℂ3 | 𝓗 | *predicted* | *observed* | **Likelihood** |
|  | 1.1 | 2.3 | 4.8 |  |  |  |  |
| /Input/1 |  |  |  |  |  |  |  |
| *cand1,1* | 1 | 1 | 0 | -3.4 | *0.24* | *415* | **0.24415** |
| *cand1,2* | 1 | 0 | 1 | -5.9 | *0.02* | *12* | **0.0212** |
| *cand1,3* | 0 | 1 | 0 | -2.3 | *0.74* | *891* | **0.74891** |
| /Input/2 |  |  |  |  |  |  |  |
| *cand2,1* | 0 | 0 | 1 | -4.8 | *0.02* | *1* | **0.021** |
| *cand2,2* | 1 | 0 | 0 | -1.1 | *0.98* | *345* | **0.98345** |

So how do we get the likelihood of the whole dataset? It's just each candidate's likelihood multiplied together:

or:

Before we move on, a brief note: You will often see models fit with observed probabilities, rather than observed counts. For the tableaux above, that would be 0.315 instead of 415, 0.009 instead of 12, 891 instead of 0.68, and so on. So then the likelihoods for each candidate would be 0.240.315, 0.020.009, and so on. Technically when you do this, you are not calculating the likelihood, but instead something called Cross Entropy. You can optimize Cross Entropy just like you would likelihood, and all the math below applies.

When to go with Likelihood and when to go with Cross Entropy:

You should use observed probabilities, and therefore cross entropy, when you want to fit to type frequency only, and you don't want any input's distribution to be more important than another's. You should use observed counts, and therefore likelihood, when you want to fit to token and type frequency. With observed counts, inputs with fewer total observations will contribute less to the likelihood calculation, and therefore may influence your learned constraint weights less.

Let's return to this likelihood function:

Now, 0.24415 is already a teeny tiny number, which most computers will round to zero. After we multiply all these probabilities together, the problem just gets worse. So, to fix this, we will optimize for the *log* of the likelihood instead of the likelihood itself.

How does this help? Well, log(x\*y) = log(x) + log(y), so:

And, log(xy) = ylog(x), so:

Now, we can plug in our actual calculation for the predicted probability, which is:

Recall also that the harmony of a candidate, is just the dot product of the constraint weights with that candidate's violations. If we also plug that into our equation, we get the full likelihood function, in terms of our weights:

Suppose we have *n* weights:

We need the gradient of this, which will be . In other words, we need a vector of the partial derivative of the likelihood function with respect to each weight. It will be the same length as .

First, let's simplify that objective function a little:

Now, let's rewrite, expanding those sums and dot products out:

This is the first term of the second input

Where r is the number of inputs

So each of these terms can be differentiated, and each entry in the gradient will be a sum of all of them. But, because they have the same form except for constants, let's just demo with a single term:

We have to do the chain rule here, that's what that last term is. Here

so: that last term in terms of *u* is

so:

and to get we actually have to apply the chain rule one more time, this time for each term of u(w1)

For a single term:

The derivative of this is just

So,

The rest of the gradient terms are just the same, but replace w1 with w2 and so on