Language and Computation

LING 227 01 / 627 01 / PSYC 327 01
Assignment #5
Due: Monday, April 28, 2014

This last time, the solutions must be handed in electronically, only. You will send a **single** Python code to `tamas.biro@yale.edu`. Do not forget to include your first and last name in the code's filename, which should not contain white spaces, please.

The goal of this homework is manifold again. You will get hands-on experience with phonology, Optimality Theory and learnability—an experience that might be generalized to computational approaches to theoretical linguistics, in a broad sense. You will also practice (or learn) how to use objects, functions and exceptions in Python—skills that you can transfer to other programming languages, as well.

You must submit the homework by the deadline, even if you have not managed to solve all the problems.

Each problem set is worth 10 points in total. Your code will not be a single running program, but a set of functions and class definitions, which we will import into our test script making use of those functions and classes. For instance, you will be asked to write a function `gen(string)` that generates a list of candidates for any `string`; if our test script with your definition generates the correct outputs for the inputs we provide, then you will get the points this part of the assignment is worth. If you give a different name to your function (such as `Gen`), or if it returns a different data structure (such as a tuple, and not a list), then the code will not run.

Problem: Learning Word Stress with Optimality Theory and GLA (10 points)

1.1 A refined model for stress in OT

In class, I introduced an extremely oversimplified example of how contemporary theoretical phonology accounts for metrical stress in the languages of the world (04/08, slides 6 to 10). The three constraints Early, Late and NonFinal accounted for the existence of three language types and the lack of a fourth possible type.

Although there is a kernel of truth in this toy example, reality is much more complex. Many languages, including English, require a more complicated system. First of all, longer words in many languages, such as in English, have not only primary, but also secondary (and even tertiary) stress. Moreover, languages often make a distinction between *heavy syllables* and *light syllables*, the earlier ones being stronger at attracting stress.[1] Still, we suppose that word stress can somehow be predicted ("computed") in all these languages.[2]

In what follows, I am introducing a model that is much closer to what is currently considered as the standard approach to stress in phonology. What is missing from the analysis is the notion of *foot*, and therefore this model does not exactly make the right predictions. Nevertheless, it will help us better understand what the *learnability* of a theoretical framework entails.

1.2 Basic components of an OT analysis

1.2.1 Gen: the generator function

The **Gen** function will map any input word to all possible stress patterns. Exactly one syllable will be assigned a *primary stress*, and all other syllables may either be left *unstressed*, or receive *secondary stress*. (Tertiary stress is ignored.) We suppose, which is not self-evident, that the input is already syllabified.

In particular, the input to our system will be a string over the two-letter alphabet $\Sigma = \{\texttt{L}, \texttt{H}\}$, where `L` stands for a light syllable and `H` for a heavy one. For instance, `LHL` is a three-syllable input word with a heavy syllable in the middle. The Gen function maps this input onto a set of strings over $\Delta = \{\texttt{L}, \texttt{H}, \texttt{1}, \texttt{2}\}$. The character `1` stands for the primary stress, and it will be inserted *after* the syllable to be stressed. Similarly, the secondary stress symbol `2` will be inserted after all syllables with a secondary stress. Thus, for any input $u \in \Sigma^*$, Gen(u) contains the strings that are arrived at by inserting the primary stress symbol `1` after exactly one of the characters in u, while any *other* character f u can be followed by `2`. For instance, Gen(`LHL`) is the set $\{\texttt{L1HL}, \texttt{L1H2L}, \texttt{L1HL2}, \texttt{L1H2L2}, \texttt{LH1L}, \texttt{LH1L2}, \texttt{L2HL1}\ldots\}$.[3]

How many candidates does an input map onto? If the input consists of n syllables, then there are n different options for assigning the primary stress. Subsequently, each of the remaining $n-1$ syllables can either be assigned a secondary stress, or be left unstressed. Therefore, the number of candidates is $n \cdot 2^{n-1}$. When you implement your `gen(string)` function, you may want to test your code by counting the number of candidates generated for inputs with different lengths.

[1]The precise definition of what counts as heavy depends on the language. Usually, a syllable is heavy, if it contains a long vowel, and/or a consonant following the vowel (the latter being called the *syllable coda*).

[2]Such is not the case in languages where the position of the stress in unpredictable, and must be stored in the lexicon. A famous example is Russian, in which the word [muka] can mean 'torture' or 'flour', depending on whether the first or the second syllable is stressed. In English, the part-of-speech of a word can influence the place of the stress: compare the verb *to record* to the noun *a record*.

[3]As an optional problem: can you prove that $\{\text{Gen}(i) | i \in \Sigma^*\}$ is a regular language, and that the relation $\{(i, o) | i \in \Sigma^*,\ o \in \text{Gen}(i)\}$ is a regular relation?

1.2.2 Constraints

Although mainstream Optimality Theory posits that the set of constraints is universal across the languages of the world, it turns out that it is far not universal across linguists themselves. Yet, there is a canonical set of constraints that have been frequently used for computational experiments on learning stress systems.

Many of these constraints refer to *metrical feet*, which we have not included in our current analysis. Therefore, let me propose an alternative set of constraints, somehow analogous to the standard ones, but not exactly equivalent to them. While they might prove questionable in the light of linguistic data, they will perfectly serve the needs of our computational experiment. Here they are:

- **Parse-Left2Right (PL2R):** A syllable must bear stress, unless its left neighbor is stressed. The number of violations assigned is equal to the number of unparsed syllables that are not immediately preceded by a stressed syllable.[4]
- **Parse-Right2Left (PR2L):** A syllable must bear stress, unless its right neighbor is stressed. The number of violations assigned is the number of unparsed syllables that are not immediately followed by a stressed syllable.
- **NoClash (NC):** Stressed syllables must not be adjacent. Assign one violation to each stressed syllable followed by another stressed syllable.
- **PrimaryStressLeft (PSL):** The primary stress must occur as early as possible. Assign one violation mark per each syllable intervening between the left edge of the word and the syllable with the primary stress.
- **PrimaryStressRight (PSR):** The primary stress must occur as late possible. Assign one violation mark per each syllable intervening between the syllable with the primary stress and the right edge of the word.
- **AllStressesLeft (ASL):** All stresses must occur as early as possible. For each stressed syllable, count the number of syllables intervening between the left edge of the word and this syllable, and then sum up these counts for all stressed syllables.[5]
- **AllStressesRight (ASR):** All stresses must occur as late as possible. For each stressed syllable, count the number of syllables intervening between this syllable and the right edge of the word, and then sum up these counts for all stressed syllables.

[4]For instance, a word initial unstressed syllable incurs a violation mark. A series of three unstressed syllables incur (at least) two violation marks.

[5]For instance, ASL(`H2 L1 H2 L L2`) $= 0 + 1 + 2 + 0 + 4 = 7$, because the leftmost stress is adjacent to the left edge of the word (no violation), then the primary stress incurs one violation mark, the middle syllable incurs two of them, the fourth syllable is unstressed (no violation of ASL), but the rightmost syllable incurs four marks, due to the four intervening syllables between the left edge of the word and this rightmost syllable.

- **WordNonFinal (WNF):** The last syllable of the word must not bear stress. Assign the word as many violations as the number of stresses on its last syllable (i.e., 0 or 1).
- **WordStressLeft (WSL):** The first syllable of the word must bear stress. Assign one violation to the candidate, if its first syllable is *not* stressed; zero, otherwise.
- **WordStressRight (WSR):** The last syllable of the word must bear stress. Assign one violation to the candidate, if its last syllable is *not* stressed; zero, otherwise. (NB: WSR$= 1-$WNF.)
- **Weight2Stress (W2S):** Heavy syllables must be stressed. Assign one violation mark per every heavy syllable that is not stressed.

1.2.3 Grammar and learning

In traditional Optimality Theory, a grammar G is a permutation (*hierarchy*) of these 11 constraints. Given a set of underlying forms (*lexicon*) $\mathcal{U}$, the Gen function defined above generates a set of candidates Gen(u) for each $u \in \mathcal{U}$. Subsequently, the ranked constraints filter out the sub-optimal candidates, yielding the most harmonic output(s) SF(u) corresponding to u. The series of filters is frequently referred to as the Eval module of OT.

Thus, during production, the speaker searches for the best output form, given a constraint hierarchy. During learning, however, the learner's task is the opposite: given a finite set of learning data (input-output pairs), she has to find a constraint hierarchy that can produce all of them. Here, we adopt mainstream OT's postulate that Gen and the constraints are given by the onset of language acquisition; alternative approaches also exist.

These eleven constraints have $11! = 39,916,800$ permutations. While many of these different rankings will define the same $u \mapsto$SF(u) mappings, it is not realistic to perform an exhaustive search on this huge space. Therefore, we need smarter approaches.

Adopting the trick introduced by Paul Boersma, we implement an OT hierarchy by assigning a *ranking value* (a real/floating point number) to each constraint. The higher the ranking value of a constraint, the higher it is ranked in the hierarchy. We do not add noise to these ranking values, but employ them for learning in the *Gradual Learning Algorithm* (GLA), as follows:

Initialized with a (random) hierarchy, the learner maintains a hypothetical grammar G_L. In each learning cycle, the learner is presented with a piece of observation, a "winner form" w generated by the teacher's grammar G_T (the target grammar). The learner recovers the underlying form u from which w was generated (not a self-evident step in general, but trivial in the case of stress assignment), and then calculates the "loser form" l that G_L would have produced. If $w \neq l$, then an error is detected, which drives this *error-driven learning algorithm* (as also explained in section 11.5.3 of Jurafsky and Martin). If $w \neq l$, then there must be at least one constraint C_w that prefers w to l

(i.e., $C_w(w) < C_w(l)$), and which is ranked high in G_T but low in G_L; hence, it should be promoted in G_L. Similarly, l wins over w in G_L because there is at least one C_l that prefers l to w (i.e., $C_l(l) < C_l(w)$), and which is ranked too high in G_L; consequently, it should be demoted.

Therefore, in order to get closer to the target grammar and after having introduced a real (floating point) ranking value for each constraint, GLA suggests that (1) the ranking values of winner preferring constraints be all increased by 1 and (2) the ranking values of the loser preferring constraints be all decreased by 1 (alternative suggestions have also been made). We hope, although we do not have guarantee for it, that after a number of learning cycles the learner's grammar will converge on a correct grammar. *Correctness* refers to the learner's ability to correctly reproduce all learning data, and not necessarily to having reached the teacher's hierarchy.

To summarize, and filling in some details, here is how in particular we shall implement the GLA algorithm:

1. We fix a lexicon $\mathcal{U} = \{\texttt{LH}, \texttt{LLL}, \texttt{LHL}, \texttt{HHL}, \texttt{LLH}, \texttt{HLHL}, \texttt{LLLL}, \texttt{LHLH}, \texttt{HHLL}, \texttt{LLHLL}\}$.

2. Teacher T is initialized with a random grammar G_T, by assigning a (uniform) random ranking value between $R_{\text{min}} = 0$ and $R_{\text{max}} = 50$ to each constraint. Similarly, learner L is also initialized with a (different) random grammar G_L.[6]

3. The learning data are presented to the learner, either in a random order, or cyclically (looping through $\mathcal{U}$, $k = 10$ times):

   - For each $u \in \mathcal{U}$, let w be the form produced by T.
   - L recovers u from w.
   - L's current grammar G_L produces l for u.
   - If $w = l$, then L is happy. Otherwise, L updates G_L by increasing the rank of all constraints that prefer w to l by ϵ, and decreasing the rank of all constraints that prefer l to w by ϵ. The parameter ϵ, called *learning plasticity*, will be set to 1.[7]

4. After learning has terminated, test if L has acquired T's grammar (that is, whether G_T and G_L encode the same constraint hierarchy); if not, check if L has acquired T's language (that is, whether all underlying forms in $\mathcal{U}$ are mapped onto the same surface forms).

5. Will you obtain the same results if you repeat the experiment with different random initializations? How frequently will learning be successful?

[6]The ranking values of the constraints in G_L will most probably grow outside of this $[R_{\text{min}}, R_{\text{max}}]$ interval during learning, which is not a problem. Moreover, you may speed up your algorithm by computing the teacher's favorites for the underlying forms $u \in \mathcal{U}$ once, in the initialization phase, rather than re-compute them each time during the learning process.

[7]Boersma and Hayes (2001) originally uses $\epsilon = 0.1$, but that may lead to problems in floating point arithmetic.

1.3 Assignment questions

This project will be implemented by making use of the modularity offered by object-oriented languages such as Python. While more efficient approaches certainly exist, I am asking you to adopt the following scheme:

- Gen is a function `gen(string)` with a string argument (over $\Sigma = \{\mathtt{L}, \mathtt{H}\}$) that returns a list of strings over $\Delta = \{\mathtt{L}, \mathtt{H}, \mathtt{1}, \mathtt{2}\}$. It raises (throws) an exception if its argument $\mathtt{string} \notin \Sigma^*$.

- A constraint is an object belonging to the class `Constraint`. It is initialized (instantiated) with an argument that is a function f, defining what it does. It has one method: `Constraint.assigns(string)` returns the number of violation marks $f(\mathtt{string})$. Moreover, it also has an attribute, `Constraint.rank`, which contains a floating point ranking value.

- Eval is implemented as a function `eval(constraints,candidates)` that takes a list of `Constraint` instances[8] and a set of candidates (in the form of a list of strings) as its two arguments, and returns the optimal subset of the candidates. The output is again a list of strings—usually with a single element, but possibly with more, and never with none.

- The function `gla(constraints, winner, plasticity=1)` implements an update step of the Gradual Learning Algorithm. It has two obligatory and one optional arguments, and no return value.[9] Its first argument is again a list of `Constraint` instances, and the second argument is a string over Δ (a piece of learning data). The function first recovers the underlying $u \in \Sigma^*$ from `winner`, then makes a call to `gen` and `eval` to compute the loser form, and finally updates the ranking values with ±`plasticity` (which is by default = 1, unless otherwise specified).

- Function `experiment(lexicon, constraints, k=10, Rmin=0, Rmax=50, plasticity=1)` with two obligatory and four optional arguments implements the experiment described on the previous page: randomly initializing the constraint ranking values in the [`Rmin`, `Rmax`] interval, both for the teacher and the learner, and then *cyclically* feeding GLA with all elements in `lexicon` (a list of underlying form strings), `k` times.

In what follows, I describe what the code you submit should contain, and I also provide several hints.

[8]The ranking values of the constraints appear as their attribute `Constraint.rank`, a higher value corresponding to a higher rank in the hierarchy. Consequently, the first argument of `eval(constraints,candidates)` may not be sorted in advance.

[9]We might have introduced a boolean return value informing the user whether an update has taken place. But a Python function does not have to always return some value.

1.3.1 Constraints

You can use the following code to define the `Constraint` class:

```
__metaclass__=type

class Constraint:
    rank=0
    def __init__(self, violationFunction):
        self.f = violationFunction
    def assigns(self,string):
        return self.f(string)
```

Subsequently, you will implement each of the eleven constraints defined above. Use the capital letter abbreviations (such as PL2R, NC, and PSL) as variable names for them. So your code should introduce these eleven variables, each pointing to a `Constraint` object, and when our test script will import your code, we should be able to apply these constraints to any (meaningful) string.

Here are two examples (each with a minor bug that you should correct):

```
def nc(string):
    viol=0
    for position in range(len(string)):
        if ( (string[position]   == '1' or string[position]   == '2')
         and (string[position-2] == '1' or string[position-2] == '2') ):
            viol += 1
    return viol
NC=Constraint(nc)

WNF=Constraint(lambda string: int(string[0] in ['1','2']))
```

Try out what, for instance, `NC.assigns('H1H2L2')` returns. You should implement all eleven constraints in a similar format.

1.3.2 Gen, the generator function

Now, you will implement `gen(string)`. Here is a fake solution, the skeleton of which you will copy, and the content of which you should replace (but if you cannot do so, you can use it so that you can solve the rest of the assignment):

```
def gen(string):
    G=[]
    l=len(string)
    if   l == 1:
        G.append(string+'1')
    elif  l == 2:
        G.append(string[0]+'1'+string[1])
        G.append(string[0]+string[1]+'1')  # well, in short: string + '1'
        G.append(string[0]+'1'+string[1]+'2')
        G.append(string[0]+'2'+string[1]+'1')
    elif  l == 3:
        G.append(string[:1]+'1'+string[1:])
        G.append(string+'1')
        G.append(string[:1]+'1'+string[1:]+'2')
        G.append(string[:1]+'2'+string[1:]+'1')
        # etc. etc.
    else:
        raise Exception("String is too long. I can't generate candidate set")
    return G
```

How to implement stress assignment for `gen`? While several solutions may exist, here is a reasonable approach (presented in Python-friendly pseudo-code). Let's collect the candidates in a list `cands`:

```
cands <-- empty list
# here we append all candidates to the list cands
return cands
```

To begin with, observe that by adding primary stress only to the original input string `underlyingForm`, we already obtain valid candidates:

```
for i: 1 to (and including) length of underlyingForm
    candidate <-- slice[0:i] of underlyingForm + '1' +
                  slice[i:end_of_string] of underlyingForm
    append candidate to cands
```

Then, let us insert secondary stresses to all possible positions. First, we consider the elements so far added to `cands` (these are the candidates with a primary stress and no secondary stress), and we create new candidates by assigning a secondary stress to their first syllable—unless it already contains a primary stress. Subsequently, we consider all the candidates in `cands` (with and without secondary stress on their first syllables), and create new candidates again by adding a secondary stress to their second syllable; and so forth. Here is the pseudo-code:

```
for i: 1 to (and including) length of underlyingForm
        candsCopy <-- copy of cands    # never iterate over an object
        for c in candsCopy             # that you modify inside the loop!
            position <--  locate position of the i'th syllable in c
            if position == (length of c - 1) or c[position+1] != '1':
                # create new candidate by inserting secondary stress
                # to position 'position', unless there is primary stress
                newCandidate <-- slice[0:position] of c + '2' +
                                 slice[position:end_of_string] of c
                append newCandidate to cands
```

To help you, here is the code for the function that finds the position of the ith syllable (‘H’ or ‘L’) in the string, despite possibly interspersed ‘1’ and ‘2’ characters:

```
def locate(string, i): # returns position of i’th H or L in string
    position = -1      # position being looked at
    syllable = 0       # count of number of syllables found so far
    while syllable < i:
        position += 1
        if string[position] == ’H’ or string[position] == ’L’:
            syllable += 1
    return position
```

Finally, a perfect implementation of Gen will also test if its argument `string` $\in \Sigma^*$. If not, it will raise (throw) an exception.

1.3.3 Eval

Your next step is to create `eval(constraints,candidates)`. This function takes two arguments, a *list* of constraints (such as `[WNF,W2S,NC,PSL...]` and a *list* of candidates (such as those returned by `gen`). The constraints are not necessarily sorted in the list of constraints, but your function will sort them by the `rank` attributes.[10]

Be sure you are familiar with Python’s following incredible construction, called *list comprehension*:

```
new_list = [function(x) for x  in original_list  if condition(x)]
```

It will select those elements `x` in `original_list` that satisfy `condition(x)`, and then create a new list by applying `function` to these values. Informally, it corresponds to $\{f(x)|x \in \text{original and condition } (x) \text{ is true}\}$. The 1st chapter of the NLTK book provides nice examples when discussing how to create text statistics (`http://www.nltk.org/book/ch01.html`).

Using it, you will be able to implement an OT constraint acting as a filter, by writing a single code line (or two). The idea in mathematical terms is:

$$\begin{aligned} m &:= \min \{ \mathit{Constr}\ (\mathit{cand}) \mid \mathit{cand} \in \mathit{candidate\ set}\ \} \\ \mathit{filtered\ set} &:= \{ \mathit{cand} \in \mathit{candidate\ set} \mid \mathit{Constr}\ (\mathit{cand}) = m \} \end{aligned}$$

Eval in Optimality Theory is a list of such filters, applied in the right order. Luckily, stress assignment defines a finite candidate set, and of manageable size, at that. Therefore, we do not need a smarter approach to Eval, but one can simply copy the filtering technique of the linguists evaluating OT tableaux.

[10]In case you use Python’s built-in sorting functions, hint 1: `key = (lambda x: x.rank)`. Hint 2: `reverse=True/False`.

Here is the skeleton with a wrong solution (that can nevertheless be used to test further parts of the assignment):

```
def eval(constraints,candidates):
    candidates=[]
    # Here you can write your code, for example
    candidates.append(candidates[0])
    return candidates
```

Keep in mind that the `eval` function will return a *list* of candidates, because nothing guarantees there will always be a single most harmonic form. You could check if some constraint hierarchies return more optimal outputs. However, most of the time (for instance, when you compute the loser and the winner forms during learning), you need a single form: `eval(constraints,candidates)[0]`.

1.3.4 Learning and experiment

Finally, you will also define the following two functions, as discussed earlier: one for a single learning step in the Gradual Learning Algorithm,

```
def gla(constraints, winner, plasticity=1):
    # updates the rank of the constraints:
    #     winner_preferring.rank += plasticity
    #      loser_preferring.rank -= plasticity
    # no return value
```

and one for the learning experiment (feeding the learner with all forms in the lexicon, and this cycle repeated k times):

```
def experiment(lexicon, constraints, k=10, Rmin=0, Rmax=50, plasticity=1):
    # Learning successful iff learner generates same outputs for all
    # underlying forms in lexicon as teacher.
    # Returns True if the learning is successful; False otherwise.
```

When initializing the teacher and the learner, you may want to use the `random` package, and in particular, the `random.uniform(min, max)` function. Furthermore, it is not necessary to define the constraints twice, for the teacher and for the learner. Rather, you can first assign them the ranks in the teacher's grammar, compute the teacher's surface forms for all the items in the `lexicon`, store that information, and only subsequently assign them the initial ranks in the learner's grammar, so that you can start the learning cycles. After the learning has terminated, you will test the learner's final grammar on each item in the `lexicon` in order to determine the function's return value.

As an optional plus, although not part of this problem set, I will appreciate a paragraph in your email describing your observations in case you will have "played" with this model: that is, you will have measured the frequency of successfully learning, as a function of the method's various parameters.

Good luck!