AMS 561 Project

Implementing String Comparison in Python to Evaluate Phonological Phenomena

Derek Andersen and Joanne Chau

May 4th, 2020

Project Objective

The objective of this project was to implement code in Python that can analyze two strings (of the same length or of differing lengths), and return a ruleset that explains the changes they underwent to go from string 1 $\rightarrow$ string 2, where string 1 is the input and string 2 is the output. This type of analysis is something commonly done in the field of computational phonology through the use of logical languages like first order logic and monadic second order logic. Ideally, we would want to implement a method for the logical analysis of strings in Python as well.

First, however, our goal is to write a program that can take two user-entered strings, and provide information about their differences, and then return a possible ruleset that describes the changes they underwent. For example, if the input string *abab* and the output string *bbbb* were provided, we would like our program to tell the user that *a* became *b*. In phonology, this type of approach is thought to be a constraint-based approach, through which we can describe the changes that humans' grammars force input strings to undergo, before they are output in speaking.

The project was evenly split up between both Derek and Joanne. We both took part in researching the topic, and coming together and sharing the information found among each other, before communicating and writing the code together in a Jupyter notebook. As the both of us shared knowledge from other classes we took together in our department, we brainstormed the material together. Most of the knowledge in this paper were from prior linguistics experience and knowledge that we gained from research on this material. We did not utilize any packages in the functions we developed for this project. We also utilized some of the coding skills and mathematical knowledge gained from the AMS 561 class from this semester.

Techniques and Tools

For this project, we used Python. We started first by dealing with two strings of the same length (e.g. *abab* and *baba*). Our first challenge was to write a function which can return the Hamming distance of two strings of the same length. Hamming distance between two strings is the number of of positions in which there is a difference between the strings at the same position. In the case of our project, we will call these differences changes. [Wika] After receiving this result, the next hurdle was to write a function that can return the changes from string 1 to string 2. The main issue in this function was the representation of the changes that would take place. We wanted to ensure that it documented the change specific to the index.

For example, if we had two strings, *abab* and *bbbb*, the function should return something that specifically states at index 0, *a* became *b* and at index 2, *a* became *b*. We decided the best way to illustrate this fact was through the usage of a dictionary, where the keys were the indices and the values were tuples of the change from string 1 to string 2.

After executing the two functions it shows that the number of entries in the dictionary is equal to the integer returned by the hamming distance. This means that the number of changes for the strings were recorded correctly for each index. This was a useful approach to understanding the changes to input strings, but will run into issues when the strings are of unequal distance. In speaking, often times, the length of string of both input and output are not the same, especially in cases where reduplication, insertion and deletion comes into play. For example, there can be a language that only allows vowels at the end of words. Assuming that their grammar only allows for the characters *a* and *b*. If the input of the string is *baba*, the output would be same. But if the input of the string is *bab*, the language speakers will insert a vowel in order to make it grammatical in their language and the output becomes *baba*. In this case, the length of the input and output are different.

We turn to Levenshtein distance for this. Levenshtein distance utilizes matrix formation between the two strings to come up with the minimal number of changes necessary. [Wikb] We implement a function that would calculate the Levenshtein distance for us and run it on the string *abab* and *baba*. [Hof] The Levenshtein distance for these two strings is 2. If looking at this based off of indices, we would assume 4 total changes, but based off of the matrix provided by Levenshtein, there are only 2. It can be interpreted as the first *a* being deleted and a final *a* is added to change from the first string to the second string.

The issue we run into now is that our currently `changes_in_string` is not capable of calculating the changes for the minimal changes needed. When both functions are run against each other, it can be shown that the number of entries in the dictionary provided by `changes_in_string` is not equal to the integer returned by the Levenshtein function. Another issue encountered is that Levenshtein accounts for strings of different lengths but the `changes_in_string` function assumes that the strings are of the same length.

Results

Our current functions are able to mark changes based off of indices of strings of the same length. We also have a working code that calculates the smallest number of changes needed from string 1 to string 2. Unfortunately, the current code is unable to calculate and provide the specific change taken place when calculating for the minimal changes between two strings.

Conclusion

In order to expand our code to further attest for strings, we would need to first implement our code for `changes_in_string` to handle strings of differing lengths. This can be done by adding dummy characters for a string. For example, the strings *abab* and *ab*, the string *ab* is shorter than the first string, so we will pad it with dummy characters to match the string lengths, and the string will become *ab$$*, *ab* or *$$ab*. A specific location for padding must be determined before the implementation of the code. This then will allow us to see all indices comparison between the strings, even of different lengths.

After this, we need to implement the Levenshtein code where we can put place holders on indices and their changes to mark the smallest number of changes. Then the program must evaluate the possible changes that must have occurred in order to obtain the minimal changes between two given strings. When this is done, in order to achieve our original intent, we would have to write more code that would help evaluate the changes into First Order Logic and then apply it to real phonological phenomenon, not just strings from a limited alphabet of *a* and *b*.

References

[Hof] Frank Hofmann. “Levenshtein Distance and Text Similarity in Python”. In: (). URL: `https://stackabuse.com/levenshtein-distance-and-text-similarity-in-python/`.

[Wika] Wikipedia. “Hamming distance”. In: *Wikipedia* (). URL: `https://en.wikipedia.org/wiki/Hamming_distance`.

[Wikb] Wikipedia. “Levenshtein distance”. In: *Wikipedia* (). URL: `https://en.wikipedia.org/wiki/Levenshtein_distance`.