MACHINE RECOGNITION OF RULE-BASED PHONOLOGICAL PATTERNS

By

ELIZABETH LORRAINE WAINWRIGHT

_____

A Thesis Submitted to the Honors College

In Partial Fulfillment of the Bachelor's Degree

With Honors in

Linguistics

THE UNIVERSITY OF ARIZONA

MAY 2017

Approved by:

_____

Dr. Adam Ussishkin

Department of Linguistics

Abstract

This thesis is project-oriented, with the goal of the research being to create a computer program that will take a set of words in a language written in the International Phonetic Alphabet (IPA) and two sets of phonemes where one undergoes a phonological change to the other and return a phonological rule describing this data. The program was written in Python. It included several functions as well as a new class termed "Phoneme" to facilitate the storage of phonemic features. A reference was made of Phoneme objects for the program to refer to for feature information while analyzing a data set. The program's parameters were that it be able to return a rule given that the data that it was analyzing contained only phonemes found in the reference and that the change be caused by a phoneme directly adjacent to the phoneme undergoing the change. Five datasets were passed through the program and, with the exception of one, were found to return a passable phonological rule. The most notable improvement to be made is the elimination of feature redundancies within the rule.

Machine Recognition of Rule-Based Phonological Patterns

Elizabeth Wainwright

University of Arizona

I. Introduction

A. Statement of Purpose

This thesis is primarily project-oriented; the goal of my research was to create a computer program that, upon being given phonological data from a language, would return to the user a phonological rule describing that language. Specifically, the program would be fed text files containing formatted words of a language transcribed in the International Phonetic Alphabet (IPA) and would return a phonological rule depending on the phonemes of interest to the user. This program, hereafter referred to as the Phonological Rule Generator (PRG), was written in Python.

B. Statement of Relevance

Human language technology is becoming increasingly important in today's society with recent technological advances, and voice recognition and synthesis are not exceptions to this trend. They are used in any technology involving the human voice, including geographical positioning systems, automated telephone systems, and search engines such as Siri. All of these applications involve machine analysis and replication of the human voice. Naturally occurring human languages are, however, incredibly complicated and often do not follow rules that are easily discernible to machines. For this reason, the field of computational linguistics turns to machine learning and statistical analysis to accomplish these tasks the majority of the time.

However, all languages do follow their own specific set of phonological rules that are based on much the same logic that forms the foundation of computer programming. For this reason, using a rule-based approach for linguistic problems in computation is much more feasible in the subfield of phonology. This program is meant to explore just how applicable a rule-based approach would be to the problem of phonological machine learning.

II. Methodology

Many components were involved in creating a functioning program that would generate phonological rules based on language data. Not only did the program have to include multiple functions for reading language data, identifying phonemes of interest, extracting the environments around those phonemes, determining the feature pattern that defines the change from the underlying phonemes, and generating a rule, but a wealth of phoneme data also had to entered manually. The following sections discuss these components in greater detail.

A. Parameters

Before the program could be written, the parameters of the capability of the program had to be decided upon. Firstly, how many phonemes would be included in the database? While a basic inventory of the various phonemes the human vocal apparatus can generate based on place and manner of articulation as well as whether or not the phoneme is voiced can be represented in under two hundred phonemes, there are actually exponentially more phonemes available to the

human vocal apparatus once features such as affrication, palatalization, and others are considered. Initially, I considered the following chart found in an instructional packet intended for use in graduate courses, composed by Steve Parker of the Graduate Institute of Applied Linguistics (Parker 10-11):

Fig. 2.1

	p	p^h	p□	b	t	d	k	g	ɸ	β	f	v	θ	ð	s	z	š	ž
syllabic	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
consonantal	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
sonorant	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
continuant	–	–	–	–	–	–	–	–	+	+	+	+	+	+	+	+	+	+
delayed release	–	–	–	–	–	–	–	–	+	+	+	+	+	+	+	+	+	+
strident	–	–	–	–	–	–	–	–	–	–	+	+	–	–	+	+	+	+
distributed	+	+	+	+	–	–	–	–	+	+	–	–	+	+	–	–	+	+
lateral	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
anterior	+	+	+	+	+	+	–	–	+	+	+	+	+	+	+	+	–	–
coronal	–	–	–	–	+	+	–	–	–	–	–	–	+	+	+	+	+	+
nasal	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
voice	–	–	–	+	–	+	–	+	–	+	–	+	–	+	–	+	–	+
aspirated	–	+	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
glottal	–	–	+	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
high	–	–	–	–	–	–	+	+	–	–	–	–	–	–	–	–	+	+
low	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
back	–	–	–	–	–	–	+	+	–	–	–	–	–	–	–	–	–	–
round	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
ATR (tense)																		

syllabic	x	ɣ	t^s	d^z	č	□□	m	ɱ	n	ñ	ŋ	l	□□	□□
consonantal	–	–	–	–	–	–	–	–	–	–	–	–	–	–
sonorant	+	+	+	+	+	+	+	+	+	+	+	+	+	+
continuant	–	–	–	–	–	–	+	+	+	+	+	+	+	+
delayed release	+	+	–	–	–	–	–	–	–	–	–		+	+
strident	+	+	+	+	+	+	–	–	–	–	–		+	+
distributed	–	–	+	+	+	+	–	–	–	–	–	–	–	–
lateral	–	–	–	–	+	+	+	–	–	+	–	–	–	–
anterior	–	–	–	–	–	–	–	–	–	–	–	+	+	–
coronal	–	–	+	+	–	–	+	+	+	–	–	+	+	+
nasal	–	–	+	+	+	+	–	–	+	+	–	+	+	+
voice	–	–	–	–	–	–	+	+	+	+	+	–	–	–
aspirated	–	+	–	+	–	+	+	+	+	+	+	+	–	+
glottal	–	–	–	–	–	–	–	–	–	–	–	–	–	–
high	–	–	–	–	–	–	–	–	–	–	–	–	–	–
low	+	+	–	–	+	+	–	–	–	+	+	–	–	–
back	–	–	–	–	–	–	–	–	–	–	–	–	–	–
round	+	+	–	–	–	–	–	–	–	–	+	–	–	–
ATR (tense)	–	–	–	–	–	–	–	–	–	–	–	–	–	–

syllabic	y	w	ʔ	h	i	ɪ	e	ɛ	æ	ə	a	ɨ/ï	u	ʊ	o	ɔ
consonantal	–	–	–	–	+	+	+	+	+	+	+	+	+	+	+	+
sonorant	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
continuant	+	+			+	+	+	+	+	+	+	+	+	+	+	+
delayed release	+	+	–	+	+	+	+	+	+	+	+	+	+	+	+	+
strident	+	+	–	+												
distributed	–	–	–	–												
lateral	–	+	–	–												
anterior	–	–	–	–												
coronal	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
nasal	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
voice	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
aspirated	+	+	–	–	+	+	+	+	+	+	+	+	+	+	+	+
glottal	–	–	–	–												
high	–	–	+	+	–	–	–	–	–	–	–	–	–	–	–	–
low	+	+	–	–	+	+	–	–	–	–	–	+	+	+	–	–
back	–	–	–	–	–	–	–	–	+	–	+	–	–	–	–	–
round	–	+	–	–	–	–	–	–	–	+	+	+	+	+	+	+
ATR (tense)	–	+	–	–	–	–	–	–	–	–	–	–	+	+	+	+
					+	–	+	–	+	+	–	±	+	–	+	–

This data was ideal for the purposes of this program due to the wide variety of phonemes represented as well as the wealth of data pertaining to the features of each phoneme included. Nevertheless, phonological change often presents as or due to the difference of affrication and similar features not as thoroughly represented here as necessary. This ultimately convinced me of the necessity of creating a new class, “Phoneme”, within the program, to allow for ease of addition of features or phonemes to the reference data. This will be explored at greater depth in Subsection IIB.

In addition to determining the scope of the features and phonemes, I also had to decide the scope of the types of phonological changes the program could recognize. The most basic of phonological problems involves the phonemes directly surrounding the phoneme that undergoes the change. However, many phonological changes are captured by phonemes that are not adjacent to the changing phoneme, or by features of syllables adjacent to the syllable in which the phoneme that undergoes the change occurs. In addition, multiple rules can present in a single word, and the order in which these rules apply can also affect the changes that occur in a word. For the purposes of this thesis, I decided to study only the phonemes directly adjacent to the changing phoneme, as any further analysis would require not only for the machine to capture the change but also for the machine to identify what type of rule it is attempting to identify in the first place. However, if this program were ever to be extended, these would be important additions to consider.

B. The Reference

To capture the features affecting a changing phoneme, the machine first must be provided with a large reference with manually annotated data regarding the features present in each phoneme. Initially, I created a dictionary mapping each phoneme to a binary string. Each position in the string stood for a feature, and the order in which these features were represented in the string were consistent throughout the entire dictionary, determined by a list of strings with the name of each feature to which the dictionary answers could be compared. If a “1” inhabited a certain position in a dictionary answer, then the phoneme had whatever feature that position

represented. If a “0” inhabited the position, then the phoneme did not hold the feature. For example, take the following entry in the dictionary:

```
Fig. 2.2.1      'n':"1100000000011000110"
```

The phoneme “n” is mapped to a binary string in which the ?th position is “1”. As, in this reference, the ?th position always represents the nasal feature, this means that “n” is +nasal.

While this reference functioned perfectly well for the test data I had selected, clearly phonemic data would have to be added depending on what phonemes occurred in the data sets entered. With the reference I was using, this constant modification would be very arduous. For ease of use, I decided to create a class in python, termed “Phoneme”. Upon initialization, each Phoneme object is provided with a symbol and a dictionary mapping each feature to its value. This would allow each feature value to be looked up by one line of code (see Figure 2.2.2 for an example involving the feature “voice”) instead of having to manually enter the dictionary and change it.

```
Fig. 2.2.2      Phoneme.feat_vals[“voice”]
```

Ultimately there was no effect on the testing of the datasets I had chosen for this particular project, but this change provides greater ease of use. The code for this is included below.

Fig. 2.2.3

```
class Phoneme:
    """This class allows for phonemes to be recorded, containing information on the
representation of their symbol as well as the values assigned to each of their
features."""
    def __init__(self, symbol, feat_vals):
        """This function initializes both the symbol and the dictionary of features
mapped to values for the phoneme."""
        self.symbol=symbol
        self.feat_vals=feat_vals

    def __repr__(self, symbol):
        """This function represents the phoneme as a symbol."""
        print(self.symbol)
```

1. Unicode

It is necessary to point out that it was at this point I experienced perhaps my greatest hurdle in developing this program. Although I could have created a Unicode friendly phonetic alphabet for use in this program, it was important to me to have the machine be able to accept data written in the International Phonetic Alphabet (IPA) for the ease of use this would provide phonologists. Unfortunately, the grand majority of IPA symbols are unreadable in Python just as they are. When run initially, therefore the program consistently threw Unicode errors. I attempted various methods to solving this problem to no avail. An example of one of these failed solution hypotheses was the use of the in-built chr() function in Python, given the character's hexadecimal form. Ultimately, though, the issue lay in the fact that many of the characters written into the program were stored in memory to a bit higher than possible for an eight-bit pattern, which I came to understand with the guidance of Dr. Peter Jansen of the University of Arizona. With 127 as the highest number available, numbers exceeding this had to be inserted manually into the reference in hexadecimal form, and then converted to their decimal form. The following example displays the hexadecimal form being used as one of the dictionary keys (in this case, the example is the velar nasal, "ŋ"):

Fig. 2.2.3 `str(int("0280", 16)):"1111001000000010010"`

The code for translation of the hexadecimal to decimal representation will be shown in the following section.

C. Finding Commonalities and Differences Between Target Phones

1. `def commonalities()`

The function `commonalities()` was built for the purpose of determining which features are shared by a set of phonemes either directly before or directly after the phonemes of interest occurring in the data. When utilized later on in the code, this function will allow the machine to determine what features of surrounding phonemes cause the change of the underlying phoneme. It takes a list of symbols, and, using the reference, generates a list of all distinctive features that the phones that the symbols represent share. It is in this function that the Unicode changes are implemented. If the phoneme is a "normal character (in other words, if its ordinal is less than 127), it is readable and its feature values can be looked up in the reference. However, if this is not the case, the hexadecimal in the reference is translated into the decimal in the following line of code:

Fig. 2.3.1

```
# If normal character, then lookup the normal way
if (ord(s) < 127):
    if ref[s][i]!=ref[symbols[0]][i]:
        add=False
# If unicode character, then get that character's decimal representation, then convert
that to a string to create a dictionary key
else:
    lookupKey = str(ord(s))
```

```
    if ref[lookupKey][i]!=ref[symbols[0]][i]:
        add=False
```

The complete function is shown in figure Figure 2.3.2:

Fig. 2.3.2

```
def commonalities(symbols):
    ref=reference()
    feats=features()
    common=[]
    if symbols.count("#")==len(symbols):
        return ["#"]
    elif "#" in symbols:
        return common
    else:
        for i in range(len(ref[symbols[0]])):
            add=True
            for s in symbols:
                #print ("symbol: " + s )

                # If normal character, then lookup the normal way
                if (ord(s) < 127):
                    if ref[s][i]!=ref[symbols[0]][i]:
                        add=False
                # If unicode character, then get that character's decimal
representation, then convert that to a string to create a dictionary key
                else:
                    lookupKey = str(ord(s))
                    if ref[lookupKey][i]!=ref[symbols[0]][i]:
                        add=False

            if add==True:
                if ref[symbols[0]][i]=="1":
                    common.append("+"+feats[i])
                else:
                    common.append("-"+feats[i])
    return common
```

2. `def differences()`

The function `differences()` was built to determine the features that differ between two sets of phonemes. This would be used to actually determine the feature change that occurred within the phoneme(s) of interest. The function takes two lists of features returned by the

function `commonalities()` and returns a list of features that appear in the first list but not the second. The complete function is shown in Figure 2.3.3:

Fig. 2.3.3

```
def differences(com1, com2):
    diff=[]
    for f in com2:
        if f not in com1:
            diff.append(f)
    return diff
```

3. `def clean()`

The function `clean()` was built to eliminate the redundancies in the various lists of features that will appear in the rule. The redundancies were hardcoded into the function. The function accepts a list of features and returns the list with the redundancies removed. This function had to go through many revisions, as it was very prone to runtime errors. This was due to the fact that if the a feature being removed is not actually in the list, an error immediately occurs. While the feature should be in the list, initially there were problems involved in earlier functions, and this would render this particular function unusable. However, after the debugging of earlier functions, this function proved useful for the representation of the final rule. The code is included in Figure 2.3.4:

Fig. 2.3.4

```
def clean(feat_list):
    if "-consonantal" in feat_list:
        feat_list.remove("+sonorant")
        feat_list.remove("+approximant")
        feat_list.remove("+dorsal")
        feat_list.remove("+pharyngeal")
        feat_list.remove("-coronal")
        feat_list.remove("-anterior")
        feat_list.remove("-distributed")
        feat_list.remove("-continuant")
        feat_list.remove("-lateral")
        feat_list.remove("-nasal")
        feat_list.remove("+voice")
        feat_list.remove("-strident")
    if "+consonantal" in feat_list:
        feat_list.remove("-ATR")
    if "+high" in feat_list:
        feat_list.remove("-low")
    if "+low" in feat_list:
        feat_list.remove("-high")
    if "-round" in feat_list and "-consonantal" in feat_list:
        feat_list.remove("-labial")
    if "-labial" in feat_list:
        feat_list.remove("-round")
    if "+labial" in feat_list and "+consonantal" in feat_list:
        feat_list.remove("+round")
    if "+round" in feat_list:
        feat_list.remove("+labial")
    if "-coronal" in feat_list:
        feat_list.remove("-anterior")
        feat_list.remove("-distributed")
    if "+anterior" in feat_list or "+distributed" in feat_list:
        feat_list.remove("+coronal")
    if "-continuant" in feat_list:
        feat_list.remove("-strident")
    if "+distributed" in feat_list:
        feat_list.remove("+dorsal")
    if "+coronal" in feat_list:
        feat_list.remove("-back")
    return feat_list
```

D. Finding the Reason for the Change

1. `def store_data()`

The function `store_data()` was built to store the words from the file of language data in a list. Because of the various characters used in the reference, the file had to be opened with UTF8 encoding. It takes a file as an argument and returns a list. The code is included in Figure 2.4.1:

Fig. 2.4.1

```
def store_data(fname):
    words=[]
    file= io.open(fname, encoding='utf-8')
    for line in file:
        words.append(line.strip())
    return words
```

2. def sort_data()

The function sort_data() was built to sort the words from the language data into either words that contain the underlying phoneme or words that contain the phoneme that to which the underlying phoneme changes. This function takes a list of words and a list of phonemes and returns a list of all words from the original list that contained at least one of the phonemes. The code is included in Figure 2.4.2:

Fig 2.4.2

```
def sort_data(words, symbols):
    spec=[]
    for w in words:
        for s in symbols:
            if s in w:
                spec.append(w)
    return spec
```

3. def get_environment()

The function get_environment() was built to extract the phonemes directly adjacent to the phonemes of interest to observe how they affected the underlying phoneme. This function took a list of words, a list of symbols, and a position, and returned a list of phonemes. Of note about this function is that it had to be able to pull the phoneme either before or after the phoneme of interest, which is why the "position" argument was required. In addition, if the phoneme of interest occurred at the beginning of a word and the position was before the

phoneme, or if it occurred at the end pf a word and the position was after the phoneme, the pound sign (#) would be appended to the list to signify the extremity of a word. The full function is included in Figure 2.4.3:

Fig. 2.4.3

```
def get_environment(words, symbols, position):
    environment=[]
    for w in words:
        for i in range(len(w)):
            if w[i] in symbols:
                if position=="front":
                    if i==0:
                        environment.append("#")
                    else:
                        environment.append(w[i-1])
                else:
                    if i==len(w)-1:
                        environment.append("#")
                    else:
                        environment.append(w[i+1])
    return environment
```

4. def get_change()

The function `get_change()` was built to allow the machine to decide both what set of phonemes constituted as the underlying set and whether the phoneme was affected by a phoneme directly preceding it or directly following it. It did this by measuring the length of each list of features. The function takes a list of words and a two lists of symbols and returns a list in which the first index describes which set of phonemes represents the phonemes of interest after they have undergone phonological change and the second describes the position in which the phoneme that affects the change resides. Of note is the fact that this is the first function that implements another function, `get_environment()`. The arguments of the function `get_change()` are passed through `get_environment()` to acquire the environment for each set of phonemes. The two lists that contain allophones that have the longest list of common

features within themselves will show the position of the phoneme that affects the change, due to the fact that these commonalities show that a change in these common features is affecting the change. Of these two lists, the list which contained the most allophones was considered to belong to the set of phonemes that was the underlying set, because it means that the underlying phoneme is being used more. While this is not always accurate, is the best indicator with the given information of which set of phonemes is the underlying one. The full function is included in Figure 2.4.4:

Fig. 2.4.4

```
def get_change(words, all1, all2):
    all_env=[get_environment(words, all1, "front"),get_environment(words, all1,
"back"),get_environment(words, all2, "front"),get_environment(words, all2, "back")]
    if all_env.index(max(all_env))==0 or all_env.index(max(all_env))==2:
        position="front"
    else:
        position="back"
    if all_env.index(max(all_env))==0 or all_env.index(max(all_env))==1:
        change=all2
    else:
        change=all1
    return [change, position]
```

E. Creating the Rule

The function `make_rule()` was built to generate the rule given the input information. It takes a file name and two sets of allophones and returns a phonological rule. The `commonalities()` function is then used to find the features in common between the two sets of allophones of interest. Then `store_data()` and `get_change()` are used to identify what position the phoneme is in that affects the change as well as which set of phonemes is the underlying one. Then `commonalities()` and `get_environment()` to find the common features within the lists of phonemes directly adjacent to the phonemes of interest.

Then, after `clean()` is used to eliminate redundancies in the feature lists, `differences()` and `commonalities()` to capture the change that is happening and to place it in the proper format, creating the first half of the rule. Then the previous information acquired regarding the environment of the phonemes of interest is used to create and format the second half of the rule. The full function is included in Figure 2.5:

Fig. 2.5

```
def make_rule(fname, all1, all2):
    common=commonalities(all1+all2)
    data=store_data(fname)
    info=get_change(data,all1,all2)
    enviro=commonalities(get_environment(data, info[0], info[1]))
    rule=str(clean(common))+"-->"+str(differences(common,commonalities(info[0])))+"/"
    if info[1]=="front":
        rule+=str(clean(enviro))+"___"
    else:
        rule+="___"+str(clean(enviro))
    return rule
```

F. User Input

With all of these functions built, the user is asked for their data file and two lists separated by spaces of the phonemes of interest. The phoneme sets can contain one or more phonemes. One of the phoneme sets is the underlying set while the other set is that of the phonemes resulting once the underlying undergoes phonological change, although it is the job of the program to determine which one is which. The code is the following:

Fig. 2.6

```
fname=input("Enter file name of sample words: ")
all1=input("Enter first set of allophones of interest separated by spaces: ")
all2=input("Enter second set of allophones of interest separated by spaces: ")
print(make_rule(fname,all1.split(),all2.split()))
```

III. The Testing Data, Yield, and Comments

For testing data. I had to find data that would fit within the parameters of the capabilities of the program. These parameters were 1) that all phonemes used in the data were also in the reference and 2) that the phonological change was affected by a phoneme directly adjacent to the phoneme that underwent the change. Three of the sets of data, Ganda liquids, German fricatives, and Greenlandic vowels, were taken from a workbook from Dr. Adam Ussishkin's Introduction to Phonology class at the University of Arizona. One data set, a Georgian set, came from Michael Kenstowicz's and Charles Kisseberth's *Generative Phonology: Description and Theory* (Kenstowicz 42). All four of these data sets met the requirements for this program. One data set, and English data set, was created by myself. To describe the process and the answer yielded by the program, I will use the Ganda liquids data set as an example. The data set is displayed in Figure 3.1:

Fig. 3.1 (Ussishkin 47)

kola
lwana
buulira
lja
luula
omugole
lumonde
eddwaliro
olungada
olulimi
wulira
beera
jukira
erjato
omuliro
effirimbi
eraddu
wawaabira
lagira

When the program is run, the following user prompts appear on the screen, each appearing after the previous one has been answered:

Fig. 3.2

```
Enter file name of sample words:
Enter first set of allophones of interest separated by spaces:
Enter second set of allophones of interest separated by spaces:
```

It is important to note that the user is expected to already know which phoneme(s) is experiencing a change. In the Ganda liquids set, I am already aware that “l” and “r” are the phonemes involved in the change, although I am not aware of which phoneme is the underlying one. In this particular instance, each set of phonemes contains only one phoneme; however, this

is not always the case. In some instances, a vector of phonemes may yield another vector of phonemes after undergoing a phonological change. This is displayed in the English data set: both the unvoiced and voiced alveolar plosives (t and d) yield the unvoiced and voiced alveolar affricates (ʧ and ʤ) respectively when followed by the retroflex alveolar liquid (ɹ). For this purpose, the program was built to take vectors of phonemes as arguments instead of just single phonemes. However, in the Ganda liquid set, the vector size of each phoneme set is one, and so the prompts are entered in the following manner:

Fig. 3.3

```
Enter file name of sample words: ganda_liquids.txt
Enter first set of allophones of interest separated by spaces: r
Enter second set of allophones of interest separated by spaces: l
```

In its final form, the program yields the following rule given the above data:

Fig. 3.4

```
['+consonantal', '+sonorant', '+approximant', '-dorsal', '-
labial', '-pharyngeal', '+anterior', '-distributed',
'+continuant', '-nasal', '+voice', '-strident']-->['+coronal',
'-lateral']/['-consonantal', '-low', '-back', '-round',
'+ATR']___
```

Ultimately, this rule is passable as a phonological rule. The program correctly identifies which phoneme is the underlying phoneme, whether or not the phoneme affecting the change

occurs before or after the changing phoneme, which features in the phoneme affecting the change actually cause the change, and what features of the changing phoneme are altered during the course of the change. Despite the use of the function `clean()` to eliminate the redundancies in the data, not all redundancies are eliminated. This is partially due to the fact that there are a great many redundancies within the features that can be applied to a phoneme, but mostly due to the fact that some features are redundant in some languages while they are not in others. For example, in English, identifying a vowel as (-consonantal, +voice) is redundant because, in English, all vowels are voiced. However, there are some languages in which some vowels are not voiced, and so this redundancy cannot be applied universally. Unless a function is built to determine what features are relevant in a language based simply off of the sparse data sets the program is given, these redundancies will have to be included in the final rule, making it much more difficult for the user to understand the information that the program is yielding on an intuitive level. This problem was also a prominent issue in all of the other data sets that were run.

IV. Moving Forward

Although the elimination of redundancies has already been discussed, there is another immediate improvement that could be made: the program's ability to identify which phoneme is underlying. Four out of the five data sets that were tested captured this correctly; the Georgian set did not. Unlike many other facets of the program, this identification is not based on a foolproof mathematical formula—the characteristics used to decide upon the underlying

phoneme and the position of the phoneme were the amount of times the phoneme occurred in the data and the amount of features the phonemes in each position had in common, respectively. While these were good features to work, there was still possibility for error. It is possible that supervised machine learning using annotated data for both of these aspects of the program could help the machine estimate this information more accurately. This would involve hand-annotating a large amount of example data sets as to which phoneme was underlying and which position was pertinent to the change, training the machine on this annotated data, and then testing the machine's accuracy on data for which the machine is not given the annotations. This route would be very labor-intensive. In addition, as discussed previously, code could be written to read the languages redundancies in features. This would also be prone to error and would require a lot more data for each language then is provided in the sets used in this project. However, if both of these issues were to be addressed the program's usability would vastly improve.

Ideally this program could be open to the public through the web. This would require some code to be written in HTML and for a Graphical User Interface (GUI) to be used, but this would increase user-friendliness and would allow for the rule to be more easily understood once it is returned.

V. What Has Been Learned

This was the first program I ever wrote for which the initial purpose of the program as well as the steps that would be required to build a program that would perform the required task

originated from me, so it was indeed a great learning experience for me. There are many considerations that go into designing a program oneself that are not necessary when the task has simply been given and it is one's goal to accomplish it. For example, I first had to reflect on whether or not my level of skill or, indeed, any level of skill at coding could accomplish the task of writing a phonological rule. When I first began learning about phonological change and was practicing finding phonological rules in language data, it occurred to me how very mathematical the process was, and that it would be quite possible to write a program to accomplish the task that is currently within the domain of human labor. Then I had to break down the process that I had learned regarding finding phonological rules and organize it in a way that could be utilized by the machine.

Of course, I quickly learned that there are many processes that take place in the human mind within the matter of milliseconds that do not come intuitively to a machine. While the basic process of identifying phonological rules within data can be easily handled by a machine, there were many steps involved that I really hadn't even considered, such as eliminating redundancies in the features and deciding which phoneme was underlying and which position was pertinent to the phonemic change. These processes required much more inference based on previous data than I had realized, and I was only forced to confront this in the process of building this program.

In addition, I was not aware of the large amount of human labor that goes into a project like this at the time I began it. Since I began this project I have become much more acquainted with natural language processing and have been introduced to the type of hand annotation that is required in this line of work, but when I first began this thesis I was wholly unaware of just how many phonemes and features were available to the human vocal apparatus and how massive a

task it would be to not just include them in the reference but even identify all of them in the first place. This project gave me an appreciation for the depth of the field of phonology and how the information in my project doesn't even begin to scratch the surface of all of the intricacies of phonological change in naturally-occurring human languages.

Beyond this, there were many issues that I came across in coding this program that I had never encountered before, the most prominent among these being that of Unicode encoding. I am much better versed in Unicode now, which can only help me in the field of computational linguistics where language data is used all of the time and is not necessary perfectly made for the program that is using it.

VI. Conclusion

The building of this program has in the end proved to be a very illuminating task for me. Not only did it stretch my programming skills past the level at which they had been at the beginning of this process, but it also taught me more about the field I am working in as well as about the initiative and process involved in building one's own program. I also now realize that improvements can be made in any process or program, and that there is always a new step that could be taken or a new capability that could be added to the program. These are lessons that I will certainly be able to use as I continue in the field of human language technology.

Appendix of Data Sets

Greenlandic Vowels (Ussishkin 50)

ivnaq
imaq
itumaq
iseʀaq
qasaloq
qilaluvaq
sakiak
oʀpeq
maʀaq
iperaq
tuluvaq
nanoq
seʀmeq
ikusik
qatigak
ugsik
neʀdloq

German Fricatives (Ussishkin 46)

axt
ho:x
ʁaʊxən
ʃaxt
laxən
bu:x
lɔx
ɪç
ɛçt
kɔɪçt
ʁaɪçən
lɪçt
lɛçəln
by:ça
lœçɐ

Georgian Laterals (Kenstowicz 42)

łamazad
leło
łxena
kbiłs
zarali
kała
pepeła
kleba
ert^{h}xeł
xeli
xoło
c^{h}ec^{h}xli
vxlec^{h}
c^{h}oli

English Alveolar Plosives

ʧɹeɪn
ʧɹæp
ʧɹeɪl
əʧɹɪbjut
ʧɹaɪ
ʧɹɪk
æʧɹəfi
eɪʧɹiəm
timz
its
mut
tulz
teɪm
reɪts
stul
stil

References

Kenstowicz, Michael J. and Charles W. Kisseberth. *Generative Phonology: Description and Theory*. New York: Academic Press, 1979.

Parker, Steve. "Handouts for Advanced Phonology." Dallas: GIAL and SIL International, 2016.

Ussishkin, Adam. "Linguistics 315--Introduction to Phonology: Workbook." Tucson: University of Arizona, 2015.