PyILM

June 7, 2016

Note

This document is the second chapter of my 2016 dissertation, with small changes, so it doesn't read like a typical user manual. Also, you may find some references to "previous chapters", which are not included here.

Contents

1 Overview of PyILM

This chapter details PyILM, a computer program written in Python for simulating language transmission, with a focus on phonology. PyILM's design is informed by theories of sound change through misperception (e.g. Ohala (1983), Blevins (2004)), and its formal implementation is based on the Iterated Learning Model (e.g. Smith et al. (2003), Kirby (2001)).

PyILM is an "agent-based" model, meaning that it involves virtual individuals who can interact with each other in limited ways. Each agent learns a sound system by listening to the output of another agent, and then re-transmits what they learned (including any potential errors) to another agent. PyILM allows users to manipulate numerous parameters of this process and run iterated learning simulations to explore how sound change happens under different conditions. Section 2.2 of this chapter gives a complete list of the parameters that a user can set. Section 4 gives some more details on how to use and configure a simulation.

1.1 Iterated Learning Models

Computational models of iterated learning, e.g. Kirby (1999, 2001), follow this basic pattern of nested loops:

```
1 Generate a speaking agent
2 Generate a learning agent
3 Loop x times:
4     Loop y times:
5         The speaker produces an utterance
6         The learner learns from this utterance
7     Remove the speaker from the simulation
8     Make the learner the speaker
9     Create a new learner
```

This pattern represents x generations of language transmission with y learning items at each generation. The corresponding loops for PyILM are given in Algorithm 1. These loops are explained in "pseudo-code", which I will continue to use throughout the chapter to explain the logic of PyILM. Pseudo-code is text consisting of valid Python expressions, most of which appears as actual lines of code in PyILM. However, some of the code has been changed to make it more readable in the context of a dissertation.

Algorithm 1 Main simulation loop

```
1  Simulation.load("config.ini")
2  speaker = BaseAgent()
3  Simulation.initialize(speaker)
4  listener = Agent()
5  for generation in range(Simulation.generations):
6      for j in range(Simulation.words_per_turn):
7          word = speaker.talk()
8          word = Simulation.transmit(word)
9          listener.listen(word)
10     Simulation.record(generation)
11     speaker.clean_up()
12     speaker = listener
13     listener = Agent()
14 Simulation.generate_output()
```

Line 1 loads user-provided details about the simulation and configures PyILM appropriately. Line 2 creates a new agent for the first generation of a simulation, and Line 3 seeds it with an initial lexicon and inventory. Line 4 creates a new "blank" listener who will learn her language from the speaker. To be clear, this initialization phase is not intended to represent any actual events in language transmission. PyILM only simulates the "evolution" of language in the sense that it simulates how languages change over time; it does not simulate the emergence of language from non-language. The first speaker in the simulation represents some speaker at some point in the history of some language. Note that the first speaker is formally a different kind of object in the program than the other speakers, since the first speaker requires a set of initialization functions, while later generations rely on learning algorithms.

Line 5 starts a loop that runs once for each generation being simulated (see section 2.2.2). Line 6 starts a loop that runs once for each word a learner hears.

In line 7 a speaker chooses a word to say (see the production algorithm, section 3.4). Line 8 simulates misperception by changing some of the segments of the word (see section 2.10). Misperceptions are context-sensitive, and probabilistic, so sometimes nothing at all happens on this line. On line 9 the learner learns from this new word (see the learning algorithm, section 3.1).

On line 10, PyILM keeps a record of what the speaker's inventory and lexicon look like before the speaker is removed from the simulation on line 11. On line 12, the new speaker creates some probability distributions to be used during the next production phase. Line 13 generates a new listener for the next loop to start over again.

Line 14 is executed only after the final generation of the simulation has finished learning. It prints a report of what happened during the simulation, using the information logged at each generation on line 10. The program then terminates.

2 Objects

2.1 Overview

PyILM was written using an object oriented approach to programming. An "object" is way of representing a concept in a computer program. In the case of PyILM, objects represent concepts relevant to sound change or phonology. Objects have *attributes* which represent properties or characteristics of the objects and their values may be fixed or mutable. Objects also have *methods* which describe what the object can do. An example of an object is the Feature object, which represents the concept of a distinctive phonological feature. Feature objects have two attributes: `name` and `sign`. These are both strings, with `name` having a value of something like "voice" or "continuant" and `sign` having a value of either "+" or "-". They also have an `equal_to` method, which is used to decide if two Feature objects are the same or not.

This example also illustrates two typographical conventions adopted in this chapter. First, the names of objects in the computer program are written with an initial capital letter to distinguish them from the use of the same words to refer to concepts in linguistics, e.g. the Feature object vs. distinctive feature. Second, the `typewriter font` is used when referring to object attributes and methods.

This section explains the main objects in the simulation, as well as their relevant attributes. Details about object methods are, for the most part, omitted here because they are generally not of relevance for understanding how the simulation works. One exception to this is the Agent object, which has methods for speech production and learning that are important to understanding the simulation. Examples of omitted methods include: methods for making equal to and not equal to comparisons, methods for generating string representations, and methods for reading and writing to files.

There are nine objects discussed in this section: Simulation, Word, Segment, Feature, FeatureSpace, Sound, Token, Agent and Misperception. To understand the relationship between them, it is useful to think of the objects in PyILM as being "stacked" inside one another. The diagram in Figure 1 is a visualization of this. Note that the figure is not a description of object inheritance - it is a visualization of how the objects fit together conceptually.

Every run of the simulation creates a new Simulation object, inside of which there are Agent objects that talk and listen to each other. Agents all have a `lexicon` attribute which contains Words, which are made up of Segments, which are made up of Features.

A speaker uses a production algorithm to transform Segments into a different kind of object called a Sound, representing an actual speech sound instead of a unit in the mental lexicon. The phonetic characteristics of speech sounds are represented by objects called Tokens.

The simulation passes Sounds through a misperception function, which may alter some of their Tokens' values, depending on the environment in which they occur. Then these Sounds are sent to the listener's learning algorithm which

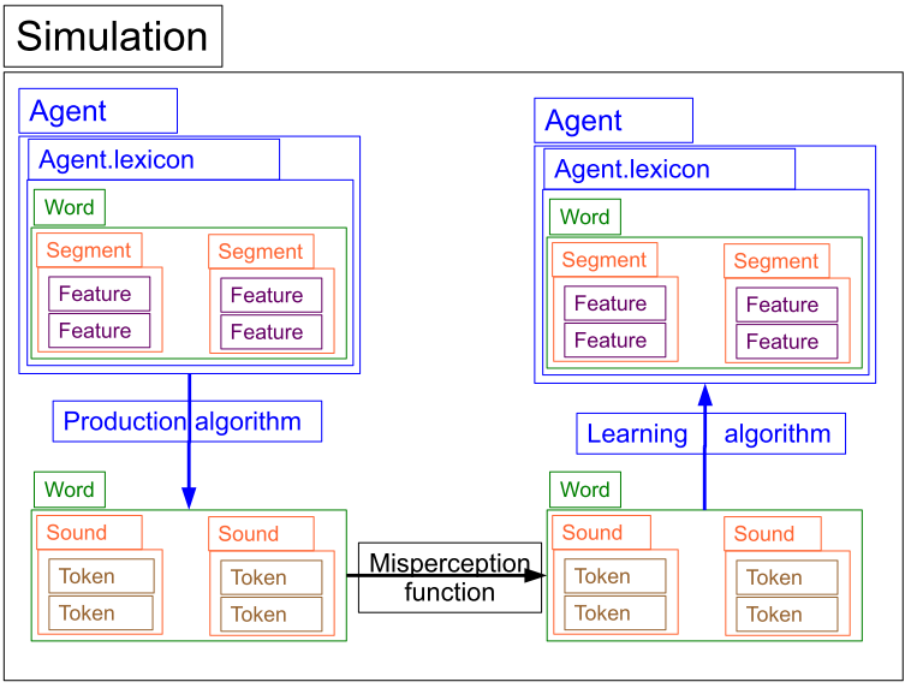

Figure 1: The objects of PyILM

creates new Features and Segments.

The transmission of a single segment is illustrated in more detail in Figure 2. A speaker starts by selecting a word from the lexicon. The lexicon is a list of meanings, each associated with a string of segment symbols, each of which are "translated" into a set of phonological features. All possible features that can be discriminated are kept together in a FeatureSpace object, where each feature is represented as an interval [0,1]. Figure 2 shows only three feature dimensions (F_1, F_2, and F_3), with each dimension represented as a number line. The points circled with solid lines are the range of values that represent [−feature] segments, and the points circled with dashed lines are the [+feature] values. The lines in black represent the values a speaker experienced during their time as a learner. The lines in red represent the particular phonetic values that the production algorithm chose on this occasion. Supposing that F_1 is [voice], F_2 is [continuant] and F_3 is [nasal], then the segment is [+voice, −continuant, −nasal]. This could be represented as /b/ (among other symbols).

The values chosen by the production algorithm are then sent to the misperception function. In this example the environment was right for a misperception to occur, and the listener is going to hear the F_1 value - the voicing value - of this segment as lower than intended by the speaker.

The red lines in the learner's FeatureSpace represent where the values were

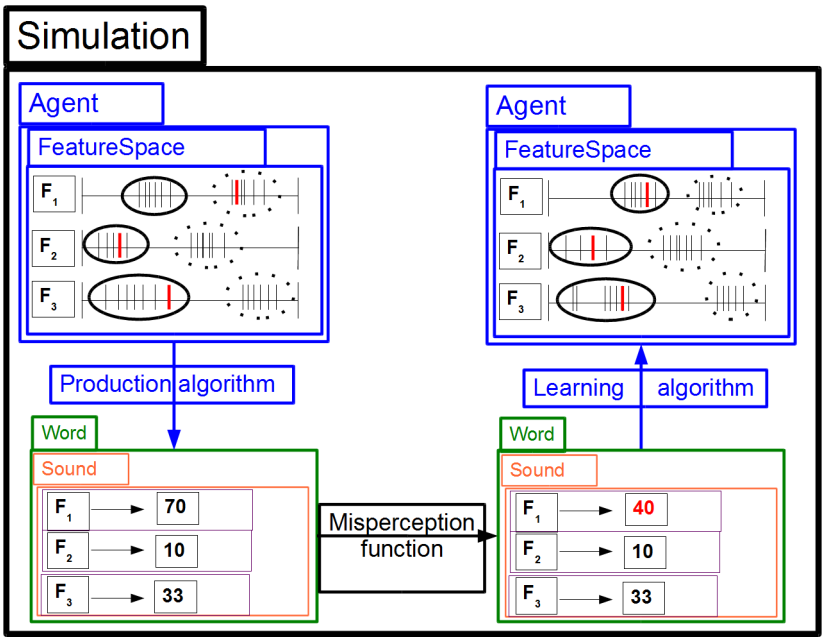

Figure 2: The transmission of a phonological segment

stored. In figure 2 the learner has already heard a number of words, and has formed some phonological categories. The boundaries will continue to shift over the course of learning.

The values for F_2 and F_3 are interpreted "correctly" by the listener, that is, her learning algorithm categorizes them as examples of the same phonological feature value as in the speaker's FeatureSpace. Due to the misperception, F_1 is categorized as an example of the opposite feature value.

2.2 Simulation

2.2.1 Overview

Each time PyILM is run, a single Simulation object is created, and the entire simulation runs "inside" of this object. The Simulation object has methods for initializing simulation details, creating and removing agents from the simulation, causing agents to speak or listen, causing misperceptions to occur, and writing the results of the simulation to file. None of these methods are discussed here. Details about Agents (2.9) and Misperceptions (2.10) can be found in their own sections.

The Simulation object has 10 attributes representing factors relevant to cultural transmission. These are discussed here, and it is possible for users to set the value of any of these attributes. See section 4 for details on how to do this.

2.2.2 `generations`

The value of this attribute determines how many generations of listeners the simulation should run for. The default value is 30. This means the simulation stops after the 30th learner has finished learning.

2.2.3 `initial_lexicon_size`

This value of this attribute sets the number of words in the initial lexicon. The default value is 30. The words of the initial lexicon are created using the invention algorithm (see section 3.5).

2.2.4 `initial_inventory`

This attribute controls the size and contents of the initial segment inventory of the simulation. The value supplied to should be a list of segment symbols separated by commas. For instance, p,t,k,b,d,g,f,s,h,m,a,i,e would be an acceptable starting inventory. The set of symbols used should correspond with symbols in a feature file (see the `features_file` attribute in section 2.2.9).

If some degree of randomness is desired, then the value supplied to `initial_inventory` should be two numbers separated by a comma. The first number represents the number of consonants and the second the number of vowels. There must be at least one consonant and at least one vowel in every simulation. The segments are randomly selected from the feature file (section 2.2.9). The simulation determines what is a consonant and what is a vowel by checking the value of the [vocalic] feature : [+voc] segments are called "vowels" and [−voc] segments are called "consonants".

If no initial inventory is supplied, then the default value is 10 random consonants and 3 random vowels.

2.2.5 `minimum_repetitions`

During the production phase, the speaking agent will produce every word in the lexicon at least `minimum_repetition` times. For example, if set to 2, then every word will produced at least twice. The default value is 1, and it cannot be set any lower.

Words in a lexicon are grouped into "frequency blocks", with the first block containing words that are produced exactly `minimum_repetition` times. Each successive block that is created has a frequency of twice the previous block. Doubling the frequency approximates a Zipfian distribution of words in natural language (Yang 2010). This blocking is done randomly for the first generation in a simulation. If any words are invented during the simulation (see section 2.2.11) they go into the least-frequent block.

In natural language, a similar Zipfian distribution holds of individual words: the most frequent word in a language is about twice as frequent as the next one, which is is twice as frequent as the next, and so on. Early testing with PyILM found that implementing frequency distributions on a per-item basis resulted in

simulations that ran for far too long. By grouping words into frequency blocks, each of which is twice as frequent as the next, the running time of the simulation is greatly improved while still maintaining a Zipf-like distribution.

2.2.6 `min_word_length`

This value sets the minimum number of syllables a word must have. The default value is 1, and it cannot be set lower.

2.2.7 `max_word_length`

This value sets the upper bound for the number of syllables a word can have. The default value is 3, and it must be equal to or greater than `min_word_length`. These min and max values are used by the agent's invention algorithm when generating new words (see section 3.5).

2.2.8 `phonotactics`

The `phonotactics` attribute is used by an agent's invention algorithm (see section 3.5) for creating new words. Invention happens in every simulation run to generate the lexicon of the very first agent. Invention otherwise only occurs if `Simulation.invention_rate` is set greater than zero (see section 2.2.11).

The value supplied to this attribute should be a single string that consists of only the letters "C" and "V". This string represents the maximal syllable structure. By default, PyILM assumes that all possible sub-syllables should be allowed. For instance, if the value supplied is "CCVC", then the set {V,CV,CCV,VC,CVC,CCVC} will serve as the set of possible syllables. The simulation determines which segments are consonants and which are vowels by looking at the segment's value of [vocalic]. [−voc] are treated as "C" and [+voc] are treated as "V".

It is possible to exclude a subset of syllables by listing them after the maximal form separated by commas. For instance, if the string supplied to this attribute is "CCVC,CCV,VC" then the simulation will use the set {V,CV,CVC,CCVC}. This is the same as the set above except that CCV and VC have been removed.

The phonotactics of a language are fixed for a given simulation. This is why `phonotactics` is an attribute of the Simulation object, as opposed to being an attribute of the Agent object. The phonotactics play a role in the outcome of a simulation (since sound-changing misperceptions are context-sensitive and phonotactics defines the set of possible contexts) so it is useful to hold them invariant for a simulation to understand their effect on sound change. The following chapter describes the output of a simulation, and there is more discussion of the specific effect of phonotactics.

The default value is "CVC".

2.2.9 `features_file`

The value supplied for this attribute should be the name of a text file which gives a phonological feature description for possible segments. PyILM comes with a default features file that describes several hundred segments using more than a dozen features, and will be sufficient in most cases. This file is a based on the ipa2spe file available in P-base (Mielke 2008). However, users can modify this file or write their own. Each line of the file must have first a symbol, then a list of phonological features, all separated by commas. An example of such a file, with a very small feature space, is given in Figure 3.

```
i,+voice,+cont,+voc,+high
ɑ,+voice,+cont,+voc,-high
ɣ,+voice,+cont,-voc,+high
iʔ,+voice,-cont,+voc,+high
i,voice,+cont,+voc,+high
v,+voice,+cont,-voc,-high
,+voice,-cont,-voc,+high
iʔ,-voice,-cont,+voc,+high
b,+voice,-cont,-voc,-high
k,-voice,-cont,-voc,+high
p,-voice,-cont,-voc,-high
ʔ,-voice,-cont,+voc,-high
f,-voice,+cont,-voc,-high
,-voice,+cont,+voc,-high
x,-voice,+cont,-voc,+high
ɑʔ,+voice,-cont,+voc,-high
```

Figure 3: Sample feature file

The features provided in this file define the dimensions of phonetic and phonological space that can be used by a language. Every Segment in an agent's lexicon in the simulation consists of some set of phonological features (see section 2.4.2). Every Sound uttered by an agent consists of a set of multi-valued phonetic features (see 2.7). The names of the features used in both cases are the same, and they are determined by the names in `features_file`.

The symbols in this file largely serve as a kind of user interface. Sounds are actually represented in PyILM as numbers in a list, but this representation is unhelpful for a human. If PyILM has to print a symbol, for example when producing a report of the simulation, it uses these symbols as a more readable representation.

2.2.10 `max_lexicon_size`

This sets a maximum size for the lexicon of the language. If the lexicon has reached maximum size and invention is required, then one of the least frequent

words in the lexicon is selected (at random) and removed from the language, to be replaced by the newly invented word. The default value is 30.

2.2.11 `invention_rate`

This represents the probability that a speaker will produce words that were not in her input. This could represent new words that have come into fashion during the speaker's life, coinages that she created herself, borrowings, or any other source of new words. Note that invention never introduces new sounds, so if inventions are considered to be like borrowings, then it would be a case of complete adaptation to the native sound system. Further, invented words always match the phonotactics of the language (see section 2.2.8). The number of invented words is determined by the `max_inventions` parameters (see section 2.2.12).

The value of this attribute must be between 0 and 1. If the value is set to 0, then agents never invent words and only the words used in the first generation will be the ones used throughout (though, of course, their phonological and phonetic properties may change). If the value is set to 1, then new words will join the lexicon each generation. See section 3.5 for more details on the invention algorithm. The default value is 0.

2.2.12 `max_inventions`

The invention phase only happens once for each agent, at the beginning of the production phase. Pseudo-code for this is given below. During the invention phase, the simulation will make `max_inventions` attempts to generate a new word and add it to the lexicon. The default value for this attribute is 0 (i.e. no inventions ever occur). The probability of a word actually being invented is set by the attribute `invention_rate` (see section 2.2.11).

```
for j in range(Simulation.max_inventions):
    n = random.random(
    #generate a random number in [0,1]
    if n <= Simulation.invention_rate:
        word = agent.create_new_word()
        agent.update_lexicon(word)
```

In a simulation where `max_inventions` has been set to X, and `invention_rate` is set to Y, the probability that X new words actually are invented at a given generation is X^Y for $X > 0$. The probability that no new words are invented is $(1-X)^Y$. Suppose that `max_inventions` is 3 and `invention_rate` is 0.2. This means that for any generation, no more than 3 new words can enter the lexicon. The probability that only 2 news words are invented is $0.2 \times 0.2 = 0.04$. The probability that no new words enter the lexicon is $0.8 \times 0.8 \times 0.8 = 0.512$.

2.2.13 `misperceptions`

This should be the name of a text file, in the same directory as PyILM, that contains a list of misperceptions that could occur over the course of a simulation. Each line should have six arguments separated by semi-colons. The PyILM Visualizer also contains an option for creating misperception files with a more intuitive user interface. See section 2.10 for more details on Misperception objects.

The first argument is the name of a misperception. The second argument is a list of phonological features, separated by commas, that describe segments that can be altered by the misperception. The third argument is the name of a feature which undergoes change if the misperception occurs. Multiple values for the third argument are not permitted, and misperceptions can only change one feature at a time. The fourth argument is a number in the interval (0,1) representing how much the phonetic feature changes if the misperception happens. The fifth argument is the environment in which the misperception can occur. Environments should be specified at the level of phonological features. A special value of * can be used to mean "context-free". The sixth and final argument should be a number in (0,1) representing the probability that the misperception occurs. The default set of misperceptions are in file called "misperceptions.txt" that is bundled with PyILM and users can also consult this file for an example of the format to follow.

```
pre-nasal nasalization bias;-nasal,+voc;nasal;.05;_+nasal,-voc;.15
word final devoicing;+voice,-nasal;voice;-.1;_#;.2
intervocalic lenition;-cont,-son;cont;-.1;+voc_+voc;.1
default vowel voicing;+voc;voc;.05;*;.75
```

In the first example, the misperception targets non-nasal vowels. The nasal value of these vowels is raised by 0.05 when these vowels appear before nasal consonants, and on any given utterance where such a context appears there is a 0.15 chance this happens. The second example targets voiced non-nasal consonants. These sounds have their voicing values decreased by 0.1 when they appear word finally, and this happens with a 0.2 probability.

The final example shows a context-free misperception, which is referred to as a "bias". In this case, vowels have their voicing values raised slightly under all circumstances. This allows vowels to be affected by the final-devoicing misperception, but not to the same extent as consonants because their voicing value gets raised a little bit anyway. (Of course, vowels could be completely excluded from the final devoicing misperception by changing the targeted segments to be include [−voc] only.) In Chapter 5, the effects of misperceptions and biases are explored in more detail.

The amount by which a misperception changes a sound cannot be greater than 1 or less than -1. This is because phonetic values in PyILM must be numbers between 0 and 1. If the effect of misperception would push a value above

1, then PyILM will force the value back down to 1. Similarly if a misperception were to push a value below 0, PyILM will raise the value back up to 0.

2.2.14 `minimum_activation_level`

This attribute should be a number in [0,1] representing how close to an existing category an input examplar must be, in order to be considered for membership in that category. Technical details of the exemplar learning algorithm are given in section 3.1.

Setting this number closer to 1 sharpens an agent's discrimination, and permits smaller distances between categories. This leads to inventories with more segments and less variation in pronunciation. Setting it all the way to 1.0 means that an input examplars only count as a member of an existing category if they have phonetic values that match exactly to all other exemplars in the category. This is a rare occurrence, so inventories tend to grow rapidly with this setting.

Setting this number closer to 0 blurs an agent's discrimination and increases the distance required between categories. Setting it all the way to 0 means that there is no distance between categories, and every input sound after the first counts as an example of whatever the learner first heard. This leads to an immediate collapse in the segmental inventory and the it reduces to a single segment within a generation.

2.2.15 `auto_increase_lexicon_size`

The attribute `initial_lexicon_size` (see section 2.2.3), is used to determine the number of words in the lexicon of the initial generation. These words are all randomly generated, using the set of sounds supplied to the `initial_inventory` (section 2.2.4) attribute. However, in this randomness, it sometimes happens that not all of the initial sounds actually make it into a lexical item. If `auto_increase_lexicon_size` is set to True, then PyILM will continue to generate words for the initial lexicon until every sound occurs at least once, even if it means surpassing `initial_lexicon_size`. If auto-increasing is set to False, then the lexicon size remained capped at the the initial value.

2.2.16 `initial_words`

This parameter allows the user to submit a list of words, separated by commas, that should appear in the lexicon of the initial generation. It is the user's responsibility to ensure that the words contain symbols which appear in the initial lexicon of the language. If a word supplied to `initial_words` contains a symbol not in the initial inventory, then PyILM will raise an exception and stop running. In short, this parameter cannot safely be used in combination with a randomly selected started inventory (see section 2.2.4).

It is possible for a user to create words that do not conform to the phonotactics of the language with this parameter, although this is not recommended as it may cause unexpected behaviour during the simulation.

The initial lexicon is guaranteed to include every word supplied to `initial_words`, even if this means going beyond the lexicon size supplied to `initial_lexicon_size` (section 2.2.3). If this occurs, PyILM will also not enforce the `auto_increase_lexicon_size` (section 2.2.15) parameter, and the initial inventory will consist only of the sounds found in the `initial_words` list. If. on the other hand, the number of initial words is smaller than the initial lexicon size, PyILM will continue to randomly generate words until the lexicon is an appropriate size, and the `auto_increase_lexicon_size` parameter works as expected.

By default, this option is not turned on and the initial lexicon will consist of randomly generated words.

2.2.17 `allow_unmarked`

Normally, sounds in a PyILM simulation are represented as lists of binary features, marked as either [+feature] or [−feature]. If the `allow_unmarked` option is set to "True", then a third feature value "n" is allowed (this is the "unmarked" value). If a sound is marked [nfeature], it means that every instance of that sound experienced by a learner had a phonetic value of 0 on a given feature dimension (see sections 2.5 and 2.6 for more details on how features work in a simulation). In practice, [nfeature] would be used in cases where a feature does not apply all to a sound, e.g. a glottal stop could be marked [nlateral] because the tongue body is not involved whatsoever in the articulation of a glottal stop.

Note that [nfeature] is not equivalent to [−feature], even though sounds with either feature value will have low phonetic values on particular dimension. If the `allow_unmarked` option is used, it is important to ensure that misperceptions are formatted properly (see section 2.10), and specifically make reference to [nfeatures] if desired.

The default value of `allow_unmarked` is "False", meaning that only binary features are used in a simulation. Every simulation reported in this dissertation was run with the default value for this option.

2.2.18 `seed`

This attribute controls the random seed used in PyILM. Its value can be any number or string. By default is it a randomly selected integer in [1,10000]

2.2.19 `seg_specific_misperceptions`

This parameter used to create a special kind of misperception, for the purposes of a testing a hypothesis about feature economy to be presented later in this dissertation. Details are given in Chapter 5, Section 5.4. This parameter takes a value of either True or False, and the default is False.

2.3 Words

Words are generalized objects that represent either an entry in a mental lexicon or an utterance of one of these lexical items. Words have two attributes: `string`, which is a list representing the segmental melody of the word, and `meaning`, which is an integer.

2.3.1 `string`

If a Word is in a lexicon, then the `string` attribute is an ordered list of Segments (see section 2.4) representing the segmental content of a word, plus two word boundary symbols "#". The value of `string` in this case is learned, and updated, as part of the learning algorithm (see section 3.1).

If a Word represents an utterance, then `string` is an ordered list of Sounds (see 2.7). In this case, the value of `string` is determined by the production algorithm (see section 3.4).

2.3.2 `meaning`

The `meaning` attribute is just an integer. The meaning attribute in used when the learning algorithm checks if an input word means the same thing as a known lexical item. Two words are considered to "mean the same thing" if their `meaning` attributes compare equal. This only very roughly models the concept of "meaning", and is certainly not representative of what a real human speaker knows about the meanings of words. This simplification is sufficient for the purposes of modeling sound change.

The first word invented in the simulation is assigned a meaning of 0. Then a counter is started, and each new word is assigned the next highest integer. One consequence of this process of generating meanings is that there can be no poly-morphemic words in any language. Every word has a single meaning, so any language generated by PyILM is completely isolating.

Another consequence is that no synonyms will ever appear in the language. If a listener hears two words that mean the same thing, it will always be two instances of the same word. The counter doesn't run backwards, so speakers will never invent a new way of saying something they can already say. There can be variations in the pronunciation of a word, due to phonetic effects or misperceptions, but there can be no completely unrelated forms with the same meaning.

2.4 Segments

Segments represent mental categories, themselves representing speech sounds, that agents can learn. Segments are the units that make up the words in an agent's lexicon. They are not the actual speech sounds that agents produce and perceive. In other words, they are like phonemes, not phones. The objects representing speech productions are called Sounds, and they are described in section 2.7.

Segments are not atomic objects. They are represented by a set of phonological features (see 2.5). Features in turn are abstractions representing a ranges of values along a particular dimension of phonetic space.

The ability to segment speech is taken for granted in PyILM and learners are assumed to be able to portion out the speech signal in such a way as to form some kind of segment-like unit. The particular phonetic and phonological characteristics of segments are, of course, learned during the simulation and not pre-determined.

Segments have three attributes: `symbol`, which is an identifier for the segment, `features`, which is a list of distinctive phonological features, and `envs`, which lists all the environments the segment appears in.

2.4.1 `symbol`

Segments all have a `symbol` attribute, which is a string of Unicode characters (normally but not necessarily of length 1). The symbols are drawn from those provided to the simulation's `features_file` variable (see section 2.2.9).

Symbols are just a convenience for the simulation, and the actual choice of a symbol can be entirely arbitrary. However, experience has found that choosing segment labels at random makes it very difficult to interpret the simulation results. The natural intuition of a linguist is to assume that IPA characters are used meaningfully, so if instead they are randomly associated with features, then reading simulation results becomes a frustrating puzzle of trying to remember what, e.g. /p/ stands for this time. Instead, PyILM tries to choose a "reasonable" symbol for a given segment by selecting a symbol for a segment whose feature description most closely matches the segment under consideration. This means that in most cases, the segment symbol will match the phonetic values in a way that makes linguistic sense, although it is always safer to inspect the actual feature values and not rely on the symbol.

In some cases, sounds can appear in a simulation that have feature specification not found in the user's feature file. For instance, a user might include a misperception that nasalizes stops under some conditions. Normally, stops are [−sonorant] and nasals are [+sonorant], but this nasalization misperception can create sounds that are [−sonorant, +nasal]. PyILM needs a symbol for such a sound, and if it cannot find a perfect match in the user's feature file, it will take a sound that matches most of the features. This can occasionally lead to unexpected results when visualized (e.g. PyILM might pick a symbol for a plosive to represent the non-sonorant nasal). Again, it is always safer to check feature values than to rely on the assigned symbol, especially in simulations with a large number of misperceptions and less-predictable outcomes.

2.4.2 `features`

This attribute is a list of distinctive Features (see 2.5) that uniquely characterize the segment. These values are inferred by the agent's learning algorithm (see section 3.1), and may change over the course of learning. They are fixed after

learning ends, and do not change once the agent becomes the speaker. The actual distribution of phonetic values corresponding to a particular segment is recorded in a FeatureSpace object. These objects are described in more detail in section 2.6.

2.4.3 `envs`

The `envs` attribute keeps track of all the environments in which a segment appears. The *environment* of a Segment is defined as as the immediately adjacent Segments to the left and right. More formally, the environment of Segment in position j of a Word's string is a tuple consisting of the Segment at position j-1 and the Segment at position j+1. Words in an agent's lexicon begin and end with word boundaries, and these provide the appropriate environment for word-initial and word-final segments. Although word boundaries are formally treated as Segments, they differ from the Segments described in this section in that they lack phonological features and they are not considered part of an agent's inventory. They only exist as part of an agent's lexicon; the objects transmitted to learners do not contain word boundaries and must be re-constructed by the learner. This has no practical effect on the simulation at the moment because speakers only ever utter one word at a time.

2.4.4 `distribution`

The distribution attribute is a normal probability distribution representing the distribution over possible phonetic feature values for the segment. An agent keeps in memory one distribution for each feature dimension. These are created at the end of the learning phase.

2.5 Features

A *feature-dimension* is a one-dimensional space, a number line, representing some salient, gradable property of speech that listeners are aware of and can use to categorize speech sounds. Feature objects represent phonologically significant ranges of values, and are created by the learning algorithm (see section 3.1). These ranges represent values for phonological features, and there are three possible feature values: "+", "−", "n". The [+feature] category is for ranges of higher values, and the [−feature] category is for ranges of lower values. The actual values depend on the details of the simulation in question. A third value is [nfeature] which is assigned to segments that are always expressed with a value of 0 for the feature (e.g. a glottal stop would be [nlateral]). By default, the "n" value is not used in simulations, and all features are binary. To use it, set Simulation.allow_unmarked to True in the configuration file.

The number of feature dimensions is determined by the features listed in `Simulation.features_file` (see 2.2.9). The clustering into features is also illustrated in figure 2 on page 8.

Feature objects have two attributes: `sign` and `name`. The `sign` attribute can have value of "+" or "-", (and possibly "n") and `name` is drawn from those provided to `Simulation.features_file`. The value of "n" is only available if the `allow_unmarked` option is turned on (see section 2.2.17). This option is turned off by default.

The set of feature-dimensions is fixed at the beginning of a simulation and does not change throughout (see section 2.2.9). This set is represented as a FeatureSpace object (see section 2.6). Having a fixed set of features does not, of course, mean that every one of them will participate in a contrast for a given simulation. For instance, it is possible to for a language to have no lateral consonants in the initial generation, meaning no segments marked [+lateral], and to not acquire any over the simulation run. So long as every sound has relatively low values along the [lateral] dimension, they will all get classified as [−lateral], and the feature will not be contrastive.

2.6 FeatureSpace

A FeatureSpace object is a multidimensional space representing all possible phonetic values. Every utterable sound in the simulation can be represented by some point in this space. FeatureSpaces have f feature-dimensions, where f is the size of the set of features provided to the `Simulation.features_file` attribute (see section 2.2.9). Points in any given dimension fall somewhere in the interval [0,1]. Formally a FeatureSpace is just a Python dictionary (a hash table) where the keys are the names of a feature-dimension and values are lists of Token objects (see section 2.8) that are stored along that feature-dimension.

This has the consequence that one phonological feature in a language corresponds to only one kind of phonetic feature along a single dimension. This is unlike natural language where a range of phonetic characteristics may be related to a phonological feature. For instance, [voice] may correspond with VOT, burst amplitude, and F0 (e.g. Lisker (1986), Raphael (2005)).

2.7 Sounds

Iterated learning models place a heavy emphasis on the fact that language exists in both a mental form and a physical form. The Segment objects discussed in section 2.4 represent part of mental language. The corresponding object representing physical speech is called a Sound. The difference between a Segment and a Sound is analogous to the difference between a phoneme and a phone. Sounds are created by the production algorithm (section 3.4), further manipulated by the misperception function (section 2.10), and serve as input to the learning algorithm (section 3.1).

Sounds as objects are similar to Segments. Sounds only have `symbol` and `features` attributes, although `features` is a list of Token objects (see section 2.8), rather than Feature objects which is the case with Segments. A Sound exists for only a single event of transmission, then is removed from the

simulation. The environment in which a Sound occurs is calculated in place by the misperception function, or the listener's learning algorithm, as needed.

2.8 Tokens

Tokens represent the spoken values of Features (section 2.5), much like Sounds (section 2.7) represent spoken Segments (section 2.4). Features actually represents ranges of phonetic values, and a Token objects represents one value from that range. A Token has four attributes: `name`, `value`, `label`, and `env`.

2.8.1 `name`

The `name` attribute is the name of whichever phonetic feature this Token represents (see section 2.2.9).

2.8.2 `value`

The `value` attribute is a number in [0,1]. In a sense, `value` represents how "strongly" a given sound expresses a particular feature (whichever feature is given for the `name` attribute). Larger values are intended to represent increasingly salient or prominent information, although what this means would depend on the feature in question. What makes something more [nasal] in actual speech would depend on, e.g. nasal airflow, degree of closure, nasality of adjacent segments, etc.

2.8.3 `label`

The `label` attribute is a reference to the `symbol` attribute of one of the Segment objects in the agent's inventory, indicating that a Token counts as an exemplar of that particular segment. The values along a given feature dimension that are associated with a particular segment will change over the course of learning, and Token labels are continually updated to keep in line with changes to the inventory.

2.8.4 `env`

The attribute `env` represent the environment in which the Token was perceived. This consists of a tuple of two references, the first a reference to the Segment on the left and the second a reference to Segment on the right. These references allow a dynamic updating of the FeatureSpace as the learner's inventory changes over the course of learning. For instance, suppose a learner has perceived a word-initial Token before a vowel which gets labeled /e/. The `env` attribute of this Token would be the tuple (#,e). If the category /e/ later gets merged with another vowel category and has its label changed, perhaps to the label /i/, then the `env` of this Token would be automatically updated to (#,i).

2.9 Agents

Agent objects represent the people who learn and transmit a language. Agents have four important attributes: `lexicon`, `inventory`, `feature_space`, and `distributions`. There are also three Agent methods described in their own section: a production algorithm (3.4), a learning algorithm (3.1), and an invention algorithm (3.5).

In addition to the Agent object, there is also a BaseAgent object, which is used for agents in the 0th generation of the simulation. The two objects share many attributes and methods, and formally speaking Agent inherits from BaseAgent. These distinctions are largely unnecessary for understanding how the simulation works, however, so they are ignored here and I present all of the relevant information under the general heading of "Agent".

2.9.1 `lexicon`

This attribute represents the set of all words that an agent knows. Words in the lexicon are represented by another kind of object called a Word (see section 2.3). Words are learned and stored in the lexicon as part of the learning algorithm explained in section 3.1. Lexicons are essentially just "storehouses" of words. The lexicon is a mapping between meanings and lists of Words that can be used to convey that meaning. Each possible Word is stored alongside the raw count of how many times it appeared in an agent's input. Multiple Words can become associated with the same meaning through misperception. For instance, in a simulation with a final devoicing misperception, the meaning 17 might be associated with the list /pad (6), pat (4)/, which would indicate that [pad] was heard 6 times during an agent's learning phase, while the word [pat] was heard 4 times.

2.9.2 `inventory`

The inventory of an agent is a list of all the Segments that appear in at least one Word in the agent's lexicon. The inventory is used in learning to make comparisons between words (see section 3.1). The inventory is also used by the invention algorithm (see section 3.5), which creates new arrangements of known segments.

2.9.3 `feature_space`

This attribute just serves to point to a FeatureSpace object (see section 2.6). This object represents a multidimensional phonetic space, and every sound that an Agent can hear or produce is represented as a point in this space. The `feature_space` of an Agent is initially empty, and it gets filled with points, which are then clustered, during learning. The production algorithm makes use of these clusters for deciding on what phonetic feature values to assign to different phonological values.

2.9.4 `distributions`

An agent's `distributions` attribute is a dictionary organized first by segment, then by feature. Each feature is mapped to a probability distribution, which is sampled by the production algorithm when it needs phonetic values. This is described in more detail in section 3.4.1.

2.10 Misperception

The idea behind misperceptions is that some sounds, in some phonetic environments, are susceptible to being perceived by the learner in a different way than the speaker intended. For instance, there is a tendency for word final voiced obstruents to be pronounced in such a way as to be perceived as voiceless (Blevins (2006)). This can lead to instances of misperception where a speaker intends /bab/ and the learner understands /bap/.

Misperception objects are intended to represent factors inherent to human communication that affect perception of sounds probabilistically, in well-defined environments. These are factors that could potentially affect speech perception at every utterance, and so become relevant to the cultural transmission of sounds. For instance, speakers produce oral vowels with more nasality before nasal consonants (Chen (1997)). This fact about the pronunciation of vowels in certain environments means that in the transmission of any language with words that contain a sequence of an oral vowel followed by a nasal consonant, there is some small probability that learners will mistakenly interpret these vowels as inherently nasal, leading to a sound change where vowels articulated as oral vowels at one generation are articulated as nasal vowels in a later generation (cf. Ohala 1983)

On the other hand, Misperception objects are not intended to represent instances of misperception caused by e.g. the conversation happening at a loud concert, or peanut butter in the speaker's mouth. These factors certainly affect perception of speech sounds, but they do not occur with enough regularity to be worth including in a simulation of cultural transmission.

Formally speaking, a misperception is a probabilistic, context-sensitive change to a Token object's `value` attribute. Here are two examples

[+vocalic] → [nasal +.1] / _[+nasal], .2 ("pre-nasal nasalization")

[+voice, −son] → [voice -.15] / _#, .3 ("final-devoicing")

The first example reads as "There is a .2 chance that Tokens representing [+vocalic] Segments have their [nasal] value increased by 0.1 if they occur in the environment before a Segment marked [+nasal]". The second example reads as "There is a .3 chance that Tokens representing voiced obstruents have their [voice] value decreased by 0.15 if they occur in word-final position".

The probabilities are arbitrary and chosen for illustration. The probability of any misperception actually occurring is an empirical question, and not one that PyILM can be used to answer. Instead, users can set this value and run multiple simulations to understand how higher and lower values affect the overall course of sound change. In fact, all aspects of a misperception are open to modification

by the user (see the `Simulation.misperceptions` attribute, section 2.2.13).

Misperceptions have six attributes: `name`, which identifies the misperception, `target`, which describes the segment susceptible to misperception, `salience`, which is a number representing units of change, `env` which describes when the change happens, and `p`, which represents the probability of a change happening. These are described in the subsections below and section 2.10.7 gives the pseudo-code for how misperceptions are handled in PyILM.

2.10.1 `name`

The `name` attribute is a string used to for referring to the Misperception. It has no role in the outcome of a simulation. In fact, its only use is for printing the report at the end of a simulation. PyILM lists misperceptions that applied during the simulation so that users can more easily understand why certain sound changes happened. Keeping this in mind, `name` should be something descriptive, such as "pre-nasal vowel nasalization" or "final obstruent devoicing".

2.10.2 `target`

The `target` attribute is one or more phonological features representing the class of sounds affected by the misperceptions. In the case of final devoicing, this attribute would probably be set to "+voice, -son, -voc".

2.10.3 `feature`

This attribute is the name of the feature that changes if the misperception occurs. In the case of final devoicing, the value of this attribute would be "voice". The `feature` attribute is often, but not necessarily, one of the features listed in the `target` attribute. Only one name is allowed for this attribute.

2.10.4 `salience`

The `salience` attribute represents the magnitude of a change caused by misperception. The attribute can be any real number in $[-1,1]$. If a misperception actually happens, then its `salience` is added directly to the `value` of the affected Token (see section 2.8). However, Token values must remain in the range $[0,1]$. If the salience would drive a Token's value beyond those bounds, the value is rounded back to 0 or to 1.

2.10.5 `env`

The `env` attribute is a string representing the environment in which a misperception takes place. There are three possible formats for this string: "X_", "_X", "X_Y", where X and Y are strings consisting of the names of one or more features separated by commas, and the underscore represents the position of the sound that might be misperceived. For instance, the following are acceptable values:

+voice_
-nasal_
_+voice,-son
+voc_+voc

2.10.6 p

The p attribute is a number in (0,1) that represents the probability of a misperception occurring.

2.10.7 How misperception happens

Misperception applies to the output of the production algorithm (see section 3.4, see also figure 1 on page 7). The output of the misperception function is sent as input to the learning algorithm. This means there are no further changes that can apply to any sounds in a word, once the word is received by the learning algorithm.

The misperception function loops through the utterance, and checks to see if any of the segments are in a position where they might be affected by misperception. This done by comparing the environment of that segment to the env attribute of the Misperception. If they match, then PyILM "rolls the dice", so to speak, and there is some probability, based on the Misperception's p attribute, that change happens. The pseudo code for this is given below.

Algorithm 2 Misperception function

```
1  for sound in utterance:
2      e = Simulation.get_environment(sound,utterance)
3      for mis in Simulation.misperceptions:
4          if Simulation.check_for_misperception(mis,e):
5          #if a misperception applies here
6              if set(mis.filter).issubset(set(sound.
                   features)):
7              #and if it applies to this sound
8                  if random.random() <= mis.p:
9                  #and if it applies on this occasion
10                     sound.features[mis.target]+=mis.
                           salience
11                     #change the feature vaules of the
                           sound
```

2.10.8 A note on misperception definitions

The way that misperceptions are defined can affect the outcome of a simulation. Misperceptions target phonological features. What this means is that when a misperception has had its full effect, and a sound has switched categories, then

the misperception will stop applying. For example, suppose that a simulation has a word-final devoicing misperception that targets sounds marked [+voice, −son, −cont], and suppose further that /b/ appears word-finally in the initial generation. Eventually a word like /ab/ will become /ap/. At this point, the devoicing misperception no longer applies to the final consonant, because it has become [−voice] and the misperception targets only [+voice] sounds.

This is a design choice for PyILM, and it is not a claim about the way that sound change operates. It is certainly not the case that the phonetic effects underlying sound change suddenly stop occurring just because of the way that some people have organized their mental grammar. The idea in PyILM is that after a sound has changed categories (e.g. from [+voice] to [−voice]), then it is irrelevant if any further phonetic effects occur. If a speaking agent has recategorized /b/ as /p/ then it does not matter if final devoicing applies to /p/ any more, since it is already voiceless.

If, on the other hand, the final devoicing misperception had been designed to target only [−son, −cont] sounds, without reference to [voice], then even after the switch from /b/ to /p/ the misperception will continue to apply. This leads to a "polarization" effect, where the phonetic values for a sound influenced by misperception will continue to get pushed to the extreme ends of a feature dimension. For example, tokens of a sound affected by the final devoicing misperception will eventually all have a [voice] value of 0. (This may also cause a further recategorization if the `allow_unmark` option is enabled; see section 2.2.17.) A misperception that raises a feature value will likewise eventually push all token of a category to have a phonetic value of 1.

For this dissertation, all misperceptions were designed in such a way as to avoid the polarization effect.

3 Algorithms

This section details three kinds of algorithms used by agents in the simulation: learning, production, and invention.

Information about an agent's phonological system is represented using an exemplar model (Johnson 2007, Pierrehumbert 2001). These are models of memory where learners keep "copies" of every experienced speech event. These copies are known as exemplars. Exemplars are stored in a multidimensional space, and can in principle be stored at any level of detail. In PyILM, this space is a FeatureSpace object (see section 2.6), and the exemplars are stored at the level of phonetic features as Token objects (see section 2.8). The number of dimension in this space is equal to the number of features listed in `Simulation.features_file`.

Both the learning and production algorithms are influenced by the exemplar model. The learning algorithm works by comparing input values to the exemplars in memory. The production algorithm generates phonetic values from a distribution that is created based on the exemplar space.

This section on Algorithms is organized from the perspective of a newly

created agent in the simulation. The first thing an agent does is learn, followed by an update of their lexicon and inventory, and finally an agent reaches the production phase.

3.1 Learning algorithm

There are two phases to the learning algorithm: parsing and updating. In the parsing phase, the learner assigns a category to each incoming sound. The results of categorization are used in the second phase to update the lexicon and inventory. After these steps have been run for each input word the simulation runs a phonological feature clustering algorithm.

3.1.1 Parsing a Word

The goal of this phase of learning is to assign each Sound of the incoming word to a Segment category. This is done by comparing the phonetic similarity of the input with all the previous inputs that are stored in memory. If the input is sufficiently similar to any segment in the learner's inventory, then it is assigned to that category. Otherwise, a new category is created. The overall learning process for a word is described in Algorithm 3.

Learning starts with the input of a Word object (see section 2.3) consisting of Sounds (see section 2.7). Sounds have a `features` attribute, although this actually consists of Token objects (see section 2.8), not Feature objects. Tokens have phonetic values, which are real numbers in [0,1]. For each phonetic dimension, the new token is first stored into the exemplar space. Then it is compared to every other token in the space, and an activation value is returned for each such comparison.

The activation function referenced on line 5 is based on Pierrehumbert (2001). It is described in Algorithm 4. Each of the existing categories is assigned an "activation" value, with higher activation values representing greater phonetic similarity. Activation of an exemplar is measured as e raised to the power of the negative difference between the input token and the exemplar. Activation of a segment category is the sum of the activation of its exemplars.

Agents have a threshold for similarity, which is controlled by a simulation parameter called `minimum_activation_level` (see section 2.2.14). It is a number in [0,1] that represents the degree to which a segment category must be activated in order for agents to consider an input token to be a member of that category. A value of 0.8, for example, means that in order for a input token to be considered a member of an existing segment category, the averaged activation of all exemplars for that category must be 80% of the maximum possible activation.

The activation function uses this `minimum_activation_level` parameter to calculate the specific minimum activation level for the given feature dimension and segment category. Then it calculates the difference between the actual activation and the minimum by treating these values as points on the curve $y = e^{-x}$ and calculating their distance. If this distance is greater than or equal

Algorithm 3 Learning algorithm

```
def learning(input_word):
    best_matches = list()
    for sound in input_word:
        activation_matrix = dict()
        for token in sound.features:
            activations = calculate_activations(agent.
                inventory,token)

            for seg,value in activations.items():
                activation_matrix[seg].append(value)
                #activation_matrix[seg][j] equals
                #how much seg is activated
                #on the jth feature

        for seg in activation_matrix:
            total = sum(activation_matrix[seg])
            activation_matrix[seg].append(total)

        activation_matrix.sort(key=lambda x:x[-1])
        best_matches.append(activation_matrix[-1])
        #best_matches[j] equals
        #the category with the highest activation
        #for the jth position in the word

    category = None
    new_word = Word()
    for seg,activation in best_matches:
        if activation >= 0:
            category = agent.inventory[seg]
        else:
            category = create_new_category(seg)
        new_word.string.append(category)
        agent.update_feature_space(category)

    agent.update_inventory(new_word)
    agent.update_lexicon(new_word)
```

Algorithm 4 Activation function

```
1  def calculate_activation(input_token):
2
3      actual_activation = sum(math.e**(-1*(input_token -
           exemplar)) for exemplar in feature_dimension)
4
5      min_activation = sum(math.e**-(1-Simulation.
           minimum_activation_level) for exemplar in
           feature_dimension)
6
7      distance = scipy.integrate.quad(lambda x: math.e**-x,
            min_activation, actual_activation)
8
9      return distance
```

to 0 the input token meets the similarity threshold for this segment category (at least on this feature dimension) and might be considered as a potential match. If the distance is a negative number, then the actual activation level is lower than the minimum and the input token is not similar enough to this segment category on this dimension.

These distances are returned to the main algorithm, and they are summed and added to the activation matrix. Then the distances on each phonetic dimension are summed, and if any of these total distances is greater than or equal to 0, then the input token is assigned to the category with the highest value. Otherwise a new segment category is created.

After learning, another algorithm searches for any "spurious" categories that might have been created. A spurious category is one where the interval of exemplar values representing the category are a sub-interval of some other category, along every dimension.

Spurious categories crop up early in the learning phase when the exemplar space is still sparsely populated, and they do not occur in every learning phase. To illustrate this, consider the following hypothetical simulation where the speaker's inventory has two fricatives /s/ and /z/. For simplicity, assume there are only three features: [nasal, continuant, voice].

The speaker produces an example of /s/, which has values [0.01, 0.9, 0.1], i.e. the sound has low nasality, high continuancy, and low voicing. This is the first sound the learner has heard, and it is assigned to the category labeled /s/, which matches the category in the speaker's inventory (although the learner does not know this, of course).

Then the speaker produces an example of /z/, with values [0.02, 0.8, 0.6]. This is nearly identical to /s/ on the nasality and continuancy dimensions, but differs quite a lot on the voicing dimension. Assume that in this case this difference is sufficient for the learner to decide that this sound is not an example

of /s/. Since /s/ is the only category the leaner knows yet, a new category has to be created for this new sound, and it is labeled /z/ (whether the learner actually does make a new category depends on the value of `minimal_activation_level`, see section 2.2.14).

Next, the speaker produces another example of /s/, this time with values [0.01, 0.85, 0.35]. This sound is similar to both /s/ and /z/ on the nasality dimension, and relatively close to both on the continuancy dimension. On the voicing dimension, the new sound is quite distant from both /s/ and /z/. The learning algorithm considers this sound to be close to neither /s/ nor /z/, and assigns to its own category, labeled /z̬/ (again, in a simulation this would depend on `minimal_activation_level)`. This category will become the spurious one. By the end of the learning phase, the range of values that the learner associates with /z̬/ are going to be a indistinguishable from those associated with /s/, since both of these sets of values were drawn from the same underlying distribution, namely the one the speaker associates with /s/.

As learning progresses, the learning agent hears more and more examples of the fricatives, and the exemplar space begins to fill up. By the end of learning, it turns out that there are exemplars of /s/ with nasality values ranging from 0 to 0.03, continuancy values ranging from 0.8 to 1.0, and voicing values ranging from 0.05 to 0.4. If the exemplar(s) associated with the category /z̬/ were to be fed back into the learning algorithm at the end of the learning phase, they would surely be categorized as /s/

To check for spurious categories, an algorithm does a pairwise comparison of every segment in the inventory. For each pair of sounds A and B, it checks if the minimum exemplar value of Sound A is greater than or equal to the minimum value Sound B, and if the maximum value of Sound A is less than or equal to the maximum value of Sound B, across every feature dimension. If both conditions are true, then Sound A is considered to be spurious. In this case, all exemplars labeled A are relabeled B, and A is removed from the inventory.

3.1.2 Creating new segment categories

When an input Sound has phonetic values that are too different from any known category, a new Segment object is created. Its Token values are analyzed, and phonological features are assigned, as described in section 3.3. A symbol is then chosen for the segment based on these phonological feature values.

The symbol is chosen in a fairly simplistic way. The program consults the possible segments in the list provided to `Simulation.feature_file` (see section 2.2.9), and assigns a score to each of them by comparing distinctive feature values. A symbol scores 1 point per feature match. The highest ranked symbols are put in a set and the rest discarded. This remaining set is further filtered to remove any symbols that are already in use in the inventory. A symbol is randomly chosen from the remaining set members. A random selection of this sort is safe, since the symbol has no effect on the outcome of the simulation, and exists purely to increase the readability of the output.

3.2 Updates

3.2.1 The lexicon

Once the input word has now been transformed into a list of Segment objects, the learner can add it into the lexicon. If a word with this meaning has never been encountered before, the agent creates a new entry for this meaning in her lexicon, and adds the input word with a frequency of 1.

If this meaning has been encountered before, the agent checks to see if this particular pronunciation is known, i.e. checks to see if there is a match between the input Word's `string` attribute and the `string` attribute of any Word already in the lexicon. If so, the frequency of that word is increased by 1, if not the input word is added to the list of possible pronunciations with a frequency of 1.

3.2.2 The inventory

In the final phase of learning, the inventory is updated. This may involve one of two things. If the input word contained phonetic values such that a new segment was created, then the inventory needs to have that segment added. Even if the input word matched entirely to known segments, the specific values associated with each of those segments must now be updated.

3.3 Determining phonological feature values

Phonological categories are determined using a k-means algorithm that clusters exemplar values along each feature dimension.

The algorithm begins by create k points in the space representing the objects that are being clustered. These initial points are called "centroids" and they represent a potential center of a cluster. Then every point in the data is added to the cluster with the closest centroid. After all the data has been classified this way, new centroids are chosen by calculating an average point for each of the existing clusters. The data is then reclassified by clustering it based on the new centroids. This process of averaging and reclassifying is repeated until the point where the algorithm chooses the same centroids two loops in a row.

In the case of the simulation, the clustering function takes two arguments: `feature`, which is the name of the feature dimension to cluster, and `k`, which is the number of clusters. By default `k=2`, since phonological features are typically modeled as binary. However, it is still possible to end up with non-binary features, depending on the actual distribution of token values.

On Line 4 the algorithm chooses two centroids that are relatively far apart from each other, which is typical for the initial choice of centroids. Then the main loop is entered on Line 5. The loop exits when the choice of centroids doesn't change across loops. Then from Lines 6-12, the tokens for a given feature dimension are then selected, and for each token, it is compared to the most recently selected set of centroids, called `new_centroids`. On the first loop these are random choices, on further loops they are calculated averages.

Algorithm 5 K-means algorithm

```
1  def kmeans(feature,k=2):
2
3      old_centroids = agent.learned_centroids
4      new_centroids = [random.uniform(.1,.25), random.
           uniform(.75,.9)]
5      while not old_centroids == new_centroids:
6          clusters = dict()
7          tokens = agent.feature_space[feature]
8          for token in tokens:
9              closest = 999
10             for c in new_centroids:
11                 if abs(token.value-c) < abs(token.value-
                       closest):
12                     closest = c
13             clusters[closest].append(token.value)
14             old_centroids = new_centroids
15             new_centroids = list()
16             for k in clusters:
17                 new_centroids.append(sum(clusters[k])/len
                       (clusters[k]))
18     agent.learned_centroids = old_centroids
19     return clusters
```

The clusters dictionary assignment on Line 13 creates a mapping from centroid values to a list of Tokens assigned to that centroids cluster. Then the `new_centroids` values are saved into `old_centroids` and the `new_centroids` is emptied out (Lines 13-15). Finally, on Lines 16 and 17, `new_centroids` is filled with new centroid values calculated as the average of the Tokens in the clusters dictionary.

The loop then returns to the beginning. If the most recent run of Line 17 calculated the same average values as the last time Line 17 was run, then the loop breaks and the program jumps to Line 18 where agent saves the values from `new_centroids` and the function returns the new cluster centroids.

There are two possible phonological features: + and -. After the k-means clustering is done, the cluster with the higher centroid is designated the [+feature] cluster, and the one with the lower centroid is designated the [−feature] cluster. If all tokens fall into a single cluster, then a feature value is chosen based on the values of the tokens. If most of the tokens have a value above .5, then [+feature] is assigned to the entire cluster, otherwise [−feature] is assigned. If the `allow_unmarked` option is set to True (see section 2.2.17), and there is a category where every token value is 0, then [nfeature] is assigned.

3.4 Production algorithm

The production algorithm selects a Word from an agent's lexicon to produce, and transforms each of the Segments in the Word into a Sound. While Segments in an Agent's lexicon are made up of phonological features, Sounds are made up of phonetic Tokens which have real-valued features. There are three steps in production described here.

3.4.1 Initialization

This step occurs at the end of the main simulation loop, just after a learner has been "promoted" to speaker (see 1). The new speaker looks through their inventory, and for each segment it estimates a distribution of phonetic values along each dimension. Agents assume the distribution is Gaussian. Pseudo code is given below. This code runs once for each segment in an agent's inventory. During testing, it was found that the distributions were better estimated using distance from the median, rather than from the mean. The Gaussian distribution is implemented using the `normalvariate` function of Python's built in `random` module.

3.4.2 Step 1

Production begins with a decision: select a word from the lexicon or invent a new word. The probability with which a new word is invented is given by the simulation's `invention_rate` attribute (see section 2.2.11). If a new word is required, the speaker uses the invention algorithm described in section 3.5 to create one. Otherwise, one is chosen from the lexicon.

Algorithm 6 Distribution estimation

```
1  def estimate(segment):
2      for feature in segment.features:
3          cloud = [token.value for token in agent.
               feature_space[feature] if token.label ==
               segment.symbol ]
4          median = agent.calculate_median(cloud)
5          mad = agent.calculate_median([abs(value-median)
               for value in cloud])
6          agent.distributions[segment][feature] = random.
               normalvariate(median, mad)
```

Rather than choosing a word directly, agents actually first select a meaning, then choose which word to produce for that meaning. Each meaning in the lexicon is associated with a list of Words, each stored alongside a raw count of how many times it appeared in the input. The production algorithm chooses a Word with probability proportional to it is frequency.

3.4.3 Step 2

The word selected by the first step consists of Segments (see section 2.4), but these are not the objects transmitted to the learner. In the second step of the production algorithm, Segments are transformed into different objects known as Sounds (see section 2.7), which represent an instance of a segment being pronounced. Agents pass through each feature of each segment in the word. For each feature, agents sample a value from the appropriate probability distribution for that segment. Pseudo-code for this algorithm is shown in Algorithm 7.

Algorithm 7 Production algorithm

```
1  def produce(lexical_item):
2      utterance = Word(list(), lexical_item.meaning)
3      for segment in lexical_item:
4          sound = Sound()
5          for feature in segment.features:
6              phonetic_value =
7              agent.distributions[segment][feature].sample
                   ()
8              sound.features[feature] = phonetic_feature
9          utterance.append(sound)
10     return utterance
```

The utterance returned at the end of the algorithm represents what the

speaker intends to produce for the listener. It is not necessarily what the listener hears. This utterance is subsequently sent through a misperception algorithm (see section 2.10) which may change the utterance in some way.

3.5 Invention algorithm

The invention algorithm serves two purposes. It is used to generate a lexicon for the initial generation of the simulation, and it is used by agents at later generations if they choose to create a new word. The words constructed by this algorithm always conform to the existing phonotactics of the language. The following pseudo-code outlines the algorithm.

Algorithm 8 Invention algorithm

```
1  def invent(agent, phonotactics):
2      word = Word()
3      syl_length=random.randint(Simulation.min_word_length,
4                                Simulation.max_word_length)
5
6      for j in range(syl_length):
7          syl_type = random.choice(Simulation.phonotactics)
8          for x in syl_type:
9              if x == 'V':
10                 seg=random.choice(agent.inventory.vowels)
11             elif x == 'C':
12                 seg = random.choice(agent.inventory.cons)
13             word.string.append(seg)
14     return word
```

Line 4 creates a new "empty" word object (see section 2.3). Line 5 randomly determines the length of the word in syllables (see section 2.2.6 and 2.2.7).

The loop that begins on line 8 will run once for each syllable in the word. Line 10 selects some possible syllable for a given phonotactics. For example, if (C)V(C) was supplied to the algorithm, then it selects randomly from the set {CVC, CV, VC, V}.

The loop that begins on line 9 runs through each segment in the syllable type chosen. For each C or V "slot", PyILM randomly selects a segment of the appropriate type. Once the entire word has been constructed, it is assigned a new meaning and then stored in the lexicon.

The invention algorithm does not check to see if there is an existing word with the same segmental material as the new one. In other words, it is possible for homophones to appear in the language. However, this has no effect on production or learning of these words, so it is basically irrelevant to the outcome of the simulation.

4 Using PyILM

4.1 Obtaining PyILM

The source code for PyILM will be available for download from https://www.github.com/jsmackie/PyILM. Running PyILM requires Python 3.4. There are also some 3rd party libraries needed: Numpy and SciPy are necessary to run the basic PyILM code, and the Visualizer requires Matplotlib and PIL (Python Image Library). All of these can be obtained from the Python Package Index at https://pypi.python.org.

4.2 Configuration files

PyILM simulations require a configuration file. These files should be saved into a folder called "config", which must be a subfolder of the main PyILM directory. A configuration file is a text file which must conform to a particular structure, described below, and its file extension must be `.ini`. Configuration files are broken up into sections, each indicated by a header in square brackets. Each line in a section may contain a parameter name follow by an equals sign followed by a value. (This follows the standard INI file format used on Windows.) An example is given below, with some discussion following.

```
[simulation]
initial_lexicon_size=30
generations=30
phonotactics=CCVCC
invention_rate=0.05 #this might be too low?
minimum_repetitions=2
min_word_length=1
max_word_length=3
[misperceptions]
#"misperceptions"
stop lenition=-voc,-cont,-son;cont;.5;+voc_+voc;.25
nasalization=-cont,-son,-nasal,-voc;nasal;.5;_+nasal,+son,-voc;.25
initial fortition=-voc,+cont,-nasal;cont;-.5;#_;.25
stop aspiration=-voc,-son,-voice,-cont;hisubglpr;.5;_+voc,+high;.25
obstruent glottalization=-voc,-son,-cont,-voice;mvglotcl;.5;_-voc,+glotcl,-
mvglotcl;.25
#"biases"
ejectives are marked=-voc,-son,-cont,+glot_cl,+mvglotcl;mvglotcl;-.1;*;.5
retroflex is marked=-ant,-distr,-cont,+cor,-son,-voc;ant;.1;*;.5
[inventory]
start=p,t,k,b,d,g,m,n,f,s,z,a,i,u
#start=10,3
[lexicon]
words=kapa,mufu,tiki,matk,bziafm
```

There are four section headers recognized by PyILM: [simulation], [misperceptions], [inventory], and [lexicon]. The order of the parameters in a section is not important. The [simulation] section is mandatory. The parameter names which can be used are listed back in Section 2.2. Any parameter that is not mentioned in the configuration file will given a default value. These defaults are likewise described in Section 2.2.

The [misperception] section is also mandatory. Each line in this section can include the name of a misperception as a parameter (any name is allowed, and spaces are permitted), and the remainder of the misperceptions details follow the equal sign. See section 2.10 for more information on how to structure a misperception.

The [inventory] section is optional, and only allows the single parameter name `start`, which takes the same possible values as the [simulation] parameter `initial_inventory` (see section 2.2.4). The [inventory] section exists to make it conceptually easier to manage the inventory separately from the other simulation parameters, and because future versions of PyILM are anticipated to have more possible parameters in this section.

The [lexicon] section is also optional, and can take the single parameter `words`, which is the same thing as using the `initial_words` parameter in the [simulation] section (see section 2.2.16).

The [simulation] and [misperception] sections should come first in a configuration file. The [inventory] and [lexicon] sections, if present, should come at the end.

If a line in the file begins with either the symbol "#" or ";" then PyILM will ignore the entire line. This can allow users to include comments to themselves about parameters. It also provides a convenient way of flipping parameter values between simulations without keeping multiple copies of a configuration files with minor differences. If these symbols are encountered in the middle of a line, they are treated normally, which is what allows misperceptions to make use of both symbols without any problems.

4.3 Running a simulation

There are two ways to run simulations. From a command line, navigate to the PyILM directory and then type

```
python pyilm.py filename
```

where `filename` is the name of your configuration file. If no filename is provided, then all the defaults are used.

To run a simulation from within another python script, use the following code, replacing the string "config.txt" with the name of the appropriate configuration file.

```
import pyilm
sim = pyilm.Simulation('config.txt')
sim.main()
```

PyILM comes bundled with another program, pyilm_batch.py, for running multiple simulations. To run a batch of N simulations, type the following in a command line

```
python pyilm_batch.py filename N
```

where `filename` is again the name of a configuration file. Supply the string `None` for the filename to use all defaults. For example, to run a batch of 25 simulations from within a Python script, use the following code:

```
import pyilm_batch
batch = pyilm_batch.Batch('config.txt', 25)
batch.run()
```

When running in batch mode, the user-supplied value for the random seed is ignored, and a different random seed is generated for each simulation, while keeping all other configuration details the same.

4.4 Viewing results

After running the first simulation, PyILM will create a new a folder called "Simulation Results", which will be placed in the same folder as `pyilm.py`. Each simulation is given its own subfolder inside of the Simulation Results folder. These subfolders are named "Simulation output (X)", where X is a number automatically assigned by PyILM.

This output folder contains a copy of the configuration file, as well as files detailing the state of the simulation at the end of each generation. Information about the exemplar space is written to files with the name `feature_distributionsX.txt` where X is the generation number. Information about the inventory and lexicon is written to files with the name `temp_outputX.txt`, with X again standing in for a generation number. It should be noted that PyILM starts counting at 0, not 1. Generation 0 is the initial generation seeded with information from the configuration file. Generation 1 is the first generation to learn from the output of another agent. A side-effect of this is that the first simulation you run will be in the folder "Simulation output (0)".

The output files can be opened and inspected, but they are not formatted to be human-readable. They are intended for use with the PyILM Visualizer, which is an independent program that displays the information in a graphical interface. As such, it is not recommended that you change any of the names of the files, or alter any of the contents, because this can cause unusual behaviour in the Visualizer.

The Visualizer can be opened by double-clicking the file `visualizer.py` which comes with PyILM. It will be located in the same folder as the main PyILM program. When the program launches, select the Data menu, then input the simulation and generation number that you wish to see. Blank lines are interpreted as the number '0'. From there, it is possible to navigate between simulations and generations using the "Forward" and "Backward" buttons on the top right, or by returning to the Data menu.

Each generation shows the segment inventory as table of buttons. Clicking on a button brings up more details about that segment, including its distribution in the lexicon, phonetic and phonological properties. More information about the simulation can be viewed under the Synchrony and Diachrony menus. Synchrony options include anything specific to the generation currently displayed, such as the lexicon. Diachrony options include the ability to plot changes over time. Misperceptions, which do not change, are listed under Synchrony.

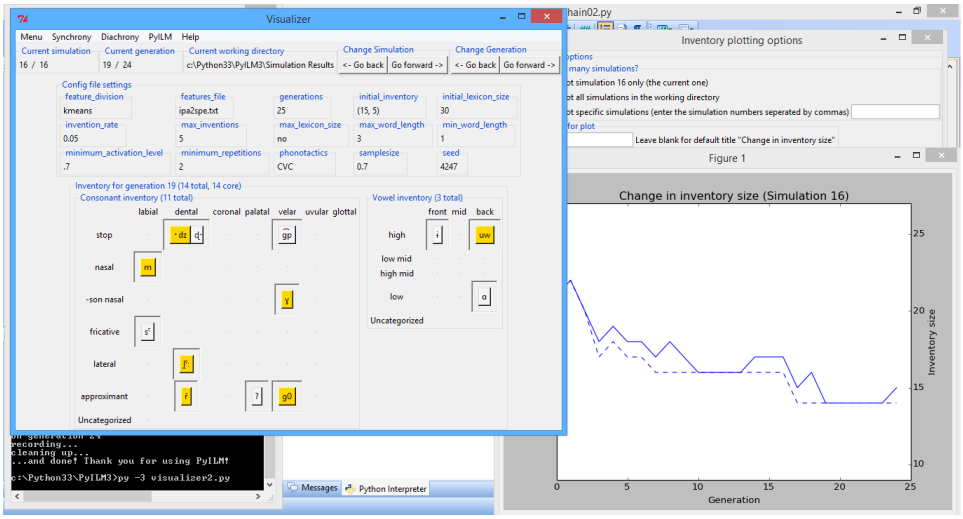

Figure 4: Screen shot of PyILM Visualizer

5 Other notes

5.1 Limitations

PyILM cannot do everything. The program is designed largely to explore the long-term consequences of misperception-based sound change for segment inventories. There are a several other ways in which sound systems can change over time that are not modeled.

5.1.1 No social contact

One of the limitations of PyILM is that there is only ever a single speaker and a single listener, so sound changes that rely on contact between speakers of different languages is not possible.

Human cultures speaking different language often live nearby and interact with each other. This often leads to languages borrowing words or morphemes

from the other language. Occasionally, entire paradigms are borrowed. This can lead to changes in a sound system if the borrowed items contain phonemes that are not part of the borrowing language. For example, click consonants have entered into some Bantu language through borrowing (Güldemann and Stoneking 2008). There is no guarantee of this occurring, of course, so it is also quite common for languages to change the sounds of loanwords so that they fit native patterns (Peperkamp 2004).

The focus of the dissertation is on how phonetic effects influence the evolution of sound inventories, so no borrowing is simulated. It would be possible, however, to implement a simple form of borrowing in PyILM with a few additions. At arbitrary points in a simulation, generate new words that contain one or more sounds not figuring already in the simulation, and add them to the speaking agent's lexicon. Loanword adaptation can be simulated by running these words through a speaking agent's learning algorithm to see which categories any novel sounds might be assigned to.

5.1.2 No deletion or epenthesis

Changes are also limited to those that affect feature values. Deletion and epenthesis do not occur. The main reason for excluding these changes is because they can change the phonotactics of a language, and phonotactics will play a relevant role in the simulations reported later in this dissertation.

In fact, deletion is technically possible in PyILM, but simply not implemented for any simulations that I report for the dissertation. Epenthesis is considerably harder to implement, and is currently not possible.

Suppose that we want to implement an epenthesis rule that inserts a vowel between two non-continuants. The effect of the epenthesis rule should be that a phonetic vowel appears; there is no underlying vowel in the lexical item that corresponds to the epenthesized vowel. Suppose it is a mid-central schwa-like vowel. Because it is a phonetic epenthesis, we cannot simply use a schwa symbol - it must be represented by a column of numbers. How do we generate these numbers?

There are three options for this. One is to generate numbers for a mid-central vowel based on the speaker's exemplar space. This is easy if the speaker happens to have such a vowel already in their inventory. If there is no such vowel, then it is difficult to come up with a general solution for which other vowel would be the "closest", since any arbitrary vowel system is possible in PyILM. In any case, whatever vowel is chosen will not be a mid-central vowel, so it will not correspond to the description of the epenthesis rule. This makes the behaviour of the simulation unpredictable from a user's perspective, and is not a good design choice.

The second option of generating numbers using the listener's exemplar space has the same problems. It is further complicated by the problem that their exemplar space continues to change throughout the learning phase so the type of epenthesized vowel would, again, vary unpredictably.

The third option is to include in a PyILM a generic vowel "generator" that

can be used to epenthesize a vowel of a predictable quality in every case. This option feels extremely artificial compared to the first two, where at least there was some semblance of changes being related to either articulation or perception. On the other hand, it does make it easier to follow the changes that occur over the course of the simulation.

5.1.3 No morphology or syntax

Words always convey a single meaning, and agents never utter more than one word at a time, so there is effectively no morphology or syntax in the simulation. Since some sound change might emerge from interactions at word or morpheme boundaries, this limitation does prevent modeling certain kinds of change. However, the changes that occur at a morpheme boundary are essentially of the same type as change that might occur within a morpheme. The root cause of the change is still a phonetic interaction of two adjacent sounds.

5.1.4 No long distance changes

Misperceptions that occur in PyILM can only target adjacent sounds. It is not possible to simulate the emergence of any types of harmony patterns, for example. Although consonant harmony is rare, it does exist, and plausible historical routes for its development have been proposed (e.g. Hansson (2007)). However, the types of consonants that emerge from long-distance changes are a subset of those that might emerge from local changes. Since the goal of this dissertation is to understand how inventories change over time, there is no particular gain to be made by including long-distance changes.

5.2 Running time

The running time of a simulation is determined by a number of different factors.

The most important are the `lexicon_size` and `min_repetition` parameters. Together, they determine the total number of words that a speaking agent will produce in a given generation. If there is no maximum lexicon size, and `invention_rate` > 0, then running time can increase for each generation if new words are added to the lexicon.

Another factor is word length, since PyILM has to check every segment in each word for possible misperceptions, and the learner has to analyze each segment. The parameters controlling word length are, of course, `min_word_length` and `max_word_length`. Phonotactics also plays a role here too, since the average word length in a CV language is going to be shorter than the average word in a CCVCC language, other things being equal.

The number of misperceptions seems to have no significant effect on total running time. Checking if a misperception applies is trivial and, in most cases, nothing happens. The number of contexts where misperceptions apply is much smaller than the total number of contexts in the entire lexicon. When a misperception does apply, the operation is, again, trivial since changing phonetic

values consists of adding two numbers together, followed by a check to ensure no phonetic value goes below 0 or above 1.

A single generation of a CV language with 30 initial words and a maximum lexicon size of 30 takes less than a second. Setting the phonotactics to CCVCC increases the time significantly, and a single generation may take 10 seconds. The recording phase, where PyILM generates an output file for use with the visualizer, also contributes to running time. The length of time it takes for a generation to be recorded depends on the number of changes that occurred in the simulation.

Another factor is the time taken in labeling segments for human readability. During the simulation, segments are simply numbered, rather than being assigned IPA symbols. This is because there is no way to know which symbol will be appropriate until the end of the learning phase, when the phonological feature values are assigned. Searching the list of all possible symbols, and comparing feature values to see which would be best, can be time consuming. PyILM looks for a short-cut by comparing against the previous generation, and where sounds have not changed feature values it simply re-uses the old symbol.

References

Blevins, J.: 2004, *Evolutionary Phonology. The Emergence of Sound Patterns*, Cambridge University Press.

Blevins, J.: 2006, A theoretical synopsis of Evolutionary Phonology, *Theoretical Linguistics* **32**(2), 117–166.

Chen, M. Y.: 1997, Acoustic correlates of English and French nasalized vowels, *The Journal of the Acoustical Society of America* **102**(4), 2360–2370.

Güldemann, T. and Stoneking, M.: 2008, A historical appraisal of clicks: a linguistic and genetic population perspective, *Annual Review of Anthropology* **37**, 93–109.

Hansson, G. Ó.: 2007, On the evolution of consonant harmony: The case of secondary articulation agreement, *Phonology* **24**(01), 77–120.

Johnson, K.: 2007, Decisions and mechanisms in exemplar-based phonology, *in* M.-J. Sole, P. S. Beddor and M. Ohala (eds), *Experimental approaches to phonology*, Oxford University Press, Oxford, pp. 25–40.

Kirby, S.: 2001, Spontaneous evolution of linguistic structure - an iterated learning model of the emergence of regularity and irregularity, *Evolutionary Computation, IEEE Transactions on* **5**(2), 102–110.

Lisker, L.: 1986, "Voicing" in English: A catalogue of acoustic features signaling /b/ versus /p/ in trochees, *Language and speech* **29**(1), 3–11.

Mielke, J.: 2008, *The emergence of distinctive features*, Oxford University Press.

Ohala, J.: 1983, The origin of sound patterns in vocal tract constraints, *in* P. F. MacNeilage (ed.), *The production of speech*, Springer-Verlag, New York, pp. 189–216.

Peperkamp, S.: 2004, A psycholinguistic theory of loanword adaptations, *in* M. Ettlinger, N. Fleisher and M. Park-Doob (eds), *Annual Meeting of the Berkeley Linguistics Society*, Vol. 30, Berkeley Linguistic Society, Berkeley, CA, pp. 341–352.

Pierrehumbert, J.: 2001, Exemplar dynamics, word frequency, lenition, and contrast, *in* J. Bybee and P. Hopper (eds), *Frequency effects and the emergence of linguistic structure*, John Benjamins Publishing Company, Amsterdam, pp. 137–157.

Raphael, L. J.: 2005, Acoustic cues to the perception of segmental phonemes, *in* D. Pisoni and R. Remez (eds), *The handbook of speech perception*, Blackwell Publishers, Oxford, pp. 182–206.

Smith, K., Kirby, S. and Brighton, H.: 2003, Iterated learning: A framework for the emergence of language, *Artificial Life* **9**(4), 371–386.

Yang, C.: 2010, Who's afraid of George Kingsley Zipf, *Unpublished manuscript* .