MorphAGram Framework for Morphological Segmentation

Preprocessing Phase

MorphAGram uses the Pitman-Yor Adaptor-Grammar Sampler (PYAGS), developed by Mark Johnson, for training. The sampler can be downloaded at:
http://web.science.mq.edu.au/~mjohnson/code/py-cfg-2013-09-23.tgz

For complete information about how the sampler works, please see
https://cocosci.princeton.edu/tom/papers/adaptornips.pdf

The sampler requires two types of inputs: a grammar and a list of training units. For the purpose of unsupervised morphological segmentation, a grammar should specify word structure, while the list of training units is a list of raw words. In addition, MorphAGram requires the text to be in the Hex format in order to meet the language-independence assumption and to escape special characters. Also the grammar should have the characters in the word list as terminals.

The first step is to provide an initial CFG (Context-Free grammar) and a list of words (one word per line) to MorpAGram, which in turn converts them into inputs to PYAGS. The initial CFG should have two parameters associated with each production rule (default values are zeros). The first number represents the value of the probability of the rule in the generator, and the second number is the value of the α parameter in the Pitman-Yor process. Below is an example CFG.

```
1 1 Word --> Prefix Stem Suffix

Prefix --> ^^^
Prefix --> ^^^ PrefixMorphs
1 1 PrefixMorphs --> PrefixMorph PrefixMorphs
1 1 PrefixMorphs --> PrefixMorph
PrefixMorph --> SubMorphs

Stem --> SubMorphs

Suffix --> $$$
Suffix --> SuffixMorphs $$$
1 1 SuffixMorphs --> SuffixMorph SuffixMorphs
1 1 SuffixMorphs --> SuffixMorph
SuffixMorph --> SubMorphs

1 1 SubMorphs --> SubMorph SubMorphs
1 1 SubMorphs --> SubMorph
SubMorph --> Chars
1 1 Chars --> Char
1 1 Chars --> Char Chars
```

MorphAGram has three learning settings; Standard, Scholar-Seeded and Cascaded. For complete details, please see <Link to our paper once published>.
Here is how to preprocess the data for each setup:

Standard Setting

#Read the initial word list and convert it into HEX
words, encoded_words, hex_chars = process_words(word_list_path)
write_encoded_words(encoded_words, word_list_output_path)
#Read the initial CFG and append the characters as terminals
grammar = read_grammar(grammar_path)
appended_grammar = add_chars_to_grammar(grammar, hex_chars)
write_grammar(appended_grammar, grammar_output_path)
 # <word_list_output_path> and <grammar_output_path> then become the inputs to PYAGS.

Scholar-Seeded Setting

#Read the initial word list and convert it into HEX
words, encoded_words, hex_chars = process_words(word_list_path)
write_encoded_words(encoded_words, word_list_output_path)
#Read the initial CFG
grammar = read_grammar(grammar_path)
#Seed scholar-seeded affixes into the grammar, where the affixes are read from a <scholar_seeded_path> file that should have the following format:

```
###PREFIXES###
anti
dis
extra
im
in

###SUFFIXES###
'
's
ing
ed
```

#<prefix_nonterminal> and <suffix_nonterminal> are two strings that specify which nonterminals the prefixes and suffixes should be seeded into, respectively.
ss_grammar = prepare_scholar_seeded_grammar(scholar_seeded_path, prefix_nonterminal, suffix_nonterminal)
write_grammar(ss_grammar, ss_grammar_path)
#Append the characters as terminals
appended_ss_grammar = add_chars_to_grammar(ss_grammar, hex_chars)
write_grammar(appended_ss_grammar, ss_grammar_output_path)
 # <word_list_output_path> and <ss_grammar_output_path> then become the inputs to PYAGS.

Cascaded Setting

#Read the initial word list and convert it into HEX
words, encoded_words, hex_chars = process_words(word_list_path)

```
write_encoded_words(encoded_words, word_list_output_path)
#Read the initial CFG and append the characters as terminals
grammar = read_grammar(grammar_path)
#Build the cascaded grammar by seeding affixes whose count is <number_of_affixes>,  from the segmentation output of a first round of learning that is written into <segmentation_output_path>.
#<nonterminal_regex> is a regular expression that specifies which affixes to copy from the segmentation output, e.g., '(Prefix|Suffix)'
#<prefix_nonterminal> and <suffix_nonterminal> are two strings that specify which nonterminals the prefixes and suffixes should be seeded into, respectively.
cascaded_grammar = prepare_cascaded_grammar(grammar, segmentation_output_path,
number_of_affixes, nonterminal_regex, prefix_nonterminal, suffix_nonterminal)
write_grammar(cascaded_grammar, cascaded_grammar_path)
#Append the characters as terminals
appended_cascaded_grammar = add_chars_to_grammar(cascaded_grammar, hex_chars)
write_grammar(appended_cascaded_grammar, cascaded_grammar_output_path)
# <word_list_output_path> and <cascaded_grammar_output_path> then become the inputs to PYAGS.
```

Note: The appended characters and seeded affixes are non-adapted by default, where the parameters of their corresponding production rules are set to 1.

Training Phase

Download and run the Pitman-Yor Adaptor-Grammar Sampler (PYAGS), developed by Mark Johnson, where the input to the sampler is the output of the preprocessing phase.
http://web.science.mq.edu.au/~mjohnson/code/py-cfg-2013-09-23.tgz

For complete information about how the sampler works, please see
https://cocosci.princeton.edu/tom/papers/adaptornips.pdf

Segmentation Phase

Note: When segmenting a file, the segmentation of seen words is read from the output map generated by parsing the PYAGS segmentation output. For unseen words, our code applies MLE-based heuristics that approximate the parse of the PYAGS grammar. We strongly recommend parsing the grammar using the CKY parsing algorithm instead of using our code for segmentation.

In order to do segmentation upon running the Pitman-Yor Adaptor-Grammar Sampler (PYAGS), MorphAGram first parses the segmentation output of the sampler and converts it into words and their respective segmentations, which are then used to segment any given input text.

Parsing PYAGS Segmentation Output

#Parse the sampler's segmentation output and produce two output files: segmented words and a map of segmented words (dictionary).
<*min_stem_length*> is an integer that specifies the minimum length of the stem.
<*word_output_path*> and <*dic_output_path*> are the file paths of the output segmented words and the output dictionary of words mapped to their segmentations, respectively.
The segmented words are separated by "+" and when a "Stem" nonterminal is part of <*nonterminals_to_parse*>, the stem gets encased in parentheses, e.g. "re+(place)+d".
map = extract_all_words(pyags_output_path, min_stem_length, nonterminals_to_parse, word_output_path, dic_output_path)

Segmenting a White-Space Tokenized Plain Text File

Use the map output from *extract_all_words* to segment a white-space tokenized plain text file.
<*map*> is the output from *extract_all_words* .
<*input_path*> is the path of the input plain text file (assumed to be white-space tokenized).
<*output_path*> is the path where the segmented plain text file is written.
<*multiway_segmentation*> is an optional boolean that forces composite affixes to be further split into simple ones when set to True.
Segment_file(map, input_path, output_path, multiway_segmentation)