Language and Computation - Final project

Andrea Ceolin, Domonique Roberts-Mack

October 6, 2018

The project: Morpheme Induction
In this project, we built an algorithm to identify English affixes based on the ideas of Harris (1955) about the entropy of phoneme sequences as a predictor of word and morphological boundaries. The algorithm has been tested on a corpus of ≈100K English words and has been asked to retrieve the top-8 frequent prefixes, the top-8 frequent suffixes and the top-100 frequent roots.

1 Introduction

1.1 Harris's algorithm

Harris (1955) proposes a method to identify morpheme boundaries based on the predictability of the phonemic string *n+1* given *n*. We can reformulate this concept in terms of entropy. If given a string *n*, there's a number of possible strings *n+1* which can occur with the same probability, all these strings share the same entropy and therefore are not predictable. If instead given a string *n* we can identify a string *n+1* which occurs more frequently than other possible *n+1* strings, than we have a string associated with a lower amount of entropy with respect to others. Harris predicts that in the first case *n+1* is part of a root, while in the second case *n+1* is part of an affix.
For instance, consider the string *ro*. In English there are several words that can be constructed just adding a random letter of the English alphabet (e.g. *road*, *robot*, *rock*, *rodent*, *roe*, etc.) and even though there are some sequences which are not attested (e.g. our database doesn't include any word starting with *rof* or *roh*), the attested sequences seem to be fairly equiprobable. Therefore, we conclude that they must be part of a root. On the contrary, if we consider the string *inte* we would notice that there is a

sequence *n+1* = *inter* which occur with a probability which is strikingly higher than those associated with other *n+1* strings. This means that we are close to a morpheme boundary.

1.2 The method and the wordlist

To test this prediction, we have built an algorithm that calculates the frequency of all the possible n strings representing prefixes and suffixes through a pairwise comparison and returns the strings with higher frequency. The database used is a list of ≈100K English words containing inflected and uninflected forms, available on the website of SIL International [1]. The code has been built on Python 2.7.10. Here's a brief look at the function that check for prexifes, *get_prefixes*:

```
1 for i in range(len(sample)):
2        word1 = sample[i]
3        for j in range(i + 1, len(sample)):
4            #check prefix
5            word2 = sample[j]
6            if word1[0] == word2[0]:
7                pre = word1[0]
8                z = 1
9                while word1[z] == word2[z]:
10                    pre = pre + word1[z]
11                    z += 1
12                    #break the loop if you finish the word
13                    if z == len(word1) or z == len(word2):
14                        break
15                    #add prefix (> 1 letter) to the list
16               if pre not in prefix_dict and len(pre) > 1:
17                    prefix_dict[pre] = 1
18               elif pre in prefix_dict and len(pre) > 1:
19                    prefix_dict[pre] += 1
20           else:
21               break
```

In line 2 a word is extracted from the sample, while in line 5 a second word is extracted. In line 6 the first letter of the word is checked, and if it happens

[1] http://www-01.sil.org/linguistics/wordlists/english/

to be the same, the algorithm checks for a prefix. If also the second letter is the same, the algorithm in 9 starts a loop in which it memorizes the sequence until it reaches the word boundary (12-14) or it finds a different letter. At that point the prefix is stored into a dictionary (16-19). We split the discussion of the method and the results in two parts: the first concerns the identification of the affixes, the second the identification of the roots.

2 First part: Prefixes and Suffixes

2.1 Identification of prefixes and suffixes: caveats

In order to correctly identify prefixes and suffixes in English there are a couple of problems to address. We discuss them in the following paragraphs.

2.1.1 English *-s* suffix. The most common affix in English is the noun plural and verb third person singular *-s* suffix. This suffix is made of a single letter: it means that for $n = s$ it is impossible to test any string of length *n+1*. For this reason, this suffix cannot be detected by our algorithm. Another problem is that the *-s* suffix is so frequent that if we put together the frequencies of all the possible combinations of *n+1* strings, it's likely to obtain a lot of strings made by random letters plus the final *-s*. For example, even though the final sequence *-ed* has a relative frequency which is much higher than any frequency of other 2-letters-final ending in *-d*, while the string *-ts* has a frequency comparable to other 2-letters-final ending in *-s*, the absolute frequency of *-ed* and *-ts* is comparable just because there are much more words ending in *s* than in *d*. To solve this problem, we decided to penalize every suffix with *s* as its final letter.

2.1.2 Affix length. An implicit assumption of 2.1.1 is that we may have longer affixes that compete in absolute frequency with combination of random letters plus shorter affixes. This is a common problem in measuring frequencies of strings of different length: it is clear that the probability of picking at random a string of $n = 2$ letters is different that the probability of picking at random a string of $n = 5$ letters. For this reason, we decided to reward the longer affixes.

2.2 Results

We ran our algorithm on the whole database creating two dictionaries containing the potential affixes. Each affix is a key to a dictionary entry that contains the number of word pairs in which it appears as its value, which can be used as a score. This score is used to rank the affixes.
Then, each score is penalized or rewarded given the conditions in 2.1. In a first experiment, the penalty is *0.5, while the reward is *(n-1)^2 with n being the length of the affix. This means that every letter after the second increases the score of the affix in an exponential way: 3-letter affix become score*4, 4-letter affix become score*9, 5-letter affix score*16 and so on.

2.2.1 Prexifes. These are the top 20 results for the prefixes:

1. re	9240378	11. pr	1636643
2. co	6481020	12. pro	1596412
3. in	3565005	13. inter	1529040
4. un	3457940	14. di	1291240
5. con	3269456	15. ma	1285991
6. de	2993559	16. st	1202182
7. over	2501892	17. su	921682
8. dis	2396408	18. pa	809793
9. ca	1750314	19. ch	758194
10. pre	1735760	20. non	744008

The first mistake we see is 9., where *ca* is identified as a prefix. Another mistake is 11., where *pr* is identified as a prefix, but we have an explanation for this: in English we have two different prefixes, *pre* and *pro*, identified in 10. and 12., starting with the same onset. This causes a big amount of pairs starting with *pr* but having different vowels following, either *o* or *e*. Since our algorithm identifies morpheme boundaries when there is a shift in the entropy of the string, *pr* is identified as a prefix. Other mistakes are 15., 16., 18. and 19.

2.2.2 Suffixes. These are the top 20 results for the suffixes:

1. ing	140672360	11. able	6901740
2. ed	41898465	12. tion	6082470
3. es	23627262	13. ies	4967318
4. ers	17832630	14. ating	4113056
5. er	17776912	15. ling	3733929
6. ting	12986352	16. ally	3728988
7. ation	12820272	17. ically	3655550
8. ly	1124643	18. ations	3622325
9. ness	9224334	19. iest	3176379
10. ted	7067776	20. ess	2936368

The first problem we have with the suffix list is 4., that is the result of combining 5. with the plural suffix *-s*. The penalty is not enough to pull it down, because it also gets a reward for having 3-letters. Another problem we have in 6., 10. and 13.-17. is that given the skew between the top-ranked affixes and the others, the first are likely to combine with random endings and make it to the final list.

2.2.3 Prefix corrections. The main problem of the prefix list is that a lot of 2-letter strings are ranked higher than longer real prefixes like *pro*, *inter* and *non*. A possible solution to this problem is to increase the reward for longer prefixes changing the exponent of $*(n-1)^2$ to $*(n-1)^3$. This is the result we get for the prefixes given this correction:

1. re	9240378	11. de	2993559
2. over	7505676	12. under	2392256
3. con	6538912	13. trans	1975616
4. co	6481020	14. ca	1750314
5. inter	6116160	15. pr	1636643
6. dis	4792816	16. non	1488016
7. in	3565005	17. di	1291240
8. pre	3471520	18. ma	1285991
9. un	3457940	19. st	1202182
10. pro	3192824	20. inte	1172448

The algorithm has improved a lot. The two prefixes *ca* and *pr* are in a lower position and longer prefixes like *under* and *trans* have been identified.

2.2.3 Suffix corrections. For the suffix list, we can remove all the suffixes that contains a suffix which has already been detected. For example, we can remove *ting, ating* and *ling* because they are ranked after *ing*, and *ally, ically* because they are ranked after *ly*. Notice that we cannot do it for *s* because otherwise we would lose real suffixes ending with *s* like *ness*:

1. ing	140672360	11. ts	6082470
2. ed	41898465	12. rs	2455085
3. es	23627262	13. ion	2179756
4. ers	17832630	14. al	1906592
5. er	17776912	15. ate	1858288
6. ation	12820272	16. le	1623571
7. ly	1124643	17. ns	1591186
8. ness	9224334	18. st	1561024
9. able	6901740	19. ity	1440840
10. tion	6082470	20. ic	1232983

The correction had the effect of removing some superset suffixes and get some other ones (like *al* and *ity*). We are still left with the problem of *ers*, which is a combination of two suffixes rather than a single suffix.

3 Second part: Roots

In our project, roots are determined in a function called *morpho*, which is a function that encompasses all the components of the project. Given a sample of words, ideally the whole corpus for our project, the function returns a list of prefixes, suffixes and roots along with their respective frequencies.
The function simply calls *get_prefixes* and *get_suffixes*, and uses the resulting lists (limited to the top-8 elements) to take the affixes off root words, as shown below:

```
1  for word in root_list_partial:
2        for suf in suffixes:
3            if suf in word and suf[-2:] == word[-2:]:
4                root = word[:(len(word)-len(suf))]
5                #correct final -e
6                if root in words:
7                    root_list_final.append(root)
8                elif root + 'e' in words:
```

```
9                   root_list_final.append(root + 'e')
10    # correct -i in -y
11    word_to_correct = []
12    corrected_word = []
13    for word in root_list_final:
14        if len(word) >= 1:
15            if word[-1] == 'i':
16                word_to_correct.append(word)
17                corrected_word.append(word.replace(word[-1], 'y'))
18    for word in corrected_word:
19            root_list_final.append(word)
20    for word in word_to_correct:
21        root_list_final.remove(word)
```

Finally, the root is checked in the entire corpus and, if present, is added to the list (lines 6-7). If the root is not present, there is come chance that it is present with a word final *-e*. If it does in fact exist in the corpus, then it is a real word and is added to the output list of roots (8-9). One last correction is checking the cases where the algorithm is returning roots with word-final *-i* from word-final *-y* that become *-i* because of the suffixation (10-17). The resulting list of roots are then turned into a sorted dictionary according to their frequency and the top 100 roots are returned. When it came to determining all of the roots, the importance of the sample containing all the words of the language, with each word likely to appear at least one time with no affixes attached, became apparent. In fact, when we check for a root to appear with or without a final *-e*, we make sure the root is present on the sample before adding it. This is to make sure that words that have lexicalized versions of prefixes, like *prefer, read, overt*, and *conscious*, to name a few, do not appear as *fer, ad, t*, and *scious*. Otherwise, checking to see if a root appears in the corpus would be a pointless check if the corpus is not big enough to contain all the words, since they are unlikely to appear. This is the list of the top-100 roots identified by the algorithm:

1. form 16	26. mix 9	51. dress 7	76. emphasize 6
2. cede 15	27. spire 9	52. course 7	77. writ 6
3. pose 15	28. close 9	53. turn 7	78. cuss 6
4. tend 14	29. fuse 9	54. take 7	79. activate 6
5. figure 13	30. verse 9	55. cover 7	80. connect 6
6. organize 13	31. assemble 9	56. sole 7	81. orient 6
7. serve 13	32. ache 9	57. affirm 7	82. cess 6
8. cite 12	33. determine 9	58. judge 7	83. print 6
9. cit 12	34. us 9	59. incline 7	84. issue 6
10. charge 11	35. produce 8	60. sol 7	85. wash 6
11. sign 10	36. prove 8	61. test 7	86. capitalize 6
12. unit 10	37. fin 8	62. compose 6	87. ton 6
13. view 10	38. side 8	63. embark 6	88. fold 6
14. state 10	39. place 8	64. cook 6	89. lie 6
15. unite 10	40. fine 8	65. port 6	90. play 6
16. insure 10	41. arm 8	66. twine 6	91. populate 6
17. stat 10	42. vent 8	67. calculate 6	92. price 6
18. vers 9	43. value 8	68. tire 6	93. educate 6
19. arrange 9	44. strain 8	69. estimate 6	94. furnish 6
20. engage 9	45. solve 8	70. expose 6	95. integrate 6
21. establish 9	46. act 8	71. join 6	96. de 6
22. use 9	47. heat 8	72. work 6	97. celebrate 6
23. possess 9	48. examine 8	73. claim 6	98. habit 6
24. scribe 9	49. face 8	74. write 6	99. load 6
25. weave 9	50. shape 7	75. lay 6	100. tone 6

The first observation is that we have both *cite* and *cit* as roots, while only the first one is attested in several inflected forms in the corpus (*incite*, *recite*, citeable, *cited*, *citer*, *citers*, *cites*). The reason why we have a *cit* form is that this root is present in the list, and probably it stands from the abbreviation *cit.*

In 12. and 15. we detect both the noun *unit* and the verb *unite*, even though the only inflected form in which the first is manifested is the unanalyzed *units.*

In 17. we have *stat* which is present in the vocabulary as an abbreviation of either the word *statistics* or the Latin adverb *statim*, which means "immediately", even though the inflected forms are related to the word *state* (14.), which is ambiguous between a noun and a verb.

The root 18., *vers*, is present in the wordlist, probably as a borrowing of the

French word for the noun "verse", even though the inflected forms refer to the English word.
Another mistaken pair is 22. *use* and 34. *us*, with the second one being derived from forms like *uses* even though the word is already marked for plural, but this is the consequence of an unsupervised analysis.
Then, we have 56. *sole* and 60. *sol*, even though the second one only results from the musical note G and therefore cannot be inflected.
74. *write* and 77. *writ* are both correct roots: the case is exactly like 12. and 15.
The mistake in 96. is the result of the fact that for some obscure reasons words like *des* and *der* are present in the database despite not being words of English.
Finally, the pair 87. *ton* (as "metric ton") and 100. *tone* (as "sound") is correct.

4 Discussion

4.1 The suffixes problem

The algorithm proved to be able to infer the list of affixes and the list of roots from the English corpus we used. The running time for a corpus of ≈100K was ≈60 minutes on a laptop, with ≈85% of the time being used for the suffix search. Most of the prefixes were correctly detected even outside from the top-8 cut. The roots were correctly detected, and all the mistakes seem to be part of some weird entries in the corpus (apart from the case *us-use*, which is difficult to handle in an unsupervised search). The suffix part proved to be the most problematic one and we tried to come out with some alternatives to improve the results.
First of all, we tried to apply the same modification we applied to the prefixes in the rewarding part, increasing the reward for longer from $*(n\text{-}1)^2$ to $*(n\text{-}1)^3$. This is the suffix list we obtain:

1. ing	281344720	11. ly	11246432
2. ation	51281088	12. ability	6629688
3. ed	41898465	13. ion	435912
4. ers	35665260	14. ate	3716576
5. ness	27673002	15. bility	3457750
6. es	23627262	16. ble	2965024
7. able	20705220	17. ity	2881680
8. ically	18277750	18. ment	2845206
9. tion	18247410	19. tive	27733345
10. er	17776912	20. ts	2551030

Now, the algorithm is retrieving some more suffix (*ability*, *ment* and the more complex *ate*) but is also generating many false positives, mostly of which are subset of other prefixes.
An alternative is to delete all the suffixes that contain no vowels in them. For this reason, we added to the code a function called *has_vowel* which can be used to remove all the affixes that contain no vowel. Still, the result does not increase in a satisfactory way:

1. ing	140672360	11. ion	2179756
2. ed	41898465	12. al	1906592
3. es	23627262	13. ate	2179756
4. ers	17832630	14. le	1623571
5. er	17776912	15. ity	1440840
6. ation	12820272	16. ic	1232983
7. ly	1124643	17. on	117995
8. ness	9224334	18. ent	1174600
9. able	6901740	19. tive	91115
10. tion	6082470	20. te	608290

4.2 Conclusions

On the overall, our results were acceptable for two reasons: first because in English words can appear without affixes, second because we were able to perform a search on a complete database. In languages in which the roots cannot appear without affixes and a complete sample is not available, the root extraction part cannot be constrained in the way we did and can lead to more false positives.
Another problem we did not address is the case in which multiple affixes are present in a word in the same position: in languages in which some affixes

never occur word-initially or word-finally, but after or before other prefixes and suffixes, it can be useful to run the final part of the algorithm multiple times to check for stacked affixes.

5 References

Harris, Z., 1955. From Phonemes to Morphemes. Language 31 (2), 190-222.

6 Notes

The code for the program has been built under Python 2.7.10.

The program *morpho.py* performs an exhaustive search of the database wordsEn.txt and has a running time of ≈60 mins on a notebook with an Intel Core i3-3217u 1.8 GHz processor.

The original file wordsEn.txt contained a couple of single-letter words (like *a*) which have been removed since they make our program crash. In fact, we need to have at least two letters to check the entropy of a string.

In order to perform the analysis on a reduced sample, we built another program, *morpho_sample.py*. Such program does not process the entire file, but asks the user to specify the sample size, which may vary between 1 and 109581.