Karen McNeil
May 11, 2012
Ling-420

Tunisian Arabic Morphological Parser

In Arabic, many linguistic units which would be considered separate words in English are written together as one word. Since conjunctions, possessive pronouns, and direct object pronouns are all cliticized, most nouns and adjectives will be inflected in some way. In addition, the complex morphology of the verbs means that each verb may be marked with either a prefix or suffix (or both) to show agreement for person, number, and gender. This is problematic for NLP purposes, since the inflected words would not be counted with the uninflected forms, and it will aggravate any data scarcity. To address this problem for my own corpus of Tunisian Arabic, I build a parser which used a combination of rule-based parsing and statistical measures which achieved a word-level accuracy rate of 89.2%.

For this project, I used a 400,000-word corpus of Tunisian Arabic that I have built. The corpus contains documents of many types, including television scripts, traditional folktales, and internet forum postings. It should be noted that Tunisian is officially an "unwritten" language (since any formal writing, including news and literature, would be written in Standard Arabic). Because of this fact, there is a great deal of variation in the orthography of the corpus texts; the informality of many of the sources compounds this effect.

Current Approaches to the Problem

A common way of performing morphological parsing in Arabic is using the Buckwalter Arabic Morphological Analyzer (BAMA), an open-source software package developed by Tim Buckwalter and distributed by the Linguistic Data Consortium (Buckwalter, 2002). This is the method that the Penn Arabic Treebank (ATB) used to construct their corpus of Standard Arabic, which is annotated for part of speech, morphology, and sentence structure. BAMA uses a group of lexicons (prefixes, suffixes, and stems) as well as tables listing valid morphological combinations in order to produce a list of possible parses for each word. The ATB's project leaders reported a parsing accuracy rate of 98.7% (Maamouri & Bies, 2004, p. 5).[1]

Unfortunately, Buckwalter's parser will not work for my data. Although Tunisian Arabic and Standard Arabic are closely related languages[2] and share many features, the differences between them in lexicon and syntax are substantial. Modifying BAMA's prefixes and suffixes to conform to the grammar of Tunisian would be relatively trivial, but there is no existing lexicon of Tunisian Arabic, and many of the most common words of the language are not present in Standard Arabic.

[1] It should be noted that this success rate is not directly comparable to mine here. The data that the ATB is based on is very clean, being mostly news and other formal sources in an established written language. This level of accuracy is not possible in a language like Tunisian, which has no established spelling conventions and in which its speakers are not educated.

[2] There is a great deal of controversy over whether the spoken varieties like Tunisian are separate "languages" or merely a "dialects" of Arabic. Suffice to say that for most NLP purposes they should be treated separately.

An alternate approach is a wholly statistical one, such as John Goldsmith pioneered with his program *Linguistica* (Goldsmith, 2001). Rather than rely on human-written lexicons, like BAMA, this program uses a calculation of entropy to determine the boundaries between morphemes, and so induces the list of prefixes, stems and suffixes automatically. This method does not work very well for my data, as it does more parsing than desired. Although often referred to within the Arabic NLP community as "morphological analysis", the task I'm trying to accomplish here can be thought of more as tokenization, since what I'm interested in doing is breaking up what would be separate words in English. Goldsmith's parser, however, does full morphological analysis; for example, in analyzing the following two verbs:

استجبتو	جبتو
astjbtw *(stajeb-t-u)*	Jbtw *(jib-t-u)*
answered-1PS-it/him	brought-1PS-it/him
"I answered him"	"I brought it"

the Buckwalter parser (and my parser) would recognize the *-t* as an inflectional ending (in this case, the first-person past-tense verb suffix), and the *-w* as a direct object pronoun, neither of which should be counted as part of the stem. A statistical parser like Linguistica, however, would also recognize that *sta-* is a feature of many verbs (it's part of the extensive derivational morphology of Arabic), and also strip that from the stem. The result would be that these two verbs (which, although they share a common root, are really quite distinct), would be lumped together as *jb*.

Sajib Dasgupta and Vincent Ng (2007) presented a very clever extension of a statistical parser in their paper "High-Performance, Language-Independent Morphological Segmentation" (2007). They were able to improve upon Goldsmith's methodology by adding two corrective processes after the initial parsing:

1) they identified parses that were probably incorrect by comparing the frequency of the induced stem with the frequency of the original form, and by computing the frequency of the induced stem with similar affixes; and
2) they induced spelling rules to account for letters which are required to change when an affix is attached (i.e. y→i for English).

Their procedure was the inspiration for this current work.

Methodology

For this project, I used a combination of the methods discussed above. Since the set of prefixes and suffixes for Tunisian Arabic is limited, brute-force statistical parsing is not necessary (in addition to being inappropriate for the task, for the reasons discussed above). Instead, I employed a rule-based parser similar to Buckwalter's. I built this parser using Python's pyparsing module, and a grammar I wrote of valid prefixes, suffixes, and word forms. In this way it is similar to the Buckwalter parser, but—because I don't have a lexicon of valid word stems—it necessarily produces erroneous parses with non-existent stems. However, by applying the some of the statistical methods that Dasgupta and Ng describe, I was able to discard many of the erroneous parses and achieve a fairly good accuracy rate.

The steps of this process were:

1) Text normalization
2) Create and hand-annotate test data
3) Parse words using pyparsing
4) Baseline evaluation
5) Identify incorrect parses using the Word-Root Frequency Ratio
6) Re-evaluate
7) Resolve ambiguous cases using Suffix Level Similarity
8) Re-evaluate

I will go over each of the steps, as well as the results achieved, in detail below.

Preprocessing

The first step was to prepare the raw Arabic text for automatic processing. To do this, I first extracted all of the corpus text (which is stored in just under 2,000 database records) to a single text file. I then ran the text through a script I wrote which strips out foreign words and punctuation, and transliterates the Arabic text into a Latin transliteration system. The transliteration system I used is a modified version of the Buckwalter transliteration, adapted for the slightly different phonology of Tunisian Arabic, and modified so that no punctuation or special characters are used. For example, the script would take an Arabic word like بـالـيـد (*b-il-yed,"* by/with-the-hand") and transliterate it into `balyd`. Importantly, the script also strips out foreign words (which are very common in Tunisian text, especially French words), and so prevents them from being included in the morphological parsing. For simplicity's sake, I choose to just remove the foreign words during development, but in actual usage I would instead mask them from the transliterator and parser, while retaining them in the text.

My next step was to create the data that the parser would be tested against. Because the gold-standard evaluation data had to be hand-parsed, I chose a modest test set of 2,000 words (representing about 5.5% of the corpus). I selected the test data from different parts of the corpus by adding the last five of every two thousand words to the test data file (and removing them from the training data file to avoid over-training). I marked the segment boundaries manually (with a '+' in between segments), then asked a native speaker of Tunisian to review my annotations and make any corrections.

Parse Words Using pyparsing

To implement the base parser using pyparsing, I wrote a basic grammar describing the morphological rules of Tunisian Arabic. Here's a small portion of the grammar:

```
conjunctions = ['w']
prepositions = ['l','b']
def_art = ["al", "l"]
poss_suffixes = ["y", "ya", "na", "k", "km", "w", "h", "ha", "hm"]
noun_suffix = oneOf(poss_suffixes) + FollowedBy(endOfString)
def_noun =  ( Optional(oneOf(conjunctions)) +
              Optional( oneOf(prepositions) ) +
              oneOf(def_art) )("prefix") + \
               SkipTo(endOfString)("stem")
```

These rules define the suffixes that can attach to a noun as one of a choice of literals that come at the end of the word. Then one of my major word types (definite noun) is defined as a combination of allowable noun-prefixes, and a word stem – which is anything in between the prefix and the end of the string. (Many word types can also have suffixes, in which case the stem is defined as anything between the prefix and either the suffix or the end of the string.) I then fed the transliterated text, one word at a time, to the parsing function. The parsing function would try the word against the definition of each word type in the grammar (i.e., possessive noun, definite noun, present-tense verb, past-tense verb), and output a list of possible parses, like this:

```
['w', 'b', 'al', 'yd']               # definite noun
- prefix: ['w', 'b', 'al']
- stem: yd
['wbalyd']                           # uninflected word
- stem: wbalyd
['w', 'balyd']                       # conjunction + uninflected word
- prefix: w
- stem: balyd
```

This word produced three possible parses, the correct of which is the first one. (The comments show the word type from the pyparsing grammar that the parse is resulting from.) For my baseline evaluation, I then checked the frequency of the parsed stem in the training data, and selected the parse with the highest stem frequency as the “right” parse.[3] To continue the above example, `'wbalyd'` would produce three parses with associated stem frequencies:

```
➔ ['w','b','al','yd'] , stem: yd , stem frequency: 0.000115
  ['wbalyd'] , stem: wbalyd , stem frequency: 2.50e-06
  ['w', 'balyd'] , stem: balyd , stem frequency: 7.50e-06
  'w+b+al+yd'
```

Since the stem of the first parse (`'yd'`) has the highest frequency of the three choices (0.000115), the algorithm selects it as the parse for this word. In this particular example, the parse chosen is also the correct parse.

Baseline Evaluation

For each of the parsed words, I transposed the test parse and corresponding gold-standard parse into a string of one’s and zero’s showing segmentation boundaries (i.e., if the letter is followed by a segment boundary, it becomes a 1; if not it becomes a 0):

```
w+b+al+yd   →  11010
w+balyd     →  10000
```

[3] This was actually the second baseline I tried. My initial baseline selected the parse with the *shortest* stem length, without regard to frequency. Although that actually performed surprisingly well (77% accuracy), it was difficult to build on in a principled way. Any frequency calculations added were very ad hoc, whereas they’re a natural extension of the stem-frequency-based selection.

Since the last letter of the word will always be a boundary that the parser doesn't have to assign, I didn't count it word either way, so each binary string is one character shorter than the input string. With the binary string I then computed the accuracy, recall and precision of the parse: if the test parse had a 1 where there was supposed to be a 0, that took a point of precision. If it had a 0 instead of a 1, minus one for recall. While the accuracy, recall, and combined F-score were scoring per segment, the accuracy is per word: whether the entire word is correct or not. So the correct parse `['w','b','al','yd']` would score 1/1 on accuracy, 3/3 on recall and 2/2 on precision. Had the parse, `['w', 'balyd']` been selected instead, it would have scored 0/1 on accuracy, 1/3 on recall, and 4/4 on precision. These scores were then used to compute a cumulative score over every word in the test data, yielding the following baseline results:

Recall: 0.4507
Precision: 0.9837
F-Score: 0.6182
Accuracy: 0.6688

When examining the log of incorrect parses, I found that this method was very likely to select for the entire word without any parsing (as you can see from the excellent precision, at the cost of a very low recall score). The words that suffered most from this were common collocations, such as b+al+Cks (*b-il-ʕaks* 'in-the-opposite'):

```
➔ ['w', 'bal', 'Cks'] , stem: Cks , stem frequency: 7.28e-05
  ['wbalCks'] , stem: wbalCks , stem frequency: 2.51e-06
  ['w', 'balCks'] , stem: balCks , stem frequency: 8.79e-05
  'w+balCks'
```

The first parsing (`'w+b+al+Cks'`) is the correct one, but as you can see from the final line of output, the algorithm selected the third parse (`'w+balCks'`), which indeed had the highest stem frequency (underlined). The problem is that this phrase is used to mean "on the contrary" and is a common discourse marker, much more common than the stem word used in its basic sense of 'opposite' or 'reflection'. So the unparsed stem of `'balCks'` was the most frequent in the corpus. But even words that weren't obvious collocations such as al-vany (*aϑ-ϑānī* 'the-second') were affected:

```
  ['alvan', 'y'] , stem:  alvan , stem frequency: 0.0
➔ ['al', 'vany'] , stem:  vany , stem frequency: 4.50e-05
  ['alvany'] , stem:  alvany , stem frequency: 0.000200
  'alvany'
```

I believe that this is due to the simple fact that definite nouns and adjectives are more common than indefinite ones in Arabic. (A phrase which is indefinite in English will often need to be translated as definite in the Arabic.) This makes the inflected forms of many words more common than the naked stem – a fact which we will see causes problems for Dasgupta and Ng's word-root frequency ratio.

Identify Incorrect Parses Using "Word-Root Frequency Ratio"

Dasgupta and Ng's hypothesis for their "incorrect attachment algorithm" was that "if a word *w* is formed by attaching an affix *m* to a root word *r*, then the corpus frequency of *w* is likely to be less than that of *r* (i.e. the frequency ratio of *w* to *r* is less than one)" (2007, p. 4). In reality, they found that the ratio is often not actually less than one, but correct parses tend to be low (less than 2 or 3), whereas incorrect parses tend to be much higher (in the tens or hundreds). The example they give is the word "candidate": their statistical parser (correctly) identifies "candid" as one of the roots of English and "-ate" as one of its affixes. However, it *incorrectly* analyzes the word "candidate" as "candid" + "ate", rather than as a root of its own.

But compare the frequency of "candid" (appearing in their corpus only 119 times) to "candidate" (which appeared 6380 times). By dividing the frequency of "candidate" by "candid", the "candid+ate" parse received a high WRFR score of 53.6. By comparison, a correct parse such as "bear+able" would have a WRFR of 0.01. Although the WRFR was not always below 1 (that hypothesis only held for 71.7% of English words and 83.6% of Bengali words) the authors determined that a WRFR threshold of 10 for suffixes and 2 for prefixes was sufficient to identify most incorrect parses.

My implementation of this idea was slightly different. Rather than using a WRFR threshold to identify incorrect parses, I computed the WRFR for each of the possible parses and selected the parse with the lowest ratio. Contrary to expectations, this more sophisticated method actually did slightly worse than the baseline method of selecting by frequency:

Recall: 0.4204
Precision: 0.9868
F-Score: 0.5896
Accuracy: 0.6603

When examining the incorrect parses log, we see the same issue of high frequency inflected forms:

```
➔ ['w', 'bal', 'Cks'] , stem:  Cks , wrfr: 0.034
  ['wbalCks'] , stem:  wbalCks , wrfr: 1.0
  ['w', 'balCks'] , stem:  balCks , wrfr: 0.028
  'w+balCks'
```

I tried implementing this measure in several different ways, but without any success. Although their basic idea was sound (completely bogus parses did have a high WRFR), it wasn't any more useful than the stem frequency. I suspect that the extent to which WRFR is useful is language dependent, and Arabic just happens to be a language that it is not very good for. I ended up not using it at all in the final parser.

Affix Similarity

The second of Dasgupta and Ng's procedures, suffix level similarity, ended up being the most useful for my parser:

Suffix level similarity is motivated by the following observation: if a word *w* combines with a suffix *x,* then *w* should also combine with the suffixes that are "morphologically similar" to *x*. To exemplify, consider the suffix "ate" and the root word "candid". The words that combine with the suffix "ate" (e.g. "alien", "fabric", "origin") also combine with suffixes like "ated", "ation" and "s". (2007, p. 5)

Using this measure alone, "candid+ate" would be identified as an inaccurate parse, since there are no instances of "candidation," "candidated," etc.

Dasgupta and Ng computed the affix similarity using a probabilistic measure, but that's not necessary for my parser, since I already know which affixes behave similarly. In fact, I intentionally defined them as such when I was writing the parsing grammar. Using pyparsing's syntax, I could have just written the affix rules in one line as:

```
noun_suffix = oneOf("y ya na k km w h ha hm") + FollowedBy(endOfString)
```

but instead I wrote all the rules separately, like this:

```
poss_suffixes = ["y", "ya", "na", "k", "km", "w", "h", "ha", "hm"]
noun_suffix = oneOf(poss_suffixes) + FollowedBy(endOfString)
```

so that I would be able to access the list of similar affixes for this step of the process.

Since I have lists of all of the similar affixes, now all I needed to do was write code testing the goodness of fit for each word type. For example, for present tense verbs:

```
def ave_vbz_freq(prefix,stem,debug=False):
    vbz_freq = []
    for pre in [p for p in vbz_prefixes if p != prefix]:
        vbz_freq.append(fd.freq(pre + stem))
        if debug: print "prefix:", pre, ", freq(", pre+stem, ") = ",
                    fd.freq(pre + stem)
    return sum(vbz_freq)/len(vbz_freq)
```

This code takes a parse that is supposed to be a present tense verb, and tests if the "stem" behaves like a verb. It does this by taking the stem and combining it with all of the verb prefixes, except the one in this parse, and computing the average frequency of all those verb forms. For example, the word `'ynZr'` (*yu-nδur,* "he sees") is parsed in the following way by the WRFR-based parser:

```
➔ ['y', 'nZr'] , stem:  nZr , wrfr: 40.067
  ['ynZr'] , stem:  ynZr , wrfr: 1.0
  'ynZr'
```

As we can see, the conjugated form of the verb is more common than the base form (`'nZr'`), so the WRFR-based parser would select the incorrect parse. The affix-similarity-based parser, however, would examine each of these parses and see if it's a good fit for the type of word it's supposed to be:

```
  ['y', 'nZr'] present verb
➔ ['y', 'nZr'] , with present verb score of 0.000435

  ['ynZr'] uninflected
  ['ynZr'] , with poss noun score of 0.0
```

```
['ynZr'] , with poss vbz score of 0.0
['ynZr'] , with poss vbd score of 4.165e-07
['ynZr'] , with discounted frequency 0.000376

'y+nZr'
```

The parser is evaluating two hypotheses here. The first is that this is a present tense verb. To test this hypothesis, the goodness of fit function tests the proposed stem in combination with all of the verbal prefixes except `'y'`, and returns the average frequency of all forms (0.000435). It then tests the second hypothesis, that this word is uninflected and the whole word is the stem. In order to do this, it tests against every kind of word this *could* be. So it adds all of the nominal affixes to the (non-existant) stem `'ynZr'` to get an average noun frequency of 0, and does the same for present and past-tense verb affixes, assigning to each parse a score. Of all these possibilities, the parse chosen is the one with the highest goodness of fit score for its word type, in this case `'y+nZr'`, which is indeed the correct parse.

Final Evaluation

The affix-similarity method greatly improved the parser's accuracy, compared to the baseline:

```
Recall: 0.8736
Precision: 0.9740
F-Score: 0.9211
Accuracy: 0.8924
```

This level of accuracy is fairly good, considering the simplicity of the grammar. I believe that with further development of the grammar and parser (especially adding conditions to address irregularities), this could be increased even more. I plan to work on that as my next step in this project, in the hopes of reaching a word-level accuracy of 93-95%.

Bibliography

Buckwalter, T. (2002). *Buckwalter Arabic Morphological Analyzer Version 1.0.* University of Pennsylvania. Linguistic Data Consortium.

Dasgupta, S., & Ng, V. (2007). High-Performance, Language-Independent Morphological Segmentation. *Proceedings of Human Language Technology (NAACL).*

Goldsmith, J. A. (2001). Unsupervised Learning of the Morphology of a Natural Language. *Computational Linguistics, 27*(2), 153-198.