LIN511 Homework 6

Paul Eberhart, Ghazelah Kazeminejad, Drake Hyman, Lauren Ashburn, Kimberly Haney

April 26, 2013

Persian

The Persian language has a variety of interesting properties relevant to producing an effective PoS tagger, or produce any computational analysis. First, Persian uses a 32 character alphabet, which is a superset of the Arabic alphabet, each character of which presents four different forms depending in its position in the word. It is written right-to-left, and is necessarily cursive, as the ligature system is significant. Syntactically, it has an approximate SOV word order, with object markers allowing the object to be relocated. It also has two different sorts of space (Half spaces and full spaces) which fill different roles in morphological and syntactic construction. There is a compounding system for constructing verbs, adverbs, prepositions that requires ngram comprehension, as the individual particles produce different semantics than the combined ngram. Short vowels are not represented in the written form, and some words, notably "of", are single short vowels, and are thus entirely unrepresented in text, analogous to English's understood pronouns.

As far as exclusively morphological features, Persian lacks clear morphological markers for a number of major word classes, the most difficult of which is that Nouns and Adjectives have no morphological distinction. There are three varieties of plural marker suffix, which are unfortunately not unique suffixes, and thus simple matching will cause over-generation. Persian also lacks a marker for proper nouns analogous to the first letter capital rule in English. Finally, it uses the same suffix for different functions on different classes of words – "i" can be appended to verbs, nouns, or adjectives, and carries out entirely different roles in each case.

Unicode

A primary challenge in manipulating languages distant from English is in the computer tools. Historically, the vast majority of software assumed Left-to-Right, Latin alphabet encoded as 7- or 8- bit ASCII. In 1987, a group of computing and language representatives, lead by Joe Becker from Xerox, and Lee Collins and Mark Davis from Apple, developed a 16-bit encoding that supported the bulk of languages in widespread use at the time. This became Unicode 1.0 in 1991, and was improved to support a much wider selection of historical and unusual languages, and abstracted away from the fixed 16-bit encoding to a variety of flexible encodings in the 2.0 standard shortly thereafter. . Unicode supports a number of interesting abstractions, such as representing characters as code points (numbers assigned to a particular character) *independent* of the glyph representation, allowing, for instance, the multiple forms of Persian letters to be encoded identically. It also introduces special non-printing characters that act as markers for text direction, such that it is possible to consistently represent, and even intermix, left-to-right and right-to-left scripts. This bidirectional text feature is known as BiDi, uses an extremely complicated semi-standard algorithm, and is the source of many awkward edge cases and non-compliant behaviors in software which otherwise properly handles Unicode.

Modern Unicode supports in excess of 110,000 characters in 100 different scripts in the standard, plus a number of community maintained non-standard local pages for other scripts, especially for various constructed languages. The most common representation used for Unicode is currently UTF-8, "Universal Character Set Transformation Format 8-bit," which uses one to four eight-bit bytes to represent each character, and conveniently uses the first byte in exactly the same way as the older,

simpler ISO/IEC 8859-1 "Latin-1" representation, which in turn uses its first seven bits in the same way as ASCII, providing a high degree of backward compatibility for the common case.

Conveniently, Python 2 includes rather competent Unicode support, including in the `re` regular expression library. It is not the *default* mode for Python, and hence there are a large number of special markings and manual encodings and decodings required to manipulate Unicode strings in Python. Less conveniently, most software handles BiDi in broken and inconstant ways, such that identical strings appear in different orders depending on the particular editor in use. In particular, note that the first and last anchors in each of the regular expressions in the attached code are displayed in LtR order, but interpreted in RtL, so they appear backwards. This is not an inherent property of the string, but a property of the BiDi behavior of the software displaying them, and in fact the different editors and terminals used in putting this project together exhibited different bugs around direction changes, including entirely ignoring them. Eve more troublesome, NLTK's Unicode support is substantially incomplete, and one of the NLTK classes used required some modification to correctly handle Persian strings.

Design

Our tagger attempts to make use of two distinct passes. First, a tagging pass is attempted using a mixture of regular expressions to distinguish morphological rules, such as prefixes and suffixes unique to particular language features, and simple lexicon matching, performed with regular expression alternation in the same tagger in order to avoid introducing additional passes.

A second pass performs simple bigram analysis, improving results on words which are difficult to morphologically classify by using syntactic analysis over pairs of words. As a simple example, the word immediately preceding a period can safely be assumed to be a Verb.

Implementation

The entire system is implemented in Python 2 and NLTK (The Natural Language Tool Kit). The first pass tagger is an instance of `RegExpTagger` from NLTK's sequential tagger tools, modfied to correctly support Unicode, as NLTK's Unicode support is incomplete. - this was simply a matter of modifying a few lines in `/usr/lib/python2.7/site-packages/nltk/tag/sequential.py` to include `.decode("utf-8")` in the calls to `re` functions, such that the text is in the correct encoding state at the time of processing, and then importing the modified copy of sequential after nltk to mask the original.

Several of the regular expressions use simple alternation to act as lexicon taggers, in order to stick to familiar tools and avoid modifying another piece of NLTK. Of these, a number of the lexical classes are generated using variable substitution; that is, regular expression components, such as lists of possible roots or affixes, are stored as string variables, which are then combined and fed into the regular expression tagger as expressions.

The output of the RegExpTagger is fed into a simple if/else structure, which captures the current and previous tuples, and compares their tags against various set syntactic rules, using a high-certainty tag on one of the current pair to set the tag of the other, thus implementing simple bigram rules. For example, the word preceding a period will always be a Verb, so if the second token in the current pair under consideration is tagged as a period, the tag of the preceding token will be changed to Verb.

The full source code is attached to the end of this document.

Result

We have a moderately successful tagging system - it will digest arbitrary blocks of Persian text, and tag the words for which it has rules. It is not capable of tagging the full language, as it relies on a number

of small lexicons, has a limited set of morphological and syntactic rules actually implemented. The primary limitation is that it cannot process compound words - the tokenizer has no way to distinguish the full spaces inside of compounds from the full spaces between words, and thus will tokenzie out the words into components, which would require relatively sophisticated, likely heuristic, ngram analysis techniques to recombine. Adding such a system is one obvious avenue for improvement, others would address the earlier points: the addition of a larger lexicon, the implementation of a larger set of rules, and more complete rules, in each of the phases. dWe have encountered suggestions that statistical tagging methods are more suited to Persian, so perhaps this design is not even the correct approach, but it is remarkably functional for such a simple system.

As far as evaluation, we had only a few fragments of hand-tagged text available to compare against. Our primary evaluation method was by setting the default rule to an invalid tag, addint rules, then introducing words we expected to trigger the rule, and evaluating if they were handled. There exists a tagged corpus of Persian text on-line, but it requires institutional credentials to use, which were not set up or available on the time scale of this project.

Three versions of a small passage, untagged, tagged manually and processed by our tagger are attached.