Rafiki: Smarter Code Autocomplete

James Gippetti, Adrien Fallou, Susan Tu

I. Abstract

Most programmers use autocompletion tools to write code faster. However, basic language-agnostic autocomplete tools suffer from two weaknesses: 1) they only complete one word at a time, and 2) they complete the current word by matching leading characters. Our project is motivated by the popularity of more advanced autocompletion plugins such as Jedi (for Python autocompletion)[1] and Visual Studio's Intellisense. Our goal is to provide smart, language-agnostic autocomplete. We demonstrate this strategy by building a tool that provides autocomplete for the Python programming language. We then discuss how our model can apply to other languages.

II. Modified HMM Model

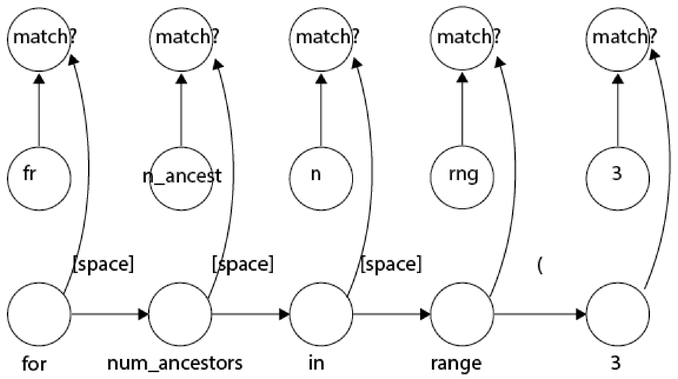

Fig 1. HMM Model with emission probabilities replaced by match probabilities.

Approach: The user must correctly input separators between abbreviated or incorrectly spelled names. Here `names' refers to names of functions, classes, variables, as well as keywords in the language. We do not attempt to un-abbreviate string or numerical literals.

Emission Probabilities: In a standard HMM model, we need to estimate P('fr'|'for'). However, there are infinitely many permutations of characters that could be attempts to abbreviate 'for', so it is unclear how we would estimate the probabilities that we would need to solve the HMM. Han, et. al. propose replacing the emission probability with a match probability. The domain for this match node (i.e., the top-most row of nodes in Figure 1) is {0,1}, and the probability associated with the transitions to them is $P(y=1 | t='fr', s='for')$, where y is the value of the match node, t is the value of the emission, and s is the value of the hidden state. Han et. al., obtain the probabilities for the match nodes by treating the assignment of the hidden state as a multiclass classification problem for which they use one versus all logisitic regression.

Features for emission probabilities:

- number of consonant letter matches
- number of capitalized latter matches
- number of non-alphabet character matches
- number of letter matches, ordering ignored
- number of letter matches, ordering enforced
- percentage of matched capital letters
- percentage of matched consonant letters
- percentage of matched letters
- percentage of matched letters
- has the word previously appeared in the file
- number of lines to preivous occurre

Transition Probabilities: The transition probability from state s_i to s_{i+1} is

$$p(s_{i+1}|s_i) = \lambda * T(s_i, c_{i+1}, s_{i+1}) + \lambda * T(s_i)$$

where T(s_i,c_{i+1},s_{i+1}) is the proportion of times that s_i followed by c_{i+1} is followed by s_{i+1} and T(s_i) is the proportion of times that statements start with s_i. (from Han, et. al.)

Parseability (in-progress work): As we choose states for successive timesteps, we only consider states such that the states up to that time step are a prefix of a valid parse. (our addition)

IV. Results

directory 1 - celery_baby.test 2 - celery_small.test	feature vector	abbrFn 0 - as is 1 - shuffle, remove 2 - no vowels, shuffle, remove	random constant (value used in abbrFn)	training size	training error	test size	test error
1	all	0	n/a	19	0.105	5	0
1	all	1	0.15	19	0.105	5	0
1	all	2	0.15	19	0.105	5	0
1	all	1	0.3	19	0.158	5	0
1	all	2	0.3	19	0.158	5	0
1	all	1	05	19	0.211	5	0
1	all	2	0.5	19	0.263	5	0
2	all	0	n/a	36	0.083	10	0.2
2	all	1	0.3	36	0.055	10	0.2
2	all	2	0.3	36	0.111	10	0.2
2	all	1	0.5	36	0.111	10	0.2
2	all	2	0.5	36	0.166	10	0.3
2	words only	1	0.5	36	0.138	10	0.2
2	words only	2	0.5	36	0.166	10	0.3
2	shared letter count	1	0.5	36	0.25	10	0.4
2	shared letter count	2	0.5	36	0.305	10	0.5

Fig 2. Test and trained on two smaller python files. Varied abbreviation function which creates the "observed" code line - can remove vowels, shuffle neighboring letters, and remove letters at random. Intuition aligns with results here - the more obscured the observed, the worse the performance. The elements in the feature vector for logistic regression were also varied. The scope features we added did not make a noticeable difference, which is expected due to the small size of our test files.

V. Prelimary Conclusions and Ongoing Work

We find that one of the weaknesses of our algorithm is its runtime: O(n^m), where n is the number of unique tokens in the training text, and m is the length (#of tokens) of the observed line of code. Next steps include a particle filtering implementation to improve speed and improvements in the logistic regression feature vector to lower error.

Han, S., Wallace, D.R., and Miller, R.C. Code Completion From Abbreviated Input. \emph{IEEE/ACM International COnference on Automated Software Engineering}, 332-43, 2009.