Aaron Lee and Jared Cole

COSC 10

Problem Set 5 Extra Credit Documentation

We implemented a few new features for extra credit. They include:

**Trigram training.** As suggested by the PS-5 question page, we implemented trigram training into the Trainer.fileTrainer() method. To implement this, we created a new Pair object which is simply a pair of strings. Then, we mapped Pairs of strings to Maps of Strings to doubles. Since the trigram essentially needs to store P(third state is a given POS | first POS and second POS), our trigram map structure simply creates a Pair holding the first POS and the second POS; it then maps that sequence of two POSses to a Map that maps the third POS to the corresponding probability. Getting the probability, then, of the trigram would look like trigram.get(new Pair(POS1, POS2)).get(POS3).

**Unigram training.** The unigram is simply P(some POS)—that is, simply the probability that any given word is a certain POS based on the distribution of POSses in the text. To perform unigram training is very simple; simply divide the number of times each POS appears by the total number of tokens to get the probability that any randomly selected token is a given POS.

Both of the above training structures (trigram and unigram), as well as the original (transitions, or bigram) had their Math.log() step removed, and the log() step restored in the Viterbi algorithm itself. We made this decision so we could implement

**Trigram interpolation.** Why else would we bother implementing both trigrams and unigrams? The general idea is that we can take a weighted sum of all three transition probabilities (trigram probability, bigram probability, unigram probability) to create a compound probability that’s more robust than the original. However, we can’t take the weighted sum of already-logarithmed values, so it was necessary to move the logs to the Viterbi algorithm itself. To take a weighted sum, we need weights; in Viterbi.java are three constants: a, b, and c, which are the weights for trigram, bigram, and unigram probabilities, respectively. The final compound probability is then logged and added to the nextScore calculation (using a = 0, b = 1, c = 0 would produce an identical result to the original bigram calculation). What’s interesting is that varying a, b, and c will affect the outcome, but with much less drama than one might expect; using naive values like 0.333 for each produces fairly good results of about 3000 wrong tags, and simply varying a, b, and c by hand, we were able to produce a range of error values between ~1200 to ~3000, but nothing outrageously bad. On the other hand, using trigram interpolation wasn’t a magic bullet that instantly produced outrageously good results, either. It would probably be unreasonable to expect it to, as well, when only some 3.5% of tags were wrong to begin with. Maybe we could’ve found the optimal values with a lot of testing; certainly, many reasonable values of a, b, and c perform close to the original bigram model, and seemingly with less variance. Further testing would be helpful to determine the best a, b, and c, but it was too tedious to do by hand, and too computationally expensive to brute-force. Furthermore, with a different (and more importantly, a bad) data set (e.g. the simple data set), the trigram model appears to be somewhat more robust to weird effects, when given the right parameters (usually a safe bet is relatively high, but not very high—about 0.4 in a and b, and a lower but non-negligible c).

**Viterbi generation.** We also implemented an extension to the Viterbi algorithm that generates the next n most likely POS tags after the last given observation. Basically, it runs through the string to get a list of “final” states to “seed” generation; it then simply continues to run through the algorithm, without taking observed values, until all n tags are generated. There are a lot of nuances whenever n > 2, however; for instance, punctuation (mostly .) plays a strange effect where strings of .s tend to have much better transition probabilities than strings of normal non-degenerate tags. This makes sense, to an extent; for most of the data set, whenever a . appears, it’s the last tag within its sentence, and doesn’t transit to any normal word. The result is that any given valid sequence of words, like DET N V DET N, will have a vanishingly small score, while a degenerate string like . . . . . will conversely have a more stable score—this is because there are no more observations to prevent penalize such degenerate sequences. In a normal Viterbi algorithm, non-periods are never tagged as periods; therefore, the unseen score would quickly rule out the possibility of getting trapped in a sequence of periods. However, since using the model generatively gives no such check, such bad results often occur.

However, when n = 1 or n = 2, the model often generates very reasonable continuations to the sequence. For instance, with n = 2, we have (where the whole model was trained on the Brown data):

Format: “input” results in “observed tags” | “generated tags”

“the” results in DET | N P

“the cat” results in DET N | . .

“the cat in” results in DET N P | DET N (example phrase: the cat in | the hat)

“the cat in the” results in DET N P DET | N P (e.g. the cat in the | hat under)

“the cat in the hat” results in DET N P DET N | . .

“the cat in the hat sat” results in DET N P DET N VD | DET N (e.g. the cat in the hat sat | the rat)

“the cat in the hat sat on” results in DET N P DET N VD P | DET N (e.g. the cat in the hat sat on | the rat)

“the cat in the hat sat on the rat” results in DET N P DET N VD P DET N | . . (a valid sentence)

What’s interesting here is that the generative algorithm seems to recognize phrases that are somewhat complete, like “the cat” or “the cat in the hat” or the full sentence “the cat in the hat sat on the rat.” In these cases, it suggests the degenerate continuation . ., which for the full sentence, is probably the best response. However, when prompted by a hanging verb or a hanging preposition, the generative algorithm recognizes that some valid response is better than the degenerate response, and suggests what is often a working pair of POS that would continue the sentence in a valid way (often DET N).

For another example of the same phenomena:

“he” results in PRO | MOD V (e.g. he will | fix)

“he will” results in PRO MOD | V DET (e.g. he will | fix the)

“he will fix” results in PRO MOD V | DET N (e.g. he will fix | the car)

“he will fix the” results in PRO MOD V DET | N P (e.g. he will fix the | car on)

“he will fix the car” results in PRO MOD V DET N | . . (a valid sentence)

“he will fix the car on” results in PRO MOD V DET N P | DET N (e.g. he will fix the car on | the weekend)

“he will fix the car on saturday” results in PRO MOD V DET N P N | . . (a valid sentence)

Once again, we see reasonable continuations until the program recognizes that what seems to be a valid sentence structure has been generated, and then it terminates.

Another interesting idea, if we notice that n = 2 can generate at least one or two seemingly-reasonable steps, is instead of trying to outright generate n > 2 new POS tags, we could generate two tags, then feed the next one tag back into the program to generate two more, and so forth. The likely result would be sentences that continue until they form a single logical unit (as seen above, often a complete sentence) and then terminates—in this context of generating sentences, this would be a very strong result.