Aaron Lee and Jared Cole

COSC 10

Problem Set 5 Documentation

Test the method on simple hard-coded graphs and input strings (e.g., from programming drill, along with others you make up). In your report, discuss the tests and how they convinced you of your code's correctness.

We hard-coded the graph from programming drill, and ran several test strings on it. verifying the true correct answer by hand each time. For the first test string, we used the same one we ran through in drill (chase watch dog chase watch); the result (NP V N V N) was the same as the solution given in drill. We also ran it on a training input (catch watch watch and chase), which faithfully reproduced the training output (N V N CNJ NP) as expected. We tested a new sentence (cat and dog watch chase chase watch), which produced the same answer that we verified separately by hand (N CNJ N V NP V N); finally, we tested the behavior of the algorithm, on a string containing completely unseen words (cat and dog immigrate to america), which should then more-or-less rely entirely on transitions scores for unseen words (as all observed values would be the unobserved default of -10). The result (N CNJ N V NP V) is as expected; after “cat and dog,” the program runs entirely based on the most likely transition, with N’s highest scoring transition to V with 8; V’s highest scoring transition being a tie between N and NP at 4; and both N’s and NP’s highest transition being again V with 8 (which means N and NP are interchangeable for the 5th word in this sentence, and the actual selection by the algorithm is arbitrary). Doing the same by hand verifies this result, with the correct true solution being “N CNJ N V NP/N V.”

These hard-coded examples convinced us that our actual viterbi algorithm code (Viterbi.viterbiAlgorithm() method) was working entirely as expected. It might not be perfectly accurate in its answers, but it behaves as the Viterbi should (again, as verified by hand) for each of the test cases above. The more rigorous tests later (simple and brown tests) provided more evidence that the core algorithm works well.

In a short report, provide some example new sentences that are tagged as expected and some that aren't, discussing why. Also discuss your overall testing performance, and how it depends on the unseen-word penalty (and any other parameters you use).

“the the the the the .” was tagged as DET DET DET DET DET . This is somewhat expected as “the” is basically always a determiner, and has no other alternative; despite the fact that the maps likely capture the fact that determiners usually don’t lead to other determiners, the unobserved penalty keeps the result accurate.

“the cat chased the rat eating the hat .” was tagged as DET N P DET N VG DET N . This is mostly correct; most of the determiner-noun pairs were right, as well as the gerund verb (VG). Chased, however, was mislabelled P (preposition); this seems bizarre at first, but it turns out that the training data didn’t use the word chased a single time, so every tag had an equal unseen word penalty, and a preposition does make some sense in this sentence (e.g. “the cat under the rat eating the hat .”).

“there’s a cat in my hat .” was tagged as EX DET N P PRO N. This is a relatively simple sentence with commonly used words (and therefore a lot of training data), and the program tagged it entirely correctly.

When previously unseen words appeared in predictable locations, the algorithm was sometimes able to fill in the gap correctly. “the oh-dear-looks-like-i-spelled-something-wrong had investigated the whereabouts of the cat .” correctly returns DET N V VN DET N P DET N . Clearly, it seems that the never-seen-before word, logically placed where a noun would be, was correctly tagged by the algorithm regardless.

However, when the gap was less obvious, the program failed in this regard. “the rat had oh-dear-looks-like-i-spelled-something-wrong the whereabouts of the cat . “ was tagged as DET N V P DET N P DET N ., which is mostly correct; however, the missing word was in this case tagged as a preposition, even though there aren’t any real prepositions that would seem to work there (but there are many verbs that might make more sense).

Another issue was with the limitations of the Markov assumption that only the previous state matters. The sentence “he had been whoops-another-spelling-mistake .” returned PRO V V . . The three recognized words were all correct, but the unrecognized word was incorrectly tagged as a period. The transition from verb to period makes sense (e.g. he ran . -> PN P .), but two verbs in a row are probably much less likely to be followed by a ., as this scenario typically indicates some sort of compound/helping verb situation; if the model had access to more temporal steps, maybe it could take into account the fact that V V -> . is much less likely than single V -> . is.

On the Brown test, our overall number of correct tags was 35109, and our number of erroneous tags was 1285, or an accuracy of 96.469% when we used an unobserved value of -100 (and natural logarithms). This is the same result as the PS-5 sample had, which is an encouraging sign that our program functions more-or-less entirely as expected.

Varying the unosberved value also behaved somewhat as we might expect. Large unobserved values (pretty much everything above -25) would have exactly the same effect—likely what’s happening is that such large unobserved values are immediately killing any paths that take them (i.e. if an unobserved observation is found, the probability of that path is basically set to 0), and the exact value of such large unobserved parameters is moot; point is, they’re large enough. These all result in the stock performance of 1285 erroneous tags on the Brown test sentences.

Very small unobserved values (-10 or less) produce extremely poor results as well, likely because they are too tolerant of unobserved words. -10 results in a whopping 4008 erroneous tags.. Increasing the unobserved value rapidly improves the accuracy of the algorithm; and unlike with a too-large (technically, too-small) unobserved value, there is a very different asymptotic behavior with low unobserved values. That is, the lower the unobserved value, the worse the program performs; it doesn’t quickly approach some worst performance (which is actually quite good) like the low (well, large magnitude but negative sign) unobserved values, but instead seems to continuously drop; even as the unobserved “penalty” becomes positive, the accuracy continues to get worse. At some point, the accuracy becomes so close to 0 that the program literally does worse than randomly guessing (U = 100000 gives an accuracy of 0.470%, while randomly guessing should be about 1/32, or 3.125%), which is honestly an amazing feat. However, a medium unobserved value actually improves performance somewhat compared to “stock” performance (U = -100). We found that values around -17 proved the best, with only 1281 erroneous tags; as the distance between U and -17 grows, this performance drops. Clearly, on the negative side, accuracy doesn’t get much worse; on the positive side, it becomes absolute garbage, as noted above.

Otherwise, there are generally many various sources of difference, even if the main algorithm is the same. For instance, there is the computation of the probabilities and their respective logarithmic “score” representations (e.g. if we used a different log base); as mentioned above, varying the unobserved value; and the way whitespace was handled (we split along all whitespace characters instead of just space); and so on.

In the simple test case, our overall number of correct tags was 32 and our number of wrong tags was 5, again identical to the results given by the sample.