Bin Dong
Professor Hwa
CS2731
5 March 2019

- How well did your various unsmoothed Ngram models perform on the English task? How high an N did you try? At what point did smoothing become a problem?
    - As expected, the Ngram models (smoothed version) didn't perform all that much better than the unsmoothed version when N is low. The reason for this is because everything that Ngram unsmooth predicts are already of the highest probability. This means that even after smoothing, the predicted letter would still be the highest.
    - An example of this would be: say that P(c | ab) = .8 and P(d | ab) = .2. Say that P(e | ab) = 0 and your vocab set is a,b,c,d, and e.
        - This means that in an unsmoothed version of Ngram, "c" has 80% probability and "d" has only 20%. The probability of "e" appearing is zero.
        - In an smoothed version, "c" will be a tiny bit under 80% and "d" will also be a tiny bit under 20%. That tiny bit difference is distributed to "e". This means that "e" no longer has a probability of zero as some of the weights are extracted from "c" and "d" to be given to "e". However, "e"'s probability is still VERY tiny (it is tiny but not zero). It won't make enough of a difference for the highest probability to be taken away from "c".
        - This means that if the unsmooth version predicts "c", then the chances of the smoothed version predicting "c" is also very very high (not 100%).
    - However, Ngram models in the smoothed version performs a lot better than Ngram model in the unsmoothed version. This is because you now have a longer history of characters. Having a longer history of characters ultimately leads to an more accurate result in the smoothed version. You can see the reasoning behind this in the next bulletpoint. Ngrams only outperforms noticeably better when N is high. Like stated before, it is hardly noticeable when N is low (say around 5). The comparisons are made below. (All the data are written to a file as well on GitHub)

Using N=9 With english/dev as test set	Unsmoothed Version	Smoothed
Total Correct	2462	3195
Total Total	5080	5080
Percentage	0.485%	0. 629%
Macro Avg. Precision	0.20	0.41
Macro Avg. Recall	0.24	0.36

Using N=9 With english/test as test set	Unsmoothed Version	Smoothed
Total Correct	3032	3950
Total Total	6400	6400
Percentage	0.474%	0.617%
Macro Avg. Precision	0.17	0.41
Macro Avg. Recall	0.23	0.37

Using N=9 on two data set, it is obvious that Smoothed version outperforms Unsmoothed version. It is also noticeable that Precision and Recall improves as well.

Using N=7 With english/dev as test set	Unsmoothed Version	Smoothed
Total Correct	2777	3111
Total Total	5080	5080
Percentage	0.547%	0.612%
Macro Avg. Precision	0.23	0.42
Macro Avg. Recall	0.27	0.36

Using N=7 With english/test as test set	Unsmoothed Version	Smoothed
Total Correct	3518	3868
Total Total	6400	6400
Percentage	0.550%	0.604%
Macro Avg. Precision	0.22	0.42
Macro Avg. Recall	0.24	0.35

With N=7, the accuracy is still extremely significant. However, it is also noticeable that the difference in accuracy and precision + recall has decreased when compared to N=9

Using N=5 With english/dev as test set	Unsmoothed Version	Smoothed
Total Correct	2820	2874

Total Total	5080	5080
Percentage	0.555%	0.566%
Macro Avg. Precision	0.30	0.42
Macro Avg. Recall	0.28	0.34

Using N=5 With english/test as test set	Unsmoothed Version	Smoothed
Total Correct	3528	3575
Total Total	6400	6400
Percentage	0.551%	0. 559%
Macro Avg. Precision	0.30	0.40
Macro Avg. Recall	0.25	0.30

When N=5, the unsmoothed version and the smoothed version basically has the same accuracy with smoothed version outperforming unsmoothed version ever-so-slightly.

Using N=3 With english/dev as test set	Unsmoothed Version	Smoothed
Total Correct	1836	1838
Total Total	5080	5080
Percentage	0. 3614%	0.3618%
Macro Avg. Precision	0.24	0.23
Macro Avg. Recall	0.15	0.14

Using N=3 With english/test as test set	Unsmoothed Version	Smoothed
Total Correct	2356	2332
Total Total	6400	6400
Percentage	0.368%	0. 364375%
Macro Avg. Precision	0.24	0.22

Macro Avg. Recall	0.15	0.13

At this point, smoothed and unsmoothed are essentially the same. Since Smoothed depends on the history of Bigram and Unigram, in the case where there are not any ties, the smoothed version may inaccurately predict the next character given that there are other factors. But as you can see, the margin of error is very small and the total accuracy are essentially the same down when N is small.

- What kind of smoothing method(s) did you implement? If there are some hyper-parameters for the smoothing methods, how did you choose the values for them? How well did these models perform?
    - I have implemented interpolation smoothing. Interpolation smoothing is a smoothing technique that takes into consideration all the rest of the Ngram models. What I mean by this is if N=3, you would have 3 models: Trigram, Bigram, and Unigram. If you have N=7, then you would have 7gram, 6gram, …, unigram.
        - Models List = Ngram, (N-1)gram, (N-2)gram, …, unigram.
    - Once you have all these N models, you can add the probabilities up.
        - For example, if N=3, and you are trying to find probability of P("c" | "ab") + P("c" | b) + P("c") + 1/ |v| where v= your vocab set.
    - However, since each of the grams have their own probability, you would want to normalize it. You can normalize it with weights (called lambdas or alphas).
        - The equation is: $\lambda_N \text{Ngram} + \lambda_{(N-1)}(\text{N-1})\text{gram} + \ldots + \lambda_{(1)}\text{Unigram} + \lambda_{(0)}(1/|v|)$
        - Where $\lambda_N + \lambda_{(N-1)} + \ldots + \lambda_{(0)} = 1$
    - In terms of picking lambdas, I believe that $\lambda_N > \lambda_{(N-1)} > \ldots > \lambda_{(1)}$. What I mean by this is this is that the weights for Ngram should be higher than the weights for (N-1)gram and the weights for (N-1)gram should be higher than the weights for (N-2)gram. This pattern repeats all the way till the end. Coming up with the lambdas that satisfy what I wanted was a bit tricky. However, thanks to an equation (basically softmax), I was able to come up with an equation that satisfies what I want. The equation is basically:
        - $N^N + N^{N-1} + \ldots + N^1 + 0.001 = \text{sum}$
        - For each of the term, divide it by sum (example would be N^N / sum).
    - This equation basically normalizes everything and once you divide everything by the sum and then add them all back up, your final result should be 1 (which is what you wanted since probability ranges from 0 to 1). The term index is essentially the weights for the Nth gram. For example, if N=3, then (N^N)/sum would be the weight for Trigram and (N^{N-1})/sum would be the weight for Bigram.
    - Notice that I have an constant of 0.001. This is because I believe that for the term (1/ |V|), the weights should be extremely small but not zero.
        - The reason why this term is important is because say that your model has never seen a particular vocab. This means that all your N gram models returned a probability of 0. Because you are smoothing, it doesn't make sense for the probability of that character to be zero and in order to solve that issue, we will pad it with (1/|v|). However, your vocab size can be extremely small… say 2 characters. This means that this padding (1/|v|) can be

extremely big. You want to ensure that it is small by setting the constant to be small.
    - I think by setting the weights high for higher order Ngrams results in a good prediction. As you can see from the data above, even when N is small, smoothing outperforms unsmoothing. in the case of unsmoothed, if there is a tie, then an arbitrary letter is selected from the characters that are tied. In the case of smoothed, if the Ngram is tied, then N-1 will be the tiebreaker. If N-1 is tied as well, then N-2 will be the tie breaker, all the way until the end. This means that for smoothing, the probability is more accurately reflected in all N models rather than just N model.
- What kind of an RNN model did you adapt for this task? Describe your architecture choices and how you tuned the hyper-parameters. How well did the model do?
    - I have attempted to implement RNN model using pytorch. I was able to see that average loss decreases however there seems to be a problem with either my implementation or the pytorch package. For higher learning rate and after a few iterations of training, loss value becomes NaN. I am not entirely sure why but if I were to tune the learning rate to be extremely small (around 0.00005), that problem goes away. However, having a smaller learning rate means that it will take a much longer time to train the model. Since I don't have infinite amount of time to do this project, I wasn't able to properly train the model. After countless hours of debugging, I decided to give up on trying to implement RNN and submit the project with step 1 and 2 completed.
- Overall, what seems to be the best working model for the English task? How well do you think your model would do on some other English corpus? Any other observations about this task?
    - If RNN was successfully implemented, I believe that RNN would perform much better than using a typical Ngram model. The reason for this is because RNN uses an arbitrary length history which beats out Ngrams which can only hold history up to size N.
- Describe the steps you took to adapt the models for the Chinese character prediction task.
    - Because I wasn't able to implement RNN, I wasn't able to adapt the RNN to predict Chinese characters. If I was able to implement RNN, I would attempt to adapt it by having a normal RNN model with Chinese characters as input. Instead of predicting character by character, I would predict the character only after feeding forward the entire word. Once I finish feeding the entire word and at the end of RNN, I would predict the final character.
        - To clarify, when using RNN for character prediction, you would make a prediction everytime you feed in a character. RNN would spit out a prediction of what it thinks the next character is.
        - However, when adapting over to Chinese RNN, I would essentially feedforward ALL the characters in the word and only make prediction once all the characters are fed forward. The prediction would be a Chinese character.
- Compare the models. How well did they do? Were there any surprises in the outcomes? Any other observations about this task?

- If I was able to successfully implement RNN that can predict Chinese characters, I think the model would be better than RNN. This is because for RNN and character prediction, I am predicting a character everytime I feedforward a character. This means that I am predicting while I am building up history. The longer the history, the more accurate it would be. Because Chinese RNN waits until you have a history before it predicts, that means that you have a full history of the word and all you are doing is predicting the character given a history. Having a history and predicting at the end should have a higher accuracy than predicting each word at every character.
- Though I also feel that they are about the same as well. Since RNN becomes more accurate the longer the history, predicting character after feeding forward a character will start off as inaccurate but overtime, it will get more accurate (because you are building history). In the end, they might both be the same.