Sujan Pantam - Assignment 2 Report
CSE 143: Intro to Natural Language Processing

1 Programming: n-gram language modeling

Model

For this first problem, I implemented unigram, bigram, and trigram language models to evaluate perplexity on a given dataset using Maximum Likelihood Estimation (MLE) without smoothing. I first tackled out-of-vocabulary (OOV) words as I replaced tokens that appeared fewer than three times with a special <UNK> token, resulting in a vocabulary of 26,602 unique tokens. I tokenized the sentences, adding <START> and <STOP> tokens, and computed the probabilities for unigrams, bigrams, and trigrams, storing these probabilities in dictionaries. To calculate perplexity, I calculated the probabilities of tokens in each sentence by getting the sentence length and the total number of tokens. Obviously bigrams and trigrams were a little different than unigram as they need to group words rather than individual words but that wasn't difficult at all.

Model	Formula
Unigram	P(w) = count(w) / N, where N is the total
Bigram	P(wi|wi−1)=count(wi−1)/count(wi−1,wi)
Trigram	P(wi|wi−2,wi−1)=count(wi−2,wi−1)count(wi−2,wi−1,wi)

Perplexity - PP(W)=P(w1,w2,...,wM)^(-1/M)

Performance

This takes O(n^2) time and overall was fine and did not take much time at all. Unigram was definitely faster than bigram and trigram, but that's to be expected.

Experimental Procedure

To begin with, I first made sure to get the right number of tokens. After this, it was pretty much smooth sailing from there as all I had to do was implement the formulas above with the data I was given. I tested on an experimental file that contained "HDTV ." as instructed to debug and check for the correct perplexity, and I got those results that matched with what should be and carried on with the actual sets.

Deliverables

1. Unigram Perplexities:
    - Train: 976.5437421980536
    - Dev: 892.2466475162531
    - Test: 896.4994914384728
2. Bigram Perplexities:
    - Train: 77.07346595679371
    - Dev: 28.29036069946888
    - Test: 28.31280883609587
3. Trigram Perplexities:
    - Train: 7.872967947059481
    - Dev: 2.787648276525293
    - Test: 2.7858397413047595

Results

Looking at the perplexities for the different models you can see that there is a decrease from Unigram to Trigram. This is good because lower perplexities mean it is better at predicting the input and the model isn't as surprised. This makes sense as in unigram we have individual words but for the others we have more data to be more prepared. I can say for sure that more grouping in n-gram modeling has an impact on the perplexity by lowering it.

2 Programming: additive smoothing

Model

This is pretty straightforward as it is pretty much like the first problem but all we have to account for is additive smoothing which only affects how MLE is done.

Performance

Overall, the performance remains the same as additive smoothing does not change anything drastically and will run in O(n^2) time.

Experimental Procedure

All I really had to do was get my working models from the first problem and just do a quick research on what is needed to account for in order to implement additive smoothing.

Deliverables

1. $\alpha = 1$
    - Unigram Perplexities: Train (972.6323766421456), Dev (890.2700616136049)
    - Bigram Perplexities: Train (66.71764821690606), Dev (27.256106180156916)
    - Trigram Perplexities: Train (6.515514785864113), Dev (2.7259312380890046)

2. α = 0.5
    - Unigram Perplexities: Train (495.28201984521763), Dev (453.2048840070817)
    - Bigram Perplexities: Train (39.51749382457388), Dev (17.538704113705734)
    - Trigram Perplexities: Train (4.9853855214215335), Dev (2.3335559150908507)
3. α = 0.7
    - Unigram Perplexities: Train (687.052983476496), Dev (628.7257928727287)
    - Bigram Perplexities: Train (51.05047748991127), Dev (21.72004647051738)
    - Trigram Perplexities: Train (5.689039316454876), Dev (2.516150319667118)
4. α = 0.6 for Test Set
    - Unigram Perplexity: 543.6054774892768
    - Bigram Perplexity: 19.634703133705231
    - Trigram Perplexity: 2.4283769644629696

Results

We can see that additive smoothing is helpful as it deals with zero probabilities that I had to deal with initially. Our performance seemed to improve as well as the perplexities were a bit lower than before which means it is working. Also, after experimenting with the alpha values, I determined that 0.6 was the best one especially when paired with the test set as requested to do so.

3 Programming: smoothing with linear interpolation

Model

Linear Interpolation Smoothing is implemented in this problem and what it does is that it combines unigram, bigram, and trigram models by assigning weighted probabilities to each, this in turn gives us a more robust estimation. In order to stay consistent, I used the perplexity calculations for trigram and applied the weights to it. Here is the formula for interpolation:

$$\theta'_{x_j|x_{j-2},x_{j-1}} = \lambda_1\theta_{x_j} + \lambda_2\theta_{x_j|x_{j-1}} + \lambda_3\theta_{x_j|x_{j-2},x_{j-1}}$$

Performance

The performance will still stay the same as we are only updating a previous model/formula by adding weights to the end of the log sum calculations. Therefore, it will still run in O(n^2) time.

Experimental Procedure

Again, this was pretty straightforward as I first went to dig deeper to understand the Linear Interpolation Smoothing and how I could/where I need to implement this in my code. After

updating the perplexity calculation for trigram with the weights, I tested if this was done right by using the experimental file with “HDTV .” with the lambda values of 0.1, 0.3, 0.6. If correct, then you would get a perplexity value of 48.1 which was what I ended up getting after debugging a couple times.

Deliverables

1. Recorded different sets of λ’s
   - Lambda Values (0.3, 0.6, 0.7): Train (8.075570729644166), Dev (3.072485889468533)
   - Lambda Values (0.1, 0.2, 0.4): Train (15.607340863847588), Dev (5.81155078750691)
   - Lambda Values (0.5, 0.8, 0.9): Train (6.198194732658697), Dev (2.363793512015611)
   - Lambda Values (0.1, 0.3, 0.6): Train (10.461560515255679), Dev (3.89351619685529)
2. Perplexity on Test Set
   - Lambda Values (0.3, 0.4, 0.5): 4.316163734711212
3. If you were to reduce the amount of training data, in this case half, this would most likely increase the perplexity as there could be lots of unseen data. Perplexity is based on how well a model predicts a sample and if there is less training data, there is going to be less exposure. This can lead to less accurate predictions and high perplexity values.
4. I converted all tokens that appeared less than 5 times to <UNK> and this is what I got:

   Train Perplexity of Unigram: 1103.4852487965237
   Dev Perplexity of Unigram: 954.2999354722378
   Test Perplexity of Unigram: 956.6888158668959

   When comparing these values to the perplexities I got from part 1, I found that the perplexity increases and this makes sense to me because we are losing more words/data and the model will make less accurate predictions as it will struggle to generalize. Lots of words will be turned to <UNK> which won’t be helpful at all.