Assignment 2 Writeup

Daniel Carrera, Jeremy Lin, Sanim Muktadir, Junchen Wang

Second Due Date: May 17, 2022

Note: For instructions on how to run the code and information on the programs' functionalities, please consult Instructions.md .

1 Programming: n-gram language modeling

To calculate perplexity, we first calculated the likelihood estimates for sentences in the Training set using the summation of the following log probabilities:

Unigram Probability:

$$\log_2\left(\frac{c(W_i)}{N}\right)$$

Bigram Probability:

$$\log_2\left(\frac{c(W_{i-1},W_i)}{c(W_{i-1})}\right)$$

Trigram Probability:

$$\log_2\left(\frac{c(W_{i-2},W_{i-1},W_i)}{c(W_{i-2},W_{i-1})}\right)$$

We then divided the likelihood estimates for all sentences in the test data by the total number of tokens within the test data M, and then raised 2 to the power of the opposite of the division result in order to get our perplexity.

1.1 Report the perplexity Scores of the unigram, bigram, and trigram language models

	unigram	**bigram**	**trigram**
training	976.544	77.073	7.592
development	892.247	28.290	2.909
test	896.499	28.313	2.907
HDTV .	658.045	63.708	39.479

Table 1: Table for 1.1

Looking at our experimental results, it is clear that the best performing language model was trigram, closely followed by the bigram language model. In contrast, the unigram language model proved to be the worst

by a significant margin of roughly 600-900 perplexity. These observations make sense because the unigram model is based on probabilistic independence of words, while the bigram and trigram models are based on the probabilistic dependence on groups of words. This means that higher n-gram degrees will extract more meaning from different groups and orderings of words as opposed to understanding single words.

2 Programming: additive smoothing

2.1 For additive smoothing with $\alpha = 1$

	unigram	**bigram**	**trigram**
training	977.508	1442.306	4486.094
development	894.390	194.391	19.680
test	898.556	194.274	19.780
HDTV .	620.881	959.965	19829.037

Table 2: Table for 2.1 $\alpha = 1$

2.2 Repeat for two other values of $\alpha > 0$ of your choosing

	unigram	**bigram**	**trigram**
training	979.925	2133.365	6401.605
development	897.659	269.548	23.583
test	901.761	269.406	23.704
HDTV .	593.687	1100.997	22912.778

Table 3: Table for 2.2 $\alpha = 2$

	unigram	**bigram**	**trigram**
training	992.182	3497.500	9065.444
development	911.364	418.539	29.037
test	915.351	418.353	29.189
HDTV .	543.310	1339.405	25163.831

Table 4: Table for 2.2 $\alpha = 5$

2.3 Report the perplexity scores for best values of the hyperparameters

	unigram	**bigram**	**trigram**
dev hyperparameters	894.390	194.391	19.680

Table 5: Table for 2.3

The best hyperparameter that minimized the perplexity values for the dev set was α = 1. A trend we observed was that as the value of α increased, so did the perplexity. The α value is used to assign a non-zero probability to unseen words, therefore improving the generalization of the model. Also with smoothing, we achieved many perplexity values that were greater than the size of the vocabulary, 26602, meaning it is worse than random guessing. Due to this, we are unsure about the accuracy of our data. We are unsure if assigning probabilities to ngrams that have not been seen before should cause a spike in perplexity like this. We also learned later that assigning a high alpha should cause the perplexities to tend toward the size of the vocabulary, providing more evidence that there was an error in our implementation because our data grew away from the size of the vocabulary as the alpha increased for bigram and trigram.

3 Programming: smoothing with linear interpolation

3.1 Report perplexity scores on training and development sets

	training	**development**
$\boldsymbol{\lambda_1 = 0.1, \lambda_2 = 0.3, \lambda_3 = 0.6}$	23.784	2.826
$\boldsymbol{\lambda_1 = 0.3, \lambda_2 = 0.3, \lambda_3 = 0.4}$	61.208	3.514
$\boldsymbol{\lambda_1 = 0.1, \lambda_2 = 0.1, \lambda_3 = 0.8}$	15.223	2.673
$\boldsymbol{\lambda_1 = 0.6, \lambda_2 = 0.3, \lambda_3 = 0.1}$	252.701	4.874
$\boldsymbol{\lambda_1 = 0.8, \lambda_2 = 0.1, \lambda_3 = 0.1}$	416.248	5.733
$\boldsymbol{\lambda_1 = 0.1, \lambda_2 = 0.8, \lambda_3 = 0.1}$	72.568	3.247

Table 6: Table for 3.1

3.2 Putting it all together, report perplexity on the test set

Perplexity on Test set: 2.679, using the following hyperparameters: $\lambda_1 = 0.1, \lambda_2 = 0.1, \lambda_3 = 0.8$. Smoothing with linear interpolation provided data that was more expected in comparison to the previously done additive smoothing.

3.3 If you use half of the training data, would it increase or decrease the perplexity

The perplexity would increase. This is probably because a larger training dataset can allow a model to understand the meaning of more words and combinations of words, therefore improving its ability to predict a sample. In contrast, less training data could cause under-fitting because it doesn't allow the model to pickup on enough words and groups of words to effectively predict a sample, leading to a less generalized model with high perplexity.

Here is empirical data with linear smoothing of $\lambda_1 = 0.1, \lambda_2 = 0.1, \lambda_3 = 0.8$, compared to the full data. As seen in the data below, a larger dataset produces lower perplexity numbers in development and testing.

	training	development	test
half	12.781	2.949	2.928
full	15.223	2.673	2.679

Table 7: Table for 3.3

3.4 If you convert all tokens that appeared less then 5 times to <unk>

The perplexities for the < 5 <unk> are higher than that of half the training data and that of the full training data. We think this is because words which appear up to 4 times compared to 2 times, might marginally contribute to the model's predictive performance. If these words are removed and marked as <unk>, their meaning becomes untapped by the model, decreasing the model's predictive performance.

Here is empirical data with linear smoothing of $\lambda_1 = 0.1, \lambda_2 = 0.1, \lambda_3 = 0.8$, compared to the half data.

	training	development	test
$\mathbf{< 5}$	15.935	3.660	3.660
half	12.781	2.949	2.928
full	15.223	2.673	2.679

Table 8: Table for 3.4