CSE 143 Assignment 1

Cameron Rabiyan
Rushil Nagabhushan
Zachary Jicha

January 26, 2020

Assignment Description:
In this assignment, we developed unigram, bigram, and trigram language models and calculated the perplexity scores. We then performed smoothing across these language models and compared the perplexity scores against the un-smoothed models.

1 Programming: n-gram language modeling

1.1 N-Gram Models and Procedures:

1.1.1 Building The Vocabulary and Preprocessing

Before we can use our N-Gram language model, we must build the vocabulary. We build the vocabulary set by first prepending <START> and appending <STOP> tokens to each sentence in the given data, measuring the number of appearances for all tokens. The total number of tokens in the training file is also recorded for use in the bigram model. This tokenization process converts all tokens that occur less than 3 times to an <UNK> token. In order to increase efficiency, we decided to place each of these tokens into a Python dictionary where the key is a token and the value is its number of appearances. We then preprocess the train, development, and test set by copying their contents into new files where words that do not appear in the vocabulary are replaced by <UNK>.

1.1.2 Unigram Model

After preprocessing the data and building a vocabulary, constructing a Unigram model is simple. The model just uses the vocabulary and corpus size that were made in the prior step. Using these, we can use the maximum likelihood estimation (MLE) equation for unigram models.

1.1.3 Bigram Model

The Bigram model is constructed by parsing the preprocessed training file. It turns sentences into bigrams and keeps a running count of the number of each bigram in a dictionary, similarly to the way the vocabulary did. Since the training file was already preprocessed so that any word appearing less than three times is replaced with <UNK> , out of vocabulary (OOV) words were already handled. The bigram model also contains a unigram model for efficiently computing the denominator of the MLE equation for bigrams.

1.1.4 Trigram Model

The Trigram model is constructed by parsing the preprocessed training file. It turns sentences into trigrams and keeps a running count of the number of each bigram in a dictionary, just like the bigram model. Also similarly to the bigram model, OOVs were handled in the preprocessing step. The trigram model also contains a bigram model for efficiently computing the denominator of the MLE equation for trigrams. The

bigram model is also used for computing the likelihood of the first token after <START>. Since this word is not a part of any trigram, its probability is calculated using the bigram model.

1.2 Calculating Perplexities

For every model described in this paper, the process for calculating the perplexity was the same. First, the file to be tested on would be parsed to be turned into sentences. Then each sentence would be parsed into the correct n-grams for the model being tested, i.e. bigrams for the Bigram model (also note that the trigram model has one bigram at the beginning of each sentence). The perplexity was then calculated one n-gram at a time using the fact that the log of the probability of a sentence is equal to the sum of the logs of the probabilities of its constituent n-grams. The results of calculating the perplexities of each n-gram model on the train, development, and test data sets are as follows (rounded to three decimal places):

N-gram Model Perplexity Scores

Data Set	Unigram	Bigram	Trigram
Train	976.544	77.073	7.873
Dev	892.247	∞	∞
Test	896.499	∞	∞

1.2.1 Analyzing Perplexities

Looking at the results of our N-Gram language model, it can be observed that the Bigram and Trigram models perform much better on the training data than the Unigram model. This makes sense as the more history we give the model, the better it should be able to predict future words. This is because any given word depends heavily on the words that came before it. However, these perplexities do not accurately represent the effectiveness of the language models. Since we trained the Bigram and Trigram language models on the training data, the models have essentially memorized the data, resulting in low perplexity scores. When looking at the Dev and Test data, the Bigram and Trigram language models are ineffective at predicting unseen n-grams. In order to combat this, we need a method of smoothing the language models to get a better perplexity score.

2 Programming: Smoothing

2.1 Linear Interpolation

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

By applying Linear Interpolation to the n-gram language models used above, we are able to *"smooth"* the language models. Without the smoothing process, each N-gram language model has its own strengths and weaknesses that can result in a wide range of perplexities. By applying a specific set of Hyperparameters to our linear interpolation formula, we are able to get a better perplexity across multiple language models.

2.1.1 Smoothed Model Experimental Procedures

The procedure for running the smoothed model is most similar to the trigram model. The file was parsed into trigrams (with a bigram at the beginning of each sentence) and each n-gram model within the smoothed model would extract its n-gram from the trigram. Perplexity was calculated in the same manner as the prior models.

2.1.2 Method for Choosing Hyperparameters

First of all, all tuning of hyperparameters was done on the development data set, and each model was then only run on the test data once. M_1 was chosen to see if the perplexity gains seen in Section 1's Trigram model could be applied to the development and test data sets. We hoped to do this by weighting the Trigram model heavily and giving the rest of the weight to the Unigram model so that the perplexity would not be infinity. M_2 and M_3 were chosen because we wanted to see if weighting the Unigram model heavily alongside either the Bigram or Trigram model would lead to perplexity gains without overfitting the data. M_4 was chosen because it was the most obvious choice to us and we were curious what the results would be. M_5 was chosen to see if a similar approach to M_0 would work when weighting the traigram model more heavily.

2.2 N-Gram Smoothed Model Results:

Smoothed Model Perplexities

Data Set	M_0	M_1	M_2	M_3	M_4	M_5
Train	11.151	8.660	307.374	54.785	17.039	10.306
Dev	352.234	1068.312	486.009	536.917	278.158	390.010
Test	351.007	1064.692	487.816	538.341	277.803	388.739

Model Hyperparameters

Model	λ_1	λ_2	λ_3
M_0	0.1	0.3	0.6
M_1	0.1	0	0.9
M_2	0.9	0.1	0
M_3	0.9	0	0.1
M_4	0.33	0.33	0.34
M_5	0.15	0.15	0.7

2.3 Hypothetical Experimentation

By analyzing the differences and changes across the set of results obtained by testing our models with varying input parameters and dataset specifications, we can get a deeper insight into how these variations affect our language models and ultimately, our output.

2.3.1 Half training data used only

Cutting the training data text file in half and processing it through our language models gives us lower, more accurate perplexity scores for the Unigram, Bigram, and Trigram models. For all sets of hyperparameters, the perplexities resulting from the change to the dataset were lower across the board for training, dev, and test data. For the M_0, the perplexities were 11.151, 352.234, 351.007, and upon applying the change to the data, the new perplexities are 10.332, 270.137, 269.028. For another example, if we take hyperparameter set M_4 (0.33, 0.33, 0.34) for instance: initially our perplexities on the smoothed model were 17.039, 278.158, 277.803, and once we apply the change to the dataset, our resulting perplexities are 15.549, 211.288, 210.755.

For Unigram modeling, since half of the training data is ignored, probabilities for known words will go up since the denominator of the Unigram MLE equation will be smaller, due to the smaller training data. Assuming unknown words are evenly distributed in the training data, their probabilities will remain around the same. For Bigram and Trigram modeling, the mechanics are similar. The denominator in the MLE equations will be smaller for known bigrams/trigrams, but this is probably not as prominent as the Unigram model and will lead to more probabilities of 0 in these models since even fewer bigrams and trigrams will be seen.

2.3.2 UNK threshold set to 5 instead of 3

Setting the UNK (unique token) threshold to 5 occurrences instead of 3 causes our Unigram and Bigram models to narrow down in perplexity and become more accurate, but this adversely affects the perplexity of our Trigram model, increasing it from 7.873 to 8.600. This is likely due to the decrease in overall frequency of unique tokens resulting from the UNK threshold change. There will always be more unique tokens in the data set for a language model with a greater UNK threshold and as the 'n' in your n-gram language model increases, the perplexity will too.

For the smoothed model, we found that this change in the UNK threshold resulted in a higher training data perplexities, but ultimately, lower and more accurate perplexities for the dev and test datasets. For example, our original smoothed model for M_4, containing hyperparameters (0.33, 0.33, 0.34) resulted in perplexities of 17.039, 278.158, 277.803. Upon increased the UNK threshold to 5 instead of the 3, the same smoothed model with the same hyperparameters resulted in perplexities of 18.206, 233.650, 233.381. The one exceptional instance was that for smoothed model with hyperparameters (0.9, 0.1, 0), the perplexities were lower across the board for training, dev, and test data: 283.260, 417.192, 418.709. As opposed to the original smoothed model: 307.374, 486.009, 487.816.

3 Observations and Conclusions

3.1 Computational Cost

One small observation is that while the Unigram, Bigram, and Trigram models are fairly efficient on their own, the smoothed models combining them are much less efficient than any one of these models on their own. While this is an obvious conclusion, it surprised us how much more time it took a smoothed model to calculate perplexities. Each of the Unigram, Bigram, and Trigram models took about 10 seconds to calculate perplexity on the training data. M_0, however, took about 50 seconds. This means that using three models can more than triple the time, most likely due to taking the log of the sum of the weighted probabilities. This would also be more exaggerated on larger data sets.

3.2 Minimizing Perplexity

From our initial testing in Section 1, one might come to the conclusion that the best n-gram model to use is the highest n feasible smoothed with a Unigram model just to get rid of 0 probability tokens. However, our experimental testing in Section 2 rebuts this. The model that fits this description, M_1, performed the worst out of all of our smoothed models and even performed worse than a Unigram model on its own. This is most likely due to the sparseness of the training data. Most trigrams will go unseen, and when an unseen trigram is encountered, the model relies entirely on the Unigram model, which is weighted to 10% of the smoothed model's probability. Also surprisingly, M_2 performed better than M_3, most likely for the same reason. The most surprising outcome to us was that M_4 performed the best out of all of the smoothed models we tested. This set of hyperparameters seemed the most obvious, and yet it also yielded the best perplexity scores. Further testing would need to be done on different test data to confirm that this is also true for data outside of our test set. If it does hold true for other data, this implies that all levels of n-gram smoothing play an important part in predicting tokens, not just the model with the highest n. This is may be because the most recent token is the most important for predicting the next token, although further inquiry is needed to confirm this.