# Unigram, Bigram, and Trigram Language Models with Smoothing
# CSE 143

Tyler Hoang - Assignment 1

January 26, 2020

This assignment asked for a unigram, bigram, and trigram language model to analyze the following datasets:
`1b_benchmark.train.tokens` - used for training the language models
`1b_benchmark.dev.tokens` - used for determining the hyperparameters for smoothing
`1b_benchmark.test.tokens` - used for evaluating the smoothing language model

With utilization of Python3, language model frequencies were kept in several dictionaries. Specific mathematical operations were performed in order to obtain perplexity scores for each respective language model, and smoothing hyperparameters were found via a simple grid-search.

## 1 Perplexity Scores

Each data set was processed for their unigram, bigram, and trigram perplexity scores. The following is the procedure that was performed on the training, development, and test sets.

### 1.1 Out Of Vocabulary Correction

By running `Training_Token_Initialization.py`, the training data set is read line by line in order to calculate the frequency of each unique token of the training set. This data is stored in a dictionary and outputted to the file `Training_Token_Data.txt`. The value found at the top of the output file is the total number of tokens in the training set, of course excluding `<START>` tokens since they are not included the vocabulary $\boldsymbol{V}$. With the `<STOP>` tokens included, there are a total of 26602 unique tokens.

After the training data has been analyzed, the data is scanned a second time to find words that have a frequency of **less than 3**, which are subsequently replaced by `<UNK>` tokens. The frequency dictionary is updated accordingly, then the development and test sets are scanned for words that are Out Of Vocabulary. Replacing OOV words with `<UNK>` tokens allows the training data frequencies to be used on the development and test sets.

### 1.2 Bigram and Trigram Frequencies

Along with the token frequencies, the bigram and trigram frequencies in the training data are also analyzed after `<UNK>` token insertion is complete. `<START>` tokens are prepended to the beginning of each sentence and `<STOP>` tokens are appended to the end of each sentence. By traversing

through each sentence, bigrams and trigrams are obtained and stored in dictionaries along with their respective frequencies.

The `Training_Bigram_Data.txt` and `Training_Trigram_Data.txt` files, which are outputted from `Training_Token_Initialization.py`, contain the frequencies of the bigrams and trigrams for the training data set.

### 1.3 Perplexity Calculation

The `UBTCalc.py` program reads in the frequency data from the training data set and outputs the perplexity scores for the desired data set. These values are outputted to the `UBT.txt` file. Unigram perplexities are calculated with the following procedure:

1. The relative frequency estimate of a single word $v$ is

$$(\text{frequency of } v)/N),$$

   where $N$ is the total number of tokens in the training set.

2. The negative log probability of every token probability is used in order to prevent an underflow from occurring. Sentences with a large amount of tokens yield an extremely tiny probability that would be rounded down to 0, essentially breaking the calculation. Negative log unigram probabilities of each sentence are summated to give us $-\log_2 p(\overline{\boldsymbol{x}}_i)$.

3. Every sentence probability is calculated, then summated.

4. This value is then divided by the total number of tokens $\boldsymbol{M}$ in the data set.

5. Taking 2 to the power of this number gives us the unigram perplexity of the data set.

For the bigram and trigram language models, both the `<START>` and `<STOP>` tokens have significance, whereas for the unigram language model, only the `<STOP>` tokens held significance. Even though `<START>` tokens are not a part of $\boldsymbol{V}$, they are necessary for context.
For the bigram model, a bigram's probability is

$$(\text{frequency of } v_{j-1}v_j)/(\text{frequency of } v_{j-1}),$$

where $(v_{j-1}, v_j)$ is a bigram, and $(v_{j-1})$ is the first unigram (the first word) of the bigram. Negative log bigram probabilities of each sentence are summated to give us $-\log_2 p(\overline{\boldsymbol{x}}_i)$.
For the trigram model, a trigram's probability is

$$(\text{frequency of } v_{j-2}v_{j-1}v_j)/(\text{frequency of } v_{j-2}v_{j-1}),$$

where $(v_{j-2}v_{j-1}, v_j)$ is a trigram, and $(v_{j-2}v_{j-1})$ is the first bigram in the trigram (the first pair of words). Negative log trigram probabilities of each sentence are summated to give us $-\log_2 p(\overline{\boldsymbol{x}}_i)$.

### 1.4 Results

Below are the perplexity scores attained for the **training** data set:

| Language Model | Perplexity Score |
|---|---|
| `Unigram` | 976.54374222006748685154064 08 |
| `Bigram` | 77.07346595628909029057462 4416 |
| `Trigram` | 6.82324604590766123723325 6129 |

Below are the perplexity scores attained for the **dev** data set:

| Language Model | Perplexity Score |
|---|---|
| `Unigram` | 892.24664751228860165996300 24 |
| `Bigram` | $\infty$ |
| `Trigram` | $\infty$ |

Below are the perplexity scores attained for the **test** data set:

| Language Model | Perplexity Score |
|---|---|
| `Unigram` | 896.49949143435562529164545 48 |
| `Bigram` | $\infty$ |
| `Trigram` | $\infty$ |

Bigram and trigram perplexity scores for the dev and test data sets resulted in $\infty$ due to the fact that for both data sets, there existed at least 1 bigram that was nonexistent in the training data. This bigram has a probability of 0, and when 0 is inserted into the perplexity equation, we yield the expression $\log_2(0)$. The IEEE floating point convention for $\log_2(0)$ is $-\infty$, which results in a perplexity score of $\infty$. It is also true that if a bigram of probability 0 is found in a data set, then there exists a trigram of probability 0 (hence the perplexity score for the trigram language model will also be $\infty$).

# 2 Smoothing

In order to fix the issue with infinite perplexity scores, we can input smoothing hyperparameters in order to remove the possiblity of there being an element of probability 0. Smoothing essentially allows us to combine all 3 language models into 1, and in doing so we must attempt to find the best hyperparameters in order to achieve the lowest perplexity score possible. Each probability to be summated for each sentence probability consists of a sum of 3 components:

$$\lambda_1\theta_{x_j} + \lambda_2\theta_{x_j|x_{j-1}} + \lambda_3\theta_{x_j|x_{j-2}x_{j-1}}$$

Since there will always be a unigram probability involved in this calculation, and assuming that $\lambda_1$ will never be 0, the final probability will always be a positive number. This prevents the possibility of $\log_2(0)$ occurring.

## 2.1 Smoothing Results

Utilization of the program `Smoothing.py` and a simple grid-search yields the following results:

| Weights ($\lambda_1$,$\lambda_2$,$\lambda_3$) | Perplexity Score |
|---|---|
| `0.8, 0.1, 0.1` | 373.67586981436725966067918 45 |
| `0.33, 0.33, 0.34` | 278.15805016792462583744942 09 |
| `0.1, 0.1, 0.8` | 482.84637619561474545260181 54 |
| `0.1, 0.8, 0.1` | 292.67752570990240437330657 66 |
| `0.2, 0.4, 0.4` | 286.36151780531301588921385 35 |
| `0.1, 0.3, 0.6` | 352.23418451893950923899102 79 |

It seems that a linear interpolated model that is more evenly distributed, in terms of language model weights, yields a lower perplexity score. Using the weights:

$$\lambda_1 = 0.33, \lambda_2 = 0.33, \lambda_3 = 0.34$$

on the **test** data set yields a perplexity score of 277.80257857875334166460676 63.

## 3 Final Remarks

Let's say we use half of the training data on the previously unseen data. This would yield a much higher perplexity score due to the fact that there will be less frequency data available. There will be more OOV words to be replaced by `<UNK>` tokens along with less bigrams and trigrams. The smoothed parameters would rely more on the unigram probabilities, yielding a much smaller $-\log_2 p(\overline{\boldsymbol{x}}_i)$ value. Reviewing the results, it is obvious that the language model weighted more towards unigram probabilities produced one of the larger perplexity scores.

If the frequency requirement changed from **less than 3** to **less than 5**, the following results occur on the **training** data set:

| Language Model | Perplexity Score |
|---|---|
| `Unigram` | 803.48524880594081129859926 77 |
| `Bigram` | 75.63322687111259853693276286 |
| `Trigram` | 7.42811299412113061068811961 6 |

The unigram perplexity scores for the dev and test data set are also reduced due to the fact that the training frequency data will contain words with a higher overall frequency. This does, however, reduce the amount of words in the language model's vocabulary. Conversely if we were to reduce the frequency requirement, the perplexity scores would increase since the training frequency data will contain words with a lower overall frequency. The vocabulary of the language model will increase because of this.

I believe that using a bigram model is the best out of the three language models used for this assignment. $n$-grams that are too large make it more difficult to pick up on words that occur more frequently than others, whereas $n$-grams that are too small make it so that every word is analyzed individually, with no regards to context. Even though perplexity scores decrease as $n$-grams increase, larger $n$-gram language models don't exactly equal a "better" language model.

The resulting perplexity scores for the linear interpolated language model were quite surprising for me. I initially assumed that a language model weighted more towards trigram probabilities would yield better perplexity scores. However, with the following weights

$$\lambda_1 = 0.1, \lambda_2 = 0.1, \lambda_3 = 0.8$$

the model produced the worst perplexity score out of all of the tested weights. In all honesty, I am still not entirely sure why this was the case, but clearly when this language model has weights that are more evenly distributed, it produces better perplexity scores.