CS6320 Assignment 1

https://github.com/pritika292/NLP-Assignment-1

Group07 Pritika Priyadarshini PXP210104 Gauri Sagane GXS220013

1 Implementation Details

Pre-processing of data -

1. **Convert all text to lowercase** - This improves consistency of the words in the dataset. The goal is to remove any potential inconsistencies that arise from using mixed case words.
2. **Standardize punctuations to a single format** - It ensures that text data is consistent and hence easier to compare and process.
3. **Clean up wild characters** - This cleaning step ensures that unwanted symbols such as ?, :, # etc. are removed since they do not hold much value in the model learning.
4. **Remove Stop words** - This is done mainly to reduce the noise in training as stop words appear frequently in almost all text documents and do not carry much information by themselves.
5. **Lemmatization (Normalization)** - Reducing words to their core meaning helps reduce vocabulary size, adds more meaningful feature vectors, hence capturing similar contexts.
6. **Add END tag for every sentence** - This is done so that ngrams have the context for the end of a sentence i.e which words are more likely to be the end of a sentence.

1.1 Unigram and bigram probability computation

The training data is preprocessed and tokenized as described above. Next, we store the **frequencies/counts** of unigrams and bigrams in a **hashmap data structure**. For unsmoothed ngrams, we calculate the ngram probability using the below relation :

$$P_{unigram}(W) = \frac{C(W)}{\text{Total number of words in train set}}$$

$$P_{bigram}(W|V) = \frac{C(V,W)}{C(V)}$$

```
def get_bigram_probability(unigram_counts, bigram_counts, output_file_path):
    bigram_prob = {}
    list_of_bigrams = bigram_counts.keys()
    with open(output_file_path, 'w') as write_handle:
        for bigram in list_of_bigrams:
            bigram_prob[bigram] = (bigram_counts[bigram])/(unigram_counts[bigram[0]])
            write_handle.write("P{} = {}".format(bigram, bigram_prob[bigram]))
            write_handle.write("\n")
    return bigram_prob

def get_unigram_probability(unigram_counts, total_train_words, output_file_path):
    unigram_prob = {}
    listOfUnigrams = unigram_counts.keys()
    with open(output_file_path, 'w') as write_handle:
        for unigram in listOfUnigrams:
            unigram_prob[unigram] = (unigram_counts[unigram])/total_train_words
            write_handle.write("P({}) = {}".format(unigram, unigram_prob[unigram]))
            write_handle.write("\n")
    return unigram_prob
```

Figure 1: Calculate unsmoothed probabilities.

1.2 Unknown word handling

First, we handle unknown words (words that have never appeared in the training corpus) in the following manner
1. We create a **special token "<UNK>"** for out-of-vocabulary words. Since we do not have a given fixed vocabulary, we create our own vocabulary based on the training data.
2. We set a **threshold value k** such that words in the training data which occur less than or equal to k times are not taken into the vocabulary.
3. Any words in the training set not found in V are replaced with "<UNK>" and their probabilities are learnt like any other word.
4. While parsing the test set, the words which are not found in the vocabulary are considered as "<UNK>" and their probability is assigned as the probability of "<UNK>".

1.3 Smoothing

Next, we handle the sparsity caused by unseen n-grams by various smoothing techniques. By Smoothing, we steal probability mass from more frequently occurring ngrams and take them into consideration for unseen or new ngrams to generalize better.

Laplace smoothing - We have implemented add-k laplace smoothing for both the unigram and bigram models.The relation is given as follows -

$$P_{Add-k}(X_i|X_i-1) = \frac{C(X_i-1, X_i)+k}{C(X_i-1)+kV}$$

1-Laplace smoothing treats all unseen n-grams as equally likely thus inflating the probabilities of rare events excessively. By using a smaller k-value, we expect better results. In general, laplace smoothing is not suitable for ngram models, so we explore further smoothing methods.

Smoothing by linear interpolation - Using simple interpolation, we compute the probability of a bigram by taking a linear combination of the bigram and unigram probabilities according to the below relation (for bigram model) -

$$P_\lambda(w|v) = \lambda_2 P_M LE(w|v) + \lambda_1 PMLE(w)$$

The sum of λ_1 and λ_2 should be 1. Typically, we give higher weights to for bigram models as they tend to be more accurate as compared to the unigram model.

```
def get_perplexity_bigram_interpolation(bigrams_train, unigrams_train, test_file_path, prob_file_path, total_train_words, lambda_weight):
    total_words = 0
    bigramProb = get_bigram_probability(unigrams_train, bigrams_train, prob_file_path)

    with open(test_file_path) as read_handle:
        lines = read_handle.readlines()
        sum_log = 0

        for line in lines:
            words = line.split(',')
            total_words += len(words)
            for i in range(len(words)-1):
                p_bigram = 0
                if (words[i], words[i+1]) in bigramProb:
                    p_bigram = lambda_weight * bigramProb[(words[i], words[i+1])]
                p_unigram = 0
                if words[i+1] in unigrams_train:
                    p_unigram =  (1-lambda_weight) * (unigrams_train[words[i+1]])/total_train_words
                p = p_unigram + p_bigram
                sum_log += np.log(p)

        sum_log = sum_log/total_words
        print("Perplexity for bigram : ", np.exp2(-1*sum_log))
```

Figure 2: Smoothing by interpolation

1.4 Implementation of perplexity

We evaluate the language models using perplexity which tells us how well our probability model predicts a sample text. Ideally, we should compare the log probabilities in order to judge which sentences were predicted well by the language model, but since we multiply probabilities, the longer the sentence the lower its probability. In order to remove influence of sentence length, we normalize the probability by the number of words in the sentence.

$$Perplexity = exp\frac{1}{N}\sum_{i=1}^{N} -log(P(W_i|W_{i1}...W_{in+1})$$

```python
def get_perplexity_unigram(unigrams_train, total_train_words, test_file_path, prob_file_path):
    total_words = 0
    unigramProb = get_unigram_probability(unigrams_train, total_train_words, prob_file_path)
    with open(test_file_path) as read_handle:
        lines = read_handle.readlines()
        sum_log = 0

        for line in lines:
            words = line.split(',')
            total_words += len(words)
            for word in words:
                pword = 0
                if word in unigramProb:
                    pword = unigramProb[word]
                sum_log += np.log(pword)

        sum_log = sum_log/total_words
        print("Perplexity for unigram : ", np.exp2(-1*sum_log))

def get_perplexity_bigram(bigrams_train, unigrams_train, test_file_path, prob_file_path):
    total_words = 0
    bigramProb = get_bigram_probability(unigrams_train, bigrams_train, prob_file_path)

    with open(test_file_path) as read_handle:
        lines = read_handle.readlines()
        sum_log = 0

        for line in lines:
            words = line.split(',')
            total_words += len(words)
            for i in range(len(words)-1):
                pbigram = 0
                if (words[i], words[i+1]) in bigramProb:
                    pbigram = bigramProb[(words[i], words[i+1])]
                sum_log += np.log(pbigram)

        sum_log = sum_log/total_words
        print("Perplexity for bigram : ", np.exp2(-1*sum_log))
```

Figure 3: Perplexity for unsmoothed ngrams

2 Eval, Analysis and Findings

2.1 Evaluation and Analysis

After pre-processing and tokenizing the training and validation dataset are written to *tokenized_train.txt* and *tokenized_val.txt*

A1_DATASET > tokenized_train.txt

```
4942  returned,room,5pm,find,dried,blood,still,floor,[END]
4943  making,complaint,customer,service,gotten,reply,hotel,[END]
4944  considering,price,point,downtown,hotel,would,think,would,try,live,5,star,rating,instead,failed,miserably,[END]
4945  luckily,many,hotel,choose,downtown,chicago,[END]
```

The unsmoothed probabilities can be found in the files *unsmoothed_bigram_prob.txt* and *unsmoothed_unigram_prob.txt*. The perplexity for **unsmoothed** unigram and bigram models on the validation set would be ***infinity***, since for unknown or new words in test set the probability would be zero which in turn results in **perplexity to be infinite**.

```python
unigram_prob_file_path = os.path.join(os.getcwd(), "A1_DATASET/unsmoothed_unigram_prob.txt")
bigram_prob_file_path = os.path.join(os.getcwd(), "A1_DATASET/unsmoothed_bigram_prob.txt")
get_perplexity_unigram(unigrams_train, total_train_words, tokenized_val_file_path, unigram_prob_file_path)
get_perplexity_bigram(bigrams_train, unigrams_train, tokenized_val_file_path, bigram_prob_file_path)
```

```
Perplexity for unigram :  inf
Perplexity for bigram :  inf
```

Figure 4: Unsmoothed Perplexities for Unigram and Bigram Model.

To resolve the above problem we handled unknown words where we replace words appearing less than a fixed frequency with <**UNK**>, and while calculating the probability for new words in the test set, we would consider probability for <**UNK**>. The perplexity after handling **unknown words** in the test set is as follows:

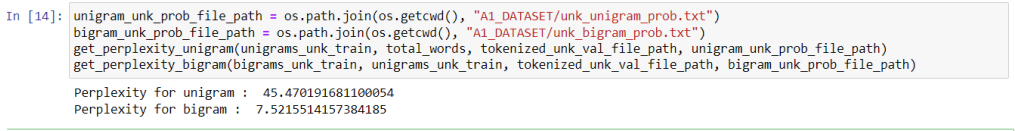

Figure 5: Perplexities after handling UNK for Unigram and Bigram Model.

We next handle the zero probabilities due to unseen ngrams using **Add-1 (Laplace) Smoothing** technique which adds one to the ngram counts before normalizing probabilities. Hence, ngrams appearing in an unknown will not have zero probabilities and we are able to calculate the perplexity. The validation set **perplexities after 1-Laplace Smoothing** are as follows:

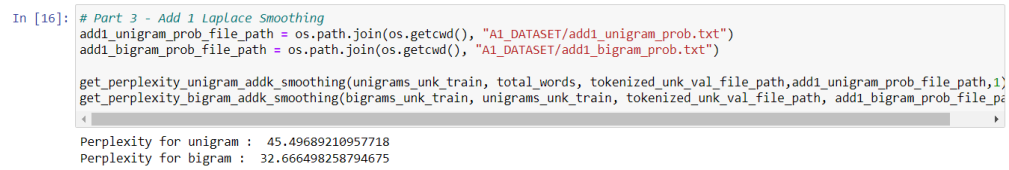

Figure 6: Perplexities with Add-1 Smoothing for Unigram and Bigram Model.

The Add-1 Smoothing adds one to all the counts irrespective of the occurrence of the words. However, we can enhance our results by using **Add-k Smoothing** technique with a fractional k i.e (k = 0.1, 0.05, etc). To get the best k for our model, we tuned it using grid search on a held-out set (20% of train). The best value for k for which the perplexity was minimum was **k = 0.05**. The validation set **perplexity for unigram and bigram models** are as follows:

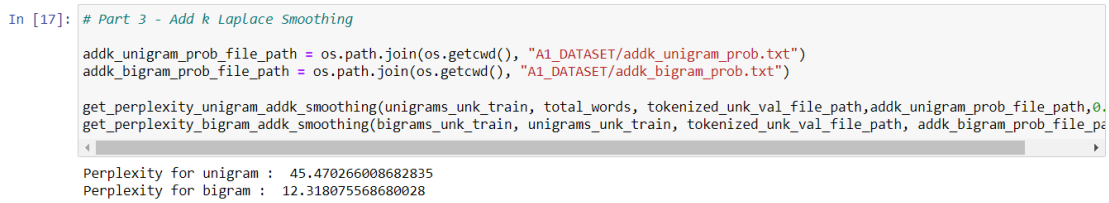

Figure 7: Perplexities with Add-k Smoothing for Unigram and Bigram Model.

Next, we implemented **Linear Interpolation Smoothing** method in which we can leverage additional source of knowledge in n-grams. Linear interpolation uses a **mix of lower ngram probabilities** as well, for computing probablities for higher ngrams. We tuned the hyperparameters on held out set (20% of train) and got the best values for λ_1 = **0.05** (for unigram) and λ_2 = **0.95** (for bigram). The validation set perplexity with linear interpolation smoothing is as follows:

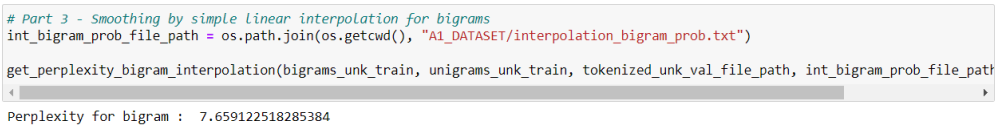

Figure 8: Perplexities with Linear Interpolation Smoothing for Bigram Model.

2.2 Findings

We can conclude from our analysis above that the perplexity for unigram and bigram models without any smoothing were **infinite** because unknown words in the test data have zero probabilities. After performing various smoothing techniques the results were significantly improved for both the models. **Add-1 and Add-k smoothing** don't improve model perplexity on validation set since they are not good for ngram models. The best results were obtained after performing **unknown word handling** and smoothing using **Linear Interpolation** technique. This is because Linear Interpolation draws probability estimated using both bigrams and unigrams. It gives more weight to bigram probabilities since they are a better indicator of context. To summarize the above findings, the overall results for perplexities of unigram and bigram models are as follows:

	Unigram PPL	Bigram PPL
Unsmoothed	inf	inf
Add-1 Smoothing	45.49	32.66
Add-k Smoothing	45.47	12.31
Linear Interpolation Smoothing	NA	7.65

Table 1: Validation set perplexities for Unigram and Bigram Models.

3 Others

3.1 Details of programming library usage

1. **Natural language toolkit (Nltk)** for various preprocessing tasks such as word and sentence tokenization, lemmatization and removal of stop words.
2. **Numpy** for mathematical computations such as log and exp since it is highly optimized to improve speed and provides scientifically accurate results.

```
python -m pip install nltk==3.5 numpy
python -m nltk.downloader all
```

3.2 Brief description of the contributions of each group member

Pritika - Researching on additional preprocessing choices on text data and tokenizing. Implementing unigram and bigram counts for all tokens and evaluate perplexity computation for different versions of the models. Unknown word handling implementation by building a vocabulary and assigning a minimum threshold. Added basic smoothing (Add-1 laplace) for ngrams.

Gauri - Reading and implementing smoothing methods such as add-k smoothing, their effect on ngram models. Experimenting with better smoothing methods - simple linear interpolation to get better results on the bigram model. Using held-out set to tune hyperparameters involved such as k (add-k), λ_1 and λ_2 (interpolation). Validated the pre-processing methods employed.

Combined work on the report to illustrate our findings and understanding of the problems solved in this assignment

3.3 Feedback for the project

This assignment provided us with the opportunity to gain a deeper understanding of ngram models and their performance with different smoothing techniques. It was of medium difficulty - we built lower order ngram models (unigrams and bigrams) which are relatively easier to implement, research and experimentation on various smoothing and unknown word handling methods added to the challenge. We took almost two weeks gain in depth knowledge and implement this assignment. This was a great way of applying theoretical knowledge learnt in class to real world data.