Group 31 Names and NetIDs: **Samkit Saraf (SXS220101)** and **Samarth Sairam (SXS200378)**

NLP – Assignment – 1 Group 31 – Report

Github Link: samkit09/NLP-Unigram-and-Bigram_Language_Models (github.com)

1. **Clear description of implementation details (including the most important code pieces) in the report (60%).**

We have divided this assignment into 3 sections namely computing unigram and bigram probabilities without smoothing, applying smoothing, and evaluating the language models by intrinsic evaluation.

First, the dataset is imported as a string and stored in a variable. Then preprocessing such as adding start tag '<s>' at the start of every sentence as well as adding ending tag '<e>'. Apart from that some characters like "\n", "-", '/', "..", "*" were replaced with white-space characters, and shortened words like " n't ", " 're ", " 'll " were replaced by their full form like "not", "are", "will", etc. We also removed words with no length of less than 1, basically, empty characters or white-space characters. This was done in order to make the context clearer.

```
def pre_process (ds):
    # defining character to replace
    special_chars = ["\n", "-", '/', "..", '.', "*"]
    sent_ends = [" . ", " ? ", " ! "]
    rep1 = " "
    rep2 = " . <e> <s> "
    #  replacing special character and escape sequence characer with white space character
    for s in special_chars:
        ds = ds.replace(s, rep1)
    #  adding sentence start '<s>' and end '<e>' tags at the start and end of each sentence
    for s in sent_ends:
        ds = ds.replace(s, rep2)

    ds = ds.replace("'re ", " are")
    ds = ds.replace("n't ", " not")
    ds = ds.replace("'t ", " not")
    ds = ds.replace("'m ", " am")
    ds = ds.replace("'d ", " would")
    ds = ds.replace("'ve ", " have")
    ds = ds.replace("'ll ", " will")
    ds = ds.replace("'", " ' ")

    ds = "<s> " + ds
    ds += " <e> "

    # lowering all the character to lowercase so the model don't take words like 'HELLO' and 'hel
    ds = ds.lower()

    return ds
```

Fig 1: Preprocessing

A. Unsmooth unigram and bigram probability computation (15%)

For computing unigram probabilities, we first counted the occurrences of words.

For computing unigram probabilities, we first counted the occurrences of words and then we divided the counts by the total number of words in the training set. All these calculations are stored in a dictionary with words as keys and probability as values.

Similarly, for bigrams, we calculated probabilities by storing the count of words coming after a certain token in the test set. Its structure is like {"token" : {words1 : count1, words2 : count2} }.

Then the counts are divided by the count of that token from the unigram counts.

B. Smoothing (15%)

Since there exist words in the test set that might not have appeared in the training set. To deal with that we have implemented Laplace smoothing and Add-k smoothing with the values ["0.005", "0.05", 2, 5].

In Laplace smoothing, 1 is added to the count of words that were stored previously. This will be useful to calculate the probability of unseen words. Whereas, in add-k smoothing, instead of adding 1 we added 'k' to all the word counts. In general, it lowers the probability of all the words slightly, but not always.

```
def get_prob_after_smoothing_bi(counts, unigram, K = 1):
    '''
    counts: key value pairs with words as keys and count as values. Used to calcu
    '''
    _prob = defaultdict(lambda:defaultdict(lambda:1/len(vocab)))
    for k,v in counts.items():
        n = unigram[k]
        temp = defaultdict(lambda:(K/(n + K*len(vocab_unk))))
        for k1,v1 in v.items():
#           try:
               # Formula for add-k smoothing
            temp[k1] = (v1+K)/(n + K * len(vocab_unk))
#           except:
#               temp[k1] = 0
        _prob[k] = temp

    return _prob
```

Fig2: Add k smoothing bigram

```
[ ]  def get_prob_after_smoothing_uni(counts, K = 1):
        N = sum(list((counts.values())))
        _prob = defaultdict(lambda:(1+K)/(N+len(vocab_unk)*K))
        for k,v in counts.items():
            # Formula for add-k smoothing

            _prob[k] = (v + K) / (N + (K*len(vocab_unk)))
        return _prob
```

Fig 3:- Add k smoothing unigram

```
[ ] # calculating probabilities with and without unknown word handling after
    uni_gram_prob = get_prob_after_smoothing_uni(unigram)
    bi_gram_prob = get_prob_after_smoothing_bi(bi_gram, unigram)
    uni_gram_prob_unk = get_prob_after_smoothing_uni(unigram_unk)
    bi_gram_prob_unk = get_prob_after_smoothing_bi(bi_gram_unk, unigram_unk)

[ ] uni_gram_prob['<unk>']

    1.1221581345243171e-05

[ ] uni_gram_prob_unk['<unk>']

    0.03429315259106313

[ ] bi_gram_prob['booked']['<unk>']

    0.00031377470975839345

[ ] bi_gram_prob_unk['booked']['<unk>']

    0.0003136762860727729
```

Fig 4: Laplace Transforms

C. Unknown word handling (15%)

We applied the no prior vocab solution for handling unknown words. For this, we replaced the words based on their frequency. If a word appeared less than 3 times, then we replaced it with the '<unk>' tag.

This helped us to deal with spelling errors or mistyped words in the training set.

For the testing set, if the word does not appear in the filtered word list. Then we are replacing that with unknown words.

```
# function to handle unknown data
def handle_unigram_unk(unigram, n = 2):
#       print(unigram)
      cp_unk = []
      for k,v in dict(unigram).items():
          if v < n:
              print(k,v)
              cp_unk.append(k)
      return cp_unk
```

```
# handeling unknown words by repllacing them with <unk> tag
words_unk = []
for word in words:
    if word in corpus_unk:
        words_unk.append("<unk>")
    else:
        words_unk.append(word)
```

D. Implementation of perplexity (15%)

We get the Sentence log sum of unigram/bigram, and we can find the length of the corpus as N.

$$Perplexity = 2^{**} -l/N \text{ , where } l = log\ sum\ of\ a\ sentence.$$

Note: When we try to calculate the perplexity of bigram on unsmoothed data, it throws an error as the probabilities of unseen data are not present.

2. Eval, Analysis, and Findings – including but may not be limited to:

(1) report training and validation set perplexity:

```
def unigram_perplexity(sentences, probabilities):
    log_sum = 0
    N = get_number_of_unigrams(sentences)
    words = sentences.split(' ')
    for word in words:
      # print(word)
      if word in probabilities:
        try:
          log_sum += math.log2(probabilities[word])
        except:
          log_sum += math.log2(probabilities['<UNK>'])

    return math.pow(2, (-1 * (float(log_sum) / N)))


def bigram_perplexity(sentences, probabilities):
    N = get_number_of_unigrams(sentences)
    log_sum = 0
    words = sentences.split(' ')
    for i in range(1, len(words)):
        prev_word = words[i - 1]
        cur_word = words[i]
        if prev_word in probabilities:
          if cur_word in probabilities[prev_word]:
            try:
              log_sum += math.log2(probabilities[prev_word][cur_word])
            except:
              log_sum += math.log2(probabilities[prev_word]['<UNK>'])
        else:
          if cur_word in probabilities['<UNK>']:
            try:
              log_sum += math.log2(probabilities['<UNK>'][cur_word])
            except:
              log_sum += math.log2(probabilities['<UNK>']['<UNK>'])


    return math.pow(2, (-1 * (float(log_sum) / N)))
```

Fig:- Perplexities

We got better perplexities in the bigram model (6.734105387492853) than in the unigram model (239.31429416264035). Also, we got less perplexity for smoothed data than unsmooth data. Lower the value of k, lower the perplexity, and better the model.

(2) how the smoothing strategy affects the performance of the validation set (30%):

The effectiveness of a smoothing approach on a validation set is influenced by several factors, such as the type of data, the smoothing method employed, and the underlying model or algorithm. Typically, smoothing is used in the context of probabilistic models. In the context of the model, we have made use of Laplace smoothing and Add-k smoothing.

1. Laplace smoothing: The fundamental principle of Laplace smoothing is to increase each event's count in the numerator by a small constant value, often 1, and to change the

denominator to reflect the increased counts. This guarantees that no event has a probability of zero and keeps the probability estimate from falling to zero.

2. Add-K smoothing: You can modify the smoothing parameter (k) with add-k smoothing to alter the level of smoothing.

• Details of programming library usage if any (6%)

We tried to create the program from the basics, that's why we did not use many third-party libraries as that would defeat the purpose of understanding and learning the concept.

We used the following libraries:

1. from collections import defaultdict: Default dictionary was used to store unigram and bigram probabilities as this data structure provides the fastest access time that will help in storing values and calculating perplexity in the fastest way possible.
2. import os: this library is used for knowing about the present working directory and getting a path for importing the dataset.
3. import random: This is just an optional library to generate sentences from the trained model. Just for fun!
4. import math: This library provides some important functions that are used for calculating perplexities.

• Brief description of the contributions of each group member (2%)

The work was equally distributed, and we worked together to complete the assignment.

Samkit worked on the pre-processing of data, computation of unigram and bigram probabilities, and smoothing of data.

Samarth Worked on preprocessing testing data, calculating perplexities, evaluation, and analysis of different models.

We both contributed equally to creating the report.

• Feedback for the project (2%) –

The project was a little challenging but more than that it was lengthy. It required a lot of focus on different things like unsmooth/smoothed probability, with/without unknown word handling, both for unigram and bigram which was confusing at first. It required quite some time to understand what was required to be done.

This project helped us to understand a lot of concepts practically. It also helped us to explore various ways in which the same task can be achieved.

It would be more helpful if we were provided with a code skeleton containing just the function names so that we can get an overview of the whole assignment. In that way, we could have focused more on just the implementation of the main concepts covered in class.