Homework 1
CS 6320: Natural Language Processing

Group – 22
Sahana Vinayak (sxv170330)
Sanatan Shrivastava (sxs220062)

October 9, 2023

Introduction/Goal

Models that assign probabilities to a sequence of words are Language Models. N-gram is a sequence of n words. For this homework, we implement a unigram (1-gram), bigram (2-gram), and some variants of these models. We First implement an unsmoothed unigram and bigram. Later, we implement the Laplace smoothing method and run the model for k = 1, 2, 3, and 4. Finally, we find the perplexity and compare the results.

Library functions imported: String (Used for preprocessing: removing punctuation/special characters from tokens), Math (Used for computation: Perplexity), and matplotlib.pyplot (Used for plotting perplexities of different models to compare them).

Dataset

The instructor provided the Dataset. The dataset consists of a training corpus to train the models and a validation corpus for validating the model. We use space between the words as a delimiter to obtain tokens.

Unsmoothed N-grams

The dataset is loaded and tokenized using the .split() function. For preprocessing, we removed all special characters (including punctuation marks) from the tokens and converted them to lowercase. We add the sequence of words (single words for unigram and sequence of two words for bigram) in a dictionary with their corresponding count (the number of times they occur in the document).

After the preprocessing step, we calculate the unsmoothed probabilities for the unigram and bigram models using the following formulas:

Unigram: $P(w) = count(w)/V$,
where w is a word in the corpus and V is the total number of words in the dataset.

Bigram: $P(w_i | w_{i-1}) = count(w_i, w_{i-1}) / count(w_{i-1})$,
where w_{i-1} and w_i are two consecutive words occurring in the dataset.

```
# Build unigram and bigram counts
unigram_counts = {}
bigram_counts = {}

for i in range(len(tokens) - 1):
  word = tokens[i]
  next_word = tokens[i + 1]

# Update unigram counts
  if word in unigram_counts:
    unigram_counts[word] += 1
  else:
    unigram_counts[word] = 1

# Update bigram counts
  if (word, next_word) in bigram_counts:
    bigram_counts[(word, next_word)] += 1
  else:
    bigram_counts[(word, next_word)] = 1
```

```
# Calculate unsmoothed unigram probabilities
total_tokens = len(tokens)
unigram_probabilities = {word: count / total_tokens for word, count in
unigram_counts.items()}

# Calculate unsmoothed bigram probabilities
bigram_probabilities = {(word1, word2): count / unigram_counts[word1] for (word1,
word2), count in bigram_counts.items()}

print("unigram probabilities: ", unigram_probabilities)
print("bigram probabilities: ", bigram_probabilities)
```

Smoothing

To handle unknown words, we use the add-1 method to assign some probability.
Since, count(UNK) is 0, in a unigram,

$$P[UNK] = 1/(N+1*V).$$

In a bigram,

$P[UNK] = 1/(count(w_{i-1}) + 1*V)$; if $count(w_{i-1})$ is not 0.
$P[UNK] = 1/V$; if $count(w_{i-1})$ is 0.

For known words, we first implement the add-1 method as follows:

For the unigram model: P(w) = (count(w) +1)/(N+V), where N is the sum of counts of all words in the dataset, and V is the total number of words in the dataset.
For the bigram model: $P(w_i|w_{i-1}) = (count(w_i, w_{i-1}) + 1)/(count(w_{i-1}) + V)$, where w_i, w_{i-1} are consecutive words in the dataset, and V is the total number of words in the dataset.

Later, we implemented add-k smoothing.
For the unigram model: P(w) = (count(w) +k)/(N+k*V), where k is positive whole number > 1, N is the sum of counts of all words in the dataset, and V is the total number of words in the dataset.
For the bigram model: $P(w_i|w_{i-1}) = (count(w_i, w_{i-1}) + K)/(count(wi-1) + k*V)$, where w_i, w_{i-1} are consecutive words in the dataset, k is positive whole number > 1, and V is the total number of words in the dataset.

```
def laplace_smoothing(unigram_counts, vocabulary_size, k=1):
       smoothed_probabilities = {}
       total_tokens = sum(unigram_counts.values())

      for word, count in unigram_counts.items():
         smoothed_probabilities[word] = (count + k) / (total_tokens + k *
vocabulary_size)

return smoothed_probabilities
```

```
def laplace_smoothing_bigram(unigram_counts, bigram_counts, k=1):
   smoothed_probabilities = {}
   for (w1, w2), count in bigram_counts.items():
      smoothed_probabilities[(w1, w2)] = (count + k) / (unigram_counts[w1] +
k*len(bigram_counts))
   return smoothed_probabilities
```

Perplexity

Perplexity, $PP = \exp((1/N) * \Sigma\text{-}\ln(P(w_i|w_{i1},..w_{in+1})))$ where N is the number of total words and $P(w_i|w_{i1},..w_{in+1})$ is the n-gram probability.
We calculated perplexity on the validation corpus.
To do this, we first loaded the validation dataset and extracted the tokens (this process is similar to the pre-processing of the training dataset). Later, we use the probabilities calculated from the training dataset to calculate the perplexity. If a word (or sequence of words) appears in the validation set but not in the training set, then we use the method discussed in the "Smoothing" section to handle their probabilities.

```
import math
def calculate_perplexity(validation_text, ngram_probabilities, n, k, unigram_counts):
```

```
#ngram_probability is a dictionary of probabilites computed from the training dataset
  tokens_old = validation_text.split()
  tokens = [removeChar(x) for x in tokens_old]
  tokens = [x for x in tokens if x != '']
  perplexity = 0
  N = len(tokens)

  for i in range(n - 1, N):
#extracting tokens based on n (n=1 for unigram, n=2 for bigram)
     ngram = tuple(tokens[i - n + 1 : i + 1])
     if ngram in ngram_probabilities:
       ngram_prob = ngram_probabilities[ngram]
     else:
        # Use a fallback probability for unknown n-grams
        if n==1:
          ngram_prob = k/(sum(unigram_counts.values())+k*N)
        elif n==2 and ngram[0] in unigram_counts:
          prev = ngram[0]
          ngram_prob = k/(unigram_counts[prev] + k*N)
        else:
          ngram_prob = 1/N
      perplexity -= math.log2(ngram_prob)
   perplexity = math.exp(perplexity / N)
   return perplexity
```

Results

The outcomes of our studies using various smoothing methods and changing values of k are presented in this section. Perplexity, which assesses the model's capacity to foretell test results, served as the main criterion we employed for evaluation.

6.1 Unigram Model

We used a simple unigram model as the basis for our experiments. At first, we calculated perplexity in an unsmoothed (i.e., raw) form. As a starting point for comparison, we used the resulting perplexity value.

To see how different degrees of smoothing influenced the perplexity of the unigram model, we then applied Laplace smoothing with k values of 1, 2, 3, and 4. The results are depicted in the figure below.

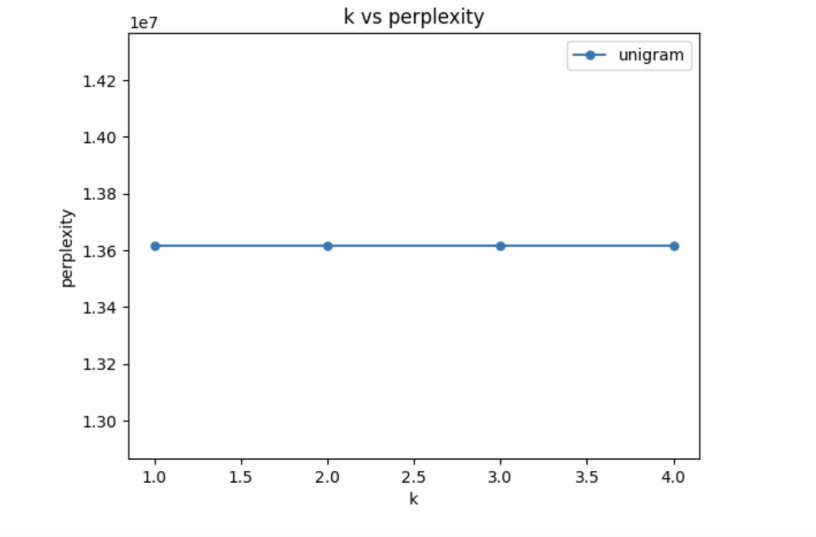

Fig. k vs perplexity graph for the unigram model

6.2 Bigram Model

With the bigram model, we expanded our studies and once more used an unsmoothed version as a starting point.
The Bigram model was then subjected to Laplace smoothing with the same range of k values in order to evaluate its effects.

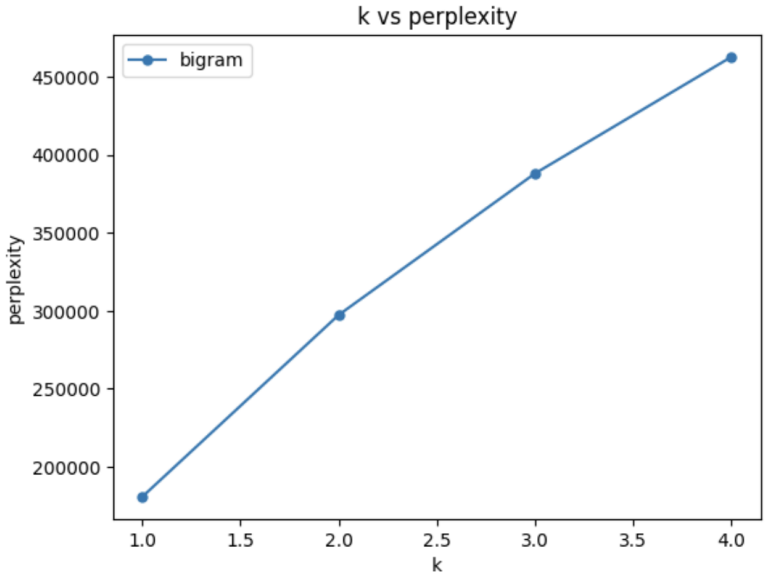

Fig. k vs perplexity graph for the bigram model

The values for k = 1,2,3,4 for unigram and bigram along with obtained perplexity are shown in the figure below.

```
for k = 1, unigram perpexility is 13616669.121676987
for k = 1, bigram perpexility is 180833.81207186953
---
for k = 2, unigram perpexility is 13616669.121676987
for k = 2, bigram perpexility is 297291.06531961076
---
for k = 3, unigram perpexility is 13616669.121676987
for k = 3, bigram perpexility is 387943.8175907893
---
for k = 4, unigram perpexility is 13616669.121676987
for k = 4, bigram perpexility is 462606.23768245155
---
```

6.3 Findings and Analysis

Our research yielded a number of conclusions and trends, including:
Smoothing: We found that both bigram and unigram models' perplexity scores were generally improved by smoothing approaches. Results for Laplace smoothing in particular were encouraging.
Impact of k: The confusion was significantly reduced when the value of k was increased in the Laplace smoothing. Lower perplexity values were produced by more aggressive smoothing at smaller k values. However, there comes a point where further increases in k might not produce appreciable gains.
Model Comparison: In terms of perplexity, the bigram model typically beat the unigram model, demonstrating that taking into account the prior word's context increased the model's predictive capacity.

The code to the above implementation can be found at:
https://github.com/Sanatan-Shrivastava/cs6320_nlp_g22

Feedback: The assignment was moderately difficult since it deals with the basic principles and their applications in NLP. We spent around 20-25 hours going through slides, understanding concepts like tokenization, N-grams, and perplexity from the recommended textbooks followed by the implementation of these concepts as per the instructions given in the assignment.

Contributions
Sahana's contribution: Preprocessing, Smoothing for Bigrams, perplexity, plotting perplexities for different k values.

Sanatan's contribution: Introduction/Goal, Dataset, Unsmoothed n-grams, Smoothing for Unigrams, Results.