CS6320 Assignment 1

https://github.com/phsaikiran/CS6320-Fall23-Group30

Group30 Hima Sai Kiran Prudhivi Abhilash Rajesh Bagalkoti
HXP220011 AXB220071

1 Implementation Details

1.1 Preprocessing

```
def preprocess(line):
  line = line.lower()
  tokens = nltk.word_tokenize(line)
  tokens = list(filter(lambda x: len(x)!=1 or
      (len(x)==1 and x not in string.punctuation), tokens))
  tokens = ["<START>"] + tokens + ["<END>"]
  return tokens
```

Listing 1: Preprocessing

1.1.1 line.lower()

This line converts all the characters in the input string line to lowercase to ensure uniformity in text data, as it makes all words lowercase and helps reduce the dimensionality of the data.

1.1.2 nltk.word_tokenize(line)

Tokenization is the process of splitting a text into individual words or tokens. We decided to use `word_tokenize()` instead of `str.split(" ")` because `str.split(" ")` was unable to remove edge cases involving new line char and words with full stop without space (like `"word."`).

1.1.3 filter string.punctuation

The purpose of this filtering is to remove one-character tokens, except for those that are punctuation marks. Punctuation marks in other words like `don't` are retained to maintain the structure of sentences.

1.1.4 Adding START and END tokens

These tokens are used to indicate the start and end of a sequence. They can help the model understand the boundaries of sentences or sequences.

1.2 Unigram and bigram probability computation

1.2.1 Unsmoothed unigrams and bigram

We count the occurrences of unigrams and bigrams in a preprocessed text and store these counts in dictionaries (unigrams and bigrams) for further analysis or processing.
We calculate the probability of each unique word in a text corpus based on the word's frequency count in the corpus (`unigram_counts`) and the total number of unigrams in the corpus

(`total_unigrams`). Similarly, for bigrams, we calculate the probability by dividing the count by the previous word count.

```
unigram_probability = {}
for u in unigram_counts.keys():
    unigram_probability[u] = unigram_counts[u]/total_unigrams
bigram_probability = {}
for b in bigram_counts.keys():
    bigram_probability[b] = bigram_counts[b]/unigram_counts[b[0]]
```

Listing 2: Unigrams and bigrams count

1.3 Unknown word handling

The code snippet filters a vocabulary of words by removing infrequent words (occurring less than two times) and adding a special `UNK` token to represent unknown words. The resulting vocabulary contains only frequent words that are considered relevant for further tasks. After adding `UNK` to the vocabulary, the unigram and bigram counts and probabilities are recalculated, considering `UNK` token in the calculations.

```
unknown_words = []
for u in unigram_counts.keys():
    if unigram_counts[u] < 2:
        unknown_words += [u]

vocab = vocab - set(unknown_words)
vocab.add("<UNK>")
```

Listing 3: Unknown words

1.4 Smoothing

```
k_unigram = {}
for uc_item in uc.keys():
    k_unigram[uc_item] = (uc[uc_item] + k) / (total_unigrams + k*vocab_len)

k_bigram = {}
for bc_item in bc.keys():
    k_bigram[bc_item] = (bc[bc_item] + k) / (uc[bc_item[0]] + k*vocab_len)
```

Listing 4: Add-k smoothing

Add-k smoothing, for unigram and bigram probabilities, is a technique used in natural language processing to estimate probabilities of events, such as word or bigram occurrences, by adding a constant (k) to the observed counts to account for unseen or zero-count events. This helps prevent zero probabilities, which can be problematic in certain probabilistic models.
Three types of smoothing were studied

- No smoothing `k=0`
- Laplace smoothing `k=1`

- Add-k smoothing `k=[0.01, 0.05, 0.1, 0.5, 0.75, 1, 1.5, 2, 5]`

1.5 Implementation of perplexity

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log(P(w_i|w_{i1},...,w_{in+1}))\right)$$

Perplexity is a measure of how well a probabilistic model predicts a given dataset. Lower perplexity values indicate better predictive performance. We calculated the perplexity of a corpus based on a unigram and bigram language model. It iterates through each token in the corpus, accumulates the log probabilities of those tokens, and then computes the perplexity score. This function is useful for evaluating how well the model predicts the given text corpus.

```
for token, next_token in list(pairwise(preprocess_line)):
    if token not in vocab:
        token = '<UNK>'
    if next_token not in vocab:
        next_token = '<UNK>'

    perplexity += np.log(bigram_model.get((token, next_token),
    bigram_model.get(('<UNK>', '<UNK>'))))

return np.exp(-perplexity/N)
```

Listing 5: Bigram perplexity

2 Eval, Analysis and Findings

Both unigram and bigram perplexity scores steadily rise. Larger ’k’ values introduce aggressive smoothing, assigning higher probabilities to unseen or rare words, which leads to less accurate predictions.

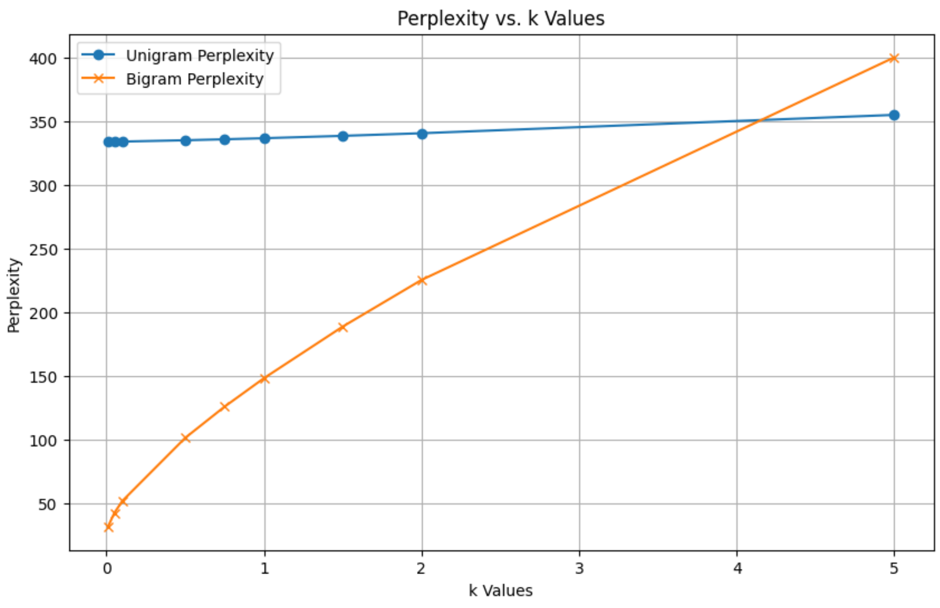

Figure 1: Example of a Code Piece.

Notably, bigram perplexity shows a more pronounced increase with higher ’k’ values, reflecting the model’s sensitivity to context. This discrepancy can be attributed to the differences in the

models themselves. Bigram models heavily rely on context, making them more sensitive to the effects of smoothing. As a result, when aggressive smoothing is applied with larger 'k' values, the uncertainty introduced can have a more substantial impact on prediction accuracy in bigram models.
Smaller 'k' values, such as 0.01 and 0.05, yield lower perplexity scores, indicating better model performance in terms of accuracy. However, they might struggle to handle unseen data effectively. Conversely, larger 'k' values, like 5, result in higher perplexity, suggesting that the models become less accurate with aggressive smoothing.

k	Unigram perplexity	Bigram perplexity
k= 0.01	334.0702538117335	31.438671310147996
k= 0.05	334.15570136142924	42.14908179696627
k= 0.1	334.266914692295	51.49526979459143
k= 0.5	335.31117648600747	101.21397784081431
k= 0.75	336.08081799724073	125.7687182112564
k= 1	336.9231410554504	148.16885572941624
k= 1.5	338.7847019583779	188.76848656053343
k= 2	340.8320573388146	225.44994311452237
k= 5	355.2634231662423	400.4942288129241

Table 1: unigram and bigram perplexity scores

Selecting the language model with a 'k' value of 0.01 is the recommended choice for several compelling reasons. Firstly, this model achieved the lowest perplexity scores among the 'k' values tested, indicating superior prediction performance on the dataset. Importantly, it strikes a well-balanced equilibrium between smoothing and accuracy. With 'k' set at 0.01, the model effectively handles unseen or rare words, ensuring robustness without compromising overall prediction accuracy.

3 Other details

3.1 Programming library usage

NLTK (Natural Language Toolkit)
String Module (Python Standard Library)
NumPy (Numerical Python)
Itertools Module (Python Standard Library)

3.2 Contributions

Team member 1: Hima Sai Kiran Prudhivi: Loading and preprocessing data, Calculating unsmoothed unigrams and bigrams, Handling unknown words
Team member 2: Abhilash Rajesh Bagalkoti: Implementing smoothing techniques, Calculating perplexity scores, Documenting findings and creating the LaTeX report

3.3 Feedback

Difficulty Level: The project's difficulty level was just right for our current level of understanding and skills.
Time Spent: Our team spent approximately 15-20 hours collectively on the project over the course of 2 weeks.
Educational Value: The project significantly contributed to our understanding of the course content. It allowed us to apply the concepts of language modeling, probability, and smoothing techniques in a practical context. The hands-on experience of calculating perplexity scores and analyzing the impact of different smoothing strategies was particularly valuable.