Assignment 1

https://github.com/matthewevans87/cs6320-nlp-assignment1

Group 22 | Matthew Evans mre042000 | Ruochen Meng rxm220120 | Saul Garay SAG180009

1 Implementation Details

The inline documented code can be found at the GitHub repository given above. The experiments can be run from the entry point in `Main.py`.

1.1 Overview

The model builds a vocabulary from the training corpus, and computes n-gram probabilities with add-k smoothing (laplace for k=1). Unknown words are handled by replacing low-frequency words with a special `<UNK>` token. The model computes perplexity on the test corpus using the smoothed probabilities.

1.2 Data Structures

The model uses type aliases for clarity:

- `Token` - a string representing a word or symbol.
- `NGram` - a tuple of Tokens representing an n-gram.
- `Dataset` - a list of strings, each string is a line from the corpus.

Further, the follow data structures are used:

- `Vocabulary: set[Token]` - the set of known tokens.
- `Token Counts: Dict[Token, int]` - maps each token to its occurrence count.
- `Probability: float` - represents the probability of an n-gram.

1.3 Vocabulary Construction

The vocabulary is constructed by iterating over the training corpus and counting the occurrences of each token.

1.3.1 Unknown word handling

Tokens which occur less than a specified threshold are replaced with the `<UNK>` token to handle unknown words in the test set. Specifically, probability mass of tokens is accumulated until a specified coverage threshold is reached, at which point the remaining tokens are replaced with `<UNK>`. The steps are as follows:

1. Sort tokens in descending order by occurrence count.
2. Iterate over token counts, accumulating mass, M, i.e.,

$$\mathrm{M}_{\mathrm{cum}} = \sum_{t_i}^{t_n} \mathrm{C}(t_i)$$

If $(\mathrm{M}_{\mathrm{cum}}/M_{\mathrm{total}}) < \mathrm{M}_{\mathrm{coverage}}$, assign the token's mass to the `<UNK>` token.

1.3.2 Implementation

Our implementation, given in 1, computes the vocabulary and applies the unknown token handling as described.

```
def _get_vocabulary(data: Dataset, coverage: float) -> set[Token]:
    token_counts: dict[str, int] = dict()
    for line in data:
        for token in NGramLanguageModel._parse_tokens(line):
            token_counts[token] = token_counts.get(token, 0) + 1

    token_counts = NGramLanguageModel.apply_unk_by_coverage(token_counts,
                                                             coverage)

    vocabulary = set(token_counts.keys())
    return vocabulary

def apply_unk_by_coverage(token_counts, coverage):
    total_mass = sum(token_counts.values())
    sorted_token_counts = sorted(token_counts.items(),
                                 key=lambda x: x[1],
                                 reverse=True)

    updated_token_counts: dict[str, int] = dict()
    cumulative_mass = 0
    unknown_count = 0
    for token, count in sorted_token_counts:
        if cumulative_mass / total_mass < coverage:
            updated_token_counts[token] = count
            cumulative_mass += count
        else:
            unknown_count += count
    updated_token_counts[UNKNOWN_TOKEN] = unknown_count
    return updated_token_counts
```

Listing 1: Vocabulary construction and unknown token handling.

1.4 Probability Computation and Smoothing

The model computes n-gram probabilities using add-k smoothing, supporting $k \geq 0$. The probability of an n-gram is computed as

$$P(w_n|w_{n-1}, \ldots, w_1) = \frac{C(w_1, \ldots, w_n) + k}{C(w_1, \ldots, w_{n-1}) + kV}$$

where $C(w_1, \ldots, w_n)$ is the count of the n-gram, $C(w_1, \ldots, w_{n-1})$ is the count of the (n-1)-gram context, V is the vocabulary size, and k is the smoothing parameter.

1.4.1 Implementation

Our implementation, given in 2, handles both unigram and higher-order n-grams, with special handling for the unigram case where the denominator is the total token count plus kV. The add-k smoothing is captured by the parameter `smoothing`.

```python
def get_probability(self, *tokens: Token) -> float:
  vocabulary_size = len(self.vocabulary)

  # handle unigram case
  if self.n == 1:
      a = (self.ngram_counts.get(tokens, 0) + self.smoothing)
      b = (self.all_tokens_count + (self.smoothing * vocabulary_size))
      return a / b if b > 0 else 0.0

  # handle ngram case
  else:
      preceding = tokens[:-1]

      # Count of the full n-gram tokens
      a = self.ngram_counts.get(tokens, 0) + self.smoothing

      # Count of the (n-1) preceding tokens
      b = self.context_counts.get(preceding, 0)
          + (self.smoothing * vocabulary_size)

      return a / b if b > 0 else 0.0
```

Listing 2: Probability calculation function with add-k smoothing.

1.5 Implementation of perplexity

Perplexity is computed as the exponentiation of the negative average log-probability of the test corpus. The formula for perplexity is given by

$$PP(W) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{i-1},\ldots,w_{i-n+1})\right)$$

where W is the test corpus, N is the total number of tokens in the test set, and $P(w_i|w_{i-1},\ldots,w_{i-n+1})$ is the probability of token w_i given its preceding $n-1$ tokens.

1.5.1 Implementation

The implementation, given in 3, iterates over each line in the test dataset, computes the log-probability of each n-gram, and accumulates these, taking their mean, to compute the overall perplexity.

```python
import numpy as np
def get_perplexity(self, datum: str) -> float:
    ngrams: list[NGram] = []

    ngrams.extend(NGramLanguageModel._get_ngrams_from_line(datum,
                                                           self.vocabulary,
                                                           self.n))
    nll = - np.log([self.get_probability(*tokens) for tokens in ngrams])
    pp: np.float64 = np.exp(np.mean(nll))

    return pp

def get_mean_perplexity(self, data: Dataset) -> float:
    pps: list[float] = []
    for line in data:
        pp = self.get_perplexity(line)
        pps.append(pp)
    return float(np.mean(pps))
```

Listing 3: Perplexity calculation.

2 Evaluation, Analysis and Findings

To evaluate our n-gram language model, we conducted experiments on three key hyperparameters: n-gram size (n), the add-k smoothing parameter (k), and vocabulary coverage. Performance was measured by perplexity on a validation set. While a lower perplexity score is generally better, our final model selection balances this metric with established principles of language modeling to ensure robustness and utility.

2.1 Effect of N-gram Size

The first experiment evaluated the impact of n-gram size (n), with other hyperparameters held constant.

Model	Training Perplexity	Validation Perplexity
Unigram ($n = 1$)	420.14	**358.86**
Bigram ($n = 2$)	702.01	835.22
Trigram ($n = 3$)	1868.71	3066.01

Table 1: Perplexity scores for different n-gram sizes (n) with $k = 1$ and 99% coverage.

2.1.1 Analysis

In theory, perplexity should decrease as n increases due to more available context. Our results show the opposite trend, an outcome that is characteristic of **data sparsity** from a limited training corpus. While the unigram ($n = 1$) model scored best, it does not retain the context of previous words in the sequence. The trigram ($n = 3$) model was too complex and suffered from extreme sparsity. Therefore, we selected the **bigram model** ($n = 2$) as it represents the best balance between capturing essential context and avoiding the unreliability of the trigram model.

2.2 Effect of Smoothing Strategy

Using the chosen bigram model, we evaluated the effect of the smoothing parameter k.

Smoothing Value (k)	**Validation Perplexity**
0 (No Smoothing)	∞
0.1	324.08
1.0 (Laplace)	**835.22**
5.0	1791.39

Table 2: Validation perplexity for a bigram model with varying k and 99% coverage.

2.2.1 Analysis

As expected, the model with no smoothing ($k = 0$) produced an infinite perplexity. The results indicate that the lowest perplexity was achieved at $k = 0.1$. However, this may represent overfitting to our specific validation set. To prioritize a more general and robust model, we selected the widely-used **Laplace smoothing ($k = 1.0$)** as the optimal choice.

2.3 Effect of Vocabulary Coverage

Finally, the impact of the vocabulary coverage threshold on the bigram model was examined.

Coverage	**Validation Perplexity**
80%	55.92
90%	163.09
95%	351.74
99%	**835.22**
100%	1112.94

Table 3: Validation perplexity for a bigram model with varying vocabulary coverage.

2.3.1 Analysis

Although the perplexity score is lowest at 80% coverage, this result is misleading and should be interpreted with caution. The apparent improvement is an artifact of the `<UNK>` token becoming a high-frequency, easily predictable "word," which does not reflect a better understanding of the language. We therefore chose **99% coverage** as our optimal value to maintain a rich and useful vocabulary.

2.4 Findings and Conclusion

Our evaluation highlights the importance of carefully interpreting metrics rather than optimizing for them blindly. After balancing empirical results with theoretical principles, our final model is a **bigram ($n = 2$) model with Laplace smoothing ($k = 1.0$) and a vocabulary coverage of 99%**. This configuration, achieving a validation perplexity of **835.22**, represents a robust baseline that prioritizes real-world utility and accuracy over raw metric optimization.

3 Miscellaneous

3.1 Python Library Usage

The implementation uses minimal external libraries, primarily relying on Python's standard library. The only external library used is NumPy, which is employed for numerical computations, specifically for calculating logarithms and exponentials in the perplexity calculation. The project was written in Python 3.13.7.

3.2 Feedback

- The coding portion was completed within approximately one day, with a few issues resolved via pull requests over the following week or so.

- The assignment was simple enough to complete with minimal experience in NLP, while still fostering practical skills.
- Providing a target perplexity as a benchmark would enhance the clarity of expectations.
- Incorporating automated grading with the option for unlimited submissions would be a valuable addition, allowing students to iteratively refine their implementations based on immediate feedback.

3.3 Contributions

- Matthew Evans: Initial project setup, vocabulary construction, unknown token handling, probability computation, smoothing, and perplexity calculation.
- Ruochen Meng: Review, bug fixes in probability calculation, improvements to unknown token handling.
- Saul Garay: Review, bug fixes in vocabulary construction and probability calculation.