Assignment 1

https://github.com/nikhilkondapalli5/CS-6320-NLP-Assignment-1.git

Group 16

Bhargava Siva Naga Sai Potluri
bxp230045

Kavimayil Periyakoravampalayam Komarasamy
kxp230053

Nikhil Sesha Sai Kondapalli
nxk240025

Sakshi Tokekar
sxt230143

1 Implementation Details

1.1 Preprocessing, Unigram and bigram probability computation

The preprocessing pipeline included the following steps:

1. **Lowercasing**: All tokens were converted to lowercase to reduce vocabulary size.
2. **Contraction Expansion**: The `contractions` Python library expanded contracted forms (e.g., *don't* → *do not*).
3. **Number Normalization**: Digit sequences were replaced with `<num>` to prevent sparsity due to numeric variation.
4. **Sentence Boundaries**: Start (`<s>`) and end (`</s>`) markers were added to each sentence for correct bigram modeling.
5. **Punctuation Handling**: Punctuation was retained, which preserved structure but increased sparsity.
6. **Tokenization**: Performed via whitespace splitting. While this approach was simple, it did not properly separate punctuation from words.

```python
def preprocessing(self, data, remove_stopwords=False):
    # pre-process individual sentence
    processed_sentences = []
    for line in data:
        processed_line = self.preprocess_single_sentence(line, remove_stopwords)
        if processed_line.strip():
            processed_sentences.append(processed_line)
    return ' '.join(processed_sentences)
```

```python
def preprocess_single_sentence(self, sentence, remove_stopwords=False):
    # Lowercasing
    sentence = sentence.lower()
    # Contractions expansion
    sentence = contractions.fix(sentence)
    # Normalize numbers -> <num>
    sentence = re.sub(r'\d+', '<num>', sentence)
    # Add sentence boundaries <s> </s> for each line/review
    sentence = sentence.strip()
    if sentence:
        sentence = '<s> ' + sentence + ' </s>'
    return sentence
```

Figure 1: Preprocessing implementation

Data Structures:

- `unigram_counts`: token → count
- `bigram_counts`: (w_1, w_2) → count
- `unigram_probabilities`: token → $P(w)$

- `bigram_probabilities`: $(w_1, w_2) \rightarrow P(w_2|w_1)$

Python dictionaries provided $O(1)$ average access/update efficiency. Pandas DataFrames were used for exporting results. This is simple and interpretable, but not memory-optimized for very large corpora.

Unigram and Bigram Models. The unigram model was constructed by counting the frequency of tokens and applying Maximum Likelihood Estimation (MLE):

$$P(w) = \frac{\text{count}(w)}{N}$$

The bigram model captured conditional dependencies between consecutive tokens:

$$P(w_i \mid w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$$

```python
# Count occurrences of each token
for token in tokens:
    unigram_counts[token] = unigram_counts.get(token, 0) + 1

# Calculate total number of tokens
total_tokens = len(tokens)

# Compute probabilities
unigram_probabilities = {}
for unigram, count in unigram_counts.items():
    unigram_probabilities[unigram] = count / total_tokens

return unigram_probabilities, unigram_counts
```

```python
# Vocabulary size
V = len(unigram_counts)

bigram_probs = {}
for (w1, w2) in bigram_counts:
    bigram_probs[(w1, w2)] = (
        bigram_counts[(w1, w2)] + k
    ) / (unigram_counts[w1] + k * V)

# For all possible bigram pairs (including unseen ones)
all_vocab = list(unigram_counts.keys())
for w1 in all_vocab:
    for w2 in all_vocab:
        if (w1, w2) not in bigram_probs:
            bigram_probs[(w1, w2)] = k / (unigram_counts[w1] + k * V)

return bigram_probs, bigram_counts
```

Figure 2: Left - Unigram Model, Right - Bigram Model

Both models initially used unsmoothed probabilities, leading to zero-probability issues for unseen events.

1.2 Smoothing (Laplace and Add-k)

To avoid zero probabilities for unseen n-grams, we implemented Laplace Smoothing and Add-k smoothing for both unigram and bigram models. Without smoothing, any unseen word or bigram would yield a probability of zero, resulting in undefined perplexity.

1.2.1 Unigram Model with Smoothing

With Add-k smoothing, we adjust the unigram probabilities as:

$$P(w) = \frac{\text{Count}(w) + k}{N + kV}$$

where:

- V is the vocabulary size,
- k is the smoothing constant (When $k = 1$,it corresponds to Laplace Smoothing).

```python
for token in tokens:
    unigram_counts[token] = unigram_counts.get(token, 0) + 1

# Vocabulary size
V = len(unigram_counts)
total_tokens = len(tokens)

unigram_probs = {}
for word in unigram_counts:
    unigram_probs[word] = (unigram_counts[word] + k) / (total_tokens + k * V)
```

Figure 3: Smoothing on Unigrams

1.2.2 Bigram Model with Smoothing

To handle unseen bigrams, we apply Add-k smoothing:

$$P(w_2 \mid w_1) = \frac{\text{Count}(w_1, w_2) + k}{\text{Count}(w_1) + kV}$$

- We selected Add-k smoothing because it is simple, guarantees non-zero probabilities, and serves as a strong baseline.
- The default $k = 1.0$ corresponds to Laplace smoothing, but smaller values (e.g., $k = 0.5, 0.1, 0.05$) can reduce over-smoothing.
- This approach stabilizes perplexity calculations, ensuring unseen words and bigrams do not collapse the model.

```
# Vocabulary size
V = len(unigram_counts)

bigram_probs = {}
for (w1, w2) in bigram_counts:
    bigram_probs[(w1, w2)] = (
        bigram_counts[(w1, w2)] + k
    ) / (unigram_counts[w1] + k * V)

# For all possible bigram pairs (including unseen ones)
all_vocab = list(unigram_counts.keys())
for w1 in all_vocab:
    for w2 in all_vocab:
        if (w1, w2) not in bigram_probs:
            bigram_probs[(w1, w2)] = k / (unigram_counts[w1] + k * V)

return bigram_probs, bigram_counts
```

Figure 4: Smoothing on Bigrams

1.3 Unknown Word Handling

To avoid assigning zero probability to unseen words, all rare tokens in the training set, defined as words with frequency less than or equal to three, were replaced with the special `<UNK>` token. This ensures that during evaluation on validation or test data, any word not present in the training vocabulary is mapped to `<UNK>`, allowing it to receive a non-zero probability. Replacing rare words with `<UNK>` also reduces vocabulary sparsity, which improves the reliability of probability estimates, particularly for bigram models. Overall, this preprocessing step enhances the model's ability to generalize to unseen data and prevents artificially inflated perplexity scores caused by zero-probability events.

```
def handle_unknown_words(self, tokens, unigram_counts, min_freq=1):
    """
    Replaces words with a frequency less than or equal to min_freq
    with the <UNK> token.
    """
    # Create a set of rare words for efficient lookup
    rare_words = set()

    # Loop over each word-count pair in the unigram_counts dictionary
    for word, count in unigram_counts.items():
        if count <= min_freq:
            rare_words.add(word)  # add the rare word to the set

processed_tokens = []
for token in tokens:
    if token in rare_words:        # if token is rare
        processed_tokens.append('<UNK>')  # replace with <UNK>
    else:
        processed_tokens.append(token)    # otherwise keep as is

return processed_tokens
```

Figure 5: Function replacing words $\leq$ min-freq with `<UNK>`

1.4 Implementation of Perplexity

To evaluate our models, we implemented perplexity as the primary metric of performance. Perplexity measures how well a probability distribution predicts a sequence of words, with lower values indicating better predictive ability.

Perplexity is calculated as follows:

$$PP = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i \mid w_{i-1})\right)$$

This provides a measure of model quality on unseen data.

```python
for token in test_tokens:
    if token in unigram_probs:
        prob = unigram_probs[token]
    else:
        # Handle unseen words by assigning probability of them
        prob = unigram_probs.get(self.unk_token, 1e-10)

    if prob > 0:
        log_prob_sum += math.log(prob)
    else:
        log_prob_sum += math.log(1e-10)

perplexity = math.exp(-log_prob_sum / N)
return perplexity
```

```python
for i in range(len(test_tokens) - 1):
    bigram = (test_tokens[i], test_tokens[i+1])

    if bigram in bigram_probs:
        prob = bigram_probs[bigram]
    else:
        # Backoff to unigram
        if test_tokens[i+1] in unigram_probs:
            prob = unigram_probs[test_tokens[i+1]]
        else:
            prob = unigram_probs.get(self.unk_token, 1e-10)

    if prob > 0:
        log_prob_sum += math.log(prob)
    else:
        log_prob_sum += math.log(1e-10)

perplexity = math.exp(-log_prob_sum / N)
```

Figure 6: Left - Unigram Perplexity; Right - Bigram Perplexity

2 Evaluation, Analysis and Findings

We successfully implemented n-gram language models by constructing unigram and bigram probabilities with proper preprocessing, incorporating methods for handling unseen words, applying smoothing techniques, and evaluating model performance using the perplexity metric.

2.1 Perplexity Results

The following are the results when words with minimum frequency ≤ 3 are replaced with `<UNK>` tokens:

```
================================================================================================
                              PERPLEXITY RESULTS
================================================================================================

Smoothing Method  Training Unigram  Training Bigram  Validation Unigram  Validation Bigram
     Un-smoothed        247.550571        32.295091          231.860777          57.029845
   Laplace (k=1)        247.756526       196.699824          232.423302         204.963898
   Add-k (k=0.5)        247.605317       138.789169          232.104380         161.558013
   Add-k (k=0.1)        247.552880        69.586103          231.902757         108.370577
  Add-k (k=0.05)        247.550577        36.479045          231.862791          98.716894

================================================================================================
```

Figure 7: Perplexity Results comparing Training and Validation Models

2.2 Training and Validation Perplexity

For the **unigram models**, perplexity values remain consistently high across both training and validation sets. This indicates that the unigram model is too simplistic to capture meaningful language dependencies. Smoothing has negligible effect on unigram results since the frequency counts in the training data already provide sufficient coverage.

For the **bigram models**, the unsmoothed version achieves very low perplexity on the training set but degrades noticeably on the validation set, highlighting overfitting and brittleness due to unseen bigrams. When smoothing is applied, the results vary significantly. Laplace smoothing, which uniformly redistributes probability mass, substantially worsens validation performance by over-smoothing and flattening the probability distribution.

In contrast, Add-k smoothing shows more promising behavior. With larger values of k, the bigram model still generalizes poorly, but as k decreases, performance improves considerably.

Small values of k offer the best trade-off, effectively handling unseen events without excessively distorting the learned distribution.

2.3 Effect of Smoothing Strategies on Validation Performance

For **unigram models**, smoothing strategies have almost no impact, and the model consistently underperforms compared to higher-order models.

For **bigram models**, smoothing plays a critical role. The unsmoothed bigram model provides strong performance but is vulnerable to zero-probability issues on unseen data. Laplace smoothing degrades performance sharply due to over-smoothing. Add-k smoothing with small k values is the most effective strategy, striking a balance between robustness to unseen bigrams and preservation of the training distribution.

2.4 Key Findings

- There is a tradeoff between replacing words $\leq$ min-frequency with `<UNK>` and the performance of the model. Smaller min-frequency limit results in high Vocabulary but also higher perplexity reducing utility. We found min-freq ≤ 3 to be giving optimal perplexity results.
- We notice bigram perplexity is lesser than unigram's in all cases for both training and validation dataset as bigram probabilities are higher than unigrams.
- Unigram models are too simple and consistently exhibit weak performance, unaffected by smoothing.
- Bigram models capture richer dependencies, but without smoothing they overfit and fail to generalize robustly.
- Laplace smoothing is too aggressive for bigram models and leads to significant performance degradation.
- Add-k smoothing with small values of k provides the best generalization, effectively balancing smoothing of unseen events with maintenance of frequency information.
- The most practical bigram configuration is with a small k value, which generalizes well while avoiding zero-probability problems.

3 Others

3.1 Programming Library Usage

In implementing the n-gram language models, we used following Python libraries to simplify tasks such as text processing, data management, and file handling. Each library was chosen for its efficiency and reliability in handling common NLP and data operations.

- **re (Regular Expressions):** Used for text normalization, including replacing numbers with a placeholder token (`<num>`).
- **contractions:** Used to expand English contractions (e.g., `"can't"` to `"cannot"`) to standardize text. This helps the model better handle tokenization and reduces variability in word forms.
- **pandas:** Used for data storage, manipulation, and export. Pandas DataFrames facilitated the creation of unigram and bigram tables, computation of probabilities, and export to CSV files for analysis.
- **os:** Used to manage directories and file paths, ensuring that output CSV files could be saved in the desired locations, even if the directories did not exist.
- **math:** Used for mathematical operations such as logarithms and exponentiation when calculating perplexity and probability distributions.

3.2 Contributions of Each Group Member

The project was completed collaboratively, with each member responsible for specific components to ensure efficiency and clarity. The contributions are summarized as follows:

- **Sakshi:** Handled data preprocessing and implemented the unsmoothed unigram and bigram models, including probability computations.
- **Kavimayil:** Implemented Laplace and Add-k smoothing techniques for both unigram and bigram models.
- **Nikhil:** Developed the `<UNK>` token strategy, replacing rare words in the corpus and evaluated findings (Final Results).
- **Bhargava:** Implemented perplexity calculations for both unigram and bigram models and evaluated findings (Final Results).

All members contributed to documentation and report preparation, ensuring a cohesive and comprehensive presentation of the project.

3.3 Project Feedback

The project was well-structured and helped us to perform careful implementation of n-gram language models, smoothing strategies, and perplexity calculations, which made it engaging but manageable within the given timeframe. On average, each group member spent approximately 10–12 hours to complete their assigned tasks, including coding, testing, and report preparation.

This project significantly enhanced our understanding of language modeling concepts, particularly in the areas of probability estimation, handling unseen words, and evaluating models using perplexity. Working with real data highlighted the importance of preprocessing and smoothing in practical applications. Moreover, the collaborative nature of the project fostered effective teamwork, communication, and coordination among group members. Overall, the project was highly valuable for reinforcing theoretical concepts through hands-on implementation.