HW2 Report

Yifei Gan

1 N-gram Language Modeling

An n-gram language model predicts the probability of a word given its preceding context of size . The implementation involved creating unigram, bigram, and trigram models. For each model:
Unigram Model: Predicts each word based on its frequency in the entire corpus.
Bigram Model: Predicts each word based on the preceding word.
Trigram Model: Predicts each word based on the preceding two words.
To handle out-of-vocabulary (OOV) words, any word that appeared fewer than three times in the training set was replaced with a special <UNK> token.

1.1 Training Procedure

The training involved the following steps:
Data Preprocessing: The text was preprocessed by removing punctuation, converting to lowercase, and tokenizing each line into words.
Vocabulary Creation: Words that appeared fewer than three times in the training data were replaced with <UNK> to handle OOV words.
Model Training: The n-gram counts were collected for each word sequence, and the MLE probabilities were calculated by dividing the count of each word by the total count of words for each context.

1.2 Code Implementation

```
def preprocess(self, text):
    text = re.sub(r'[^a-zA-Z0-9\s]', '', text).lower()
    tokens = text.split()
    return tokens
```

The preprocessing step involved removing punctuation, converting all text to lowercase, and tokenizing the sentences. This was done to ensure uniformity and reduce variations due to capitalization or special characters. The preprocess function takes a line of text and returns a list of tokens after cleaning.

```
for word in word_counts:
    if word_counts[word] < 3:
        self.vocab.add('<UNK>')
    else:
        self.vocab.add(word)
```

During the vocabulary creation, I handled out-of-vocabulary words by converting words that appeared fewer than three times into a special <UNK> token. This helped in managing rare words and ensured the model could handle unknown words during evaluation.

```
for tokens in tokenized_lines:
    tokens = ['<START>'] * (self.n - 1) +
        [token if token in self.vocab else '<UNK>'
        for token in tokens] + ['<STOP>']
    for i in range(len(tokens) - self.n + 1):
        context = tuple(tokens[i:i + self.n - 1])
        word = tokens[i + self.n - 1]
        self.model[context][word] += 1
```

The training process was implemented using an n-gram counting approach. For each line in the training data, tokens were collected, and the n-gram counts were updated accordingly. The training phase produced the counts necessary to estimate the conditional probabilities for each n-gram context.

```
def get_ngram_prob(self, context, word):
    context_count = sum(self.model[context].values())
    word_count = self.model[context][word]
    return word_count / context_count if
        context_count > 0 else 0.0
```

To predict the next word, the conditional probability of each word given the preceding context was calculated. Add-one smoothing was applied to ensure that unseen n-grams still had non-zero probabilities.

```
def calculate_perplexity(self, corpus):
    total_log_prob = 0
    total_tokens = 0

    for line in corpus:
        tokens = ['<START>'] * (self.n - 1) +
            [token if token in self.vocab else
            '<UNK>' for token in
            self.preprocess(line)] + ['<STOP>']
        total_tokens += len(tokens) - (self.n - 1)

        for i in range(len(tokens) - self.n + 1):
            context = tuple(tokens[i:i + self.n - 1])
```

Model	Training Perplexity	Development Perplexity	Test Perplexity
Unigram	1129.00	1038.47	1037.81
Bigram	98.74	**inf**	**inf**
Trigram	8.06	**inf**	**inf**

Table 1: Part 1 Perplexity scores

Model	Training Perplexity	Development Perplexity	Test Perplexity
Unigram	1130.10	1040.78	1040.17
Bigram	2013.95	2429.31	2414.68
Trigram	6750.76	11054.39	11004.95

Table 2: Part 2 Perplexity scores

```
            word = tokens[i + self.n - 1]
            prob = self.get_ngram_prob(context, word)
            if prob > 0:
                total_log_prob += math.log(prob)
            else:
                return float('inf')

    avg_log_prob = total_log_prob / total_tokens
    perplexity = math.exp(-avg_log_prob)
    return perplexity
```

The perplexity calculation was used to evaluate how well the model predicted the test data. Lower perplexity indicates better predictive performance. The implementation calculated perplexity by summing the log probabilities of each word in the sequence and then normalizing by the length of the sequence.

1.3 Result

The perplexity scores for the unigram, bigram, and trigram models are presented in Table 1. The results show that the unigram model had a reasonable perplexity across all datasets, suggesting that it was somewhat effective at predicting individual words. The bigram and trigram models, while theoretically capable of capturing more context, resulted in infinite perplexity scores on the development and test sets. This indicates that these models encountered many unseen n-grams during evaluation, leading to zero probabilities and, consequently, infinite perplexity.The high perplexity scores for the bigram and trigram models are likely due to data sparsity. With larger n-grams, the likelihood of encountering unseen contexts increases, and without proper smoothing, the model cannot generalize well to these contexts. This highlights the need for more sophisticated smoothing techniques to ensure that even unseen n-grams are assigned non-zero probabilities.

2 Smoothing with Linear Interpolation

To address the data sparsity issue, linear interpolation smoothing was applied. This technique combines the probabilities from unigram, bigram, and trigram models to calculate a smoothed probability for each word, thereby ensuring that even unseen contexts receive a non-zero probability. The smoothed probability for a word given a context is defined as follows: where the weights are non-negative and sum to 1. These weights determine the contribution of each n-gram model to the final probability.
To find the optimal values for the interpolation weights , we used Bayesian optimization. This approach efficiently searches the parameter space to minimize the perplexity on the development set. By using Optuna, a popular optimization framework, we ensured that the model found the best possible combination of weights that improved generalization to the development and test sets. The optimization objective was to minimize perplexity, which measures how well the model predicts the words in the dataset.

2.1 Code Implementation

```
def get_smoothed_prob(self, context, word):
    prob = 0.0
    for i in range(self.n):
        sub_context = context[len(context) - i:]
        prob += self.lambdas[i] * self.get_ngram_prob
            (sub_context, word)
    return prob
```

Linear interpolation smoothing was applied to combine unigram, bigram, and trigram probabilities. The goal was to calculate a smoothed probability for each word, ensuring that unseen contexts contributed meaningfully to the model's predictions. The weights were optimized to achieve the best possible performance.

```
def optimize_lambdas(model, train_corpus, dev_corpus):
    def objective(trial):
        lambdas = [trial.suggest_uniform(f'lambda_{i}'
            , 0, 1) for i in range(model.n)]
        lambdas = [l / sum(lambdas) for l in lambdas]
        model.lambdas = lambdas
        return model.calculate_perplexity(dev_corpus)

    study = optuna.create_study(direction='minimize')
    study.optimize(objective, n_trials=50)
    return study.best_params
```

I used Optuna to find the best combination of interpolation weights. The objective function minimized the perplexity on the development set, and constraints were applied to ensure that the weights summed to 1. This automated hyperparameter tuning allowed us to efficiently explore the parameter space without manually adjusting the values.

2.2 Results

The unigram model had the lowest perplexity scores compared to the bigram and trigram models. The training perplexity was 1130.10, while the development and test perplexities were 1040.78 and 1040.17, respectively. These values suggest that the unigram model was reasonably effective at capturing the overall word frequency distribution but lacked the ability to capture contextual relationships between words. The relatively similar perplexity values across training, development, and test sets indicate that the model was not significantly overfitting to the training data, though its simplicity limited its ability to accurately model word sequences.

The bigram model had significantly higher perplexity scores compared to the unigram model. The training perplexity was 2013.95, while the development and test perplexities were 2429.31 and 2414.68, respectively. This indicates that while the bigram model was able to incorporate some contextual information from the preceding word, it struggled with data sparsity and was unable to generalize effectively. The higher perplexity on the development and test sets suggests that the model over-relied on specific bigram sequences seen during training, leading to poorer performance on unseen data.

The trigram model showed the highest perplexity scores among all the models. The training perplexity was 6750.76, while the development and test perplexities were 11054.39 and 11004.95, respectively. This indicates that the trigram model faced severe data sparsity issues, as it relied on two preceding words for prediction, making it more susceptible to encountering unseen n-grams during evaluation. The large gap between training and evaluation perplexities also suggests significant overfitting, as the model was able to memorize specific trigram sequences from the training data but struggled to generalize to the development and test sets.