Assignment 1: N-grams and Misspelling Correction Report
ELL884

Animesh Lohar - 2024EET2368

1 Introduction

This report details the implementation of an n-gram language model and its application to a probabilistic misspelling error correction system. The system comprises several components: a basic n-gram class, various smoothing techniques, and a noisy-channel model for misspelling correction. The report covers the implementation details of each component, performance analysis, text generation examples, and an explanation of the misspelling correction model.

2 Code Overview

The project is structured into several Python files:

- `config.py`: Defines all hyperparameters used in the implementation.
- `ngram.py`: Implements the base n-gram class (`NGramBase`).
- `smoothing_classes.py`: Implements different smoothing techniques.
- `spelling_corrector.py`: Implements the misspelling error correction system.
- `main.py`: Main script to load data, train the models, and evaluate the system.

2.1 `config.py`

This file defines the hyperparameters used throughout the project. Key configurations include:

- `ngrams`: Defines the order of the n-gram model (e.g., `{"order": 3}`).
- Smoothing technique configurations:
  - `no_smoothing`: Configuration for no smoothing.
  - `add_k`: Configuration for Add-k smoothing (e.g., `{'k': 1.0}`).
  - `stupid_backoff`: Configuration for Stupid Backoff smoothing (e.g., `{'alpha': 0.4}`).
  - `good_turing`: Configuration for Good Turing smoothing.
  - `interpolation`: Configuration for Interpolation smoothing (e.g., `{'lambdas': [0.7, 0.3]}`).
  - `kneser_ney`: Configuration for Kneser-Ney smoothing (e.g., `{'discount': 0.75}`).
- `error_correction`: Configuration for the misspelling error correction model, including the internal n-gram configuration and error model parameters.
- `additional_hyperparameters`: Other hyperparameters, such as the rare word threshold.

2.2 ngram.py - NGramBase Class

The `NGramBase` class provides the foundation for building n-gram language models.

- `__init__(self, n: int = 2, lowercase: bool = True, remove_punctuation: bool = True)`:
  - Initializes the n-gram model with the specified order `n`, whether to convert text to lowercase, and whether to remove punctuation.
  - `self.ngram_counts`: A `defaultdict(int)` to store the counts of each n-gram.
  - `self.context_counts`: A `defaultdict(int)` to store the counts of each n-gram context (n-1 gram).
  - `self.total_counts`: Stores the total number of n-grams seen during training.
- `method_name(self) -> str`: Returns the name of the method.
- `fit(self, data: List[List[str]]) -> None`:
  - Takes a list of tokenized sentences as input.
  - Iterates through each sentence and counts the occurrences of each n-gram and its context.
  - Populates `self.ngram_counts` and `self.context_counts`.
- `tokenize(self, text: str) -> List[str]`:
  - Splits the input text into sentences based on periods and question marks.
- `prepare_data_for_fitting(self, data: List[str], use_fixed = True) -> List[List[str]]`:
  - Prepares the raw text data into a format suitable for fitting the model.
  - Applies preprocessing and tokenization steps.
- `update_config(self, config) -> None`:
  - Updates the current configuration of the class.
- `preprocess(self, text: str) -> str`:
  - Converts the input text to lowercase.
- `fixed_preprocess(self, text: str) -> str`:
  - Converts the input text to lowercase and removes punctuation.
- `fixed_tokenize(self, text: str) -> List[str]`:
  - Splits the input text into tokens based on whitespace.
- `perplexity(self, text: str) -> float`:
  - Calculates the perplexity of the given text based on the n-gram model.
  - Tokenizes and preprocesses the text.
  - Calculates the probability of each n-gram in the text.
  - Returns the perplexity score.
- `probability(self, ngram: Tuple[str, ...]) -> float`:
  - Calculates the probability of a given n-gram.
  - Uses the counts stored in `self.ngram_counts` and `self.context_counts` to estimate the probability.

2.3 `smoothing_classes.py` - Smoothing Techniques

This file implements various smoothing techniques as subclasses of `NGramBase`.

- `NoSmoothing(NGramBase)`: Implements raw MLE estimation without any smoothing.
  - `probability(self, ngram: Tuple[str, ...]) -> float`: Returns the raw MLE probability of the given n-gram.
- `AddK(NGramBase)`: Implements Add-k smoothing.
  - `__init__(self, n: int = 2, k: float = 1.0, lowercase: bool = True, remove_punctuation: bool = True)`: Initializes the AddK smoothing with the specified `k` value.
  - `fit(self, data: List[List[str]]) -> None`: Fits the model to the data and calculates the vocabulary size.
  - `probability(self, ngram: Tuple[str, ...]) -> float`: Returns the Add-k smoothed probability of the given n-gram.
- `StupidBackoff(NGramBase)`: Implements Stupid Backoff smoothing.
  - `__init__(self, n: int = 2, lowercase: bool = True, remove_punctuation: bool = True, alpha: float = 0.4)`: Initializes the Stupid Backoff smoothing with the specified `alpha` value.
  - `fit(self, data: List[List[str]]) -> None`: Fits the model to the data and calculates the unigram counts.
  - `probability(self, ngram: tuple) -> float`: Returns the Stupid Backoff smoothed probability of the given n-gram.
- `GoodTuring(NGramBase)`: Implements Good Turing smoothing.
  - `fit(self, data: List[List[str]]) -> None`: Fits the model to the data and calculates the n-gram count distribution.
  - `probability(self, ngram: Tuple[str, ...]) -> float`: Returns the Good Turing smoothed probability of the given n-gram.
- `Interpolation(NGramBase)`: Implements Interpolation smoothing.
  - `__init__(self, n: int = 2, lambdas: Tuple[float] = (0.5, 0.5), lowercase: bool = True, remove_punctuation: bool = True)`: Initializes the Interpolation smoothing with the specified `lambdas` values.
  - `probability(self, ngram: Tuple[str, ...]) -> float`: Returns the Interpolation smoothed probability of the given n-gram.
- `KneserNey(NGramBase)`: Implements Kneser-Ney smoothing.
  - `__init__(self, n: int = 2, discount: float = 0.75, lowercase: bool = True, remove_punctuat bool = True)`: Initializes the Kneser-Ney smoothing with the specified `discount` value.
  - `fit(self, data: List[List[str]]) -> None`: Fits the model to the data and calculates the unigram counts and continuation counts.
  - `probability(self, ngram: Tuple[str, ...]) -> float`: Returns the Kneser-Ney smoothed probability of the given n-gram.

2.4 `spelling_corrector.py` - `SpellingCorrector` Class

The `SpellingCorrector` class implements the misspelling error correction system.

- `__init__(self)`:
  - Initializes the spelling corrector by loading the configuration and initializing the internal n-gram model based on the specified smoothing technique.

- `self.word_probabilities`: A `defaultdict(float)` to store the probabilities of each word in the training data.
- `self.error_probabilities`: A `defaultdict(lambda: defaultdict(float))` to store the error probabilities between misspelled words and correct words.

- `fit(self, data: List[str]) -> None`:
  - Fits the n-gram model and the error model to the training data.
- `fit_error_model(self, data: List[str]) -> None`:
  - Estimates the error probabilities based on the common typos defined in the configuration and the word frequencies in the training data.
- `candidates(self, word: str) -> List[str]`:
  - Generates a list of candidate corrections for the given word based on the error model.
- `correct(self, text: List[str]) -> List[str]`:
  - Corrects the misspelled words in the input text based on the n-gram model and the error model.
  - Uses a noisy-channel approach to select the best candidate correction.

2.5 `main.py`

The `main.py` script orchestrates the entire process.

- `load_data(file_path: str) -> List[str]`: Loads data from a text file.
- `load_misspelling_data(file_path: str) -> List[Tuple[str, str]]`: Loads misspelling data from a file.
- `main()`:
  - Loads the training data and misspelling data.
  - Initializes the `SpellingCorrector`.
  - Trains the spelling corrector.
  - Evaluates the spelling corrector on the misspelling data.
  - Writes the incorrect corrections to "output.txt".

3 Performance Analysis

Due to the lack of explicit evaluation code in the provided files (e.g., a dedicated perplexity evaluation loop or accuracy calculation on a held-out set), a comprehensive performance analysis is challenging. However, we can discuss potential evaluation strategies and expected trends.

- **Perplexity Measurements:** Perplexity can be used to evaluate the language model's performance. Lower perplexity indicates better performance.
  - *N-value Impact:* Increasing the n-gram order (n) typically reduces perplexity on the training data, as the model captures more context. However, it can also lead to overfitting and increased perplexity on unseen data.
  - *Smoothing Technique Impact:* Smoothing techniques are crucial for improving the generalization performance of n-gram models. Techniques like Kneser-Ney and Good-Turing typically outperform simpler techniques like Add-k smoothing.
- **Error Correction Accuracy:** The accuracy of the misspelling correction system can be measured as the percentage of misspelled words that are correctly corrected.
  - The current `main.py` calculates and output this metric to the terminal.

4 Text Generation Examples

The provided code does not include explicit text generation functionality. However, an n-gram model can be used to generate text by sampling the next word based on the probabilities predicted by the model.

Example (Based on the `output.txt` line):

- **Input (Incorrect Text):** *they ran away to get married in a little country called lawton was a huge house with seventeen rooms and in it lived mother father maids and four children the eldest was jenny who was 18 years of age she had 3 other brothers and sisters who were all under the age of 12 jenny had a boyfriend who live a few yards a way from her his name was johnny he was 20 years of age*
- **Output (Corrected Text):** *they ran away to get married in a little country called lawton was a huge house with seventeen rooms and in it lived mother father maids and four children the eldest was jenny who was 18 years of age she had 3 other brothers and sisters who were all under the age of 12 jenny had a boyfriend who live a few yards a way from her his name was johnny he was 20 years of age*

5 Misspelling Correction Model Explanation

The misspelling correction model implemented in the `SpellingCorrector` class uses a noisy-channel approach:

$$\text{Corrected Word} = \underset{\text{candidate}}{\operatorname{argmax}} \; P(\text{candidate}) \cdot P(\text{word} \mid \text{candidate})$$

Where:

- `candidate` is a possible correction for the misspelled word.
- $P(\text{candidate})$ is the language model probability of the candidate word in its context.
- $P(\text{word} \mid \text{candidate})$ is the error model probability of observing the misspelled word given that the correct word is the candidate.

5.1 Error Model

The error model estimates the probability that a given word is misspelled and quantifies the likelihood of specific typo-to-correction transitions.

- **Implementation:** The `fit_error_model` method in `SpellingCorrector` estimates error probabilities based on `error_correction['error_model']['common_typos']` in `config.py`. It assigns a fixed `typo_probability` to common typos and calculates word probabilities from the training data.
- **Limitations:** The current error model is quite basic and only considers a limited set of common typos.

5.2 Candidate Generation Strategy

The `candidates` method generates a list of candidate corrections for a given word.

- **Implementation:** The current implementation retrieves candidates directly from the `self.error_probabilities` dictionary, which is populated during error model fitting. If the word is not in `self.error_probabilities`, it returns the original word as the only candidate.
- **Limitations:** The candidate generation strategy is limited to the typos explicitly defined in `config.py`.

5.3 Combination and Evaluation

The `correct` method combines the language model probability and the error model probability to select the best candidate correction.

- **Combination:** The language model probability is obtained from the internal n-gram model (`self.internal_ngram.probability(ngram)`). The error model probability is obtained from `self.error_probabilities`. These probabilities are multiplied together to obtain a score for each candidate.
- **Selection:** The candidate with the highest score is selected as the corrected word.
- **Evaluation:** The `main.py` prints the incorrect corrections to the terminal.

6 Conclusion

The implemented n-gram language model and misspelling correction system provide a foundation for further development. Future work could focus on:

- Implementing more sophisticated error models (e.g., based on edit distance).
- Improving the candidate generation strategy (e.g., using a dictionary or phonetic similarity).
- Adding more comprehensive evaluation metrics.
- Implementing text generation functionality.