N-grams and Misspelling Correction
ELL884 Assignment 1

February 14, 2025
Shreyas Shimpi
2024AIB2291

---

1 Performance Analysis of N-gram Models and Smoothing Techniques

For this evaluation, we used perplexity as our metric. This intrinsic evaluation method requires a test dataset to compare different models. We selected two sentences for testing and calculated perplexity scores by varying the parameters. Since perplexity is computed as the inverse of the probability assigned to the test corpus, a lower perplexity indicates a better-performing model.

The perplexity of the model on a test sequence is defined as:

$$\text{Perplexity}(W) = \sqrt[N]{\frac{1}{P(W)}} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\ln P(w_i \mid w_{i-n+1}^{i-1})\right)$$

A lower perplexity indicates that the model better predicts the test data.

Test Case - Perplexity Calculation

Test Sentence: *Ron Slammed his foot on accelerator*

Smoothing Technique	**Order = 1**	**Order = 2**	**Order = 3**
No Smoothing	1262.86	∞	∞
Add_k (k=1)	1268.54	8078.78	20472.80
Stupid Backoff	1262.86	433.00	361.06
Good Turing	1262.86	47213.34	240.09
Interpolation	1262.86	594.28	777.64
KneserNey	58439.00	703.73	552.72

Table 1: Perplexity Calculation for Test Case

2 Examples of Text Generated by the Model

In these examples, some misspellings are corrected while others remain uncorrected.

Corrupt	Truth	Prediction
he warked about a mile	he walked about a mile	he walked about a mile
they live happly ever after	they live happily ever after	they live happily ever after
3 triumh	3 triumph	3 triumph
and then he went on his motorbikes	and then he went on his motorbike	and then he went on his motorbike
the and	the end	the and

Table 2: Examples of Corrupt, Truth, and Predicted Text

3 Explanation of the Probabilistic Misspelling Error Correction Model

This model is implemented in the `SpellingCorrector` class and integrates vocabulary expansion, candidate generation based on edit distance, and correction using an n-gram language model with configurable smoothing. The following subsections describe each component of the implementation.

This comprehensive vocabulary is crucial for determining which words are correctly spelled and which ones might need correction.

3.1 N-Gram Language Model Initialization

The spelling corrector incorporates an n-gram language model that is responsible for evaluating the likelihood of word sequences. The initialization process includes the following steps:

- A configuration dictionary is used to select the smoothing method to be applied. The available methods include no smoothing, add-k, stupid backoff, Good-Turing, interpolation, and Kneser-Ney.
- Based on the selected method, the corresponding smoothing class is instantiated and configured.
- The internal n-gram model then extracts its vocabulary from the training data, which becomes the reference set for identifying correctly spelled words.

3.2 Model Fitting

To effectively evaluate sentence likelihoods, the language model must first be trained on a set of sentences. The training process includes:

- Preprocessing and tokenizing the training sentences using the same methods as used during vocabulary creation.
- Converting the processed data into a suitable format for the n-gram model.
- Fitting the model on the processed data so that it learns the probability distribution of word sequences.

Once trained, the model is capable of computing metrics such as perplexity, which is used to assess how likely a given sentence is according to the learned language patterns.

3.3 Spelling Correction Logic

The correction mechanism operates in several key steps:

1. **Tokenization and Baseline Evaluation:** An input sentence is first preprocessed and tokenized in a manner consistent with the training phase. The model then calculates a baseline perplexity for the original sentence.
2. **Identifying Misspelled Words:** Each token is compared against the internal vocabulary. If a token is not found in the vocabulary, it is flagged as potentially misspelled.
3. **Generating Candidate Corrections:** For each potentially misspelled word, the system generates candidate corrections by producing words that are within a certain edit distance (using deletion, insertion, replacement, and transposition operations).
4. **Perplexity Evaluation:** Each candidate is substituted back into the sentence, and the modified sentence's perplexity is computed. A lower perplexity indicates that the candidate word fits better within the context of the sentence.
5. **Selection of the Best Candidate:** The candidate that leads to the greatest improvement (i.e., the lowest perplexity) is chosen as the correction for the misspelled word.

This process is repeated for each token in the sentence, ensuring that the corrected sentence has the highest possible likelihood according to the language model.

4 Experimentation with Other Techniques

This section outlines the additional heuristics integrated into the model, including the use of Jaccard similarity and phonetic similarity for candidate ranking.

4.1 Heuristics Overview

After incorporating additional heuristics—such as **context windows, frequency bonuses, stricter edit distance thresholds**—the revised implementation achieved a score of:

- **Score:** 0.8132284726688362

4.2 Jaccard Similarity on Character n-grams

This section computes the Jaccard similarity between character n-gram sets. The following code implements the n-gram generation and Jaccard similarity.

```
def get_ngrams(word, n=2):
    """Return a set of character n-grams for the given word."""
    if len(word) < n:
        return {word}
    return {word[i:i+n] for i in range(len(word) - n + 1)}

def jaccard_similarity(query, candidate, n=2):
    """Compute Jaccard similarity between n-gram sets of query and
       candidate."""
    q_ngrams = get_ngrams(query, n)
    c_ngrams = get_ngrams(candidate, n)
    intersection = q_ngrams & c_ngrams
    union = q_ngrams | c_ngrams
    return len(intersection) / len(union) if union else 0.0
```

Listing 1: Jaccard Similarity on Character n-grams

4.3 Phonetic Similarity (Soundex)

Phonetic similarity is computed using a simple Soundex algorithm. The code snippet below shows the Soundex implementation and how two words are compared.

```
def soundex(word):
    """Return the Soundex code for the given word."""
    if not word:
        return "0000"
    word = word.upper()
    codes = {
        "B": "1", "F": "1", "P": "1", "V": "1",
        "C": "2", "G": "2", "J": "2", "K": "2", "Q": "2", "S": "2", "X": "
           2", "Z": "2",
        "D": "3", "T": "3",
        "L": "4",
        "M": "5", "N": "5",
        "R": "6"
    }
    first_letter = word[0]
    encoded = [codes.get(ch, "") for ch in word[1:]]
    filtered = []
```

```
    prev = ''
    for code in encoded:
        if code != prev and code != "":
            filtered.append(code)
        prev = code
    soundex_code = first_letter + "".join(filtered)
    return (soundex_code + "000")[:4]

def phonetic_sim(query, candidate):
    """Return 1.0 if Soundex codes match, else 0.0."""
    return 1.0 if soundex(query) == soundex(candidate) else 0.0
```

Listing 2: Phonetic Similarity (Soundex)

4.4 Final Score Calculation

In addition to the individual similarity measures, the final candidate score is computed as a weighted sum of the heuristic scores:

$$\text{FinalScore} = w_{\text{jaccard}} \cdot \text{JaccardSim} + w_{\text{phonetic}} \cdot \text{PhoneticSim} + w_{\text{ngram}} \cdot \text{NGramProb}$$

with weights:

- $w_{\text{jaccard}} = 0.45$
- $w_{\text{phonetic}} = 0.25$
- $w_{\text{ngram}} = 0.30$

The candidate with the highest FinalScore is selected.

4.5 Results (Using Jaccard and Phoenetics Similarity)

The final implementation achieved a score of **0.83** as per `grading.py`.

5 Conclusion

In this assignment, we presented a comprehensive approach to n-gram language modeling and probabilistic misspelling correction. Our methodology combined vocabulary expansion, dynamic candidate generation via edit distance, and context-aware correction using various smoothing techniques. Evaluations based on perplexity demonstrated that smoothing methods significantly impact performance—with techniques such as No Smoothing and Add_k yielding competitive results.

The experiments highlighted the balance between model complexity and correction accuracy. Although additional heuristics (e.g., context windows, frequency bonuses, stricter edit distance thresholds) provided valuable insights, the baseline configurations often produced superior results. Future enhancements may include exploring more sophisticated smoothing techniques and candidate ranking algorithms to further improve context-sensitive error correction.