

Spell Corrector

1. Introduction

For our Natural Language Processing course, we undertook the project of developing a spell corrector. The goal was to create a Python program capable of identifying misspelled words within a text and proposing probable corrections. This endeavor allowed us to delve into fundamental NLP techniques, particularly those related to text normalization and probabilistic language modeling.

The design and implementation of our spell corrector were significantly inspired by Peter Norvig's well-documented approach. However, we also drew upon broader principles in the field of computational linguistics and automatic spelling correction, incorporating concepts such as edit distance and probabilistic scoring.

2. Methodology and Implementation

The core idea behind our spell corrector is to generate a set of potential corrections for a given word and then select the most probable candidate from this set. This process relies heavily on a corpus for statistical information and a defined set of edit operations.

2.1. Corpus Utilization

We used the "big.txt" file as the corpus for our spell corrector. This large text file provides the statistical foundation for determining word likelihoods.

- **Loading and Processing:** we implemented the `load_corpus` method to read the corpus file. This method utilizes another function we wrote, `words`, which tokenizes the text into a list of lowercase words using regular expressions (`re.findall(r'\w+', text.lower())`). The word frequencies are stored in a `collections.Counter` object named `self.WORDS`.
- **Word Probabilities:** The total number of words in the corpus is stored in `self.TOTAL_WORDS`. This count is crucial for the probability method, which calculates $P(\text{word}) = \frac{\text{TOTAL_WORDScount}(\text{word})}{\text{TOTAL_WORDS}}$ for any given word. This simple unigram probability serves as a measure of how common a word is.

2.2. Candidate Generation

To find potential corrections, we generate candidates at edit distances 1 and 2 from the input word. Edit distance refers to the minimum number of single-character edits (insertions, deletions, substitutions) required to change one word into the other.

- **edits1 Function:** Our `edits1` method generates all possible strings that are one

edit away from the input word. The edits we included are:

- **Deletions:** Removing a character (e.g., "apple" -> "aple").
- **Transpositions:** Swapping two adjacent characters (e.g., "apple" -> "aplep").
- **Replacements:** Changing one character for another (e.g., "apple" -> "apply").
- **Insertions:** Adding a character (e.g., "apple" -> "appple"). These operations are standard in spell correction and are components of measures like the Damerau-Levenshtein distance, which specifically includes transpositions as a single edit operation. Damerau noted that a large percentage of human spelling errors involve a single instance of one of these four error types.

A distinctive feature of our edits1 implementation is the assignment of custom costs to each edit type:

- Deletions: Cost 1.0
 - Transpositions: Cost 0.8
 - Replacements: Cost 0.9
 - Insertions: Cost 1.2 We chose these costs to reflect our hypothesis about the relative likelihood of these errors, with transpositions being the 'cheapest' (most common) and insertions the 'most expensive' (least common).
- **edits2 Function:** To expand the search space, our edits2 method generates candidates that are two edits away. It does this by applying edits1 to all the candidates generated by the first edits1 pass. The cost of a two-edit candidate is the product of the costs of the two individual edits.

2.3. Candidate Selection

Once candidates are generated, they need to be evaluated and the best one selected. This is handled by our candidates function.

- **Prioritization Strategy:**
 1. If the input word itself is in our corpus (self.WORDS), we assume it's correct and return it with its probability.
 2. Otherwise, we generate candidates using edits1(word).
 3. These candidates are then passed to our known method, which filters out any words not present in the corpus and calculates a score for the known ones. The score is defined as $\text{score} = \text{probability}(\text{word}) / \text{cost}$. This formula is our way of combining the word's general frequency (probability) with the likelihood of the specific edits needed to reach it (represented by the inverse of the cost). Candidates are then sorted by this score in descending order.
 4. If no known candidates emerge from edits1, we repeat the process using edits2(word).
 5. If no candidates are found even after edits2, we return the original input word

with its probability (which will be low or zero if it's not in the corpus).

- **Word Correction:** The final correction is determined by the correction method, which simply returns the candidate with the highest score from the list provided by candidates.

2.4. Sentence Correction

To make the tool more practical, we implemented a `correct_sentence` method. This method tokenizes an input sentence, applies the correction logic to each word (after converting to lowercase), and reassembles the sentence.

3. Theoretical Foundations

Our spell corrector is grounded in well-established principles in NLP and information theory.

- **Probabilistic Approach (Norvig):** The overarching strategy aligns with Peter Norvig's model. The goal is to find a correction c for a misspelled word w that maximizes the probability $P(c|w)$. Using Bayes' Theorem, this is expressed as:
$$P(c|w) = P(w)P(w|c)P(c)$$
Since $P(w)$ is constant for all candidates c , we aim to maximize $P(w|c)P(c)$.
- **Language Model ($P(c)$):** In our system, $P(c)$ (the prior probability of the candidate word) is estimated by its frequency in the corpus: `self.WORDS[c] / self.TOTAL_WORDS`. This is a simple unigram language model. More complex models like n -grams could also be used here.
- **Error Model ($P(w|c)$):** The term $P(w|c)$ represents the probability that the typo w would be produced if the user intended to write c . Norvig's original model gives priority based on edit distance (0, then 1, then 2) and then by $P(c)$. Our implementation refines the error model by introducing varying costs for different edit types within `edits1`. The scoring function probability / cost means that candidates generated by lower-cost (more likely) edits receive higher scores, effectively giving a higher $P(w|c)$ to such transformations. This approach is a way of modeling the "noisy channel" through which the correct word c passes to become the misspelled word w . The goal of the noisy channel model in spelling correction is to find the most probable intended word c , given the observed (possibly erroneous) word w .
- **Edit Distance:** The concept of edit distance is central. Our `edits1` function, which includes deletions, insertions, substitutions, and transpositions, covers the operations considered by the Damerau-Levenshtein distance for an edit distance of 1. Historically, Damerau found that over 80% of spelling errors are the result of a single error of one of these four types, making edit distance 1 (and

subsequently 2) a practical focus for spell correctors.

4. Testing and Evaluation

To ensure the reliability and correctness of our spell corrector, we developed a comprehensive suite of unit tests using Python's unittest framework, contained in `test_spell.py`.

- **Key Test Areas:**

- `test_known_words`: Verifies that known words are correctly identified.
- `test_edits1_contains_known_word`: Ensures `edits1` can generate a known correction (e.g., "speling" to "spelling").
- `test_edits1_weights`: Checks if the assigned edit costs are floats and within a reasonable range.
- `test_correction_typo`: Tests specific typo corrections like "korrektud" to "corrected" and "bicycle" to "bicycle".
- `test_correction_with_stats_output`: Validates the output structure and logic, ensuring the best candidate has the highest score.
- `test_corpus_stats`: Confirms that corpus statistics (total words, unique words) are plausible.
- `test_empty_input`: Checks behavior with an empty sentence.

These tests were invaluable during development for catching bugs and remain essential for verifying functionality if we make future modifications.

5. Results and Demonstration

Our spell corrector performs as expected for a range of common misspellings.

- **Corpus Insights:** The `corpus_stats` method provides a quick overview of the language model. For example, it shows the total words (e.g., over 1 million for 'big.txt'), unique words (e.g., tens of thousands), and the most frequent words. This helps confirm the corpus has loaded correctly.

Corpus statistics:

Total words : 1,115,585

Unique words: 29,157

Top 10 most frequent words:

the : 79,809

of : 40,024

and : 38,312

...

- **Word Correction Example:** For an input like "korrektud", our system suggests "corrected". The `correction_with_stats` method can show the candidate list:

Input word: korrektud

Suggested correction: corrected

Candidate corrections with scores (probability / cost):

Word	Score
------	-------

corrected	0.00001703
-----------	------------

correct	0.00000506
---------	------------

...

- **Sentence Correction Example:**

Original sentence:

This is a smple sentence with som misspelled wrds.

Corrected sentence:

This is a simple sentence with some misspelled words.

(Note: "Ths" -> "This" is a good correction. "is" and "a" are already correct. "smple" -> "simple", "sentence" -> "sentence", "som" -> "some", "wrds" -> "words" are all successful corrections by our program).

6. Discussion and Future Work

We are pleased with the functionality of this spell corrector, which effectively handles many common spelling errors based on the principles we implemented. The use of custom edit costs is an interesting refinement that allows for a more nuanced error model than simply relying on edit distance alone.

However, there are several limitations and areas for future improvement:

- **Context Insensitivity:** Our corrector evaluates words in isolation. It cannot distinguish between homophones based on context (e.g., "there" vs. "their" vs. "they're") if the misspelled word could be validly corrected to any of them with similar scores. More advanced systems use n-gram models or other contextual language models to address this.
- **Fixed Edit Costs:** The edit costs in `edits1` are currently hand-tuned. These could potentially be learned from a corpus of common misspellings and their corrections to create a more data-driven error model.
- **Limited Edit Distance:** For practical performance, our corrector primarily relies on edit distances 1 and 2. More complex errors might not be caught.
- **Real-Word Errors:** The system is mainly designed for non-word error correction

(i.e., typos that result in non-existent words). Detecting real-word errors (where a valid but incorrect word is used) is a more challenging task that often requires syntactic and semantic analysis.

- **Phonetic Errors:** Errors like "fone" for "phone" might be missed if their edit distance is large. Incorporating phonetic matching algorithms like Soundex or Metaphone could help address these. Kukich (1992) discusses various techniques, including phonetic similarity and the use of n-grams for both isolated-word and context-dependent correction.

Future work could involve exploring these areas, such as integrating a trigram model for contextual scoring or implementing a module for learning edit probabilities.

7. Conclusion

Developing this spell corrector has been a valuable learning experience. We successfully implemented a system based on Peter Norvig's probabilistic framework, enhanced with custom edit costs and grounded in established NLP concepts like edit distance and the noisy channel model. The program demonstrates proficiency in correcting a variety of common spelling errors. While there are avenues for further sophistication, this project serves as a solid foundation and a practical demonstration of core spell-checking techniques.

8. References

1. Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3), 171-176.²
2. Brill, E., & Moore, R. C. (2000). An improved error model for noisy channel spelling correction. *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*,³ 286-293.
3. Norvig, Peter. "How to Write a Spelling Corrector." *norvig.com*, Accessed May 25, 2025. Available at: <https://www.norvig.com/spell-correct.html>
4. Jurafsky, D., & Martin, J. H. (2023). *Speech and Language Processing* (3rd ed. draft). Chapters 2 & 3. Accessed May 25, 2025. Available at: <https://web.stanford.edu/~jurafsky/slp3/>
5. Kukich, K. (1992). Techniques for automatically correcting words in text. *ACM Computing Surveys (CSUR)*, 24(4), 377-439.