

Development of an English Spell Checker

submitted by

Anwar Islam

Class Roll No: 002210503014

Exam Roll: MCA244014

Master of Computer Application

under the supervision of

Dr. Sudip Kumar Naskar

in the

Department of Computer Science and Engineering

Faculty of Engineering and Technology

Jadavpur University

2024

Faculty of Engineering and Technology
Jadavpur University

CERTIFICATE OF RECOMMENDATION

To Whom It May Concern

I hereby recommend that the project report entitled “Development of an English Spell Checker” has been carried out by Anwar Islam (Reg. No.: 163606 of 2022-23, Roll No.: 002210503014, Exam Roll: MCA244014), under my guidance and supervision and be accepted in partial fulfilment of the requirement for the degree of Master of COMPUTER APPLICATION in DEPARTMENT of COMPUTER SCIENCE and ENGINEERING, JADAVPUR UNIVERSITY during the academic year 2023-2024.

Dr. Sudip Kumar Naskar
Project Supervisor
Dept. of Computer Science and
Engineering
Jadavpur University

Prof. (Dr.) Nandini Mukhopadhyay
Head of the Department
Dept. of Computer Science & Engineering
Jadavpur University

Prof. Dipak Laha
Dean, Faculty Council
of Engineering & Technology
Jadavpur University, Kolkata-032

Faculty of Engineering and Technology
Jadavpur University

CERTIFICATE OF APPROVAL

This is to certify that the project report entitled “**Development of an English Spell Checker**” is a bonafide record of work carried out by Anwar Islam, in fulfillment of the requirements for the award of the degree of Master of Computer Application in the Department of Computer Science and Engineering, Jadavpur University. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the project report only for the purpose for which it has been submitted.

Signature of Examiner 1

Date :

Signature of Examiner 2

Date :

Faculty of Engineering and Technology
Jadavpur University

**DECLARATION OF COMPLIANCE WITH
ACADEMIC ETHICS**

I hereby declare that this project report entitled " Development of an English Spell Checker " is an outcome of my genuine and independent work, as part of my Master of Computer Application program.

All information in this document has been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, all materials and results that are not original to this work have been fully cited and referenced.

Anwar Islam

Registration No.: 163606 of 2022-23

Examination Roll No.: MCA244014

Thesis Title: - Development of an English Spell Checker

Signature with Date

ACKNOWLEDGEMENT

With my most sincere and gratitude, I would like to thank Dr. Sudip Kumar Naskar, Department of Computer Science & Engineering, my supervisor, for his overwhelming support throughout the duration of the project. His motivation always gave me the required inputs and momentum to continue with my work, without which the project work would not have taken its current shape. His valuable suggestions and numerous discussions have always inspired new ways of thinking. I feel deeply honored that I got this opportunity to work under him.

I would like to express my sincere thanks to all my teachers for providing sound knowledge base and cooperation.

I would like to thank all the faculty members of the Department of Computer Science & Engineering of Jadavpur University for their continuous support.

Anwar Islam

Class Roll No.: 002210503014

Examination Roll No.: MCA244014

Department of Computer Science & Engineering

Jadavpur University

Abstract

Spelling correction is an important part of modern word processing systems, helping to accurately interpret user input. This thesis explores the development of a contextual, edit distance corrector built on foundational computational linguistics techniques. The work is organized around the noise channel model, a complex framework that corrects typographical errors by leveraging the likelihood of specific character permutations alongside prior word probabilities.

Initially, we define practices and apply them to perform possible corrections for deletion, insertion, substitution errors, and transposition errors. This function is validated using English vocabulary definitions obtained from the NLTK corpus to ensure that candidate disciplines are properly English words. At the core of our spell corrector is a noise channel model, which combines possible typos with unigram probabilities derived from a large text corpus. This model is enhanced by the inclusion of a bigram language model, which considers context around typos in a sentence to improve the accuracy of correction.

Our approach includes constructing two grams of frequency cells and using Laplace smoothing to compute conditional probabilities. The final spelling correction task combines these possibilities to determine the most likely correction for a given typo by considering the noise channel and context.

The analysis highlights the effectiveness of the system in correcting isolated words and spelling errors in sentences. However, it also highlights limitations, such as the need for extensive corpus data, the potential for contextual confusion in ambiguity. This thesis makes assumptions about the development process and concludes with recommendations for future improvements to improve the accuracy and usability of the maintenance script.

This review contributes to a foundational understanding of the spelling correction tool and provides a practical application that balances simplicity and efficiency, and paves the way for advanced text processing and recognition tools the relevant information.

Contents

Introduction	10
1.1 Background	10
1.2 Problem Statement.....	10
1.3 Objectives.....	11
Literature Survey	12
2.1 Early Approaches.....	12
2.2 Statistical Models.....	12
2.3 Contexts-Aware Systems.....	12
2.4 Machine Learning and Neural Networks.....	12
2.5 Evaluation Methods.....	13
2.6 Current Trends and Challenges.....	13
Methodology	14
3. Data Acquisition and Pre-processing	14
3.1 Obtain a Lexical Resource.....	14
3.2 Read and Clean Corpus.....	14
3.3 Merge Corpora.....	15
System Description	17
4.1 Error Model Definition and Implementation	17
4.1.1 Deletion Errors.....	18
4.1.2 Insertion Errors.....	18
4.1.3 Substitution Errors.....	19
4.1.4 Transposition Errors.....	20
4.2 Noisy Channel Model and Basic Word Correction	21
4.2.1 Define Noisy Channel Model.....	21
4.2.1 Test Basic Word Correction.....	23

Experiment and Results	25
5. Error Model/Likelihood Calculation	25
5.1 Extracting and Cleaning the Typo Corpus.....	25
5.2 Making Error-Specific Corpora.....	26
5.3 Count Character Permutations.....	27
5.4 Apply Laplace Smoothing.....	27
5.5 Calculate Conditional Probabilities.....	28
5.6 Compute Noisy Channel Probability.....	29
5.7 Identify Best Candidates.....	29
6. Noisy Channel Spelling Corrector	31
6.1 Identifying the Best Candidate for a Typo.....	31
6.1.1 Function Definition.....	31
6.1.2 Initialize Dictionary.....	31
6.1.3 Handle Different Error Types.....	31
6.1.4 Sort Candidates.....	32
6.1.5 Return Best Candidate.....	32
6.1.6 Error Handling.....	32
6.2 User Input Version.....	32
6.3 Correcting Typos in a Sentence.....	32
6.3.1 Spot Typos.....	32
6.3.2 Correct Each Typo.....	33
6.3.3 Error Handling.....	33
6.3.4 Return Corrected Sentence.....	33
7. Integrating Context	35
7.1 Creating Bigrams and a Bigram Corpus.....	35
7.1.1 Bigram Corpus Function.....	35
7.1.2 Corpus Preparation.....	35
7.1.3 Create Bigrams.....	36
7.2 Bigram Frequency and Smoothed Probabilities.....	36
7.2.1 Bigram Frequency Dictionary.....	36

7.2.2 Word Frequency Calculation.....	37
7.2.3 Smoothed Probabilities.....	37
7.3 Final Function for Context-Aware Spell Correction.....	38
7.3.1 Noisy Channel Probability.....	38
7.3.2 Bigram Probability.....	39
7.3.3 Combine Probabilities.....	39
7.3.4 Error Handling.....	40
7.3.5 Return Correct Sentence.....	40
Conclusion and Future Work	41
8. Evaluation and Conclusion	41
8.1 Problems.....	41
8.2 Short Coda.....	42
Reference	44

Introduction

Chapter 1

1.1 Background

Spelling correction is a fundamental aspect of text processing systems, enhancing both the clarity and quality of digital communication. As digital text becomes increasingly prevalent, the necessity for accurate and efficient spelling correction mechanisms has grown. Traditional spelling correctors, which focus on basic word-level corrections, laid the groundwork for contemporary systems that offer real-time corrections and personalized suggestions. However, these advanced systems still rely on the basic principles of spelling correction, which remain critically important.

This project revisits these foundational principles to create a context-sensitive, one-edit distance spelling corrector. By delving into the core processes of detecting and correcting typographical errors, the project aims to enhance understanding of the fundamental techniques underlying more sophisticated modern systems. The motivation behind this work is to refine these core techniques to improve current technologies and inform future developments in spelling correction.

1.2 Problem Statement

Typographical errors, or typos, are a common occurrence in written text, resulting from fast typing, inattention, or unfamiliarity with correct spelling. These errors can impede communication and reduce the clarity of the text. While existing spelling correction tools are generally effective, they often rely on large datasets and complex algorithms, which may not be practical in resource-constrained environments. Additionally, these tools

might not always consider the contextual relevance of words, leading to incorrect corrections.

This project aims to develop a spelling corrector that effectively identifies and corrects typographical errors within a context-aware framework. Specifically, it focuses on corrections that are one edit distance away from the original word, encompassing deletions, insertions, substitutions, and transpositions. The goal is to create a system that combines basic spelling correction techniques with sophisticated contextual analysis to improve the accuracy and relevance of corrections.

1.3 Objectives

The primary objective of this project is to develop a spelling corrector that leverages the noisy channel model to correct typos effectively while considering the context in which the errors occur. The specific objectives include:

1. **Defining and Implementing Basic Error Correction Functions:** Develop functions to handle common typographical errors, including deletions, insertions, substitutions, and transpositions.
2. **Creating an English Lexicon for Validation:** Utilize a lexical resource to ensure that the candidate corrections are valid English words.
3. **Applying the Noisy Channel Model:** Integrate prior word probabilities and error likelihoods to determine the most probable corrections.
4. **Building a Bigram Language Model:** Use bigram probabilities to incorporate context into the correction process, enhancing the accuracy of corrections.
5. **Evaluating the System:** Test the spelling corrector on various text samples to assess its performance and identify areas for improvement.
6. **Addressing Limitations and Future Work:** Analyze the limitations of the current system and suggest potential enhancements for future research.

Literature Survey

Chapter 2

2.1 Early Approaches

The earliest spell checkers were rule-based systems, relying on predefined rules to identify and correct spelling errors. These systems often used a dictionary look-up approach, flagging words not found in the dictionary as errors and suggesting similar words based on simple algorithms like Levenshtein distance.

2.2 Statistical Models

The development of statistical models marked a significant advancement in spell checking. These models use large corpora to calculate the probability of a word or phrase, enabling the correction of more complex errors. The noisy channel model is a prominent statistical approach, treating spelling correction as a process of decoding a "noisy" (erroneous) input to recover the original intended text.

2.3 Contexts-Aware Systems

More recent approaches incorporate context to improve correction accuracy. These systems analyze the surrounding text to determine the most likely correction for a misspelled word. Techniques such as bigrams and trigrams, which consider pairs or triplets of words, help in understanding context and improving suggestions. This is crucial for resolving ambiguities where a simple dictionary-based approach might fail.

2.4 Machine Learning and Neural Networks

The advent of machine learning and neural networks has further revolutionized spell checking. These approaches learn from large datasets of text and corrections, improving over time as they encounter more data. Neural networks, especially deep learning models, can understand complex patterns and dependencies in text, making them highly effective for spelling correction tasks.

2.5 Evaluation Methods

Evaluating spell checkers involves measuring their accuracy in identifying and correcting errors. Common metrics include precision, recall, and F1 score, which assess the system's performance in terms of correctly identified errors and accurate corrections. Benchmark datasets and standardized evaluation protocols are used to compare different systems objectively.

2.6 Current Trends and Challenges

Current research trends focus on integrating advanced natural language processing (NLP) techniques and developing multilingual spell checkers. Challenges remain in handling diverse error types, including those arising from non-native speakers, and ensuring the systems are robust across various domains and text types.

Methodology

Chapter 3

3. Data Acquisition and Pre-processing

Data acquisition and pre-processing are foundational steps in the data science workflow, setting the stage for effective analysis and model building.

3.1 Obtain a Lexical Resource

A lexical resource is a collection of linguistic information that is used for language processing tasks. Examples include dictionaries, thesauruses, word lists, and specialized databases like WordNet. These resources provide essential information about words, such as their meanings, synonyms, antonyms, part of speech, and usage examples.

Steps:

- 1. Identify the Required Resource:** Depending on your project, you might need a general-purpose dictionary (like WordNet) or a domain-specific glossary.
- 2. Access or Download :** Many lexical resources are available for free online. For instance, WordNet can be downloaded and accessed via libraries like NLTK in Python.
- 3. API Access:** Some lexical resources are accessible via APIs. For example, the Merriam-Webster dictionary provides API access for programmatic queries.

3.2 Read and Clean Corpus

A corpus is a large collection of texts used for linguistic research and NLP. Cleaning the corpus involves preparing the raw text data for analysis by removing or correcting any issues.

Steps:

1. Read the Corpus:

Load Data: Use appropriate libraries to read the text data. For instance, you can use Python's `open()` function for reading plain text files or libraries like `pandas` to read structured text data.

Encoding Handling: Ensure that the text is read with the correct encoding (e.g., UTF-8) to avoid issues with special characters.

2. Clean the Corpus:

Lowercasing: Convert all text to lowercase to ensure uniformity (e.g., "Apple" and "apple" are treated the same).

Remove Punctuation: Strip punctuation marks unless they are important for your analysis.

Remove Stop Words: Remove common words that do not contribute much to meaning (e.g., "and", "the").

Tokenization: Split text into individual words or tokens. This can be done using libraries like NLTK or spaCy.

Stemming/Lemmatization: Reduce words to their root form. Stemming chops off the end of words (e.g., "running" to "run"), while lemmatization uses a dictionary to find the base form (e.g., "better" to "good").

Handling Special Characters: Remove or replace special characters, numbers, and other non-text elements.

3.3 Merge Corpora

Merging corpora involves combining multiple text datasets into a single, unified corpus. This is necessary when dealing with data from multiple sources or domains.

Steps:

- 1. Data Compatibility:** Ensure that all corpora to be merged are compatible in terms of language, encoding, and format.
- 2. Standardization:** Pre-process each corpus similarly to ensure consistency. This includes applying the same cleaning steps to each dataset.
- 3. Combine Data:** Append or concatenate the text data from different corpora. In Python, this can be done using list operations or by combining Data Frames in pandas.
- 4. De-duplication:** Remove duplicate entries that might have resulted from overlapping corpora.
- 5. Final Cleaning:** Perform a final round of cleaning to address any issues introduced during merging.

Chapter 4

4.1 Error Model Definition and Implementation

An error model is a systematic representation of the types and frequencies of errors that can occur in a dataset. Implementing an error model involves identifying, categorizing, and simulating these errors to better understand their impact and develop strategies for detection and correction.



Words within 1 of across

Error	Candidate Correction	Correct Letter	Error Letter	Type
acress	actress	t	-	deletion
acress	cress	-	a	insertion
acress	caress	ca	ac	transposition
acress	access	c	r	substitution
acress	across	o	e	substitution
acress	acres	-	s	insertion
acress	acres	-	s	insertion

16

4.1.1 Deletion Errors

Deletion errors occur when one or more characters are missing from the text. This can happen due to accidental keystrokes, data corruption, or transmission issues.

Examples:

Original: "hello"

With Deletion Error: "helo" (missing 'l')

Implementation:

Identification: Use techniques like string comparison algorithms (e.g., Levenshtein distance) to identify where deletions might have occurred.

Simulation: Randomly remove characters from strings in a controlled manner to understand the impact on text processing tasks.

Correction: Implement error correction methods such as context-based prediction or dictionary lookups to suggest possible corrections.

Steps:

1. **Character-Based Models:** Analyze text at the character level to identify missing characters.
2. **Contextual Analysis:** Use the surrounding context to predict the missing character(s). For example, in "helo", the context suggests the missing character could be 'l'.

4.1.2 Insertion Errors

Insertion errors happen when extraneous characters are added to the text. These errors can be introduced by accidental keystrokes or data noise.

Examples:

Original: "hello"

With Insertion Error: "heylllo" (extra 'y')

Implementation:

Identification: Similar to deletion errors, string comparison algorithms can help spot insertion errors.

Simulation: Introduce random characters into the text to study how insertion errors affect downstream processes.

Correction: Develop algorithms to detect and remove unlikely characters based on context and word frequency.

Steps:

1. **Character-Based Detection:** Identify extra characters using algorithms like Levenshtein distance.

4.1.3 Substitution Errors

Substitution errors occur when one character is mistakenly replaced by another. This is common in manual data entry and OCR (Optical Character Recognition) errors.

Examples:

Original: "hello"

With Substitution Error: "hezlo" ('l' replaced with 'z')

Implementation:

Identification: Utilize confusion matrices that map common substitutions (e.g., 'o' and '0') and use string matching algorithms to detect anomalies.

Simulation: Replace characters in a controlled manner to simulate common substitution errors.

Correction: Apply context-aware models and error dictionaries to suggest corrections.

Steps:

1. **Character Mapping:** Create a mapping of commonly substituted characters (e.g., 'm' with 'n').
2. **Contextual Analysis:** Use the context around the substituted character to predict the intended character.
3. **Machine Learning Models:** Train models on large datasets with known substitution errors to improve detection and correction.

4.1.4 Transposition Errors

Transposition errors involve the swapping of two adjacent characters. These are typical in fast or careless typing.

Examples:

Original: "hello"

With Transposition Error: "hlelo" (swapping 'e' and 'l')

Implementation:

Identification: Identify common transpositions using algorithms designed to detect adjacent character swaps.

Simulation: Swap characters in text to simulate the effect of transposition errors.

Correction: Implement models that recognize common transpositions and suggest corrections based on context.

Steps:

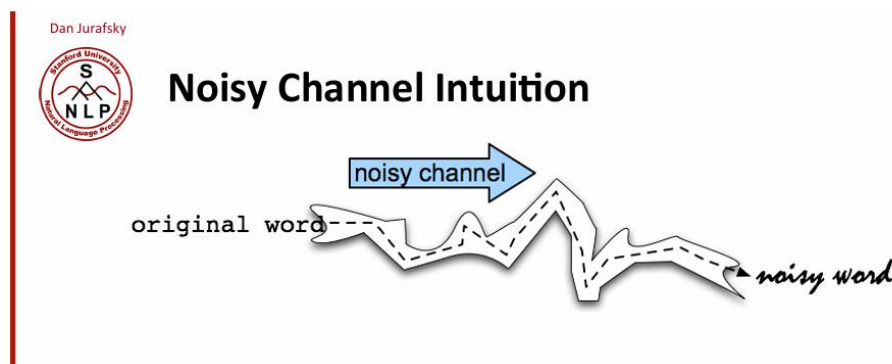
1. **Adjacent Character Analysis:** Focus on detecting swaps between adjacent characters.
2. **Pattern Recognition:** Identify common patterns of transpositions (e.g., 'teh' instead of 'the').
3. **Automated Correction:** Use algorithms that can detect and revert transpositions, often using context to verify the correction.

4.2 Noisy Channel Model and Basic Word Correction

The Noisy Channel Model is a theoretical framework used to explain and correct errors in various types of communication, including text processing. This model is particularly useful in natural language processing (NLP) tasks such as spelling correction, where the goal is to infer the most likely intended word given a potentially erroneous input.

4.2.1 Define Noisy Channel Model

The Noisy Channel Model conceptualizes the process of communication as occurring over a "noisy channel" that introduces errors. In the context of text correction, the model helps in inferring the original word (intended word) from a potentially corrupted version (observed word).



Key Components

1. **Original Word (W):** The word that the sender intended to communicate.
2. **Observed Word (x):** The word received by the receiver, which may contain errors.
3. **Error Model ($P(x|W)$):** The probability of observing the word x given that the original word was W . This models the types of errors (deletion, insertion, substitution, transposition) that could occur.
4. **Language Model ($P(W)$):** The prior probability of the word W occurring in the language. This is often derived from word frequency in a large corpus.

5. **Posterior Probability ($P(W|x)$):** The probability of the original word W given the observed word x .

Using Bayes' Theorem, the relationship can be defined as:

Dan Jurafsky



Noisy Channel

- We see an observation x of a misspelled word
- Find the correct word w

$$\begin{aligned}\hat{w} &= \operatorname{argmax}_{w \in V} P(w | x) \\ &= \operatorname{argmax}_{w \in V} \frac{P(x | w)P(w)}{P(x)} \\ &= \operatorname{argmax}_{w \in V} P(x | w)P(w)\end{aligned}$$

11

For practical purposes in word correction, we aim to maximize $P(W|x)$, which simplifies to:

$$\hat{W} = \operatorname{argmax}_{w \in V} P(x|w) \cdot P(w)$$

Where $\hat{W}$ is the word that maximizes the product of the likelihood of the observed word given the original word and the prior probability of the original word.

Error Types Modeled



Computing error probability: confusion matrix

```
del[x,y]:    count(xy typed as x)
ins[x,y]:    count(x typed as xy)
sub[x,y]:    count(x typed as y)
trans[x,y]:  count(xy typed as yx)
```

Insertion and deletion conditioned on previous character

21

1. **Deletion Errors:** Characters are missing from the word.
2. **Insertion Errors:** Extra characters are added to the word.
3. **Substitution Errors:** One character is replaced by another.
4. **Transposition Errors:** Two adjacent characters are swapped.

4.2.2 Test Basic Word Correction

$$\hat{w} = \operatorname{argmax}_{w \in C} \overbrace{P(x|w)}^{\text{channel model}} \overbrace{P(w)}^{\text{prior}}$$

To test basic word correction using the Noisy Channel Model, the following steps are typically undertaken:

Steps:

1. **Corpus Preparation:**

Language Model ($P(W)$): Build a frequency-based language model from a large corpus to estimate the probability of each word.

	count(w)	p(w)
across	220	0.0001707359
access	56	0.0000434600
actress	7	0.0000054325
caress	3	0.0000023282

Error Model ($P(x|W)$): Gather data on common types of spelling errors to estimate the likelihood of observing an erroneous word given the original word.

2. Generate Candidates:

For a given observed word x , generate a list of candidate words W that are within a reasonable edit distance. Tools like Levenshtein distance can help generate these candidates.

```
[ ] intended_possible_words("acress")  
['actress', 'across', 'caress', 'access', 'cress']
```

3. Compute Probabilities:

For each candidate word W , compute the product $P(x|W) \cdot P(W)$

$P(x|W)$: Estimate based on common error patterns (e.g., substitution of similar-looking characters).

$P(W)$: Retrieve from the language model.

4. Select the Best Candidate:

Choose the word x that maximizes $P(x|W) \cdot P(W)$

Experiment and Results

Chapter 5

Error Model/Likelihood Calculation

In this section, I develop functions that will tell me the probabilities of some characters in the target being mistyped as some other characters in the typo. In order to do that, I used a [typo corpus](<http://norvig.com/ngrams/spell-errors.txt>) by Norvig, made a dictionary out of it, and divided the typos corpus into the different **error-specific typo corpora** (deletion, insertion, substitution, transposition).

For each created corpus, I then developed functions to give me **conditional probabilities** for character permutations in that specific error type. Finally, I developed functions that give the best candidates for each type of typo, which is the **noisy channel probability**.

5.1 Extracting and Cleaning the Typo Corpus

The first step in developing an error model is to extract and clean a corpus of typos. This involves:

1. **Loading the Typo Corpus:** Use a pre-existing corpus of typos, such as Peter Norvig's typo corpus.
2. **Parsing the Corpus:** Read the corpus file and split each line into a target word and its corresponding typo.
3. **Filtering by Edit Distance:** Ensure that each typo-target pair has an edit distance of one, focusing on simple typographical errors

for initial analysis. This ensures the corpus is relevant for the intended spelling correction model.

```

  Edit Distance

[ ] import re
    import editdistance

[ ] editdistance.eval("acress", "actress")
    1

[ ] editdistance.eval("atcress", "actress")
    2

```

5.2 Making Error-Specific Corpora

After cleaning the typo corpus, the next step is to create error-specific corpora, categorizing errors into deletions, insertions, substitutions, and transpositions:

1. **Categorize Errors:** Separate the typos into different types:
 - Deletions:** Where a character is missing.
 - Insertions:** Where an extra character is added.
 - Substitutions:** Where one character is replaced by another.
 - Transpositions:** Where two adjacent characters are swapped.
2. **Inverted Index:** For each typo, retrieve the corresponding target word by indexing the typo dictionary. This helps in calculating the frequency of each error type.

```

[ ] deletion_

[ ] insertion_

[ ] transposition_

[ ] substitution_

```



Channel model

Kernighan, Church, Gale 1990

$$P(x|w) = \begin{cases} \frac{\text{del}[w_{i-1}, w_i]}{\text{count}[w_{i-1} w_i]}, & \text{if deletion} \\ \frac{\text{ins}[w_{i-1}, x_i]}{\text{count}[w_{i-1}]}, & \text{if insertion} \\ \frac{\text{sub}[x_i, w_i]}{\text{count}[w_i]}, & \text{if substitution} \\ \frac{\text{trans}[w_i, w_{i+1}]}{\text{count}[w_i w_{i+1}]}, & \text{if transposition} \end{cases}$$

24

5.3 Count Character Permutations

For each error type, count the occurrences of specific character permutations (e.g., 'nw' for 'n' in deletion).

```
[ ] del_diff("anwar", "nwar")
    ' | a '
```

```
[ ] del_diff("anwar", "anar")
    ' n | nw '
```

```
[ ] del_diff("anware", "anwar")
    ' r | re '
```

5.4 Apply Laplace Smoothing

Add one to each count and adjust the denominator to account for all possible bigrams or characters, ensuring no permutation has zero probability.

We need to use something called **Laplace or add-1 smoothing**, where we 'steal' some of the probabilities from existing conditional probabilities in the created dictionary, and 'give' them to bigrams that are not there. Below, from [slide 31](https://web.stanford.edu/class/cs276/handouts/spell_correction.ppt) by Manning & Nayak, the mechanism is explained:

So, we do the 'stealing and giving' by **adding 1 to each count** and **adding the length of whatever source we are using to the denominator**. In the case of deletion, it will be the **length of the bigram count dictionary** or the **length of the letter at position zero** count for when the typo is at the beginning of the word.

Introduction to Information Retrieval

Smoothing probabilities: Add-1 smoothing

- But if we use the confusion matrix example, unseen errors are impossible!
- They'll make the overall probability 0. That seems too harsh
 - e.g., in Kernighan's chart $q \rightarrow a$ and $a \rightarrow q$ are both 0, even though they're adjacent on the keyboard!
- A simple solution is to add 1 to all counts and then if there is a $|A|$ character alphabet, to normalize appropriately:

$$\text{If substitution, } P(x|w) = \frac{\text{sub}[x,w] + 1}{\text{count}[w] + A}$$

31

5.5 Calculate Conditional Probabilities

Divide the adjusted counts by the total occurrences of the relevant bigrams or characters to get smoothed probabilities.

```
[ ] cond_prob_del = del_prob_smooth(deletion_)
    cond_prob_del

'b|ba': 0.0047694753577106515,
'o|op': 0.004622496147919877,
'o|os': 0.004608294930875576,
'i|iv': 0.004477611940298508,
'a|ab': 0.00436046511627907,
'o|om': 0.003931847968545216,
'k|k_': 0.0037243947858473,
```

5.6 Compute Noisy Channel Probability

For each typo, calculate the likelihood of the intended word by multiplying the conditional probability of the typo given the intended word by the prior probability of the intended word.

```
[ ] del_best_match("he")  
  
({'the': '0.0007336561',  
  'her': '0.0001545999',  
  'she': '0.0000456145',  
  'hen': '0.0000001801',  
  'hey': '0.0000000852',  
  'hem': '0.0000000291',  
  'hue': '0.0000000165',  
  'hew': '0.0000000014'},  
 'the')
```

```
[ ] transp_best_match("onw")  
  
({'now': '0.0000136013', 'own': '0.0000043515'}, 'now')  
  
[ ] transp_best_match("solt")  
  
({'slot': '0.0000000072'}, 'slot')
```

5.7 Identify Best Candidates

Select the word with the highest noisy channel probability as the best candidate for the given typo.

```
[ ] print(overall_best_match("acress"))  
    print(all_cand_probs_sort)  
  
('across', ['across', 'access', 'actress', 'caress'])  
{'across': '0.0000075219', 'access': '0.0000000778', 'actress': '0.0000000685', 'caress': '0.0000000178'}
```

Chapter 6

Noisy Channel Spelling Corrector

6.1 Identifying the Best Candidate for a Typo

Identifying the best candidate for a typo involves several crucial steps that integrate various functions and data structures to evaluate and select the most probable correction for a given misspelled word.

6.1.1 Function Definition

The first step is to define a function, **overall_best_match**, which takes a string representing a typo and aims to return the most appropriate correction. This function serves as the core component, orchestrating the subsequent steps to determine the best candidate.

6.1.2 Initialize Dictionary

Next, an empty dictionary, **all_cand_probs**, is initialized to store potential correction candidates and their associated probabilities. This dictionary will be populated through various error-handling functions.

6.1.3 Handle Different Error Types

The function must handle different types of typographical errors, such as deletions, insertions, substitutions, and transpositions. Each error type is addressed using specific functions (**del_best_match**, **insert_best_match**, **subst_best_match**, and **transp_best_match**), which generate candidates and calculate their probabilities. These candidates are then added to the **all_cand_probs** dictionary.

6.1.4 Sort Candidates

Once the candidates for all error types are collected, they are sorted based on their probabilities in descending order. This sorting helps in identifying the candidate with the highest likelihood of being the correct word.

6.1.5 Return Best Candidate

The candidate with the highest probability is identified and returned as the best correction for the typo. This process involves selecting the first entry in the sorted list of candidates.

6.1.6 Error Handling

Finally, the function includes error-handling mechanisms to manage cases where no suitable candidates are found. If the **all_cand_probs** dictionary remains empty after evaluating all error types, the function returns an error message indicating that the correction is not in the lexicon. This ensures that the system can gracefully handle instances where a typo cannot be corrected.

6.2 User Input Version

The user input version of the spelling corrector is designed to make the correction process interactive and user-friendly. This version allows users to input a word or sentence and receive immediate feedback on potential corrections.

6.3 Correcting Typos in a Sentence

Correcting typos in an entire sentence involves identifying and correcting multiple errors within a given context. This process requires a systematic approach to detect and rectify each typo while considering the context provided by surrounding words.

6.3.1 Spot Typos

The first step in correcting typos within a sentence is to identify the words that are likely misspelled. This involves:

1. **Function Definition:** Define a function that takes a sentence as input and identifies potential typos.
2. **Tokenization:** Split the sentence into individual words. This makes it easier to analyze each word separately.
3. **Lexicon Check:** Compare each word against a predefined vocabulary (lexicon) to spot words that are not present. These words are flagged as potential typos.

6.3.2 Correct Each Typo

Once potential typos are identified, the next step is to correct them:

1. **Iterate Through Typos:** Loop through each identified typo in the sentence.
2. **Find Best Match:** For each typo, use the `overall_best_match` function to find the most likely correction. This function considers various error types and selects the best candidate based on noisy channel probabilities and context.
3. **Replace Typos:** Replace each typo in the sentence with its best correction candidate. This ensures that the sentence gradually becomes more accurate as each typo is corrected.

6.3.3 Error Handling

Error handling is crucial to ensure the process is robust and user-friendly:

1. **Unrecognized Typos:** If a typo is identified but no suitable correction is found (e.g., the word is not in the lexicon), the function should handle this gracefully.
2. **Notification:** Inform the user if certain typos could not be corrected. This transparency helps maintain user trust and provides feedback for further improvement.

6.3.4 Return Corrected Sentence

After all typos have been processed, the final step is to return the corrected sentence:

1. **Reconstruct Sentence:** Join the corrected words back into a complete sentence.
2. **Output:** Provide the corrected sentence as output to the user.

Example Workflow

1. **Input Sentence:** "the snow is faling and it is beautiful"
2. **Spot Typos:** Identify "faling" and "beautifful" as potential typos.
3. **Correct Each Typo:**
 - Find best match for "faling" (e.g., "falling")
 - Find best match for "beautifful" (e.g., "beautiful")
4. **Return Corrected Sentence:** "the snow is falling and it is beautiful"

Chapter 7

Integrating Context

To enhance the accuracy of spelling correction, it is crucial to consider the context in which a typo occurs. Integrating context helps in selecting the most appropriate correction based on the surrounding words, leveraging a bigram language model. This section outlines the process of creating bigrams, building a bigram corpus, and using this contextual information to improve the spelling correction system.

7.1 Creating Bigrams and a Bigram Corpus

A bigram is a sequence of two adjacent words in a text corpus. By analyzing bigrams, we can determine the likelihood of a word following another word, which provides valuable context for correcting typos. To achieve this, we need to create a bigram corpus from a large and diverse text dataset.

7.1.1 Bigram Corpus Function

The initial step in integrating context into our spell correction model is defining a function to generate bigrams from a list of words. This function processes a text corpus to create pairs of consecutive words, which are crucial for calculating bigram probabilities. The bigram function operates by taking a list of words as input and outputting a list of bigrams. Each bigram consists of two consecutive words from the input list, enabling the model to understand word pairs and their frequencies within the text.

7.1.2 Corpus Preparation

Before creating bigrams, it is essential to prepare the corpus to ensure it is clean and suitable for analysis. This involves removing punctuation and combining multiple text sources to create a comprehensive dataset.

1. **Remove Punctuation:** Punctuation marks can create meaningless bigrams (e.g., ("hello", ",")). Removing punctuation ensures that only relevant word pairs are considered.
2. **Combine Text Sources:** To create a robust bigram corpus, we combine multiple text sources. In this project, we use both the **big.txt** corpus and the Reuters corpus from NLTK. This combination provides a large and diverse dataset, enhancing the reliability of the bigram probabilities.

7.1.3 Create Bigrams

With the prepared corpus, the next step is to generate the list of bigrams. This list will serve as the foundation for calculating bigram frequencies and probabilities.

1. **Generate Bigrams:** Using the bigram function defined earlier, generate bigrams from the prepared corpus. Each bigram represents a pair of consecutive words in the text.
2. **Frequency Calculation:** Count the occurrences of each bigram in the corpus to create a frequency dictionary. This dictionary records how often each bigram appears, which is crucial for calculating probabilities.

7.2 Bigram Frequency Dictionary and Smoothed Probabilities

To integrate context effectively into the spelling corrector, we need to calculate the probabilities of word sequences using bigrams. This section outlines the steps for creating a bigram frequency dictionary, calculating word frequencies, and applying smoothing techniques to ensure robust probability estimates.

7.2.1 Bigram Frequency Dictionary

The bigram frequency dictionary records how often each pair of consecutive words (bigrams) appears in the corpus. This information is essential for calculating the probabilities of different word sequences.

1. **Generate Bigrams:** From the prepared corpus, generate bigrams using the function defined in the previous section.
2. **Count Bigram Occurrences:** For each bigram, count how often it appears in the corpus. This count provides the raw frequency data needed to calculate probabilities.
3. **Store in Dictionary:** Store these counts in a dictionary, where the key is the bigram (a tuple of two words), and the value is the count of occurrences.

This dictionary allows us to look up the frequency of any given bigram in the corpus, providing the foundational data for probability calculations.

7.2.2 Word Frequency Calculation

In addition to bigram frequencies, we need the frequencies of individual words to normalize bigram probabilities. This step involves calculating how often each word appears in the corpus.

1. **Count Word Occurrences:** Iterate through the corpus and count how often each word appears. This count provides the frequency data for individual words.
2. **Store in Dictionary:** Store these counts in a dictionary, where the key is the word, and the value is the count of occurrences.

These word frequencies are used as the denominator in the probability calculations for bigrams, ensuring that the probabilities are correctly normalized.

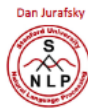
7.2.3 Smoothed Probabilities

To handle cases where certain bigrams do not appear in the corpus, we apply Laplace smoothing. This technique ensures that all possible bigrams have a non-zero probability, which is crucial for making reliable predictions.

1. **Apply Laplace Smoothing:** Add a small constant (typically 1) to each bigram count and adjust the denominator by the vocabulary

size (the total number of unique words in the corpus). This adjustment prevents any bigram probability from being zero.

- Add-1 estimate:
$$P_{Add-1}(w_i | w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + V}$$



Laplace-smoothed bigrams

$$P^*(w_n | w_{n-1}) = \frac{C(w_{n-1}w_n) + 1}{C(w_{n-1}) + V}$$

	i	want	to	eat	chinese	food	lunch	spend
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

2. **Calculate Probabilities:** Use the smoothed counts to calculate the probability of each bigram. This probability represents the likelihood of a word following another word in the corpus, given the observed frequencies.

7.3 Final Function for Context-Aware Spell Correction

The final function for context-aware spell correction integrates the noisy channel model with the bigram language model. This comprehensive approach calculates the probabilities of possible corrections, considering both the likelihood of individual word errors and the context provided by surrounding words. Here we outline the steps for creating this function and ensuring it effectively corrects spelling errors within a sentence.

7.3.1 Noisy Channel Probability

To begin, we calculate the probability of the typo given the intended word using the noisy channel model. This involves:

1. **Character Error Model:** Determine the likelihood of specific characters being mistyped based on historical data (e.g., a corpus of common typographical errors).

2. **Conditional Probability:** Calculate the probability of each potential intended word, given the observed typo.

Dan Jurafsky



Noisy channel probability for across

Candidate Correction	Correct Letter	Error Letter	x w	P(x word)	P(word)	$10^9 \cdot P(x w)P(w)$
actress	t	-	c ct	.000117	.0000231	2.7
cress	-	a	a #	.00000144	.000000544	.00078
caress	ca	ac	ac ca	.00000164	.00000170	.0028
access	c	r	r c	.000000209	.0000916	.019
across	o	e	e o	.0000093	.000299	2.8
acres	-	s	es e	.0000321	.0000318	1.0
acres	-	s	ss s	.0000342	.0000318	1.0

27

7.3.2 Bigram Probability

Next, we incorporate the context provided by surrounding words using the bigram language model. This involves:

1. **Bigram Frequency:** Use the bigram frequency dictionary to find how often each candidate word follows the preceding word and precedes the following word in the corpus.
2. **Conditional Bigram Probability:** Calculate the probability of each candidate word appearing in the context of the surrounding words using the formula:

Dan Jurafsky



Using a bigram language model

- "a stellar and versatile **acress** whose combination of sass and glamour..."
- Counts from the Corpus of Contemporary American English with add-1 smoothing
- $P(\text{actress}|\text{versatile}) = .000021$ $P(\text{whose}|\text{actress}) = .0010$
- $P(\text{across}|\text{versatile}) = .000021$ $P(\text{whose}|\text{across}) = .000006$
- $P(\text{"versatile actress whose"}) = .000021 * .0010 = 210 \times 10^{-10}$
- $P(\text{"versatile across whose"}) = .000021 * .000006 = 1 \times 10^{-10}$

29

7.3.3 Combine Probabilities

To determine the most likely correction, we combine the noisy channel probability and the bigram probability:

1. **Multiplicative Combination:** Multiply the noisy channel probability by the bigram probability to get the final probability for each candidate word.
2. **Rank Candidates:** Sort the candidate words by their combined probabilities in descending order.

7.3.4 Error Handling

In any complex system, error handling is crucial to ensure robustness. For our spell corrector:

1. **Missing Words:** Handle cases where the typo or the surrounding words are not found in the lexicon or corpus.
2. **Boundary Conditions:** Address scenarios where the typo is at the beginning or end of a sentence, affecting the availability of preceding or following words.
3. **Fallback Mechanism:** Implement a fallback mechanism to handle cases where no valid candidates are found, such as defaulting to the most similar word in the lexicon.

7.3.5 Return Correct Sentence

Finally, the corrected sentence is assembled:

1. **Substitute Typos:** Replace each identified typo in the sentence with its best candidate correction.
2. **Reconstruct Sentence:** Join the corrected words back into a complete sentence, preserving the original structure and punctuation as much as possible.

Conclusion and Future Work

Chapter 8

Evaluation and conclusion

While this spelling corrector took me a lot of effort and I tried to account for a lot of factors (I am impressed by early programmers who had to figure this out all by themselves), clearly there are still big limitations.

8.1: Problems

8.1 It doesn't correct when there is a space/lack of space.

```
[132] better_spell_corrector("it cannot kn ow that kn_ow is know")  
"kn_ow" has been recognised as a typo, but cannot be corrected at this time  
'it cannot in of that kn_ow is know'
```

8.2 It is a bit of a problem when it comes to **lowercase/uppercase** and **punctuation**.

The bigram model deletes punctuation and so there is no way for the spell corrector to know that something should be uppercase after a dot, for instance. In order to account for that, there should be a condition that tells you that if the prior word is a dot in the bigram, then the word should be uppercase.

Additionally, the spell corrector takes the dot as a typo, although that is not very problematic, since it just leaves it there.

```
[137] better_spell_corrector("it cannot know this. i should be big")  
"." has been recognised as a typo, but cannot be corrected at this time  
'it cannot know this. i should be big'
```

8.3 When a word is not in the corpus (e.g., low frequency), but it is the correct candidate, it will not be given --> maybe smoothing would have helped here, too. This is what I had mentioned already.

```
[138] better_spell_corrector("the xylphone is a nice instrument")  
"xylphone" has been recognised as a typo, but cannot be corrected at this time  
'the xylphone is a nice instrument'
```

8.4 Does not take into account tokenization problems.

```
[133] better_spell_corrector("does not understand that this i sjust a problem of adding a letter of one word to the begin  
'does not understand that this i just a problem of wadding a letter of one word to the beginning of the next word'")
```

8.5 Does not take into account homophones. Some mistakes are due to phonetic similarity, such as writing *fone* instead of *phone*.

```
✓ [134] better_spell_corrector("i called with my fone")  
0s  
'i galled with my one'
```

8.6 Does not deal well with something misspelled as a real-word. Sometimes, a word is an English word, but the meant word is still another one... My spell corrector does not do well with that.

```
[139] better_spell_corrector("He flower is red. He should be the!")  
". " has been recognised as a typo, but cannot be corrected at this time  
"! " has been recognised as a typo, but cannot be corrected at this time  
'he flower is red. he should be the!'
```

8.7 Finally, my spell corrector does not take into account semantics or agreement. For instance, correcting a singular verb to a plural verb if the noun is plural.

```
▶ better_spell_corrector("the mice is grey")  
↩ 'the mice is grey'
```

8.2 Short Coda

To some extent, my spelling corrector managed to improve when I added context, which was already amazing. Of course, as seen above, many exceptions are not captured, but it does a fair job at correcting simple examples, especially when words are high-frequency and/or when the

bigrams are present in the bigrams corpora. Thus, usage of big corpora is essential to such tasks.

This project proved to be a great exercise to understand even further the complexity of language and how much we, as humans, have to deal with, and therefore how incredibly hard it is to transfer this knowledge to computers.

Reference

In order to develop my program, I followed several sources:

[original article](<https://www.aclweb.org/anthology/C90-2036.pdf>) by Kernighan, Church and Gale (1990)

[this chapter on spelling corrections](<https://web.stanford.edu/~jurafsky/slp3/B.pdf>) and [this

chapter on n-gram models](<https://web.stanford.edu/~jurafsky/slp3/3.pdf>) from the *Speech and Language Processing* book by Jurafsky and Martin (2020)

[these slides on language modeling](https://web.stanford.edu/~jurafsky/slp3/slides/LM_4.pdf)

[these slides on spelling corrections](<https://web.stanford.edu/class/cs124/lec/spelling.pdf>) by Jurafsky

[this chapter on spelling correction](<http://norvig.com/ngrams/ch14.pdf>) by Norvig in the *Beautiful Data* book (Segaran and Hammerbacher, 2009)

[this guide on how to write a spelling corrector](<http://norvig.com/spell-correct.html>) by Norvig

[big.txt corpus](<http://norvig.com/big.txt>) and the [spell-errors.txt corpus](<http://norvig.com/ngrams/spell-errors.txt>) as my training set, both kindly provided by Norvig [here](<http://norvig.com/ngrams/>)

[these slides on Laplace smoothing](https://web.stanford.edu/class/cs276/handouts/spell_correction.ppt) by Christopher Manning and Pandu Nayak

[the typo corpus by
Birkbeck](<https://www.dcs.bbk.ac.uk/~ROGER/missp.dat>) as my test
set.

... And, of course, [Stack Overflow](<https://stackoverflow.com/>) for
my infinite questions, problems and bugs.