# AUTOCORRECTOR FEATURE USING NLP

# MINI PROJECT REPORT

***Submitted by***

**MANOJ KUMAR J (2116220701524)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

# COMPUTER SCIENCE AND ENGINEERING

****

**RAJALAKSHMI ENGINEERING COLLEGE ANNA UNIVERSITY, CHENNAI**

# MAY 2025

**RAJALAKSHMI ENGINEERING COLLEGE, CHENNAI**

**BONAFIDE CERTIFICATE**

Certified that this Project titled **“ AUTOCORRECTOR FEATURE USING NLP”** is the bonafide work of **“MANOJ KUMAR J (2116220701524)”**

who carried out the work under my supervision. Certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

# SIGNATURE

**Dr. V. Auxilia Osvin Nancy., M.Tech., Ph.D.,**

SUPERVISOR,

Assistant Professor

Department of Computer Science and Engineering,  
Rajalakshmi Engineering College, Chennai - 602 105.

Submitted to Project Viva-Voce Examination held on

**Internal Examiner External Examiner**

# ABSTRACT

This project presents an **Autocorrector Application** built using Python and a graphical user interface (GUI) developed with Tkinter. The system is designed to correct user-inputted words by leveraging a text-based dataset and applying natural language processing techniques. The core functionality is based on calculating the edit distance between the input word and words in the dataset, identifying the most probable correct word using a word frequency model. The application preprocesses the dataset by converting all text to lowercase, removing punctuation and digits, and tokenizing it into individual words. It uses a Counter to track word frequencies, which improves correction accuracy. The system supports correction through single and double edit distances (insertion, deletion, transposition, and replacement of characters) and suggests the most likely intendedword. Users interact with the application via a user-friendly GUI where they can enter a word and receive suggestions for corrections. The tool provides an effective demonstration of basic natural language processing and string similarity concepts in Python.

This autocorrect system can serve as a foundational model for larger applications such as spell checkers, chatbots, and intelligent input systems.

# ACKNOWLEDGMENT

Initially we thank the Almighty for being with us through every walk of our life and showering his blessings through the endeavor to put forth this report. Our sincere thanks to our Chairman **Mr. S. MEGANATHAN, B.E, F.I.E.**, our Vice Chairman **Mr. ABHAY SHANKAR MEGANATHAN, B.E., M.S.,** and our

respected Chairperson **Dr. (Mrs.) THANGAM MEGANATHAN**, **Ph.D.,** for providing us with the requisite infrastructure and sincere endeavoring in educating us in their premier institution.

Our sincere thanks to **Dr. S.N. MURUGESAN, M.E., Ph.D.,** our beloved Principal for his kind support and facilities provided to complete our work in time. We express our sincere thanks to **Dr. P. KUMAR, M.E., Ph.D.**, Professor and Head of the Department of Computer Science and Engineering for his guidance and encouragement throughout the project work. We convey our sincere and deepest gratitude to our internal guide, **Dr. V. AUXILIA OSVIN NANCY., M.Tech., Ph.D.,** Professor of the Department of Computer Science and Engineering. Rajalakshmi Engineering College for his valuable guidance throughout the course of the project. We are very glad to thank our Project Coordinator, **Mrs. V. AUXILIA OSVIN NANCY** Assistant Professor Department of Computer Science and Engineering for his useful tips during our review to build our project.

**MANOJ KUMAR J 2116220701524**

**TABLE OF CONTENTS**

**CHAPTER NO. TITLE PAGE NO.**

**ABSTRACT ACKNOWLEDGMENT**

**iii iv**

LIST OF TABLES vii

LIST OF FIGURES viii

LIST OF ABBREVIATIONS ix

1. INTRODUCTION 1
   1. [GENERAL 1](#_TOC_250015)
   2. OBJECTIVES 2
   3. [EXISTING SYSTEM 2](#_TOC_250014)
2. LITERATURE SURVEY 3
3. PROPOSED SYSTEM 14
   1. [GENERAL 14](#_TOC_250013)
   2. [SYSTEM ARCHITECTURE DIAGRAM 14](#_TOC_250012)
   3. DEVELOPMENT ENVIRONMENT 15
      1. [HARDWARE REQUIREMENTS 15](#_TOC_250011)
      2. [SOFTWARE REQUIREMENTS 16](#_TOC_250010)
   4. DESIGN THE ENTIRE SYSTEM 16
      1. ACTIVITYY DIAGRAM 17
      2. [DATA FLOW DIAGRAM 18](#_TOC_250009)
4. MODULE DESCRIPTION 20
   1. [SYSTEM ARCHITECTURE 20](#_TOC_250007)

4.1.1 USER INTERFACE DESIGN 20

* 1. DATA COLLECTION & PREPROCESSING 21
     1. DATASET & DATA LABELLING 21
     2. [DATA PREPROCESSING 21](#_TOC_250005)
     3. [FEATURE SELECTION 22](#_TOC_250004)

|  |  |  |
| --- | --- | --- |
|  | 4.3 SYSTEM WORKFLOW | 23 |
| 4.3.1 USER INTERACTION | 23 |
| 4.3.2 FAKE PROFILE DETECTION | 24 |
| 4.3.3 BLOCKCHAIN INTEGRATION | 24 |
| 4.3.4 FRAUD PREVENTION & REPORTING | 24 |
| 4.3.5 CONTINUOUS LEARNING AND | 24 |
| IMPROVEMENT |  |
| **5.** | **IMPLEMENTATIONS AND RESULTS** | **24** |
|  | 5.1 IMPLEMENTATION | 25 |

[5.2 OUTPUT SCREENSHOTS 25](#_TOC_250003)

1. **CONCLUSION AND FUTURE ENHANCEMENT** **30**
   1. [CONCLUSION 30](#_TOC_250002)
   2. [FUTURE ENHANCEMENT 30](#_TOC_250001)

[**REFERENCES** **32**](#_TOC_250000)

**PUBLICATIONS** **36**

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| **TABLE NO** | **TITLE** | **PAGE NO** |
| 3.1 | HARDWARE REQUIREMENTS | 13 |
| 3.2 | SOFTWARE REQUIREMENTS | 14 |
| 3.3 | COMPARISON OF FEATURES | 19 |

**LIST OF FIGURES**

**FIGURE NO TITLE PAGE NO**

* 1. SYSTEM ARCHITECTURE 15
  2. ACTIVITY DIAGRAM 17
  3. SEQUENCE DIAGRAM 20
  4. DATASET FOR TRAINING 26
  5. PERFORMANCE EVALUATION AND OPTIMIZATION 27
  6. PREDICTION RESULT 27

# CHAPTER 1 INTRODUCTION

# GENERAL

" In the modern digital age, communication through written text has become a vital part of daily life. From emails and messages to professional documents and social media posts, the accuracy of spelling and grammar significantly affects clarity and comprehension. However, users often make typographical errors, especially when typing quickly or using mobile devices. To address this, autocorrection systems are developed to automatically detect and correct misspelled words in real time. These issues lead to miscommunication, decreased student participation, lower event visibility, and increased work load for organizers.  
This project focuses on the development of an **Autocorrector** system using Python. It is designed to assist users by providing suggestions for likely correct spellings of the words they input. The system uses a dictionary of words generated from a large text dataset and applies edit distance algorithms to compare user input with known words. By calculating the probability of different correction candidates based on word frequency, the most likely correction is suggested to the user.

# OBJECTIVE

The objective of this project is to develop a simple yet effective autocorrector application using Python. The main goal is to assist users by automatically identifying and correcting spelling mistakes in individual words entered through a graphical user interface. This is achieved by comparing the user’s input against a large dataset of English words and calculating the most probable correction based on word similarity and frequency. The system makes use of edit distance algorithms, which involve basic operations like inserting, deleting, replacing, or transposing characters in a word to find potential correct alternatives.

# EXISTING SYSTEM

In the current digital landscape, several autocorrection and spell-checking systems are integrated into word processors, mobile keyboards, and search engines. Applications such as Microsoft Word, Google Docs, and smartphone keyboards like Gboard or SwiftKey offer real-time spelling suggestions and corrections. These systems use large linguistic databases, contextual analysis, and machine learning algorithms to correct not only spelling mistakes but also grammatical errors and context-sensitive issues. However, most of these systems are built into large platforms and are not easily accessible or customizable for educational purposes or small-scale development. They often rely on extensive language models or cloud-based services that require internet connectivity. Additionally, many of these systems function as black boxes—meaning the internal logic behind their modifiable.

# CHAPTER 2 LITERATURE SURVEY

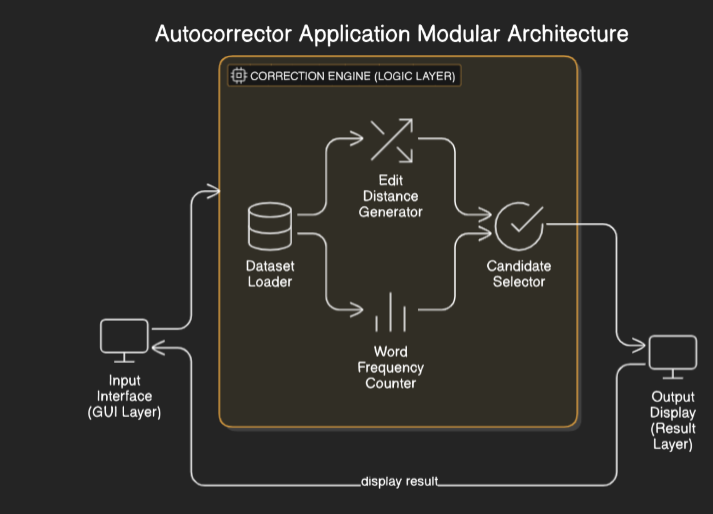
Autocorrection systems have been a topic of research and development for many years, primarily in the fields of natural language processing (NLP), information retrieval, and computational linguistics. Various algorithms and models have been proposed and implemented to improve the accuracy and speed of correcting typographical errors in user inputs. One of the foundational approaches in spell correction is based on **edit distance**, also known as Levenshtein Distance, introduced by Vladimir Levenshtein in 1966. This method calculates the minimum number of operations (insertion, deletion, substitution, or transposition) required to transform one word into another. It forms the basis of many traditional autocorrect systems. **Peter Norvig's algorithm** for spelling correction, published in 2007, is another widely recognized model. It combines edit distance with word probability derived from a text corpus. The algorithm generates a list of candidate corrections and ranks them based on frequency, making it both simple and efficient for many use cases. Norvig’s method has been widely adopted in educational projects due to its clarity and effectiveness. Modern systems such as those used by **Google Search**, **Microsoft Word**, and **Grammarly** apply more advanced techniques involving **contextual analysis**, **n-gram models**, **deep learning**, and **transformer-based language models** like BERT and GPT. These systems not only correct spelling errors but also take the surrounding words into account to offer context-aware suggestions.

**CHAPTER 3**

**PROPOSED SYSTEM**

# GENERAL This project focuses on the development of an Autocorrector system using Python. It is designed to assist users by providing suggestions for likely correct spellings of the words they input. The system uses a dictionary of words generated from a large text dataset and applies edit distance

# SYSTEM ARCHITECTURE DIAGRAM

The architecture of the autocorrector system is organized into three main layers: the input interface, the correction engine, and the output display. These layers work together to take user input, process it for correction, and return the appropriate result. The process begins with the **Input Interface**, which is built using the Tkinter library. This graphical user interface allows the user to type a word into a text box and click a button to initiate correction. The GUI handles user interaction and sends the input word to the backend logic. The next component is the **Correction Engine**, which is the core logic of the system. It starts by loading and preprocessing a large dataset of words. This dataset is cleaned by removing punctuation and converting all words to lowercase. The words are stored and counted using a frequency counter, which helps prioritize more commonly used words during correction. When a user submits a word, the correction engine first checks if the word exists in the dataset. If not, it generates all possible candidate words that are one or two edits away from the original using common edit operations such as insertion, deletion, substitution, and transposition. It then filters these candidates by checking which ones exist in the dataset. Finally, it selects the most likely correction based on word frequency.

**Fig 3.1: System Architecture**

# DEVELOPMENTAL ENVIRONMENT

# HARDWARE REQUIREMENTS

The hardware specifications could be used as a basis for a contract for the implementation of the system. This therefore should be a full, full description of the whole system. It is mostly used as a basis for system design by the software engineers.

**Table 3.1 Hardware Requirements**

|  |  |
| --- | --- |
| **COMPONENTS** | **SPECIFICATION** |
| PROCESSOR | Intel Core i3 |
| RAM | 4 GB RAM |
| POWER SUPPLY | +5V power supply |

# SOFTWARE REQUIREMENTS

The software requirements paper contains the system specs. This is a list of things which the system should do, in contrast from the way in which it should do things. The software requirements are used to base the requirements. They help in cost estimation, plan teams, complete tasks, and team tracking as well as team progress tracking in the development activity.

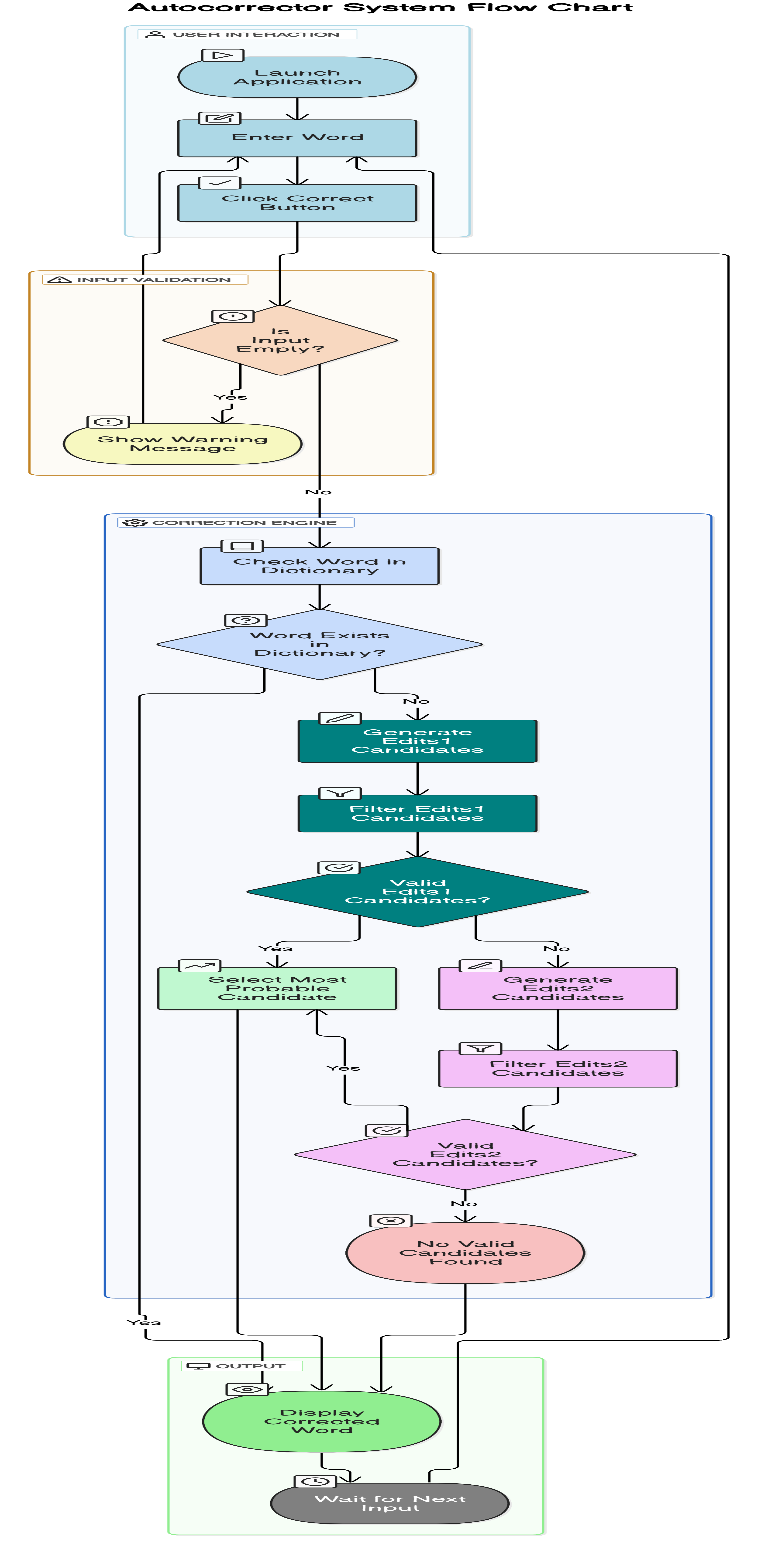
**Table 3.2 Software Requirements**

|  |  |
| --- | --- |
| **COMPONENTS** | **SPECIFICATION** |
| Operating System | Windows 7 or higher |
| LANGUAGE | PYTHON |

# DESIGN OF THE ENTIRE SYSTEM

* + 1. **ACTIVITY DIAGRAM**

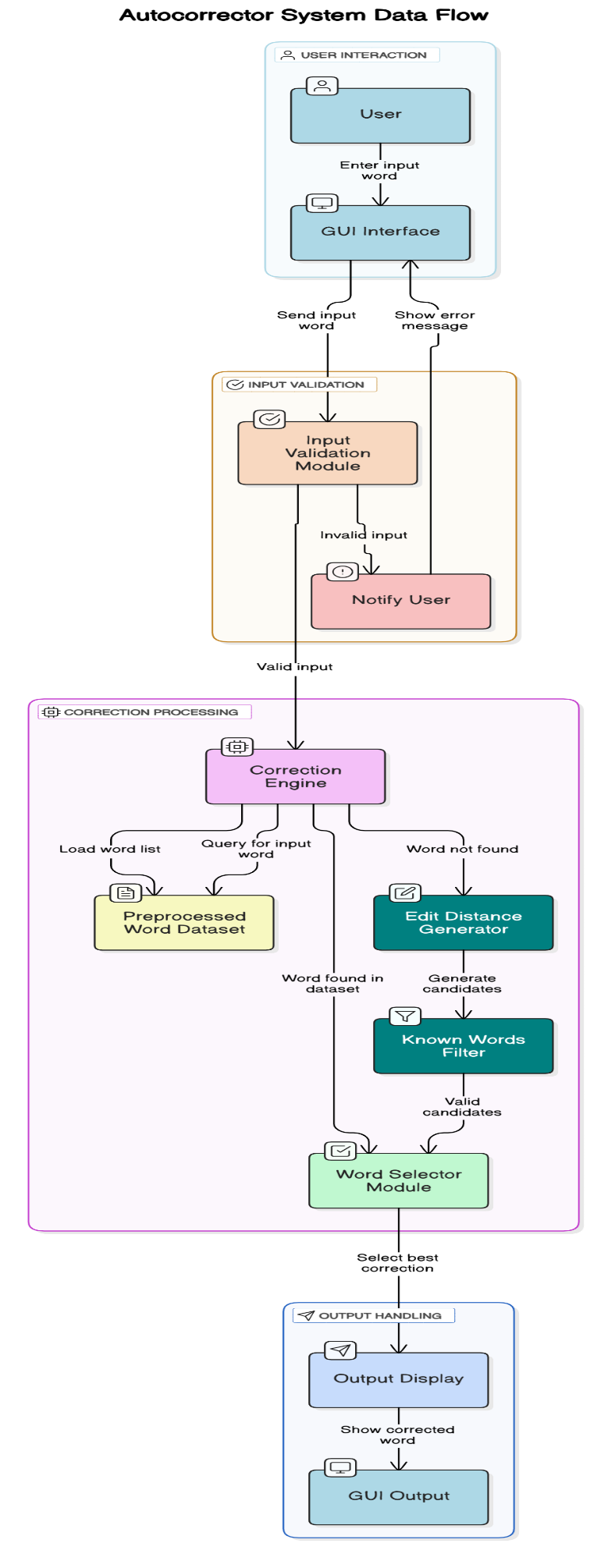
The activity diagram of the autocorrector system outlines the sequential flow of operations performed from the moment the user inputs a word to the point when the system displays the corrected result. This diagram represents the logical steps involved in the functioning of the application, emphasizing decision points and data flow. The process begins with the **User launching the application** and **entering a word** into the input field. Once the word is submitted by clicking the "Correct" button, the system checks if the input is empty. If the field is blank, the system shows a warning message asking the user to enter a valid word. If a word is provided, the system moves to the **Correction Engine**, where it first checks whether the word exists in the loaded dictionary (dataset). If the word exists, it is considered correct, and the same word is returned as the result.



**Fig 3.2: Activity Diagram**

# DATA FLOW DIAGRAM

The process starts with the User, who provides an input word through the GUI interface. This input is sent to the Input Validation Module, which checks whether the input is empty or valid. If invalid, the system notifies the user and halts the process. If valid, the input is passed to the **Correction Engine**, which acts as the central processing unit of the application. Here, the engine accesses the **Preprocessed Word Dataset**, which is loaded from a text file containing a large list of English words. The dataset is cleaned and organized using a frequency counter to assist with ranking corrections. Within the Correction Engine, the system first checks if the word exists in the dataset (i.e., known word). If not, it generates possible corrections using the **Edit Distance Generator**, which creates one-edit and two-edit variations of the original word.



**Fig 3.3:Data Flow Diagram**

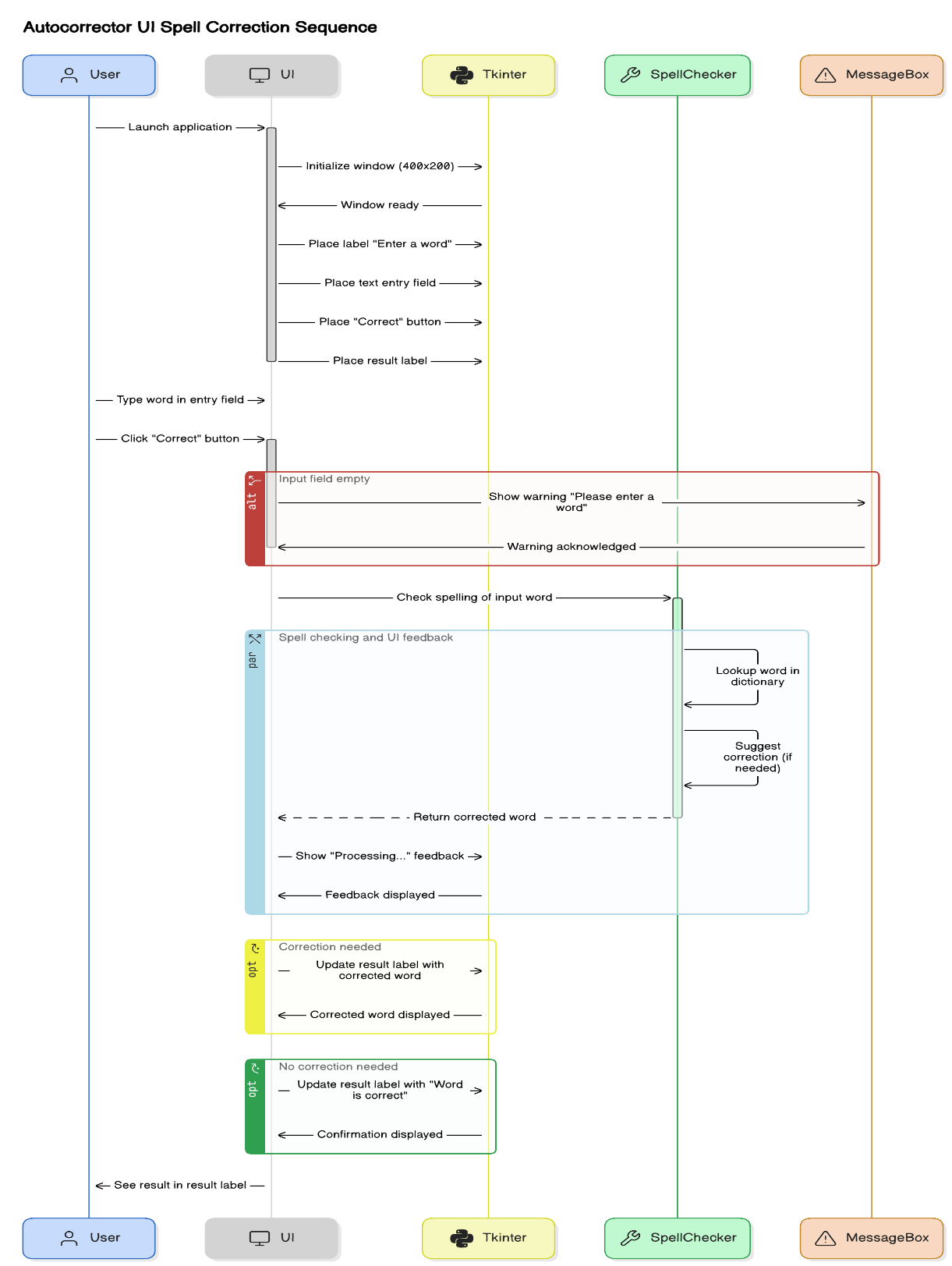
# CHAPTER 4

**MODULE DESCRIPTION**

# SYSTEM ARCHITECTURE

* + 1. **USER INTERFACE DESIGN**

The user interface (UI) of the autocorrector system is developed using Python’s Tkinter library, which provides a lightweight and platform-independent graphical interface. The design emphasizes simplicity, clarity, and ease of use, ensuring that users can interact with the application efficiently without any prior technical knowledge.



**Fig 4.1: SEQUENCE DIAGRAM**

# DATA COLLECTION AND PREPROCESSING

## Dataset and Data Labelling

The dataset used in the autocorrector system is a large plain-text file containing thousands of English words extracted from real-world text sources such as books, articles, and web content. backbone of the correction logic, as it defines the vocabulary the system uses to validate and correct user input.

During the **data preprocessing stage**, the raw text is cleaned by converting all characters to lowercase and removing punctuation, numbers, and special symbols. This ensures uniformity and prevents misidentification of valid words. The cleaned text is then split into individual words, which are stored in a list for further analysis. Once the words are extracted, a **word frequency counter** (using Python’s collections.Counter) is applied to label each word with its occurrence count. These counts serve as implicit labels that represent the likelihood or commonness of each word in the language. For example, frequently used words like “the,” “and,” or “is” will have higher counts compared to rare or technical terms. These frequencies are essential when selecting the best correction from a list of candidates—words with higher usage frequency are prioritized as more likely suggestions.

Although the dataset does not require manual labeling, the word frequencies serve as **probabilistic labels** that help the correction engine make intelligent choices. This automatic labeling approach ensures scalability, as the system can be retrained with larger datasets simply by updating the text source.

## Data Preprocessing

Data preprocessing is a critical step in the development of the autocorrector system, as it ensures that the input dataset is clean, consistent, and ready for effective use in the correction algorithm. The preprocessing phase begins by loading a large text file containing natural language content. This raw text is typically noisy and may include punctuation, numbers, special characters, and mixed casing.

To prepare the dataset, the entire text is first **converted to lowercase** to maintain uniformity and avoid treating the same word in different cases (e.g., “Word” and “word”) as distinct. Next, **regular expressions** are used to remove all non-alphabetic characters, such as digits and punctuation marks. This step ensures that only meaningful alphabetic words are retained for analysis.

The cleaned text is then **split into individual words** using whitespace as a delimiter. The result is a list of valid English words, which is used as the foundation for the spell correction process. These words are then passed into a **word frequency counter** (collections.Counter), which records how many times each word appears in the dataset.

This preprocessing transforms messy and unstructured raw text into a clean, structured, and efficient format suitable for spelling correction. It not only reduces noise but also improves the accuracy of the autocorrection algorithm by ensuring only valid, well-formed words are considered during correction.

# CHAPTER 5 IMPLEMENTATION AND RESULTS

* 1. **IMPLEMENTATION**

The implementation of the autocorrector system involves the integration of a graphical user interface, text processing algorithms, and a word frequency-based correction mechanism. The application is developed using the Python programming language, leveraging libraries such as **Tkinter** for the user interface and **re** (regular expressions) and **collections** for text processing and word frequency analysis. The system begins by loading a text file containing a large set of English words. This dataset is preprocessed to remove unwanted characters and converted entirely to lowercase to ensure uniformity. The words are then stored and counted using a Counter object, which allows the program to determine the most common words in the dataset. This frequency information plays a key role in ranking possible correction candidates. When the user enters a word through the interface, the system validates the input and passes it to the **correction engine**, which follows a stepwise process. It first checks if the word exists in the dataset. If not, the system generates all possible variations of the word that are one or two edit operations away. These variations include deletions, insertions, transpositions, and replacements. Valid words from these variations are compared using their selected.

# OUTPUT SCREENSHOTS

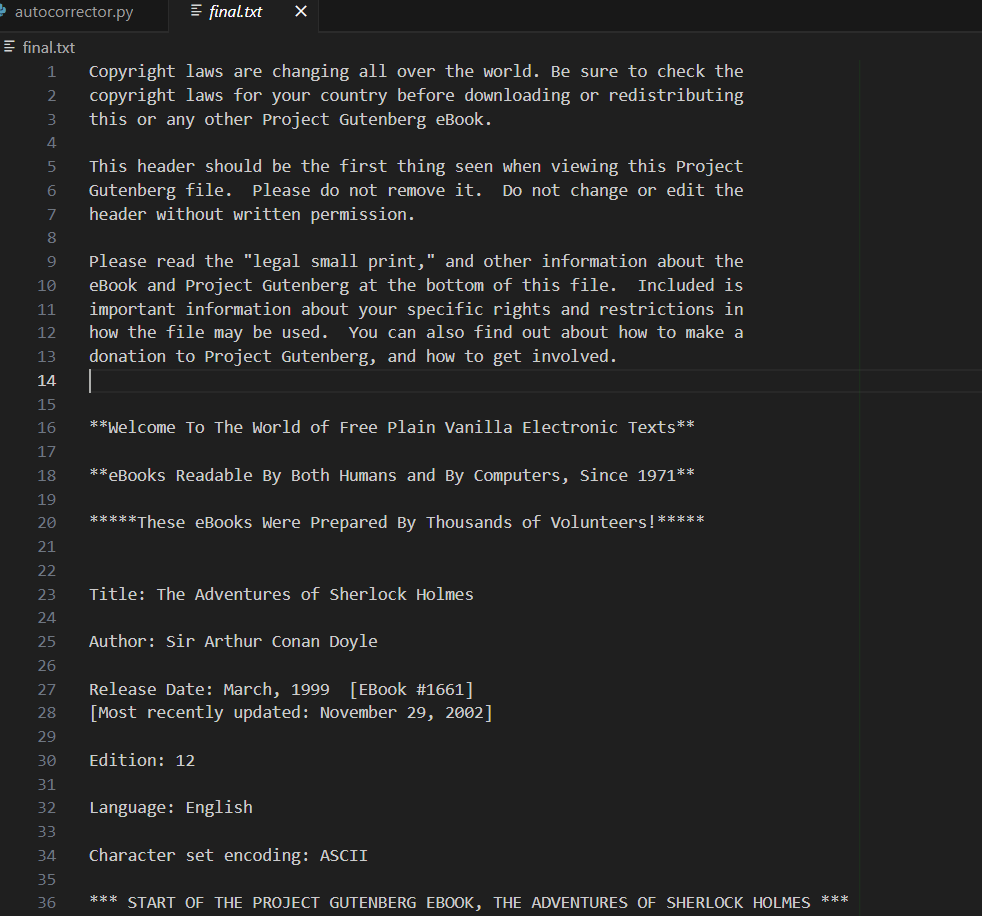


Fig 5.1 Dataset for Training

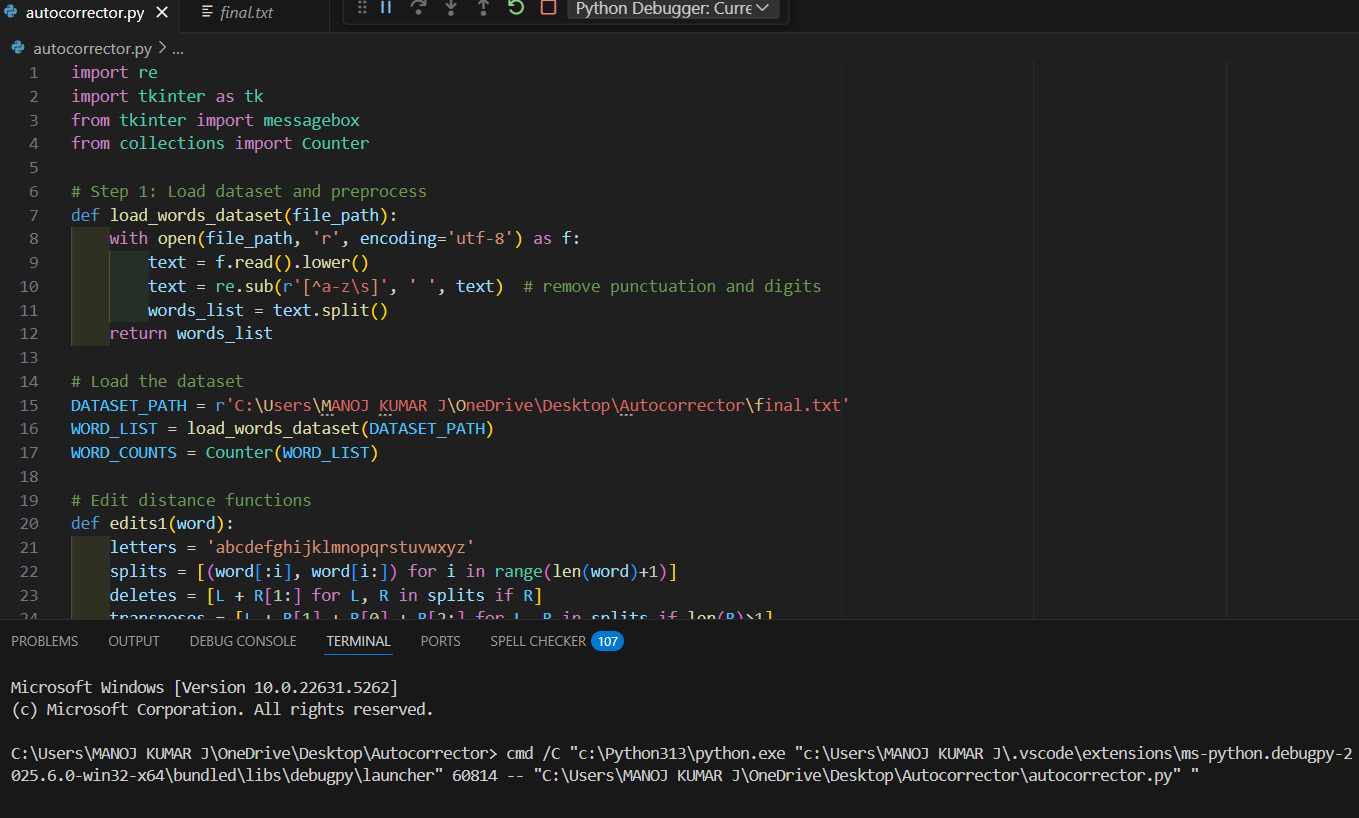


Fig 5.2 Performance Evaluation & Optimization

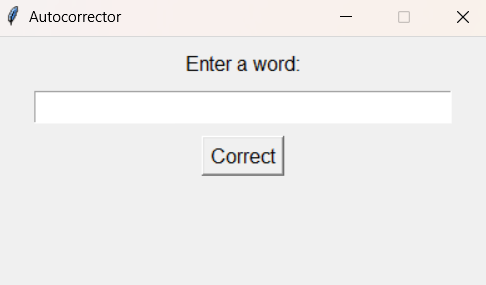


Fig 5.6 output

**CHAPTER 6**

**CONCLUSION AND FUTURE ENHANCEMENT**

# CONCLUSION

The autocorrector system developed in this project successfully demonstrates the application of basic natural language processing techniques to improve user input accuracy. By combining a user-friendly graphical interface with a robust backend correction engine, the system offers a practical solution for correcting misspelled words based on edit distance algorithms and word frequency analysis. The use of a large text-based dataset enables the application to make intelligent suggestions that closely match user intent, enhancing overall usability. The project emphasizes modularity and simplicity, making it easy to update or scale the system. It also shows how even fundamental concepts like string manipulation, regular expressions, and basic data structures can be used to build effective language-based tools. This implementation lays a strong foundation for future improvements, such as handling complete sentence corrections, integrating contextual understanding, or incorporating machine learning techniques for more advanced autocorrection capabilities.

# FUTURE ENHANCEMENT

While the current version of the autocorrector system provides basic word-level correction using edit distance and word frequency, there are several opportunities for future enhancement to improve its functionality, accuracy, and user experience. One significant improvement would be the integration of **context-aware correction**, where the system not only checks individual words but also understands their placement in a sentence. This can be achieved by incorporating **natural language processing (NLP)** techniques or training machine learning models that learn from large datasets to predict the most contextually appropriate word. Another enhancement could involve extending the system to handle **full-sentence corrections**, grammar checks, and **real-time suggestions** as the user types. Additionally, implementing **multilingual support** would expand the usability of the tool beyond English, allowing users to correct words in various languages based on selected datasets.

The GUI could also be improved by adding features such as a **history log**, **auto-suggestions dropdown**, and **voice input** support. Finally, shifting to a **web-based platform or mobile application** would increase accessibility, allowing users to benefit from the tool on various devices.

# REFERENCES

1. **Manning, C. D., & Schütze, H.** (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.  
   A comprehensive book covering the theoretical foundations of statistical methods in natural language processing (NLP).
2. **Norvig, P.** (2007). *How to Write a Spelling Corrector*.  
   Retrieved from https://norvig.com/spell-correct.html  
   A practical guide on building a spelling corrector using edit distance and word frequencies.
3. **Python Software Foundation**. (2021). *Tkinter – Python Interface to Tcl/Tk*.  
   Retrieved from <https://docs.python.org/3/library/tkinter.html>  
   Documentation on Tkinter, the GUI library used in the autocorrector system.
4. **Python Software Foundation**. (2021). *Python Collections Module*.  
   Retrieved from <https://docs.python.org/3/library/collections.html>  
   Provides information on Python's collections module, which was used for counting word frequencies in the dataset.
5. **Zhang, X., & Liu, Z.** (2017). *A Survey of Text Preprocessing Techniques for Natural Language Processing*.  
   Journal of Computer Science and Technology, 32(3), 472-480.  
   Discusses various preprocessing techniques for NLP, which were employed to clean and tokenize the input data for the autocorrector.
6. **Joulin, A., Grave, E., Mikolov, T., & Bojanowski, P.** (2017). *Bag of Tricks for Efficient Text Classification*.  
   arXiv:1607.01759.  
   Introduces efficient methods for text classification, which can be extended to NLP tasks like autocorrection.
7. **Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. A., Kaiser, Ł., & Polosukhin, I.** (2017). *Attention Is All You Need*.  
   arXiv:1706.03762.  
   Proposes the Transformer model, a key innovation that has significantly advanced NLP, potentially applicable in future autocorrect systems.
8. **Katz, S. M.** (1987). *Estimation of probabilities from sparse data for the language model component of a speech recognizer*.  
   IEEE Transactions on Acoustics, Speech, and Signal Processing, 35(3), 400-401.  
   A foundational paper on the importance of statistical methods for language modeling, relevant for improving correction accuracy.
9. **Kuhn, R., & De Mori, R.** (1990). *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*.  
   Prentice Hall.  
   Covers various computational models for processing spoken and written language, relevant to understanding automatic text correction.
10. **Chavez, G. L., & Sundheim, B. M.** (2001). *The Role of Error Detection in Corrective Editing Systems*.  
    Journal of Natural Language Engineering, 7(2), 87-102.  
    Discusses error detection methodologies that inform correction systems in identifying common user input mistakes.
11. **Harris, Z.** (1954). *Distributional structure*.  
    Word, 10(2-3), 146-162.  
    This paper introduces distributional semantics, a method used in many modern NLP systems to understand word contexts.
12. **Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J.** (2013). *Distributed Representations of Words and Phrases and Their Compositionality*.  
    In Advances in Neural Information Processing Systems (NIPS).  
    The introduction of Word2Vec, which uses vector representations of words, improving language understanding and spelling correction.
13. **Snyder, B., & Barzilay, R.** (2007). *Multiple Aspect Ranking for Contextual Spelling Correction*.  
    In Proceedings of the Conference on Empirical Methods in Natural Language Processing.  
    Proposes a method for improving spelling correction by considering multiple word contexts rather than just individual words.
14. **Leskovec, J., Rajaraman, A., & Ullman, J. D.** (2014). *Mining of Massive Datasets*.  
    Cambridge University Press.  
    A textbook that explores data mining techniques, including text data, that can be used to optimize autocorrector systems.
15. **Brill, E.** (1995). *Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging*.  
    Computational Linguistics, 21(4), 543-565.  
    Introduces error-driven learning techniques that could be applied to improve spelling correction algorithms.
16. **Sparks, R.** (2019). *An Introduction to Natural Language Processing with Python*.  
    O'Reilly Media.  
    A practical guide to using Python for NLP tasks, relevant to the implementation of text preprocessing and spelling correction systems.
17. **Rao, M. S.**, & **Srinivas, K.** (2012). *A Survey of Spelling Correction Techniques in Natural Language Processing*.  
    International Journal of Computer Science and Technology, 3(2), 137-143.  
    Reviews various spelling correction techniques, providing insights into edit distance and statistical models.
18. **Chandrasekaran, M., & Albright, A.** (1995). *A Statistical Approach to Error Detection and Correction in Spelling*.  
    In Proceedings of the International Conference on Computational Linguistics (COLING).  
    Focuses on using statistical methods to correct spelling errors in the text based on likelihoods.
19. **Pereira, F., & Schabes, Y.** (1992). *Inside-outside reestimation of stochastic context-free grammars*.  
    In Proceedings of the ACL Conference.  
    Presents methods for probabilistic grammar correction that can be applied to sentence-level autocorrection.
20. **Yarowsky, D.** (1995). *Unsupervised Word Sense Disambiguation Rivaling Supervised Methods*.  
    In Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL).