AUTOCORRECTOR FEATURE

USING NLP

Manoj kumar J

UG Scholar, Department of Science and Engineering (CSE)

Rajalakshmi Engineering College (REC), Chennai, Tamil Nadu, India.

220701524@rajalakshmi.edu.in

***Abstract—*** ***This project showcases a Python-driven Autocorrector Application, featuring a user interface built with Tkinter. It leverages natural language processing techniques to compare words entered by users against a dataset, enabling it to suggest corrections. The system generates potential corrections by calculating edit distances, which include insertions, deletions, substitutions, and transpositions, ultimately selecting the word with the highest frequency of use. During preprocessing, punctuation is removed from the dataset, and words undergo tokenization and case conversion. Through the graphical user interface, users can input words and receive immediate suggestions. This application introduces fundamental concepts of natural language processing and serves as a foundation for developing chatbots, intelligent text-input systems, and spell checkers.  
 Keywords—*** ***autocorrect, edit distance, natural language processing, spell checker, word frequency, Python, Tkinter, string matching,*** ***text preprocessing, user interface***

# **INTRODUCTION**

Spelling correction is really important for making things easier for users when they're communicating online, whether they're typing in a document or searching on Google. This paper introduces a new autocorrect system built with Python and a user-friendly interface made with Tkinter. The app spots and fixes typos by using natural language processing, specifically looking at how similar words are (edit distance) and how often words show up (word frequency analysis). After going through a bunch of text data, the system suggests possible corrections when the user types something. This project shows how useful NLP and string similarity can be, and it also gives us a basic tool to build more complex language apps on.

**II. DATASETS**

This autocorrector system uses a dataset that's a plain text file, packed with a bunch of English words taken from how people really talk and write. Before it's used, the text gets cleaned up a bit: everything's turned lowercase, numbers and punctuation are ditched, and the whole thing is split into separate words. Then, Python's `collections.Counter` module steps in to figure out which words show up the most. This word frequency info is super important for how the autocorrector actually works, letting it make smarter guesses about what you meant to type based on how often words are used in real life. Because of this dataset, the system can offer up more accurate and contextually appropriate spelling suggestions for the words people type in.

**III. LITERATURE SURVEY**

Spelling correction has been a hot topic in the world of Natural Language Processing (NLP) for ages. One of the first methods used was the Levenshtein Distance, which calculates the minimum number of edits to turn one word into another. Peter Norvig then came up with a super popular algorithm that combined edit distances with word frequency data from a massive dataset, and this formed the basis for many of the autocorrect tools we use today.

We've seen some cool progress lately with models that use stats and machine learning. Folks used to rely on things like N-gram models and Hidden Markov Models (HMMs) to fix mistakes based on context. But with deep learning getting big, we now use neural networks like Recurrent Neural Networks (RNNs) and even fancier models based on Transformers, such as BERT, to not only correct spelling in a smart, context-aware way but also to check grammar.  
  
That said, these sophisticated models demand vast amounts of data and significant computing power. On the flip side, models based on rules or frequency are simpler, faster, and easier to understand, which makes them perfect for applications where resources are limited. This particular project takes a frequency-based approach and pairs it with edit distance measurements, allowing for fast and reliable word correction even when working offline.

**IV. DATA PREPROCESSING**

Data preprocessing is a crucial step in preparing the dataset for effective autocorrection. The raw text corpus is initially converted to lowercase to maintain uniformity and reduce case sensitivity. All punctuation marks, digits, and special characters are removed using regular expressions to ensure that only valid alphabetic words are retained. The cleaned text is then tokenized—split into individual words—creating a list of candidate terms. A frequency count of these words is generated using Python's Counter module, enabling the system to rank potential corrections by their likelihood of occurrence. This preprocessing ensures that the input to the autocorrect algorithm is clean, relevant, and optimized for accurate predictions

**V. ARCHITECTURE**

The autocorrector system is designed with a modular structure, consisting of three primary components: Input Interface, Correction Engine, and Output Display. The Input Interface, built with Tkinter, offers a straightforward graphical user interface for users to input words. This input is then relayed to the Correction Engine, where it undergoes text preprocessing, edit distance calculations, and word frequency analysis.  
The engine verifies known words and creates single-edit and double-edit variations through insertion, deletion, replacement, and transposition operations. It then selects the most likely correction based on word frequency data. The Output Display component presents the suggested correction in the GUI.  
This system is lightweight, operates locally, and doesn't require an internet connection or external APIs, ensuring efficiency and portability. The modular design facilitates easy maintenance and future enhancements, such as integration with larger natural language processing systems or web-based platforms.

**VI. TRAIN THE DATASET**

Different from typical machine learning models, the autocorrect application doesn't need to be trained in a supervised manner. It works by using a frequency-based method, examining a vast collection of text. The system essentially learns on its own by going through the dataset and creating a frequency dictionary with Python's Counter class. It keeps track of how many times each word appears, using this count as a probability guide to figure out the most likely correct word. This frequency model helps the application favor commonly used words when suggesting fixes. Being non-parametric and rule-based, the model can correct text in real-time without any extra training. This approach means it can start up quickly, use fewer resources, and still be pretty accurate in practice.

**VII. METHODOLOGY**

The autocorrector is a smart system that relies on a set of rules. It cleverly combines several techniques, such as preparing the text, figuring out how different words are, and analyzing how often words appear. To start, it loads up a massive collection of English text. Then, it cleans things up by turning everything to lowercase and getting rid of punctuation, numbers, and any special characters. After that, it breaks the text down into individual words and creates a dictionary that keeps track of how often each word appears.  
When you type in a word, the system quickly checks if it's in its frequency dictionary. If the word isn't there, it comes up with a list of possible corrections using functions that calculate edit distances, specifically using edits1 for words with one character off, and edits2 for words with two characters off. These edits include adding a character, deleting one, substituting one, or even swapping two around. From the potential corrections that it finds in its dictionary, it picks the one that appears most frequently as the best guess. This method allows for pretty fast and surprisingly accurate spelling correction, all without needing anything overly complicated.

**VIII RESULTS AND DISCUSSION**

The autocorrector app was put through its paces with a bunch of misspelled words to see how accurate and quick it was. Most of the time, especially when the typo was just a letter or two off from a real word, the system did a great job figuring out the right word. Using edit distance, along with word frequency data from everyday language, really helped   
 pinpoint the most likely correction.

The user interface, built with Tkinter, was snappy and easy to use, giving instant feedback to whatever the user typed. The app worked like a charm on less powerful devices, providing corrections almost instantly. While it handled simple and medium typos well, it struggled a bit with more complicated ones or those needing context, since it doesn't look at nearby words or grammar. All in all, the results show that the system is dependable for basic autocorrection, making it a good fit for integration into simple.

**IX CONCLUSION**

This project effectively showcases the creation of a resource-efficient autocorrect application using Python and Tkinter. Through a frequency-based model and algorithms that calculate edit distance, the system adeptly suggests corrections for words entered by users. The simplicity of the text preprocessing and word frequency methods effectively removes the necessity for intricate machine learning models. This allows for swift, real-time functionality even on systems with limited resources. The application highlights fundamental principles of natural language processing and string manipulation, establishing a solid base for developing more sophisticated systems. Future development could explore adding contextual understanding and incorporating deep learning approaches to address intricate grammatical mistakes and enhance prediction precision.

**X REFERENCES**

**[1] P. Norvig, “How to Write a Spelling Corrector,” [Online]. Available: https://norvig.com/spell-correct.html**

**[2] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions and reversals,” *Soviet Physics Doklady*, vol. 10, pp. 707–710, 1966.**

**[3] S. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed., Prentice Hall, 2023.**

**[4] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.**

**[5] Python Software Foundation, “Python Language Reference, Version 3.10,” [Online]. Available:** [**https://www.python.org/**](https://www.python.org/)

**[6] Tkinter Documentation - Python GUI Programming, [Online]. Available:** [**https://docs.python.org/3/library/tkinter.html**](https://docs.python.org/3/library/tkinter.html)

**[7] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*, Cambridge University Press, 2020.**

**[8] M. H. Nadir and M. F. Noor, “An Efficient Spell Checker Using Levenshtein Distance Algorithm,” *International Journal of Computer Applications*, vol. 130, no. 14, pp. 7–11, Nov. 2015.**

**[9] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” *arXiv preprint* arXiv:1301.3781, 2013.**

**[10] M. D. Riley, “Some applications of decision trees to natural language processing,” in *Proceedings of the Second International Workshop on Parsing Technologies*, 1991, pp. 339–352.**