

**SAVEETHA SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

# CAPSTONE PROJECT REPORT

**PROJECT TITLE**

GRAMMAR AUTOCORRECTOR WITH COSINE SIMILARITY USING NLTK LIBRARY

# REPORT SUBMITTED BY

192211106 T.Naveen

**REPORT SUBMITTED TO**

Dr. E. MONIKA

# COURSE CODE / COURSE NAME

CSA1369 / THEORY OF COMPUTATION WITH PROBLEM SOLVING

SLOT C

# DATE OF SUBMISSION

11.09.2024

# ABSTRACT

# This project focuses on grammar correction is a critical task for improving the quality and accuracy of written communication. This paper presents an approach for grammar autocorrection using the Natural Language Toolkit (NLTK) library combined with Cosine Similarity. The proposed system first tokenizes and processes the input text using NLTK, applying part-of-speech tagging and syntactic parsing to identify potential grammatical errors. Following this, a vectorization technique is employed to represent both the erroneous sentence and possible corrected versions as vectors in a high-dimensional space. By calculating the Cosine Similarity between these vectors, the model identifies the closest grammatically correct sentence based on linguistic similarity. The method leverages pre-trained language models and lexical databases to suggest corrections, enhancing both contextual understanding and grammatical accuracy. This approach provides an efficient and scalable solution for automatic grammar correction, with potential applications in educational tools, writing assistants, and automated text evaluation systems.

# INTRODUCTION

Introduction to Grammar Autocorrection Using Cosine Similarity with NLTK

Grammar autocorrection is an essential task in natural language processing (NLP), focused on identifying and correcting grammatical errors in written text. The purpose is to improve the quality of text by ensuring that it adheres to standard grammatical rules, enhancing readability and coherence. Traditional grammar correction techniques often rely on rule-based approaches or machine learning models. However, integrating similarity measures like cosine similarity can help refine the process by comparing the grammatical structure of incorrect sentences with correct alternatives. This combination of approaches allows for more robust error detection and correction.

# Cosine similarity is a measure that calculates the cosine of the angle between two non-zero vectors in an inner product space, often used to determine the similarity between two texts or sentences. In the context of grammar correction, it can be applied by representing sentences as vectors and then comparing them to find the closest match to a correctly structured sentence. When paired with vectorization techniques such as Term Frequency-Inverse Document Frequency (TF-IDF) or word embeddings (e.g., Word2Vec or GloVe), cosine similarity becomes a powerful tool to measure the closeness of sentences and suggest corrections.

The NLTK (Natural Language Toolkit) library provides essential tools for text processing in Python, including tokenization, parsing, and part-of-speech tagging, making it a suitable framework for grammar autocorrection tasks. NLTK can help break down sentences into individual components, identify potential errors, and then suggest corrections by comparing the erroneous sentence to a pool of correct sentences using cosine similarity. This hybrid approach leverages both the syntactic analysis capabilities of NLTK and the semantic comparison power of cosine similarity, allowing for more contextually accurate grammar corrections.

# By combining NLTK's NLP functionalities with cosine similarity, grammar autocorrection can be enhanced to account for not only rule-based corrections but also context and meaning. This approach provides a more flexible solution that can adapt to different writing styles and complex sentence structures. Overall, using cosine similarity in grammar correction systems enriches the process by introducing a method to compare sentence vectors and suggest corrections based on semantic closeness, making the corrections more precise and context-aware.

# LITERATURE REVIEW

Grammar correction systems have evolved significantly over the years, ranging from early rule-based approaches to more sophisticated machine learning-based models. Initially, grammar correctors were built upon predefined rules and language models, which could identify common grammatical errors like subject-verb agreement or improper punctuation **(Chodorow et al., 2010).** However, these rule-based methods were often limited in their flexibility and struggled with more complex linguistic structures. As natural language processing (NLP) evolved, researchers began leveraging probabilistic models and statistical learning techniques to develop more adaptive grammar correction tools. Statistical methods improved accuracy by using large corpora to learn patterns and predict likely corrections based on context (Ng et al., 2014). This shift marked a significant advancement in the field, allowing systems to better handle diverse and ambiguous language inputs.

The integration of similarity measures such as cosine similarity further enhanced the performance of grammar correction systems. Cosine similarity, which measures the cosine of the angle between two non-zero vectors, is widely used in text analysis to compare the similarity between word embeddings or sentence representations **(Manning et al., 2008).** In grammar correction, cosine similarity helps identify the most appropriate corrections by comparing the vectorized representations of erroneous sentences to a reference corpus of grammatically correct sentences. This approach allows for more context-aware corrections, especially when combined with word embeddings techniques like Word2Vec or GloVe, which capture the semantic meaning of words (Mikolov et al., 2013). The use of cosine similarity in NLP tasks has become a standard practice in many modern grammar correction systems.

# The NLTK (Natural Language Toolkit) library, a widely-used Python package for NLP, provides a range of tools for text processing, tokenization, and linguistic analysis (Bird et al., 2009). NLTK supports the implementation of both rule-based and statistical grammar correction systems, offering resources like corpora, syntax parsers, and machine learning models. For instance, researchers can use NLTK to tokenize and preprocess text, calculate cosine similarity scores, and integrate these into grammar correction workflows. By leveraging NLTK's capabilities along with similarity measures, developers can create more robust grammar correction systems that balance traditional rule-based methods with the adaptive capabilities of machine learning and similarity-driven approaches.

# RESEARCH PLAN

The research plan for " Grammar autocoorector with cosine similarity using library " is structured into five key phases. The first phase, Project Initiation and Planning, involves defining the project’s scope and objectives, identifying stakeholders, and establishing a comprehensive project plan with timelines and resource allocation. Next, in the Data Collection and Preprocessing phase, relevant textual data is gathered from various sources and processed using NLTK, including steps such as tokenization, stopword removal, and lemmatization. This ensures that the text data is clean and suitable for analysis. The third phase, Development and Implementation, focuses on implementing text vectorization using TF-IDF and calculating cosine similarity scores to measure text similarity. This phase also ensures that the system is scalable and efficient. In the Testing and Evaluation phase, the system undergoes rigorous testing, including unit tests, integration tests, and user acceptance testing, to ensure accuracy and reliability. Finally, the Documentation, Deployment, and Feedback phase involves documenting the development process, preparing for deployment, and gathering user feedback to refine and enhance the system. This structured approach ensures a thorough and effective implementation of text similarity analysis using cosine similarity and NLTK.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| S.NO | DESCRIPTION | 04.09.2024  DAY-01 | 05.09.2024  DAY-02 | 06.09.2024  DAY-03 | 09.09.2024  DAY-04 | 10.09.2024  DAY-05 |
| 1. | Project Initiation and Planning |  |  |  |  |  |
| 2. | Requirement Analysis and Design |  |  |  |  |  |
| 3. | Development and Implementation |  |  |  |  |  |
| 4. | Testing and Refinement |  |  |  |  |  |
| 5. | Documentation, Deployment, and Feedback |  |  |  |  |  |

**Fig. 1 Timeline chart**

**Day 1: Project Initiation and Planning (1 day)**

* The goal of this project is to develop a tool that can automatically correct grammar using the NLTK (Natural Language Toolkit) library. It will involve. Identifying grammar mistakes in input sentences.
* Implementing a cosine similarity-based approach to find the most appropriate corrections by comparing the input sentence with grammatically correct sentence structures..
* NLTK for text processing and tokenization, Cosine similarity for comparing sentence vectors, additional libraries such as scikit-learn for vectorization and similarity measurement. A corpus of correct sentences to serve as a comparison database (e.g., a dataset from linguistic sources).

**Day 2: Requirement Analysis and Design (1 day)**

* Gather relevant textual data for analysis, which may include sample texts or datasets from available repositories.
* Implement preprocessing steps using NLTK, including tokenization, stopword removal, and lemmatization, to ensure that text data is clean and ready for similarity analysis.
* Finalize the preprocessing pipeline and verify its effectiveness in standardizing text data for further analysis.

**Day 3: Development and Implementation (2 days)**

* Develop the core functionality of the text similarity system, focusing on implementing TF-IDF vectorization and cosine similarity calculations. Utilize Python libraries such as NLTK and scikit-learn for efficient text processing and similarity measurement.
* Code and test the core functionalities, including text vectorization and similarity scoring. Ensure that the system can handle various text inputs and accurately compute similarity scores.
* Integrate additional features such as threshold-based classification to label text pairs as "similar" or "not similar."

**Day 4: Testing and Refinement (1 day)**

* Conduct comprehensive testing, including unit tests for individual components, integration tests to ensure proper functionality across the system, and user acceptance testing to validate the system’s effectiveness.
* Identify and resolve any bugs or issues discovered during testing, and gather feedback from initial users to address usability concerns and performance issues.
* Refine the system based on feedback and testing results to ensure that it meets the project's objectives and performs reliably.

**Day 5: Documentation, Deployment, and Feedback (1 day)**

# Document the entire development process, including methodologies for text preprocessing, vectorization, and similarity calculation, as well as the testing and evaluation procedures.

# Prepare the system for deployment, ensuring that it is configured for optimal performance and usability.

# Deploy the system in a testing environment for final validation and gather feedback from stakeholders and end-users to assess its effectiveness in recognizing similar texts. Evaluate the project’s success in achieving its objectives and identify areas for future improvement.

# METHODOLOGY

# Step 1: Download and install Python from the official website <https://www.python.org/downloads/>. Choose the version compatible with your operating system.

# Step2: Open the command prompt or terminal and install the necessary Python libraries for the project. Execute the following commands:

**pip install nltk**

**pip install scikit-learn**

# Step 3: Open a Python interpreter or script and import the NLTK library. Run the following commands to download the necessary NLTK data:

**import nltk**

**nltk.download('punkt')**

**nltk.download('stopwords')**

**nltk.download('wordnet')**

# Step 4: Preprocess Text Data

* Collect the text data you want to analyze and store it in a text file or a Python list.
* Create a Python script to preprocess the text using NLTK. Implement functions for tokenization, stopword removal, and lemmatization. Ensure your script saves the preprocessed text for further analysis.

# Step 5: Implement Cosine Similarity Calculation

* In your Python script, implement TF-IDF vectorization using scikit-learn. Write functions to convert the preprocessed text into TF-IDF vectors.
* Calculate cosine similarity between text vectors using scikit-learn's cosine\_similarity function. Ensure your script includes functionalities to compute and display similarity scores.

# Step 6: Test and Validate the System

* Run your Python script to preprocess the text data, compute similarity scores, and display the results.
* Verify the correctness of the similarity calculations by comparing the output with expected results. Ensure that the system handles various text inputs effectively.

# Step 7: Refine and Optimize

* Based on testing results, make any necessary adjustments to improve the accuracy and efficiency of the text similarity analysis.
* Optimize preprocessing and vectorization steps to handle larger datasets and improve performance.

# Step 8: Document and Review

* Document the development process, including the implementation details for preprocessing, vectorization, and similarity calculation.
* Review the results and gather feedback from potential users or stakeholders to assess the effectiveness of the system in recognizing similar texts.

# PYTHON CODE:

# import nltk

# from nltk.corpus import words

# from nltk.tokenize import word\_tokenize

# from sklearn.feature\_extraction.text import CountVectorizer

# from sklearn.metrics.pairwise import cosine\_similarity

# # Download necessary NLTK data

# nltk.download('punkt')

# nltk.download('words')

# # List of English words

# word\_list = words.words()

# # Sample correct sentences (you can add more sentences)

# correct\_sentences = [

# "The cat is on the mat.",

# "The quick brown fox jumps over the lazy dog.",

# "She sells sea shells by the sea shore.",

# "I am going to the market."

# ]

# def cosine\_similarity\_text(word1, word2):

# """Calculate cosine similarity between two words."""

# vectorizer = CountVectorizer().fit\_transform([word1, word2])

# vectors = vectorizer.toarray()

# cos\_sim = cosine\_similarity(vectors)

# return cos\_sim[0][1]

# def correct\_word(word):

# """Find the closest word based on cosine similarity."""

# max\_sim = 0

# best\_word = word

# for candidate in word\_list:

# sim = cosine\_similarity\_text(word, candidate)

# if sim > max\_sim:

# max\_sim = sim

# best\_word = candidate

# return best\_word

# def autocorrect\_sentence(input\_sentence):

# """Autocorrect words in the sentence and suggest the closest correct sentence."""

# input\_words = word\_tokenize(input\_sentence)

# # Correct individual words

# corrected\_sentence = []

# for word in input\_words:

# if word.lower() in word\_list:

# corrected\_sentence.append(word)

# else:

# corrected\_word = correct\_word(word.lower())

# corrected\_sentence.append(corrected\_word)

# corrected\_sentence = ' '.join(corrected\_sentence)

# # Find the closest correct sentence

# max\_sim = 0

# best\_sentence = corrected\_sentence

# for sentence in correct\_sentences:

# sim = cosine\_similarity\_text(corrected\_sentence, sentence)

# if sim > max\_sim:

# max\_sim = sim

# best\_sentence = sentence

# return best\_sentence

# # Test the autocorrector

# input\_sentence = "The kat is on the mat."

# corrected\_sentence = autocorrect\_sentence(input\_sentence)

# print("Corrected Sentence:", corrected\_sentence)

**OUTPUT**

# CONCLUSION

Using cosine similarity with the NLTK library to correct grammar is an innovative approach that leverages the power of vector space models in natural language processing. By representing sentences or phrases as vectors in a high-dimensional space, cosine similarity allows for the comparison of textual units based on their contextual similarity rather than mere surface-level matching. This method helps in identifying and correcting grammatical errors by comparing the input text with a corpus of correctly constructed sentences, thereby providing more accurate suggestions for grammatical improvements.

The NLTK library, with its robust set of tools for text processing and linguistic analysis, facilitates the implementation of cosine similarity for grammar correction. It provides functionalities for tokenization, part-of-speech tagging, and vector space modeling, which are crucial for calculating similarity scores between text units. By leveraging these tools, one can build a system that not only detects grammatical errors but also suggests contextually appropriate corrections, enhancing the overall quality of the text.

In conclusion, employing cosine similarity with the NLTK library for grammar correction presents a sophisticated approach to improving text quality. It harnesses the strengths of vector space models to provide context-aware corrections, offering a more nuanced and accurate grammar-checking process. This method represents a significant advancement over traditional rule-based grammar checkers, aligning with modern trends in natural language processing and machine learning.

# REFERENCES

1. **Priya, Shunmuga, D. Karthika Renuka, and L. Ashok Kumar**. "Towards improving speech recognition model with post-processing spell correction using BERT." Journal of Intelligent & Fuzzy Systems 43.4 (2022): 4873-4882.
2. **Priya, M. S., Renuka, D. K., Kumar, L. A., & Rose, S. L. (2022).** Multilingual low resource Indian language speech recognition and spell correction using Indic BERT. Sādhanā, 47(4), 227.Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013).
3. **Dua, Mohit, Bhavesh Bhagat, and Shelza Dua**. "An amalgamation of integrated features with DeepSpeech2 architecture and improved spell corrector for improving Gujarati language ASR system." International Journal of Speech Technology (2024): 1-13.
4. **Bhagat, Bhavesh, and Mohit Dua**. "Improved spell corrector algorithm and deepspeech2 model for enhancing end-to-end Gujarati language ASR performance." e-Prime-Advances in Electrical Engineering, Electronics and Energy 7 (2024): 100441.
5. **Patil, Kavita T., Ram P. Bhavsar, and B. V. Pawar.** "Contrastive study of minimum edit distance and cosine similarity measures in the context of word suggestions for misspelled Marathi words." Multimedia Tools and Applications 82.10 (2023): 15573-15591.
6. **Kedia, A., & Rasu, M. (2020).** Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications. Packt Publishing Ltd.Jurafsky, D., & Martin, J. H. (2021).
7. ***Pal, S., Pramanik, P. K. D., Maity, A., & Choudhury, P. (2021****). Learner question’s correctness assessment and a guided correction method: enhancing the user experience in an interactive online learning system. PeerJ Computer Science, 7, e532.*
8. **Hapke, Hannes, Cole Howard, and Hobson Lane.** Natural Language Processing in Action: Understanding, analyzing, and generating text with Python. Simon and Schuster, 2019.
9. **Shetty, Pramit, Kaushal Yadav, and Prithvi Kunder.** "Automated essay grading system using NLP techniques." J. IJEAT 9.5 (2020): 1033-1042.
10. **Shukla, K., Vashishtha, E., Sandhu, M., & Choubey, R. (2023).** Natural Language Processing: Unlocking the Power of Text and Speech Data. Xoffencer International Book Publication House, 251.