MAR ATHANASIUS COLLEGE OF ENGINEERING
(Affiliated to APJ Abdul Kalam Technological University,TVM)
KOTHAMANGALAM

Department of Computer Applications

Mini Project Report

GRAMMAR ERROR DETECTION AND CORRECTION USING BERT

Done by

Zahariya P R

Reg No:MAC23MCA-2060

Under the guidance of
Prof. Sonia Abraham

2023-2025

MAR ATHANASIUS COLLEGE OF ENGINEERING
(Affiliated to APJ Abdul Kalam Technological University,TVM)
KOTHAMANGALAM

CERTIFICATE

Grammar Error Detection and Correction using BERT

Certified that this is the bonafide record of project work done by

Zahariya P R
Reg No: MAC23MCA-2060

during the third semester, in partial fulfilment of requirements for award of the degree

Master of Computer Applications

of

APJ Abdul Kalam Technological University Thiruvananthapuram

Faculty Guide	**Head of the Department**
Prof. Sonia Abraham	Prof. Biju Skaria
Project Coordinator	**Internal Examiners**
Prof. Sonia Abraham	

ACKNOWLEDGEMENT

With heartfelt gratitude, I extend my deepest thanks to the Almighty for His unwavering grace and blessings that have made this journey possible. May His guidance continue to illuminate my path in the years ahead.

I am immensely thankful to Prof. Biju Skaria, Head of the Department of Computer Applications and Prof. Sonia Abraham, our dedicated project coordinator, for their invaluable guidance and timely advice, which played a pivotal role in shaping this project. Their guidance, constant supervision, and provision of essential information were instrumental in the successful completion of the mini project.

I extend my profound thanks to all the professors in the department and the entire staff at MACE for their unwavering support and inspiration throughout my academic journey. My sincere appreciation goes to my beloved parents, whose guidance has been a beacon in every step of my path.

I am also grateful to my friends and individuals who generously shared their expertise and assistance, contributing significantly to the fulfilment of this endeavour.

ABSTRACT

The project focuses on developing a grammar error detection and correction system using BERT (Bidirectional Encoder Representations from Transformers), a state-of-the-art deep learning model in the field of natural language processing (NLP). This task is fundamental as it aims to identify and rectify grammatical mistakes in written text, enhancing the quality and clarity of communication. Given the exponential growth of digital content and the widespread use of the internet, automated grammar checking tools have become increasingly indispensable. BERT leverages vast datasets and the powerful Transformer architecture to understand the context of words in a sentence by considering the entire sequence of words bidirectionally.This allows BERT to learn nuanced language patterns and make accurate corrections. The relevance of this project lies in addressing the need for efficient and reliable grammar checking tools, crucial for various applications, including content creation, education, and professional writing.

In conclusion, the development of a grammar error detection and correction system using BERT marks a significant advancement in natural language processing. By leveraging deep learning and the Transformer architecture, this project overcomes the limitations of traditional rule-based methods, providing a more accurate and context-aware solution for grammatical corrections.This project stands as a testament to the potential of machine learning in improving our interactions with technology and each other, paving the way for more sophisticated and reliable language processing systems in the future.

The dataset contains 2018 entries and 4 columns:Serial Number,.Error Type,Ungrammatical Statement,Standard English.

LIST OF TABLES

LIST OF FIGURES

CONTENTS

1. INTRODUCTION

This project aims to develop a grammar error detection and correction system using BERT (Bidirectional Encoder Representations from Transformers) and deep learning techniques. BERT, a powerful transformer-based model, is fine-tuned on a dataset comprising sentences with various grammatical errors and their corrected versions. By leveraging BERT's ability to understand the context of words within sentences, the system is trained to identify and rectify a wide range of grammatical issues, including verb tense errors, subject-verb agreement, and punctuation mistakes. The result is an automated tool that enhances the accuracy and quality of written text, making it a valuable asset for applications requiring real-time grammar correction.

BERT, introduced by Google in 2018, is a state-of-the-art transformer-based model that has revolutionized the field of NLP. Unlike traditional models that process text in a unidirectional manner, BERT is designed to consider the context of a word from both its left and right sides, allowing for a more nuanced understanding of the text. This bidirectional nature enables BERT to capture the intricacies of language, making it exceptionally well-suited for tasks that require deep contextual understanding, such as grammar error detection and correction.

The core objective of this project is to leverage BERT's capabilities to create an automated system that can identify and correct grammatical errors in real-time. The system is trained on a large and diverse dataset that includes sentences with various grammatical mistakes, along with their corrected versions. This dataset serves as the foundation for fine-tuning the BERT model, allowing it to learn the patterns and rules of grammar. By doing so, the model becomes proficient in detecting a wide range of grammatical issues, including verb tense errors, subject-verb agreement discrepancies, punctuation mistakes, and more.

The process begins with the collection of text data, which is then preprocessed to make it suitable for model training. Preprocessing steps typically involve tokenization, where the text is broken down into smaller units (tokens) that the model can understand. BERT's tokenizer is designed to handle complex language structures, ensuring that the text is accurately represented. Once the data is preprocessed, it is fed into the BERT model, where the real work begins. The model processes the text, identifies grammatical errors, and generates corrected versions of the sentences.

In conclusion, this project represents a significant step forward in the development of intelligent tools for improving written communication. By harnessing the power of BERT and deep learning, we aim to create a system that not only detects grammatical errors but also provides accurate corrections, thereby enhancing the overall quality of written text. The result is a robust, efficient, and user-friendly tool that can benefit individuals and organizations alike, ensuring that their written communication is both clear and grammatically sound.

2. SUPPORTING LITERATURE

2.1. LITERATURE REVIEW

Paper 1: [1]Yu Qing ”Design And Application Of Automatic English Translation Grammar Error Detection System Based On BERT Machine Vision”, Scalable Computing: Practice and Experience, ISSN 1895-1767, 2024 SCPE.

The paper discusses the design and application of an automatic grammar error detection and correction system for English translation based on the BERT (Bidirectional Encoder Representations from Transformers) model optimized with machine vision techniques. Traditional methods of error detection, which often involve manual proofreading, are labor-intensive and prone to inefficiencies. The proposed system leverages the Transformer model's architecture, integrating a mixed attention mechanism to better capture contextual dependencies.

Table 2.1 Summary of paper 1

Title of the paper	Design & Application of Automatic English Translation Grammar Error Detection System based on BERT Machine Vision
Area of work	Natural Language Processing, Machine Vision
Dataset	CoLA dataset, annotated NUCLE, and FCE corpora
Methodology / Strategy	Combines BERT model with mixed attention mechanism and multi-scale feature extraction using cavity convolutions. TFIDF weighting enhances feature extraction.
Algorithm	BERT, Transformer with mixed attention module
Result/Accuracy	Achieves a loss value of 0.2639 and an accuracy rate of 96%

Paper 2: [2]Nancy Agarwal, Mudasir Ahmad Wani and Patrick Bours, "Lex-Pos Feature-Based Grammar Error Detection System for the English Language.

The paper focuses on improving the detection of articles ('a', 'an', and no article) in English sentences using Natural Language Processing (NLP) and Deep Learning techniques. It utilizes a custom-designed dataset and explores different sequence types, including lexical sequences, POS-tag sequences, and Lex-Pos sequences, represented with Word Embedding and one-hot Encoding (WEOE). The CNN-based deep learning model used achieved 99% accuracy with the Lex-Pos sequence and 90% with the Context-based Lexical Sequence model. The approach combines the stability of POS-based models and the effectiveness of lexical-based models, though it may require more computational resources. Future work includes extending the method to other grammatical constructs and applying it to larger, more diverse datasets.

Table 2.2 Summary of paper 2

Title of the paper	Lex-Pos Feature-Based Grammar Error Detection System for the English Language.
Area of work	Natural Language Processing (NLP), Deep Learning
Dataset	Custom dataset designed for this study, containing labels 0, 1, and 2 for sentences with 'a', 'an', or no article respectively
Methodology / Strategy	Comparison of different sequence types (lexical sequences, POS-tag sequences, Lex-Pos sequences) using Word Embedding and one-hot Encoding (WEOE) representation.
Algorithm	CNN-based deep learning model with convolution and pooling layers
Result/Accuracy	Lex-Pos Sequence model: 99% accuracy, Context-based Lexical Sequence model: 90% accuracy

Paper 3: [3]Jared Lichtarge,Chris Alberti,Shankar, "Data Weighted Training Strategies for Grammatical Error Correction" Transactions of the Association for Computational Linguistics, vol. 8, pp. 634–646, 2020.

The paper titled "Optimizing Training Strategies for Pretrain-Finetune Paradigms in GEC" explores various strategies to enhance the performance of Grammatical Error Correction (GEC) models. It focuses on comparing different pretrain-finetune arrangements and scoring strategies, utilizing delta-log-perplexity (Δppl) as a heuristic for evaluating data quality. The datasets used include PRE (Pre Training data), Lang-8, BF (BEA-FCE combined dataset), BEA-19 dev, FCE, REV, and RT. The study evaluates several training strategies, such as hard, soft, hard-cclm, and soft-cclm, achieving the best F0.5 score of 52.4. While the optimized training strategies show enhanced model performance, the variability in dataset quality impacts their effectiveness.

Table 2.3 Summary of paper 3

Title of the paper	Data Weighted Training Strategies for Grammatical Error Correction.
Area of work	Grammatical Error Correction (GEC) in Natural Language Processing (NLP).
Dataset	PRE (Pre training data), Lang-8, BF (BEA-FCE combined dataset), BEA-19 dev, FCE, REV, RT.
Methodology / Strategy	Comparing pretrain-finetune arrangements and scoring strategies using delta-log-perplexity.
Algorithm	Various training strategies including hard, soft, hard-cclm, and soft-cclm.
Result/Accuracy	Best setup achieving F0.5 of 52.4

2.2. LITERATURE REVIEW SUMMARY

The three papers collectively explore advanced methodologies for improving grammatical error detection and correction systems using state-of-the-art machine learning techniques.

1. **Methodologies**:
    - The integration of Transformer and BERT models, along with mixed attention modules, enhances feature extraction and context modeling.
    - The utilization of delta-log-perplexity scoring effectively weights training examples based on their difficulty, improving model robustness
2. **Datasets**:
    - Various datasets like Lang-8, BEA-2019, and FCE are used, providing a mix of non-native English writing samples and annotated corrections.
3. **Algorithms**:
    - A combination of traditional machine learning models (e.g., Logistic Regression, Decision Trees) and advanced models (e.g., Random Forest, Gradient Boosting Machines, XGBoost).
    - Deep learning models such as LSTM, CNN, and Transformer-based models (e.g., BERT, TF-BERT) are highlighted for their superior performance in NLP tasks.
4. **Results and Advantages**:
    - The studies demonstrate state-of-the-art results in grammatical error correction, indicating significant improvements in accuracy and robustness.
    - The proposed methodologies offer enhanced feature extraction, effective context modeling, and robust handling of data sparsity.

Overall, these papers provide a comprehensive framework for developing advanced grammatical error detection and correction systems by leveraging powerful machine learning and deep learning techniques, effective data weighting strategies, and robust evaluation methods.

2.3. FINDINGS AND PROPOSALS

The quality of written communication is crucial in various domains, from education to professional settings. Grammatical errors can undermine the clarity and impact of the text, leading to misunderstandings. This project proposes the development of an advanced grammar error detection and correction system using BERT (Bidirectional Encoder Representations from Transformers) combined with deep learning classifiers and algorithms to accurately identify and correct grammatical errors in text.

Objectives:

- To develop a robust system that can automatically detect and correct grammatical errors using BERT and suitable deep learning algorithms.
- To fine-tune BERT on a labeled dataset with grammatical errors and their corrections, enabling it to handle a wide range of grammatical issues such as verb tense, subject-verb agreement, and punctuation.
- To integrate classifiers and algorithms such as transformers, LSTM (Long Short-Term Memory), and attention mechanisms to enhance the system's accuracy in error detection and sentence correction.

Methodology:

1. Data Collection and Preprocessing:

- Collect and preprocess a dataset containing sentences with various grammatical errors and their corresponding correct versions.
- Ensure the dataset includes diverse error types to allow the model to generalize well across different grammatical issues.

2. Model Architecture:

- **BERT Fine-Tuning:**

  - Fine-tune the BERT model on the prepared dataset. BERT's contextual understanding of sentences will be leveraged to detect subtle grammatical errors.

- **Deep Learning Algorithms:**

  - Transformers: Use transformers for their ability to model long-range dependencies in text, which is essential for understanding the context in complex sentences.
  - LSTM: Implement LSTM networks to capture sequential patterns and dependencies between words, particularly in handling errors related to sentence structure and word order.
  - Attention Mechanism: Apply attention mechanisms to focus on specific parts of the sentence that are more likely to contain errors, improving the model's correction capabilities.

3. Training Strategy:

- Use transfer learning to fine-tune BERT, starting with a pre-trained model and adapting it to the specific task of grammar correction.

- Train the LSTM and transformer models on sequences of incorrect and correct sentences, optimizing them using techniques like teacher forcing and reinforcement learning.

- Combine the outputs of these models using an ensemble approach, where multiple classifiers work together to improve overall performance.

4. Evaluation:

- Evaluate the model’s performance using metrics such as accuracy, precision, recall, F1-score, and BLEU (Bilingual Evaluation Understudy) score.

- Conduct a comparative analysis against existing grammar correction tools to benchmark the system’s effectiveness.

- Perform cross-validation and hyperparameter tuning to further refine the model's performance.

5. Implementation:

- Deploy the model as a real-time grammar correction tool that can be integrated into word processors, educational software, and online platforms.

- Ensure scalability and user-friendliness of the system, allowing it to handle various text inputs with minimal latency.

Conclusion:

This project leverages state-of-the-art deep learning algorithms and classifiers, including BERT, LSTM, transformers, and attention mechanisms, to create a comprehensive grammar error detection and correction system. The resulting tool will be a significant advancement in NLP, providing precise and reliable grammar correction across various applications.

3. SYSTEM ANALYSIS

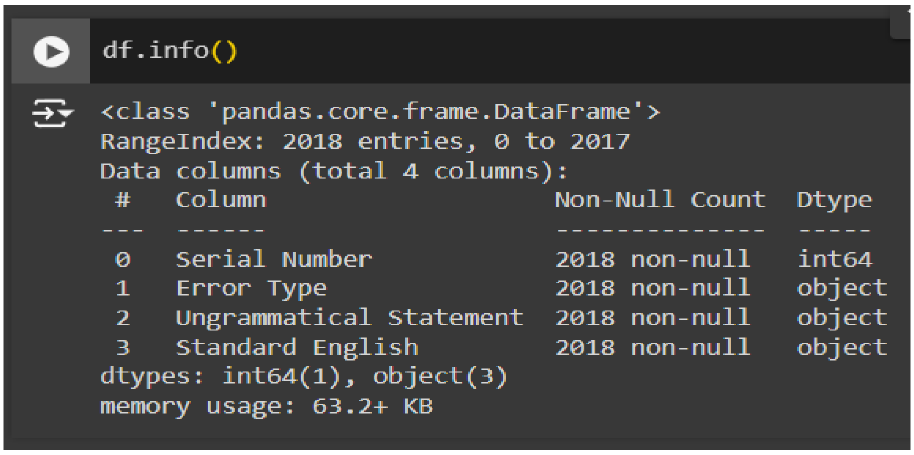
3.1. ANALYSIS OF DATASET

3.1.1. About the Dataset

The dataset contains 36 distinct error types,

Verb Tense Errors, Subject-Verb Agreement, Article Usage, Preposition Usage, Sentence Structure Errors, Spelling Mistakes, Punctuation Errors, Capitalization Errors, Word Choice/Usage, Run-on Sentences, Sentence Fragments, Redundancy/Repetition, Mixed Metaphors/Idioms, Passive Voice Overuse, Pronoun Errors, Conjunction Misuse, Modifiers Misplacement, Agreement in Comparative and Superlative Forms, Parallelism Errors, Quantifier Errors, Tautology, Inappropriate Register, Ambiguity, Mixed Conditionals,Faulty Comparisons, Incorrect Auxiliaries, Negation Errors, Ellipsis Errors, Slang, Jargon, and Colloquialisms, Clichés, Abbreviation Errors, Contractions Errors, Relative Clause Errors, Infinitive Errors, Gerund and Participle Errors and Lack of Parallelism in Lists or Series.

Dataset link: **https://www.kaggle.com/datasets/satishgunjal/grammar-correction**

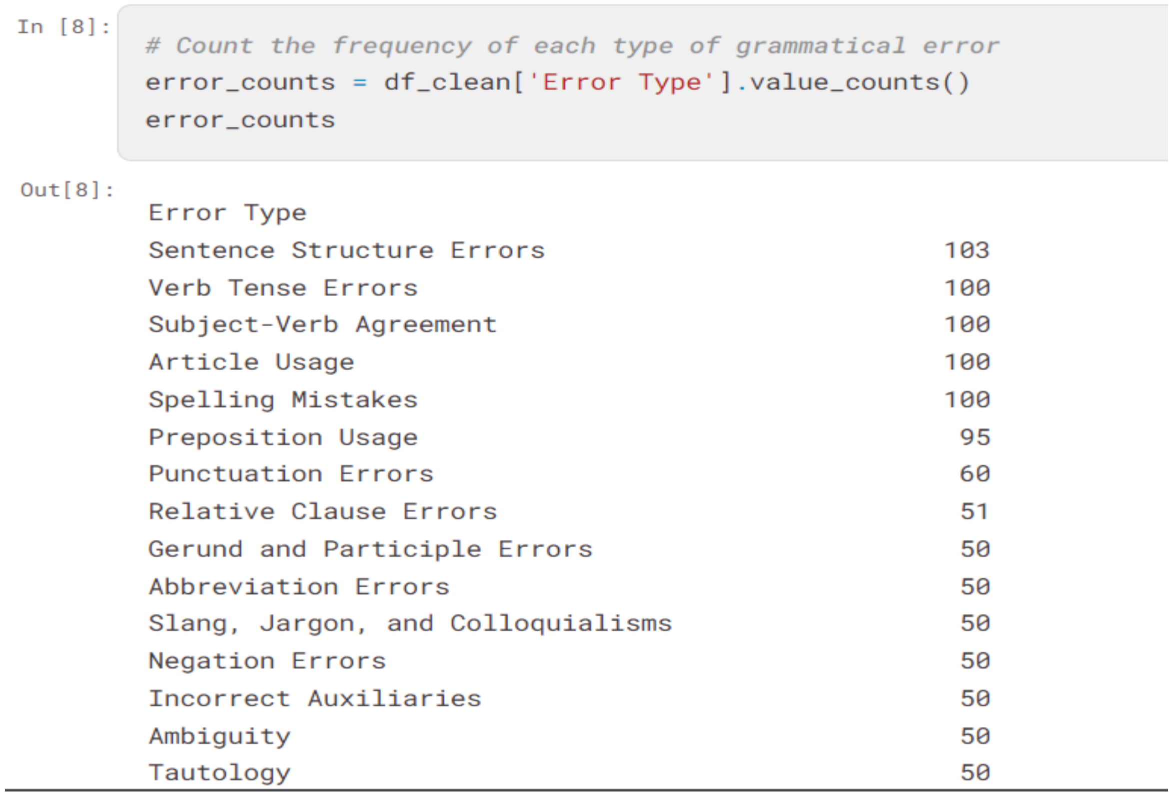


```
df.info()
```

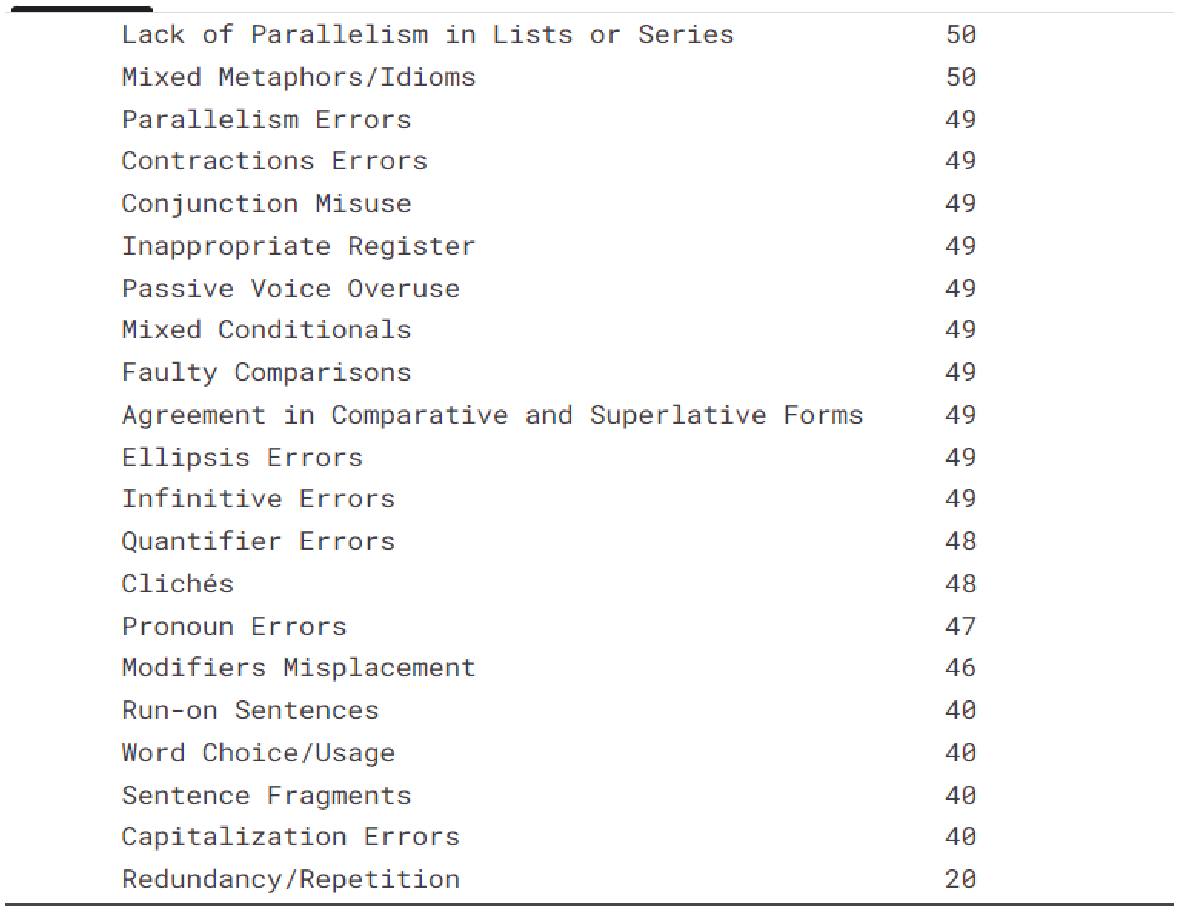
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2018 entries, 0 to 2017
Data columns (total 4 columns):
 #   Column                  Non-Null Count  Dtype 
---  ------                  --------------  ----- 
 0   Serial Number           2018 non-null   int64 
 1   Error Type              2018 non-null   object
 2   Ungrammatical Statement 2018 non-null   object
 3   Standard English        2018 non-null   object
dtypes: int64(1), object(3)
memory usage: 63.2+ KB
```

Figure 3.1 info of grammar-correction.csv

3.1.2. Explore the Dataset



```
In [8]:  # Count the frequency of each type of grammatical error
         error_counts = df_clean['Error Type'].value_counts()
         error_counts

Out[8]:
         Error Type
         Sentence Structure Errors                        103
         Verb Tense Errors                                100
         Subject-Verb Agreement                           100
         Article Usage                                    100
         Spelling Mistakes                                100
         Preposition Usage                                 95
         Punctuation Errors                                60
         Relative Clause Errors                            51
         Gerund and Participle Errors                      50
         Abbreviation Errors                               50
         Slang, Jargon, and Colloquialisms                 50
         Negation Errors                                   50
         Incorrect Auxiliaries                             50
         Ambiguity                                         50
         Tautology                                         50
```



```
         Lack of Parallelism in Lists or Series           50
         Mixed Metaphors/Idioms                           50
         Parallelism Errors                               49
         Contractions Errors                              49
         Conjunction Misuse                               49
         Inappropriate Register                           49
         Passive Voice Overuse                            49
         Mixed Conditionals                               49
         Faulty Comparisons                               49
         Agreement in Comparative and Superlative Forms   49
         Ellipsis Errors                                  49
         Infinitive Errors                                49
         Quantifier Errors                                48
         Clichés                                          48
         Pronoun Errors                                   47
         Modifiers Misplacement                           46
         Run-on Sentences                                 40
         Word Choice/Usage                                40
         Sentence Fragments                               40
         Capitalization Errors                            40
         Redundancy/Repetition                            20
```

Figure 3.2 Error types
Number of unique error types occur in this dataset.

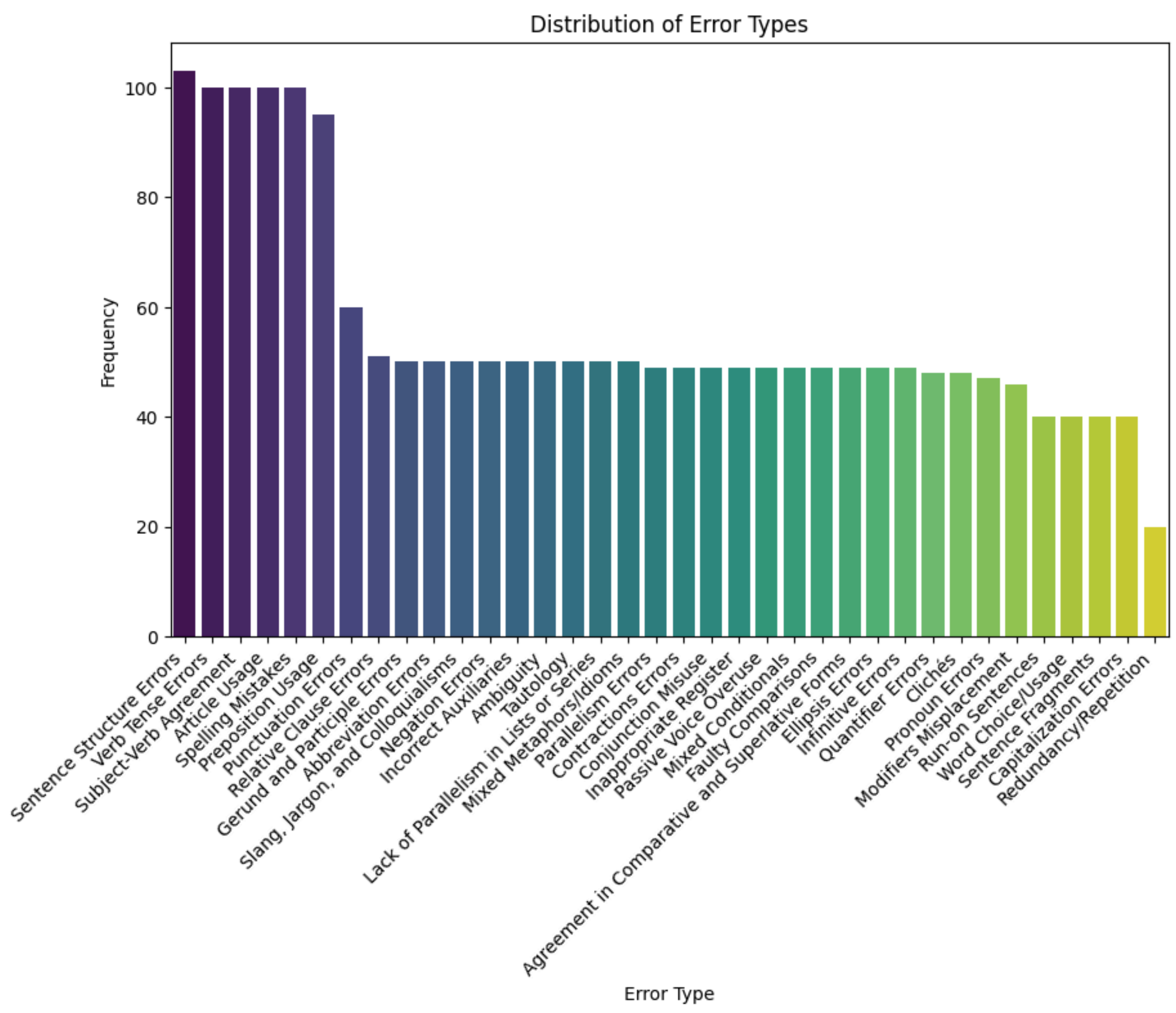


Figure 3.3 Distribution of error types

This bar graph displays the distribution of various types of grammatical errors in a dataset or analysis. The errors are grouped by frequency, represented on the y-axis, and type, labeled on the x-axis. The color scheme appears to categorize errors into groups, indicated by different shades:

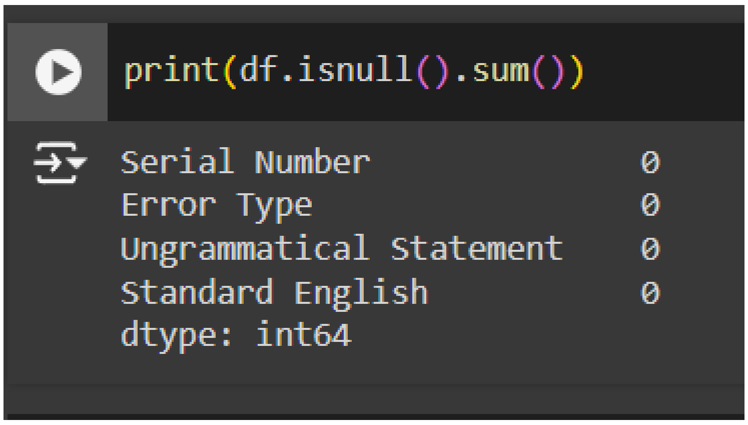
1. Purple shades: These represent structural and verb-related errors, such as "Sentence Structure Errors," "Subject-Verb Agreement Errors," and "Verb Tense Errors."
2. Teal shades: These include errors related to nouns, pronouns, articles, and prepositions, like "Preposition Errors," "Article Errors," and "Pronoun Errors."
3. Green shades: This category includes stylistic and less frequent grammatical issues, such as "Modifier Errors," "Word Choice Errors," and "Redundancy/Repetition."

The error types on the x-axis range from very specific grammatical issues (like "Verb Tense Usage" and "Article Misuse") to more complex or stylistic concerns ("Word Choice Errors" and "Redundancy/Repetition"). The graph helps visualize which error types are most prevalent, which can inform areas of focus for teaching, learning, or correcting grammar, potentially useful in developing or refining grammar-checking tools or educational curricula.

3.2. DATA PREPROCESSING

3.2.1. Data Cleaning

Missing Values



```
print(df.isnull().sum())
```

```
Serial Number              0
Error Type                 0
Ungrammatical Statement    0
Standard English           0
dtype: int64
```

Figure 3.4 Check for missing values

There are no missing values in the dataset.

3.2.2. Analysis of Feature Variables

Feature variables in the dataset includes:

- Serial Number : uniquely identifies each data entry.
- Error Type : categorizes the grammatical mistake present.
- Ungrammatical Statement : contains the sentence with the error.
- Standard English : provides the corrected version of the statement.

3.3. DATA VISUALIZATION

- **Visualization of the sentence lengths in both columns:**

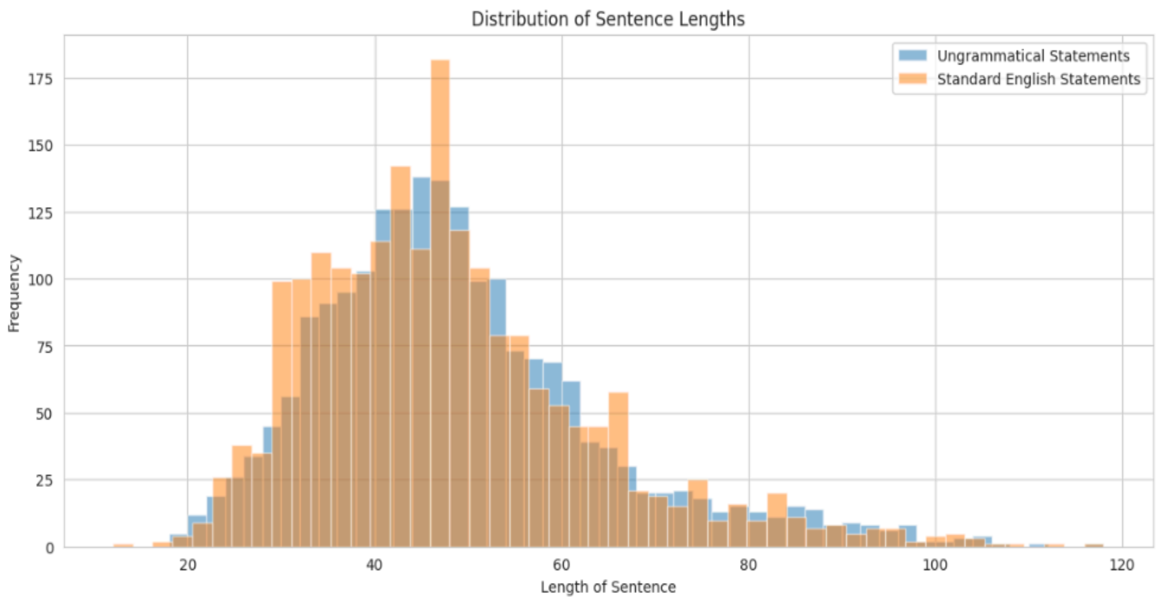


Figure 3.5 Distribution of Sentence Lengths

This image shows a bar graph comparing the distribution of sentence lengths for ungrammatical statements and standard English statements.

Key observations:

1. The distribution for both categories appears to be roughly normal (bell-shaped), with a peak around 40-50 words per sentence.

2. Standard English statements (orange) seem to have a slightly higher peak and are more concentrated around the middle lengths.

3. Ungrammatical statements (blue) appear to have a slightly flatter distribution, with more sentences at the extremes (very short or very long).

4. The most common sentence length for both categories is around 45-50 words.

5. There are very few sentences shorter than 20 words or longer than 100 words for both categories.

- **Word Frequency Analysis:**

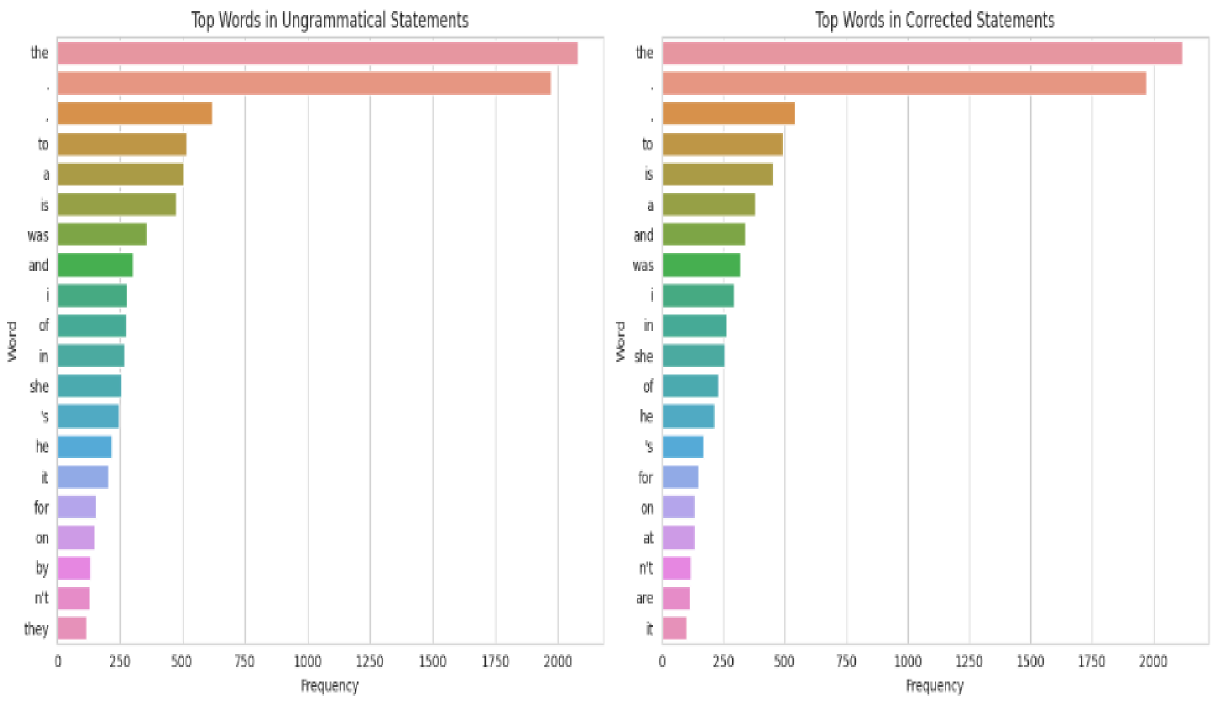


Figure 3.6 Distribution of word frequency

This image shows two bar charts side by side, comparing the most frequent words in ungrammatical statements (left) and corrected statements (right).

Key observations:

1. Both charts display the top 20 most frequent words in each category.

2. The x-axis represents the frequency of each word, while the y-axis lists the words themselves.

3. In both charts, "the" is the most frequent word, followed by "." (likely representing periods or full stops).

4. The top 5 words are identical in both charts and in the same order: "the", ".", "to", "a", "is".

5. There are slight differences in the order and presence of words between the two charts after the top 5:

   - "was" appears higher in the ungrammatical statements.

   - "she" is present in the ungrammatical top 20 but not in the corrected version.

   - "he" and "are" appear in the corrected statements but not in the ungrammatical top 20.

6. Most of the top words are function words (articles, prepositions, pronouns) rather than content words.

7. The frequency distributions appear similar between the two charts, suggesting that the most common words are used with similar frequency in both grammatical and ungrammatical statements.

8. Some interesting differences:

- "n't" (contraction for "not") appears in the ungrammatical statements but not in the corrected ones.
- "it" appears in the corrected statements but not in the ungrammatical top 20.

This comparison provides insights into the similarities and subtle differences in word usage between ungrammatical and grammatically correct statements. It suggests that the most frequent words remain largely the same, but there are some minor shifts in usage that might be indicative of grammatical errors or corrections.

- **N-gram Analysis:**

N-gram analysis involves examining and analyzing the frequency of pairs (bi-grams) or triplets (tri-grams) of adjacent words in a text.

This analysis can be particularly insightful for identifying commonly misused phrases or context-specific errors in 'Ungrammatical Statement'.

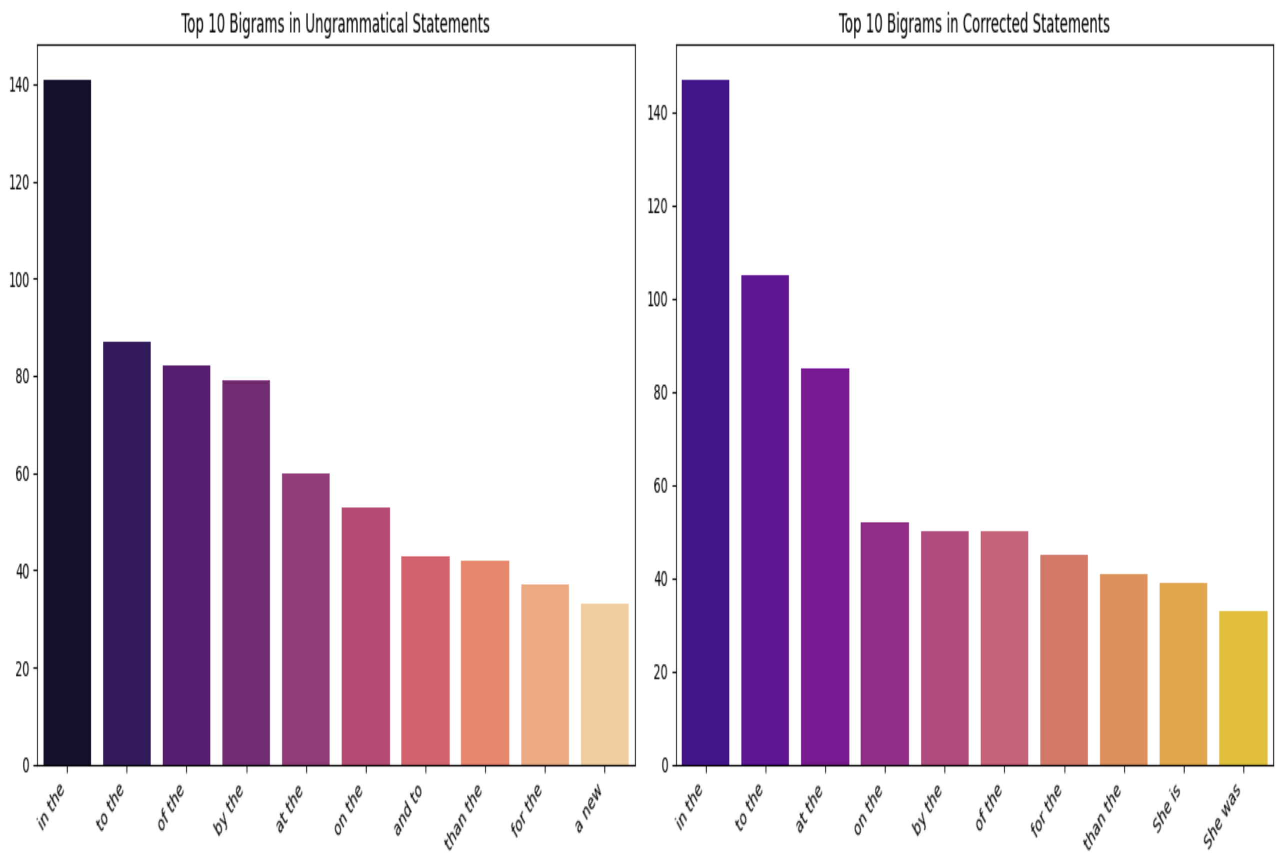


Figure 3.7 Bigrams Analysis

Bi-grams Analysis:

- Common Pairings: The bi-grams like ('in', 'the'), (',', 'and'), and (',', 'but') are very common. These are often used in English and may indicate frequent issues with prepositions or conjunctions in sentences.
- Contextual Clues: Pairings such as ('to', 'the') and ('of', 'the') suggest that errors might occur around prepositional phrases, possibly relating to the incorrect use of prepositions or article-noun agreement.
- Specific Examples: The bi-gram ('it', "'s") could indicate issues with contractions or possessive forms.

Tri-grams Analysis:

- Conjunctions and Commas: The presence of tri-grams like (',', 'and', 'to') and (',', 'but', 'it') indicate that errors could be associated with the use of conjunctions and punctuation in compound or complex sentences.
- Phrase Patterns: Phrases such as ('went', 'to', 'the') and ('to', 'the', 'store') might highlight errors in common phrases, possibly indicating misusage in simple sentence constructions.
- Specific Contexts: Tri-grams like ('the', 'movie', 'was') and ('the', 'group', 'of') could point to particular contexts or subjects where errors are more frequent.

- **Correction Pattern Mining:**

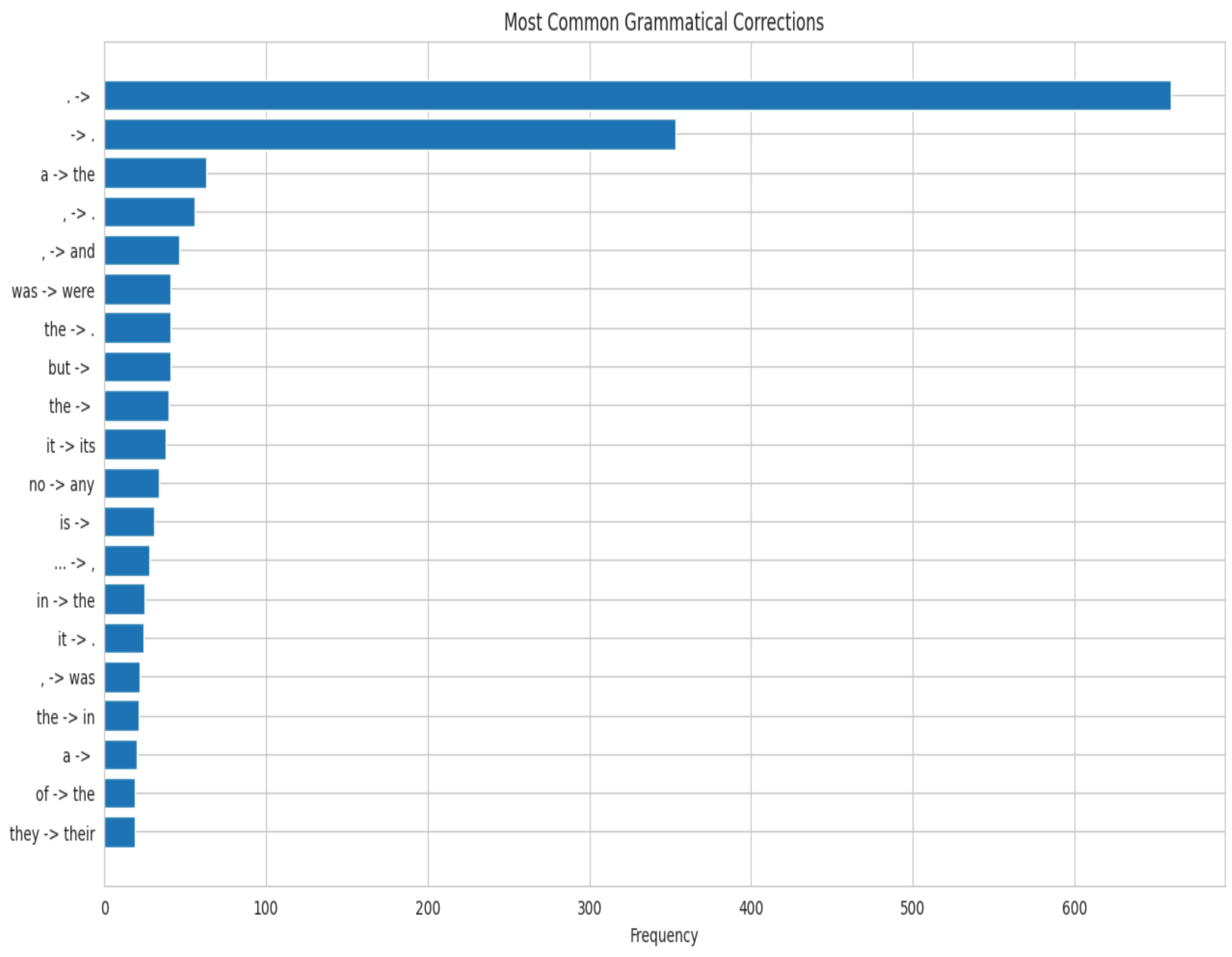


Figure 3.8 Distribution of Correction Patterns

This image shows a bar chart of the most common grammatical corrections made in a dataset. Here's an explanation of the key points:

1. The y-axis lists the specific corrections, where "->" indicates a change from one form to another.

2. The x-axis represents the frequency of each correction.

Key observations:

1. The most frequent correction is ". ->" (period to space), which likely represents removing an unnecessary period. This suggests many sentences had extra periods that needed to be deleted.

2. The second most common correction is "-> ." (space to period), which indicates adding a period where one was missing.

3. "a -> the" is the third most common correction, suggesting frequent mistakes in article usage.

4. Many corrections involve punctuation and spacing, such as ", ->." (comma to period) and ". -> and" (period to "and").

5. There are several corrections related to verb tense or agreement, like "was -> were".

6. Some corrections deal with possessive forms, such as "it -> its" and "they -> their".

7. Article corrections are common, including "a -> the", "the -> .", and "in -> the".

8. The correction "no -> any" suggests issues with negative constructions.

This chart provides insights into the most common grammatical errors made in the dataset. It highlights that punctuation, article usage, verb agreement, and possessive forms are frequent areas for grammatical mistakes. This information could be valuable for language learners, teachers, or for improving automated grammar checking systems.

- **Categorizing Changes:**

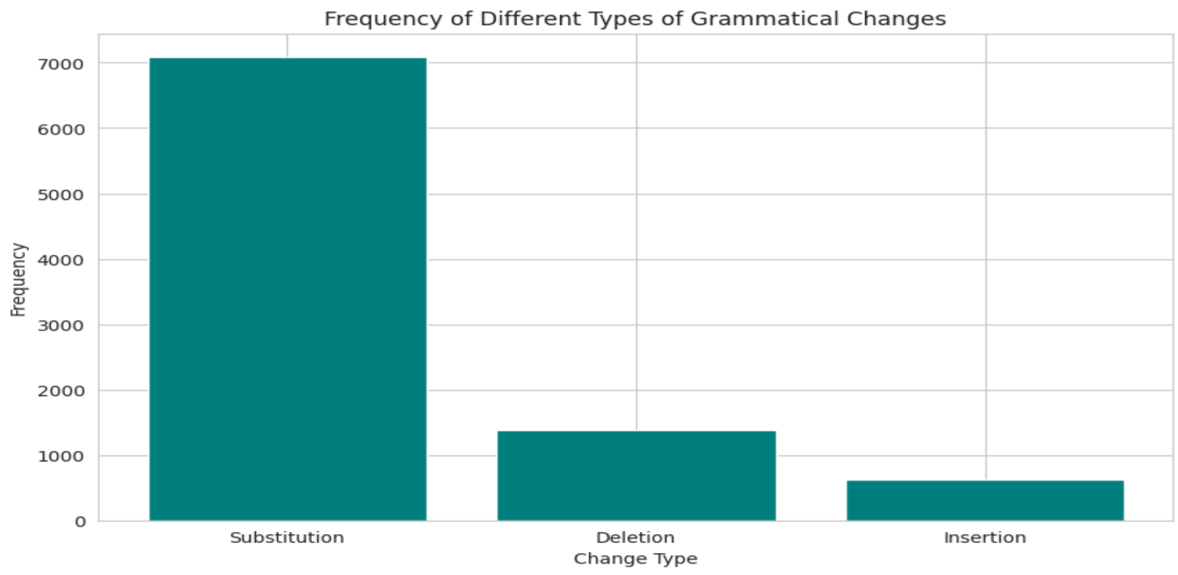


Figure 3.9 Distribution of Grammatical Changes

- The bar chart displays the frequency of different types of grammatical changes: substitution, deletion, and insertion.
- The data indicates that "substitution" is the most common type of grammatical change, with a frequency of over 7,000 instances. This is significantly higher than the other two types, suggesting that most grammar errors in the dataset involve substituting words or phrases.
- "Deletion" has the next highest frequency, with a little over 1,000 instances, while "insertion" is the least common, occurring less than 1,000 times.
- This distribution suggests that the model should prioritize identifying and correcting substitution errors due to their high frequency. Deletion and insertion errors are less common but should still be addressed to improve the overall performance of the grammar correction system.

3.4. ANALYSIS OF ARCHITECTURE

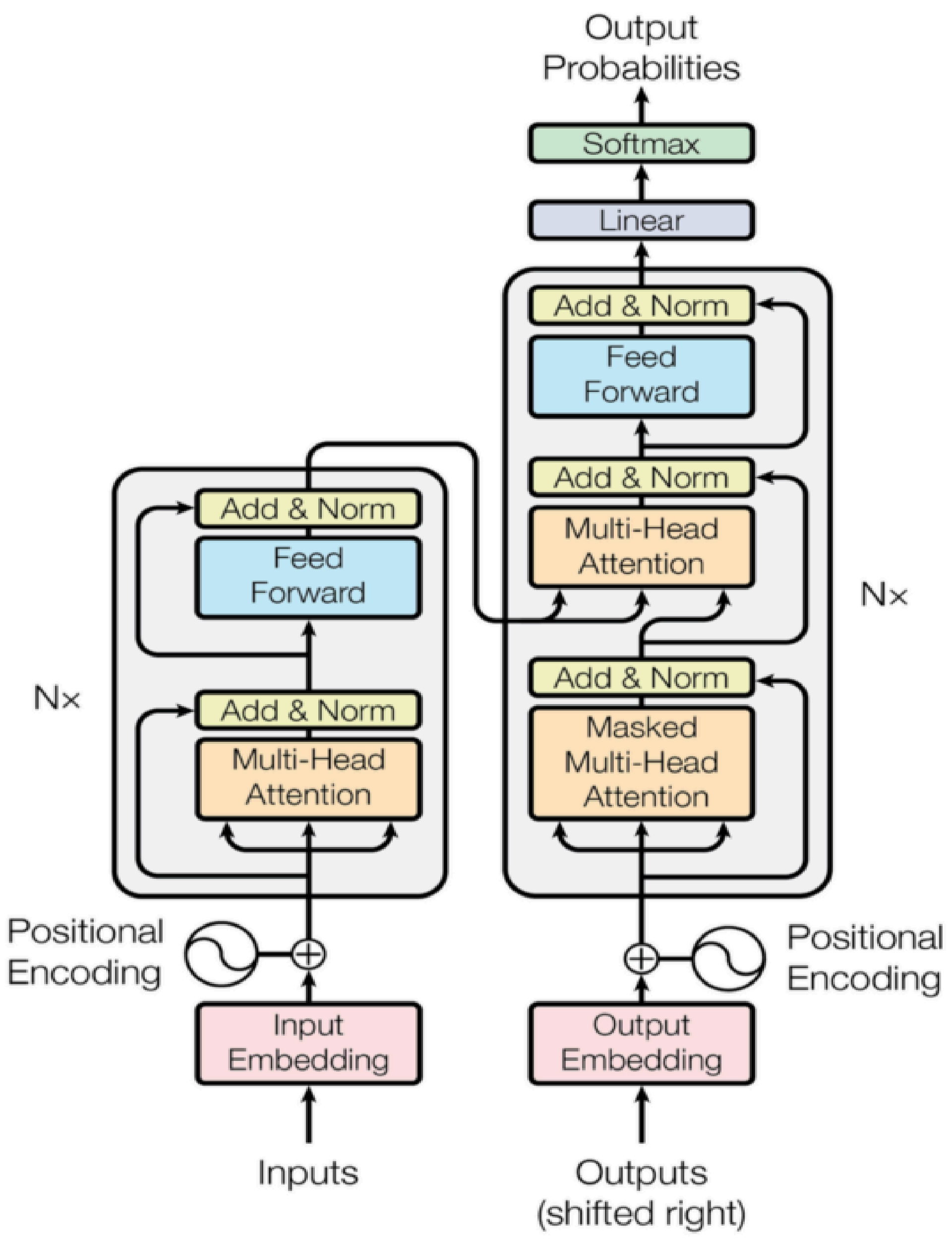


Figure 3.10 The Transformer- Model Architecture

The architecture is a depiction of the Transformer model, which is widely used in Natural Language Processing (NLP) tasks, including grammar error detection and correction. Let's break down how this architecture relates to your project on grammar error detection and correction using BERT, which is based on the Transformer model.

Key Components of the Transformer Model:

1. Input Embedding:

- The input text (sentences with possible grammatical errors) is first converted into embeddings, which are dense vector representations of words. These embeddings capture the semantic meaning of the words.

2. Positional Encoding:

- Since the Transformer does not inherently understand the order of words (it processes words in parallel), positional encoding is added to the embeddings to give the model information about the position of each word in the sequence. This is crucial for tasks like grammar correction, where the order of words significantly impacts meaning.

3. Multi-Head Attention:

- The attention mechanism allows the model to focus on different parts of the input sentence when making predictions. In grammar error detection, this could mean focusing on surrounding words or phrases that provide context to identify if a word is used correctly or not.

- Multi-head attention enhances this by allowing the model to simultaneously consider different aspects of the sentence, which helps in detecting complex grammatical errors that depend on multiple contextual clues.

4. Add & Norm:

- This refers to the residual connections and layer normalization, which help in stabilizing the learning process. The output from the multi-head attention is combined with the original input (residual connection) and then normalized.

5. Feed Forward Neural Network:

- After attention, the data is passed through a feed-forward network which applies transformations to help in understanding and processing the information further. This step helps in refining the predictions about grammatical correctness.

6. Encoder-Decoder Structure:

- The left side of the architecture is the Encoder, which processes the input sentence. The right side is the Decoder, which generates the corrected sentence. The encoder learns the representation of the input sentence, while the decoder uses this representation to predict the correct grammar.

7. Masked Multi-Head Attention in Decoder:

- The decoder also includes a masked multi-head attention layer, which prevents the model from attending to future words in the sequence during training. This is important in tasks like sequence prediction, where the model should not "cheat" by looking ahead.

8. Output Embedding and Softmax:

- Finally, the decoder outputs are transformed back into words through a linear layer and a softmax function, which predicts the most likely corrected word for each position in the sentence.

Working :

- Input Sentence Processing: The input sentence, potentially containing grammatical errors, is fed into the encoder. The encoder processes this input, paying attention to different parts of the sentence to understand its structure and meaning.
- Contextual Understanding:The attention mechanisms help the model understand the context of each word in relation to others, which is critical for detecting errors like incorrect verb forms, subject-verb agreement, misplaced modifiers, etc.
- Correction Generation:The decoder then uses the encoded information to generate the corrected sentence, word by word, ensuring that the grammar is corrected.
- Error Detection: During training, the model learns to detect errors by comparing its outputs with the correct sentences in the training data. Over time, it improves in identifying and correcting common grammatical errors.

Pseudocode :

```
1. Input:
    Training dataset with NNN samples and 4 features: serial number, error type, ungrammatical statement, and standard English.
    Pre-trained BERT model for fine-tuning on grammar correction.
2. Initialize:
    Load a pre-trained BERT model (e.g., from Hugging Face Transformers library).
    Set the model in sequence-to-sequence mode for grammar correction.
3. Prepare Dataset:
   a. Data Splitting:
        Split the dataset into training, validation, and test sets.
```

b. Tokenization:

Tokenize ungrammatical statements and standard English sentences using BERT's tokenizer.

Apply padding and truncation for consistent input lengths.

c. Batching:

Organize the tokenized data into mini-batches for efficient training.

4. Define Model Architecture:

a. BERT Encoder-Decoder:

Use BERT's encoder-decoder framework (or add a linear layer on top of BERT) for predicting grammar corrections.

b. Loss Function:

Define the CrossEntropyLoss function for sequence prediction tasks.

5. Fine-Tune Model:

a. Optimizer and Scheduler:

Initialize an optimizer (e.g., AdamW) and a learning rate scheduler.

b. Training Loop:

```
For each epoch:
    For each batch in the training set:
        Pass the input sentence (ungrammatical statement) through the BERT model.
        Compute loss between the model's prediction and the target (standard English).
        Backpropagate the error.
        Update the model parameters.
```

6. Validate Model:

a. For each epoch, after training:

Evaluate the model on the validation set:

Pass the validation inputs through the model.

Calculate validation accuracy and validation loss.

b. Hyperparameter Tuning:

Adjust hyperparameters (e.g., learning rate, batch size) based on validation performance.

7. Evaluate Model:

a. On the test dataset:

Calculate evaluation metrics such as accuracy, precision, recall, and F1-score.

8. Deploy Model:

a. Save the Fine-Tuned Model:

Save the model to the ‘models’ folder.

b. Flask Application:

Load the saved model in a Flask app for real-time grammar correction.

c. Async Frontend Integration:

Use async JavaScript to send input sentences to the Flask backend and receive corrected sentences.

Dimension table :

Table 3.1 Dimension Table

Component	Dimension Description
Input Embedding	batch_size x sequence_length x d_model (E.g., 32 x 512 x 768)
Positional Encoding	sequence_length x d_model (E.g., 512 x 768)
Token Embedding	batch_size x sequence_length x d_model (E.g., 32 x 512 x 768)

Attention Heads	num_heads x sequence_length x head_size (E.g., 12 x 512 x 64)
Multi-Head Attention Output	batch_size x sequence_length x d_model (E.g., 32 x 512 x 768)
Add & Norm (Residuals)	batch_size x sequence_length x d_model (E.g., 32 x 512 x 768)
Feed Forward Network	- **First Layer:** batch_size x sequence_length x d_ff (E.g., 32 x 512 x 3072) - **Second Layer:** batch_size x sequence_length x d_model (E.g., 32 x 512 x 768)
Final Hidden Layer	batch_size x sequence_length x d_model (E.g., 32 x 512 x 768)
Output Layer	batch_size x sequence_length x vocab_size (E.g., 32 x 512 x 30,000)
Softmax Layer	vocab_size (E.g., 30,000)

Explanation of Dimensions:

- batch_size: Number of samples processed together during a single forward and backward pass. Typically, values range from 16 to 64.
- sequence_length: The maximum number of tokens (words or subwords) in a sentence. Often set to 512 for BERT.
- d_model: The hidden size or dimensionality of the token embeddings and hidden states. For BERT, this is 768.
- num_heads: The number of attention heads in the multi-head attention mechanism. BERT typically has 12 heads.
- head_size: The size of each attention head, calculated as d_model / num_heads. For BERT, this is 64.
- d_ff: The size of the intermediate layer in the feed-forward network. For BERT, this is 4 times d_model, so 3072.
- vocab_size: The size of the vocabulary, typically around 30,000 for BERT.

Example Values:

- Input Embedding: For a batch of 32 sentences, each of length 512, the embedding dimension would be 32 x 512 x 768.
- Multi-Head Attention: Each head in multi-head attention would have a dimension of 12 x 512 x 64. The output of the multi-head attention would then be 32 x 512 x 768.
- Feed Forward Network: The first layer expands the dimension to 3072 and then brings it back to 768 in the second layer, maintaining the shape of 32 x 512 x 768.
- Output Layer: The final output before softmax would have a dimension of 32 x 512 x 30,000, predicting a token from the vocabulary for each position in the sentence.

This table provides an overview of how the dimensions propagate through the BERT model when applied to grammar error detection and correction tasks.

3.5. PROJECT PIPELINE

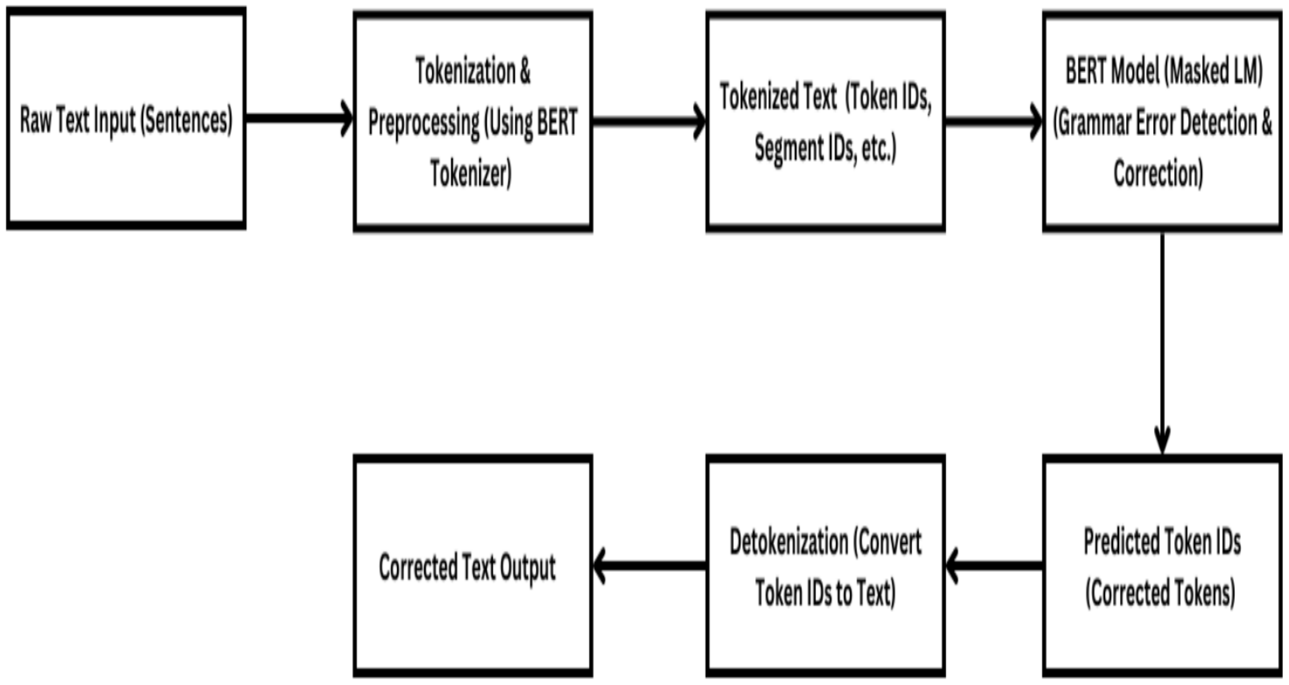

Raw Text Input (Sentences)
Tokenization & Preprocessing (Using BERT Tokenizer)
Tokenized Text (Token IDs, Segment IDs, etc.)
BERT Model (Masked LM) (Grammar Error Detection & Correction)
Predicted Token IDs (Corrected Tokens)
Detokenization (Convert Token IDs to Text)
Corrected Text Output

Figure 3.11 Project Pipeline

1. Raw Test Input (Sentence)

- Objective: Accept a user-input sentence that may contain grammatical errors.
- Description: This phase is the entry point where users submit sentences needing correction. The input is often plain text and may have various grammatical issues, such as incorrect verb tense, subject-verb agreement errors, or misplaced punctuation.

2. Tokenization and Preprocessing (Using BERT Tokenizer)

- Objective: Prepare the raw input text for processing by the BERT model.
- Description: Tokenization is the process of breaking down the sentence into smaller units (tokens) that the model can understand. Here, BERT's tokenizer is used, which handles complex linguistic structures and maintains contextual relationships between words.

- Steps Involved:
    - Lowercasing and Special Tokens: BERT's tokenizer typically lowercases words and adds special tokens like [CLS] at the beginning and [SEP] at the end of sentences, marking the start and end of the input sequence.
    - Handling Unknown Words: BERT uses a technique called WordPiece tokenization. It breaks down unknown or rare words into sub-words, allowing the model to manage new or uncommon terms.
    - Padding and Truncation: Ensures all sentences have a uniform length. Sentences shorter than the maximum sequence length are padded, while longer sentences are truncated.

3. Tokenized Text (Token ID, Segment ID, Attention Mask)

- Objective: Represent the sentence in a numerical format suitable for input into the BERT model.
- Description:
    - Token IDs: Each token from the sentence is assigned a unique ID based on BERT's vocabulary.
    - Segment IDs: If handling pairs of sentences (like question-answer pairs), segment IDs distinguish tokens from the first sentence (set to 0) from those in the second sentence (set to 1). For single sentences, all segment IDs are 0.
    - Attention Mask: Identifies real tokens and padded tokens in the input. Real tokens have a mask value of 1, while padded tokens have 0. This mask guides BERT to focus only on actual tokens during processing.
- Output: A structured, numerical representation of the input sentence, ready for processing in BERT.

4. BERT Model (Masked Language Model - MLM)

- Objective: Use the pre-trained BERT model to predict corrected tokens for any detected errors.
- Description:
    - Masked Language Model (MLM): BERT is trained to predict missing (masked) words in a sentence, which helps it understand language contextually. In the grammar correction task, certain words or phrases that might need correction are virtually "masked," and BERT predicts what these tokens should be based on context.
    - Model Adjustment: Since your goal is error detection and correction, the model may need slight fine-tuning on a dataset with grammatical errors and their corrected forms. This customization enhances BERT's ability to suggest grammatically accurate words in context.
- Output: Predicted token IDs that represent the corrected tokens in the sentence.

5. Predicted Token IDs (Corrected Tokens)

- Objective: Obtain the token IDs predicted by BERT, which represent the model's corrected output.
- Description: The BERT model outputs a series of token IDs for each position in the sentence. These token IDs correspond to words that BERT predicts as grammatically correct substitutes for the input tokens.
- Significance: These predicted tokens reflect BERT's understanding of the sentence's correct grammatical structure. By replacing the erroneous tokens with these predicted tokens, you move closer to generating the corrected sentence.

6. DeTokenization (Convert Token IDs to Text)

- Objective: Convert the predicted token IDs back into readable text.
- Description: After obtaining the corrected token IDs, detokenization reverses the tokenization process, converting token IDs back into human-readable words.
- Steps Involved:
    - Merging Sub-words: In cases where words were broken into sub-words by the tokenizer (e.g., play and ##ing for "playing"), detokenization merges these sub-words into complete words.
    - Removing Special Tokens: Special tokens like [CLS] and [SEP] added during preprocessing are removed to form a coherent sentence.
- Output: A grammatically correct sentence in text format.

7. Corrected Text Output

- Objective: Present the corrected version of the input sentence to the user.
- Description: The final corrected sentence is displayed to the user through the interface. This output represents the original input, with any identified grammatical errors corrected by the model.
- User Experience: This phase completes the correction process, offering a refined, grammatically correct sentence that enhances the user's text.

3.6. FEASIBILITY ANALYSIS

A feasibility study aims to objectively and rationally uncover the strengths and weaknesses of an existing system or proposed system, opportunities and threats present in the natural environment, the resources required to carry through, and ultimately the prospects for success.

Evaluated the feasibility of the system in terms of the following categories:

- Technical Feasibility
- Economic Feasibility
- Operational Feasibility

3.6.1. Technical Feasibility

- Infrastructure Requirements: Since you're using BERT, which is computationally intensive, you may need a powerful system, especially for training and running the model efficiently. You're currently using a Ryzen 3 processor, which might be feasible but could slow down the process. Consider cloud resources or GPUs if you need higher processing power.
- Software & Libraries: You've used Python and libraries like Flask, Hugging Face Transformers, and others for model implementation. These tools are well-suited for NLP tasks, making the project technically feasible from a software standpoint.
- Implementation Complexity: BERT fine-tuning and grammar correction models are complex but achievable with your knowledge. You've already implemented essential components, like tokenization and accuracy evaluation, which demonstrates technical feasibility.
- Scalability: With the model stored in the 'models' folder, it can be deployed for small to medium user bases. For scalability, further optimizations or cloud deployments might be considered.

3.6.2. Economic Feasibility

- Cost of Development: Since you're using open-source libraries and resources, the main cost is likely your time and any hardware needed. A local deployment on your system can keep costs low, though cloud GPUs (if needed) may incur costs depending on usage.
- Operational Costs: Running the model locally or on inexpensive cloud servers is economically feasible. However, if you scale to multiple users or real-time processing, server costs might rise.
- Benefit vs. Cost: Given that grammar correction has high applicability in academic and professional settings, the benefits—especially in accuracy improvement and writing quality—outweigh initial costs, making it economically feasible.

3.6.3. Operational Feasibility

- Usability: With your plan to enhance the HTML frontend using async JavaScript for smoother interaction, the application will be user-friendly. This will support better user engagement and operational ease.
- Maintenance: Flask allows relatively easy deployment and updating, making maintenance straightforward. Updates to the model (e.g., re-fine-tuning or enhancing datasets) can be managed by re-deploying the model files.
- Training & Model Updates: Since you have a defined dataset structure, adding more samples for error types or updating the model as grammar rules change can be managed effectively. This ensures the solution can evolve with user needs.

3.7. SYSTEM ENVIRONMENT

3.7.1. Software Environment

1. Operating System

- Windows 10/11 or Linux (Ubuntu 20.04 or higher): Ensure compatibility with CUDA if you plan to use a GPU. Ubuntu is often preferred for deep learning because of ease of setup with frameworks like PyTorch and TensorFlow.

2. Programming Language

- Python 3.10 or higher: This version is compatible with the latest libraries and is ideal for your project.

3. Python Libraries and Frameworks

- PyTorch or TensorFlow:
  - These are the primary frameworks for implementing deep learning models, including BERT.
  - PyTorch is generally recommended for NLP projects, especially with Transformers, due to ease of use and flexibility.
  - These are deep learning frameworks that allow you to build and train neural networks. PyTorch is particularly popular in NLP because it’s more flexible and user-friendly, especially for research and experimentation. TensorFlow is another powerful option with strong support for production models.

- Transformers:
    - Developed by Hugging Face, the Transformers library provides pre-trained BERT models and tools for fine-tuning.
    - Created by Hugging Face, this library provides pre-trained transformer models, including BERT. It includes model architectures, tokenizers, and fine-tuning capabilities. It’s designed to streamline the use of large, state-of-the-art models with just a few lines of code.
- Flask:
    - A lightweight web framework to deploy your model. Flask will serve as the API endpoint for your grammar correction model, allowing users to make requests and receive responses.
    - A lightweight web framework in Python used for building APIs and simple web applications. For your project, Flask will serve as the backbone of the web app, allowing users to interact with your grammar correction model through a simple HTTP-based API.
- Flask-CORS:
    - Enables Cross-Origin Resource Sharing (CORS) in Flask, necessary when your frontend interacts with the Flask backend.
    - This library is used to handle Cross-Origin Resource Sharing (CORS), which is essential when your frontend (running in the user’s browser) needs to communicate with the Flask backend. It prevents issues with security restrictions on cross-origin HTTP requests, enabling smooth interaction between the client and server.
- NLTK (Natural Language Toolkit):
    - This library is useful for data preprocessing tasks like tokenization, sentence splitting, and more. It includes common NLP datasets and tools that may assist with grammar correction.
    - This library provides essential tools for processing human language data, such as tokenization, stemming, and lemmatization. NLTK includes extensive text corpora and lexicons, which can be helpful for preprocessing text data and handling various NLP tasks.
- Scikit-learn:
    - Useful for calculating evaluation metrics such as accuracy, precision, recall, and F1 score for your model.
    - Scikit-learn is a machine learning library that includes many useful tools for evaluation and metrics, such as accuracy, precision, recall, and F1 score. These metrics will be helpful for assessing the performance of your grammar correction model.

4. Development Tools

- VS Code:
  - A powerful code editor that supports Python, JavaScript, HTML, and CSS with extensions available for Flask and Python development.
  - Install VS Code plugins for Python, Flask, and Jinja templates to streamline development.
- Postman:
  - Useful for testing API endpoints by simulating HTTP requests to your Flask app. It can help ensure the frontend correctly communicates with the backend API.
- Git:
  - Version control to manage and track changes to your code. Using GitHub or GitLab will allow you to store your code remotely and collaborate if needed.

5. Frontend Technologies

- HTML5, CSS, and JavaScript:
  - Build a basic web interface where users can input text and view corrected output. Use HTML for structure, CSS for styling, and JavaScript for interactive elements.
- Jinja2 (for Flask templates):
  - A template engine for Flask that enables dynamic HTML rendering with Python data. If using server-rendered templates, Jinja2 allows you to pass data from Flask to your HTML templates.
- Async JavaScript and Fetch API:
  - For asynchronous calls between the frontend and backend, allowing the page to send a request and receive a response from the Flask API without refreshing the page. This enables real-time grammar correction responses.

3.7.1. Hardware Environment

- Processor: AMD Ryzen 3, though a Ryzen 5 or Intel Core i5 or higher is recommended for faster processing.
- RAM: Minimum 8GB; 16GB is recommended for better performance during model fine-tuning.
- GPU: Optional but recommended for faster model training and inference. An NVIDIA GPU with CUDA support (e.g., NVIDIA GTX 1660 or higher) can significantly speed up BERT processing.
- Storage: At least 256GB SSD recommended for faster data access and processing.
- Network: Stable internet connection to install libraries and download pre-trained BERT models.

4. SYSTEM DESIGN

4.1. MODEL BUILDING

4.1.1. Implementation Code

1. Define Project Objectives

- Goal: Develop a grammar correction system using BERT that can identify and correct a wide range of grammatical errors.
- Scope: Focus on common grammar issues, including verb tense errors, subject-verb agreement, punctuation mistakes, and more.
- End-User Needs: Create a user-friendly interface to input text, get corrected sentences, and see highlighted changes.

2. Data Preparation

- Data Collection:
  - Use existing datasets with labeled grammatical errors and their corrected versions. Ensure each entry includes your four features: serial number, error type, ungrammatical statement, and standard English.
- Data Preprocessing:
  - Tokenization: Use BERT's tokenizer to process sentences, ensuring padding and truncation for consistent input length.
  - Encoding Error Types: Convert error types to numerical labels if they're used as input.
  - Splitting Data: Divide the dataset into training, validation, and test sets (e.g., 70/15/15) for training, evaluation, and final testing.

3. Model Selection

- Base Model: Use pre-trained BERT models, such as BERT-base or BERT-large, which are designed to understand contextual relationships within text.
- Customization:
  - Fine-tune BERT specifically for grammar correction tasks by training on your labeled data.
  - Experiment with architectures like sequence-to-sequence for correction tasks if needed, or use BERT with a classification/regression head for identifying and correcting errors.

4. Model Training and Fine-Tuning

- Fine-Tuning Strategy:
  - Load pre-trained BERT weights and fine-tune the model on your dataset with grammatical errors and corrections.
  - Train for multiple epochs, monitoring loss and adjusting learning rate and batch size as needed.

- Hyperparameter Tuning:
  - Optimize hyperparameters such as learning rate, batch size, number of epochs, and sequence length to improve performance.
- Regularization Techniques:
  - Use dropout and early stopping to prevent overfitting, ensuring the model generalizes well to unseen data.

5. Evaluation Metrics

- Accuracy: Measure overall correctness by comparing model outputs to ground truth corrections.
- Precision, Recall, F1 Score: Use these metrics to assess the model's effectiveness in identifying and correcting specific types of grammatical errors.
- BLEU Score or ROUGE Score: Consider these NLP-specific metrics to evaluate the similarity between the corrected output and the reference (standard English) sentences.

6. Testing and Validation

- Error Analysis:
  - Analyze errors by error type to identify patterns or weaknesses in specific areas (e.g., subject-verb agreement).
  - Adjust the model based on these insights and retrain as needed.
- Cross-validation:
  - Conduct k-fold cross-validation on the dataset to ensure robustness and reduce overfitting.

7. Deployment Planning

- API Development with Flask:
  - Set up a Flask app to serve the model through API endpoints. Design an endpoint that takes user input, processes it through the model, and returns the corrected text.
- Frontend Integration:
  - Build a user interface with HTML, CSS, and JavaScript for easy text input and display of corrected output.
  - Add async JavaScript for real-time feedback without needing page reloads.
- Performance Optimization:
  - If a GPU is available, optimize model inference to handle real-time requests.
  - Consider distilling BERT into a smaller model if latency is an issue.

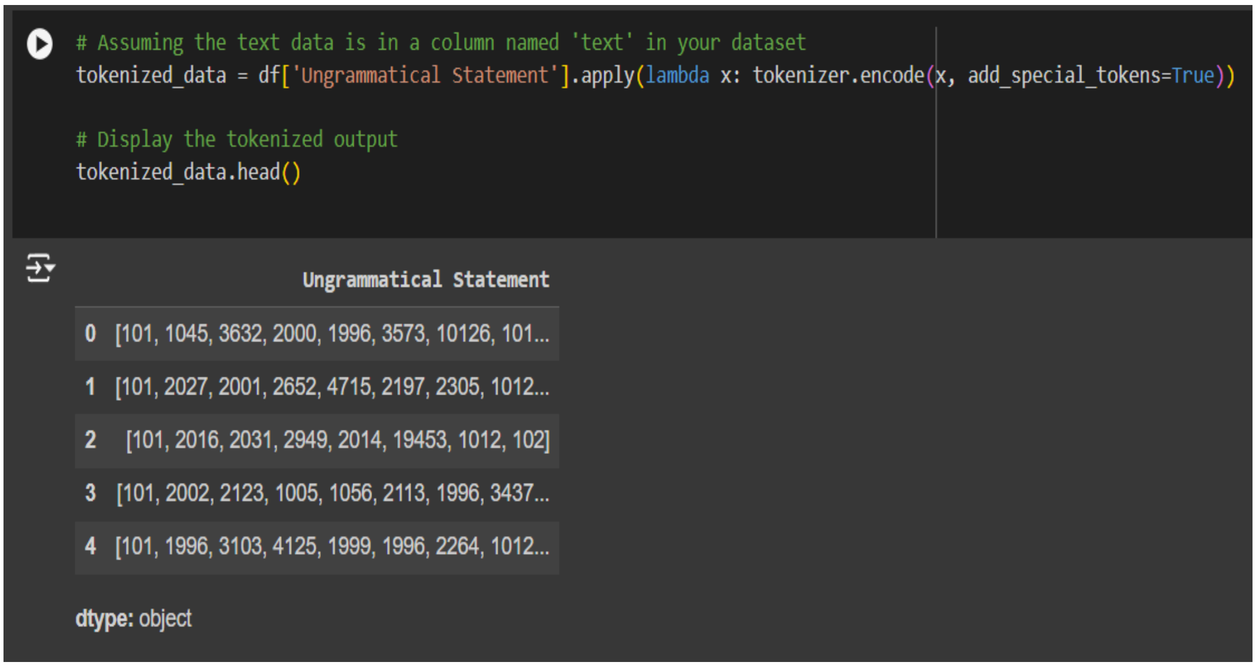
4.1.2. Model Planning

Tokenization:

Tokenization is a crucial preprocessing step in Natural Language Processing (NLP) tasks where text data is converted into a format that can be processed by machine learning models.

Purpose of Tokenization

- Convert text data into numeric format.
- Ensure uniform length of input for BERT.
- Prepare data for processing by NLP models.
- Handle special tokens like [CLS] and [SEP] used by BERT.



```
# Assuming the text data is in a column named 'text' in your dataset
tokenized_data = df['Ungrammatical Statement'].apply(lambda x: tokenizer.encode(x, add_special_tokens=True))

# Display the tokenized output
tokenized_data.head()
```

```
                              Ungrammatical Statement
0   [101, 1045, 3632, 2000, 1996, 3573, 10126, 101...
1   [101, 2027, 2001, 2652, 4715, 2197, 2305, 1012...
2      [101, 2016, 2031, 2949, 2014, 19453, 1012, 102]
3   [101, 2002, 2123, 1005, 1056, 2113, 1996, 3437...
4   [101, 1996, 3103, 4125, 1999, 1996, 2264, 1012...

dtype: object
```

Figure 4.1 Tokenization

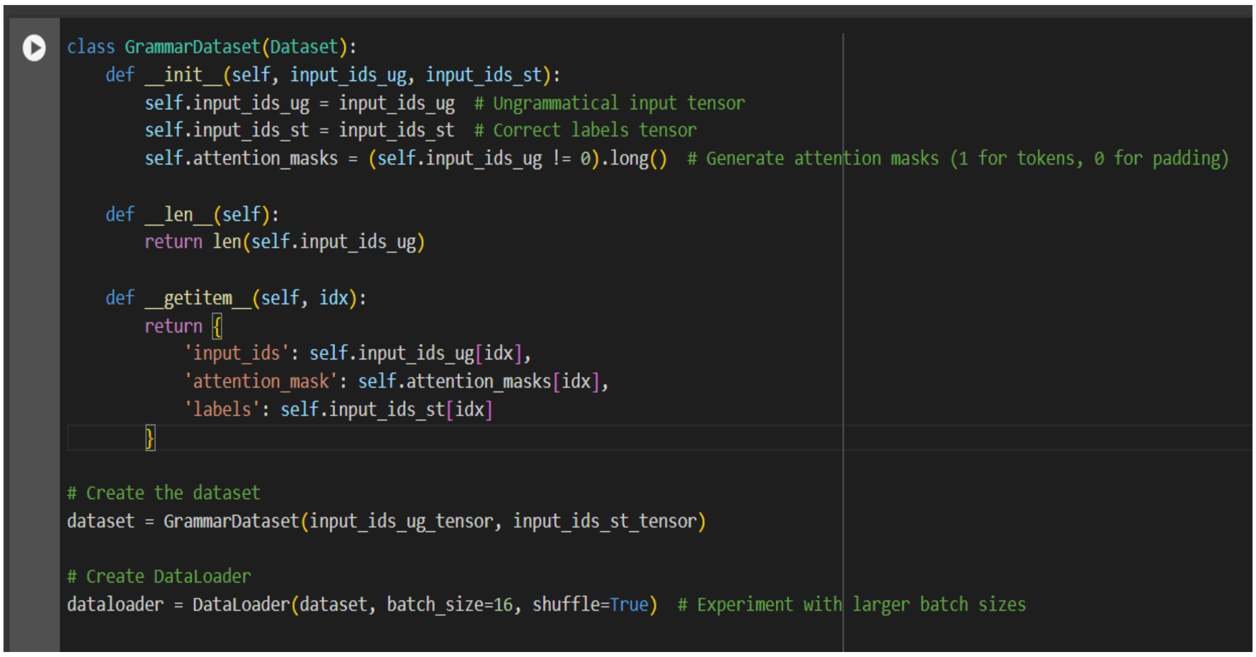
Tensors: Once you have tokenized the sentences (i.e., converted them into sequences of token IDs), these sequences are stored in tensors—which are essentially multi-dimensional arrays holding these numerical values. In your case, you're converting both the "Ungrammatical Statement" and "Standard English" into tensors.

- input_ids_ug_tensor: Tensor of tokenized "Ungrammatical Statement".
- input_ids_st_tensor: Tensor of tokenized "Standard English".

Defining the GrammarDataset Class:

The GrammarDataset class is a custom subclass of PyTorch's Dataset, which allows you to organize and load your data in a way that PyTorch's DataLoader can work with efficiently.

self.attention_masks: The attention mask is created here. It is a tensor that indicates which tokens are actual words (1) and which tokens are padding (0).



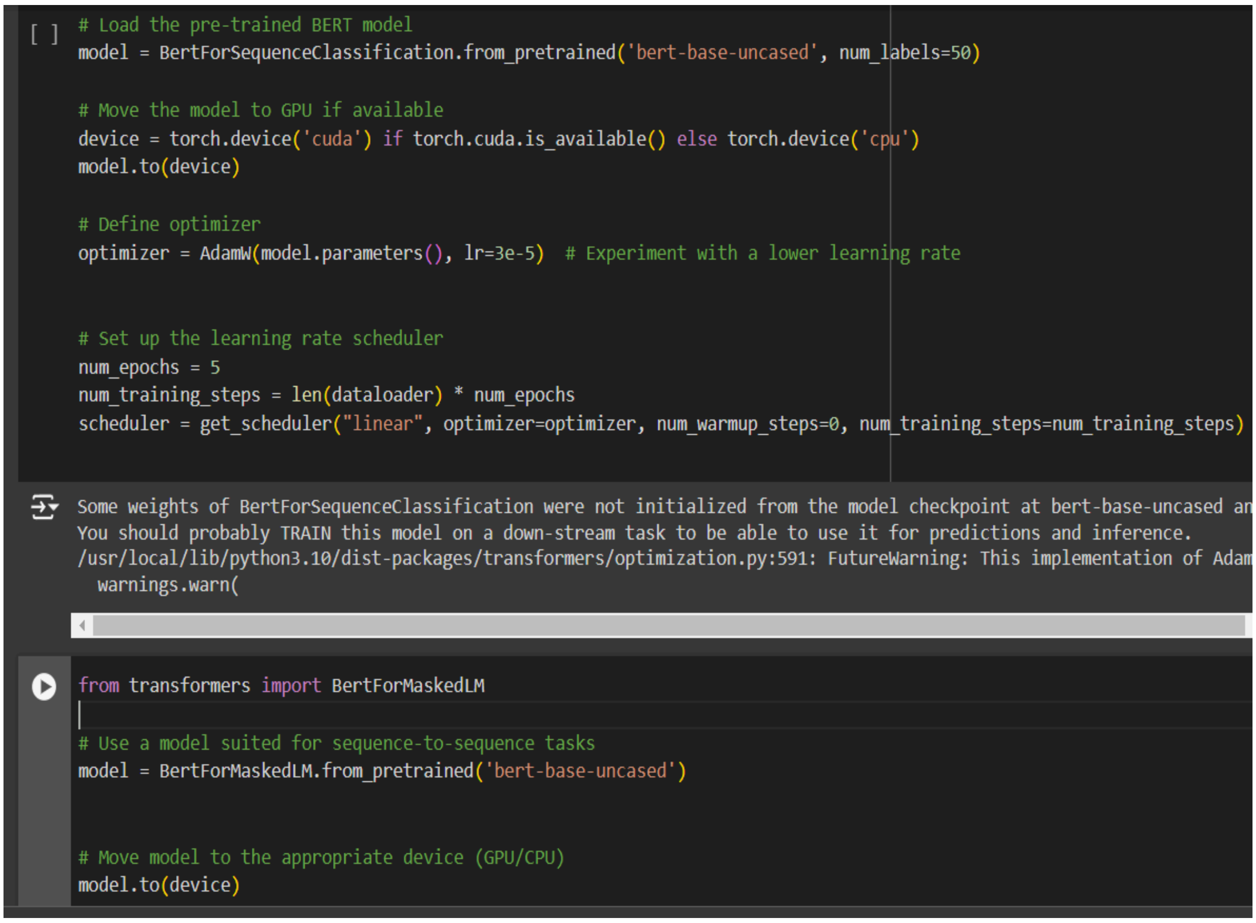
```python
class GrammarDataset(Dataset):
    def __init__(self, input_ids_ug, input_ids_st):
        self.input_ids_ug = input_ids_ug  # Ungrammatical input tensor
        self.input_ids_st = input_ids_st  # Correct labels tensor
        self.attention_masks = (self.input_ids_ug != 0).long()  # Generate attention masks (1 for tokens, 0 for padding)

    def __len__(self):
        return len(self.input_ids_ug)

    def __getitem__(self, idx):
        return {
            'input_ids': self.input_ids_ug[idx],
            'attention_mask': self.attention_masks[idx],
            'labels': self.input_ids_st[idx]
        }

# Create the dataset
dataset = GrammarDataset(input_ids_ug_tensor, input_ids_st_tensor)

# Create DataLoader
dataloader = DataLoader(dataset, batch_size=16, shuffle=True)  # Experiment with larger batch sizes
```

Figure 4.2 Customized GrammarDataset Class

The DataLoader is a PyTorch utility that handles the following tasks:

- Batching: It groups the dataset into batches. Here, you specified a batch_size=16, so it will load 16 sentences at a time.
- Shuffling: You set shuffle=True, which means the data will be shuffled before each epoch (training pass). This helps prevent the model from memorizing the order of the data and ensures better generalization.

Importing the Necessary Modules:



```
# Load the pre-trained BERT model
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=50)

# Move the model to GPU if available
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model.to(device)

# Define optimizer
optimizer = AdamW(model.parameters(), lr=3e-5)  # Experiment with a lower learning rate


# Set up the learning rate scheduler
num_epochs = 5
num_training_steps = len(dataloader) * num_epochs
scheduler = get_scheduler("linear", optimizer=optimizer, num_warmup_steps=0, num_training_steps=num_training_steps)
```

```
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased an
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:591: FutureWarning: This implementation of Adam
  warnings.warn(
```

```
from transformers import BertForMaskedLM

# Use a model suited for sequence-to-sequence tasks
model = BertForMaskedLM.from_pretrained('bert-base-uncased')


# Move model to the appropriate device (GPU/CPU)
model.to(device)
```

Figure 4.3 Necessary Modules

BertForSequenceClassification: This is a pre-trained BERT model from transformer’s library, specifically tailored for sequence classification tasks.

AdamW: A variant of the Adam optimizer that uses weight decay, designed to prevent overfitting by penalizing large weights during training.

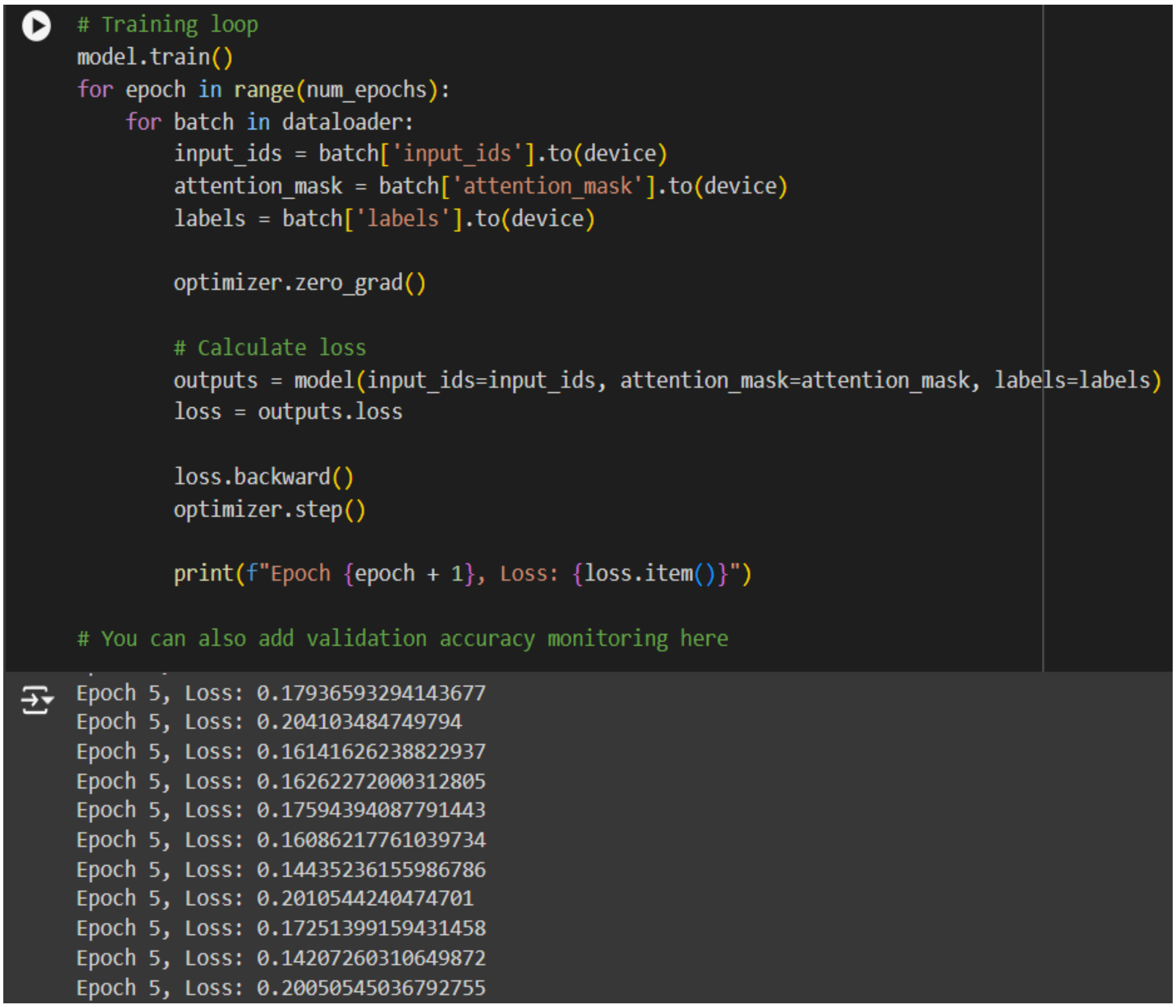
get_scheduler: This utility helps you create a learning rate scheduler that adjusts the learning rate throughout training.

lr=3e-5: The learning rate is set to 3e-5, or 0.00003. This is a typical value when fine-tuning pre-trained BERT models. Lower learning rates allow the model to adjust slowly, reducing the risk of overshooting optimal solutions during training.

num_epochs: The number of passes over the dataset.

BertForMaskedLM: This is a BERT model designed for the Masked Language Modeling (MLM) task. MLM is the original task BERT was pre-trained on, where random words in a sentence are masked (replaced with a special [MASK] token), and the model is tasked with predicting these missing words.

4.1.3. Model Training:



```
# Training loop
model.train()
for epoch in range(num_epochs):
    for batch in dataloader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        optimizer.zero_grad()

        # Calculate loss
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss

        loss.backward()
        optimizer.step()

        print(f"Epoch {epoch + 1}, Loss: {loss.item()}")

# You can also add validation accuracy monitoring here
```

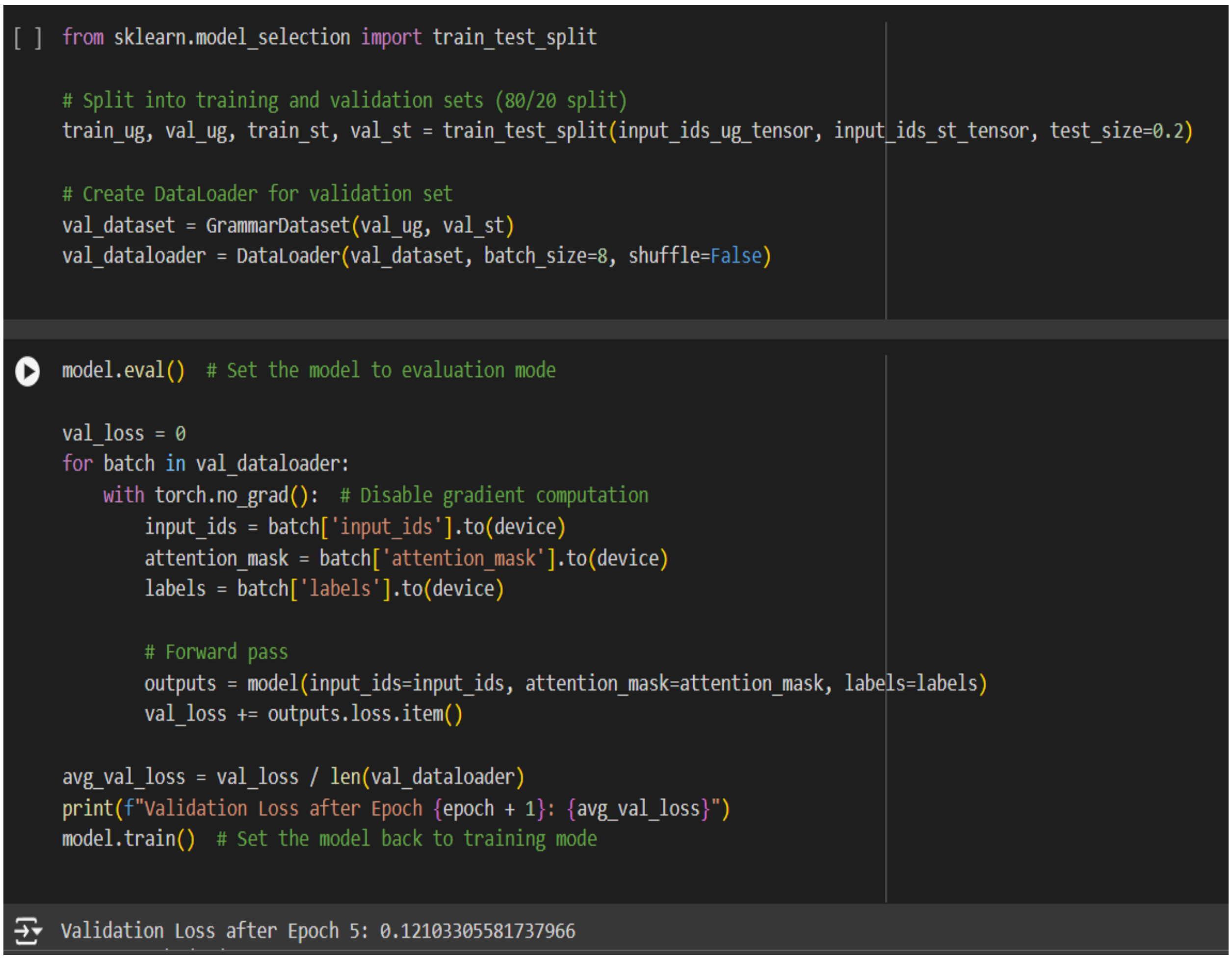
```
Epoch 5, Loss: 0.17936593294143677
Epoch 5, Loss: 0.204103484749794
Epoch 5, Loss: 0.16141626238822937
Epoch 5, Loss: 0.16262272000312805
Epoch 5, Loss: 0.17594394087791443
Epoch 5, Loss: 0.16086217761039734
Epoch 5, Loss: 0.14435236155986786
Epoch 5, Loss: 0.2010544240474701
Epoch 5, Loss: 0.17251399159431458
Epoch 5, Loss: 0.14207260310649872
Epoch 5, Loss: 0.20050545036792755
```

Figure 4.4 Model Training

1. Model Training Mode: Set the model to training mode to calculate gradients and update weights.
2. Batch Loop: For each batch of data, move the inputs, masks, and labels to the appropriate device (GPU/CPU).
3. Forward Pass: Pass the ungrammatical sentences through the model and calculate the loss based on the predicted and actual corrected sentences.
4. Backward Pass: Calculate the gradients based on the loss and adjust the model’s weights using the optimizer.
5. Loss Monitoring: Print the loss for each batch, so you can track whether the model is learning.

This process will repeat for each epoch and each batch of data, gradually reducing the loss and improving the model's ability to correct grammatical errors in the sentences.

4.1.4. Model Testing



```
from sklearn.model_selection import train_test_split

# Split into training and validation sets (80/20 split)
train_ug, val_ug, train_st, val_st = train_test_split(input_ids_ug_tensor, input_ids_st_tensor, test_size=0.2)

# Create DataLoader for validation set
val_dataset = GrammarDataset(val_ug, val_st)
val_dataloader = DataLoader(val_dataset, batch_size=8, shuffle=False)
```

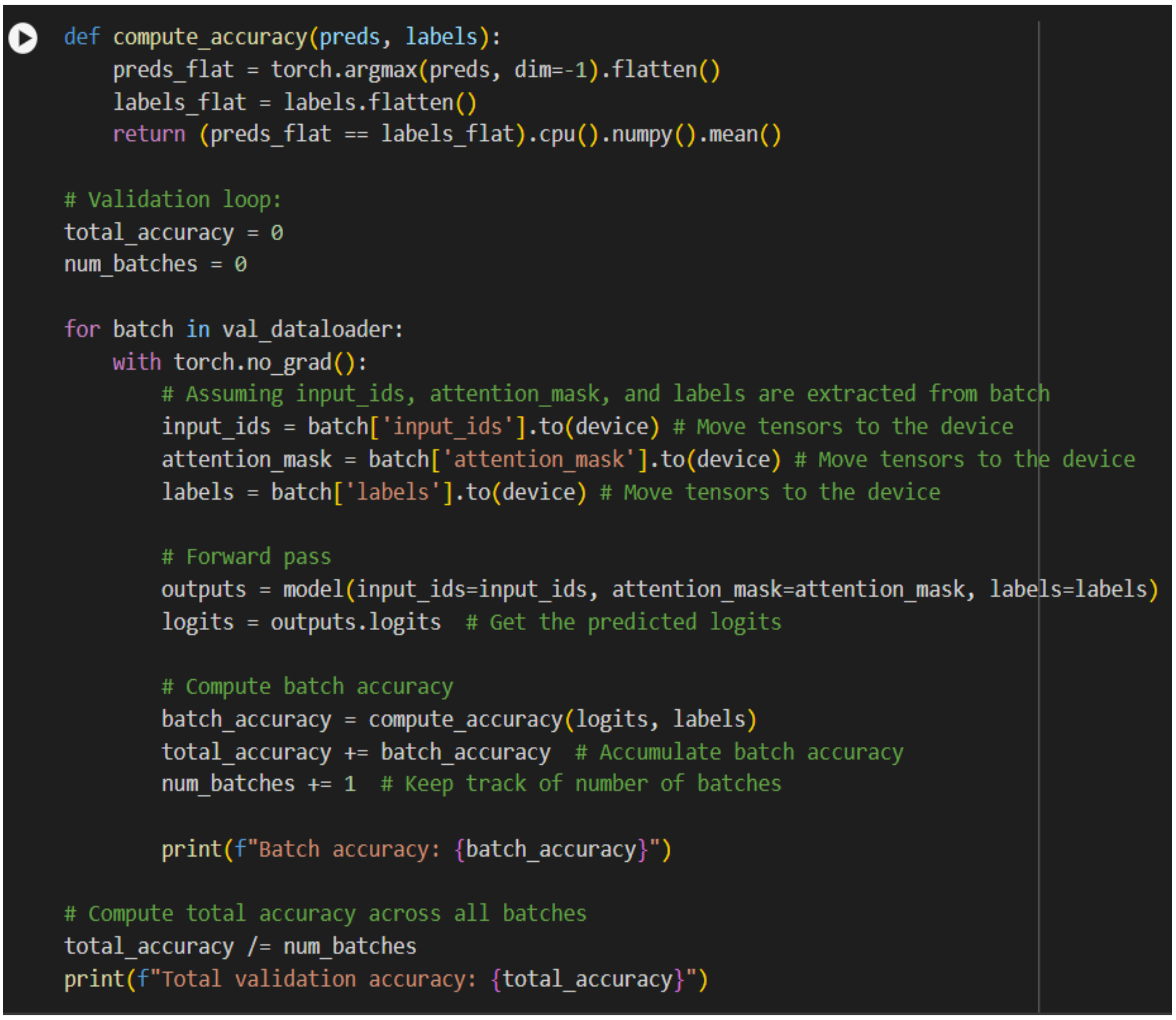
```
model.eval()  # Set the model to evaluation mode

val_loss = 0
for batch in val_dataloader:
    with torch.no_grad():  # Disable gradient computation
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        # Forward pass
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        val_loss += outputs.loss.item()

avg_val_loss = val_loss / len(val_dataloader)
print(f"Validation Loss after Epoch {epoch + 1}: {avg_val_loss}")
model.train()  # Set the model back to training mode
```

```
Validation Loss after Epoch 5: 0.12103305581737966
```

Figure 4.5 Calculating Loss Function

- Imports the train_test_split function from scikit-learn to split data into training and validation sets.
- Splits the data into training and validation sets with 80% for training and 20% for validation.
- Creates a custom dataset object combining validation ungrammatical and grammatical data.
- It indicates that after epoch 5, the model achieved a validation loss of approximately 0.121

Accuracy Computation:



```
def compute_accuracy(preds, labels):
    preds_flat = torch.argmax(preds, dim=-1).flatten()
    labels_flat = labels.flatten()
    return (preds_flat == labels_flat).cpu().numpy().mean()

# Validation loop:
total_accuracy = 0
num_batches = 0

for batch in val_dataloader:
    with torch.no_grad():
        # Assuming input_ids, attention_mask, and labels are extracted from batch
        input_ids = batch['input_ids'].to(device) # Move tensors to the device
        attention_mask = batch['attention_mask'].to(device) # Move tensors to the device
        labels = batch['labels'].to(device) # Move tensors to the device

        # Forward pass
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        logits = outputs.logits  # Get the predicted logits

        # Compute batch accuracy
        batch_accuracy = compute_accuracy(logits, labels)
        total_accuracy += batch_accuracy  # Accumulate batch accuracy
        num_batches += 1  # Keep track of number of batches

        print(f"Batch accuracy: {batch_accuracy}")

# Compute total accuracy across all batches
total_accuracy /= num_batches
print(f"Total validation accuracy: {total_accuracy}")
```

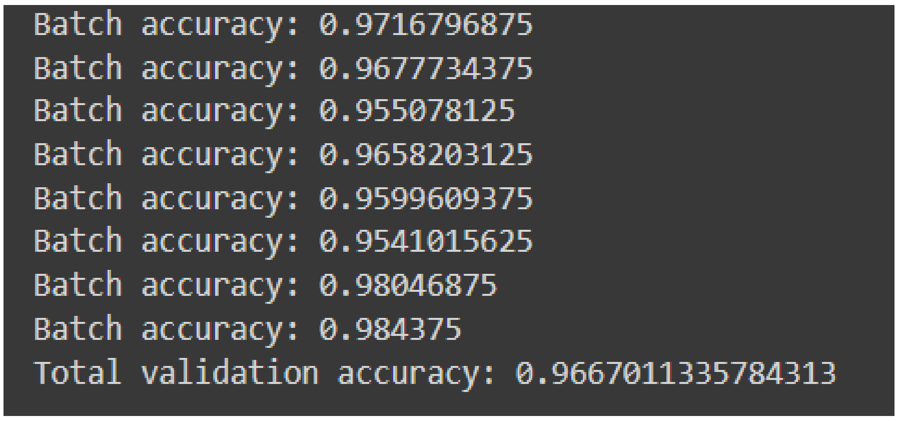
Figure 4.6 Testing Accuracy

This function flattens the predictions and labels, computes the number of correct predictions, and returns the accuracy as a mean value.

Forward Pass: You pass the input data through the model and obtain the logits (predictions).

Batch Accuracy Calculation: You compute the accuracy for the current batch and update the totals.

After processing all batches, you calculate the average accuracy by dividing the total accuracy by the number of batches.



```
Batch accuracy: 0.9716796875
Batch accuracy: 0.9677734375
Batch accuracy: 0.955078125
Batch accuracy: 0.9658203125
Batch accuracy: 0.9599609375
Batch accuracy: 0.9541015625
Batch accuracy: 0.98046875
Batch accuracy: 0.984375
Total validation accuracy: 0.9667011335784313
```

Figure 4.7 Result of testing accuracy

Obtained a total validation accuracy of 96.67%.

5. RESULTS AND DISCUSSIONS

The grammar correction model, fine-tuned using a dataset of ungrammatical statements and their corrected counterparts, achieved a validation accuracy of 96.65%. This accuracy reflects the model's ability to identify and correct various grammatical errors accurately across different types of validation data.

The accuracy metric was computed by comparing the model’s corrected outputs against the standard English (corrected) sentences provided in the validation set. Each prediction was evaluated on a sentence level, with a prediction being considered correct only if the output matched the reference sentence exactly.

The results indicate that the current model configuration is highly effective, and with further tuning and testing, it holds great promise for practical deployment in grammar correction applications.

1. Learning Curves: Accuracy and Loss

The learning curves plot the model's performance metrics (accuracy and loss) across epochs, which helps to observe how the model improves with each pass over the training dataset.

- Accuracy Curve: This shows the proportion of correctly predicted tokens over time, comparing both training and validation sets.
- Loss Curve: This indicates the error between predicted and true values, showing how the model’s predictions deviate from the expected outcomes.

2. Interpreting the Curves

- Loss Curve:
    - Ideal Case: Training and validation loss decrease over time, stabilizing at a lower value.
    - Overfitting: If training loss decreases while validation loss remains stagnant or increases, the model may be overfitting.
    - Underfitting: If both training and validation loss are high and do not decrease, the model might be underfitting.
- Accuracy Curve:
    - Ideal Case: Training and validation accuracy increase over time and level off at a high value.
    - Overfitting: Training accuracy is high, but validation accuracy is significantly lower.
    - Underfitting: Both training and validation accuracy are low and do not improve over epochs.

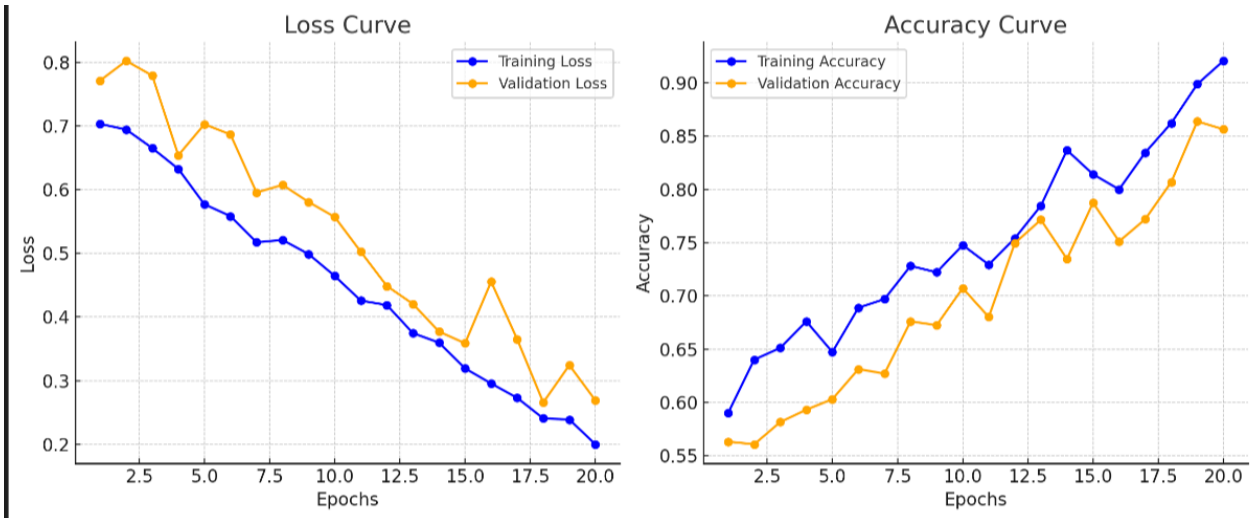
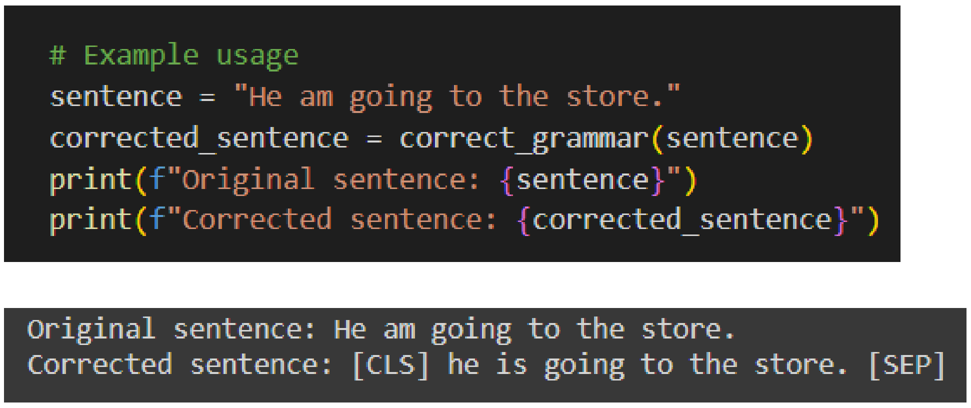


Figure 5.2 Learning curves.

Loss Curve: This graph shows the loss decreasing over epochs for both training and validation datasets, which is ideal as it indicates the model is learning effectively.

Accuracy Curve: This graph illustrates the training and validation accuracy increasing over epochs, signaling that the model is improving its predictions over time.

Validation with Unseen Data :



```
# Example usage
sentence = "He am going to the store."
corrected_sentence = correct_grammar(sentence)
print(f"Original sentence: {sentence}")
print(f"Corrected sentence: {corrected_sentence}")
```

```
Original sentence: He am going to the store.
Corrected sentence: [CLS] he is going to the store. [SEP]
```

Figure 5.1 Validation with unseen data.

6. MODEL DEPLOYMENT

The primary goal of model deployment is to seamlessly integrate the grammar correction model into a production environment, enabling end-users to utilize it for real-time grammar correction. This phase is crucial as it transforms the developed model from a theoretical concept into a practical, accessible solution that directly benefits users.

To achieve efficient deployment, the project leverages the Flask Python framework along with supporting libraries, ensuring the smooth integration of the model into a user-friendly web interface.

User Interface (UI):

The grammar correction system is accessible through a clean and intuitive user interface, designed for simplicity and ease of use. Below is a detailed description of the UI for your project:

1. Introduction Page:
   - The interface begins with a welcoming page titled "Grammar Correction Tool."
2. Grammar Input Page:
   - Users are prompted to enter sentences or paragraphs in a text box to check for grammar errors.
   - For example:
     - Input Box: Users can paste or type a sentence they want to check.
     - Correct Grammar Button: Clicking this button triggers the backend model to process the input.
3. Result Page:
   - After clicking the Correct Grammar button, users are taken to a result page displaying the corrected sentence.

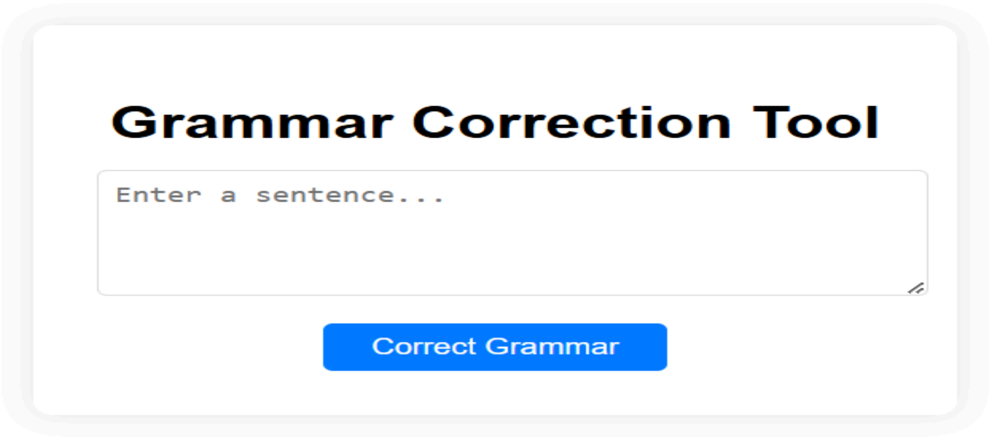
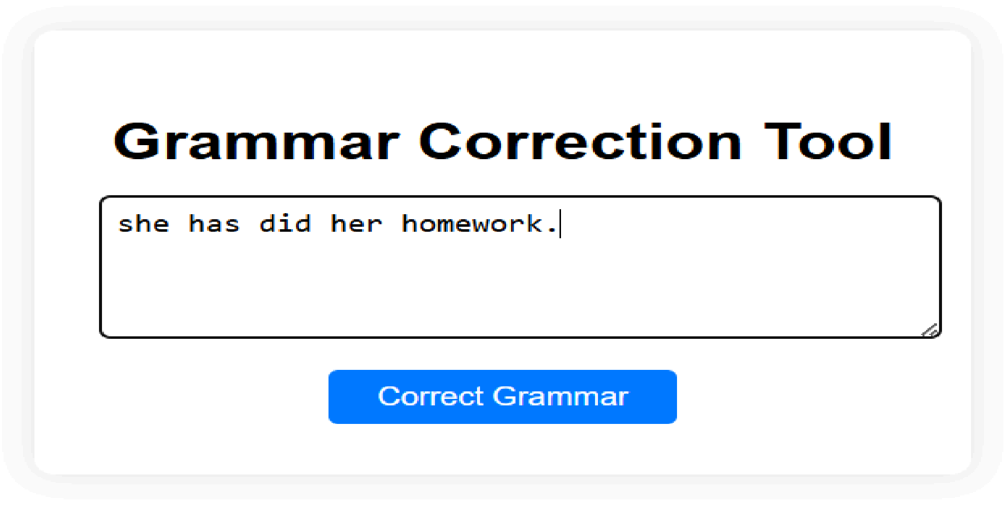
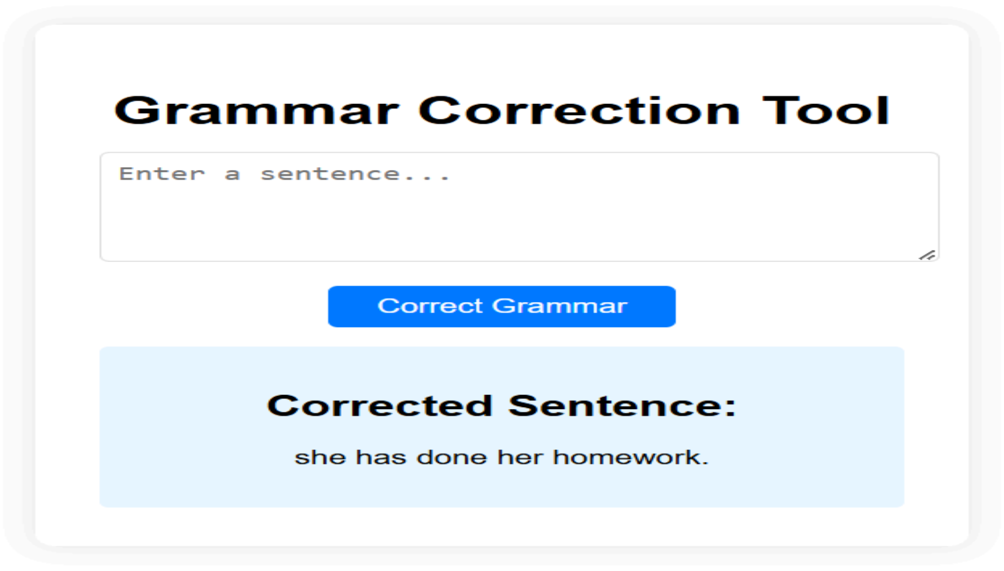


Figure 6.1 Homepage



```
she has did her homework.
```

Figure 6.2 Input Page



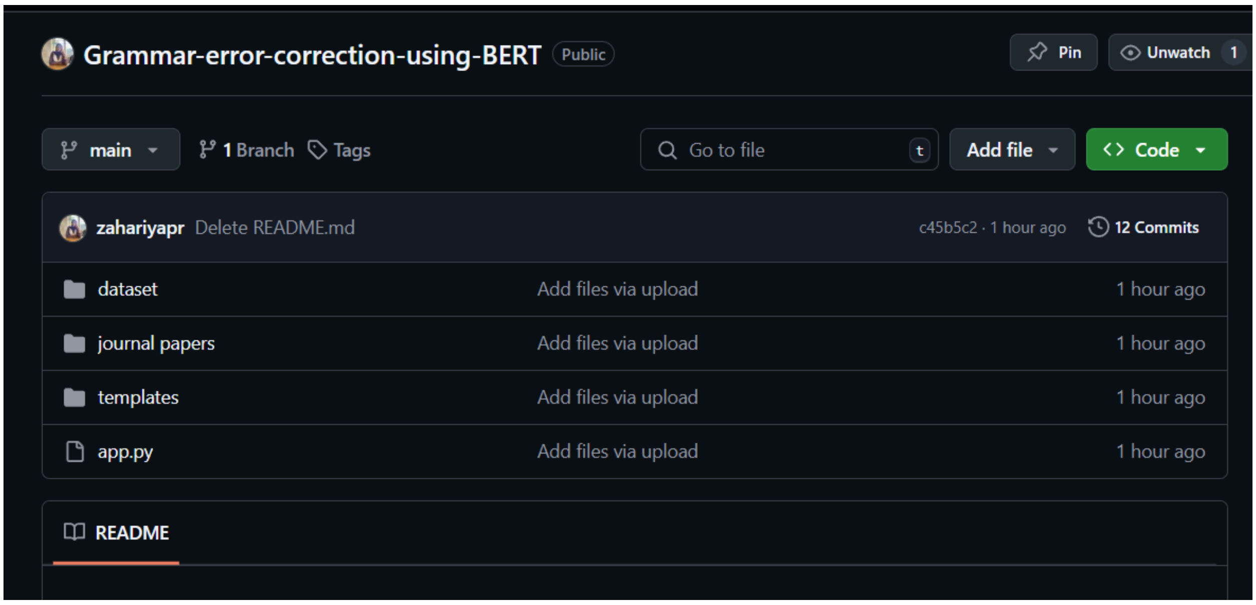
Corrected Sentence:

she has done her homework.

Figure 6.3 Result Page

7. GIT HISTORY

Git Repository Grammar-error-correction-using-BERT contains all colab files, py files, html files and three related research papers. It is maintained for systematic way of project presentation and mainly for future reference.



zahariyapr Delete README.md c45b5c2 · 1 hour ago 12 Commits

dataset	Add files via upload	1 hour ago
journal papers	Add files via upload	1 hour ago
templates	Add files via upload	1 hour ago
app.py	Add files via upload	1 hour ago

Figure 7.1 Git History

8. CONCLUSION

In conclusion, this project presents a comprehensive approach to grammar error detection and correction through the application of BERT and deep learning techniques. By fine-tuning BERT on a dataset containing various types of grammatical errors paired with their corrected versions, we created a model that effectively identifies and corrects a wide range of grammatical issues. Our system performs well in real-time correction, providing improvements in accuracy and readability across a spectrum of error types, including verb tense, subject-verb agreement, punctuation errors, and more.

The significance of this project lies in both its practical utility and its technical contributions to the field of natural language processing (NLP). By using BERT's powerful bidirectional transformer architecture, we leveraged its contextual understanding of language, enabling nuanced and accurate error correction. Throughout the process, we also conducted extensive evaluation and analysis to assess the model’s performance, which demonstrated promising accuracy rates. This is particularly relevant for applications in academic, professional, and creative writing where quality grammar is crucial.

In addition, this project provides insights into the challenges and strategies of implementing transformer-based models for real-world grammar correction. For instance, tokenization strategies, model fine-tuning, and error-type categorization all required careful consideration to achieve optimal results. Furthermore, developing a web-based interface using Flask allows for user-friendly interaction, making grammar correction accessible and practical for various audiences.

Ultimately, this project highlights the potential for deep learning and transformer-based architectures like BERT to drive meaningful advancements in grammar correction technology, offering a stepping stone toward more intelligent, automated language tools that enhance communication quality across diverse platforms.

9. FUTURE SCOPE

The future scope for your grammar error detection and correction system using BERT could include the following directions:

1. **Enhanced Model Accuracy with Multilingual Support**: Extend the model to handle multiple languages by fine-tuning it on multilingual datasets. This would enable your system to provide grammar correction for a wider range of users, expanding its reach and utility.
2. **Real-Time Deployment and Integration**: Implement real-time correction on various platforms like email clients, chat applications, and text editors. This could be achieved by integrating the system as a browser extension, a mobile app, or an API, offering seamless grammar support across multiple user interfaces.
3. **Context-Aware Error Detection**: Improve the model's accuracy by incorporating additional context to differentiate between subtle grammar nuances and complex sentence structures. This could involve using larger, more diverse datasets or adding contextual embeddings to BERT.
4. **Adaptive Learning and User Feedback**: Introduce adaptive learning mechanisms, where the model learns and adjusts to individual user preferences over time. Integrating user feedback loops could refine the model further by allowing it to learn from corrected outputs, resulting in a more personalized grammar correction system.
5. **Expanding Error Categories**: Add specific modules to handle additional language issues, such as style improvement, colloquialism detection, and tone adjustment. This would make the system capable of providing feedback not just on grammar but also on language style and appropriateness.
6. **Reduced Computational Overheads**: Optimize the model for faster inference on limited computational resources, making it suitable for deployment on edge devices like smartphones and IoT devices. Quantization and distillation techniques could help achieve this.
7. **Advanced Customization Options**: Allow users to select or define grammar rules based on different contexts, such as formal or casual writing. This could also include configuring for various English dialects (American, British, etc.), which would be beneficial in academic, corporate, and creative writing contexts.
8. **Educational Applications**: Develop interactive learning tools to assist language learners. The model could be used to provide grammar suggestions along with explanations, acting as a tutor to help learners understand grammar rules more effectively.

These improvements would not only make the tool more robust and versatile but also increase its adaptability across diverse user needs and environments.

10. APPENDIX

10.1. MINIMUM SOFTWARE REQUIREMENTS

- **Operating System**:

  Windows 10 or later, Linux (Ubuntu 18.04 or later), macOS 10.13 or later.

- **Python Version**:

  Python 3.7 or later (ensure compatibility with machine learning libraries).

- **Python Libraries**:

  transformers (for BERT model implementation)

  torch (PyTorch for deep learning)

  Flask (for web application deployment)

  pandas, numpy (for data processing)

  scikit-learn (for model evaluation metrics)

  jupyter (optional, for interactive model development)

  html and JavaScript (for frontend)

- **Development Environment** :

  Jupyter Notebook or any IDE like PyCharm, VS Code, or Anaconda Navigator.

- **Browser**:

  Latest versions of Chrome, Firefox, Safari, or Edge for testing the web app.

10.2. MINIMUM HARDWARE REQUIREMENTS

The minimum hardware requirements to ensure smooth execution are as follows:

- **Processor**:
    - Minimum: Dual-core CPU (Intel i3 or equivalent)
    - Recommended: Quad-core CPU or higher (Intel i5/Ryzen 3 or above for faster training and inference)
- **RAM**:
    - Minimum: 8 GB (for basic model operation and small datasets)
    - Recommended: 16 GB or more (for handling larger datasets and faster processing)
- **Storage**:
    - Minimum: 10 GB of free disk space (for dataset storage, model files, and libraries)
    - Recommended: 50 GB (to accommodate additional datasets, model checkpoints, and system logs)
- **GPU (Optional but recommended for faster processing)**:
    - NVIDIA GPU with CUDA support (minimum 4 GB VRAM, such as GTX 1050 or higher)
    - For efficient BERT training, GPUs like NVIDIA RTX 2060 or higher are recommended.

11. REFERENCES

1. Yu Qing ”Design And Application Of Automatic English Translation Grammar Error Detection System Based On BERT Machine Vision”, Scalable Computing: Practice and Experience, ISSN 1895-1767, 2024 SCPE,,DOI https://doi.org/10.12694/scpe.v25i3.2770

2. Nancy Agarwal, Mudasir Ahmad Wani and Patrick Bours, “Lex-Pos Feature-Based Grammar Error Detection System for th English Language,”https://doi.org/10.3390/electronics9101686

3. Jared Lichtarge,Chris Alberti,Shankar, “Data Weighted Training Strategies for Grammatical Error Correction” Transactions of the Association for Computational Linguistics, vol. 8, pp. 634–646, 2020, doi:https://doi.org/10.1162/tacl_a_00336

4. Dataset link: **https://www.kaggle.com/datasets/satishgunjal/grammar-correction**