TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A

PROJECT REPORT

ON

NEPALI GRAMMAR CORRECTION

SUBMITTED BY:

RISHAV BHATTARAI (PUL076BCT058)

SUMIT ARYAL (PUL076BCT087)

TAKSHAK BIKRAM BIST(076BCT093)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING

April, 2024

Page of Approval

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

The undersigned certifies that they have read and recommended to the Institute of Engineering for acceptance of a project report entitled **"Nepali Grammar Correction"** submitted by **Rishav Bhattarai**, **Sumit Aryal**, **Takshak Bikram Bist** in partial fulfillment of the requirements for the Bachelor's degree in Electronics & Computer Engineering.

.....
Supervisor
Anku Jaiswal
Assistant Professor
Department of Electronics and Computer Engineering,
Pulchowk Campus, IOE, TU.

.....
External examiner
Subodh Nepal, PhD
Director
Department of Information and Broadcasting
Ministry of Communication and Information Technology
Government of Nepal

Date of approval: 2024/04/18

Copyright

Acknowledgments

We want to express our deepest gratitude and appreciation to all those who have contributed to the completion of this major project.

First and foremost, we are immensely thankful for the guidance and unwavering support provided by our project supervisor, **Asst. Prof. Anku Jaiswal**. Her expertise has been instrumental in shaping the direction of this report and keeping us on the right track.

Furthermore, we express our gratitude to the faculty members of the Department of Electronics and Computer Engineering, with special acknowledgment to **Asst. Prof. Dr. Aman Shakya**, the cluster head of Audio and Natural Language Processing. We would also like to thank our mentor **Mr. Bikram Adhikari** for his valuable suggestions during the course of this project. Their continuous encouragement, knowledge, and dedication to excellence have inspired us to push our boundaries and strive for the highest quality in our work.

In conclusion, we would like to express our deep gratitude to everyone who has played a part in the development of this major project final report. Your contributions, whether big or small, have made a significant impact on its quality and outcome. Thank you for believing in us and being an integral part of this incredible journey.

Abstract

Nepali GEC plays a crucial role in improving the quality of written Nepali text. An annotated corpus of Nepali sentences along with sentences generated by augmenting correct sentences to generate a diverse range of grammatical errors are used for training. The augmentation is done by identifying the Part of Speech tag and root words of verbs and adjectives using Lemmatizer. This study uses BERT models MuRIL and NepBERTa to fine-tune for the GED task of Nepali text. The models performances were assessed using accuracy and training/validation loss, providing a comprehensive assessment of the model's effectiveness in error detection for the Nepali Language which forms the crucial step for GEC. The GEC system developed here, makes use of MLM models of both MuRIL and NepBERTa to predict the mask tokens in input erroneous sentence and thus gives the suggestions which are filtered by the GED model.

Keywords: Nepali Grammar Correction, Nepali Grammar Error Detection, Nepali GEC Corpus

Contents

Page of Approval **ii**

Copyright **iii**

Acknowledgements **iv**

Abstract **v**

Contents **vii**

List of Figures **viii**

List of Tables **ix**

List of Abbreviations **x**

1 Introduction **1**

1.1 Background 2

1.2 Problem Statement 2

1.3 Objectives 3

1.4 Scope 4

2 Literature Review **5**

2.1 Related Work 5

2.2 Related Theory 7

3 Methodology **21**

3.1 Feasibility Study 21

3.1.1 Technical Feasibility 21

3.1.2 Operational Feasibility 22

3.2 Requirement Analysis 22

3.2.1 Functional Requirement 22

3.2.2 Non-Functional Requirement 23

3.3 Corpus Creation 25

3.4 System Design 28
3.4.1 Transformer-Based Training Approach 28
3.4.2 Fine-Tuning Strategy 29

4 Experimental Setup **31**
4.1 Dataset Collection and Preprocessing 31
4.2 Data Augmentation 31
4.3 Corpus Statistics 33
4.4 Model 33
4.5 Model Training and Tuning 34
4.6 GEC Engine 35
4.7 Tools and Libraries 36
4.8 Performance Metrics 39

5 System design **40**
5.1 System Context Diagram 40
5.2 Data Flow Diagram 41
5.3 Use Case Diagram 42
5.4 System Sequence Diagram 44
5.5 Activity Diagram 45

6 Results & Discussion **46**

7 Conclusion **51**

8 Limitations and Future Enhancement **53**
8.1 Limitations 53
8.2 Future Enhancements 53

References **53**

Appendix **56**
Appendix A: Code Snippets 56
Appendix B: Screenshots of Outputs 59

List of Figures

2.1 The Transformer Architecture 11
2.2 BERT For Sequence Classification 14
2.3 BERT For Masked Language Modeling 15

3.1 Flow of how GED model works. 29
3.2 Flow of how the GEC system works. 30

5.1 System Context diagram 40
5.2 Data Flow Diagram diagram 41
5.3 Use case diagram 43
5.4 System Sequence Diagram 44
5.5 Activity Diagram 45

6.1 Performance of Vanilla Transformer over training and validation data 46
6.2 Processing time for MuRIL as MLM 49
6.3 Processing time for NepBERTa as MLM 49

8.1 Code Snippet For GED 56
8.2 Code Snippet For Creating Masked Sentences 57
8.3 Code Snippet For GEC 58
8.4 Output of GED for Correct Sentence 59
8.5 Output of GED for Inccorrect Sentence 59
8.6 Output of GEC Using MuRIL as MLM Model 59
8.7 Output of GEC Using NepBERTa as MLM Model 60
8.8 Output of GEC When Grammatically Correct Sentence As Input 60
8.9 Output of GEC When Model Doesn't Generate Any Suggestion 60

List of Tables

3.1 Verb Inflection 25
3.2 Homophones Error 25
3.3 Punctuation Error 26
3.4 Sentence Structure Error 26
3.5 Pronoun Error 27
3.6 Main Verb Missing Error 27
3.7 Auxiliary Verb Missing Error 27

4.1 Statistics of the Nepali GEC Corpus. 33
4.2 Description of Dataset 34
4.3 Example of Masked Sentences 36

6.1 Training information on models. 47
6.2 Performance of MuRIL. 47
6.3 Performance of NepBERTa. 47
6.4 Example Result 48

List of Abbreviations

GED	Grammar Error Detection
GEC	Grammar Error Correction
BERT	Bidirectional Encoder Representations from Transformers
CUDA	Compute Unified Device Architecture
MuRIL	Multilingual Representations for Indian Languages
NLP	Natural Language Processing
NLU	Natural Language Understanding
SGD	Stochastic Gradient Descent
NMT	Neural Machine Translation
ML	Machine Learning
CPU	Central Processing Unit
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
VCS	Version Control System
POS	Part of Speech
MLM	Masked Language Model
GPT	Generative Pretrained Transformers
CSRF	Cross-Site Request Forgery
XSS	Cross-Site Scripting
HTTP	Hypertext Transmission Protocol
URL	Uniform Resource Locator
ORM	Object-Relational Mapping
MVC	Model-View-Controller
CoNLL	Conference on Computational Natural Language Learning

1. Introduction

The Nepali language is the official language of Nepal, spoken by millions of people as their native tongue. Proper grammar usage is essential for effective communication and written expression in Nepali. However, due to the complexity of Nepali grammar rules, it is common to encounter grammatical errors in written texts, which can hinder comprehension and negatively impact the quality of communication. Nepali is written in the Devanagari script. The same script is used for Hindi, Marathi and Sanskrit. There are 13 vowels and 36 consonants in the Nepali language. [1].

With our Nepali Grammatical correction system, we aim to provide a valuable tool for individuals and organizations to improve the quality and accuracy of their written Nepali communication sentence by sentence. This system will aid in enhancing language proficiency, facilitating effective formal communication, and supporting various domains such as education, administration, and content creation.

Our system analyzes Nepali text input to identify common grammatical errors such as incorrect verb conjugations, sentence structure issues, homonym and punctuation issues. Instead of providing immediate corrections, the system initially detects errors and subsequently offer suggestions and corrections to rectify these issues. This approach assists users in producing grammatically accurate Nepali text based on the errors discussed and also leverages the usage of masked language models in suggesting correct sentences.

Addressing the development of such a system posed several challenges, primarily due to the absence of a comprehensive and representative dataset for training the model, the necessity to adapt existing natural language processing (NLP) techniques to the unique characteristics of the Nepali language, and the complexities inherent in Nepali grammar rules. Consequently, our focus shifted towards resolving some of the grammatical issues encountered during the project's development phase. By addressing these specific errors, we aimed to demonstrate that if the model performs well on these instances, it should also effectively handle other types of grammatical errors in Nepali text.

The successful implementation of this Nepali grammar correction system will not only contribute to improved written communication but also promote the preservation and understanding of the Nepali language. By enabling users to produce accurate and error-free Nepali text, we aim to enhance language proficiency and foster effective formal communication in various domains.

1.1 Background

The Nepali language is an Indo-Aryan language and also the official language of Nepal and is widely spoken by millions of people, plays a crucial role in communication, education, administration and also serves as the *lingua franca* for the natives in Nepal who do not speak Nepali as their first language. The term "Nepali" derived from the name of the country Nepal, was formally adopted by the Government of Nepal in 1933. This occurred when the *Gorkha Bhasa Prakashini Samiti*, a government institution established in 1913 (B.S. 1970) to promote the Gorkha language, changed its name to Nepali *Bhasa Prakashini Samiti* (Nepali Language Publishing Committee) in 1933 (B.S. 1990). This organization, now known as *Sajha Prakashan*, has since been dedicated to the advancement of the Nepali language.[2] Part 1 of the Nepali Constitution addresses the designation of the official language of the Federal Democratic Republic of Nepal[3]. According to Article 6, the official language of the nation encompasses all languages spoken as mother tongues in Nepal. Article 7 specifies that the official language of Nepal is Nepali, written in the Devanagari script as नेपाली.[3]

However, Nepali grammar is complex and prone to errors because of its inherent complexities.[1] These errors can have a significant impact on the clarity and effectiveness of written communication. Traditionally, grammar correction has relied on manual proofreading, which is time-consuming and subject to human limitations. Therefore, there is a growing need for automated systems that can detect and correct grammar errors in Nepali text.

1.2 Problem Statement

The problem we aimed to address in this major project is the lack of an efficient and reliable Nepali Grammatical Error Detection and Correction system. There has been a lot of research and development in the field of grammatical error correction in some languages, especially English. So, existing grammar correction tools primarily focus on major languages such as English and provide a high level of accuracy for the languages like English. However, research and development of Grammatical Error Correction for low-resource language such as the Nepali language is very scarce. So their effectiveness in handling Nepali grammar errors is very limited. We expect for our GEC system to detect the grammatical error in Nepali text and provide correct alternatives.

1.3 Objectives

The primary objective of this major project is to develop a robust Nepali Grammatical Error Detection and Correction system based on neural networks. Our project aims to:

1. Explore and analyze the existing approaches and techniques in the field of NLP for grammatical error detection and correction.
2. Design and build Nepali Grammatical Error Detection and Correction System.
3. Evaluate the performance of the developed systems using various metrics.

1.4 Scope

The scope of this project encompasses the development of a prototype Nepali Grammatical Error Detection and Correction system. The system accepts Nepali text as input, detects whether the input sentence is grammatically correct or not, and provides suggestions for corrections. It focuses on correcting errors related to grammar rules, syntax, and sentence structure. However, semantic errors and context-based corrections are beyond the scope of this project although they can be added in the future.

2. Literature Review

The Nepali language is categorized as a "low-resource" language, indicating that there has been limited research conducted in the field of Nepali Language Processing. Consequently, there is a dearth of comprehensive studies and resources available for this language. The amount of work that has been done in Nepali Natural Language Processing is very scarce. Out of all the work that has been done, most of them are in word classification, Part-of-speech (POS) tagging, morphological analysis, sentiment analysis, and word sense disambiguation.

Being a descendant of the Indo-European language and being written in Devnagari script, the Nepali language inherits its properties from languages like Sanskrit, etc. The distinction between inflection and derivation plays a vital role in linguistics and is very significant in Nepali. That being said, Nepali grammar is composed of morphology and syntax [4]. In the Nepali language, the sentence is formed in Subject + Object + Verb (SOV) order. So, the morphology of the Nepali language is agglutinating in nature. That said, the Nepali language has a huge population but the amount of resources and work on the Nepali language is significantly less.

The amount of research and work that has been done in the field of Nepali Grammar Error Correction is very few. There have been some systems that can check the spelling but there is no system that can check the Nepali grammar end to end. That's why, a Nepali Grammar Error Detection and Correction system is important.

2.1 Related Work

Significant advancements have been made in the field of Natural Language Processing (NLP), although it's important to note that the progress in Nepali NLP specifically has been relatively limited. Despite this, there have been notable achievements, including the development of important linguistic tools and resources. The process of stemming, a pivotal facet of language processing, has been tackled through the conception of rule-based, statistical, and hybrid stemmers tailored for the Nepali language [5]. Moreover, noteworthy headway in Nepali NLP extends to the development of essential linguistic resources. In addition to stemmers, endeavors have culminated in the creation of Nepali-specific word embeddings. Furthermore, POS taggers have been devised for the Nepali language, constituting indispensable preprocessing steps for substantial Nepali NLP undertakings, including the construction of a Nepali grammatical error correction system which, despite the progress in various aspects, remains uncharted territory in the domain.

The genesis of Grammar Error Correction (GEC) systems can be traced through various languages, including English, German, and more. However, the emergence of GEC gained substantial recognition with the organization of the CoNLL-2014 shared task.[6] Prior to this event, despite efforts in other languages, the comprehensive development and attention to GEC were notably catalyzed by the CoNLL-2014 initiative.[6]

Research on GEC for low-resource languages presents unique challenges due to the scarcity of annotated data and limited linguistic resources. Despite these difficulties, several studies have focused on addressing GEC in low-resource language settings, aiming to make writing assistance tools accessible to a wider range of non-native speakers.

One approach explored in low-resource GEC is the utilization of limited parallel data and unsupervised pre-training. Researchers have adapted neural encoder-decoder models and leveraged unsupervised learning techniques to overcome the lack of annotated data. By utilizing the available parallel data and incorporating unsupervised pre-training, these approaches have shown promising results in improving GEC accuracy for low-resource languages[7]. For certain types of error correction, a character-level transformer [8] could be used to capture subtle character-level patterns and correct errors like misspellings and typos after which they could be fed to the word level transformers for correcting a much rather complex grammatical problem given enough resources and data [9].

Another direction of research in low-resource GEC involves transfer learning. Researchers have investigated the transferability of the language models trained on high-resource languages to low-resource languages. By leveraging the pre-trained models from high-resource languages, these approaches aim to transfer knowledge and improve GEC performance in low-resource settings. Transfer learning techniques have demonstrated effectiveness in leveraging the linguistic similarities across languages to enhance the accuracy of GEC for low-resource languages like Nepali Language[10].

Unsupervised learning techniques have also been explored in the context of low-resource GEC. By relying on monolingual data, researchers have developed unsupervised neural machine translation (NMT) methods to tackle GEC for low-resource languages. These approaches leverage self-training and iterative refinement to improve the performance of GEC systems without relying on parallel annotated data. Unsupervised learning methods have shown promising results in achieving competitive accuracy in low-resource GEC, even in the absence of parallel training data [11].

Furthermore, multilingual pre-training has emerged as a promising direction for low-resource GEC. By leveraging the linguistic relationships between multiple languages, researchers have utilized multilingual pre-training to enhance the performance of GEC systems for low-resource languages. This approach enables the model to transfer knowledge across

languages and improve its ability to correct errors in low-resource settings. Multilingual pre-training has demonstrated its efficacy in improving the accuracy of GEC for languages with limited resources [12].

2.2 Related Theory

1. Encoder-Decoder Framework

   The encoder-decoder architecture is a common approach for sequence-to-sequence tasks in NLP. It consists of two main components; an encoder that processes the input sequence and captures its representations, and a decoder that generates the output sequence based on the encoded framework. It has been widely used for machine translation and text summarization. [13]

2. Attention Mechanism

   The attention mechanism is a technique that enables a model to selectively concentrate on specific sections of the input sequence while processing each element of the output sequence. By assigning varying weights or significance to different positions in the input sequence, the attention mechanism allows the model to effectively capture pertinent information, leading to enhanced performance in a range of sequence-to-sequence tasks. This mechanism has gained extensive adoption and further development within the field of natural language processing, substantially improving the model's ability to comprehend and generate contextually relevant sequences. [14]

3. Data Preprocessing

   Data Preprocessing is an important step in preparing data for machine learning tasks. It involves transforming, cleaning and organizing data to ensure its quality, consistency, and compatibility with the chosen algorithms. Various steps are taken to transform raw text data into a format suitable for NLP models. Some common preprocessing steps are removing punctuation, tokenizing the text into words or subwords, converting text to lowercase, removing stop words, and performing stemming and lemmatization to normalize word forms. Data preprocessing is crucial in NLP to improve the quality and efficiency of subsequent analysis or modeling tasks. The following are the crucial steps involved in data preprocessing:

   (a) Tokenization

   Tokenization is a way of splitting text into smaller parts, like words or even individual characters. It's an important step when working with text on computers because it helps us understand and work with the text more easily. By breaking text into smaller units called tokens, we can analyze and process the text in

a more detailed way, like counting words or finding patterns. It's like breaking a sentence into separate words, so we can understand each word better and do more with them. Tokenization plays a crucial role in various natural language processing tasks, such as machine translation, sentiment analysis, and named entity recognition. In Nepali, tokens can be separated based on whitespaces or specific language rules. For example, the sentence "मेरो नाम राम हो" (meaning "My name is Ram") can be tokenized into the following tokens: ["मेरो","नाम","राम ","हो"].

(b) Stemming

Stemming is a linguistic process used in natural language processing to reduce words to their base or root form. It involves removing prefixes, suffixes, and other affixes from words to extract the core morphological meaning. Stemming aims to simplify words so that variations of the same word are treated as a single entity, which can aid in various language processing tasks. Stemming is useful for various NLP tasks, such as information retrieval, search engines, and text analysis, where recognizing different forms of a word as the same can help improve efficiency and accuracy. In Nepali, stemming is done by separating the suffix (प्रत्यय). For example, from the word "लेख्नुहोस्", the root form "लेख्नु" could be extracted by stemming the suffix(प्रत्यय), "-होस्".

(c) Lemmatization

A lemmatizer is a linguistic tool used in natural language processing to reduce words to their base or dictionary form, known as the lemma. Unlike stemming, lemmatization considers the word's context and part of speech to ensure that the resulting lemma is a valid word with its intended meaning. Lemmatization is a more accurate method compared to stemming, as it produces linguistically valid forms. Lemmatization is particularly useful in applications where maintaining semantic accuracy is crucial, such as language translation, sentiment analysis, and information retrieval. By converting words to their dictionary forms, lemmatization helps improve the accuracy of linguistic analysis and enhances the overall quality of language processing tasks.

(d) Word Embedding

Word embedding is a technique used in natural language processing (NLP) to represent words as dense numerical vectors in a high-dimensional space. It aims to capture the semantic and contextual relationships between words, allowing machines to understand and work with words in a more meaningful way. Word embeddings have numerous applications in NLP. They are used to improve perfor-

mance in various tasks, such as language modeling, sentiment analysis, machine translation, and named entity recognition. By representing words as dense vectors, word embeddings enable machines to process and understand language more effectively, facilitating better performance in a wide range of NLP applications.

(e) POS Tagger

A Part-of-Speech (POS) tagger is a language processing tool that assigns grammatical tags to words in a text based on their syntactic roles within a sentence. These tags indicate the word's part of speech, such as noun, verb, adjective, adverb, etc. POS tagging is valuable for understanding sentence structure, analyzing grammatical relationships, and extracting linguistic features. For example, for a Nepali sentence, त्यो ठाउँमा सुनको खजुर खेती गर्दै छ।, the POS tagging would be ["त्यो (सर्वनाम)" , "ठाउँमा (नाम)", "सुनको(नाम)", "खजुर(नाम)", " खेती(क्रियाविशेषण)", "गर्दै(क्रिया)", "छ(क्रिया)"], where सर्वनाम means pronoun, नाम means noun, क्रियाविशेषण means the adverb and the क्रिया means the verb.

(f) Data Augmentation

Data Augmentation is a technique that is used to artificially increase the size and diversity of training data sets by applying a transformation to the original data or by modifying it. In other words, we can say that data augmentation is a process of creating an entirely new dataset from an available dataset by introducing some changes to it. Data Augmentation can maintain the diversity in the dataset as well as increase its size. In the case of Nepali Grammatical Error Correction, data augmentation can be employed to generate additional training data that exhibits different types of errors, linguistic variations as well as sentence structure. This can be achieved by word replacement, sentence paraphrasing, grammatical transformations, etc. By incorporating these data augmentation techniques, the training data set for Nepali GEC can be expanded to include a wider range of error patterns, sentence structures, and other linguistic variations. This enables the model to learn more effectively and generalize its error correction abilities to diverse Nepali texts.

4. Transformer

A transformer is a groundbreaking neural network architecture that has revolutionized various natural language processing tasks. Unlike traditional sequence-based models, transformers leverage a self-attention mechanism to capture complex relationships between words in a sequence, enabling them to consider context over long distances effectively. This architecture consists of an encoder-decoder framework or solely an

encoder, with each layer featuring multi-head self-attention and feed-forward neural networks. Transformers have demonstrated exceptional proficiency in tasks like machine translation, text generation, and sentiment analysis, among others, by learning contextual information and dependencies within input data. They have significantly contributed to advancements in understanding language semantics and have become a cornerstone of modern NLP, paving the way for innovative developments in the field.

The Transformer, introduced in the paper "Attention is All You Need" by Vaswani et al. [9], revolutionized sequence modeling in natural language processing and other domains. This model is distinct for its attention mechanism, which enhances training speed and effectiveness.

At its core, the Transformer is built upon a mechanism called "self-attention." Unlike traditional sequence models like LSTMs or RNNs, which process data sequentially, the Transformer leverages self-attention to weigh the significance of different words in a sequence against each other. This attention mechanism enables the model to consider the entire input sequence simultaneously, capturing dependencies between words efficiently.

The key components of the Transformer are the encoder and decoder layers. The encoder processes the input sequence, utilizing self-attention to create contextual representations for each word in the sequence. Meanwhile, the decoder generates an output sequence by attending to the encoded information and predicting the next word based on the context.

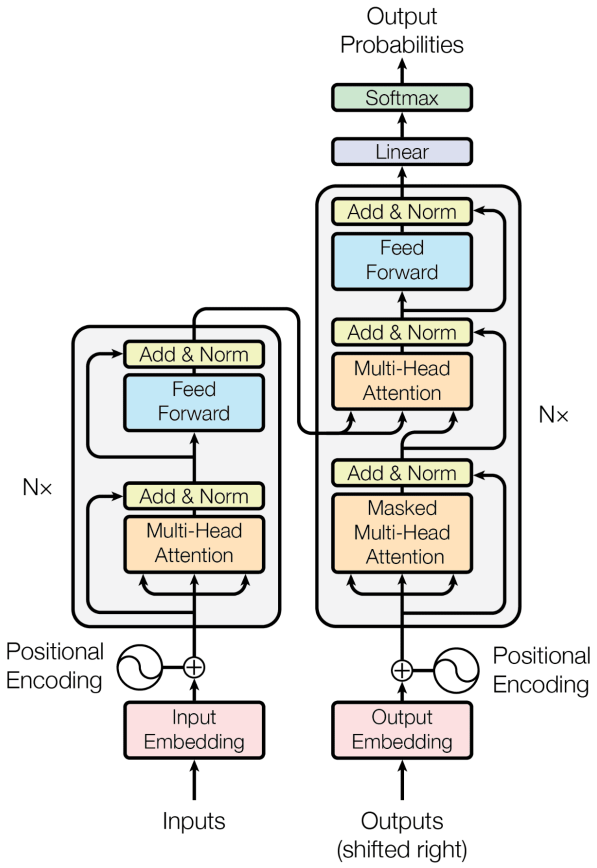

Figure 2.1: The Transformer Architecture
[9]

The transformer consists of stacks of encoder and decoder blocks with each block including self-attention, recurrent connections, and feed-forward neural networks. Below are the descriptions of the encoder and decoder blocks.

(a) Encoder

It is responsible for processing the input sequence of tokens, $X_T = [x_1, x_2, x_3,..., x_n]$, and producing a sequence of hidden states that capture the meaning and context of each token in the input by incorporating its positional encodings. The positional encoding for each token is calculated by using the following mathematical expression:

$$\mathrm{PE}_{(\mathrm{pos},2i)} = \sin\left(\frac{\mathrm{pos}}{10000^{\frac{2i}{d_{\mathrm{model}}}}}\right)$$
$$\mathrm{PE}_{(\mathrm{pos},2i+1)} = \cos\left(\frac{\mathrm{pos}}{10000^{\frac{2i}{d_{\mathrm{model}}}}}\right)$$

$\text{PE}_{(\text{pos},2i)}$ represents the $2i^{\text{th}}$ dimension of the positional encoding for the word at position pos.

$\text{PE}_{(\text{pos},2i+1)}$ represents the $(2i+1)^{\text{th}}$ dimension of the positional encoding for the word at position pos.

pos refers to the position of the word in any given sequence.

d_{model} denotes the dimensionality of the model.

Each encoder layer includes a self-attention mechanism that computes the attention scores between all pairs of tokens in the input sequence and a feed-forward neural network that applies a non-linear transformation(softmax) to the output of the self-attention mechanism. This self-attention mechanism computes a weighted sum of the hidden states for each token in the input sequence, where the weights are based on the similarity between the token and all other tokens in the sequence. This allows the encoder to focus on the most relevant parts of the input sequence for each token, taking into account the context in which it appears. However, the transformer model uses multi-head self-attention to capture multiple relationships, increase expressive power, become robust to variations in data, and address regularization. The mathematical expression for calculating each head's self-attention is as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Here, Q, K, and V are matrices of queries, keys, and values respectively. The d_k is the dimension of keys that is utilized to scale the resultant score. For single head attention, $d_k = d_{\text{model}}$ but for multi-head attention, d_k is given by:

$$d_k = \frac{d_{\text{model}}}{h}$$

Where h is the number of heads. The multi-head attention is essentially the amalgamation of each head's outcome which can be represented as,

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{Head}_1, \text{Head}_2, ..., \text{Head}_h)W_o$$

where, $\text{Head}_i = \text{Attention}(QW_i, KW_i, VW_i)$

Finally, a feed-forward neural network takes the response of the multi-head self-attention followed by a recurrence connection and applies a non-linear transformation to the output of the self-attention mechanism, which allows the encoder to capture more complex relationships between the tokens in the input sequence.

(b) Decoder

It is responsible for generating the output sequence based on the encoded input sequence generated by the encoder. The decoder is auto-regressive takes the encoded input sequence and generates the output sequence token-by-token. Each decoder layer comprises of masked multi-head self-attention layer, multi-head attention layer, and feed-forward neural network layer. First, masked self-attention is computed over the target sequence Y. The masked multi-head self-attention layer is similar to the self-attention layer in the encoder but with a mask applied to ensure that the decoder cannot attend to future tokens in the output sequence. This sublayer allows the decoder to attend to relevant parts of the output sequence generated so far and capture the dependencies between the tokens in the output sequence. Next, attention is computed over the encoded hidden representations H. The multi-head attention layer is responsible for attending to the encoded input sequence generated by the encoder. This sub-layer enables the decoder to incorporate information from the input sequence into the output sequence and produce a translated version of the input sequence. Then, a position-wise feed-forward network is applied to the output representation obtained in the previous step. The feed-forward neural network layer applies a non-linear transformation including residual connections and layer normalization to the output of the attention layers to generate the final output sequence. During training, the decoder uses teacher forcing, where the true previous token is fed as input to the decoder at each time step. During inference, the decoder generates the output sequence token-by-token by recursively predicting the most likely token at each time step based on the previous tokens and the encoded input sequence.

5. Fine-tuning

Fine-tuning refers to adjusting and calibrating various elements to achieve optimal performance or functionality. In diverse contexts, from music to machinery, fine-tuning involves delicate modifications aimed at achieving precision, accuracy, and efficiency. In machine learning, fine-tuning refers to the process of tweaking pre-trained models to adapt them to specific tasks or datasets, enhancing their ability to make accurate predictions or classifications within a particular domain. It's a crucial step, akin to refining the details of a masterpiece, ensuring that the system performs optimally and meets the desired criteria. Fine-tuning demands a keen eye for detail and a deep understanding of the nuances within the system being adjusted, ultimately resulting in improved performance and effectiveness.

6. Bidirectional Encoder Representations from Transformers (BERT)

   BERT is a transformer-based machine-learning technique for Natural Language Processing (NLP) pre-training. It was introduced by researchers at Google AI Language in 2018. It is a pre-trained encoder based model that leverages bidirectional training, allowing it to learn from the entire sequence of words simultaneously. BERT employs a masked language modeling task, where randomly selected words are masked, and the model is trained to predict the masked words based on the context. Additionally, it is trained on a next-sentence prediction task to understand the relationship between sentences. BERT generates contextualized word embeddings, where the representation of a word is influenced by the words around it, enabling a better understanding of word meanings in context. After the pre-training, BERT can be used for specific downstream tasks such as Sequence Classification, Masked Language Modeling, etc

   (a) BERT For Sequence Classification

   A classification layer is added on top of BERT architecture. The layer consists of fully connected layers followed by a softmax activation function. The output dimensionality of the final layer depends on the number of classes in the classification task.

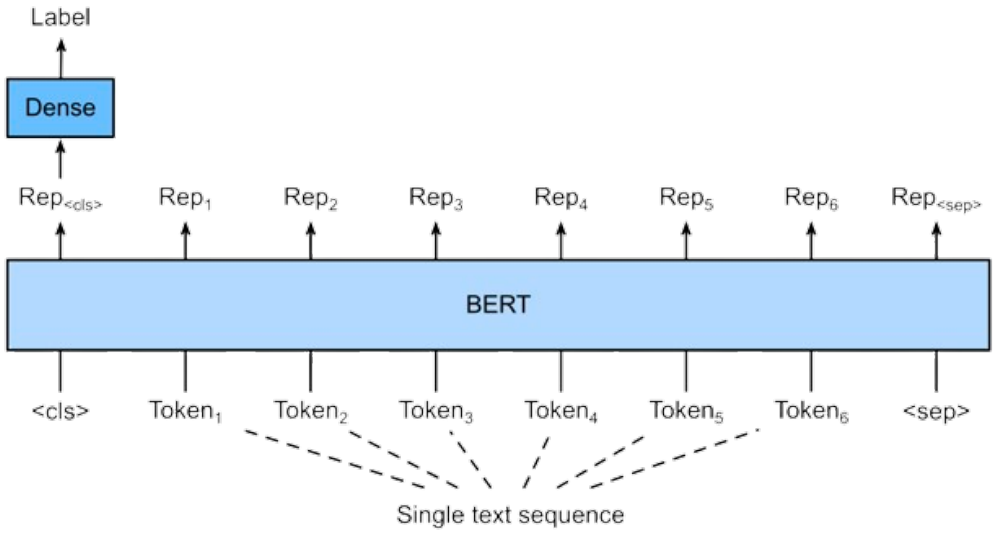

Figure 2.2: BERT For Sequence Classification

   (b) BERT For Masked Language Modeling

   An input token is masked and bidirectional representations of the words, capturing syntactic and semantic dependencies from both left and right contexts are used to predict the masked token.

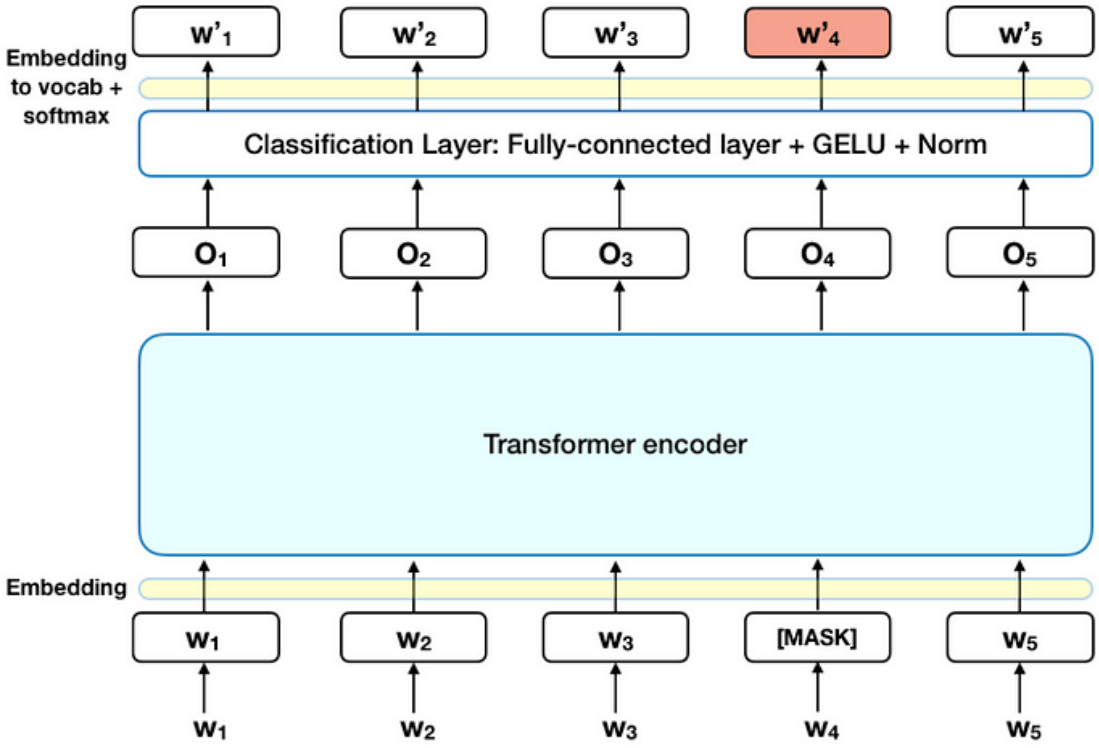

Figure 2.3: BERT For Masked Language Modeling

7. Optimizers

   Optimizers are algorithms or methods used to minimize an error function(loss function) or to maximize the efficiency of production. Optimizers are mathematical functions which are dependent on the model's learnable parameters. Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

   (a) RMSProp

   RMSProp (Root Mean Square Propagation) is an optimization algorithm designed to address the limitation of traditional gradient descent algorithms like slow convergence and difficulty in choosing a suitable learning rate. RMSProp is an adaptive learning rate optimization algorithm which works by exponentially decaying the learning rate every time the squared gradient is less than a certain threshold. While training, the gradients are usually such that one changes a lot while other does not change much.

   The update rule for RMSProp can be described as follows:

$$v_t = \beta v_{t-1} + (1-\beta) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} g_t$$

where:

- v_t is the exponentially decaying average of squared gradients.
- g_t is the gradient at iteration t.
- β is the decay rate parameter (typically close to 1).
- η is the learning rate.
- ϵ is a small constant to prevent division by zero (typically a small value like 10^{-8}).
- θ_t and θ_{t+1} are the parameter vectors at iteration t and $t+1$ respectively.

RMSProp adapts the learning rate separately for each parameter by using a moving average of squared gradients. It effectively normalizes the gradients based on their past magnitudes, helping to stabilize and speed up the training process.

(b) AdaGrad

AdaGrad (Adaptive Gradient Algorithm) is an optimization algorithm that adapts the learning rate of each parameter based on the historical gradients for that parameter. It performs larger updates for infrequent parameters and smaller updates for frequent parameters.

The update rule for AdaGrad can be described as follows:

$$G_t = G_{t-1} + g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t} + \epsilon} g_t$$

where:

- G_t is the sum of squared gradients up to iteration t.
- g_t is the gradient at iteration t.
- η is the learning rate.
- ϵ is a small constant to prevent division by zero (typically a small value like 10^{-8}).
- θ_t and θ_{t+1} are the parameter vectors at iteration t and $t+1$ respectively.

AdaGrad effectively reduces the learning rate for parameters that are updated frequently and increases it for parameters that are updated infrequently. However, it suffers from a diminishing learning rate problem, where the learning rate becomes too small over time, making it less effective for later stages of training.

(c) Adam

Adaptive Moment Estimation(Adam) stands out as being one of the most highly efficient optimization algorithms designed to adjust the learning rate for each parameters while training. At its core, Adam optimizer is designed to adapt to the characteristics of the data. It does this by maintaining the individual learning rates for each parameter in our model. These rates are adjusted as the training progresses based on the input data it encounters.

Adam (Adaptive Moment Estimation) is an adaptive learning rate optimization algorithm that combines the advantages of two other popular optimization algorithms: AdaGrad and RMSProp.

The update rule for Adam can be described as follows:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

where:

- m_t and v_t are the first and second moment estimates of the gradients respectively.
- g_t is the gradient of the learnable parameters at iteration t.
- β_1 and β_2 are the exponential decay rates for the moment estimates (typically close to 1).
- η is the learning rate.
- ϵ is a small constant to prevent division by zero (typically a small value like 10^{-8}).
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of the moments to account for the initialization bias.

- θ_t and θ_{t+1} are the parameter vectors at iteration t and $t+1$ respectively.

Adam is known for its efficiency and robustness in training deep neural networks.

(d) AdamW

AdamW is a modification of this Adam optimizer. AdamW solves this problem by decoupling weight decay from the gradient-based optimization step. It achieves this by applying weight decay directly to the weights after each optimization step, rather than including it in the update rule. This means that weight decay is only applied to the parameters that should be regularized, such as the weights, and not to the ones that shouldn't, such as the bias terms. This results in improved performance and better convergence.

The update rule for AdamW is similar to Adam, but it includes the weight decay term in the parameter update step:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t - \lambda \theta_t
\end{aligned}
$$

where:

- m_t and v_t are the first and second moment estimates of the gradients respectively.
- g_t is the gradient at iteration t.
- β_1 and β_2 are the exponential decay rates for the moment estimates (typically close to 1).
- η is the learning rate.
- ϵ is a small constant to prevent division by zero (typically a small value like 10^{-8}).
- λ is the weight decay coefficient.
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of the moments to account for the initialization bias.
- θ_t and θ_{t+1} are the parameter vectors at iteration t and $t+1$ respectively.

AdamW improves upon Adam by decoupling weight decay from the adaptive learning rate, allowing for more stable and effective training. This separation prevents the learning rate from being affected by the weight decay term, leading to better generalization performance.

8. CUDA®

   Compute Unified Device Architecture (CUDA) is a parallel computing platform and API model created by NVIDIA® which allows for softwares and programs to use the GPUs for accelerated general purpose processing.. It allows developers to harness the computational power of NVIDIA® GPUs (Graphics Processing Units) for general-purpose processing tasks beyond just graphics rendering.

   CUDA® enables developers to write programs that can execute on the GPU, taking advantage of the massive parallel processing capabilities it offers. This allows for significant acceleration of tasks that can be parallelized, such as scientific simulations, machine learning, image and video processing, and more. CUDA® is a software layer that gives the computer programs direct access to the GPU's virtual instruction set and its parallel computational elements to perform complex matrix computations parallely which helps in effectively training newer and complex deep learning models known today.

9. Frontend

   Frontend development refers to the practice of creating user interfaces (UIs) and implementing the visual and interactive elements of web applications or websites. It involves the use of various technologies and languages, including HTML, CSS, and JavaScript, to build the client-side components that users directly interact with. HTML (Hypertext Markup Language) provides the structure and content of web pages, defining elements such as headings, paragraphs, lists, and links. CSS (Cascading Style Sheets) is responsible for controlling the visual appearance and presentation of these elements, including layout, typography, colors, and animations. JavaScript, a programming language, adds interactivity, dynamics, and behavior to web pages, enabling features like form validation, animations, and single-page applications (SPAs). In our case, the front end is done using the Django templating engine and Python forms.

10. Backend

    Backend development refers to the server-side logic and infrastructure that powers web applications and websites. It involves the design, implementation, and maintenance

of the components that handle data processing, business logic, and communication with databases and other external services. In our case, the backend was done using Django. Django is a high-level, open-source Python web framework that follows the Model-View-Template (MVT) architectural pattern. It is designed to simplify the development of secure and maintainable web applications by providing a wide range of built-in features and tools. Some of the tools include Object Relational Mapper, Admin Interface, Templating System, URL Routing and Form Handling

3. Methodology

3.1 Feasibility Study

The section evaluates the technical and operational feasibility.

3.1.1 Technical Feasibility

1. Availability of datasets

   Being a low resource language, there was limited availability of datasets for the Nepali language so a lot of time was required to gather and filter out the Nepali texts to make a reliable Nepali dataset of texts and inflect the errors on the dataset to generate the pair of correct and error inflected sentences.

2. Computing resources

   Cloud computing services, including Google Colab™ and Kaggle Notebooks™, were used to leverage their scalable computing resources for training. However, the problem of financing comes into play as the free version of these platforms comes with limitations, especially in terms of resource availability and reliability. Occasional downtimes and connectivity issues disrupt the ongoing tasks. In the case of Google Colab™, access to GPU resources is not available 24/7, which hinders the model training process. As a result, the models were trained locally on a laptop computer equipped with a dedicated NVIDIA® GeForce® RTX™ 4060 GPU, consisting of 8GB VRAM. The device also features an Intel® Core™ i9-13950HX CPU and 32GB DDR5 system memory.

3. Tools and Frameworks

   Python will be our main language for development and we can also utilize various open-source frameworks such as PyTorch, HugggingFace Transformers library and scikit-learn, among others, for implementing machine learning (ML) functionalities in the project. CUDA version 11.8 was also used to harness the power of parallel computing capability of the GPU and the torch version 2.2.0 (cuda11.8 as compute platform) was used alongside. Tensorflow version 2.9.0 was used.

3.1.2 Operational Feasibility

1. User Acceptance

   The grammar detection and correction system is intended for students, teachers, language enthusiasts, content creators, academic institutions, and language technology researchers. Surveys, and interviews, can be conducted to assess the interest, acceptance, and usability of the system among the target users.

2. Scalability

   The scalability of the system depends on the two main factors i.e. more and more training data along with the training resource availability and also the user growth. Due to the low availability of the training resource to train on a huge dataset, the system could take a very long time to scale and also the low availability of dataset affects the scalability over time as more and more data have to be collected and augmented to emulate the grammatical errors in Nepali text. User growth also affects the system as more and more users would feed the data in the system to get the predictions which helps in gathering huge data over time due to the users feeding the data.

3.2 Requirement Analysis

3.2.1 Functional Requirement

1. Text Input

   The system should allow users to input Nepali text sentences or paragraphs for grammar correction.

2. Grammar Correction

   The system should employ neural networks to analyze and correct grammar errors in the Nepali text.

3. Correction suggestions

   The system should provide suggestions and explanations for the grammar corrections made.

4. Sentence-level correction

   The system should handle individual sentences for correction.

5. Grammar rule coverage

   The system should cover a wide range of Nepali grammar rules to effectively correct errors.

6. User Feedback

   Provide a mechanism for users to provide feedback on the accuracy of the grammar corrections.

3.2.2 Non-Functional Requirement

1. Performance

   a. R1.1: System response time should be optimized, with a maximum response time of 5 seconds for user input processing.

   c. R1.2: Page loading time should be minimized, with pages loading within 3 seconds to enhance user satisfaction.

   d. R1.3: The system should exhibit low latency for data retrieval and processing to ensure efficient grammar correction.

2. Scalability

   a. R2.1: The system should be scalable to accommodate an increasing number of users without compromising performance.

   b. R2.2: The system should handle at least 50 concurrent users to support simultaneous usage.

3. Security

   a. R3.1: Sensitive user data and information should be securely encrypted to protect user privacy.

4. Usability

   a. R4.1: The system should have a user-friendly interface for easy navigation and usage.

   b. R4.2: Clear and concise error messages should be provided to assist users in addressing any issues.

   c. R4.3: The system should maintain a consistent layout and design across all application modules.

   d. R4.4: The system should be responsive and compatible with different screen sizes and resolutions.

5. Reliability

   a. R5.1: The system should have a high level of reliability, with a low probability of failure.

   b. R5.2: The system should be designed to recover quickly from any failures that occur.

6. Availability

   a. R6.1: The system should have high availability with minimal downtime to ensure continuous access.

   b. R6.2: Data recovery mechanisms should be in place, including backup systems, to mitigate the impact of any potential disasters.

7. Maintainability

   a. R7.1: The system should be designed for easy maintenance and updates to support ongoing improvements.

   b. R7.2: A version control system should be implemented to track changes and facilitate efficient development processes.

   c. R7.3: Clear and detailed documentation of the system's API should be provided for ease of understanding and integration.

8. Compliance

   a. R8.1: The system should comply with relevant laws, regulations, and industry standards to ensure legal and ethical operation.

9. Internationalization

   a. R9.1: The system should support different date and time formats to accommodate international users.

   b. R9.2: Cultural conventions and sensitivities should be considered to ensure inclusivity and user satisfaction.

10. Extensibility

    a. R10.1: The system should be designed to be extensible, allowing for the addition of new features and functionality.

    b. R10.2: The system should have a modular design to facilitate easy replacement of individual components.

3.3 Corpus Creation

One of the clinical tasks of this project is the corpus creation. In the case of Grammatical Error Correction, one of the major barriers is the availability of a large corpus. In recent years, the creation of such a corpus has been done in various other major languages such as English, German, etc but it is not the case with Nepali language. Hence, we take a leap forward to develop a large parallel corpus for the Nepali Grammatical Error Correction. To accomplish the task, we've categorized different issues such as verb inflections, Homophones(words that sound the same but have different spelling) errors, sentence structure flaws, punctuation mistakes, and incomplete sentences due to sentence flaws. The incomplete sentences are further divided into two categories such as missing subject (pronoun) and missing verb. The corpus includes the specific grammatical errors as described below:

1. Verb Inflection

   Verb inflection refers to the modification of a verb that expresses a different grammatical form of the verb. By undergoing verb inflection, the relationship between the verb and its associated subject will be disrupted which creates an error in the sentence.

   For example,

Incorrect	Correct
बाबाले सर्प बारे अरू बढी केही बोल्छ ।	बाबाले सर्प बारे अरू बढी केही बोल्नुभएन ।

Table 3.1: Verb Inflection

2. Homophones Error

   Homophones are words that sound alike but have different meanings and spellings. They play a significant role in communication often leading to confusion due to the similarity in pronunciation. The use of wrong homophones disrupts the sentence meaning leading to incorrect sentences. For example,

Incorrect	Correct
दुर्गम क्षेत्रका अरू जनताले पनि उनीहरू बाट पात सिक्नुपर्छ ।	दुर्गम क्षेत्रका अरू जनताले पनि उनीहरू बाट पाठ सिक्नुपर्छ ।

Table 3.2: Homophones Error

3. Punctuation Error

Punctuations are the symbols that aid in clarity, structure, and comprehension. In Nepali language punctuation marks such as commas(,), full stop( ), question mark(?), exclamation mark(!), and others. The incorrect use of punctuation leads to misunderstanding or ambiguity. They affect the clarity of the sentence. For example,

Incorrect	Correct
तर यसका लागि निजी स्कूलहरू मात्र दोषी छैनन् ?	तर यसका लागि निजी स्कूलहरू मात्र दोषी छैनन् ।

Table 3.3: Punctuation Error

4. Sentence Structure

Sentence structure refers to the arrangement of words and phrases to form coherent and meaningful sentences. The incorrect arrangement of words in a sentence results in a grammatically incorrect sentence. The incorrect sentence usually changes the meaning of the sentence and makes it difficult to understand the sentence. For example,

Incorrect	Correct
एकै कोठा मा सुत्ने दाजुभाइ पनि बीच कुराकानी हुन छाडेको छ ।	एकै कोठा मा सुत्ने दाजुभाइ बीच पनि कुराकानी हुन छाडेको छ ।

Table 3.4: Sentence Structure Error

5. Sentence Fragments

Sentence Fragments are incomplete collections of words that lack a subject or a verb, which doesn't form a complete sentence. When writers omit necessary components, these fragments can cause confusion or ambiguity in communication, potentially leading to misunderstanding. It is further divided into two parts as

(a) Subject Missing: This sentence fragment contains those sentences in which the subject is not included. In case of subject missing, we cannot remove the noun as it plays a crucial role in conveying the intended message. So only the pronoun is removed. A pronoun is a word that substitutes for a noun or noun phrase. It's used to avoid repeating the same noun multiple times in a sentence or paragraph. Pronouns can refer to people, places, things, or ideas previously mentioned or understood in the context of the conversation or text. The absence of the pronoun in a sentence leads to ambiguity or a lack of clarity regarding the subject or object being referenced. For example:

Incorrect	Correct
सूचना क्रान्तिको दुनिया मा मख्ख परेर ठूलो भ्रान्ति पालिरहेका छौं ।	हामी सूचना क्रान्तिको दुनिया मा मख्ख परेर ठूलो भ्रान्ति पालिरहेका छौं ।

Table 3.5: Pronoun Error

(b) Verb Missing: This sentence fragment contains those sentences in which the verb is not included. It is further divided into two parts such as Main Verb Missing Error and Auxiliary Verb Missing Error which are described as follows.

i. Main verb Missing Error

Main verbs, also known as principal verbs or lexical verbs, are fundamental components of sentences that convey the action or state of being. Unlike auxiliary verbs (helping verbs), which assist the main verb in forming verb phrases, main verbs stand alone and carry the primary meaning in a sentence.

Incorrect	Correct
यो टेक्निक पनि प्रभाववाद सँग सम्बद्ध ।	यो टेक्निक पनि प्रभाववाद सँग सम्बद्ध छ ।

Table 3.6: Main Verb Missing Error

ii. Auxiliary Verb Missing Error

Auxiliary verbs, also known as helping verbs, are used alongside main verbs to add functional or grammatical meaning to a sentence. They assist the main verb in conveying various aspects such as tense, mood, voice, or aspect. Absence of the auxiliary verb in a sentence results in a loss of information regarding aspects like tense, mood, voice, or aspect, leading to ambiguity or an incomplete expression of the action or state described in the sentence. For example:

Incorrect	Correct
खाद्यान्नकै हक मा पनि सके सम्म खेर गरी खाना नै नबनाए हुने ।	खाद्यान्नकै हक मा पनि सके सम्म खेर जाने गरी खाना नै नबनाए हुने ।

Table 3.7: Auxiliary Verb Missing Error

3.4 System Design

3.4.1 Transformer-Based Training Approach

The gathered data will undergo a process of tokenization, wherein sentences will be transformed into individual words through techniques such as word segmentation or byte-pair encoding. This step is crucial in preparing the text for input into the transformer model, breaking down the text into smaller units or tokens to facilitate language understanding and grammar error correction with the added advantage of self-attention mechanisms that enable the model to capture long-range dependencies and contextual information.

Training a transformer model for Nepali GEC involves a direct approach where pairs of error and correct sentences as from the created corpus serve as the primary training data. The training process revolves around feeding the transformer model with pairs of sentences where one sentence would contain the grammatical errors(input) and the other presents the corrected version(output). These pairs serve as direct training instances for the model, guiding it to understand the associations between erroneous and correct Nepali sentences, with self-attention aiding in capturing intricate linguistic nuances.

Each sentence pair would undergo tokenization and vectorization, converting the sentences into numerical representations for these sentences X = [x_1, x_2, x_3, ..., x_n] and Y = [y_1, y_2, y_3, ..., y_m] where x_i and y_j are the i^{th} and j^{th} word of the erroneous and correct sentences X and Y respectively. The model's training involves iterative optimization using techniques like back-propagation, gradient-based optimization algorithms like SGD or Adam, and careful adjustments of hyper-parameters, with self-attention playing a pivotal role in enhancing the model's understanding of contextual dependencies.

The model would learn directly from the provided error-correction pairs aiming to imbue the transformer model with the ability to discern and rectify Nepali-specific grammatical errors solely through the presented error-corrected sentence pairs.

3.4.2 Fine-Tuning Strategy

In contrast to the direct training approach, an alternative method for Nepali GEC involves leveraging the Large Language Models BERT. These pretrained models have a comprehensive understanding of the nuances involved in Nepali linguistics.

Leveraging such pre-trained models, would reduce the burden of pretraining the model on large Nepali corpus as it takes a lot of computing time for a low end device that is used throughout the completion of the project. The process of fine-tuning the model on our corpora for Nepali GEC becomes essential.

To carry out the task for GEC, we use a pre-trained BERT model, which is then fine-tuned with the generated corpora for correct and erroneous sentences for single sentence classification to identify whether the sentence is grammatically correct or not. This fine-tuning will give us the GED model. We would use the MLM of the BERT model to come up with alternate sentences and use the fine-tuned GED model to come up with the correct suggestions.

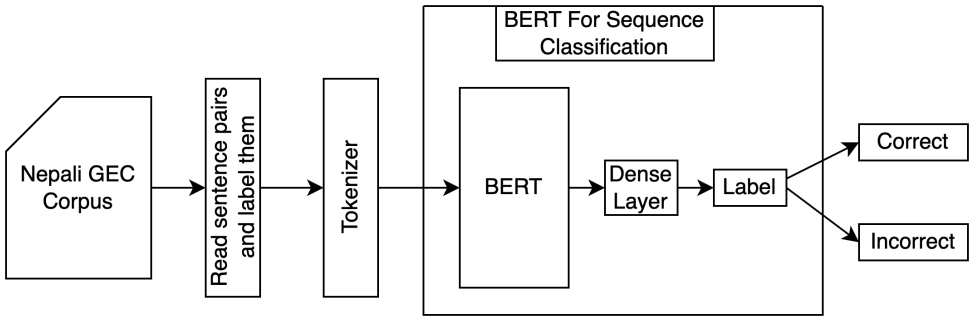

Figure 3.1: Flow of how GED model works.

The constructed corpora is read and labelled as correct and incorrect and passed to the tokenizer which preprocesses and tokenize the raw text input into a format suitable for input for our BERT model. The model iterates over the corpus learning its intricacies generating an intermediate value as BERT is an encoder model. The output is passed through a Dense layer which is a classification layer helping in classifying whether the input sentence is correct or not. The model learns from the sentences and their corresponding labels for classification purpose.

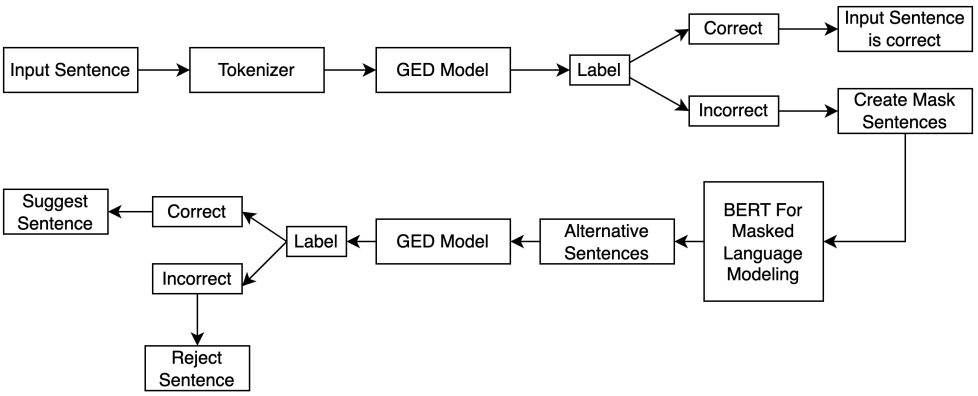

Figure 3.2: Flow of how the GEC system works.

The GED model forms the core of our GEC ecosystem. If the GED model detects that the input sentence is incorrect, the masked sentences are created where [MASK] token is injected to the parts of the sentence. Then the BERT MLM is used to generate sentences by predicting the [MASK] tokens in all the sentences. For each incorrect sentence, we inject [MASK] in two different ways i.e. masking each word and adding [MASK] token in each space of the sentence. The generated sentences are yet again passed to the GED model to detect whether the sentences are correct or not. The incorrect labeled sentences are discarded whereas the correct sentences are passed as suggestion to the end user.

4. Experimental Setup

The setup for the experiment involved in this project in order is as follows:

4.1 Dataset Collection and Preprocessing

The project utilized data sourced from a variety of freely available news portals in the public domain, with Rabindra Lamsal facilitating the process[15]. These portals included Ekantipur, Nagariknews, Setopati, Onlinekhabar, Karobardaily, Ratopati, News24nepal, Reportersnepal, Baahrakhari, Hamrokhelkud, and Aakarpost[15]. Subsequently, a data cleaning process was undertaken to refine the collected data. This involved discarding sentences that exceeded 20 words or were less than 3 words in length, while also accounting for punctuation marks. Additionally, any characters not conforming to the Devnagari Script were eliminated, and English numerals were converted to Nepali numerals for consistency within the text. Following this, a step was taken to extract unique sentences to remove redundancy. Sentences containing only one parenthesis or single or double quotes were also discarded. Finally, the processed sentences were stored in a .txt file for future reference and utilization.

4.2 Data Augmentation

The different types of errors discussed above are generated on the collected data employing noise injection techniques. Each sentence is regarded as a set of words, denoted by S = $\{W_1, W_2, ..., W_{N-1}, W_N\}$ where N represents the sentence length which is a positive integer. Every word $W_i \in S$ is viewed as a collection of Nepali characters: $W_i = \{C_1, C_2, ..., C_{M-1}, C_M\}$ where M stands for the length of the word which is also a positive integer. It is ensured that each artificially flawed sentence contains only one mistake. However, some sentences may contain multiple words with errors, leading to several flawed versions. Therefore, this process yields one correct sentence alongside multiple incorrect forms.

Firstly, the verbs are extracted from the data using the POS Tagger. After this, the lemma of the verbs are extracted using the hybrid approach of the Lemmatizer. From the verbs and lemma, the suffixes are collected which is as A = $\{a_1, a_2, ..., a_D\}$ where $a_i \in A$ is the i^{th} suffix. The elements within the set A are organized into sub-lists based on the similarity of the suffix. These sub-lists are represented as $D_j = [d_1, d_2, ..., d_F]$ such that $d_i \in A$ and D_j is the j^{th} sub-list. Then, a dictionary is created from the similar groups, which is as D = $\{G_1 : D_1, G_2 : D_2, ..., G_N : D_N\}$ where G_i is the i^{th} group name and D_i is its corresponding list of similar suffixes. Then we iterate each verb of the sentence and

determine whether it is found in the dictionary or not. If found the suffix of the verb is replaced with a similar suffix from the dictionary.

For the Homophones error, the website [16] was scraped for the homophones and missing homophones were added manually. A dictionary is created for the homophones such as H = $\{H_1 : P_1, H_2 : P_2, ..., H_i : P_i\}$ where h_j is the j^{th} word and p_j is its respective homophone. To generate the error, we iterate through each word W_i in a sentence S, and if the word is found in the homophones' dictionary key, the word is replaced by its respective value. This creates an erroneous version of the correct sentence.

In order to generate punctuation error, we go through each character C_i of the sentence S, and if a punctuation symbol is found. An error is generated with a random probability. In case of full stop( ), question mark(?) and exclamation mark(!), they are either removed or replaced with the one which has not occurred. In case of other symbols, they are removed with the random probability. Alternatively, we induce errors in sentence structure by randomly swapping the positions of two words within a sentence with a random probability.

In order to generate error for the missing subject and the missing verb, similar approach is used. The use of POS taggers were helpful in generating corresponding POS tags for each word W_i in the sentence S, denoted as $S_t = \{pt_1, pt_2, .. , pt_n\}$ for n words in S. Then, we iterate through the tag set S_t and if the POS tag indicates a pronoun, then the pronoun is removed to generate pronoun missing error. In the POS tagging process, auxiliary verbs and main verbs aren't distinguished. To determine the main and auxiliary verbs in a sentence, a list of verbs within the sentence is extracted. The final verb that completes the sentence structure is considered the main verb, while the preceding verbs are regarded as auxiliary verbs based on specific rules crafted for this purpose. So by removing the auxiliary verb, auxiliary verb missing error is generated and by removing the main verb, main verb missing error is generated.

4.3 Corpus Statistics

The developed Nepali GEC corpus comprises seven distinct types of errors The verb inflection errors are found to be the most frequent(39.39%) and the Pronouns error are found to be the least frequent (3.89%). The reason for verb inflection error to be most frequent is that, it also refers to the errors related in the subject-verb agreements, errors related to numbers and some other errors where some words might end up wrong with the verb inflected. The whole error inflection statistic can be summarized by the table as follows:

Error Types	**Number of Instances**
Verb Inflection	3202676
Pronouns	316393
Sentence Structure	1001038
Auxiliary Verb Missing	1031388
Main Verb Missing	1031388
Punctuations Errors	1044203
Homophones Errors	503524
Total Errors = 8130496	

Table 4.1: Statistics of the Nepali GEC Corpus.

The amount of different error types is justified as none of them were introduced manually as of now. All the instances have been crafted automatically based on the underlying corpus and predefined suffixes which are carefully extracted. Moreover, error related to word choice is the least common in the corpus which is logical considering the fact that the Nepali language has a relatively small number of homonyms, On the other hand, the most prominent error type in the corpus is related to verb inflection which is not surprising given the fact that the Nepali language has a wide range of verb inflection suffixes and also due to the fact that the inflection covers a wide range of errors.

4.4 Model

For Nepali grammar error correction, two distinct pre-trained from the same BERT architecture were used i.e. MuRIL and NepBERTa. MuRIL is a BERT base model pre-trained on 17 languages with their transliterated counterparts which also includes Nepali[17]. Similarly, NepBERTa is also a BERT based model trained on about 800M Nepali words.[18] The models being pre-trained on Nepali languages thus understand the intricacies that lie within the Nepali language and thus was fine-tuned to perform the task of GED by using the Bert For Sequence Classification. Similarly, the Bert For Masked Language Modeling versions of

this models are used to generate the suggestions of corrected grammar sentence.

MuRIL is available in two versions i.e. base and large. The base version was selected for the task as the model gets larger, the need of resources increases. So, the base version was best fit for the limited resources. The total number of parameters for the Sequence Classification Model is 237,557,762 and for the Masked Language Modeling model is 237,755,045. The length of the Tokenizer is 197,285.

In case of the NepBERTa, the total number of parameters for the Sequence Classification model is 109,483,778 and for the Masked Language Modeling model is 109,514,298. The length of the Tokenizer is 30,523.

4.5 Model Training and Tuning

Before starting the training process, the data preparation is done. The total number sentence pairs from the above statistics was found to be 8,130,496. This sentence pair was split between training and validation data where 95% of the entire dataset was used to construct the training dataset i.e. 7,723,971 and 5% of the entire dataset was used to construct the test dataset i.e. 406,525. As the dataset is in the form of pairs i.e. of correct sentences and incorrect sentences, "label 0" was given to correct sentences while "label 1" was given to incorrect sentences. The following table shows the characteristic of the training and validation dataset.

Split	**Number of Correct Sentences**	**Number of Incorrect Sentences**	**Total Sentences**
Train	2,568,682	7,514,122	10,082,804
Valid	365,606	405,905	771,511

Table 4.2: Description of Dataset

The training data is used to fine tune the model for GED task and valid dataset is used to evaluate the fine-tuned model to see how well does the model actually work.

The models(MuRIL and NepBERTa) being already pre-trained on Nepali text understand the intricacies of Nepali language and thus the burden of pre-training on large unlabelled corpus is relieved hence saving the time required for tuning the models. So, the pre-trained models is fine-tuned for a downstream task i.e. Sequence Classification for the GED task. For the GEC task, MLM model of the same pre-trained models are utilized following the GED task.

At first, to train the model for the Sequence Classification task, the MuRIL model was loaded from the HuggingFace hub using the HuggingFace transformers library. The tokenizer

was also loaded along with the model. As the task of GED is sensitive to case folding, the tokenizer is initialized such that the tokens aren't case folded (*do_lowercase = False*) and are used as it is. Then, the dataset is tokenized using the tokenizer and Dataset Dictionary was created for both training and validation datasets. The Dictionary and the model were then loaded to the GPU and the training process was initialized. The following were the hyper-parameters while training the model:

- Epoch = 1
- Train Batch Size = 256
- Valid Batch Size = 256
- Loss Function = Cross Entropy Loss
- Optimizer = AdamW
- Optimizer Parameters:
  - Learning Rate = $5e^{-5}$
  - $\beta_1 = 0.9$
  - $\beta_2 = 0.999$
  - $\epsilon = 1e^{-8}$

The same approach as explained above is used for the NepBERTa model while the number of training epoch was increased to 2.

4.6 GEC Engine

The GEC Engine is made up of the GED model and the MLM model. The GED model is the heart of the GEC engine which labels the input sentence as correct or incorrect. When an input is passed to the engine, it first undergoes through the GED model. If the input sentence is classified as correct, the user is notified that the sentence is correct and no correction is necessary. In case of incorrect, a set of masked sentences are created from the input sentences. The masked sentences are created by masking each word of the input sentence and masking each space between the words. After the masked sentences are generated, those sentences are passed to the MLM model which is either MuRIL or NepBERTa. The MLM model generates the masked token which replaces the [MASK] in the mask sentence. The sentence is then passed to the GED model, which labels the sentence. If the sentence is labelled as incorrect, the sentence is rejected while if the sentence is labelled as correct, it is passed as a suggestion to the user.

Here is an example of generated masked sentences:

Input sentence	यो एउटा उदाहरण हो ।
Masked Sentences	[MASK] एउटा उदाहरण हो ।
	[MASK] यो एउटा उदाहरण हो ।
	यो [MASK] उदाहरण हो ।
	यो [MASK] एउटा उदाहरण हो ।
	यो एउटा [MASK] हो ।
	यो एउटा [MASK] उदाहरण हो ।
	यो एउटा उदाहरण [MASK] ।
	यो एउटा उदाहरण [MASK] हो ।
	यो एउटा उदाहरण हो [MASK] यो एउटा उदाहरण हो [MASK] ।
	यो एउटा उदाहरण हो [MASK] ।

Table 4.3: Example of Masked Sentences

4.7 Tools and Libraries

The development of the Nepali GEC system involves the use of various tools and technologies which are listed as follows:

1. Programming Language

   Python has a rich ecosystem of libraries and frameworks supporting a wide range of Machine Learning tasks involving the NLP tasks leading to the suitability use in the development of the Nepali GEC system.

2. Backend

   The backend is implemented in Django. Django is a high-level web framework written in Python that encourages rapid development and clean, pragmatic design. It follows the MVC architectural pattern, emphasizing the concept of reusability and "pluggability" of components. Django's primary goal is to simplify the creation of complex, database-driven websites by providing built-in tools and features for common tasks. One of the key features of Django is its ORM system, which abstracts database interactions and allows developers to define data models using Python classes. This abstraction enables developers to work with databases using familiar Python syntax, without needing to write raw SQL queries. Django also includes a powerful template engine that facilitates the separation of presentation and logic in web applications. Templates in Django are HTML files with embedded Python code, allowing for dynamic content generation based on data passed from views. Routing and handling

HTTP requests is handled through Django's URL dispatcher, which maps URL patterns to corresponding views. Views are Python functions or classes that process incoming requests and return HTTP responses, typically by rendering templates or serializing data. Additionally, Django provides built-in support for user authentication, session management, form handling, and security features such as protection against common web vulnerabilities like XSS and CSRF.

3. Transformers Library

   The Transformers library, developed by Hugging Face, is a comprehensive toolkit for NLP tasks. It offers easy access to state-of-the-art pre-trained models like BERT and GPT, along with tools for fine-tuning and deploying these models for various NLP tasks. MuRIL and NepBERTa were also loaded and fine-tuned for the GED task using the transformer library. It helped loading and saving the pre-trained and fine-tuned models easily thus helping for training and inference.

4. PyTorch

   PyTorch was used to build and train our neural network model for Nepali GEC. These frameworks provide efficient implementations of neural network layers, optimization algorithms, and other essential components required for model training also utilizing the GPU resources for way faster computation than what a normal CPU could do. In this project, PyTorch was used for training and fine-tuning our GEC models, handling the data pipeline, and optimizing the model's performance.

5. Development Environments

   IDEs like PyCharm, Jupyter Notebook, or Visual Studio Code provide a user-friendly coding environment with features such as code editing, debugging, and execution. These IDEs can enhance productivity during the development of the GEC system. For this project, Visual Studio Code with Jupyter Notebook was used to build and develop our model and the PyCharm was used to create the API endpoints for the GEC model to interact with the frontend. NVIDIA® CUDA® version 11.8 was used to provide the required parallel computing platform to train such large models on NVIDIA® GeForce® RTX™ 4060. Torch version 2.2.0 (CUDA 11.8 as compute platform) was also installed using pip package manager to load the parameters and train the model on the GPU.

   Besides that Google Collab and Kaggle Notebooks were also used for testing and debugging purposes alongside locally before actually leaving the device to train on the large dataset. Google Collab and Kaggle Notebooks were also used to test the fine-tuned model on new data to observe the performance.

6. Version Control Systems

   Version control systems like Git can be employed to manage code repositories and collaborate with team members. They allow for tracking changes, branching, and merging of code, ensuring a systematic and organized development process. Because of our familiarity with git, we will be using git to maintain the code repositories and collaborate. Github was also used as a platform to push our progress remotely and make changes locally as well as remotely.

4.8 Performance Metrics

For this project, the performance metrics might be the points listed below:

1. Accuracy

   Accuracy represents the proportion of correctly classified instances over the total number of instances. Calculated by dividing the number of correctly classified instances by the total number of instances and multiplying the result by 100 to express it as a percentage.

2. Processing Time

   Calculate how long it takes the system to examine and fix grammar mistakes in a given input. It shows how quickly and effectively the system responds.

5. System design

5.1 System Context Diagram

The diagram illustrates the system context diagram of an application that allows users to correct Nepali text. The system works as follows:

A user begins by entering the Nepali text they wish to have corrected into the application's interface. This could be a form of written Nepali content containing errors or issues the user wants to fix. After entering the text, the user submits it to the application's backend processing system. This triggers the grammar correction model to analyze the submitted text. It identifies and corrects any grammatical mistakes, misspellings, incorrect word usage, or other language issues present. Once the model finishes correcting the text, the application sends the revised, error-free version back to the user interface for display. The corrected Nepali content is now available for the user to review. Additionally, the application provides an option for the user to submit feedback on the quality and accuracy of the corrections made by the system. This feedback loop allows the developers to continuously enhance the performance of the underlying grammar correction algorithm over time.

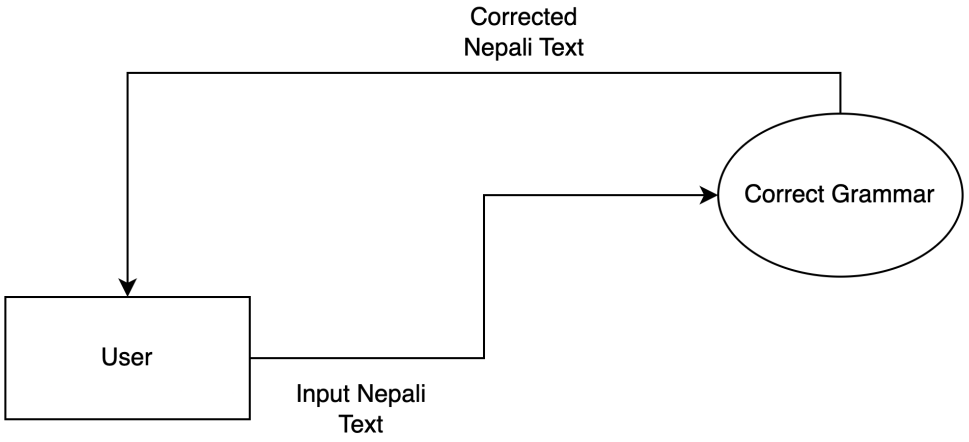

Figure 5.1: System Context diagram

5.2 Data Flow Diagram

This diagram illustrates the level 1 DFD of an application that allows user to correct Nepali text. A user begins by entering the Nepali text they wish to have corrected into the interface of application. The input text can be any form of written Nepali content. After entering the text, the user then submits the text to the Grammatical Error Detection Model. The model then detects if the text is correct or incorrect. If the text is correct, then it sends the input text back to user. If the input text is incorrect then it sends the incorrect text to grammatical error correction engine. The grammatical error correction engine then generates the correct sentence and returns it to the user.

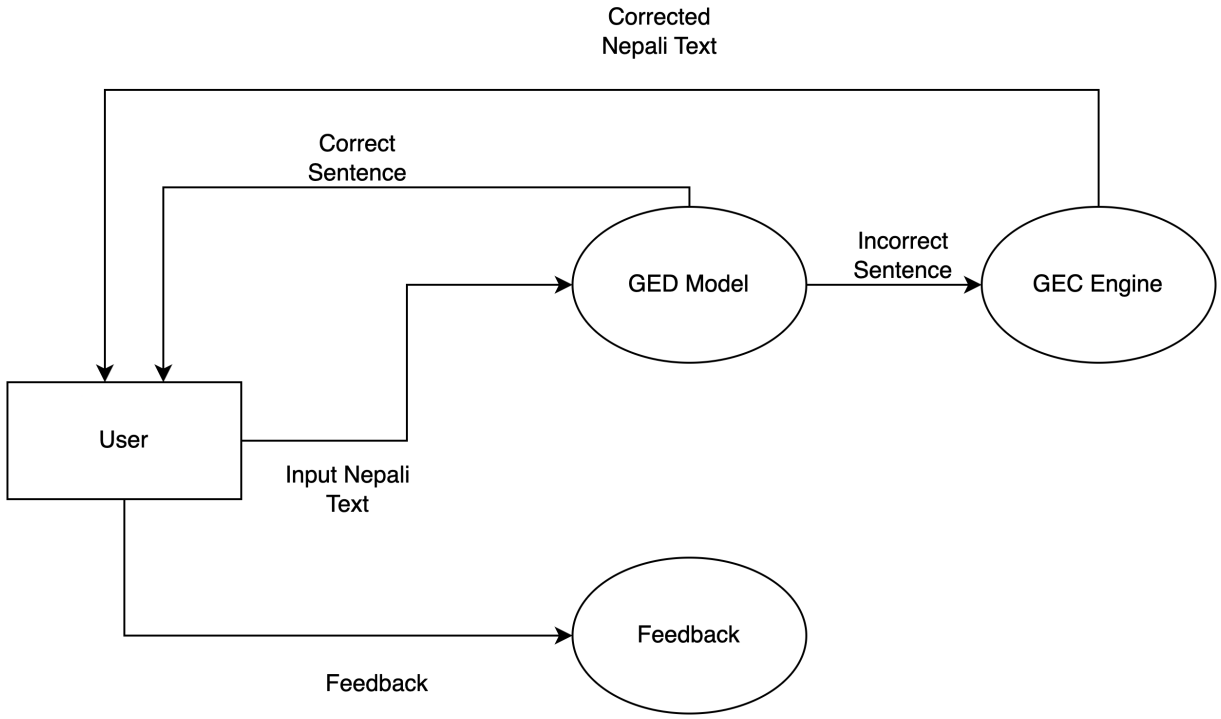

Figure 5.2: Data Flow Diagram diagram

5.3 Use Case Diagram

The use case diagram shows the different ways that a user can interact with a Nepali grammar correction system. The system allows users to input Nepali text, correct the grammar of the text, and provide feedback on the correction.

The main use cases of the system are as follows:

1. Input Nepali text: The user can input Nepali text into the system by typing it in or by pasting it from another document.

2. Correct grammar: The system can correct the grammar of the Nepali text. The system uses a grammar correction model to identify and correct grammatical errors in the text.

3. Provide feedback: The user can provide feedback on the grammar correction that the system has made. The user can indicate whether the correction is correct or incorrect, and they can also provide suggestions for how the correction could be improved.

The use case diagram also shows the different actors that interact with the system. The main actors are the user and the grammar correction model. The user is the person who inputs the Nepali text and provides feedback on the correction. The grammar correction model is a computer program that identifies and corrects grammatical errors in Nepali text.

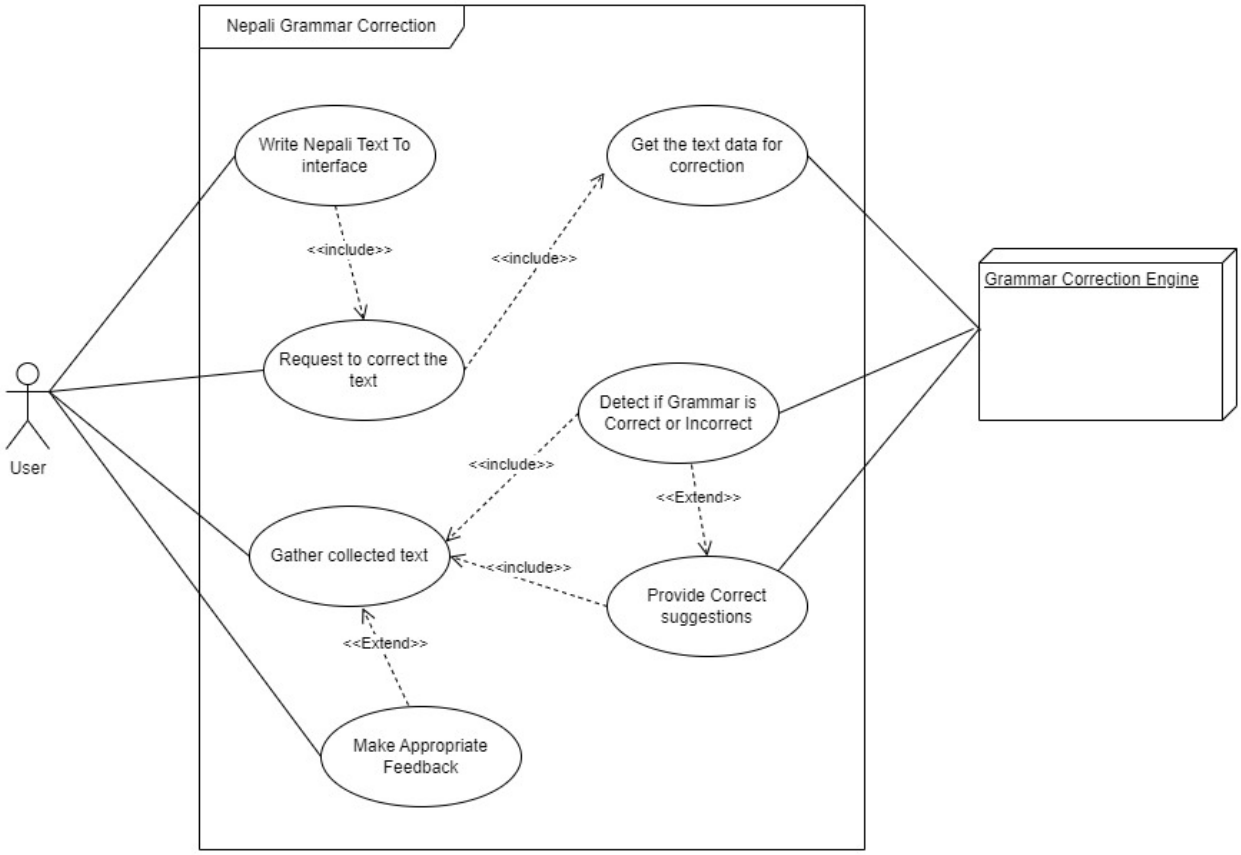

Figure 5.3: Use case diagram

5.4 System Sequence Diagram

The system sequence diagram portrays the flow of interactions within a web-based application designed to facilitate the correction of Nepali text. It begins with the user's initiation of the process by inputting the Nepali text they desire to correct. Following this, upon submission of the text by the user, the system undertakes the correction process. This correction phase involves identifying and rectifying any grammatical or spelling errors within the text. Once the correction is complete, the corrected sentence is transmitted to the front end of the application. In the front end, the corrected sentence is formatted and presented within the user interface, allowing the user to view the revised text seamlessly. This process ensures that users can easily input Nepali text, have it corrected efficiently, and promptly view the corrected version through the application's interface, enhancing the overall user experience and aiding in language accuracy.

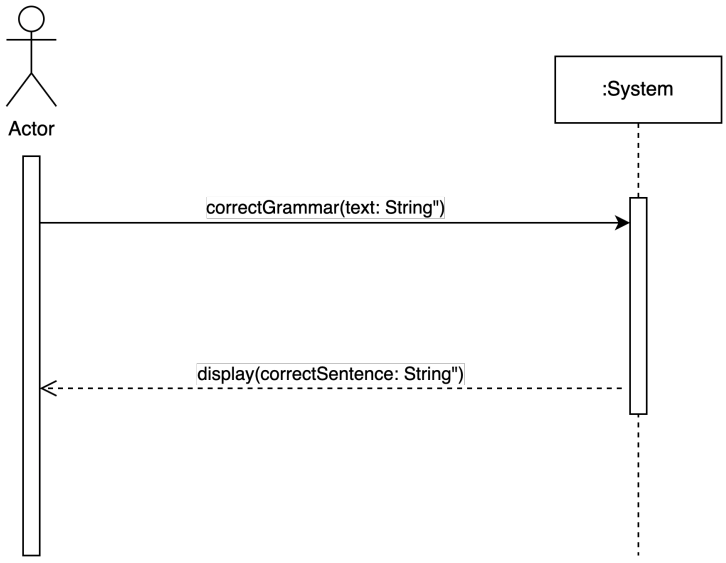

Figure 5.4: System Sequence Diagram

5.5 Activity Diagram

This diagram depicts the activities within our system. The user submits the sentence s/he wishes to have corrected. The system then performs a correction operation in the provided sentence by the user. Finally, the system sends the corrected text to the user interface.

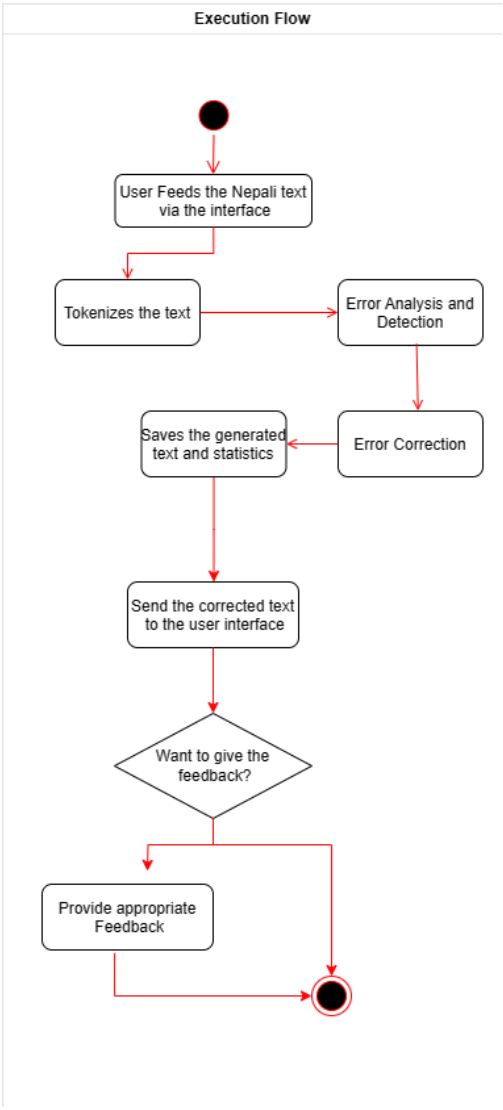

Figure 5.5: Activity Diagram

6. Results & Discussion

As discussed in the methodology, the first approach was adopted which was training a vanilla transformer model for the GEC task. So, the model was initially trained on 10,000 pairs of correct and incorrect sentences randomly sampled from the corpus. The model was over-fitted for those sentence pairs which should always be the first task needed to be done for a language modeling task. It generated output on the same training set with a considerable tolerance on similar types of errors encountered on it. So we decided to further work on this idea by letting the vanilla transformer model train on 100,000 pairs of correct and erroneous sentences sampled from the corpus. The total parameters of the model was 63,574,887 which is well suited as per the compute resources available. The model was left to train for about 2 days on the 100,000 pairs of sentences for 27 epochs. The training and validation loss is visualized below:

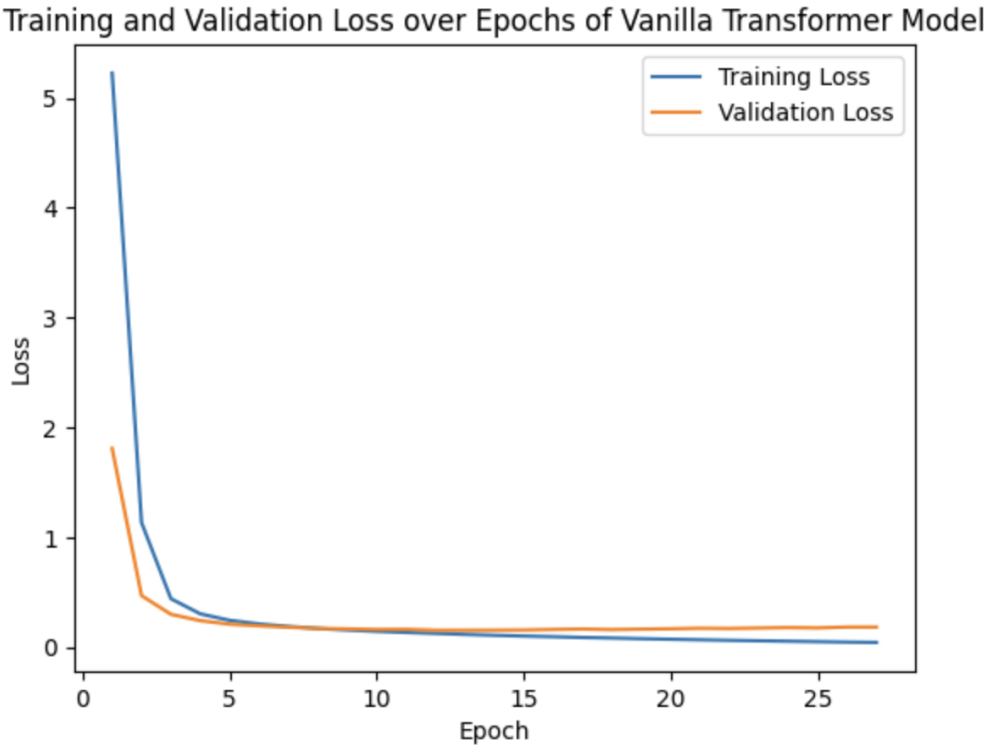

Figure 6.1: Performance of Vanilla Transformer over training and validation data

But to an utter disappointment, the model performed poorly even on training set and the performance was even worse on the validation set. To effectively learn the nuances of the Nepali grammar, the model of this size does not seem to be capable and a much larger

model will be required to effectively model such nuances for better GEC tasks. But due to the limitation of enough compute resource available to train such massive model and also on the humongous corpora we generated, the idea of letting the vanilla transformer model train on the entire corpus was dropped for now and the alternative approach of using the pre-trained models and fine-tuning for our custom dataset was adopted.

The two models i.e. MuRIL and NepBERTa were then successfully fine-tuned for 2,568,682 correct labeled sentences and 7,514,122 incorrect labeled sentences and the validation of the fine-tuned models is done on 365,606 correct labeled sentences and 405,905 incorrect labeled sentences. This fine-tuning task was done for GED tasks which forms the core of the GEC engine. The training information for these models is tabulated as follows:

Model	**Number of Epochs Trained**	**Number of Trainable Parameters**	**Tokenizer Length**
MuRIL	1	237,557,762	197285
NepBERTa	2	109,514,298	30523

Table 6.1: Training information on models.

Also, the performance shown by MuRIL for our GED task is summarized by the table as follows:

Model	**Training Loss**	**Validation Loss**	**Accuracy**
MuRIL	0.242700	0.217756	0.911515

Table 6.2: Performance of MuRIL.

The performance shown by NepBERTa for our GED task is summarized in the table as follows:

Epoch	**Training Loss**	**Validation Loss**	**Accuracy**
1	0.339700	0.385597	0.794744
2	0.277600	0.344654	0.817336

Table 6.3: Performance of NepBERTa.

Despite being trained only for one epoch, MuRIL seemed to perform much better than NepBERTa being trained for two epochs for GED tasks. This is because the complexity

of MuRIL was much higher than that of NepBERTa as shown by the number of trainable parameters in both the models. Thus MuRIL is effectively seen to be able to learn about the patterns involved in correct and erroneous sentences. Thus for the GED task, which forms the core of our GEC engine, we decided to use the fine-tuned MuRIL model. For predicting the `[MASK]`, either of the two pre-trained model can be used which then gives us possible sentences to be suggested after they have been checked by the GED model for correctness.

An example of the model outputs are shown below:

Input sentence	नयाँ संविधान कार्यान्वयनको लागि निश्चित समयसीमाभित्रै तीन तहको निर्वाचन गर् ।
Baseline Output	नयाँ संविधान कार्यान्वयनको लागि निश्चित समयसीमाभित्रै तीन तहको निर्वाचन गर्नुपर्नेछ ।
NepBERTa as MLM Output	नयाँ संविधान कार्यान्वयनको लागि निश्चित समयसीमाभित्रै तीन तहको निर्वाचन हुनुपर्छ । नयाँ संविधान कार्यान्वयनको लागि निश्चित समयसीमाभित्रै तीन तहको निर्वाचन गर्यौं ।
MuRIL as MLM Output	नयाँ संविधान कार्यान्वयनको लागि निश्चित समयसीमाभित्रै तीन तहको निर्वाचन हुनेछ । नयाँ संविधान कार्यान्वयनको लागि निश्चित समयसीमाभित्रै तीन तहको निर्वाचन गर्यौं ।

Table 6.4: Example Result

The processing time of the sentences depends upon the length of the sentence. Upon visualizing the time taken for the system to generate the suggestions, it was found that the time taken by the system to generate suggestions is linearly dependent on the sentence length. From the corpus, 10 sentences were randomly selected for each sentence length and the average processing time was calculated for each length. Both the NepBERTa and MuRIL were evaluated on this metric which is shown in the figure below:

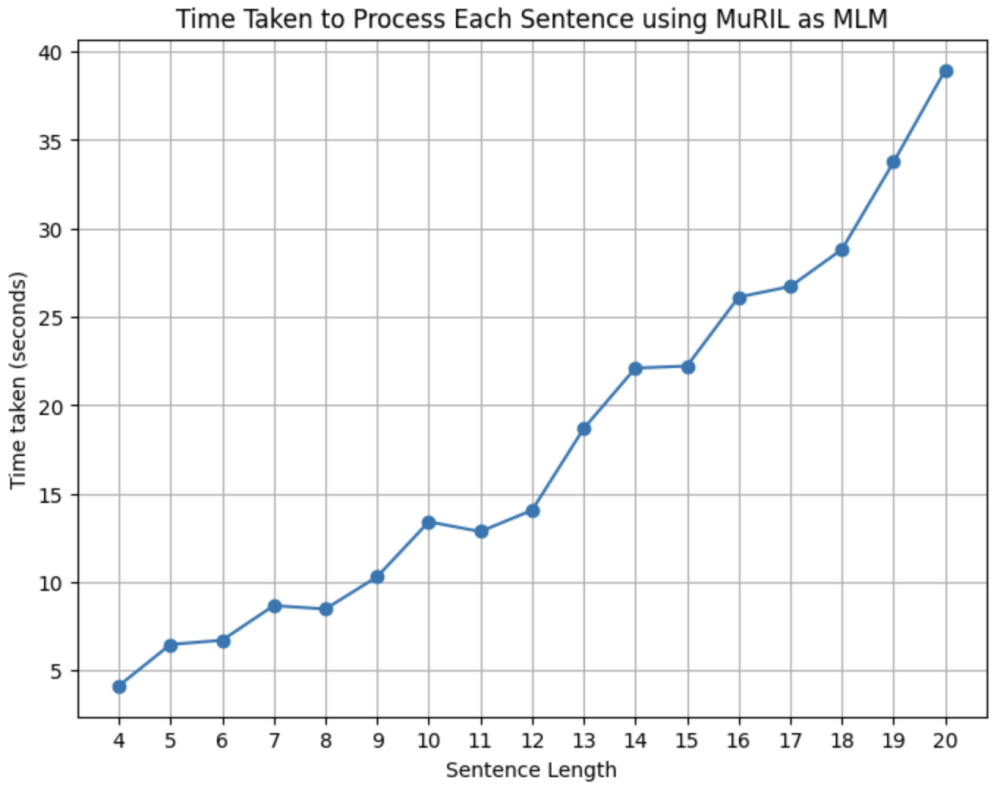

Figure 6.2: Processing time for MuRIL as MLM

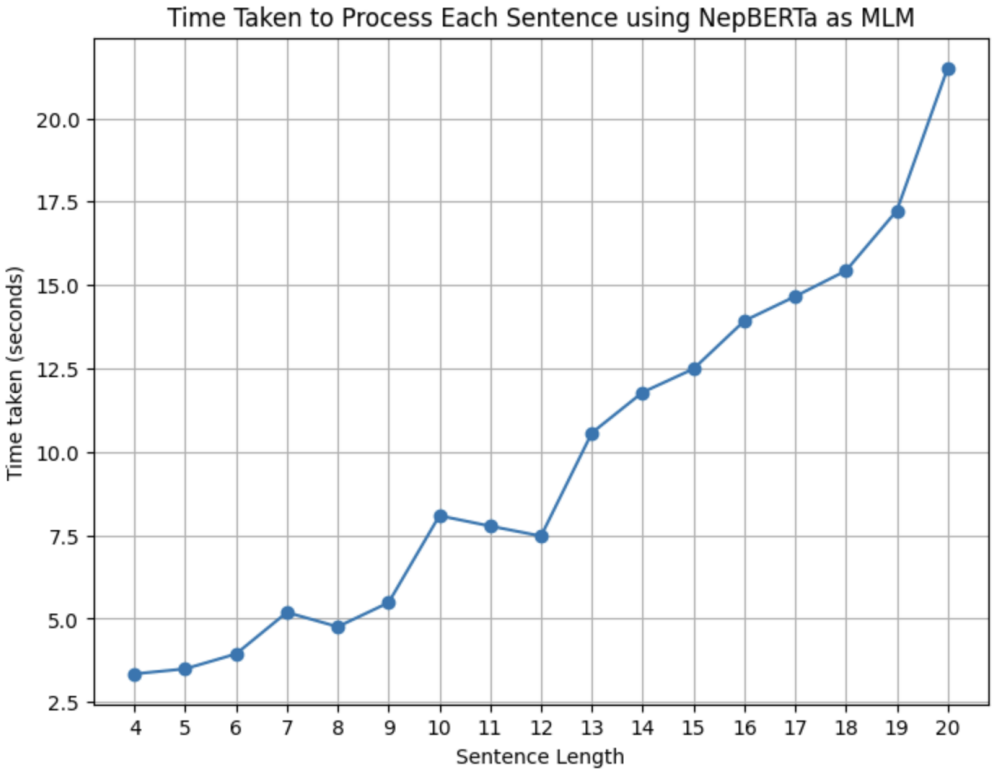

Figure 6.3: Processing time for NepBERTa as MLM

The slight variations can be seen as the sentence undergoes the tokenization process. In the tokenization process, a single word can be decomposed into multiple tokens, so more masked sentences are generated which increases the number of sentences to be processed by the BERT MLM model hence fluctuating the processing time.

7. Conclusion

This project is aimed at automating the Nepali GEC tasks which serves as an invaluable tool for individuals and organizations to improve the quality and accuracy of their written Nepali text sentence by sentence. The system aims to detect grammatical errors and also generate possible suggestions and corrections to rectify those issues by leveraging the usage and the nature of the pre-trained LLMs in suggesting such correct sentences.

Chapter 2 discusses about Nepali language being a low-resource language. In Nepali, sentences follow the Subject-Object-Verb (SOV) order, reflecting the language's agglutinating nature. Research and development in Nepali Grammar Error Correction remain sparse, with only a few systems focusing on spelling checking rather than comprehensive grammar correction. We hope that our work on GEC acts as the stepping stone for more and more research projects in the field of Nepali GEC and in Nepali NLP as a whole.

The BERT Architecture forms the core of our project Ecosystem as it is used for both GED and GEC tasks. The BERT model for sequence classification makes use of a Dense layer at the end which helps in performing the GED-related task. This GED model serves a huge purpose along with the MLM of BERT as MLM for BERT helps generate suggestions by predicting the `[MASK]` token and GED helps in identifying new suggestions as grammatically correct or not.

Chapter 3 explains the feasibility of the project and also about how the system is built along with the corpus creation. The corpus includes errors limited to seven different types of grammatical errors including verb inflections, punctuation, homonyms, sentence structure, and sentence fragment errors. It also includes the possible approaches to go about building the GEC system which include training the revolutionary transformer model from scratch or making use of the existing pre-trained BERT models to train for GED tasks and use it for the GEC engine.

Chapter 4 talks about the Experimental setup for the construction of the project which included configuring and updating the CUDA® for the NVIDIA® GPU whilst installing the PyTorch framework for deep learning tasks. Other than that other libraries like transformers are used for the project. It also includes information about the corpus statistics which gives about the data distribution to be used throughout training and validation of the models. Two models MuRIL and NepBERTa were the choices for fine-tuning on GED task and also use their MLM models to come up with the suggestions. Details on the hyper-parameters

are also discussed in the section.

The two models were fine-tuned on the custom dataset with MuRIL performing better than NepBERTa on GED tasks for the project. So it was clear that MuRIL would be used for the GED purpose. For predicting the masking, MLM of either of the models could be used and the Result section shows the suggestion provided by both the models. Finally, the user interface of the project would get the actual suggestions so that s/he would get the idea on how to rectify the error incurred.

8. Limitations and Future Enhancement

8.1 Limitations

1. **Incomplete Representation of Grammatical Errors:** The corpus used for training may not encompass all types of grammatical errors encountered in Nepali text. Errors such as verb inflection, pronoun misuse, sentence structure inconsistencies, missing auxiliary or main verbs, punctuation errors, and homophone errors might be inadequately represented.

2. **Challenges in BERT:** As the Masked Language Modeling of BERT is used for the GEC task, the MLM is only able to predict a single word at a particular instant. So, sentences with more than a single error is quite unlikely to be corrected by the GEC engine. Furthermore, As the GED model is the core of the GEC system. If the GED model classifies the incorrect sentence as correct sentence the GEC system won't be able to provide any helpful suggestions to the user. Also, BERT may struggle with complex linguistic contexts and limited coverage of specialized vocabulary, posing challenges in accurately correcting sentences.

8.2 Future Enhancements

1. **Expansion of Training Corpus:** To enhance the model's effectiveness, efforts should be made to enrich the training corpus with a more comprehensive range of grammatical errors encountered in Nepali text. This includes collecting and annotating diverse examples of errors, covering a wide array of linguistic phenomena.

2. **Enhancement of Nepali GEC System** Despite creating a large parallel corpus for Nepali GEC task, due to lack of adequate training resources, we couldn't take advantage of the huge corpus for training an end to end model for GEC task. Initially, we trained a vanilla transformer model for a small dataset, but the results produced by that model wasn't satisfactory. So, with the availability of large training resources, a more robust and complex model can be trained fully on the curated dataset.

References

[1] Bal Krishna Bal. Structure of nepali grammar. *PAN Localization, Madan Puraskar Pustakalaya, Kathmandu, Nepal*, pages 332–396, 2004.

[2] Sajha. Chinari: Analysis on different aspects of nepali language, 2022.

[3] Government of Nepal. Constitution of nepal, 2021.

[4] Dr. Laxmi Khatiwada. Inflection and derivation in nepali noun, adjective and adverb 1 inflection and derivation in nepali, 06 2013.

[5] Pravesh Koirala and Aman Shakya. A nepali rule based stemmer and its performance on different NLP applications. *CoRR*, abs/2002.09901, 2020.

[6] Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto, and Christopher Bryant. The CoNLL-2014 shared task on grammatical error correction. In Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto, and Christopher Bryant, editors, *Proceedings of the Eighteenth Conference on Computational Natural Language Learning: Shared Task*, pages 1–14, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[7] Christopher Bryant and Hwee Tou Ng. How far are we from fully automatic high quality grammatical error correction? In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 697–707, Beijing, China, July 2015. Association for Computational Linguistics.

[8] Nikolay Banar, Walter Daelemans, and Mike Kestemont. Character-level transformer-based neural machine translation. *CoRR*, abs/2005.11239, 2020.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[10] Ikumi Yamashita, Satoru Katsumata, Masahiro Kaneko, Aizhan Imankulova, and Mamoru Komachi. Cross-lingual transfer learning for grammatical error correction. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages

4704–4715, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics.

[11] Roman Grundkiewicz, Marcin Junczys-Dowmunt, and Kenneth Heafield. Neural grammatical error correction systems with unsupervised pre-training on synthetic data. In *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 252–263, Florence, Italy, August 2019. Association for Computational Linguistics.

[12] Huiyuan Lai, Antonio Toral, and Malvina Nissim. Multilingual pre-training with language and task adaptation for multilingual text style transfer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 262–271, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[15] Rabindra Lamsal. A large scale nepali text corpus, 2020.

[16] Bhupendra BC. Homophones in nepali language. https://bhupendrabc.blogspot.com/2020/11/blog-post_25.html, 2020.

[17] Simran Khanuja, Diksha Bansal, Sarvesh Mehtani, Savya Khosla, Atreyee Dey, Balaji Gopalan, Dilip Kumar Margam, Pooja Aggarwal, Rajiv Teja Nagipogu, Shachi Dave, Shruti Gupta, Subhash Chandra Bose Gali, Vish Subramanian, and Partha P. Talukdar. Muril: Multilingual representations for indian languages. *CoRR*, abs/2103.10730, 2021.

[18] Sulav Timilsina, Milan Gautam, and Binod Bhattarai. NepBERTa: Nepali language model trained in a large corpus. In Yulan He, Heng Ji, Sujian Li, Yang Liu, and Chua-Hui Chang, editors, *Proceedings of the 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 273–284, Online only, November 2022. Association for Computational Linguistics.

Appendix

Appendix A: Code Snippets

```python
def check_GE(sents):
    """Check if the input sentences have grammatical errors

    :param sents: list of sentences
    :return: predictions, probabilities
    :rtype: (numpy.ndarray, numpy.ndarray)
    """

    predictions_list = []
    probabilities_values_list = []

    for sent in sents:
        # Tokenize Sentences
        inputs = ged_tokenizer(sent, return_tensors="pt")

        with torch.no_grad():
            # Get Logits
            logits = ged_model(**inputs)[0]

        # Get Probability
        probability_value = torch.softmax(logits, dim=1).cpu().numpy()

        # Get Predictions
        prediction = np.argmax(logits.cpu().numpy(), axis=1)

        predictions_list.append(prediction)
        probabilities_values_list.append(probability_value)

    return np.concatenate(predictions_list), np.concatenate(probabilities_values_list)
```

Figure 8.1: Code Snippet For GED

```
def create_mask_set(sentence):
  """For each input sentence create 2 sentences
     (1) [MASK] each word
     (2) [MASK] for each space between words
  """

  sentences = []

  sent = lm_tokenizer.tokenize(sentence)

  for i in range(len(sent)):
    # (1) [MASK] each word
    new_sent = sent[:]
    new_sent[i] = '[MASK]'
    text = " ".join(new_sent).replace(" ##","")
    sentences.append(text)

    # (2) [MASK] for each space between words
    new_sent = sent[:]
    new_sent.insert(i, '[MASK]')
    text = " ".join(new_sent).replace(" ##","")
    sentences.append(text)

  return sentences
```

Figure 8.2: Code Snippet For Creating Masked Sentences

```python
def correct_grammar(org_sentence, sentences):
  """
  Correct the grammar of the input sentences.
  :param org_sentence: Original sentence to check grammar for
  :param sentences: List of MASKed sentences of the original sentence
  :return: List of grammar-checked sentences
  """

  prediction, probability = check_GE([org_sentence])
  if prediction == 0:
    return ["The sentence is correct"]
  else:

    new_sentences = []

    for sentence in sentences:
      tokenized_input = lm_tokenizer.encode(sentence, return_tensors="pt")

      if lm_tokenizer.mask_token_id not in tokenized_input:
        continue

      mask_index = torch.where(tokenized_input == lm_tokenizer.mask_token_id)[1].tolist()[0]

      with torch.no_grad():
        outputs = model(tokenized_input)
        predictions = outputs[0]

      predicted_index = torch.argmax(predictions[0, mask_index]).item()
      predicted_token = lm_tokenizer.convert_ids_to_tokens([predicted_index])[0]

      tokenized_input[0][mask_index] = predicted_index

      generated_sentence = lm_tokenizer.decode(tokenized_input[0], skip_special_tokens=True)

      prediction, probability = check_GE([generated_sentence])

      exps = [np.exp(i) for i in probability[0]]
      sum_of_exps = sum(exps)
      softmax = [j / sum_of_exps for j in exps]

      if not prediction and softmax[0] > 0.55:
        new_sentences.append(generated_sentence)

      if not new_sentences:
        return ["No Correction Found"]

    return new_sentences
```

Figure 8.3: Code Snippet For GEC

Appendix B: Screenshots of Outputs

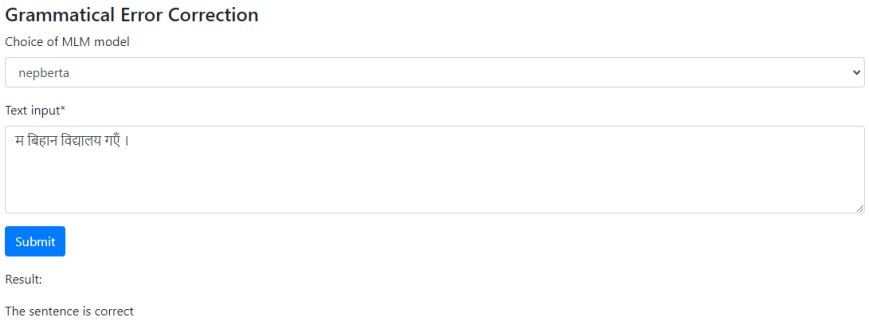

Figure 8.4: Output of GED for Correct Sentence

Figure 8.5: Output of GED for Inccorrect Sentence

Figure 8.6: Output of GEC Using MuRIL as MLM Model

Figure 8.7: Output of GEC Using NepBERTa as MLM Model

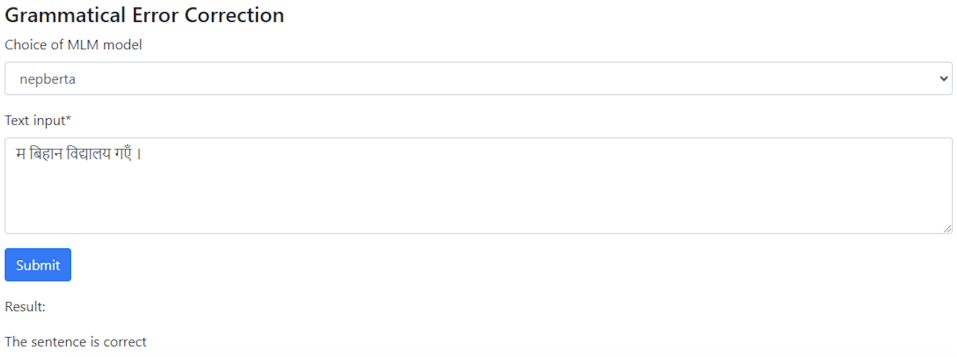

Figure 8.8: Output of GEC When Grammatically Correct Sentence As Input

Figure 8.9: Output of GEC When Model Doesn't Generate Any Suggestion