Article

Not peer-reviewed version

Sentence Classification Using Transfer Learning with BERT

Abhishek Verma * and Nallarasan V

Posted Date: 29 May 2025

doi: 10.20944/preprints202505.2360.v1

Keywords: BERT; Transfer Learning; Classification; Fine-Tuning; Transformers

 doi:10.20944/preprints202505.2360.v1

Article

Sentence Classification Using Transfer Learning with BERT

Abhishek Verma * and Nallarasan V

Department of Information Technology, SRMIST, Chennai, Tamil Nadu,Chennai, India; nallarav@srmist.edu.in
* Correspondence: av6651@srmist.edu.in

Abstract: The rise of machine learning is enhancing various products in the software industry in a variety of ways by applying a diverse set of algorithms. One such application is to verify whether a sentence is grammatically correct or not. We find the use of such applications in Writing Assistance tools, Language Learning platforms, Automated Scoring platforms, Content Moderation platforms, and Email platforms. In our research, we present Sentence Classification using Transfer Learning with BERT (Bidirectional Encoder Representations from Transformers). We leverage transfer learning with the Hugging Face Transformers library to fine-tune the BERT model, incorporating hyperparameter sweeps via Weights and Biases to optimize learning rate, batch size, and the number of epochs. The process involves downloading and preprocessing the dataset, tokenizing sentences with the BERT tokenizer, and preparing input data with special tokens, padding, and attention masks. The dataset is split into training and validation sets, and a custom training loop is implemented to fine-tune the model, logging performance metrics to Weights and Biases. Additionally, a test function is provided to predict the grammaticality of example sentences using the fine-tuned model.

Keywords: BERT; transfer learning; classification; fine-tuning; transformers

1. Introduction

The growth of artificial intelligence is fueled by several factors, including abundant data, transformer models, and high-performance computing powered by GPUs. These technological advancements have led to the creation of powerful models, particularly BERT (Bidirectional Encoder Representations from Transformers). After the discovery of the transformer model, the artificial intelligence race has accelerated. The exceptional performance of artificial intelligence models (e.g., BERT) and their access through user-friendly interfaces have brought classification to the forefront of daily and commonplace use. Techniques like classification are involved in building application features like Document Classification. Now, artificial intelligence is transforming the world, driving innovations in a wide range of industries and emerging applications.

The field of natural language processing has significantly advanced with the adoption of BERT-based models for grammatical error correction (GEC), offering robust solutions for improving text quality across various applications. These models harness BERT's contextual embeddings to accurately detect and correct grammatical errors, enhancing the clarity of written communication. For instance, [1] demonstrates how fine-tuning BERT within an encoder-decoder framework improves GEC performance, making it suitable for applications requiring precise text corrections. The GECToR model, as described in [2], utilizes BERT to predict token-level edit operations, enabling efficient grammar corrections in diverse contexts. Similarly, [3] introduces Seq2Edits, which leverages BERT for span-level edit predictions, addressing complex errors in professional and creative writing. The iterative editing approach in [4] uses BERT to refine text through multiple passes, ideal for high-stakes writing tasks. Additionally, [5] employs BERT with synthetic data to enhance GEC robustness, broadening its applicability across varied linguistic scenarios.

 doi:10.20944/preprints202505.2360.v1

2. Identifying the Research Gap and Contribution

Despite the progress in BERT-based GEC, a notable gap remains in deploying these models in resource-constrained, real-world settings outside of controlled research environments [1]. Existing studies often prioritize model accuracy over practical challenges like computational efficiency and adaptability to diverse text domains. Our research aims to address this in future by developing a BERT-based GEC system optimized for general-purpose applications, utilizing Parameter-Efficient Fine-Tuning (PEFT) techniques such as LoRA and Prompt Tuning to minimize computational overhead. We incorporate zero-shot and few-shot inference strategies to ensure flexibility across different writing styles and genres. Model performance is evaluated using ROUGE and error-type classification metrics, ensuring reliable corrections. This work bridges the divide between theoretical GEC advancements and practical, scalable solutions, enabling broader adoption in professional, creative, and technical writing contexts.

3. Preliminaries of Classification Models for Text Processing

Classification models for text processing aim to categorize or label textual data, enabling tasks such as grammatical error correction (GEC), sentiment analysis, and named entity recognition. These models learn to map input text to discrete outputs, using contextual or sequential patterns in the data. A variety of classification models, including transformer-based models like BERT and non-transformer approaches like LSTMs and CNNs, have been applied to text processing challenges. In this section, we present the preliminaries of these models, as illustrated in Figure 1.

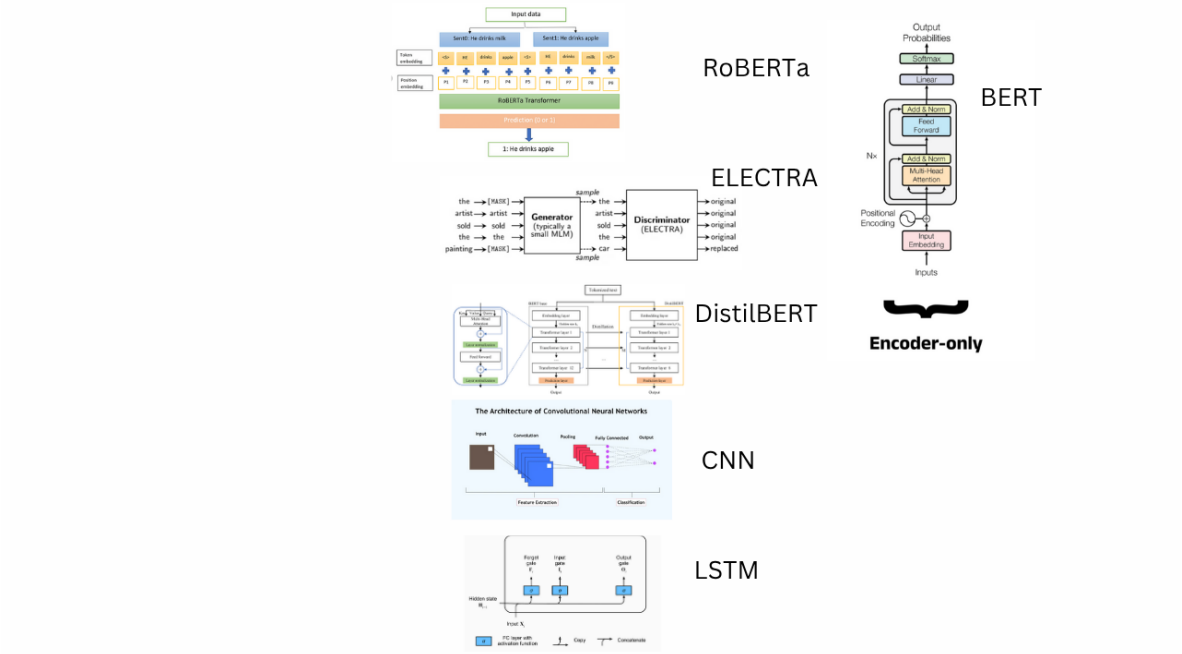

Figure 1. Overview of Classification Models for Text Processing.

3.1. BERT (Bidirectional Encoder Representations from Transformers)

BERT is a transformer-based model that leverages bidirectional context to understand text, pre-trained on large corpora using masked language modeling and next-sentence prediction. It consists of stacked encoder layers that process input text in parallel, capturing deep contextual relationships. Fine-tuning BERT for classification tasks, such as GEC, involves adding a task-specific layer to predict labels or edit operations. Its bidirectional nature allows it to excel in tasks requiring nuanced understanding of text, making it highly effective for applications like error detection and correction.

 doi:10.20944/preprints202505.2360.v1

3.2. RoBERTa (Robustly Optimized BERT Approach)

RoBERTa builds on BERT by optimizing its pre-training process, using larger datasets, longer training times, and removing the next-sentence prediction objective. It employs dynamic masking and larger batch sizes to improve contextual representations. For classification tasks, RoBERTa's enhanced embeddings lead to superior performance in applications like GEC and sentiment analysis, offering robustness in handling diverse text inputs. Its architecture remains similar to BERT but delivers improved accuracy due to refined pre-training strategies [6].

3.3. DistilBERT

DistilBERT is a distilled version of BERT, designed to be smaller and faster while retaining most of BERT's performance. It reduces parameters by 40 percent through knowledge distillation, where a smaller model is trained to mimic BERT's outputs. This makes DistilBERT ideal for resource-constrained environments, such as real-time text classification or GEC on mobile devices. Despite its reduced size, it maintains strong performance in tasks like error correction and sequence labeling, balancing efficiency and accuracy [7].

3.4. ELECTRA

ELECTRA introduces a novel pre-training approach where the model acts as a discriminator, distinguishing real tokens from generated replacements in a text. Unlike BERT's generative pre-training, ELECTRA's discriminative task makes it more sample-efficient, achieving high performance with less computational cost. For classification tasks like GEC, ELECTRA excels in detecting subtle errors and predicting corrections, offering a scalable alternative for text processing applications [8].

3.5. Long Short-Term Memory (LSTM) Networks

LSTMs are recurrent neural networks designed to capture long-term dependencies in sequential data, making them suitable for text classification tasks. They use memory cells and gates to regulate information flow, mitigating issues like vanishing gradients. While less contextually rich than transformers, LSTMs are effective for tasks like sentiment analysis and GEC in resource-limited settings, especially when paired with word embeddings like GloVe or Word2Vec. Their sequential processing suits applications where computational resources are constrained [9].

3.6. Convolutional Neural Networks (CNNs) for Text

CNNs for text processing apply convolutional layers to capture local patterns, such as n-grams, in text data. These models excel in extracting features from fixed-size windows of text, making them efficient for tasks like sentiment classification or topic categorization. While not as adept at capturing long-range dependencies as transformers, CNNs offer computational simplicity and are effective for short-text classification or as a complement to other models in hybrid architectures [10].

The various classification models exhibit different strengths in terms of contextual understanding, computational efficiency, and adaptability to specific tasks. Transformer-based models like BERT, RoBERTa, DistilBERT, and ELECTRA dominate in tasks requiring deep contextual analysis, such as GEC, due to their bidirectional processing and large-scale pre-training. Non-transformer models like LSTMs and CNNs provide lightweight alternatives for simpler tasks or constrained environments. Combining the strengths of these models, where feasible, can enhance performance for complex text processing applications.

4. BERT for Sentence Classification

Sentence classification models like BERT have been widely adopted in educational applications due to their ability to understand contextual nuances in text and classify sentences or text segments with high accuracy. This section provides examples of how BERT and similar models are applied in educational contexts, focusing on practical implementations.

 doi:10.20944/preprints202505.2360.v1

4.1. Student Support and Personalized Learning

BERT-based models are used in educational platforms to classify student responses, feedback, or queries, enabling personalized learning experiences. For example, BERT can analyze open-ended student answers in online assessments to classify them as correct, partially correct, or incorrect, providing immediate feedback. Platforms like Duolingo leverage BERT-like models to classify learner responses in language exercises, identifying error types (e.g., grammar, vocabulary) and tailoring subsequent exercises to address specific weaknesses. Similarly, tools like Quizlet use BERT to classify flashcard responses, adapting difficulty based on user performance.

4.2. Faculty and Administrative Assistance

BERT-based classifiers streamline grading and administrative tasks by automating the evaluation of written responses. For instance, EdX and Coursera employ BERT to classify short-answer responses in MOOCs, categorizing them into predefined rubrics for grading efficiency. These models can also detect sentiment in student feedback forms, helping faculty gauge course satisfaction or identify areas needing improvement. Additionally, BERT-powered chatbots, like those integrated into Canvas LMS, classify student inquiries (e.g., technical issues, course content queries) to route them to appropriate support channels, reducing administrative burden.

4.3. Curriculum Planning and Educational Analytics

In curriculum planning, BERT is used to classify and analyze educational content for alignment with learning objectives. Tools like Turnitin's AI-powered features use BERT to classify student submissions for plagiarism or originality, while also identifying writing quality issues (e.g., coherence, clarity). Gradescope employs BERT-based models to classify patterns in student submissions, detecting common misconceptions or errors to inform curriculum adjustments. These models also support analytics by classifying student performance data into categories like "at-risk" or "proficient," enabling targeted interventions.

4.4. Educational Standards Chatbots and Knowledge Access

BERT excels in classifying educational standards and aligning content with frameworks like Common Core or NGSS. For example, tools like Achieve3000 use BERT to classify reading passages by difficulty level and alignment with specific standards, ensuring content matches student needs. BERT-based chatbots, such as those developed by ETS, classify teacher queries about standards, retrieving relevant guidelines or suggesting aligned lesson plans. These models also support inclusive education by classifying text for readability, adapting materials for diverse learners, or detecting cultural biases in content.

4.5. Research Implementation: Sentiment Analysis for Classroom Feedback

Our research showcases a BERT-based sentence classification system for analyzing classroom feedback. Using a fine-tuned BERT model with adapters (e.g., AdapterHub), we classify student and teacher feedback into positive, negative, or neutral sentiments. The system employs transfer learning to adapt pre-trained BERT weights to educational datasets, achieving high accuracy on metrics like F1-score and precision. Toxicity detection is integrated using a secondary BERT classifier to flag inappropriate feedback, ensuring safe and constructive classroom interactions. This implementation demonstrates BERT's potential for real-time, scalable feedback analysis in educational settings.

5. Design Aspects

The application of transfer learning with BERT has revolutionized natural language processing (NLP) tasks, particularly sentence classification, in educational contexts. Sentence classification, such as sentiment analysis of student feedback, topic categorization of classroom discussions, or automated grading of short-answer responses, benefits from BERT's ability to capture contextual nuances in text. This section outlines the key design aspects for developing and deploying BERT-based sentence

 doi:10.20944/preprints202505.2360.v1

classification systems tailored to educational needs, ensuring scalability, efficiency, and domain-specific performance, as illustrated in Figure 2.

Figure 2. Overview of the BERT-based sentence classification pipeline for educational applications, such as sentiment analysis or topic categorization.

5.1. Model Architecture and Pre-Training

BERT's transformer-based architecture leverages bidirectional contextual understanding, making it highly effective for sentence classification tasks. Unlike traditional unidirectional models, BERT processes text in both directions, capturing rich semantic relationships critical for educational applications. For instance, in sentiment analysis of student feedback, BERT can discern subtle emotional cues, distinguishing between nuanced responses like "The lecture was challenging but rewarding" and "The lecture was too difficult."

The pre-trained BERT model, typically trained on large corpora like Wikipedia and BookCorpus, provides a robust foundation for transfer learning. In educational settings, the model is fine-tuned on domain-specific datasets, such as student feedback surveys, classroom discussion transcripts, or assessment responses. The architecture consists of multiple transformer layers, each with self-attention mechanisms, enabling the model to weigh the importance of words in context. For sentence classification, a classification head (e.g., a fully connected layer with softmax activation) is added to the final hidden state of the [CLS] token, which aggregates the sentence's contextual representation.

Training and fine-tuning BERT require significant computational resources due to its large parameter set (e.g., 110M parameters for BERT-Base, 340M for BERT-Large). High-performance computing infrastructure, such as multi-GPU clusters with NVIDIA's latest architectures, is essential to handle the computational load. These GPUs support mixed-precision training (e.g., FP16 and FP8), reducing memory usage and accelerating training while maintaining model accuracy through dynamic precision adjustments.

5.2. Fine-Tuning and Customization

Fine-tuning BERT for sentence classification involves adapting the pre-trained model to specific educational tasks. This process requires careful design to balance performance, resource efficiency, and domain relevance.

Fine-Tuning Process: Fine-tuning begins with loading a pre-trained BERT model, followed by appending a task-specific classification head. The model is then trained on a labeled dataset relevant to the educational task, such as categorizing student feedback into positive, neutral, or negative sentiments. The dataset is typically split into training, validation, and test sets to optimize hyperparameters (e.g., learning rate, batch size) and prevent overfitting. Techniques like gradient accumulation and learning rate scheduling (e.g., using a linear warmup followed by decay) ensure stable convergence, especially on smaller educational datasets.

Domain-Specific Customization: To enhance performance, the model is fine-tuned on education-specific datasets, such as course evaluations, open-ended quiz responses, or discussion forum posts. For example, a BERT model fine-tuned on a dataset of student essays can classify responses based on clarity or argumentative strength, supporting automated grading systems. Parameter-efficient

 doi:10.20944/preprints202505.2360.v1

fine-tuning (PEFT) techniques, such as Low-Rank Adaptation (LoRA), are employed to update only a small subset of parameters, reducing computational costs while adapting the model to educational contexts.

Ethical Considerations and Bias Mitigation: Fine-tuning must address potential biases in educational datasets, such as skewed sentiment distributions or cultural biases in student responses. Techniques like reinforcement learning with human feedback (RLHF) or adversarial training can mitigate harmful biases, ensuring fair and inclusive outputs. Regular audits of model outputs and dataset curation are essential to maintain fairness.

5.3. Deployment and Inference Optimization

Deploying BERT-based sentence classification models in educational environments requires optimizing for low-latency inference and scalability to support real-time applications, such as live feedback analysis during virtual classes.

Inference Optimization: Techniques like model quantization (e.g., reducing weights to INT8 precision) and knowledge distillation (e.g., training a smaller "student" model from the BERT "teacher") reduce inference time and resource demands. These optimizations are crucial for applications like real-time sentiment analysis in online learning platforms, where rapid response generation enhances user experience.

System Integration: The classification model is integrated with educational platforms, such as learning management systems (LMS) or virtual tutoring environments. APIs facilitate seamless interaction, allowing the model to process inputs (e.g., student comments) and return classifications (e.g., sentiment labels) in real time.

Scalability and Sustainability: High-performance computing infrastructure supports scalable deployment, leveraging cloud or on-premises GPU clusters to handle large-scale inference tasks. Energy-efficient hardware, such as GPUs with low-power modes, aligns with sustainable computing goals in educational institutions.

5.4. Evaluation and Monitoring

Continuous evaluation and monitoring ensure the BERT-based classifier remains effective and relevant in dynamic educational contexts.

Evaluation Metrics: Model performance is assessed using metrics like accuracy, precision, recall, and F1-score, tailored to the classification task. For imbalanced datasets, such as sentiment analysis with predominantly positive feedback, metrics like macro-F1 or area under the ROC curve (AUC-ROC) provide a balanced evaluation.

Monitoring and Retraining: Post-deployment, the model is monitored for performance drift, which may occur due to evolving educational contexts (e.g., new teaching methods or student demographics). Automated monitoring pipelines detect degradation in classification accuracy, triggering retraining with updated datasets.

6. Case Study: BERT-Based Sentence Classification for Grammatical Correction

In our research, we present Sentence Classification using Transfer Learning with BERT. We leverage transfer learning with the Hugging Face Transformers library to fine-tune the BERT model, incorporating hyperparameter sweeps via Weights and Biases to optimize learning rate, batch size, and the number of epochs. The process involves downloading and preprocessing the dataset, tokenizing sentences with the BERT tokenizer, and preparing input data with special tokens, padding, and attention masks. The dataset is split into training and validation sets, and a custom training loop is implemented to fine-tune the model, logging performance metrics to Weights and Biases. We also implement a prediction method to predict if a sentence is grammatically correct or not.

 doi:10.20944/preprints202505.2360.v1

grammatically correct or incorrect. It was first published in May of 2018, and is one of the tests included in the "GLUE Benchmark" on which models like BERT are competing.

```
[1] !pip install wget
    import wget
    import os

    print('Downloading dataset...')

    # The URL for the dataset zip file.
    url = 'https://nyu-mll.github.io/CoLA/cola_public_1.1.zip'

    # Download the file (if we haven't already)
    if not os.path.exists('./cola_public_1.1.zip'):
        wget.download(url, './cola_public_1.1.zip')
```

```
Collecting wget
  Downloading wget-3.2.zip (10 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9655 sha256=fccabf2afbfb9c27e49d3f57a1c57e4be4ffce155262082e8d15f1dae04a0592
  Stored in directory: /root/.cache/pip/wheels/40/b3/0f/a40dbd1c6861731779f62cc4babcb234387e11d697df70ee97
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
Downloading dataset...
```

```
[3] if not os.path.exists('./cola_public/'):
        !unzip cola_public_1.1.zip
```

```
Archive:  cola_public_1.1.zip
   creating: cola_public/
  inflating: cola_public/README
   creating: cola_public/tokenized/
  inflating: cola_public/tokenized/in_domain_dev.tsv
  inflating: cola_public/tokenized/in_domain_train.tsv
  inflating: cola_public/tokenized/out_of_domain_dev.tsv
   creating: cola_public/raw/
  inflating: cola_public/raw/in_domain_dev.tsv
  inflating: cola_public/raw/in_domain_train.tsv
  inflating: cola_public/raw/out_of_domain_dev.tsv
```

Figure 3. Dataset Setup.

```
  inflating: cola_public/tokenized/out_of_domain_dev.tsv
   creating: cola_public/raw/
  inflating: cola_public/raw/in_domain_dev.tsv
  inflating: cola_public/raw/in_domain_train.tsv
  inflating: cola_public/raw/out_of_domain_dev.tsv
```

```
[4] import pandas as pd

    # Load the dataset into a pandas dataframe.
    df = pd.read_csv("./cola_public/raw/in_domain_train.tsv", delimiter='\t', header=None, names=['sentence_source', 'label', 'label_notes', 'sentence'])

    # Report the number of sentences.
    print('Number of training sentences: {:,}\n'.format(df.shape[0]))

    # Display 10 random rows from the data.
    df.sample(10)
```

```
Number of training sentences: 8,551
```

	sentence_source	label	label_notes	sentence
7110	sgww85	1	NaN	Kim seems to be just surviving, and Terry in d...
1971	r-67	1	NaN	Tom will force you to marry no student.
6724	m_02	1	NaN	The Labrador ate all the food which we left on...
2074	rhl07	1	NaN	Smith threw the first baseman the ball.
6861	m_02	1	NaN	The other plan she rejected out of hand.
2075	rhl07	0	*	Smith threw the first base the ball.
2125	rhl07	1	NaN	The music lent a festive air to the party.
4646	ks08	1	NaN	Chris was handed a note.
7205	sks13	0	*	it is the tall man come from the back that Mar...
4783	ks08	0	*	Who do you believe that invited Sara?

```
[5] # Get the lists of sentences and their labels.
    sentences = df.sentence.values
    labels = df.label.values
```

Figure 4. Loading the Dataset.

Tokenization

As mentioned earlier, the sentences that are to be fed into the BERT model must be tokenized using the BERT tokenizer. Let's take a look at an example.

```
[6] from transformers import BertTokenizer

    # Load the BERT tokenizer.
    print('Loading BERT tokenizer...')
    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)
    '''
    Original:  Our friends won't buy this analysis, let alone the next one we propose.
    Tokenized:  ['our', 'friends', 'won', "'", 't', 'buy', 'this', 'analysis', ',', 'let', 'alone', 'the', 'next', 'one', 'we', 'propose', '.']
    Token IDs:  [2256, 2814, 2180, 1005, 1056, 4965, 2023, 4106, 1010, 2292, 2894, 1996, 2279, 2028, 2057, 16599, 1012]
    '''
```

```
Loading BERT tokenizer...
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 1.34kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 2.64MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 7.20MB/s]
config.json: 100% 570/570 [00:00<00:00, 20.5kB/s]
'\nOriginal:  Our friends won\'t buy this analysis, let alone the next one we propose.\nTokenized:  [\'our\', \'friends\', \'won\', "\'", \'t\', \'buy\', \'this\', \'analysis\', \',\', \'let\', \'alone\', \'the\', \'next\', \'one\', \'we\', \'propose\', \'.\']\nToken IDs:  [2256, 2814, 2180, 1005, 1056, 4965, 2023, 4106, 1010, 2292, 2894, 1996, 2279, 2028, 2057, 16599, 1012]\n'
```

Before we process the entire dataset using this tokenizer, there are a few conditions that we need to satisfy in order to setup the training data for BERT:

- Add special tokens to the start and end of each sentence. At the end of every sentence, we need to append the special `[SEP]` token and for classification tasks, we must prepend the special `[CLS]` token to the beginning of every sentence.
- Pad & truncate all sentences to a single constant length
- Explicitly differentiate real tokens from padding tokens with the "attention mask". The "Attention Mask" is simply an array of 1s and 0s

Variables Terminal Executing (13m 16s) T4 (Python 3)

Figure 5. Tokenization Process.

 doi:10.20944/preprints202505.2360.v1


```
max_len = 0

# For every sentence...
for sent in sentences:

    # Tokenize the text and add `[CLS]` and `[SEP]` tokens.
    input_ids = tokenizer.encode(sent, add_special_tokens=True)

    # Update the maximum sentence length.
    max_len = max(max_len, len(input_ids))

print('Max sentence length: ', max_len)
```

```
Max sentence length:  47
```

```
[ ] !pip install wandb
```

```
Requirement already satisfied: wandb in /usr/local/lib/python3.11/dist-packages (0.19.11)
Requirement already satisfied: click!=8.0.0,>=7.1 in /usr/local/lib/python3.11/dist-packages (from wandb) (8.2.0)
Requirement already satisfied: docker-pycreds>=0.4.0 in /usr/local/lib/python3.11/dist-packages (from wandb) (0.4.0)
Requirement already satisfied: gitpython!=3.1.29,>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from wandb) (3.1.44)
Requirement already satisfied: platformdirs in /usr/local/lib/python3.11/dist-packages (from wandb) (4.3.8)
Requirement already satisfied: protobuf!=4.21.0,!=5.28.0,<7,>=3.19.0 in /usr/local/lib/python3.11/dist-packages (from wandb) (5.29.4)
Requirement already satisfied: psutil>=5.0.0 in /usr/local/lib/python3.11/dist-packages (from wandb) (5.9.5)
Requirement already satisfied: pydantic<3 in /usr/local/lib/python3.11/dist-packages (from wandb) (2.11.4)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.11/dist-packages (from wandb) (6.0.2)
Requirement already satisfied: requests<3,>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from wandb) (2.32.3)
Requirement already satisfied: sentry-sdk>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from wandb) (2.28.0)
Requirement already satisfied: setproctitle in /usr/local/lib/python3.11/dist-packages (from wandb) (1.3.6)
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (from wandb) (75.2.0)
Requirement already satisfied: typing-extensions<5,>=4.4 in /usr/local/lib/python3.11/dist-packages (from wandb) (4.13.2)
Requirement already satisfied: six>=1.4.0 in /usr/local/lib/python3.11/dist-packages (from docker-pycreds>=0.4.0->wandb) (1.17.0)
Requirement already satisfied: gitdb<5,>=4.0.1 in /usr/local/lib/python3.11/dist-packages (from gitpython!=3.1.29,>=1.0.0->wandb) (4.0.12)
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.11/dist-packages (from pydantic<3->wandb) (0.7.0)
Requirement already satisfied: pydantic-core==2.33.2 in /usr/local/lib/python3.11/dist-packages (from pydantic<3->wandb) (2.33.2)
Requirement already satisfied: typing-inspection>=0.4.0 in /usr/local/lib/python3.11/dist-packages (from pydantic<3->wandb) (0.4.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.0.0->wandb) (3.4.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.0.0->wandb) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.0.0->wandb) (2.4.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.0.0->wandb) (2025.4.26)
Requirement already satisfied: smmap<6,>=3.0.1 in /usr/local/lib/python3.11/dist-packages (from gitdb<5,>=4.0.1->gitpython!=3.1.29,>=1.0.0->wandb) (5.0.2)
```

Figure 6. Determining Maximum Sentence Length.

Initialize Wandb And Sweep Configs

```
[9] import wandb
sweep_config = {
    'method': 'random', #grid, random
    'metric': {
      'name': 'val_accuracy',
      'goal': 'maximize'
    },
    'parameters': {

        'learning_rate': {
            'values': [ 5e-5, 3e-5, 2e-5]
        },
        'batch_size': {
            'values': [16, 32]
        },
        'epochs':{
            'values':[2, 3, 4]
        }
    }
}
sweep_defaults = {
        'learning_rate': 5e-5,

        'batch_size': 32,

        'epochs':2
}

sweep_id = wandb.sweep(sweep_config)
```

```
wandb: Logging into wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.me/wandb-server)
wandb: You can find your API key in your browser here: https://wandb.ai/authorize?ref=models
wandb: Paste an API key from your profile and hit enter: ··········
wandb: WARNING If you're specifying your api key in code, ensure this code is not shared publicly.
wandb: WARNING Consider setting the WANDB_API_KEY environment variable, or running `wandb login` from the command line.
wandb: No netrc file found, creating one.
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
Create sweep with ID: ld49211x
Sweep URL: https://wandb.ai/abhishek-verma4607-srm-institute-of-science-and-technology/uncategorized/sweeps/ld49211x
```

Figure 7. Weights and Biases Sweep Configurations.

```
[11] import torch
from transformers import AutoTokenizer  # Ensure you import the tokenizer

# Assuming `sentences` and `labels` are predefined lists
# Example: sentences = ["sentence 1", "sentence 2", ...], labels = [0, 1, ...]

# Initialize the tokenizer (e.g., BERT tokenizer)
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")  # Replace with your model

# Tokenize all sentences and map tokens to their word IDs
input_ids = []
attention_masks = []

# For every sentence...
for sent in sentences:
    # `encode_plus` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
        sent,                      # Sentence to encode
        add_special_tokens=True,       # Add '[CLS]' and '[SEP]'
        max_length=64,             # Pad & truncate all sentences
        padding="max_length",          # Explicitly set padding to max_length
        truncation=True,               # Explicitly enable truncation
        return_attention_mask=True,    # Construct attention masks
        return_tensors="pt"            # Return PyTorch tensors
    )

    # Add the encoded sentence to the list
    input_ids.append(encoded_dict["input_ids"])
    attention_masks.append(encoded_dict["attention_mask"])

# Convert the lists into tensors
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(labels)

# Print sentence 0, now as a list of IDs
```

Figure 8. Tokenization Implementation.

 doi:10.20944/preprints202505.2360.v1

Train-Test Split

```
[12] from torch.utils.data import TensorDataset, random_split

     # Combine the training inputs into a TensorDataset.
     dataset = TensorDataset(input_ids, attention_masks, labels)

     # Create a 90-10 train-validation split.

     # Calculate the number of samples to include in each set.
     train_size = int(0.9 * len(dataset))
     val_size = len(dataset) - train_size

     # Divide the dataset by randomly selecting samples.
     train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

     print('{:>5,} training samples'.format(train_size))
     print('{:>5,} validation samples'.format(val_size))

7,695 training samples
  856 validation samples

[13] from torch.utils.data import DataLoader, RandomSampler, SequentialSampler
     import wandb
     # WANDB PARAMETER
     def ret_dataloader():
         batch_size = wandb.config.batch_size
         print('batch_size = ', batch_size)
         train_dataloader = DataLoader(
                     train_dataset,  # The training samples.
                     sampler = RandomSampler(train_dataset), # Select batches randomly
                     batch_size = batch_size # Trains with this batch size.
                 )

         validation_dataloader = DataLoader(
                     val_dataset, # The validation samples.
                     sampler = SequentialSampler(val_dataset), # Pull out batches sequentially.
                     batch_size = batch_size # Evaluate with this batch size.
                 )
         return train_dataloader,validation_dataloader
```

Figure 9. Train-Test Split.

```
[13]     batch_size = wandb.config.batch_size
         print('batch_size = ', batch_size)
         train_dataloader = DataLoader(
                     train_dataset,  # The training samples.
                     sampler = RandomSampler(train_dataset), # Select batches randomly
                     batch_size = batch_size # Trains with this batch size.
                 )

         validation_dataloader = DataLoader(
                     val_dataset, # The validation samples.
                     sampler = SequentialSampler(val_dataset), # Pull out batches sequentially.
                     batch_size = batch_size # Evaluate with this batch size.
                 )
         return train_dataloader,validation_dataloader
```

Load Pre-trained BERT model

```
[15] from transformers import BertForSequenceClassification, BertConfig
     from torch.optim import AdamW  # Import AdamW from torch.optim

     def ret_model():
         model = BertForSequenceClassification.from_pretrained(
             "bert-base-uncased",
             num_labels=2,
             output_attentions=False,  # Whether the model returns attention weights
             output_hidden_states=False,  # Whether the model returns all hidden-states
         )

         return model

[16] def ret_optim(model):
         print('Learning_rate = ',wandb.config.learning_rate )
         optimizer = AdamW(model.parameters(),
                           lr = wandb.config.learning_rate,
                           eps = 1e-8
                         )
         return optimizer
```

Figure 10. Loading Pre-trained BERT Model.

```
[17] from transformers import get_linear_schedule_with_warmup

     def ret_scheduler(train_dataloader,optimizer):
         epochs = wandb.config.epochs
         print('epochs =>', epochs)
         # Total number of training steps is [number of batches] x [number of epochs].
         # (Note that this is not the same as the number of training samples).
         total_steps = len(train_dataloader) * epochs

         # Create the learning rate scheduler.
         scheduler = get_linear_schedule_with_warmup(optimizer,
                                                     num_warmup_steps = 0, # Default value in run_glue.py
                                                     num_training_steps = total_steps)
         return scheduler

[18] import numpy as np

     # Function to calculate the accuracy of our predictions vs labels
     def flat_accuracy(preds, labels):
         pred_flat = np.argmax(preds, axis=1).flatten()
         labels_flat = labels.flatten()
         return np.sum(pred_flat == labels_flat) / len(labels_flat)
     import time
     import datetime

     def format_time(elapsed):
         '''
         Takes a time in seconds and returns a string hh:mm:ss
         '''
         # Round to the nearest second.
         elapsed_rounded = int(round((elapsed)))

         # Format as hh:mm:ss
         return str(datetime.timedelta(seconds=elapsed_rounded))

[ ]  #torch.multiprocessing.set_start_method('spawn', force=True)
```

The Train Function

Figure 11. Scheduler and Accuracy Prediction.

 doi:10.20944/preprints202505.2360.v1

The Train Function

```
[21] import random
     import numpy as np
     import torch
     import wandb
     import time

     def train():
         # Initialize W&B
         wandb.init()

         # Set device
         device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
         print(f"Device: {device}")

         # Load model and move to device
         model = ret_model()
         model.to(device)

         # Load dataloaders and optimizer
         train_dataloader, validation_dataloader = ret_dataloader()
         optimizer = ret_optim(model)
         scheduler = ret_scheduler(train_dataloader, optimizer)

         # Set seed for reproducibility
         seed_val = 42
         random.seed(seed_val)
         np.random.seed(seed_val)
         torch.manual_seed(seed_val)
         if device.type == 'cuda':
             torch.cuda.manual_seed_all(seed_val)

         # Store training statistics
         training_stats = []

         # Measure total training time
         total_t0 = time.time()
         epochs = wandb.config.epochs

         # For each epoch...
```

Figure 12. Training Code (Part 1).

```
[21]     # For each epoch...
         for epoch_i in range(epochs):
             print(f"\n======== Epoch {epoch_i + 1} / {epochs} ========")
             print('Training...')

             # Measure epoch time
             t0 = time.time()
             total_train_loss = 0

             # Training mode
             model.train()

             # For each batch...
             for step, batch in enumerate(train_dataloader):
                 if step % 40 == 0 and not step == 0:
                     elapsed = format_time(time.time() - t0)
                     print(f'  Batch {step:>5,} of {len(train_dataloader):>5,}. Elapsed: {elapsed}.')

                 # Unpack batch
                 b_input_ids, b_input_mask, b_labels = batch
                 b_input_ids = b_input_ids.to(device)
                 b_input_mask = b_input_mask.to(device)
                 b_labels = b_labels.to(device)

                 # Clear gradients
                 model.zero_grad()

                 # Forward pass
                 outputs = model(
                     input_ids=b_input_ids,
                     attention_mask=b_input_mask,
                     labels=b_labels
                 )

                 # Extract loss and logits
                 loss = outputs.loss  # Access loss attribute
                 logits = outputs.logits  # Access logits attribute

                 # Log loss to W&B
                 wandb.log({'train_batch_loss': loss.item()})
                 total_train_loss += loss.item()

                 # Backward pass
```

Figure 13. Training Code (Part 2).

```
[21]             # Backward pass
                 loss.backward()

                 # Clip gradients to prevent exploding gradients
                 torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

                 # Update parameters and learning rate
                 optimizer.step()
                 scheduler.step()

             # Calculate average training loss
             avg_train_loss = total_train_loss / len(train_dataloader)
             training_time = format_time(time.time() - t0)
             wandb.log({'avg_train_loss': avg_train_loss})

             print(f"  Average training loss: {avg_train_loss:.2f}")
             print(f"  Training epoch took: {training_time}")

             # Validation
             print("\nRunning Validation...")
             t0 = time.time()
             model.eval()
             total_eval_accuracy = 0
             total_eval_loss = 0

             for batch in validation_dataloader:
                 b_input_ids, b_input_mask, b_labels = batch
                 b_input_ids = b_input_ids.to(device)
                 b_input_mask = b_input_mask.to(device)
                 b_labels = b_labels.to(device)

                 with torch.no_grad():
                     outputs = model(
                         input_ids=b_input_ids,
                         attention_mask=b_input_mask,
                         labels=b_labels
                     )

                 # Extract loss and logits
                 loss = outputs.loss
                 logits = outputs.logits
```

Figure 14. Training Code (Part 3).

 doi:10.20944/preprints202505.2360.v1


```
wandb:  learning_rate: 5e-05
Tracking run with wandb version 0.19.11
Run data is saved locally in /content/wandb/run-20250527_210910-ynjov6t5
Syncing run sparkling-sweep-6 to Weights & Biases (docs)
Sweep page: https://wandb.ai/abhishek-verma4607-srm-institute-of-science-and-technology/uncategorized/sweeps/ld4921lx
View project at https://wandb.ai/abhishek-verma4607-srm-institute-of-science-and-technology/uncategorized
View sweep at https://wandb.ai/abhishek-verma4607-srm-institute-of-science-and-technology/uncategorized/sweeps/ld4921lx
View run at https://wandb.ai/abhishek-verma4607-srm-institute-of-science-and-technology/uncategorized/runs/ynjov6t5
Device: cuda
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
batch_size =  16
Learning_rate =  5e-05
epochs => 3

======== Epoch 1 / 3 ========
Training...
  Batch    40 of   481. Elapsed: 0:00:07.
  Batch    80 of   481. Elapsed: 0:00:15.
  Batch   120 of   481. Elapsed: 0:00:23.
  Batch   160 of   481. Elapsed: 0:00:30.
  Batch   200 of   481. Elapsed: 0:00:38.
  Batch   240 of   481. Elapsed: 0:00:46.
  Batch   280 of   481. Elapsed: 0:00:53.
  Batch   320 of   481. Elapsed: 0:01:01.
  Batch   360 of   481. Elapsed: 0:01:09.
  Batch   400 of   481. Elapsed: 0:01:16.
  Batch   440 of   481. Elapsed: 0:01:24.
  Batch   480 of   481. Elapsed: 0:01:31.
  Average training loss: 0.51
  Training epoch took: 0:01:31

Running Validation...
  Accuracy: 0.81
  Validation Loss: 0.44
  Validation took: 0:00:03

======== Epoch 2 / 3 ========
Training...
  Batch    40 of   481. Elapsed: 0:00:07.
  Batch    80 of   481. Elapsed: 0:00:15.
  Batch   120 of   481. Elapsed: 0:00:23.
  Batch   160 of   481. Elapsed: 0:00:30.
  Batch   200 of   481. Elapsed: 0:00:38.
  Batch   240 of   481. Elapsed: 0:00:46.
  Batch   280 of   481. Elapsed: 0:00:53.
```

Figure 15. Training Epochs.

References

1. M. Kaneko, M. Mita, S. Kiyono, and J. Suzuki, "Encoder-decoder models can benefit from pre-trained bert for grammatical error correction," *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020. [Online]. Available: https://aclanthology.org/2020.acl-main.565
2. K. Omelianchuk, V. Atrasevych, A. Chernodub, and Y. Skurzhanskyi, "Gector – grammatical error correction: Tag, don't rewrite," *Proceedings of the 15th Workshop on Innovative Use of NLP for Building Educational Applications*, 2020. [Online]. Available: https://aclanthology.org/2020.bea-1.16
3. F. Stahlberg and S. Kumar, "Seq2edits: Sequence transduction using span-level edit operations with bert," *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020. [Online]. Available: https://aclanthology.org/2020.emnlp-main.393
4. A. Awasthi, S. Sarawagi, and P. Goyal, "Parallel iterative edit models for local sequence transduction," *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019. [Online]. Available: https://aclanthology.org/D19-1280
5. J. Lichtarge, A. Dabrowska, C. Alberti, and J. Devlin, "Grammatical error correction with bert-based synthetic data," *Proceedings of the 14th International Conference on Computational Linguistics and Intelligent Text Processing (CICLing)*, 2020. [Online]. Available: https://arxiv.org/abs/1911.09329
6. Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019. [Online]. Available: https://arxiv.org/abs/1907.11692
7. V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019. [Online]. Available: https://arxiv.org/abs/1910.01108
8. K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *International Conference on Learning Representations (ICLR)*, 2020. [Online]. Available: https://arxiv.org/abs/2003.10555
9. S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735
10. Y. Kim, "Convolutional neural networks for sentence classification," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014. [Online]. Available: https://aclanthology.org/D14-1181