BERT for Error Detection

Comparing Fine-Tuning and Embedding Classification

Bachelor Thesis

Faculty of Science, University of Bern

submitted by

Cécile Semelet

from Bern, Switzerland

Supervision:

PD Dr. Kaspar Riesen

Corina Masanti

Institute of Computer Science (INF)

University of Bern, Switzerland

Abstract

This thesis explores more efficient and effective alternatives to fine-tuning large language models for grammatical error detection, focusing on binary sentence classification into correct and incorrect sentences. Fine-tuning pre-trained models like BERT is currently a leading approach, but it is computationally expensive and time-consuming.

The goal of this thesis is to use BERT embeddings as input for classifiers such as SVM and XGBoost to achieve faster and improved results. Two main strategies were evaluated: First, embeddings from a pre-trained BERT model without fine-tuning were used as input, aiming for faster results. Secondly, BERT was fine-tuned using a dataset composed of correct and incorrect sentences, and the fine-tuned embeddings were extracted and used as input with these same classifiers, with the goal of getting more accurate results. These approaches were compared to a fine-tuned BERT model using its classification head as a baseline.

This thesis found that using pre-trained embeddings with no fine-tuning steps gave no satisfactory result, with no dataset achieving an accuracy of over 60 percent. In contrast, embeddings from a fine-tuned BERT model produced results comparable to the fine-tuned BERT baseline. However, they did not exceed it, which makes using BERT's fine-tuned model more efficient. These findings highlight how important the fine-tuning step is for BERT for error detection.

Acknowledgements

I would like to express my sincere gratitude to Prof. Dr. Riesen for his valuable feedback and continuous support throughout the course of this thesis. My heartfelt thanks also go to Corina Masanti for her guidance, numerous discussions, and constructive feedback during our many meetings. I am equally grateful to my employer for providing flexibility in my working hours, which allowed me to dedicate the necessary time and focus to this work.

Finally, I wish to thank my family and friends for their unwavering support and encouragement over the past months.

Contents

Chapter 1

Introduction

This thesis is set in the field of Artificial Intelligence (AI). AI has the goal of enabling computers to exhibit intelligent behavior. There are many different applications, ranging from customer support chatbots to the automatic booking of calendars to the creation of deepfakes. Within AI, machine learning enables computers to learn from data to solve different tasks. Instead of being explicitly programmed to solve each task, machine learning algorithms improve their performance autonomously using training data [1]. An example is classification. A classifier is a model that learns to distinguish between different categories, which are labeled beforehand based on patterns in the data, for example, to differentiate dogs and cats. Once trained, it can assign unseen data points to the most likely class [2].

One subfield of machine learning is natural language processing (NLP) [3], which focuses on understanding, interpreting, and generating human language. This includes tasks such as language translation, sentiment analysis, question answering, and summarization. Among these, a key task is text classification, which is the focus of this thesis.

A challenging form of text classification is error detection, which aims to differentiate between correct and incorrect sentences. Different approaches can be used to achieve this, such as rule-based approaches, the use of classifiers, and statistical approaches. This task has applications in educational tools, grammar correction software, and automated writing assistance [4].

Recent advances in NLP using transformer-based large language models have opened new possibilities to improve error detection. These models are built using the transformer architecture, a neural network design that introduced the concept of self-attention, allowing each word in a sentence to "pay attention" to the other words. Transformers are highly scalable and support parallel computing, reducing training time [5].

An important component in transformer models is embeddings, vector repre-

sentations of words that encode their meaning. More specifically, transformer models generate and use contextualized embeddings, where the representation of each word depends on its surrounding context and therefore changes. Such an approach is helpful, since words can take on different meanings depending on how they are used [6].

This thesis focuses on BERT, a transformer-based model that has shown strong performance in many applications, including sentence classification [7].

BERT is trained in two stages:

- **Pre-training:** BERT is trained on large amounts of unlabeled text using tasks such as masked language modeling and next-sentence prediction. This enables the model to learn general language representations. Google's pre-trained BERT models are widely available and often reused as a starting point for NLP tasks.

- **Fine-tuning:** For a specific downstream task, BERT is fine-tuned using labeled data. This step adjusts the model's weights slightly to improve performance on the target task. For classification tasks, a classification head is added at the end of the model.

The goal of this thesis is to investigate alternative approaches to the standard method of using a fine-tuned BERT model for classification of sentences as grammatically correct or incorrect. Although this approach typically yields strong accuracy, fine-tuning is computationally expensive and time consuming. This thesis aims to use pre-trained BERT embeddings without the fine-tuning step as input for classifiers such as SVM and XGBoost to achieve faster training, potentially reducing computational cost. The hypothesis is that using high-quality embeddings with strong classifiers might have a similar performance as BERT's fine-tuned model, without the fine-tuning step. As a second experiment, BERT's fine-tuned embeddings are also extracted and used with the classifiers. The aim here is not to reduce training time, as the time-consuming fine-tuning step must be performed, but to evaluate whether traditional classifiers can outperform the built-in BERT classification head in terms of accuracy.

To summarize, we evaluated three methods using three synthetically generated datasets of 100,000 sentences each, labeled `0` (no error) or `1` (contains an error):

1. **Fine-tuning BERT on the task:** Use BERT's integrated classification head to predict sentence-level errors.

2. **Using pre-trained embeddings with a classifier:** Extract contextualized embeddings from a pre-trained model and pass them to external classifiers. Two classifiers are tested: Support Vector Machines and XGBoost.

3. **Using fine-tuned contextualized embeddings with a classifier:** Fine-tune BERT in the task and then extract contextualized embeddings from the fine-tuned model. These embeddings are used as input for the same classifiers.

This thesis begins by introducing fundamental concepts of machine learning in Chapter 2, including the commonly used classifiers SVM and XGBoost. Then it delves into NLP, specifically embeddings, transformer architecture, and BERT in Chapter 3. The experimental setup and results are presented and analyzed in Chapter 4. Finally, Chapter 5 offers a summary of the results and outlines potential directions for future work.

Chapter 2

Machine Learning Basics

In this chapter, machine learning fundamentals essential to understanding this thesis are introduced. First, in Section 2.1, we begin with a general introduction to machine learning, discussing core concepts such as how machine learning models work, what common learning paradigms exist, and what different datasets are used. Next in Section 2.2, we focus on classifiers, a type of machine learning model. The classification algorithms support vector machine (SVM) and XGBoost are explained. Finally, in Section 2.3, we provide an overview of Principal Component Analysis (PCA), a dimensionality reduction technique that is used for data preprocessing and analysis in the experiments.

2.1 Machine Learning Introduction

Machine learning is a branch of AI that uses algorithms to learn from data. Instead of being programmed with explicit rules, machine learning models learn patterns from data and are trained to perform a wide range of tasks, such as classifying emails as spam or not, creating text from a prompt, and categorizing the emotional intent of a sentence [1].

2.1.1 Building Blocks of Machine Learning Models

The goal of a machine learning model is to make predictions about the data given as input. As described by Goodfellow et al. [8], machine learning models work by using a function $f_\theta : \mathcal{X} \to \mathcal{Y}$ that converts an input $\mathbf{x} \in \mathcal{X}$ (for example, an image or a word) into an output $\hat{\mathbf{y}} = f_\theta(\mathbf{x})$ (for example, a class label or a numerical prediction).

Depending on the task, a wide variety of models can be applied, ranging from linear regressors for predicting continuous values such as house prices to decision

trees for assigning categorical labels and deep neural networks for complex tasks such as image or language understanding.

A model's behavior is determined by its parameters, for example the weights and biases in a neural network and splits in a decision tree. The function f_θ has parameters θ, which are trained during the optimization process to achieve better results on a given task. They are learned from data and are not set in advance. The parameters are typically called weights.

Training is performed using an optimization algorithm, which iteratively updates the model's weights in order to reduce prediction error until a threshold of accuracy is reached. It can consist of several epochs, where each epoch is going through the whole training dataset once.

To support training and to check how accurate the model prediction is, a loss function is used. The loss function measures how far off the model's predictions are from the true values in the training data. It guides the optimization process by assigning a numerical penalty to incorrect predictions. Common loss functions include:

- **Mean Squared Error (MSE)** commonly used in regression tasks:

$$\mathcal{L}_{\text{MSE}}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}^{(i)} - y^{(i)}\right)^2,$$

  where $\hat{y}^{(i)}$ is the model prediction for the i-th sample, $y^{(i)}$ is the corresponding true value, and n is the number of samples.

- **Cross-Entropy Loss** widely used in classification tasks:

$$\mathcal{L}_{\text{CE}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log\left(\hat{y}_k\right),$$

  where $\mathbf{y} \in \{0,1\}^K$ is the one-hot encoded ground-truth label, $\hat{\mathbf{y}} \in [0,1]^K$ is the predicted probability distribution over the K classes, and K is the total number of classes.

In order to initialize the training procedure and the model architecture itself, hyperparameters are needed. They are set by the programmer when starting the model. They are external to the model, and their value cannot be trained. Examples are learning rate and network depth. In order to find the best hyperparameters, several model configurations have to be tested. To avoid having to search for the best hyperparameters by hand, one can use grid search to automatically iterate through different hyperparameters.

2.1.2 Learning Paradigms

Machine learning algorithms can be grouped by the kind of data they use for training. In this thesis, we concentrate on two paradigms, supervised and self-supervised learning, because they are used by the models evaluated in later chapters. There are other paradigms, such as unsupervised learning and reinforcement learning.

Supervised learning In this method, the algorithm uses labeled data to learn patterns in the data, and the model is trained to predict that label from the input. Supervised tasks fall into two broad categories:

- **Classification task:** The label y is categorical, e.g. $\{\,\text{spam}, \text{not-spam}\,\}$ or $\{\,\text{cat}, \text{dog}, \text{bird}\,\}$. The model predicts either the most likely class or a probability distribution over multiple classes.
- **Regression task:** The label y is a numerical value, such as a house price. The model then predicts a numerical value for new data.

Supervised learning is powerful when large, high-quality labeled datasets are available, but labeling can be costly or impractical in many domains [9].

Self-supervised learning Self-supervised methods create their own labels from the data, eliminating the need for human-annotated labels. A task is constructed in which part of the input is masked or otherwise hidden, and the model is trained to reconstruct or predict that missing information. Self-supervised learning offers a practical route when unlabeled data is abundant but labeled data is scarce [10].

2.1.3 Data Workflow

There are three types of datasets used for machine learning: the training, testing and validation dataset. The training dataset is used to train the model. The optimization algorithm adjusts the parameters to reduce the loss on this data. The validation dataset is used to evaluate the different hyperparameter configurations. With this dataset, the best-performing configuration is selected. The testing dataset is used once at the end of development to estimate how well the final model generalizes to new data. It must remain completely untouched during training and validation to avoid bias, since otherwise the model might just fit the test data.

2.1.4 Generalization and Model Performance

An important challenge in machine learning is to ensure that a model makes satisfactory predictions with not only training data but also with unseen data. Two

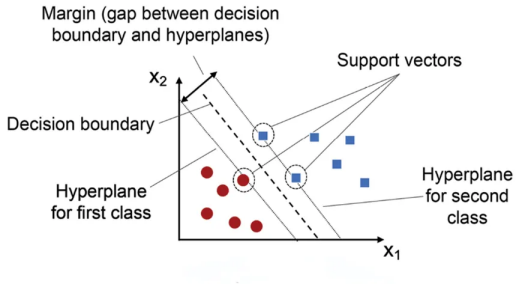

Figure 2.1: Decision boundary and support vectors in SVM [11].

common failures are underfitting and overfitting. With underfitting, the model is too simple to capture the underlying structure of the data. This results in poor performance on both the training and test data. With overfitting, the model fits the training data too closely, including noise or outliers. It performs well with training data but poorly with new data. Balancing model complexity and training data size is critical for the model to generalize well to new data.

2.2 Classifiers

Classifiers are a fundamental component of supervised machine learning. Classification can be binary, where there are only two possible categories (example: detecting spam e-mails), or multi-class, where there are more than two categories (example: labeling plants using pictures). Numerous classifiers exist, including random forest, logistic regression, and others [2].

We are going to focus on two classification algorithms central to this thesis: SVM and XGBoost. Since the goal of the thesis is to distinguish between two classes, correct and incorrect sentences, we will explore binary classification tasks.

2.2.1 Support Vector Machines

The support vector machine works by identifying the optimal hyperplane that maximally separates data points of different classes in a high-dimensional space while maximizing the distance between the hyperplane and the nearest data point from each class. These closest data points, which determine the position of the hyperplane, are called support vectors, as shown in Figure 2.1. The larger the margin, the better the model is expected to generalize to unseen data.

During training, the algorithm learns the hyperplane parameters from the labeled data. Once this decision boundary is fixed, a new point is classified by identifying on which side of the hyperplane it falls [12].

Hyperparameters There are several important hyperparameters that can be adjusted to improve the performance of an SVM model:

- **Kernel**
  The kernel determines how the SVM transforms the input data. It projects the data in a higher-dimensional space, where it may be easier to separate the classes with a clear boundary. An example of how a kernel works is shown in Figure 2.2.

  Common kernel types include:

  - **Linear**: useful when the data is already roughly linearly separable.
  - **RBF (Radial Basis Function)**: allows the model to create curved boundaries and adapt to more complex data distributions.
  - **Polynomial**: fits curved decision boundaries with a polynomial shape, depending on the degree.

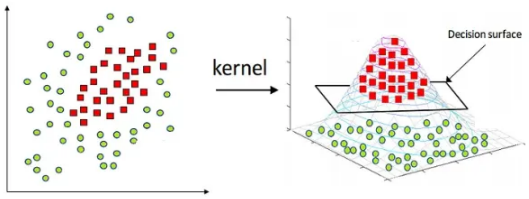

Figure 2.2: Illustration of kernel projecting data points from 2D space to 3D space [13]

- **C (Regularization Strength)**
  The C parameter controls how much the model tries to avoid misclassifying training examples. A high C value forces the SVM to classify all training points correctly if possible, which can lead to a complex boundary that overfits. A smaller C allows the model to make some mistakes in training in exchange for a simpler and more general boundary that may perform better on new data.

- **Gamma**
  Gamma defines how far the influence of a single training point reaches. A high gamma means that each point has a narrow area of influence, which leads the model to create highly flexible boundaries that closely follow the training data. This can lead to overfitting. A low gamma spreads the influence more broadly, creating smoother, simpler decision boundaries.

2.2.2 XGBoost

XGBoost (eXtreme Gradient Boosting) is based on the use of several decision trees. A decision tree is a model that divides input data into smaller and smaller groups based on specific feature values. It starts with a root node, and at each decision node, the data is split depending on whether a feature value is above or below a certain value. This process continues until a final leaf node is reached, which holds the prediction of the class label [14]. While decision trees are easy to understand and interpret, they tend to overfit training data, especially if they are very deep or complex [15].

XGBoost addresses this problem by using a technique called gradient boosting, where many decision trees are trained one after the other. Each new tree is trained to correct the errors made by the previous ones. With this approach, the model improves over time, focusing more on examples that were previously misclassified. The "gradient" in gradient boosting refers to the use of gradient descent to lower the loss function, similar to how neural networks are trained. Instead of adjusting weights, XGBoost adjusts the structure and values of trees based on the gradient of a loss function [16].

Hyperparameters

- **Maximal depth**
  This parameter controls how deep each individual tree in the ensemble can grow. Deeper trees can capture more complex patterns in the training data. However, if the trees are too deep, the model is likely to overfit and perform poorly on unseen data.

- **Learning rate**
  The learning rate determines the contribution of each new tree to the overall model. A lower learning rate means that the model learns more slowly and cautiously, often resulting in better generalization. However, smaller values typically require training more trees to achieve satisfactory performance.

- **Number of estimators**
  This parameter specifies the number of trees used in the model. Using too many trees can cause overfitting, while too few may result in underfitting and poor performance.

- **Subsample**
  This controls the fraction of the training data that is randomly sampled (without replacement) to grow each tree. Using a value less than 1.0 introduces

randomness into the training process, which can help prevent overfitting by making the model more robust.

- **Column subsample ratio per tree**
  This parameter specifies the fraction of features (columns) that are randomly sampled for each tree. By limiting the features used per tree, the model can reduce overfitting and increase its ability to generalize to new data.

2.3 Principal Component Analysis

Principal Component Analysis (PCA) is a machine learning technique used to reduce the dimensionality of vectors while preserving as much of the original information as possible. By applying PCA, we can represent the data with fewer variables, which helps simplify models, reduce storage size, and potentially improve generalization by eliminating noise and redundant features. For example, if one variable represents speed in km/h and another represents speed in miles/h, they provide the same information. One can be removed without any loss of information. Additionally, PCA is useful for visualization when reducing data to two or three dimensions.

PCA works by identifying the directions (called principal components) that capture the most significant patterns in the data. To do so, the variance is calculated. Variance is a measure of how spread out the data is around the mean. The higher the variance of a variable, the more information is contained. As an example, in figure 2.3 we can see that the variance of principal component 1 is much higher than the variance of principal component 2. Mathematically, for a set of values $x_1, x_2, \ldots, x_n$, the variance σ^2 is defined as:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

where μ is the mean of the values.

Before computing principal components, the data is typically standardized. This ensures that variables on different scales contribute equally to the analysis. Standardization involves subtracting the mean of each feature and dividing by its standard deviation:

$$z = \frac{value - mean}{standard\ deviation}$$

This prevents variables with larger ranges (e.g., 0–1000) from dominating those

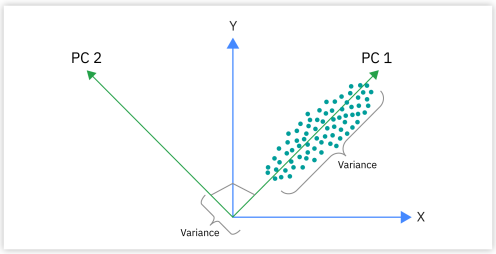

Figure 2.3: Illustration of principal components and their variance [17]

with smaller ranges (e.g., 0-1). Once standardized, PCA transforms the dataset into a new coordinate system, where each axis corresponds to a principal component. These principal components are linear combinations of the original variables and are ordered by the amount of variance they capture: the first principal component accounts for the most variance, the second accounts for the next largest amount, and so on. The data is projected onto these components to obtain a lower-dimensional representation [18].

Chapter 3

Natural Language Processing Basics

In Section 3.1, we introduce the fundamentals of natural language processing. Section 3.2 explores different types of embeddings and Section 3.3 provides an overview of the transformer family, including the standard transformer architecture in Section 3.3.1 and the transformer-based model BERT used in this thesis in Section 3.3.2 with its training process.

3.1 Natural Language Processing Introduction

Natural language processing (NLP) focuses on teaching machines to understand, interpret, and generate human language. NLP has a wide range of tasks, including text generation, which includes machine translation and dialogue. Another common task is sentiment analysis, where the goal is to classify the emotions expressed in text. Named entity recognition is also an important function, aiming to categorize words into predefined categories such as names and locations. Additionally, NLP is widely used in spam detection systems to filter out unwanted emails and in error detection tools that identify mistakes in written text.

Early NLP systems relied on hand-crafted rules. These were gradually replaced by statistical models and, more recently, by neural networks. Neural networks are designed after the human brain and consist of layers of nodes, which learn by adjusting weighted connections between nodes. Today's state of the art is driven by large language models (LLMs), neural networks with hundreds of millions to trillions of parameters, most of which are based on the transformer architecture. LLMs are first pre-trained on large amounts of text in a self-supervised manner and subsequently fine-tuned using task-specific labeled data. They have shown strong performance on a wide range of natural language tasks [3].

3.2 Text Representation

3.2.1 Embeddings

In order to use text data to train machine learning algorithms, the data needs to be converted to a numerical format that computers can process. A solution is the use of embeddings, which represent words and sentences as vectors in a high-dimensional space. These vectors capture different properties of the text and can be used for downstream machine learning tasks. There are different types of embeddings, varying in complexity and usefulness [6].

One-hot encoded embeddings The simplest method to generate embeddings is one-hot encoding. With this approach, each word is represented by a binary vector that has a length equal to the size of the vocabulary. All values are zero except for a single one in the index corresponding to that word. This helps distinguish the words; however, it does not encode anything about what the words mean or what the relationship between the words is. In addition, the size of the vectors grows with the dataset and can become too large.

Static embeddings Later, more sophisticated statistical methods were designed. These methods aim to capture semantic relationships between words based on how often they appear next to each other. Word2vec used to be the go-to approach [19]. With this algorithm, words that appear in a similar context are assigned vectors similar to each other. The idea behind this was that words that appear close to each other also have a similar meaning. These methods were a breakthrough, allowing models to understand semantic similarities, such as *king* - *man* = *queen* - *woman*, visualized in Figure 3.1. However, these models still have the shortcoming that each word is assigned to a single vector, regardless of the context in which the word is used. These embeddings are called static.

Contextualized embeddings The current state-of-the-art uses contextualized embeddings. These embeddings help us not only to distinguish the words but also to encode the words' meaning depending on the context. Words get a different embedding vector depending on the sentence they are used in. So, the same word will have a different embedding each time.

These embeddings are generated by models such as BERT or GPT. They have become the foundation for NLP tasks, enabling significant advances.

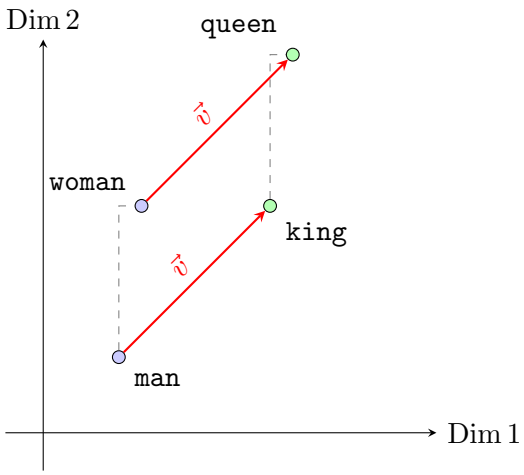

Figure 3.1: Static embeddings capture semantic analogies: the vector from man to king closely matches the vector from woman to queen

Word-wise and sentence-wise embeddings Embeddings can be created word-wise or sentence-wise. With word-wise embeddings, each word is represented by a vector, and with sentence-wise embeddings, each sentence is represented by a vector. Figure 3.2 illustrates the difference between the two approaches. This thesis investigates both types of embeddings and compares their performance in downstream classification tasks.

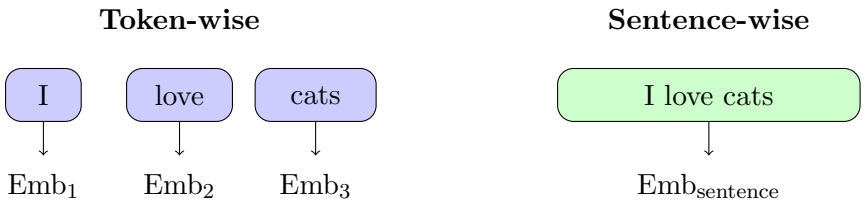

Figure 3.2: Comparison of token-wise (left) and sentence-wise (right) embeddings.

3.2.2 Tokenization

A tokenizer is an algorithm that splits a sentence into smaller subwords called tokens. Most embeddings are not directly generated from whole words but smaller subword units. This is because having an embedding for each word would result in an extremely large vocabulary and might fail to capture relationships between words. The words *eating* and *eat* would have completely different embeddings. Instead, the word *eating* is divided into *eat* and *ing*, both of which get different embeddings. This approach also improves the model's ability to understand unseen words, since it still might know the subword units.

Not all models use the same subunits as tokens. Models that represent different

languages will use a different token library. As an example, tokens used for the English language are going to be different from tokens used for the French language, since not the same word splits are useful. Figure 3.3 illustrates how a sentence is split into subword tokens by a typical tokenizer.

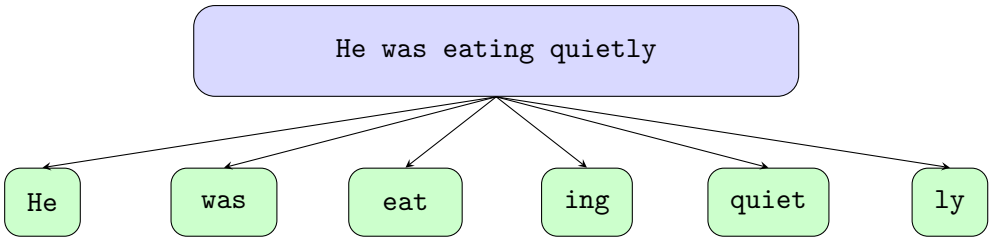

Figure 3.3: Example of tokenizing a sentence into subword units.

3.3 Transformer Family

Originally developed for translation tasks, the transformer architecture is now at the heart of models like BERT and GPT and has revolutionized NLP through self-attention mechanisms. Transformers are meant to solve any task that transforms an input sequence into an output sequence. They learn to understand and generate text by analyzing patterns in large amounts of text data. Transformers also have the advantage of processing data in parallel, which makes GPUs useful and overall training time much faster. The transformer was introduced in the paper *Attention Is All You Need* [5] by Vaswani, developed at Google Brain.

3.3.1 Transformer Architecture

The transformer architecture has two components: an encoder and a decoder. In the base configuration, each stack contains six identical layers, though modern variants scale this to dozens or even hundreds.

The encoder processes the input sequence and transforms it into contextualized embeddings. The decoder takes those embeddings and iteratively generates the output sequence. An illustration of the overall architecture is shown in Figure 3.4 [20].

In this thesis, we are going to focus on the encoder part.

Encoder Before entering the pipeline, the sentences are tokenized and assigned embeddings. Each input token is mapped to a fixed-length vector composed of token embeddings, obtained via a tokenizer, and positional embeddings, which encode

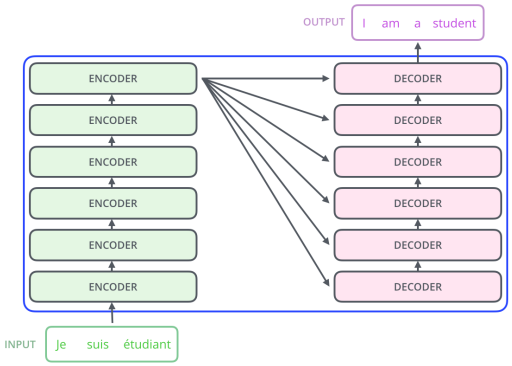

Figure 3.4: Example of translation task with the transformer architecture. [20]

word order. These vectors are summed and passed into the first encoder layer. All encoding layers are identical except for their weights.

As shown in Figure 3.5, each encoder layer consists of:

1. **Multi-head self-attention**, which allows the model to attend to multiple parts of the input simultaneously. This will be explained in more detail in the next paragraph.

2. **Feed-forward network**, applied independently to each token.

3. **Residual connections and layer normalization** after each sublayer, which help to stabilize training.

The embeddings are transformed in each step and then given to the next encoding layer until the last layer is reached. The output of the last encoder layer is the contextualized embeddings.

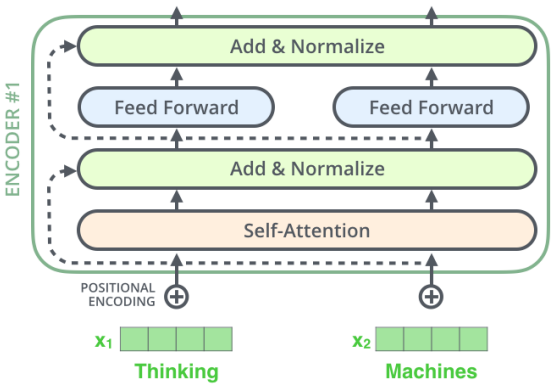

Figure 3.5: Illustration of an encoder. [20]

Self-attention Self-attention enables a transformer to understand how influential different words are to each other. Consider the sentence *The animal did not cross the street because it was too tired.* Humans instantly resolve that *it* refers to *animal*, not *street*. Self-attention calculates how important each word in the sentence is to the other words so that the transformer can use that information. In our example, *animal* would be quite important to *it* so it would obtain a high number, and *street* would get a lower number.

To learn such dependencies, the model assigns three vectors to each token: a query vector, a key vector, and a value vector. The idea is to use the query vector of one token to measure its compatibility with the key vectors of all other words. The value vector carries the information content of the word. To find out how much the token *street* should pay attention to the rest of the sentence, its query is dot-multiplied with the keys of all other tokens, producing a set of raw similarity scores:

$$\text{score}_j = \text{dot}\big(Q_{street},\, K_j\big).$$

These scores are fed through a soft-max layer so that they become positive and add up to one. The result is a list of attention weights α_j that tells the model how relevant each token j is to *street*.

Finally, the value vectors are combined using those weights:

$$\text{new\_rep}_{street} = \sum_j \alpha_j \, V_j.$$

This new vector reflects the combined influence of all other words, weighted by how relevant they are. So, more important words have hidden states with larger numbers, and less important words have hidden states with lower numbers. The same calculation is carried out in parallel for every token in the sentence, allowing the model to capture long-range relationships efficiently without recurrent processing.

Transformers typically employ multi-head attention, which is the practice of running the entire self-attention calculation multiple times in parallel, each time with its own learned projection matrices for queries, keys, and values. This gives the transformers the ability to focus on different parts of the sentence at the same time.

3.3.2 BERT

BERT is one of the most influential transformer-based models in NLP. BERT stands for Bidirectional Encoder Representation of Transformers and was introduced in the paper "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" in 2018 [7]. Like the name suggests, BERT can use context from both sides of the sentence, whereas most models like GPT use only the words before the token. This property makes BERT less suitable for open-ended sentence generation but highly effective for tasks that require a sentence-level understanding, such as classification and question answering.

BERT Architecture

BERT's architecture only uses the encoder stack of the transformer. In the *BERT_base* configuration, the model consists of:

- 12 identical encoder layers,
- 12 self-attention heads per layer,
- Embedding size of 768,
- Approximately 110 million parameters.

BERT Tokenization

When BERT tokenizes the sentences, it adds a CLS token at the beginning. This token is given an embedding and represents the whole sentence. It can be used for classification tasks. This concept is implemented in BERT through WordPiece tokenization [21].

BERT Training

BERT is trained in two steps: pre-training and fine-tuning. The goal of pre-training is to teach the model general language representations. Then, fine-tuning is used to adapt the model to specific downstream tasks.

Pre-training Pre-training uses large amounts of unlabeled text with two self-supervised learning tasks. It first creates static embeddings for the tokens, which can then be looked up for later use. It also initializes the weights for the model so that later sentences can be passed through the model to get contextualized embeddings.

BERT uses two strategies for pre-training.

1. **Masked Language Modeling (MLM)**: 15% of input tokens are randomly selected for prediction. Of these, 80% are replaced with `[MASK]`, 10% with a random token, and 10% remain unchanged. The model is trained to predict the original tokens. This allows BERT to learn the bidirectional context for every token.

   Original: `The cat sat on the mat and yawned.`
   Masked: `The cat sat on the [MASK] and yawned.`

2. **Next Sentence Prediction (NSP)**: The model receives two sentences and predicts whether the second sentence logically follows the first. This helps BERT learn relationships between sentences.

   IsNext:
   A: `I like being outside.`
   B: `The weather is wonderful.`

   NotNext:
   A: `I like being outside.`
   B: `Bacteria can be killed by antibiotics.`

Google trained BERT on Wikipedia and on BooksCorpus [22] for four days on 16 cloud TPUs. Now, these models can be used by anyone for fine-tuning on downstream tasks without needing to pre-train the models themselves. This concept, known as transfer learning, refers to taking a model that has been trained on a large, general-purpose dataset and adapting it to a specific task using a smaller, task-specific dataset [23].

At the end of pre-training, we can create contextualized embeddings for all sentences by passing them through the model. This works by first tokenizing the sentence, then looking up the pre-trained static embeddings, then passing the embeddings through the pre-trained model to obtain the contextualized embeddings. An example of the workflow is shown in Figure 3.6.

Fine-tuning During this stage, the pre-trained model is further trained on a smaller, labeled dataset relevant to the target task. The goal is to train BERT to solve a particular task, for example, the classification of a sentence. Fine-tuning adjusts the pre-trained weights slightly to optimize performance on the target task, enabling BERT to achieve strong results even on relatively small datasets, as the majority of the model's parameters have already been pre-trained to capture general

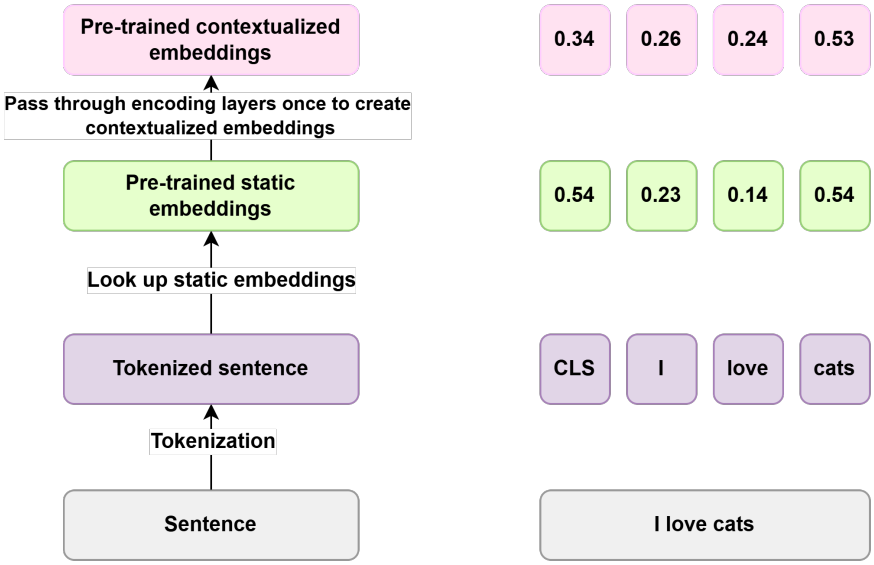

Figure 3.6: Visualization of process of pre-trained embedding extraction. An Illustration of the steps can be found on the right side, and example values on the left side.

linguistic knowledge. Fine-tuning usually requires only a few epochs and modest computational resources.

Since BERT does not include a decoder stack, the output of the final encoder layer consists of contextualized embeddings for each input token. To perform classification tasks, a task-specific classification head followed by a softmax function is added on top of the model. BERT passes the embedding of the CLS token to this layer, producing a probability distribution over the target classes. Both the classifier head and the entire BERT model are trained jointly.

Once BERT is fine-tuned, there are two options for using the model. The output of the classification head can be used directly. The other option is to extract the fine-tuned embeddings, similarly as with a pre-trained model.

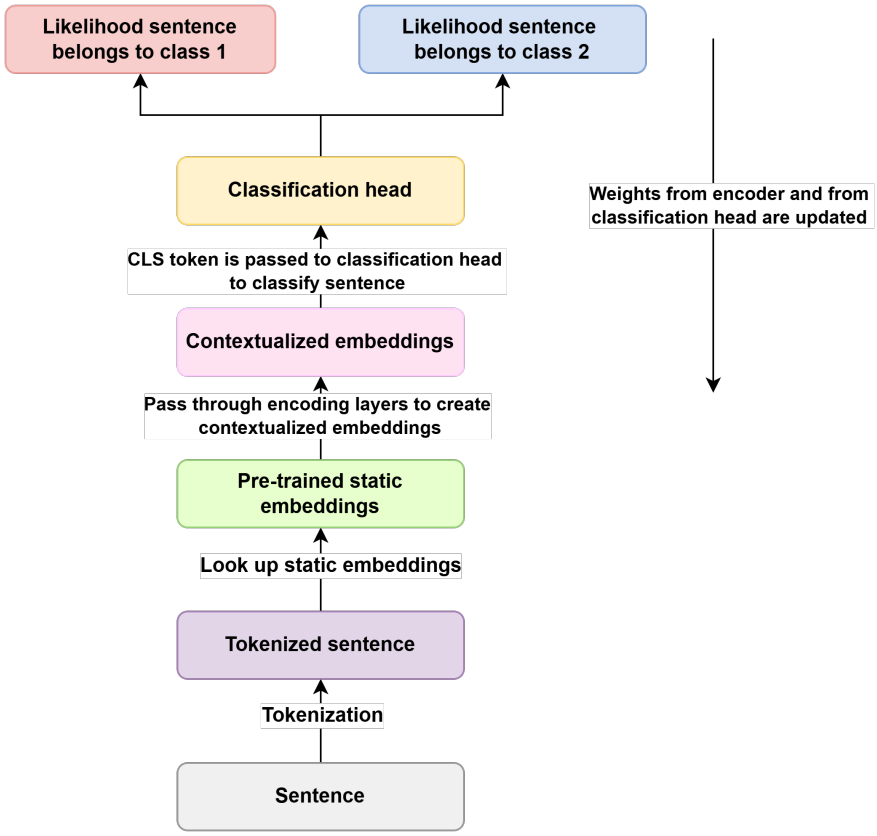

Figure 3.7: Visualization of BERT fine-tuning process with a binary classification head.

Chapter 4

Methods and Experimental Evaluation

In this chapter, the methods and findings are described. Section 4.1 offers an overview of the experiments. The datasets are then described in detail in Section 4.2. Next, Section 4.3 details the fine-tuning process, and Section 4.4 describes embedding extraction and preprocessing. Section 4.5 covers how the training was done using the classifiers SVM and XGBoost. Finally, the results are presented in Section 4.6.

4.1 Overview

The primary objective of this thesis is to evaluate the performance of a fine-tuned BERT model compared to traditional machine learning classifiers using contextualized embeddings as features.

Three methods are evaluated:

1. **Using pre-trained embeddings with classifiers:** Contextualized embeddings are extracted from the pre-trained model without fine-tuning. These embeddings serve as input features for the classifiers SVM and XGBoost. This approach aims to assess whether comparable accuracy to the fine-tuned BERT model could be achieved using these embeddings, potentially reducing training time and computational cost.

2. **Using fine-tuned embeddings with classifiers:** After fine-tuning the BERT model on the dataset, contextualized embeddings are extracted and used with the same classifiers. Although this method requires fine-tuning, it investigates whether such embeddings can outperform the fine-tuned BERT model itself when paired with traditional classifiers.

For approaches 2 and 3, two types of embeddings are tested:

- Sentence-wise embeddings
- Token-wise embeddings reduced via PCA to 50 and 400 dimensions

In the end, the embeddings are reduced to 2 dimensions using PCA and projected on a 2-dimensional space. This was done to visualize how easily separable the wrong and correct sentence examples are. These reduced embeddings are only used for visualization purposes and not for error prediction purposes.

4.2 Datasets

The data consists of three synthetically generated datasets. Each dataset contains 100,000 sentences, which are labeled as either correct (label 0) or incorrect (label 1). There are 50'000 correct and 50'000 incorrect sentences included in each dataset. Although the original datasets also include additional information such as corrected versions of the sentences, only the sentences and their corresponding labels are retained and used, since the rest is not necessary for supervised machine learning. The three datasets each target a specific type of grammatical error: capitalization, conjugation, and case errors. Each of their incorrect sentences has an error of that type. Here are a few examples of sentences in the datasets:

- **Capitalization errors:** Involve incorrect usage of uppercase and lowercase letters, particularly violating German capitalization rules.
  Example: "Zum aktuellen Zeitpunkt **Wohnen** 7551 Einwohner in Grossmehring."
  Correction: "Zum aktuellen Zeitpunkt **wohnen** 7551 Einwohner in Großmehring."

- **Conjugation errors:** Have incorrect conjugation or tense usage.
  Example: "Lieferung war im Jahre 1991 der niederdeutsch-hochdeutsche Hauptteil des Werkes im Manuskript **vollenden**."
  Correction: "Lieferung war im Jahre 1991 der niederdeutsch-hochdeutsche Hauptteil des Werkes im Manuskript **vollendet**."

- **Case errors:** Refer to incorrect grammatical cases in sentence structure.
  Example: "**Ihr** ersten Auftritt in dieser Besetzung hielt die Gruppe am 31. Januar 1975 im The Nowhere Club in Bicester ab."
  Correction: "**Ihren** ersten Auftritt in dieser Besetzung hielt die Gruppe am 31. Januar 1975 im The Nowhere Club in Bicester ab."

All sentences longer than 1'000 characters were removed from the datasets. This affected only six entries from the capitalization dataset, which were excluded due to being irregular entries (e.g. coordinate data). After this filtering step, the sentence lengths range from 4 to 268 words, with an average sentence length of 24.42 words. The datasets are loaded as pandas dataframes.

Each dataset is trained and evaluated separately, allowing the models to focus on learning a specific type of error. Apart from the removal of the six sentences, there are no preprocessing steps involved for the data.

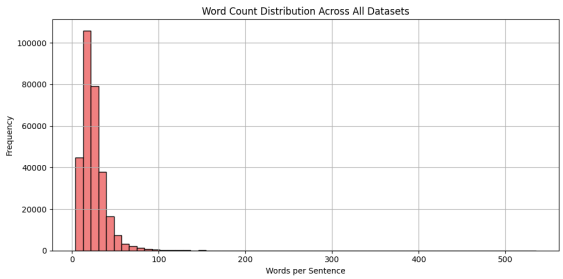

Figure 4.1: Word count distribution across all datasets.

4.3 Fine-tuning BERT

Tokenization All sentences are tokenized using BertTokenizerFast from Hugging Face, which implements the WordPiece tokenization algorithm [24]. This tokenizer has a vocabulary size of 119'547 tokens. The CLS token is prepended to each sentence, and the sentences are tokenized and padded or truncated to a uniform length of 128 tokens. The input IDs are passed to the model and converted into embeddings of length 768, as designed for the BERT architecture.

Table 4.1 shows a tokenized example sentence. Each token is assigned an ID, and the attention mask indicates whether the token is meaningful (1) or padding (0).

Model architecture The pre-trained bert-base-multilingual-cased model was selected as the foundation for the experiments [25]. This variant supports 104 languages, including German, and preserves case distinctions, which is essential for detecting capitalization errors. The model follows the standard BERT base architecture with 12 transformer layers, 12 self-attention heads, and 768-dimensional hidden embeddings. It was originally trained on Wikipedia across all supported languages.

To adapt BERT for the binary classification task, the BertForSequenceClassification class from the Hugging Face library is used with the number of labels set to

Token	Attention Mask	Input ID
[CLS]	1	101
Mit	1	12699
der	1	10118
Unter	1	20409
zei	1	53228
chn	1	61497
ung	1	10716
des	1	10139
Vertrag	1	25239
es	1	10171
...	...	...
[PAD]	0	0
[PAD]	0	0

Table 4.1: Tokenized input with attention masks and corresponding input IDs.

two [26]. This adds a binary classification head that predicts whether a sentence is correct or contains a grammatical error.

The model is fine-tuned on the training set, with validation using the validation set at the end of each epoch to see how well the model is doing, and a final evaluation performed on the test set to assess generalization. 50'000 sentences are in the training set, 40'000 in the test set, and 10'000 in the evaluation set.

Training configuration Training is performed using the Hugging Face Trainer API on an NVIDIA A100 GPU in Google Colab, significantly reducing training time compared to CPU-based computation.

The following hyperparameters and settings are used during training:

- **Training duration:** 16 epochs.
- **Batch size:** 32. This means that during each training or evaluation step, the model processes 32 input examples simultaneously.
- **Optimizer:** AdamW. An optimizer adjusts the model's weights during training to minimize the loss function.
- **Learning rate scheduler:** Cosine annealing with 200 warm-up steps.
- **Weight decay:** 0.04, used to regularize the model and prevent overfitting.
- **Mixed-precision training:** Enabled with fp16=True to accelerate computation and reduce memory consumption.

During training, three standard classification metrics are monitored:

- **Accuracy**: Proportion of correctly classified samples
- **Precision**: Proportion of correctly predicted positive instances out of all predicted positives
- **Recall**: Proportion of correctly predicted positive instances out of all actual positives

The metrics are computed on the validation set at the end of each epoch using the Hugging Face evaluate library. The model from the epoch with the highest validation accuracy is saved and later evaluated on the test set to assess generalization.

4.4 Extraction and Preprocessing of Embeddings

To extract the embeddings, the same pre-trained model bert-base-multilingual-cased was selected. First, the embeddings are extracted from the pre-trained BERT model. After fine-tuning the model on the dataset, embeddings are extracted from the fine-tuned version.

Extraction of pre-trained embeddings First, the pre-trained model is loaded with its full architecture, which includes the embedding layers for tokens and positions, twelve transformer layers consisting of multi-head self-attention, feed-forward networks, layer normalization, and residual connections, as well as the pre-trained weights learned during MLM and NSP.

To obtain contextualized embeddings, the sentences are passed through all 12 transformer layers of the model. This is necessary to generate meaningful token-level and sentence-level representations that reflect sentence context, including meaningful CLS token embeddings, since without a forward pass, all default CLS tokens are identical and uninformative.

Two types of embeddings are extracted: sentence-wise embeddings, which correspond to the final-layer embedding of the CLS token for each input, and token-wise embeddings, which are the contextualized embeddings of all tokens from the final layer.

The embeddings are extracted separately for each split of the three datasets (training, validation, and test) and stored for use in downstream evaluation and analysis.

Extraction of fine-tuned embeddings After fine-tuning the model on the datasets, the embedding extraction process is repeated. The sentences are passed through the full fine-tuned model to produce contextualized embeddings that incorporate task-specific knowledge. The same two forms of embeddings are extracted.

Preprocessing of embeddings The sentence-wise embeddings are used directly as input to the classifiers without any additional preprocessing. Each of these embeddings has a dimension of 768, which can be efficiently handled by both SVM and XGBoost models. In contrast, token-wise embeddings have a shape of $(128, 768)$, one 768-dimensional embedding per token, for up to 128 tokens per sentence. Since standard classifiers require inputs of shape $(n, 1)$ (that is, flat feature vectors), token-wise embeddings are first flattened to a single vector of length $128 \times 768 = 98,304$. This high dimensionality is impractical for use with models like SVM or XGBoost, which struggle to process such large input vectors due to computational and memory constraints.

To address this issue, dimensionality reduction is applied using PCA. Specifically, flattened token-wise embeddings are reduced in dimensionality using IncrementalPCA from the RAPIDS cuML library for GPU acceleration [27]. IncrementalPCA is used instead of standard PCA because the dataset is too large to fit into memory all at once. IncrementalPCA works batch by batch, which makes it memory-efficient. A batch size of 500 is used to avoid exceeding the available system memory during the reduction process.

The flattened embeddings are reduced to two target sizes: 50 and 400 dimensions, resulting in new input vectors of shape $(50,)$ and $(400,)$, respectively. PCA is applied separately to all token-wise embeddings, and the reduced embeddings are saved. The variance retained from each dataset is saved for later analysis.

4.5 Training with Classifiers

The contextualized embeddings are used as features to train the classifiers SVM and XGBoost. Again, 50'000 sentences are used for training, 40'000 for testing, and 10'000 for evaluation.

4.5.1 SVM

To evaluate the classification task using SVM, the library cuSVM is used [28]. cuSVM is a CUDA-based implementation of SVM designed to be able to use GPU

to accelerate training and prediction. It significantly outperforms CPU-based implementations of SVM in terms of speed.

Preprocessing All input features are first normalized using standardScaler by Scikit-learn [29]. After scaling, the NumPy arrays are converted to cupy.ndarray so that both the data and the model reside on the GPU.

Hyperparameter search To assess the performance of the model, we focus on the rbf kernel and systematically explore different values of C and gamma. The search covers the following grid:

```
param_grid = {
    'C': [0.0000001, 0.000001, ..., 1000, 10000],
    'gamma': [0.0000001, 0.000001, ..., 0.01, 0.1, 1],
    'kernel': ['rbf']
}
```

Each combination of parameters is evaluated on the evaluation dataset, and the accuracy is recorded. The best model is then retrained on the training dataset, and finally the accuracy of the testing dataset is measured.

4.5.2 XGBoost

To evaluate the effectiveness of gradient-boosted decision trees on the classification task, we use the GPU-accelerated XGBClassifier from the XGBoost library [16]. The computations are once again performed on the GPU using cuDF. Again, all the features of the inputs are first normalized using standardScaler by Scikit-learn.

Hyperparameter search. To find the best-performing hyperparameters, one grid-search is performed over all the datasets:

These hyperparameters are tested:

```
param_grid = {
    'max_depth': [2, 3, 5],
    'learning_rate': [0.0001, 0.001, 0.01, 0.1],
    'n_estimators': [100, 200, 300, 400],
    'subsample': [0.7, 0.8],
    'colsample_bytree': [0.8, 1.0]
}
```

Each configuration is evaluated with the evaluation dataset and the model with the highest accuracy is retrained on the full training set.

4.6 Results

4.6.1 Fine-Tuned BERT

The fine-tuned BERT model achieves strong and consistent performance across all datasets, as anticipated given the model's capacity to learn complex grammatical patterns. The model surpasses 91 % on case and exceeds 94 % on both capitalization and conjugation. The slightly reduced accuracy on the case dataset suggests that these errors have fewer distinguishing patterns for BERT to pick up on compared to the other datasets. Table 4.2 summarizes the calculated accuracy.

Dataset	Accuracy BERT fine-tuned (%)
Case	91.5
Capitalization	94.5
Conjugation	94.0

Table 4.2: Accuracy (%) of the fine-tuned BERT model on the three datasets.

4.6.2 Classifiers with Pre-Trained Embeddings

Sentence-Wise Pre-Trained Embeddings

First, the sentence-wise pre-trained embeddings are evaluated using both the SVM and XGBoost classifiers. As shown in Table 4.3, SVM achieves accuracies of 53.3% for case, 55.0% for capitalization, and 59.7% for conjugation. XGBoost yielded marginally lower performance, with 51.4%, 52.2%, and 54.0% on the respective datasets. In all cases, the accuracy remains well below 60% and is quite close to chance. Similarly to BERT's fine-tuning results, the case dataset underperforms slightly compared to the other datasets.

These results fall significantly short of the fine-tuned BERT performance. The low scores for both classifiers indicate that sentence-wise pre-trained embeddings lack enough information for grammatical error detection. This suggests that fine-tuning is necessary to achieve strong results.

To better understand the impact of hyperparameter choices in SVM, heatmaps are generated to visualize accuracy across a grid of C and gamma values (Figure 4.2). The heatmap below shows the results for the conjugation dataset. The best-performing region corresponds to approximately $C \approx 100$ and $\gamma \approx 10^{-5}$, and the performance appears to be going down in surrounding regions. Despite tuning, the maximum accuracy achieved is still well below the BERT baseline.

Dataset	SVM	XGBoost	BERT
Case	53.3	51.4	91.5
Capitalization	55.0	52.2	94.5
Conjugation	59.7	54.0	94.0

Table 4.3: Accuracy (%) of the SVM and XGBoost models with sentence-wise pre-trained embeddings, compared with fine-tuned BERT.

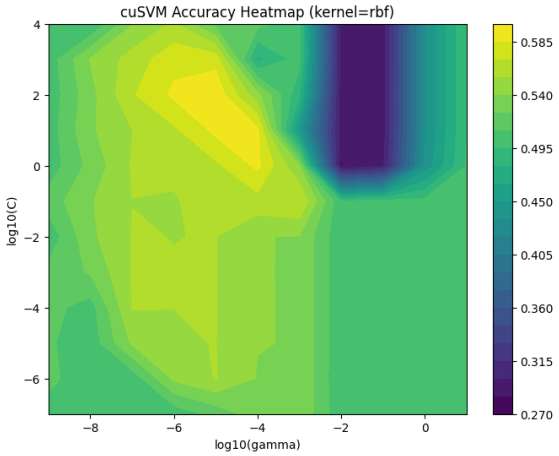

Figure 4.2: Heatmap depicting accuracy for sentence-wise embeddings with dataset conjugation using SVM.

Token-Wise Pre-Trained Embeddings

The pre-trained token-wise embeddings are subsequently reduced in dimensionality using PCA. As shown in Table 4.4, the 400-dimensional representations retained roughly twice as much variance as their 50-dimensional counterparts across all datasets, suggesting that the higher-dimensional reduction preserves substantially more of the original embedding information.

Dataset	PCA-50	PCA-400
Case	24	46
Capitalization	22	43
Conjugation	24	46

Table 4.4: Variance retained (%) by PCA for pre-trained embeddings.

The reduced embeddings are then evaluated using both SVM and XGBoost (Table 4.5). The 50-dimensional embeddings yielded near-chance accuracy, while the 400-dimensional embeddings showed only marginal improvement, remaining far below the performance of fine-tuned BERT. The better accuracy using the embeddings reduced to 400 dimensions could be explained by the higher variance retained by

the model, permitting the models to pick up on more informative patterns. Again, XGBoost has a slightly lower accuracy compared to SVM, XGBoost did not seem to be able to pick up on any patterns with the 50-dimensional embeddings, with the accuracy staying at about 50 %.

This indicates that token-wise embeddings reduced with PCA perform worse than BERT's own CLS token. Overall, these results suggest that PCA-reduced token-wise embeddings are less effective for this task than BERT's CLS tokens, and that SVM has a small but consistent advantage over XGBoost when working with pre-trained embeddings.

Dataset	**SVM 50**	**SVM 400**	**XGBoost 50**	**XGBoost 400**	**BERT**
Case	51.5	52.0	50.1	51.0	91.5
Capitalization	51.3	51.9	50.0	50.6	94.5
Conjugation	51.6	52.0	50.1	50.8	94.0

Table 4.5: Accuracy (%) of SVM and XGBoost models with token-wise pre-trained embeddings (50 and 400 dimensions), compared with fine-tuned BERT.

To further investigate, for the SVM model heatmaps are generated for both dimensionalities to visualize accuracy across the hyperparameter grid. As with sentence-wise embeddings, no parameter region produced performance near that of BERT. The results can be seen in Figures 4.3 and 4.4.

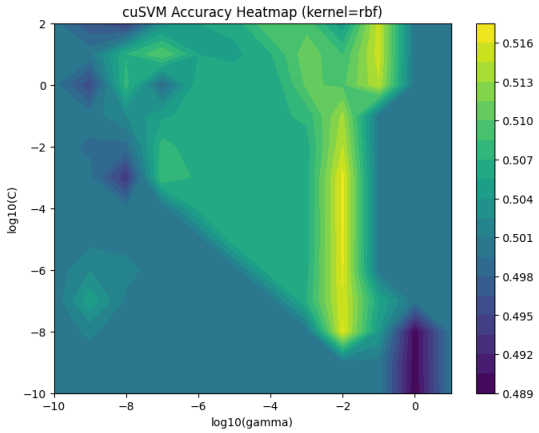

Figure 4.3: Heatmap depicting accuracy for token-wise embeddings reduced to 50 dimensions with dataset conjugation using SVM.

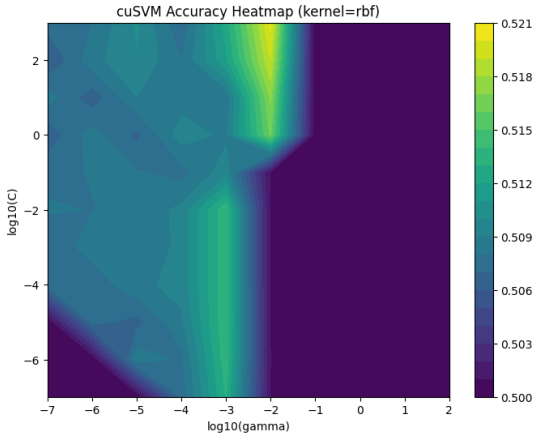

Figure 4.4: Heatmap depicting accuracy for token-wise embeddings reduced to 400 dimensions with dataset conjugation using SVM.

4.6.3 Classifiers with Fine-Tuned Embeddings

Sentence-Wise Fine-Tuned Embeddings

Once the fine-tuned embeddings are extracted, sentence-wise embeddings are evaluated using the classifiers. As shown in Table 4.6, SVM performance closely matches or slightly surpasses that of fine-tuned BERT across all datasets, achieving 91.6 % accuracy on case, 94.6 % on capitalization, and 94.0 % on conjugation. XGBoost exactly matches the results of the fine-tuned BERT model. This indicates that the fine-tuned embeddings encode sufficient task-relevant information for the classifiers to perform at a high level. However, their inability to exceed BERT's built-in classification head suggests that, while effective, such classifiers cannot surpass the performance of the original fine-tuned model. Overall, the results confirm that fine-tuning substantially increases the utility of sentence-wise embeddings, enabling even traditional classifiers to achieve accuracy comparable to BERT's own classification head.

Dataset	**SVM**	**XGBoost**	**BERT fine-tuned**
Case	91.6	91.5	91.5
Capitalization	94.6	94.5	94.5
Conjugation	94.0	94.0	94.0

Table 4.6: Accuracy (%) of SVM and XGBoost models with fine-tuned sentence-wise embeddings, compared with fine-tuned BERT.

A heatmap is also generated for the conjugation dataset using SVM (Figure 4.5). Unlike the sparse high-performance "islands" observed in earlier heatmaps with pre-trained embeddings, many hyperparameter configurations achieve consistently high accuracy here.

Token-Wise Fine-Tuned Embeddings

The token-wise fine-tuned embeddings, reduced to 50 and 400 dimensions using PCA, retain substantially more variance than their pre-trained counterparts, reaching 100 % with some dataset (Figure 4.7). This could mean that the fine-tuning process amplifies task-relevant patterns in the data, enabling PCA to preserve a larger proportion of the original information.

SVM and XGBoost both achieve strong results with the fine-tuned token-wise embeddings, as shown in Table 4.8, with performance generally approaching that of fine-tuned BERT. The main exception is the capitalization dataset, where XGBoost's accuracy dropped sharply to 33% and 42% for the 50- and 400-dimensional

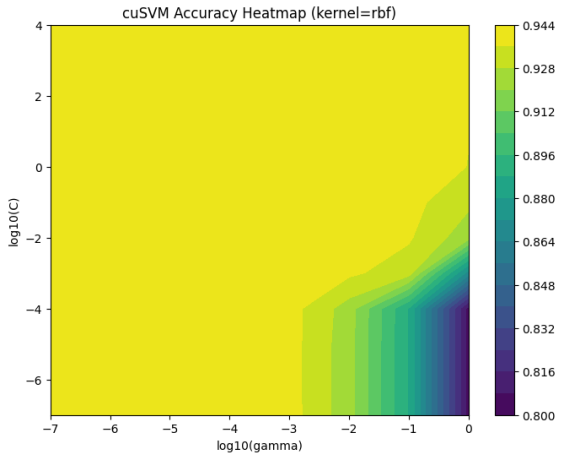

Figure 4.5: Heatmap depicting accuracy for sentence-wise fine-tuned embeddings with dataset conjugation

Dataset	**PCA-50**		**PCA-400**	
	Pre-trained	**Fine-tuned**	**Pre-trained**	**Fine-tuned**
Case	24	84	46	90
Capitalization	22	99	43	100
Conjugation	24	100	46	100

Table 4.7: Variance retained (%) by PCA for pre-trained vs. fine-tuned embeddings.

embeddings, respectively. This suggests that, for this particular task, the relevant information in the token-wise embeddings may not be effectively captured by XGBoost, though the underlying reason remains unclear, since SVM was able to do so. Aside from this anomaly, the results indicate that reduced token-wise embeddings retain sufficient task-relevant information for both classifiers to perform at a high level.

Dataset	**SVM 50**	**SVM 400**	**XGBoost 50**	**XGBoost 400**	**BERT**
Case	91.4	91.4	91.0	91.3	91.5
Capitalization	90.0	94.0	33.0	42.0	94.5
Conjugation	94.0	94.0	94.0	94.0	94.0

Table 4.8: Accuracy (%) of SVM and XGBoost models with fine-tuned token-wise embeddings (50 and 400 dimensions), compared with fine-tuned BERT.

Figures 4.6 and 4.7 present the heatmaps for 50 and 400 dimensions respectively. While both show a large range of hyperparameter configurations achieving high accuracy, the sentence-wise embeddings display slightly greater robustness across the hyperparameter space.

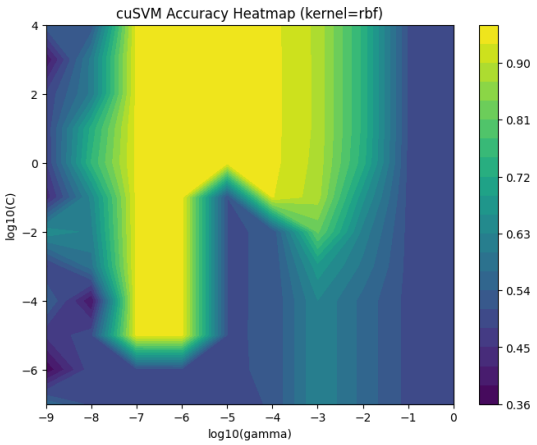

Figure 4.6: Heatmap depicting accuracy for token-wise fine-tuned embeddings (50 dimensions) on dataset Conjugation.

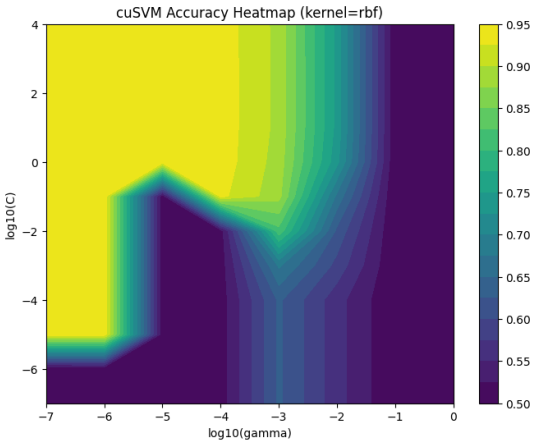

Figure 4.7: Heatmap depicting accuracy for token-wise fine-tuned embeddings (400 dimensions) on dataset Conjugation.

4.6.4 Visualizations

For qualitative analysis, the embeddings of the conjugation dataset are reduced to two dimensions using PCA. The embeddings are first reduced to two dimensions with PCA and then projected onto a two-dimensional space.

Figure 4.8 shows the visualization before fine-tuning. The two classes are not clearly separable, which explains why the classifiers might have struggled to classify these embeddings.

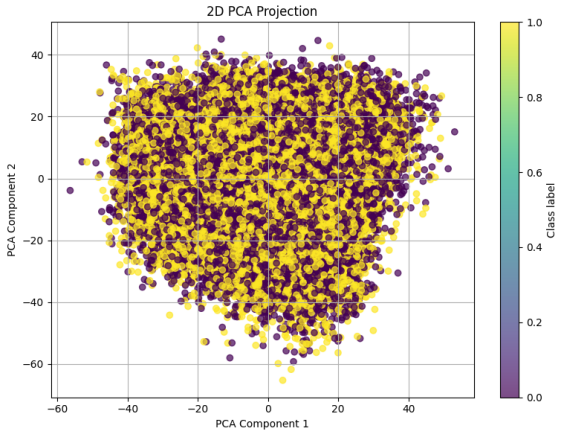

Figure 4.8: Pre-trained embeddings of the conjugation dataset reduced to two dimensions, with yellow being the incorrect sentences and purple being the correct sentences.

The result after epoch 22000, once the embeddings are fine-tuned can be seen in Figure 4.9. Here, separating the embeddings seems much easier since the classes are clustered.

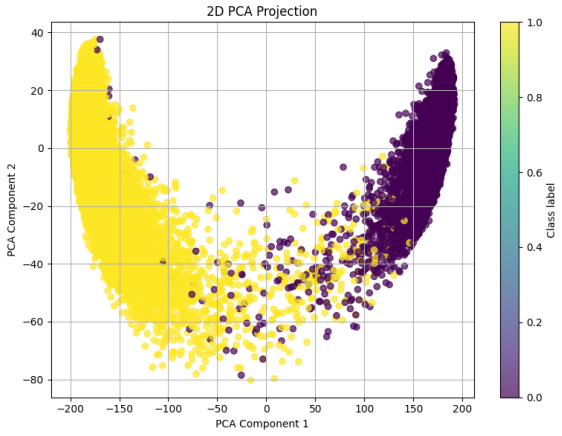

Figure 4.9: Fine-tuned embeddings at epoch 22000 of the conjugation dataset reduced to two dimensions, with yellow being the incorrect sentences and purple being the correct sentences.

The contrast between Figures 4.8 and 4.9 highlights the impact of fine-tuning. While the pre-trained embeddings show considerable overlap between classes, fine-tuning restructures the embedding space so that class boundaries become more distinct. This transformation likely underlies the large performance gains observed in the classification experiments.

Chapter 5

Conclusions and Future Work

This thesis has the goal of finding new, more efficient ways of detecting errors in sentences. There are three types of grammatical errors tested: conjugation, capitalization, and case errors. Three synthetically made datasets are available for this task. This thesis explores whether it is possible to use embeddings generated with a pre-trained BERT model with the classifiers SVM and XGBoost to distinguish between correct and incorrect sentences. These models are compared with a fine-tuned BERT model using its built-in classification head. Since fine-tuning BERT uses a lot of time and computational power, achieving similar result with a pre-trained model could save a significant amount of resources. Extracting fine-tuned embeddings to use for the classifiers was also tested, with the goal of outperforming BERT's built-in classification head.

The results show that BERT's fine-tuned model consistently achieves accuracies of over 90 %. The pre-trained embeddings, when combined with traditional classifiers, underperform substantially compared to the fine-tuned model. Table 5.1 reports the average accuracy reached with all sentence-wise and token-wise embeddings, compared to BERT's fine-tuned model. The accuracy gap between the pre-trained embeddings and BERT's model is significant, reaching over 35 %.

Model	**Sentence-wise**	**Token-wise**	**BERT fine-tuned**
Average accuracy (%)	54.2	51.0	93.3

Table 5.1: Average accuracy (%) of sentence-wise and token-wise pre-trained embeddings compared with fine-tuned BERT.

These findings underscore the critical role of fine-tuning in adapting large language models for error detection. Despite the time and computational power saved by bypassing the fine-tuning step, the resulting performance is not sufficient. Fine-tuning proves to be necessary to enable embeddings to encompass task-relevant

information, with both sentence-wise and token-wise representations. Without it, embeddings fail to capture the necessary task-specific context, limiting the effectiveness of downstream classifiers.

The results further highlight that fine-tuned embeddings perform substantially better than the pre-trained embeddings, having accuracies close to matching or slightly exceeding BERT's built-in classification head except for XGBoost with capitalization (Table 5.2). The outliers of the capitalization dataset with XGBoost are removed from this table to better report standard performance.

Model	**Sentence-wise**	**Token-wise**	**BERT fine-tuned**
Average accuracy (%)	93.4	92.5	93.3

Table 5.2: Average accuracy (%) of sentence-wise and token-wise fine-tuned embeddings (excluding XGBoost capitalization anomaly) compared with fine-tuned BERT.

This demonstrated that given task-specific embeddings, traditional classifiers are able to detect errors in sentences effectively. The markedly lower performance observed with pre-trained embeddings therefore appears to be attributed to the quality of the embeddings themselves rather than to limitations of the classifiers. However, as the classifiers do not significantly outperform BERT's built-in classification head, directly using the latter remains the more efficient approach.

Across all categories, including pre-trained and fine-tuned embeddings, sentence-wise embeddings consistently outperform token-wise embeddings by a few percent, with token-wise embeddings reduced to 400 dimensions achieving higher accuracy than those reduced to 50 dimensions (Table 5.3). This difference may be attributed to the longer vector length of sentence-wise embeddings (768 dimensions) compared to the 50- and 400-dimensional token-wise versions, allowing them to encode more information. Furthermore, BERT's CLS token is specifically designed to capture sentence-level meaning, which appears to yield more task-relevant information. The results are also likely influenced by the PCA process, which substantially reduced the dimensionality of the token-wise embeddings. Employing an alternative dimensionality reduction algorithm might alter these outcomes. If the full-length token-wise embeddings could be used without dimensionality reduction, their performance might be significantly higher.

With all types of embeddings, SVM appears to be more stable and achieve higher performance compared to XGBoost. Table 5.4 compares the average accuracy of both classifiers in pre-trained and fine-tuned settings, excluding the fine-tuned capitalization dataset.

Embedding type	Sentence-wise	Token-wise 50	Token-wise 400
Average accuracy (%)	73.8	69.5	70.3

Table 5.3: Average accuracy (%) of sentence-wise and token-wise embeddings (50 and 400 dimensions) across pre-trained and fine-tuned settings, excluding outlier XGBoost capitalization results.

Classifier	Pre-trained avg. (%)	Fine-tuned avg. (%)
SVM	53.1	93
XGBoost	51.1	92.9

Table 5.4: Average accuracy (%) of SVM and XGBoost in pre-trained and fine-tuned settings, excluding the XGBoost capitalization anomaly.

The results indicate that SVM is more effective at classifying embeddings for grammatical error detection. This advantage may stem from SVM's ability to perform well in high-dimensional spaces.

Future work Instead of using BERT_base_multilingual for the extraction of embedding, future work could explore German-only models, which may produce embeddings that are more tailored and thus more informative for this task. In general, other pre-trained embedding sources, such as those derived from GPT models, could be investigated, as their different generation mechanisms may yield better task-specific representations. Likewise, only SVM and XGBoost classifiers were tested here; evaluating additional classifiers could reveal alternative approaches with superior performance. Another interesting avenue for exploration would be to include both the correct and incorrect versions of the same sentence in the datasets, to assess whether such paired examples improve the models' ability to learn error patterns.

Bibliography

[1] Yuchen Jiang, Xiang Li, Hao Luo, Shen Yin, and Okyay Kaynak. Quo vadis artificial intelligence? *Discover Artificial Intelligence*, 2, 2022.

[2] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Artificial Intelligence Review*.

[3] Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: state of the art, current trends and challenges. *Multimedia Tools and Applications*, 82(3):3713–3744, 2023.

[4] Christopher Bryant, Zheng Yuan, Muhammad Reza Qorib, Hannan Cao, Hwee Tou Ng, and Ted Briscoe. Grammatical error correction: A survey of the state of the art. *arXiv*, 2022. arXiv:2211.05166.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017. [cs.CL].

[6] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*, 2019. [cs.CL].

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[9] Tingting Jiang. Supervised machine learning: A brief primer. *Clinical and Translational Medicine*, 10(5):e132, 2020.

[10] Linus Ericsson, Henry Gouk, Chen Change Loy, and Timothy M. Hospedales. Self-supervised representation learning: Introduction, advances and challenges. *Entropy*, 24(4):551, 2022.

[11] Ajitesh Kumar. Support vector machine (svm) python example. Blog post on VitalFlux (Analytics Yogi), March 2023. Accessed: 2025-08-10.

[12] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[13] Abhishek Jain. Svm kernels and its type. Blog post on Medium, September 2024. Accessed: 2025-08-10.

[14] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman & Hall/CRC, 1984.

[15] Erblin Halabaku and Eliot Bytyči. Overfitting in machine learning: A comparative analysis of decision trees and random forests. *Intelligent Automation & Soft Computing*, 39(6):987–1006, 2024.

[16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2016.

[17] IBM. What is principal component analysis (pca)? IBM Think blog, 2023. Accessed: 2025-08-10.

[18] Jonathon Shlens. A tutorial on principal component analysis. `https://arxiv.org/abs/1404.1100`, 2014. arXiv:1404.1100.

[19] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[20] Jay Alammar. The illustrated transformer. Blog post, 2018.

[21] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. In *arXiv preprint arXiv:1609.08144*, 2016.

[22] Yukun Zhu, Ryan Kiros, et al. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2015.

[23] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[24] Hugging Face. Bert tokenizer. https://huggingface.co/transformers/v3.2.0/model_doc/bert.html, 2020. Accessed: 2025-08-09.

[25] Hugging Face. Bert base multilingual cased. https://huggingface.co/bert-base-multilingual-cased, 2019. Accessed: 2025-08-09.

[26] Hugging Face. Bert for sequence classification. https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertForSequenceClassification, 2020. Accessed: 2025-08-09.

[27] RAPIDS AI. cuml: Incremental principal component analysis (incrementalpca). https://docs.rapids.ai/api/cuml/stable/api.html#incrementalpca, 2025. Accessed: 2025-08-09.

[28] Austin Carpenter. cusvm: A cuda implementation of support vector classification and regression. 2009. Software package for high-speed SVM training and prediction using GPUs, achieving up to 73× speedups over CPU implementations.

[29] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. In *Journal of Machine Learning Research*, volume 12, pages 2825–2830, 2011.

Erklärung

gemäss Art. 30 RSL Phil.-nat.18

Name/Vorname: Semelet Cécile

Matrikelnummer: 20-320-115

Studiengang: Computer Science

Bachelor ☑ Master ☐ Dissertation ☐

Titel der Arbeit: Fine-tuning vs. Embedding Classification
A Study on BERT for Error Detection

LeiterIn der Arbeit: PD Dr. Kaspar Riesen
Corina Masanti

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 10.08.2025

Ort/Datum

C. Semelet

Unterschrift